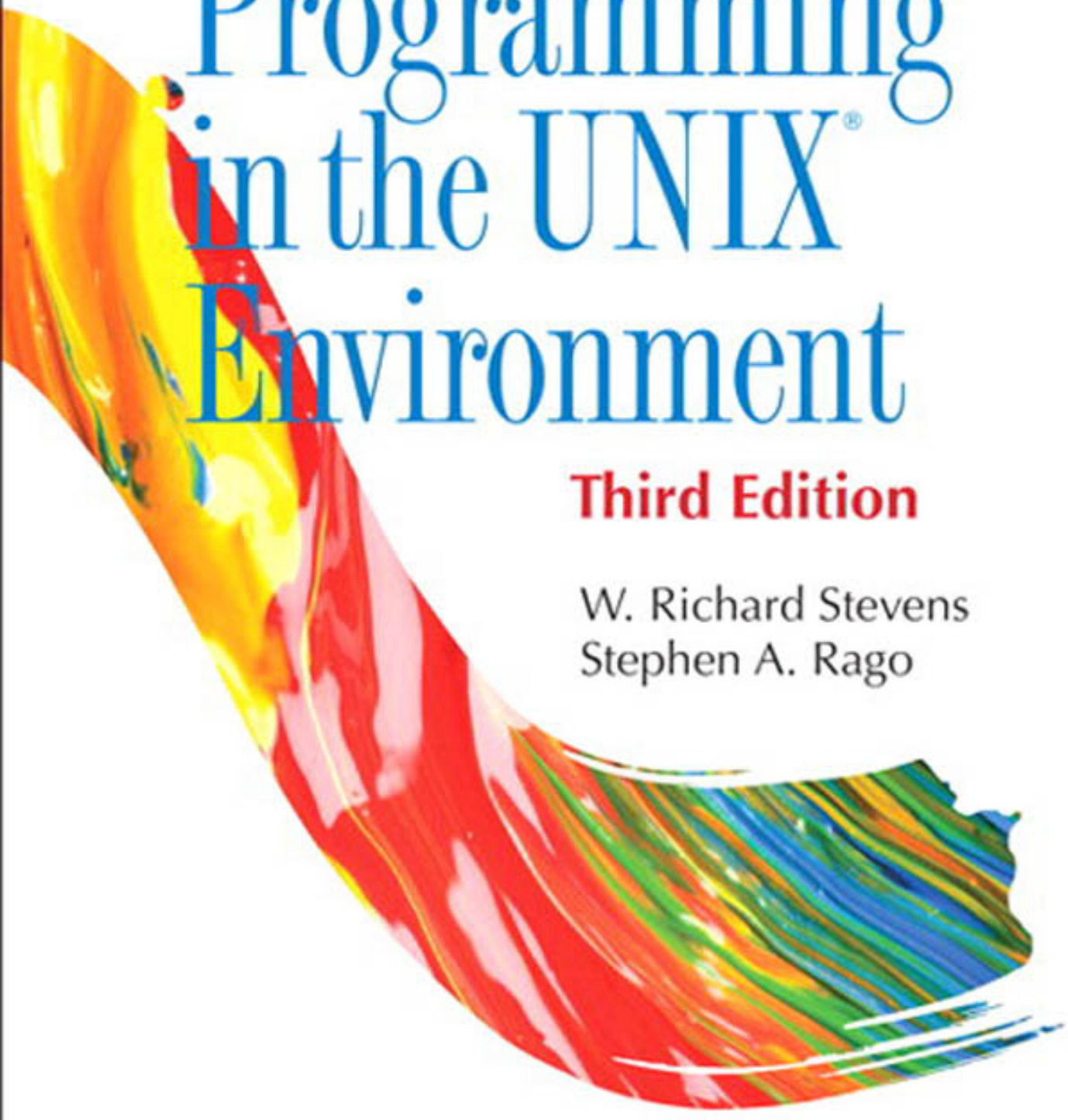
Advanced Programming in the UNIX® Environment

Third Edition

W. Richard Stevens
Stephen A. Rago



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The Addison-Wesley Professional Computing Series



Visit informit.com/series/professionalcomputing
for a complete list of available publications.

The Addison-Wesley Professional Computing Series was created in 1990 to provide serious programmers and networking professionals with well-written and practical reference books. There are few places to turn for accurate and authoritative books on current and cutting-edge technology. We hope that our books will help you understand the state of the art in programming languages, operating systems, and networks.

Consulting Editor Brian W. Kernighan

Make sure to connect with us!
informit.com/socialconnect

Addison Wesley | informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE | Safari Books Online

Praise for *Advanced Programming in the UNIX® Environment, Second Edition*

"Stephen Rago's update is a long overdue benefit to the community of professionals using the versatile family of UNIX and UNIX-like operating environments. It removes obsolescence and includes newer developments. It also thoroughly updates the context of all topics, examples, and applications to recent releases of popular implementations of UNIX and UNIX-like environments. And yet, it does all this while retaining the style and taste of the original classic."

—Mukesh Kacker, cofounder and former CTO of Pronto Networks, Inc.

"One of the essential classics of UNIX programming."

—Eric S. Raymond, author of *The Art of UNIX Programming*

"This is the definitive reference book for any serious or professional UNIX systems programmer. Rago has updated and extended the classic Stevens text while keeping true to the original. The APIs are illuminated by clear examples of their use. He also mentions many of the pitfalls to look out for when programming across different UNIX system implementations and points out how to avoid these pitfalls using relevant standards such as POSIX 1003.1, 2004 edition, and the Single UNIX Specification, Version 3."

—Andrew Josey, Director, Certification, The Open Group, and
Chair of the POSIX 1003.1 Working Group

"*Advanced Programming in the UNIX® Environment, Second Edition,* is an essential reference for anyone writing programs for a UNIX system. It's the first book I turn to when I want to understand or re-learn any of the various system interfaces. Stephen Rago has successfully revised this book to incorporate newer operating systems such as GNU/Linux and Apple's OS X while keeping true to the first edition in terms of both readability and usefulness. It will always have a place right next to my computer."

—Dr. Benjamin Kuperman, Swarthmore College

Praise for the First Edition

"*Advanced Programming in the UNIX® Environment* is a must-have for any serious C programmer who works under UNIX. Its depth, thoroughness, and clarity of explanation are unmatched."

—*UniForum Monthly*

"Numerous readers recommended *Advanced Programming in the UNIX® Environment* by W. Richard Stevens (Addison-Wesley), and I'm glad they did; I hadn't even heard of this book, and it's been out since 1992. I just got my hands on a copy, and the first few chapters have been fascinating."

—*Open Systems Today*

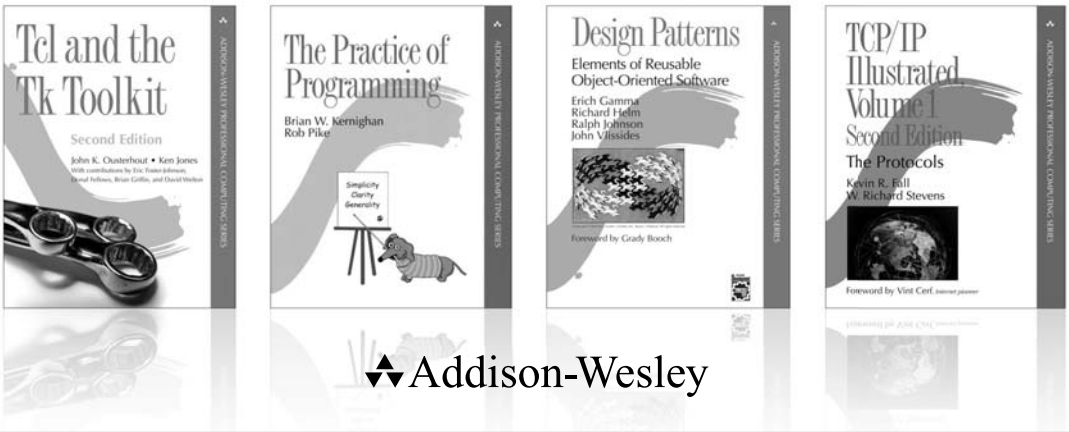
"A much more readable and detailed treatment of [UNIX internals] can be found in *Advanced Programming in the UNIX® Environment* by W. Richard Stevens (Addison-Wesley). This book includes lots of realistic examples, and I find it quite helpful when I have systems programming tasks to do."

—*RS/Magazine*

Advanced Programming in the UNIX® Environment

Third Edition

The Addison-Wesley Professional Computing Series



Visit informit.com/series/professionalcomputing
for a complete list of available publications.

The Addison-Wesley Professional Computing Series was created in 1990 to provide serious programmers and networking professionals with well-written and practical reference books. There are few places to turn for accurate and authoritative books on current and cutting-edge technology. We hope that our books will help you understand the state of the art in programming languages, operating systems, and networks.

Consulting Editor Brian W. Kernighan



Make sure to connect with us!
informit.com/socialconnect

Addison Wesley | informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE | Safari Books Online

ALWAYS LEARNING PEARSON

Advanced Programming in the UNIX® Environment

Third Edition

W. Richard Stevens
Stephen A. Rago

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data
Stevens, W. Richard.
Advanced programming in the UNIX environment/W. Richard Stevens, Stephen A. Rago. — Third edition.
pages cm
Includes bibliographical references and index.
ISBN 978-0-321-63773-4 (pbk. : alk. paper)
1. Operating systems (Computers) 2. UNIX (Computer file) I. Rago, Stephen A. II. Title.
QA76.76.O63S754 2013
005.4'32—dc23

2013004509

Copyright © 2013 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-63773-4
ISBN-10: 0-321-63773-9
Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.
First printing, May 2013

For my parents, Len & Grace

This page intentionally left blank

Contents

Foreword to the Second Edition **xix**

Preface **xxi**

Preface to the Second Edition **xxv**

Preface to the First Edition **xxix**

Chapter 1. UNIX System Overview **1**

1.1 Introduction 1
1.2 UNIX Architecture 1
1.3 Logging In 2
1.4 Files and Directories 4
1.5 Input and Output 8
1.6 Programs and Processes 10
1.7 Error Handling 14
1.8 User Identification 16
1.9 Signals 18
1.10 Time Values 20
1.11 System Calls and Library Functions 21
1.12 Summary 23

Chapter 2. UNIX Standardization and Implementations **25**

2.1 Introduction 25

2.2 UNIX Standardization 25
2.2.1 ISO C 25
2.2.2 IEEE POSIX 26
2.2.3 The Single UNIX Specification 30
2.2.4 FIPS 32
2.3 UNIX System Implementations 33
2.3.1 UNIX System V Release 4 33
2.3.2 4.4BSD 34
2.3.3 FreeBSD 34
2.3.4 Linux 35
2.3.5 Mac OS X 35
2.3.6 Solaris 35
2.3.7 Other UNIX Systems 35
2.4 Relationship of Standards and Implementations 36
2.5 Limits 36
2.5.1 ISO C Limits 37
2.5.2 POSIX Limits 38
2.5.3 XSI Limits 41
2.5.4 `sysconf`, `pathconf`, and `fpathconf` Functions 42
2.5.5 Indeterminate Runtime Limits 49
2.6 Options 53
2.7 Feature Test Macros 57
2.8 Primitive System Data Types 58
2.9 Differences Between Standards 58
2.10 Summary 60

Chapter 3. File I/O 61

3.1 Introduction 61
3.2 File Descriptors 61
3.3 `open` and `openat` Functions 62
3.4 `creat` Function 66
3.5 `close` Function 66
3.6 `lseek` Function 66
3.7 `read` Function 71
3.8 `write` Function 72
3.9 I/O Efficiency 72
3.10 File Sharing 74
3.11 Atomic Operations 77
3.12 `dup` and `dup2` Functions 79
3.13 `sync`, `fsync`, and `fdatasync` Functions 81
3.14 `fcntl` Function 82

3.15 ioctl Function 87
3.16 /dev/fd 88
3.17 Summary 90

Chapter 4. Files and Directories 93

4.1 Introduction 93
4.2 stat, fstat, fstatat, and lstat Functions 93
4.3 File Types 95
4.4 Set-User-ID and Set-Group-ID 98
4.5 File Access Permissions 99
4.6 Ownership of New Files and Directories 101
4.7 access and faccessat Functions 102
4.8 umask Function 104
4.9 chmod, fchmod, and fchmodat Functions 106
4.10 Sticky Bit 108
4.11 chown, fchown, fchownat, and lchown Functions 109
4.12 File Size 111
4.13 File Truncation 112
4.14 File Systems 113
4.15 link, linkat, unlink, unlinkat, and remove Functions 116
4.16 rename and renameat Functions 119
4.17 Symbolic Links 120
4.18 Creating and Reading Symbolic Links 123
4.19 File Times 124
4.20 futimens, utimensat, and utimes Functions 126
4.21 mkdir, mkdirat, and rmdir Functions 129
4.22 Reading Directories 130
4.23 chdir, fchdir, and getcwd Functions 135
4.24 Device Special Files 137
4.25 Summary of File Access Permission Bits 140
4.26 Summary 140

Chapter 5. Standard I/O Library 143

5.1 Introduction 143
5.2 Streams and FILE Objects 143
5.3 Standard Input, Standard Output, and Standard Error 145
5.4 Buffering 145
5.5 Opening a Stream 148

5.6 Reading and Writing a Stream 150
5.7 Line-at-a-Time I/O 152
5.8 Standard I/O Efficiency 153
5.9 Binary I/O 156
5.10 Positioning a Stream 157
5.11 Formatted I/O 159
5.12 Implementation Details 164
5.13 Temporary Files 167
5.14 Memory Streams 171
5.15 Alternatives to Standard I/O 174
5.16 Summary 175

Chapter 6. System Data Files and Information 177

6.1 Introduction 177
6.2 Password File 177
6.3 Shadow Passwords 181
6.4 Group File 182
6.5 Supplementary Group IDs 183
6.6 Implementation Differences 184
6.7 Other Data Files 185
6.8 Login Accounting 186
6.9 System Identification 187
6.10 Time and Date Routines 189
6.11 Summary 196

Chapter 7. Process Environment 197

7.1 Introduction 197
7.2 `main` Function 197
7.3 Process Termination 198
7.4 Command-Line Arguments 203
7.5 Environment List 203
7.6 Memory Layout of a C Program 204
7.7 Shared Libraries 206
7.8 Memory Allocation 207
7.9 Environment Variables 210
7.10 `setjmp` and `longjmp` Functions 213
7.11 `getrlimit` and `setrlimit` Functions 220
7.12 Summary 225

Chapter 8. Process Control 227

8.1 Introduction 227

8.2 Process Identifiers 227
8.3 `fork` Function 229
8.4 `vfork` Function 234
8.5 `exit` Functions 236
8.6 `wait` and `waitpid` Functions 238
8.7 `waitid` Function 244
8.8 `wait3` and `wait4` Functions 245
8.9 Race Conditions 245
8.10 `exec` Functions 249
8.11 Changing User IDs and Group IDs 255
8.12 Interpreter Files 260
8.13 `system` Function 264
8.14 Process Accounting 269
8.15 User Identification 275
8.16 Process Scheduling 276
8.17 Process Times 280
8.18 Summary 282

Chapter 9. Process Relationships 285

9.1 Introduction 285
9.2 Terminal Logins 285
9.3 Network Logins 290
9.4 Process Groups 293
9.5 Sessions 295
9.6 Controlling Terminal 296
9.7 `tcgetpgrp`, `tcsetpgrp`, and `tcgetsid` Functions 298
9.8 Job Control 299
9.9 Shell Execution of Programs 303
9.10 Orphaned Process Groups 307
9.11 FreeBSD Implementation 310
9.12 Summary 312

Chapter 10. Signals 313

10.1 Introduction 313
10.2 Signal Concepts 313
10.3 `signal` Function 323
10.4 Unreliable Signals 326
10.5 Interrupted System Calls 327
10.6 Reentrant Functions 330
10.7 `SIGCLD` Semantics 332

10.8 Reliable-Signal Terminology and Semantics 335
10.9 `kill` and `raise` Functions 336
10.10 `alarm` and `pause` Functions 338
10.11 Signal Sets 344
10.12 `sigprocmask` Function 346
10.13 `sigpending` Function 347
10.14 `sigaction` Function 349
10.15 `sigsetjmp` and `siglongjmp` Functions 355
10.16 `sigsuspend` Function 359
10.17 `abort` Function 365
10.18 `system` Function 367
10.19 `sleep`, `nanosleep`, and `clock_nanosleep` Functions 373
10.20 `sigqueue` Function 376
10.21 Job-Control Signals 377
10.22 Signal Names and Numbers 379
10.23 Summary 381

Chapter 11. Threads 383

11.1 Introduction 383
11.2 Thread Concepts 383
11.3 Thread Identification 384
11.4 Thread Creation 385
11.5 Thread Termination 388
11.6 Thread Synchronization 397
11.6.1 Mutexes 399
11.6.2 Deadlock Avoidance 402
11.6.3 `pthread_mutex_timedlock` Function 407
11.6.4 Reader–Writer Locks 409
11.6.5 Reader–Writer Locking with Timeouts 413
11.6.6 Condition Variables 413
11.6.7 Spin Locks 417
11.6.8 Barriers 418
11.7 Summary 422

Chapter 12. Thread Control 425

12.1 Introduction 425
12.2 Thread Limits 425
12.3 Thread Attributes 426
12.4 Synchronization Attributes 430
12.4.1 Mutex Attributes 430

12.4.2 Reader–Writer Lock Attributes 439
12.4.3 Condition Variable Attributes 440
12.4.4 Barrier Attributes 441
12.5 Reentrancy 442
12.6 Thread-Specific Data 446
12.7 Cancel Options 451
12.8 Threads and Signals 453
12.9 Threads and `fork` 457
12.10 Threads and I/O 461
12.11 Summary 462

Chapter 13. Daemon Processes 463

13.1 Introduction 463
13.2 Daemon Characteristics 463
13.3 Coding Rules 466
13.4 Error Logging 469
13.5 Single-Instance Daemons 473
13.6 Daemon Conventions 474
13.7 Client–Server Model 479
13.8 Summary 480

Chapter 14. Advanced I/O 481

14.1 Introduction 481
14.2 Nonblocking I/O 481
14.3 Record Locking 485
14.4 I/O Multiplexing 500
14.4.1 `select` and `pselect` Functions 502
14.4.2 `poll` Function 506
14.5 Asynchronous I/O 509
14.5.1 System V Asynchronous I/O 510
14.5.2 BSD Asynchronous I/O 510
14.5.3 POSIX Asynchronous I/O 511
14.6 `readv` and `writev` Functions 521
14.7 `readn` and `writen` Functions 523
14.8 Memory-Mapped I/O 525
14.9 Summary 531

Chapter 15. Interprocess Communication 533

15.1 Introduction 533
15.2 Pipes 534
15.3 `popen` and `pclose` Functions 541

15.4 Coprocesses 548
15.5 FIFOs 552
15.6 XSI IPC 556
15.6.1 Identifiers and Keys 556
15.6.2 Permission Structure 558
15.6.3 Configuration Limits 559
15.6.4 Advantages and Disadvantages 559
15.7 Message Queues 561
15.8 Semaphores 565
15.9 Shared Memory 571
15.10 POSIX Semaphores 579
15.11 Client–Server Properties 585
15.12 Summary 587

Chapter 16. Network IPC: Sockets 589

16.1 Introduction 589
16.2 Socket Descriptors 590
16.3 Addressing 593
16.3.1 Byte Ordering 593
16.3.2 Address Formats 595
16.3.3 Address Lookup 597
16.3.4 Associating Addresses with Sockets 604
16.4 Connection Establishment 605
16.5 Data Transfer 610
16.6 Socket Options 623
16.7 Out-of-Band Data 626
16.8 Nonblocking and Asynchronous I/O 627
16.9 Summary 628

Chapter 17. Advanced IPC 629

17.1 Introduction 629
17.2 UNIX Domain Sockets 629
17.2.1 Naming UNIX Domain Sockets 634
17.3 Unique Connections 635
17.4 Passing File Descriptors 642
17.5 An Open Server, Version 1 653
17.6 An Open Server, Version 2 659
17.7 Summary 669

Chapter 18. Terminal I/O 671

18.1 Introduction 671

18.2 Overview 671
18.3 Special Input Characters 678
18.4 Getting and Setting Terminal Attributes 683
18.5 Terminal Option Flags 683
18.6 `stty` Command 691
18.7 Baud Rate Functions 692
18.8 Line Control Functions 693
18.9 Terminal Identification 694
18.10 Canonical Mode 700
18.11 Noncanonical Mode 703
18.12 Terminal Window Size 710
18.13 `termcap`, `terminfo`, and `curses` 712
18.14 Summary 713

Chapter 19. Pseudo Terminals 715

19.1 Introduction 715
19.2 Overview 715
19.3 Opening Pseudo-Terminal Devices 722
19.4 `pty_fork` Function 726
19.5 `pty` Program 729
19.6 Using the `pty` Program 733
19.7 Advanced Features 740
19.8 Summary 741

Chapter 20. A Database Library 743

20.1 Introduction 743
20.2 History 743
20.3 The Library 744
20.4 Implementation Overview 746
20.5 Centralized or Decentralized? 750
20.6 Concurrency 752
20.7 Building the Library 753
20.8 Source Code 753
20.9 Performance 781
20.10 Summary 786

Chapter 21. Communicating with a Network Printer 789

21.1 Introduction 789
21.2 The Internet Printing Protocol 789
21.3 The Hypertext Transfer Protocol 792
21.4 Printer Spooling 793

21.5 Source Code 795
21.6 Summary 843

Appendix A. Function Prototypes **845**

Appendix B. Miscellaneous Source Code **895**

B.1 Our Header File 895
B.2 Standard Error Routines 898

Appendix C. Solutions to Selected Exercises **905**

Bibliography **947**

Index **955**

Foreword to the Second Edition

At some point during nearly every interview I give, as well as in question periods after talks, I get asked some variant of the same question: "Did you expect Unix to last for so long?" And of course the answer is always the same: No, we didn't quite anticipate what has happened. Even the observation that the system, in some form, has been around for well more than half the lifetime of the commercial computing industry is now dated.

The course of developments has been turbulent and complicated. Computer technology has changed greatly since the early 1970s, most notably in universal networking, ubiquitous graphics, and readily available personal computing, but the system has somehow managed to accommodate all of these phenomena. The commercial environment, although today dominated on the desktop by Microsoft and Intel, has in some ways moved from single-supplier to multiple sources and, in recent years, to increasing reliance on public standards and on freely available source.

Fortunately, Unix, considered as a phenomenon and not just a brand, has been able to move with and even lead this wave. AT&T in the 1970s and 1980s was protective of the actual Unix source code, but encouraged standardization efforts based on the system's interfaces and languages. For example, the SVID—the System V Interface Definition—was published by AT&T, and it became the basis for the POSIX work and its follow-ons. As it happened, Unix was able to adapt rather gracefully to a networked environment and, perhaps less elegantly, but still adequately, to a graphical one. And as it also happened, the basic Unix kernel interface and many of its characteristic user-level tools were incorporated into the technological foundations of the open-source movement.

It is important that papers and writings about the Unix system were always encouraged, even while the software of the system itself was proprietary, for example Maurice Bach's book, *The Design of the Unix Operating System*. In fact, I would claim that

a central reason for the system's longevity has been that it has attracted remarkably talented writers to explain its beauties and mysteries. Brian Kernighan is one of these; Rich Stevens is certainly another. The first edition of this book, along with his series of books about networking, are rightfully regarded as remarkably well-crafted works of exposition, and became hugely popular.

However, the first edition of this book was published before Linux and the several open-source renditions of the Unix interface that stemmed from the Berkeley CSRG became widespread, and also at a time when many people's networking consisted of a serial modem. Steve Rago has carefully updated this book to account for the technology changes, as well as developments in various ISO and IEEE standards since its first publication. Thus his examples are fresh, and freshly tested.

It's a most worthy second edition of a classic.

Murray Hill, New Jersey Dennis Ritchie
March 2005

Preface

Introduction

It's been almost eight years since I first updated *Advanced Programming in the UNIX Environment*, and already so much has changed.

- Before the second edition was published, The Open Group created a 2004 edition of the Single UNIX Specification, folding in the changes from two sets of corrigenda. In 2008, The Open Group created a new version of the Single UNIX Specification, updating the base definitions, adding new interfaces, and removing obsolete ones. This was called the 2008 version of POSIX.1, which included version 7 of the Base Specification and was published in 2009. In 2010, this was bundled with an updated curses interface and reissued as version 4 of the Single UNIX Specification.
- Versions 10.5, 10.6, and 10.8 of the Mac OS X operating system, running on Intel processors, have been certified to be UNIX® systems by The Open Group.
- Apple Computer discontinued development of Mac OS X for the PowerPC platform. From Release 10.6 (Snow Leopard) onward, new operating system versions are released for the x86 platform only.
- The Solaris operating system was released in open source form to try to compete with the popularity of the open source model followed by FreeBSD, Linux, and Mac OS X. After Oracle Corporation bought Sun Microsystems in 2010, it discontinued the development of OpenSolaris. Instead, the Solaris community formed the Illumos project to continue open source development based on OpenSolaris. For more information, see `http://www.illumos.org`.

- In 2011, the C standard was updated, but because systems haven't caught up yet with the changes, we still refer to the 1999 version in this text.

Most notably, the platforms used in the second edition have become out-of-date. In this book, the third edition, I cover the following platforms:

1. FreeBSD 8.0, a descendant of the 4.4BSD release from the Computer Systems Research Group at the University of California at Berkeley, running on a 32-bit Intel Pentium processor.
2. Linux 3.2.0 (the Ubuntu 12.04 distribution), a free UNIX-like operating system, running on a 64-bit Intel Core i5 processor.
3. Apple Mac OS X, version 10.6.8 (Darwin 10.8.0) on a 64-bit Intel Core 2 Duo processor. (Darwin is based on FreeBSD and Mach.) I chose to switch to an Intel platform instead of continuing with one based on the PowerPC, because the latest versions of Mac OS X are no longer being ported to the PowerPC platform. The drawback to this choice is that the processors covered are now slanted in favor of Intel. When discussing issues of heterogeneity, it is helpful to have processors with different characteristics, such as byte ordering and integer size.
4. Solaris 10, a derivative of System V Release 4 from Sun Microsystems (now Oracle), running on a 64-bit UltraSPARC IIi processor.

Changes from the Second Edition

One of the biggest changes to the Single UNIX Specification in POSIX.1-2008 is the demotion of the STREAMS-related interfaces to obsolescent status. This is the first step before these interfaces are removed entirely in a future version of the standard. Because of this, I have reluctantly removed the STREAMS content from this edition of the book. This is an unfortunate change, because the STREAMS interfaces provided a nice contrast to the socket interfaces, and in many ways were more flexible. Admittedly, I am not entirely unbiased when it comes to the STREAMS mechanism, but there is no debating the reduced role it is playing in current systems:

- Linux doesn't include STREAMS in its base system, although packages (LiS and OpenSS7) are available to add this functionality.
- Although Solaris 10 includes STREAMS, Solaris 11 uses a socket implementation that is not built on top of STREAMS.
- Mac OS X doesn't include support for STREAMS.
- FreeBSD doesn't include support for STREAMS (and never did).

So with the removal of the STREAMS-related material, an opportunity exists to replace it with new topics, such as POSIX asynchronous I/O.

In the second edition, the Linux version covered was based on the 2.4 version of the source. In this edition, I have updated the version of Linux to 3.2. One of the largest

area of differences between these two versions is the threads subsystem. Between Linux 2.4 and Linux 2.6, the threads implementation was changed to the Native POSIX Thread Library (NPTL). NPTL makes threads on Linux behave more like threads on the other systems.

In total, this edition includes more than 70 new interfaces, including interfaces to handle asynchronous I/O, spin locks, barriers, and POSIX semaphores. Most obsolete interfaces are removed, except for a few ubiquitous ones.

Acknowledgments

Many readers have e-mailed comments and bug reports on the second edition. My thanks to them for improving the accuracy of the information presented. The following people were the first to make a particular suggestion or point out a specific error: Seth Arnold, Luke Bakken, Rick Ballard, Johannes Bittner, David Bronder, Vlad Buslov, Peter Butler, Yuching Chen, Mike Cheng, Jim Collins, Bob Cousins, Will Dennis, Thomas Dickey, Loïc Domaigné, Igor Fuksman, Alex Gezerlis, M. Scott Gordon, Timothy Goya, Tony Graham, Michael Hobgood, Michael Kerrisk, Youngho Kwon, Richard Li, Xueke Liu, Yun Long, Dan McGregor, Dylan McNamee, Greg Miller, Simon Morgan, Harry Newton, Jim Oldfield, Scott Parish, Zvezdan Petkovic, David Reiss, Konstantinos Sakoutis, David Smoot, David Somers, Andriy Tkachuk, Nathan Weeks, Florian Weimer, Qingyang Xu, and Michael Zalokar.

The technical reviewers improved the accuracy of the information presented. Thanks to Steve Albert, Bogdan Barbu, and Robert Day. Special thanks to Geoff Clare and Andrew Josey for providing insights into the Single UNIX Specification and helping to improve the accuracy of Chapter 2. Also, thanks to Ken Thompson for answering history questions.

Once again, the staff at Addison-Wesley was great to work with. Thanks to Kim Boedigheimer, Romny French, John Fuller, Jessica Goldstein, Julie Nahil, and Debra Williams-Cauley. In addition, thanks to Jill Hobbs for providing her copyediting expertise this time around.

Finally, thanks to my family for their understanding while I spent so much time working on this updated edition.

As before, the source code presented here is available at `www.apuebook.com`. I welcome e-mail from any readers with comments, suggestions, or bug fixes.

Warren, New Jersey
January 2013

Stephen A. Rago
`sar@apuebook.com`

This page intentionally left blank

Preface to the Second Edition

Introduction

Rich Stevens and I first met through an e-mail exchange when I reported a typographical error in his first book, *UNIX Network Programming*. He used to kid me about being the person to send him his first errata notice for the book. Until his death in 1999, we exchanged e-mail irregularly, usually when one of us had a question we thought the other might be able to answer. We met for dinner at USENIX conferences and when Rich was teaching in the area.

Rich Stevens was a friend who always conducted himself as a gentleman. When I wrote *UNIX System V Network Programming* in 1993, I intended it to be a System V version of Rich's *UNIX Network Programming*. As was his nature, Rich gladly reviewed chapters for me, and treated me not as a competitor, but as a colleague. We often talked about collaborating on a STREAMS version of his *TCP/IP Illustrated* book. Had events been different, we might have actually done it, but since Rich is no longer with us, revising *Advanced Programming in the UNIX Environment* is the closest I'll ever get to writing a book with him.

When the editors at Addison-Wesley told me that they wanted to update Rich's book, I thought that there wouldn't be too much to change. Even after 13 years, Rich's work still holds up well. But the UNIX industry is vastly different today from what it was when the book was first published.

- The System V variants are slowly being replaced by Linux. The major system vendors that ship their hardware with their own versions of the UNIX System have either made Linux ports available or announced support for Linux. Solaris is perhaps the last descendant of UNIX System V Release 4 with any appreciable market share.

- After 4.4BSD was released, the Computing Science Research Group (CSRG) from the University of California at Berkeley decided to put an end to its development of the UNIX operating system, but several different groups of volunteers still maintain publicly available versions.
- The introduction of Linux, supported by thousands of volunteers, has made it possible for anyone with a computer to run an operating system similar to the UNIX System, with freely available source code for the newest hardware devices. The success of Linux is something of a curiosity, given that several free BSD alternatives are readily available.
- Continuing its trend as an innovative company, Apple Computer abandoned its old Mac operating system and replaced it with one based on Mach and FreeBSD.

Thus, I've tried to update the information presented in this book to reflect these four platforms.

After Rich wrote *Advanced Programming in the UNIX Environment* in 1992, I got rid of most of my UNIX programmer's manuals. To this day, the two books I keep closest to my desk are a dictionary and a copy of *Advanced Programming in the UNIX Environment*. I hope you find this revision equally useful.

Changes from the First Edition

Rich's work holds up well. I've tried not to change his original vision for this book, but a lot has happened in 13 years. This is especially true with the standards that affect the UNIX programming interface.

Throughout the book, I've updated interfaces that have changed from the ongoing efforts in standards organizations. This is most noticeable in Chapter 2, since its primary topic is standards. The 2001 version of the POSIX.1 standard, which we use in this revision, is much more comprehensive than the 1990 version on which the first edition of this book was based. The 1990 ISO C standard was updated in 1999, and some changes affect the interfaces in the POSIX.1 standard.

A lot more interfaces are now covered by the POSIX.1 specification. The base specifications of the Single UNIX Specification (published by The Open Group, formerly X/Open) have been merged with POSIX.1. POSIX.1 now includes several 1003.1 standards and draft standards that were formerly published separately.

Accordingly, I've added chapters to cover some new topics. Threads and multithreaded programming are important concepts because they present a cleaner way for programmers to deal with concurrency and asynchrony.

The socket interface is now part of POSIX.1. It provides a single interface to interprocess communication (IPC), regardless of the location of the process, and is a natural extension of the IPC chapters.

I've omitted most of the real-time interfaces that appear in POSIX.1. These are best treated in a text devoted to real-time programming. One such book appears in the bibliography.

I've updated the case studies in the last chapters to cover more relevant real-world examples. For example, few systems these days are connected to a PostScript printer

via a serial or parallel port. Most PostScript printers today are accessed via a network, so I've changed the case study that deals with PostScript printer communication to take this into account.

The chapter on modem communication is less relevant these days. So that the original material is not lost, however, it is available on the book's Web site in two formats: PostScript (`http://www.apuebook.com/lostchapter/modem.ps`) and PDF (`http://www.apuebook.com/lostchapter/modem.pdf`).

The source code for the examples shown in this book is also available at `www.apuebook.com`. Most of the examples have been run on four platforms:

1. FreeBSD 5.2.1, a derivative of the 4.4BSD release from the Computer Systems Research Group at the University of California at Berkeley, running on an Intel Pentium processor
2. Linux 2.4.22 (the Mandrake 9.2 distribution), a free UNIX-like operating system, running on Intel Pentium processors
3. Solaris 9, a derivative of System V Release 4 from Sun Microsystems, running on a 64-bit UltraSPARC IIi processor
4. Darwin 7.4.0, an operating environment based on FreeBSD and Mach, supported by Apple Mac OS X, version 10.3, on a PowerPC processor

Acknowledgments

Rich Stevens wrote the first edition of this book on his own, and it became an instant classic.

I couldn't have updated this book without the support of my family. They put up with piles of papers scattered about the house (well, more so than usual), my monopolizing most of the computers in the house, and lots of hours with my face buried behind a computer terminal. My wife, Jeanne, even helped out by installing Linux for me on one of the test machines.

The technical reviewers suggested many improvements and helped make sure that the content was accurate. Many thanks to David Bausum, David Boreham, Keith Bostic, Mark Ellis, Phil Howard, Andrew Josey, Mukesh Kacker, Brian Kernighan, Bengt Kleberg, Ben Kuperman, Eric Raymond, and Andy Rudoff.

I'd also like to thank Andy Rudoff for answering questions about Solaris and Dennis Ritchie for digging up old papers and answering history questions. Once again, the staff at Addison-Wesley was great to work with. Thanks to Tyrrell Albaugh, Mary Franz, John Fuller, Karen Gettman, Jessica Goldstein, Noreen Regina, and John Wait. My thanks to Evelyn Pyle for the fine job of copyediting.

As Rich did, I also welcome electronic mail from any readers with comments, suggestions, or bug fixes.

Warren, New Jersey
April 2005

Stephen A. Rago
`sar@apuebook.com`

This page intentionally left blank

Preface to the First Edition

Introduction

This book describes the programming interface to the Unix system—the system call interface and many of the functions provided in the standard C library. It is intended for anyone writing programs that run under Unix.

Like most operating systems, Unix provides numerous services to the programs that are running—open a file, read a file, start a new program, allocate a region of memory, get the current time-of-day, and so on. This has been termed the *system call interface*. Additionally, the standard C library provides numerous functions that are used by almost every C program (format a variable's value for output, compare two strings, etc.).

The system call interface and the library routines have traditionally been described in Sections 2 and 3 of the *Unix Programmer's Manual*. This book is not a duplication of these sections. Examples and rationale are missing from the *Unix Programmer's Manual*, and that's what this book provides.

Unix Standards

The proliferation of different versions of Unix during the 1980s has been tempered by the various international standards that were started during the late 1980s. These include the ANSI standard for the C programming language, the IEEE POSIX family (still being developed), and the X/Open portability guide.

This book also describes these standards. But instead of just describing the standards by themselves, we describe them in relation to popular implementations of the standards—System V Release 4 and the forthcoming 4.4BSD. This provides a real-world description, which is often lacking from the standard itself and from books that describe only the standard.

Organization of the Book

This book is divided into six parts:

1. An overview and introduction to basic Unix programming concepts and terminology (Chapter 1), with a discussion of the various Unix standardization efforts and different Unix implementations (Chapter 2).
2. I/O—unbuffered I/O (Chapter 3), properties of files and directories (Chapter 4), the standard I/O library (Chapter 5), and the standard system data files (Chapter 6).
3. Processes—the environment of a Unix process (Chapter 7), process control (Chapter 8), the relationships between different processes (Chapter 9), and signals (Chapter 10).
4. More I/O—terminal I/O (Chapter 11), advanced I/O (Chapter 12), and daemon processes (Chapter 13).
5. IPC—Interprocess communication (Chapters 14 and 15).
6. Examples—a database library (Chapter 16), communicating with a PostScript printer (Chapter 17), a modem dialing program (Chapter 18), and using pseudo terminals (Chapter 19).

A reading familiarity with C would be beneficial as would some experience using Unix. No prior programming experience with Unix is assumed. This text is intended for programmers familiar with Unix and programmers familiar with some other operating system who wish to learn the details of the services provided by most Unix systems.

Examples in the Text

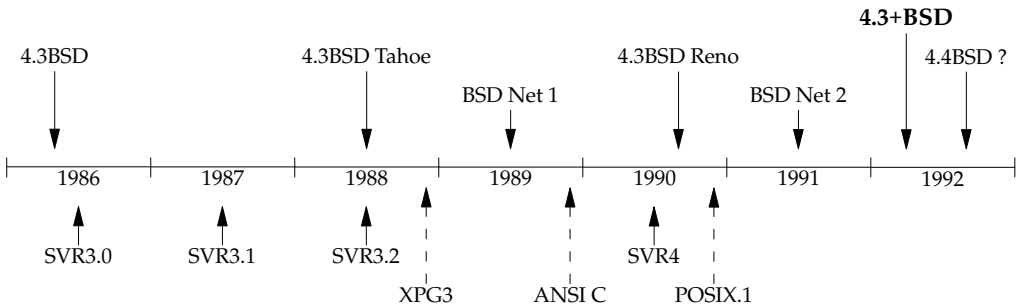
This book contains many examples—approximately 10,000 lines of source code. All the examples are in the C programming language. Furthermore, these examples are in ANSI C. You should have a copy of the *Unix Programmer's Manual* for your system handy while reading this book, since reference is made to it for some of the more esoteric and implementation-dependent features.

Almost every function and system call is demonstrated with a small, complete program. This lets us see the arguments and return values and is often easier to comprehend than the use of the function in a much larger program. But since some of the small programs are contrived examples, a few bigger examples are also included (Chapters 16, 17, 18, and 19). These larger examples demonstrate the programming techniques in larger, real-world examples.

All the examples have been included in the text directly from their source files. A machine-readable copy of all the examples is available via anonymous FTP from the Internet host `ftp.uu.net` in the file `published/books/stevens.advprog.tar.Z`. Obtaining the source code allows you to modify the programs from this text and experiment with them on your system.

Systems Used to Test the Examples

Unfortunately all operating systems are moving targets. Unix is no exception. The following diagram shows the recent evolution of the various versions of System V and 4.xBSD.



4.xBSD are the various systems from the Computer Systems Research Group at the University of California at Berkeley. This group also distributes the BSD Net 1 and BSD Net 2 releases—publicly available source code from the 4.xBSD systems. SVRx refers to System V Release x from AT&T. XPG3 is the X/Open Portability Guide, Issue 3, and ANSI C is the ANSI standard for the C programming language. POSIX.1 is the IEEE and ISO standard for the interface to a Unix-like system. We'll have more to say about these different standards and the various versions of Unix in Sections 2.2 and 2.3.

> **In this text we use the term *4.3+BSD* to refer to the Unix system from Berkeley that is somewhere between the BSD Net 2 release and 4.4BSD.**
>
> At the time of this writing, 4.4BSD was not released, so the system could not be called 4.4BSD. Nevertheless a simple name was needed to refer to this system and *4.3+BSD* is used throughout the text.

Most of the examples in this text have been run on four different versions of Unix:

1. Unix System V/386 Release 4.0 Version 2.0 (“vanilla SVR4”) from U.H. Corp. (UHC), on an Intel 80386 processor.
2. 4.3+BSD at the Computer Systems Research Group, Computer Science Division, University of California at Berkeley, on a Hewlett Packard workstation.
3. BSD/386 (a derivative of the BSD Net 2 release) from Berkeley Software Design, Inc., on an Intel 80386 processor. This system is almost identical to what we call 4.3+BSD.
4. SunOS 4.1.1 and 4.1.2 (systems with a strong Berkeley heritage but many System V features) from Sun Microsystems, on a SPARCstation SLC.

Numerous timing tests are provided in the text and the systems used for the test are identified.

Acknowledgments

Once again I am indebted to my family for their love, support, and many lost weekends over the past year and a half. Writing a book is, in many ways, a family affair. Thank you Sally, Bill, Ellen, and David.

I am especially grateful to Brian Kernighan for his help in the book. His numerous thorough reviews of the entire manuscript and his gentle prodding for better prose hopefully show in the final result. Steve Rago was also a great resource, both in reviewing the entire manuscript and answering many questions about the details and history of System V. My thanks to the other technical reviewers used by Addison-Wesley, who provided valuable comments on various portions of the manuscript: Maury Bach, Mark Ellis, Jeff Gitlin, Peter Honeyman, John Linderman, Doug McIlroy, Evi Nemeth, Craig Partridge, Dave Presotto, Gary Wilson, and Gary Wright.

Keith Bostic and Kirk McKusick at the U.C. Berkeley CSRG provided an account that was used to test the examples on the latest BSD system. (Many thanks to Peter Salus too.) Sam Nataros and Joachim Sacksen at UHC provided the copy of SVR4 used to test the examples. Trent Hein helped obtain the alpha and beta copies of BSD/386.

Other friends have helped in many small, but significant ways over the past few years: Paul Lucchina, Joe Godsil, Jim Hogue, Ed Tankus, and Gary Wright. My editor at Addison-Wesley, John Wait, has been a great friend through it all. He never complained when the due date slipped and the page count kept increasing. A special thanks to the National Optical Astronomy Observatories (NOAO), especially Sidney Wolff, Richard Wolff, and Steve Grandi, for providing computer time.

Real Unix books are written using troff and this book follows that time-honored tradition. Camera-ready copy of the book was produced by the author using the groff package written by James Clark. Many thanks to James Clark for providing this excellent system and for his rapid response to bug fixes. Perhaps someday I will really understand troff footer traps.

I welcome electronic mail from any readers with comments, suggestions, or bug fixes.

Tucson, Arizona
April 1992

W. Richard Stevens
`rstevens@kohala.com`
`http://www.kohala.com/~rstevens`

1

UNIX System Overview

1.1 Introduction

All operating systems provide services for programs they run. Typical services include executing a new program, opening a file, reading a file, allocating a region of memory, getting the current time of day, and so on. The focus of this text is to describe the services provided by various versions of the UNIX operating system.

Describing the UNIX System in a strictly linear fashion, without any forward references to terms that haven't been described yet, is nearly impossible (and would probably be boring). This chapter provides a whirlwind tour of the UNIX System from a programmer's perspective. We'll give some brief descriptions and examples of terms and concepts that appear throughout the text. We describe these features in much more detail in later chapters. This chapter also provides an introduction to and overview of the services provided by the UNIX System for programmers new to this environment.

1.2 UNIX Architecture

In a strict sense, an operating system can be defined as the software that controls the hardware resources of the computer and provides an environment under which programs can run. Generally, we call this software the *kernel*, since it is relatively small and resides at the core of the environment. Figure 1.1 shows a diagram of the UNIX System architecture.

The interface to the kernel is a layer of software called the *system calls* (the shaded portion in Figure 1.1). Libraries of common functions are built on top of the system call

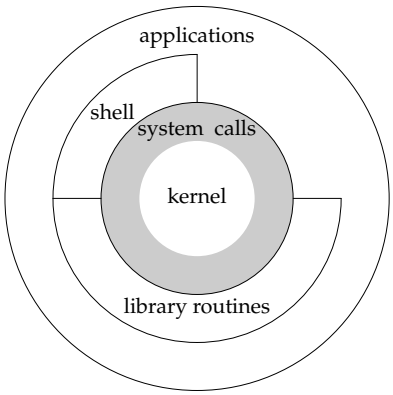


Figure 1.1 Architecture of the UNIX operating system

interface, but applications are free to use both. (We talk more about system calls and library functions in Section 1.11.) The shell is a special application that provides an interface for running other applications.

In a broad sense, an operating system consists of the kernel and all the other software that makes a computer useful and gives the computer its personality. This other software includes system utilities, applications, shells, libraries of common functions, and so on.

For example, Linux is the kernel used by the GNU operating system. Some people refer to this combination as the GNU/Linux operating system, but it is more commonly referred to as simply Linux. Although this usage may not be correct in a strict sense, it is understandable, given the dual meaning of the phrase *operating system*. (It also has the advantage of being more succinct.)

1.3 Logging In

Login Name

When we log in to a UNIX system, we enter our login name, followed by our password. The system then looks up our login name in its password file, usually the file `/etc/passwd`. If we look at our entry in the password file, we see that it's composed of seven colon-separated fields: the login name, encrypted password, numeric user ID (205), numeric group ID (105), a comment field, home directory (`/home/sar`), and shell program (`/bin/ksh`).

```
sar:x:205:105:Stephen Rago:/home/sar:/bin/ksh
```

All contemporary systems have moved the encrypted password to a different file. In Chapter 6, we'll look at these files and some functions to access them.

Shells

Once we log in, some system information messages are typically displayed, and then we can type commands to the shell program. (Some systems start a window management program when you log in, but you generally end up with a shell running in one of the windows.) A *shell* is a command-line interpreter that reads user input and executes commands. The user input to a shell is normally from the terminal (an interactive shell) or sometimes from a file (called a *shell script*). The common shells in use are summarized in Figure 1.2.

Name	Path	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
Bourne shell	`/bin/sh`	•	•	copy of `bash`	•
Bourne-again shell	`/bin/bash`	optional	•	•	•
C shell	`/bin/csh`	link to `tcsh`	optional	link to `tcsh`	•
Korn shell	`/bin/ksh`	optional	optional	•	•
TENEX C shell	`/bin/tcsh`	•	optional	•	•

Figure 1.2 Common shells used on UNIX systems

The system knows which shell to execute for us based on the final field in our entry in the password file.

The Bourne shell, developed by Steve Bourne at Bell Labs, has been in use since Version 7 and is provided with almost every UNIX system in existence. The control-flow constructs of the Bourne shell are reminiscent of Algol 68.

The C shell, developed by Bill Joy at Berkeley, is provided with all the BSD releases. Additionally, the C shell was provided by AT&T with System V/386 Release 3.2 and was also included in System V Release 4 (SVR4). (We'll have more to say about these different versions of the UNIX System in the next chapter.) The C shell was built on the 6th Edition shell, not the Bourne shell. Its control flow looks more like the C language, and it supports additional features that weren't provided by the Bourne shell: job control, a history mechanism, and command-line editing.

The Korn shell is considered a successor to the Bourne shell and was first provided with SVR4. The Korn shell, developed by David Korn at Bell Labs, runs on most UNIX systems, but before SVR4 was usually an extra-cost add-on, so it is not as widespread as the other two shells. It is upward compatible with the Bourne shell and includes those features that made the C shell popular: job control, command-line editing, and so on.

The Bourne-again shell is the GNU shell provided with all Linux systems. It was designed to be POSIX conformant, while still remaining compatible with the Bourne shell. It supports features from both the C shell and the Korn shell.

The TENEX C shell is an enhanced version of the C shell. It borrows several features, such as command completion, from the TENEX operating system (developed in 1972 at Bolt Beranek and Newman). The TENEX C shell adds many features to the C shell and is often used as a replacement for the C shell.

The shell was standardized in the POSIX 1003.2 standard. The specification was based on features from the Korn shell and Bourne shell.

The default shell used by different Linux distributions varies. Some distributions use the Bourne-again shell. Others use the BSD replacement for the Bourne shell, called `dash` (Debian Almquist shell, originally written by Kenneth Almquist and later ported to Linux). The default user shell in FreeBSD is derived from the Almquist shell. The default shell in Mac OS X is the Bourne-again shell. Solaris, having its heritage in both BSD and System V, provides all the shells shown in Figure 1.2. Free ports of the shells are available on the Internet.

Throughout the text, we will use parenthetical notes such as this to describe historical notes and to compare different implementations of the UNIX System. Often the reason for a particular implementation technique becomes clear when the historical reasons are described.

Throughout this text, we'll show interactive shell examples to execute a program that we've developed. These examples use features common to the Bourne shell, the Korn shell, and the Bourne-again shell.

1.4 Files and Directories

File System

The UNIX file system is a hierarchical arrangement of directories and files. Everything starts in the directory called *root*, whose name is the single character `/`.

A *directory* is a file that contains directory entries. Logically, we can think of each directory entry as containing a filename along with a structure of information describing the attributes of the file. The attributes of a file are such things as the type of file (regular file, directory), the size of the file, the owner of the file, permissions for the file (whether other users may access this file), and when the file was last modified. The `stat` and `fstat` functions return a structure of information containing all the attributes of a file. In Chapter 4, we'll examine all the attributes of a file in great detail.

We make a distinction between the logical view of a directory entry and the way it is actually stored on disk. Most implementations of UNIX file systems don't store attributes in the directory entries themselves, because of the difficulty of keeping them in synch when a file has multiple hard links. This will become clear when we discuss hard links in Chapter 4.

Filename

The names in a directory are called *filenames*. The only two characters that cannot appear in a filename are the slash character (`/`) and the null character. The slash separates the filenames that form a pathname (described next) and the null character terminates a pathname. Nevertheless, it's good practice to restrict the characters in a filename to a subset of the normal printing characters. (If we use some of the shell's special characters in the filename, we have to use the shell's quoting mechanism to reference the filename, and this can get complicated.) Indeed, for portability, POSIX.1 recommends restricting filenames to consist of the following characters: letters (`a–z`, `A–Z`), numbers (`0–9`), period (`.`), dash (`-`), and underscore (`_`).

Two filenames are automatically created whenever a new directory is created: `.` (called *dot*) and `..` (called *dot-dot*). Dot refers to the current directory, and dot-dot refers to the parent directory. In the root directory, dot-dot is the same as dot.

The Research UNIX System and some older UNIX System V file systems restricted a filename to 14 characters. BSD versions extended this limit to 255 characters. Today, almost all commercial UNIX file systems support at least 255-character filenames.

Pathname

A sequence of one or more filenames, separated by slashes and optionally starting with a slash, forms a *pathname*. A pathname that begins with a slash is called an *absolute pathname*; otherwise, it's called a *relative pathname*. Relative pathnames refer to files relative to the current directory. The name for the root of the file system (/) is a special-case absolute pathname that has no filename component.

Example

Listing the names of all the files in a directory is not difficult. Figure 1.3 shows a bare-bones implementation of the `ls`(1) command.

```
#include "apue.h"
#include <dirent.h>

int
main(int argc, char *argv[])
{
    DIR             *dp;
    struct dirent   *dirp;

    if (argc != 2)
        err_quit("usage: ls directory_name");

    if ((dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);
    while ((dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

Figure 1.3 List all the files in a directory

The notation `ls`(1) is the normal way to reference a particular entry in the UNIX system manuals. It refers to the entry for `ls` in Section 1. The sections are normally numbered 1 through 8, and all the entries within each section are arranged alphabetically. Throughout this text, we assume that you have a copy of the manuals for your UNIX system.

Historically, UNIX systems lumped all eight sections together into what was called the *UNIX Programmer's Manual*. As the page count increased, the trend changed to distributing the sections among separate manuals: one for users, one for programmers, and one for system administrators, for example.

> Some UNIX systems further divide the manual pages within a given section, using an uppercase letter. For example, all the standard input/output (I/O) functions in AT&T [1990e] are indicated as being in Section 3S, as in `fopen`(3S). Other systems have replaced the numeric sections with alphabetic ones, such as C for commands.

Today, most manuals are distributed in electronic form. If your manuals are online, the way to see the manual pages for the `ls` command would be something like

```
man 1 ls
```

or

```
man -s1 ls
```

Figure 1.3 is a program that just prints the name of every file in a directory, and nothing else. If the source file is named `myls.c`, we compile it into the default `a.out` executable file by running

```
cc myls.c
```

> Historically, cc(1) is the C compiler. On systems with the GNU C compilation system, the C compiler is gcc(1). Here, `cc` is usually linked to `gcc`.

Some sample output is

```
$ ./a.out /dev
.
..
cdrom
stderr
stdout
stdin
fd
sda4
sda3
sda2
sda1
sda
tty2
tty1
console
tty
zero
null
                    many more lines that aren't shown
mem
$ ./a.out /etc/ssl/private
can't open /etc/ssl/private: Permission denied
$ ./a.out /dev/tty
can't open /dev/tty: Not a directory
```

Throughout this text, we'll show commands that we run and the resulting output in this fashion: Characters that we type are shown in **`this font`**, whereas output from programs is shown `like this`. If we need to add comments to this output, we'll show

the comments in *italics*. The dollar sign that precedes our input is the prompt that is printed by the shell. We'll always show the shell prompt as a dollar sign.

Note that the directory listing is not in alphabetical order. The `ls` command sorts the names before printing them.

There are many details to consider in this 20-line program.

- First, we include a header of our own: `apue.h`. We include this header in almost every program in this text. This header includes some standard system headers and defines numerous constants and function prototypes that we use throughout the examples in the text. A listing of this header is in Appendix B.
- Next, we include a system header, `dirent.h`, to pick up the function prototypes for `opendir` and `readdir`, in addition to the definition of the `dirent` structure. On some systems, the definitions are split into multiple header files. For example, in the Ubuntu 12.04 Linux distribution, `/usr/include/dirent.h` declares the function prototypes and includes `bits/dirent.h`, which defines the `dirent` structure (and is actually stored in `/usr/include/x86_64-linux-gnu/bits`).
- The declaration of the `main` function uses the style supported by the ISO C standard. (We'll have more to say about the ISO C standard in the next chapter.)
- We take an argument from the command line, `argv[1]`, as the name of the directory to list. In Chapter 7, we'll look at how the `main` function is called and how the command-line arguments and environment variables are accessible to the program.
- Because the actual format of directory entries varies from one UNIX system to another, we use the functions `opendir`, `readdir`, and `closedir` to manipulate the directory.
- The `opendir` function returns a pointer to a `DIR` structure, and we pass this pointer to the `readdir` function. We don't care what's in the `DIR` structure. We then call `readdir` in a loop, to read each directory entry. The `readdir` function returns a pointer to a `dirent` structure or, when it's finished with the directory, a null pointer. All we examine in the `dirent` structure is the name of each directory entry (`d_name`). Using this name, we could then call the `stat` function (Section 4.2) to determine all the attributes of the file.
- We call two functions of our own to handle the errors: `err_sys` and `err_quit`. We can see from the preceding output that the `err_sys` function prints an informative message describing what type of error was encountered ("Permission denied" or "Not a directory"). These two error functions are shown and described in Appendix B. We also talk more about error handling in Section 1.7.
- When the program is done, it calls the function `exit` with an argument of 0. The function `exit` terminates a program. By convention, an argument of 0 means OK, and an argument between 1 and 255 means that an error occurred. In Section 8.5, we show how any program, such as a shell or a program that we write, can obtain the `exit` status of a program that it executes. □

Working Directory

Every process has a *working directory*, sometimes called the *current working directory*. This is the directory from which all relative pathnames are interpreted. A process can change its working directory with the `chdir` function.

For example, the relative pathname `doc/memo/joe` refers to the file or directory `joe`, in the directory `memo`, in the directory `doc`, which must be a directory within the working directory. From looking just at this pathname, we know that both `doc` and `memo` have to be directories, but we can't tell whether `joe` is a file or a directory. The pathname `/usr/lib/lint` is an absolute pathname that refers to the file or directory `lint` in the directory `lib`, in the directory `usr`, which is in the root directory.

Home Directory

When we log in, the working directory is set to our *home directory*. Our home directory is obtained from our entry in the password file (Section 1.3).

1.5 Input and Output

File Descriptors

File descriptors are normally small non-negative integers that the kernel uses to identify the files accessed by a process. Whenever it opens an existing file or creates a new file, the kernel returns a file descriptor that we use when we want to read or write the file.

Standard Input, Standard Output, and Standard Error

By convention, all shells open three descriptors whenever a new program is run: standard input, standard output, and standard error. If nothing special is done, as in the simple command

```
ls
```

then all three are connected to the terminal. Most shells provide a way to redirect any or all of these three descriptors to any file. For example,

```
ls > file.list
```

executes the `ls` command with its standard output redirected to the file named `file.list`.

Unbuffered I/O

Unbuffered I/O is provided by the functions `open`, `read`, `write`, `lseek`, and `close`. These functions all work with file descriptors.

Example

If we're willing to read from the standard input and write to the standard output, then the program in Figure 1.4 copies any regular file on a UNIX system.

```
#include "apue.h"

#define BUFFSIZE    4096

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

Figure 1.4 Copy standard input to standard output

The `<unistd.h>` header, included by `apue.h`, and the two constants `STDIN_FILENO` and `STDOUT_FILENO` are part of the POSIX standard (about which we'll have a lot more to say in the next chapter). This header contains function prototypes for many of the UNIX system services, such as the `read` and `write` functions that we call.

The constants `STDIN_FILENO` and `STDOUT_FILENO` are defined in `<unistd.h>` and specify the file descriptors for standard input and standard output. These values are 0 and 1, respectively, as required by POSIX.1, but we'll use the names for readability.

In Section 3.9, we'll examine the `BUFFSIZE` constant in detail, seeing how various values affect the efficiency of the program. Regardless of the value of this constant, however, this program still copies any regular file.

The `read` function returns the number of bytes that are read, and this value is used as the number of bytes to write. When the end of the input file is encountered, `read` returns 0 and the program stops. If a read error occurs, `read` returns –1. Most of the system functions return –1 when an error occurs.

If we compile the program into the standard name (`a.out`) and execute it as

```
./a.out > data
```

standard input is the terminal, standard output is redirected to the file `data`, and standard error is also the terminal. If this output file doesn't exist, the shell creates it by default. The program copies lines that we type to the standard output until we type the end-of-file character (usually Control-D).

If we run

```
./a.out < infile > outfile
```

then the file named `infile` will be copied to the file named `outfile`. □

In Chapter 3, we describe the unbuffered I/O functions in more detail.

Standard I/O

The standard I/O functions provide a buffered interface to the unbuffered I/O functions. Using standard I/O relieves us from having to choose optimal buffer sizes, such as the `BUFFSIZE` constant in Figure 1.4. The standard I/O functions also simplify dealing with lines of input (a common occurrence in UNIX applications). The `fgets` function, for example, reads an entire line. The `read` function, in contrast, reads a specified number of bytes. As we shall see in Section 5.4, the standard I/O library provides functions that let us control the style of buffering used by the library.

The most common standard I/O function is `printf`. In programs that call `printf`, we'll always include `<stdio.h>`—normally by including `apue.h`—as this header contains the function prototypes for all the standard I/O functions.

Example

The program in Figure 1.5, which we'll examine in more detail in Section 5.8, is like the previous program that called `read` and `write`. This program copies standard input to standard output and can copy any regular file.

```
#include "apue.h"

int
main(void)
{
    int     c;

    while ((c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

Figure 1.5 Copy standard input to standard output, using standard I/O

The function `getc` reads one character at a time, and this character is written by `putc`. After the last byte of input has been read, `getc` returns the constant `EOF` (defined in `<stdio.h>`). The standard I/O constants `stdin` and `stdout` are also defined in the `<stdio.h>` header and refer to the standard input and standard output. □

1.6 Programs and Processes

Program

A *program* is an executable file residing on disk in a directory. A program is read into memory and is executed by the kernel as a result of one of the seven `exec` functions. We'll cover these functions in Section 8.10.

Processes and Process ID

An executing instance of a program is called a *process*, a term used on almost every page of this text. Some operating systems use the term *task* to refer to a program that is being executed.

The UNIX System guarantees that every process has a unique numeric identifier called the *process ID*. The process ID is always a non-negative integer.

Example

The program in Figure 1.6 prints its process ID.

```
#include "apue.h"

int
main(void)
{
    printf("hello world from process ID %ld\n", (long)getpid());
    exit(0);
}
```

Figure 1.6 Print the process ID

If we compile this program into the file `a.out` and execute it, we have

```
$ ./a.out
hello world from process ID 851
$ ./a.out
hello world from process ID 854
```

When this program runs, it calls the function `getpid` to obtain its process ID. As we shall see later, `getpid` returns a `pid_t` data type. We don't know its size; all we know is that the standards guarantee that it will fit in a long integer. Because we have to tell `printf` the size of each argument to be printed, we have to cast the value to the largest data type that it might use (in this case, a long integer). Although most process IDs will fit in an `int`, using a `long` promotes portability. □

Process Control

There are three primary functions for process control: `fork`, `exec`, and `waitpid`. (The `exec` function has seven variants, but we often refer to them collectively as simply the `exec` function.)

Example

The process control features of the UNIX System are demonstrated using a simple program (Figure 1.7) that reads commands from standard input and executes the commands. This is a bare-bones implementation of a shell-like program.

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char    buf[MAXLINE];   /* from apue.h */
    pid_t   pid;
    int     status;

    printf("%% ");  /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {      /* child */
            execlp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

Figure 1.7 Read commands from standard input and execute them

There are several features to consider in this 30-line program.

- We use the standard I/O function `fgets` to read one line at a time from the standard input. When we type the end-of-file character (which is often Control-D) as the first character of a line, `fgets` returns a null pointer, the loop stops, and the process terminates. In Chapter 18, we describe all the special terminal characters—end of file, backspace one character, erase entire line, and so on—and how to change them.
- Because each line returned by `fgets` is terminated with a newline character, followed by a null byte, we use the standard C function `strlen` to calculate the length of the string, and then replace the newline with a null byte. We do this because the `execlp` function wants a null-terminated argument, not a newline-terminated argument.
- We call `fork` to create a new process, which is a copy of the caller. We say that the caller is the parent and that the newly created process is the child. Then `fork` returns the non-negative process ID of the new child process to the parent,

and returns 0 to the child. Because `fork` creates a new process, we say that it is called once—by the parent—but returns twice—in the parent and in the child.

- In the child, we call `execlp` to execute the command that was read from the standard input. This replaces the child process with the new program file. The combination of `fork` followed by `exec` is called spawning a new process on some operating systems. In the UNIX System, the two parts are separated into individual functions. We'll say a lot more about these functions in Chapter 8.
- Because the child calls `execlp` to execute the new program file, the parent wants to wait for the child to terminate. This is done by calling `waitpid`, specifying which process to wait for: the `pid` argument, which is the process ID of the child. The `waitpid` function also returns the termination status of the child—the `status` variable—but in this simple program, we don't do anything with this value. We could examine it to determine how the child terminated.
- The most fundamental limitation of this program is that we can't pass arguments to the command we execute. We can't, for example, specify the name of a directory to list. We can execute `ls` only on the working directory. To allow arguments would require that we parse the input line, separating the arguments by some convention, probably spaces or tabs, and then pass each argument as a separate parameter to the `execlp` function. Nevertheless, this program is still a useful demonstration of the UNIX System's process control functions.

If we run this program, we get the following results. Note that our program has a different prompt—the percent sign—to distinguish it from the shell's prompt.

```
$ ./a.out
% date
Sat Jan 21 19:42:07 EST 2012
% who
sar      console  Jan  1 14:59
sar      ttys000  Jan  1 14:59
sar      ttys001  Jan 15 15:28
% pwd
/home/sar/bk/apue/3e
% ls
Makefile
a.out
shell1.c
% ^D                                   type the end-of-file character
$                                      the regular shell prompt
```

□

The notation `^D` is used to indicate a control character. Control characters are special characters formed by holding down the control key—often labeled `Control` or `Ctrl`—on your keyboard and then pressing another key at the same time. Control-D, or `^D`, is the default end-of-file character. We'll see many more control characters when we discuss terminal I/O in Chapter 18.

Threads and Thread IDs

Usually, a process has only one thread of control—one set of machine instructions executing at a time. Some problems are easier to solve when more than one thread of control can operate on different parts of the problem. Additionally, multiple threads of control can exploit the parallelism possible on multiprocessor systems.

All threads within a process share the same address space, file descriptors, stacks, and process-related attributes. Each thread executes on its own stack, although any thread can access the stacks of other threads in the same process. Because they can access the same memory, the threads need to synchronize access to shared data among themselves to avoid inconsistencies.

Like processes, threads are identified by IDs. Thread IDs, however, are local to a process. A thread ID from one process has no meaning in another process. We use thread IDs to refer to specific threads as we manipulate the threads within a process.

Functions to control threads parallel those used to control processes. Because threads were added to the UNIX System long after the process model was established, however, the thread model and the process model have some complicated interactions, as we shall see in Chapter 12.

1.7 Error Handling

When an error occurs in one of the UNIX System functions, a negative value is often returned, and the integer `errno` is usually set to a value that tells why. For example, the `open` function returns either a non-negative file descriptor if all is OK or –1 if an error occurs. An error from `open` has about 15 possible `errno` values, such as file doesn't exist, permission problem, and so on. Some functions use a convention other than returning a negative value. For example, most functions that return a pointer to an object return a null pointer to indicate an error.

The file `<errno.h>` defines the symbol `errno` and constants for each value that `errno` can assume. Each of these constants begins with the character `E`. Also, the first page of Section 2 of the UNIX system manuals, named `intro`(2), usually lists all these error constants. For example, if `errno` is equal to the constant `EACCES`, this indicates a permission problem, such as insufficient permission to open the requested file.

On Linux, the error constants are listed in the `errno`(3) manual page.

POSIX and ISO C define `errno` as a symbol expanding into a modifiable lvalue of type integer. This can be either an integer that contains the error number or a function that returns a pointer to the error number. The historical definition is

```
extern int errno;
```

But in an environment that supports threads, the process address space is shared among multiple threads, and each thread needs its own local copy of `errno` to prevent one thread from interfering with another. Linux, for example, supports multithreaded access to `errno` by defining it as

```
extern int *__errno_location(void);
#define errno (*__errno_location())
```

There are two rules to be aware of with respect to `errno`. First, its value is never cleared by a routine if an error does not occur. Therefore, we should examine its value only when the return value from a function indicates that an error occurred. Second, the value of `errno` is never set to 0 by any of the functions, and none of the constants defined in `<errno.h>` has a value of 0.

Two functions are defined by the C standard to help with printing error messages.

```
#include <string.h>

char *strerror(int errnum);
```

Returns: pointer to message string

This function maps *errnum*, which is typically the `errno` value, into an error message string and returns a pointer to the string.

The `perror` function produces an error message on the standard error, based on the current value of `errno`, and returns.

```
#include <stdio.h>

void perror(const char *msg);
```

It outputs the string pointed to by *msg*, followed by a colon and a space, followed by the error message corresponding to the value of `errno`, followed by a newline.

Example

Figure 1.8 shows the use of these two error functions.

```
#include "apue.h"
#include <errno.h>

int
main(int argc, char *argv[])
{
    fprintf(stderr, "EACCES: %s\n", strerror(EACCES));
    errno = ENOENT;
    perror(argv[0]);
    exit(0);
}
```

Figure 1.8 Demonstrate `strerror` and `perror`

If this program is compiled into the file `a.out`, we have

```
$ ./a.out
EACCES: Permission denied
./a.out: No such file or directory
```

Note that we pass the name of the program—`argv[0]`, whose value is `./a.out`—as the argument to `perror`. This is a standard convention in the UNIX System. By doing this, if the program is executed as part of a pipeline, as in

```
prog1 < inputfile | prog2 | prog3 > outputfile
```

we are able to tell which of the three programs generated a particular error message. □

Instead of calling either `strerror` or `perror` directly, the examples in this text use the error functions shown in Appendix B. These functions let us use the variable argument list facility of ISO C to handle error conditions with a single C statement.

Error Recovery

The errors defined in `<errno.h>` can be divided into two categories: fatal and nonfatal. A fatal error has no recovery action. The best we can do is print an error message on the user's screen or to a log file, and then exit. Nonfatal errors, on the other hand, can sometimes be dealt with more robustly. Most nonfatal errors are temporary, such as a resource shortage, and might not occur when there is less activity on the system.

Resource-related nonfatal errors include `EAGAIN`, `ENFILE`, `ENOBUFS`, `ENOLCK`, `ENOSPC`, `EWOULDBLOCK`, and sometimes `ENOMEM`. `EBUSY` can be treated as nonfatal when it indicates that a shared resource is in use. Sometimes, `EINTR` can be treated as a nonfatal error when it interrupts a slow system call (more on this in Section 10.5).

The typical recovery action for a resource-related nonfatal error is to delay and retry later. This technique can be applied in other circumstances. For example, if an error indicates that a network connection is no longer functioning, it might be possible for the application to delay a short time and then reestablish the connection. Some applications use an exponential backoff algorithm, waiting a longer period of time in each subsequent iteration.

Ultimately, it is up to the application developer to determine the cases where an application can recover from an error. If a reasonable recovery strategy can be used, we can improve the robustness of our application by avoiding an abnormal exit.

1.8 User Identification

User ID

The *user ID* from our entry in the password file is a numeric value that identifies us to the system. This user ID is assigned by the system administrator when our login name is assigned, and we cannot change it. The user ID is normally assigned to be unique for every user. We'll see how the kernel uses the user ID to check whether we have the appropriate permissions to perform certain operations.

We call the user whose user ID is 0 either *root* or the *superuser*. The entry in the password file normally has a login name of `root`, and we refer to the special privileges of this user as superuser privileges. As we'll see in Chapter 4, if a process has superuser privileges, most file permission checks are bypassed. Some operating system functions are restricted to the superuser. The superuser has free rein over the system.

Client versions of Mac OS X ship with the superuser account disabled; server versions ship with the account already enabled. Instructions are available on Apple's Web site describing how to enable it. See `http://support.apple.com/kb/HT1528`.

Group ID

Our entry in the password file also specifies our numeric *group ID*. This, too, is assigned by the system administrator when our login name is assigned. Typically, the password file contains multiple entries that specify the same group ID. Groups are normally used to collect users together into projects or departments. This allows the sharing of resources, such as files, among members of the same group. We'll see in Section 4.5 that we can set the permissions on a file so that all members of a group can access the file, whereas others outside the group cannot.

There is also a group file that maps group names into numeric group IDs. The group file is usually `/etc/group`.

The use of numeric user IDs and numeric group IDs for permissions is historical. With every file on disk, the file system stores both the user ID and the group ID of a file's owner. Storing both of these values requires only four bytes, assuming that each is stored as a two-byte integer. If the full ASCII login name and group name were used instead, additional disk space would be required. In addition, comparing strings during permission checks is more expensive than comparing integers.

Users, however, work better with names than with numbers, so the password file maintains the mapping between login names and user IDs, and the group file provides the mapping between group names and group IDs. The `ls -l` command, for example, prints the login name of the owner of a file, using the password file to map the numeric user ID into the corresponding login name.

Early UNIX systems used 16-bit integers to represent user and group IDs. Contemporary UNIX systems use 32-bit integers.

Example

The program in Figure 1.9 prints the user ID and the group ID.

```
#include "apue.h"

int
main(void)
{
    printf("uid = %d, gid = %d\n", getuid(), getgid());
    exit(0);
}
```

Figure 1.9 Print user ID and group ID

We call the functions `getuid` and `getgid` to return the user ID and the group ID. Running the program yields

```
$ ./a.out
uid = 205, gid = 105
```

□

Supplementary Group IDs

In addition to the group ID specified in the password file for a login name, most versions of the UNIX System allow a user to belong to other groups. This practice started with 4.2BSD, which allowed a user to belong to up to 16 additional groups. These *supplementary group IDs* are obtained at login time by reading the file `/etc/group` and finding the first 16 entries that list the user as a member. As we shall see in the next chapter, POSIX requires that a system support at least 8 supplementary groups per process, but most systems support at least 16.

1.9 Signals

Signals are a technique used to notify a process that some condition has occurred. For example, if a process divides by zero, the signal whose name is `SIGFPE` (floating-point exception) is sent to the process. The process has three choices for dealing with the signal.

1. Ignore the signal. This option isn't recommended for signals that denote a hardware exception, such as dividing by zero or referencing memory outside the address space of the process, as the results are undefined.
2. Let the default action occur. For a divide-by-zero condition, the default is to terminate the process.
3. Provide a function that is called when the signal occurs (this is called "catching" the signal). By providing a function of our own, we'll know when the signal occurs and we can handle it as we wish.

Many conditions generate signals. Two terminal keys, called the *interrupt key*—often the DELETE key or Control-C—and the *quit key*—often Control-backslash—are used to interrupt the currently running process. Another way to generate a signal is by calling the `kill` function. We can call this function from a process to send a signal to another process. Naturally, there are limitations: we have to be the owner of the other process (or the superuser) to be able to send it a signal.

Example

Recall the bare-bones shell example (Figure 1.7). If we invoke this program and press the interrupt key, the process terminates because the default action for this signal, named `SIGINT`, is to terminate the process. The process hasn't told the kernel to do anything other than the default with this signal, so the process terminates.

To catch this signal, the program needs to call the `signal` function, specifying the name of the function to call when the `SIGINT` signal is generated. The function is named `sig_int`; when it's called, it just prints a message and a new prompt. Adding

11 lines to the program in Figure 1.7 gives us the version in Figure 1.10. (The 11 new lines are indicated with a plus sign at the beginning of the line.)

```
  #include "apue.h"
  #include <sys/wait.h>

+ static void sig_int(int);       /* our signal-catching function */
+
  int
  main(void)
  {
      char    buf[MAXLINE];   /* from apue.h */
      pid_t   pid;
      int     status;

+     if (signal(SIGINT, sig_int) == SIG_ERR)
+         err_sys("signal error");
+
      printf("%% ");  /* print prompt (printf requires %% to print %) */
      while (fgets(buf, MAXLINE, stdin) != NULL) {
          if (buf[strlen(buf) - 1] == '\n')
              buf[strlen(buf) - 1] = 0; /* replace newline with null */

          if ((pid = fork()) < 0) {
              err_sys("fork error");
          } else if (pid == 0) {      /* child */
              execlp(buf, buf, (char *)0);
              err_ret("couldn't execute: %s", buf);
              exit(127);
          }

          /* parent */
          if ((pid = waitpid(pid, &status, 0)) < 0)
              err_sys("waitpid error");
          printf("%% ");
      }
      exit(0);
  }
+
+ void
+ sig_int(int signo)
+ {
+     printf("interrupt\n%% ");
+ }
```

Figure 1.10 Read commands from standard input and execute them

In Chapter 10, we'll take a long look at signals, as most nontrivial applications deal with them. □

1.10 Time Values

Historically, UNIX systems have maintained two different time values:

1. Calendar time. This value counts the number of seconds since the Epoch: 00:00:00 January 1, 1970, Coordinated Universal Time (UTC). (Older manuals refer to UTC as Greenwich Mean Time.) These time values are used to record the time when a file was last modified, for example.

   The primitive system data type `time_t` holds these time values.

2. Process time. This is also called CPU time and measures the central processor resources used by a process. Process time is measured in clock ticks, which have historically been 50, 60, or 100 ticks per second.

   The primitive system data type `clock_t` holds these time values. (We'll show how to obtain the number of clock ticks per second with the `sysconf` function in Section 2.5.4.)

When we measure the execution time of a process, as in Section 3.9, we'll see that the UNIX System maintains three values for a process:

- Clock time
- User CPU time
- System CPU time

The clock time, sometimes called *wall clock time*, is the amount of time the process takes to run, and its value depends on the number of other processes being run on the system. Whenever we report the clock time, the measurements are made with no other activities on the system.

The user CPU time is the CPU time attributed to user instructions. The system CPU time is the CPU time attributed to the kernel when it executes on behalf of the process. For example, whenever a process executes a system service, such as `read` or `write`, the time spent within the kernel performing that system service is charged to the process. The sum of user CPU time and system CPU time is often called the *CPU time*.

It is easy to measure the clock time, user time, and system time of any process: simply execute the `time`(1) command, with the argument to the `time` command being the command we want to measure. For example:

```
$ cd /usr/include
$ time -p grep _POSIX_SOURCE */*.h > /dev/null

real    0m0.81s
user    0m0.11s
sys     0m0.07s
```

The output format from the `time` command depends on the shell being used, because some shells don't run `/usr/bin/time`, but instead have a separate built-in function to measure the time it takes commands to run.

In Section 8.17, we'll see how to obtain these three times from a running process. The general topic of times and dates is covered in Section 6.10.

1.11 System Calls and Library Functions

All operating systems provide service points through which programs request services from the kernel. All implementations of the UNIX System provide a well-defined, limited number of entry points directly into the kernel called *system calls* (recall Figure 1.1). Version 7 of the Research UNIX System provided about 50 system calls, 4.4BSD provided about 110, and SVR4 had around 120. The exact number of system calls varies depending on the operating system version. More recent systems have seen incredible growth in the number of supported system calls. Linux 3.2.0 has 380 system calls and FreeBSD 8.0 has over 450.

The system call interface has always been documented in Section 2 of the *UNIX Programmer's Manual*. Its definition is in the C language, no matter which implementation technique is actually used on any given system to invoke a system call. This differs from many older operating systems, which traditionally defined the kernel entry points in the assembly language of the machine.

The technique used on UNIX systems is for each system call to have a function of the same name in the standard C library. The user process calls this function, using the standard C calling sequence. This function then invokes the appropriate kernel service, using whatever technique is required on the system. For example, the function may put one or more of the C arguments into general registers and then execute some machine instruction that generates a software interrupt in the kernel. For our purposes, we can consider the system calls to be C functions.

Section 3 of the *UNIX Programmer's Manual* defines the general-purpose library functions available to programmers. These functions aren't entry points into the kernel, although they may invoke one or more of the kernel's system calls. For example, the `printf` function may use the `write` system call to output a string, but the `strcpy` (copy a string) and `atoi` (convert ASCII to integer) functions don't involve the kernel at all.

From an implementor's point of view, the distinction between a system call and a library function is fundamental. From a user's perspective, however, the difference is not as critical. From our perspective in this text, both system calls and library functions appear as normal C functions. Both exist to provide services for application programs. We should realize, however, that we can replace the library functions, if desired, whereas the system calls usually cannot be replaced.

Consider the memory allocation function `malloc` as an example. There are many ways to do memory allocation and its associated garbage collection (best fit, first fit, and so on). No single technique is optimal for all programs. The UNIX system call that handles memory allocation, `sbrk`(2), is not a general-purpose memory manager. It increases or decreases the address space of the process by a specified number of bytes. How that space is managed is up to the process. The memory allocation function, `malloc`(3), implements one particular type of allocation. If we don't like its operation, we can define our own `malloc` function, which will probably use the `sbrk` system call. In fact, numerous software packages implement their own memory allocation algorithms with the `sbrk` system call. Figure 1.11 shows the relationship between the application, the `malloc` function, and the `sbrk` system call.

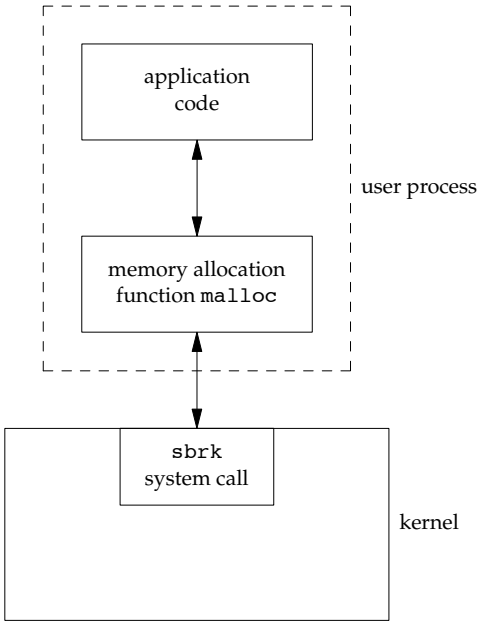


Figure 1.11 Separation of `malloc` function and `sbrk` system call

Here we have a clean separation of duties: the system call in the kernel allocates an additional chunk of space on behalf of the process. The `malloc` library function manages this space from user level.

Another example to illustrate the difference between a system call and a library function is the interface the UNIX System provides to determine the current time and date. Some operating systems provide one system call to return the time and another to return the date. Any special handling, such as the switch to or from daylight saving time, is handled by the kernel or requires human intervention. The UNIX System, in contrast, provides a single system call that returns the number of seconds since the Epoch: midnight, January 1, 1970, Coordinated Universal Time. Any interpretation of this value, such as converting it to a human-readable time and date using the local time zone, is left to the user process. The standard C library provides routines to handle most cases. These library routines handle such details as the various algorithms for daylight saving time.

An application can either make a system call or call a library routine. Also realize that many library routines invoke a system call. This is shown in Figure 1.12.

Another difference between system calls and library functions is that system calls usually provide a minimal interface, whereas library functions often provide more elaborate functionality. We've seen this already in the difference between the `sbrk` system call and the `malloc` library function. We'll see this difference again later, when

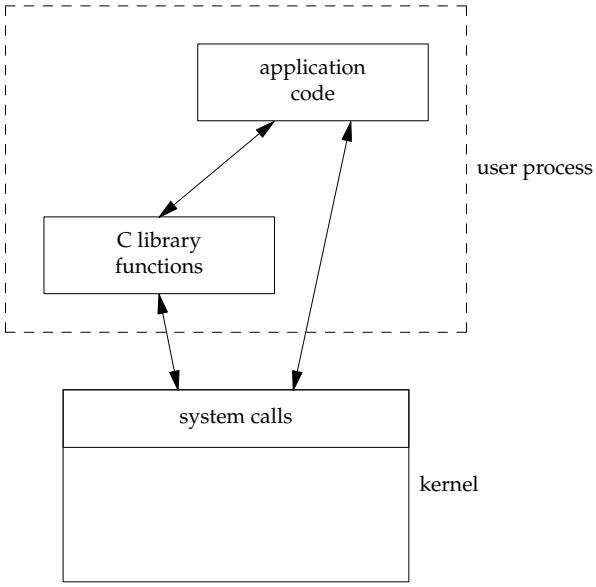


Figure 1.12 Difference between C library functions and system calls

we compare the unbuffered I/O functions (Chapter 3) and the standard I/O functions (Chapter 5).

The process control system calls (`fork`, `exec`, and `waitpid`) are usually invoked by the user's application code directly. (Recall the bare-bones shell in Figure 1.7.) But some library routines exist to simplify certain common cases: the `system` and `popen` library routines, for example. In Section 8.13, we'll show an implementation of the `system` function that invokes the basic process control system calls. We'll enhance this example in Section 10.18 to handle signals correctly.

To define the interface to the UNIX System that most programmers use, we have to describe both the system calls and some of the library functions. If we described only the `sbrk` system call, for example, we would skip the more programmer-friendly `malloc` library function that many applications use. In this text, we'll use the term *function* to refer to both system calls and library functions, except when the distinction is necessary.

1.12 Summary

This chapter has provided a short tour of the UNIX System. We've described some of the fundamental terms that we'll encounter over and over again. We've seen numerous small examples of UNIX programs to give us a feel for what the remainder of the text talks about.

The next chapter is about standardization of the UNIX System and the effect of work in this area on current systems. Standards, particularly the ISO C standard and the POSIX.1 standard, will affect the rest of the text.

Exercises

1.1 Verify on your system that the directories dot and dot-dot are not the same, except in the root directory.

1.2 In the output from the program in Figure 1.6, what happened to the processes with process IDs 852 and 853?

1.3 In Section 1.7, the argument to `perror` is defined with the ISO C attribute `const`, whereas the integer argument to `strerror` isn't defined with this attribute. Why?

1.4 If the calendar time is stored as a signed 32-bit integer, in which year will it overflow? How can we extend the overflow point? Are these strategies compatible with existing applications?

1.5 If the process time is stored as a signed 32-bit integer, and if the system counts 100 ticks per second, after how many days will the value overflow?

2

UNIX Standardization and Implementations

2.1 Introduction

Much work has gone into standardizing the UNIX programming environment and the C programming language. Although applications have always been quite portable across different versions of the UNIX operating system, the proliferation of versions and differences during the 1980s led many large users, such as the U.S. government, to call for standardization.

In this chapter we first look at the various standardization efforts that have been under way over the past two and a half decades. We then discuss the effects of these UNIX programming standards on the operating system implementations that are described in this book. An important part of all the standardization efforts is the specification of various limits that each implementation must define, so we look at these limits and the various ways to determine their values.

2.2 UNIX Standardization

2.2.1 ISO C

In late 1989, ANSI Standard X3.159-1989 for the C programming language was approved. This standard was also adopted as International Standard ISO/IEC 9899:1990. ANSI is the American National Standards Institute, the U.S. member in the International Organization for Standardization (ISO). IEC stands for the International Electrotechnical Commission.

The C standard is now maintained and developed by the ISO/IEC international standardization working group for the C programming language, known as ISO/IEC JTC1/SC22/WG14, or WG14 for short. The intent of the ISO C standard is to provide portability of conforming C programs to a wide variety of operating systems, not only the UNIX System. This standard defines not only the syntax and semantics of the programming language but also a standard library [Chapter 7 of ISO 1999; Plauger 1992; Appendix B of Kernighan and Ritchie 1988]. This library is important because all contemporary UNIX systems, such as the ones described in this book, provide the library routines that are specified in the C standard.

In 1999, the ISO C standard was updated and approved as ISO/IEC 9899:1999, largely to improve support for applications that perform numerical processing. The changes don't affect the POSIX interfaces described in this book, except for the addition of the `restrict` keyword to some of the function prototypes. This keyword is used to tell the compiler which pointer references can be optimized, by indicating that the object to which the pointer refers is accessed in the function only via that pointer.

Since 1999, three technical corrigenda have been published to correct errors in the ISO C standard—one in 2001, one in 2004, and one in 2007. As with most standards, there is a delay between the standard's approval and the modification of software to conform to it. As each vendor's compilation systems evolve, they add more support for the latest version of the ISO C standard.

> A summary of the current level of conformance of `gcc` to the 1999 version of the ISO C standard is available at `http://gcc.gnu.org/c99status.html`. Although the C standard was updated in 2011, we deal only with the 1999 version in this text, because the other standards haven't yet caught up with the relevant changes.

The ISO C library can be divided into 24 areas, based on the headers defined by the standard (see Figure 2.1). The POSIX.1 standard includes these headers, as well as others. As Figure 2.1 shows, all of these headers are supported by the four implementations (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10) that are described later in this chapter.

> The ISO C headers depend on which version of the C compiler is used with the operating system. FreeBSD 8.0 ships with version 4.2.1 of `gcc`, Solaris 10 ships with version 3.4.3 of `gcc` (in addition to its own C compiler in Sun Studio), Ubuntu 12.04 (Linux 3.2.0) ships with version 4.6.3 of `gcc`, and Mac OS X 10.6.8 ships with both versions 4.0.1 and 4.2.1 of `gcc`.

2.2.2 IEEE POSIX

POSIX is a family of standards initially developed by the IEEE (Institute of Electrical and Electronics Engineers). POSIX stands for Portable Operating System Interface. It originally referred only to the IEEE Standard 1003.1-1988—the operating system interface—but was later extended to include many of the standards and draft standards with the 1003 designation, including the shell and utilities (1003.2).

Of specific interest to this book is the 1003.1 operating system interface standard, whose goal is to promote the portability of applications among various UNIX System environments. This standard defines the services that an operating system must

Header	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
<assert.h>	•	•	•	•	verify program assertion
<complex.h>	•	•	•	•	complex arithmetic support
<ctype.h>	•	•	•	•	character classification and mapping support
<errno.h>	•	•	•	•	error codes (Section 1.7)
<fenv.h>	•	•	•	•	floating-point environment
<float.h>	•	•	•	•	floating-point constants and characteristics
<inttypes.h>	•	•	•	•	integer type format conversion
<iso646.h>	•	•	•	•	macros for assignment, relational, and unary operators
<limits.h>	•	•	•	•	implementation constants (Section 2.5)
<locale.h>	•	•	•	•	locale categories and related definitions
<math.h>	•	•	•	•	mathematical function and type declarations and constants
<setjmp.h>	•	•	•	•	nonlocal goto (Section 7.10)
<signal.h>	•	•	•	•	signals (Chapter 10)
<stdarg.h>	•	•	•	•	variable argument lists
<stdbool.h>	•	•	•	•	Boolean type and values
<stddef.h>	•	•	•	•	standard definitions
<stdint.h>	•	•	•	•	integer types
<stdio.h>	•	•	•	•	standard I/O library (Chapter 5)
<stdlib.h>	•	•	•	•	utility functions
<string.h>	•	•	•	•	string operations
<tgmath.h>	•	•	•	•	type-generic math macros
<time.h>	•	•	•	•	time and date (Section 6.10)
<wchar.h>	•	•	•	•	extended multibyte and wide character support
<wctype.h>	•	•	•	•	wide character classification and mapping support

Figure 2.1 Headers defined by the ISO C standard

provide if it is to be "POSIX compliant," and has been adopted by most computer vendors. Although the 1003.1 standard is based on the UNIX operating system, the standard is not restricted to UNIX and UNIX-like systems. Indeed, some vendors supplying proprietary operating systems claim that these systems have been made POSIX compliant, while still leaving all their proprietary features in place.

Because the 1003.1 standard specifies an *interface* and not an *implementation*, no distinction is made between system calls and library functions. All the routines in the standard are called *functions*.

Standards are continually evolving, and the 1003.1 standard is no exception. The 1988 version, IEEE Standard 1003.1-1988, was modified and submitted to the International Organization for Standardization. No new interfaces or features were added, but the text was revised. The resulting document was published as IEEE Standard 1003.1-1990 [IEEE 1990]. This is also International Standard ISO/IEC 9945-1:1990. This standard was commonly referred to as *POSIX.1*, a term which we'll use in this text to refer to the different versions of the standard.

The IEEE 1003.1 working group continued to make changes to the standard. In 1996, a revised version of the IEEE 1003.1 standard was published. It included the 1003.1-1990 standard, the 1003.1b-1993 real-time extensions standard, and the interfaces for multithreaded programming, called *pthreads* for POSIX threads. This version of the

standard was also published as International Standard ISO/IEC 9945-1:1996. More real-time interfaces were added in 1999 with the publication of IEEE Standard 1003.1d-1999. A year later, IEEE Standard 1003.1j-2000 was published, including even more real-time interfaces, and IEEE Standard 1003.1q-2000 was published, adding event-tracing extensions to the standard.

The 2001 version of 1003.1 departed from the prior versions in that it combined several 1003.1 amendments, the 1003.2 standard, and portions of the Single UNIX Specification (SUS), Version 2 (more on this later). The resulting standard, IEEE Standard 1003.1-2001, included the following other standards:

- ISO/IEC 9945-1 (IEEE Standard 1003.1-1996), which includes
    - IEEE Standard 1003.1-1990
    - IEEE Standard 1003.1b-1993 (real-time extensions)
    - IEEE Standard 1003.1c-1995 (pthreads)
    - IEEE Standard 1003.1i-1995 (real-time technical corrigenda)
- IEEE P1003.1a draft standard (system interface amendment)
- IEEE Standard 1003.1d-1999 (advanced real-time extensions)
- IEEE Standard 1003.1j-2000 (more advanced real-time extensions)
- IEEE Standard 1003.1q-2000 (tracing)
- Parts of IEEE Standard 1003.1g-2000 (protocol-independent interfaces)
- ISO/IEC 9945-2 (IEEE Standard 1003.2-1993)
- IEEE P1003.2b draft standard (shell and utilities amendment)
- IEEE Standard 1003.2d-1994 (batch extensions)
- The Base Specifications of the Single UNIX Specification, version 2, which include
    - System Interface Definitions, Issue 5
    - Commands and Utilities, Issue 5
    - System Interfaces and Headers, Issue 5
- Open Group Technical Standard, Networking Services, Issue 5.2
- ISO/IEC 9899:1999, Programming Languages–C

In 2004, the POSIX.1 specification was updated with technical corrections; more comprehensive changes were made in 2008 and released as Issue 7 of the Base Specifications. ISO approved this version at the end of 2008 and published it in 2009 as International Standard ISO/IEC 9945:2009. It is based on several other standards:

- IEEE Standard 1003.1, 2004 Edition
- Open Group Technical Standard, 2006, Extended API Set, Parts 1−4
- ISO/IEC 9899:1999, including corrigenda

Figure 2.2, Figure 2.3, and Figure 2.4 summarize the required and optional headers as specified by POSIX.1. Because POSIX.1 includes the ISO C standard library functions, it also requires the headers listed in Figure 2.1. All four figures summarize which headers are included in the implementations discussed in this book.

In this text we describe the 2008 edition of POSIX.1. Its interfaces are divided into required ones and optional ones. The optional interfaces are further divided into 40 sections, based on functionality. The sections containing nonobsolete programming interfaces are summarized in Figure 2.5 with their respective option codes. Option codes are two- to three-character abbreviations that identify the interfaces that belong to

Header	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
<aio.h>	•	•	•	•	asynchronous I/O
<cpio.h>	•	•	•	•	cpio archive values
<dirent.h>	•	•	•	•	directory entries (Section 4.22)
<dlfcn.h>	•	•	•	•	dynamic linking
<fcntl.h>	•	•	•	•	file control (Section 3.14)
<fnmatch.h>	•	•	•	•	filename-matching types
<glob.h>	•	•	•	•	pathname pattern-matching and generation
<grp.h>	•	•	•	•	group file (Section 6.4)
<iconv.h>	•	•	•	•	codeset conversion utility
<langinfo.h>	•	•	•	•	language information constants
<monetary.h>	•	•	•	•	monetary types and functions
<netdb.h>	•	•	•	•	network database operations
<nl_types.h>	•	•	•	•	message catalogs
<poll.h>	•	•	•	•	poll function (Section 14.4.2)
<pthread.h>	•	•	•	•	threads (Chapters 11 and 12)
<pwd.h>	•	•	•	•	password file (Section 6.2)
<regex.h>	•	•	•	•	regular expressions
<sched.h>	•	•	•	•	execution scheduling
<semaphore.h>	•	•	•	•	semaphores
<strings.h>	•	•	•	•	string operations
<tar.h>	•	•	•	•	tar archive values
<termios.h>	•	•	•	•	terminal I/O (Chapter 18)
<unistd.h>	•	•	•	•	symbolic constants
<wordexp.h>	•	•	•	•	word-expansion definitions
<arpa/inet.h>	•	•	•	•	Internet definitions (Chapter 16)
<net/if.h>	•	•	•	•	socket local interfaces (Chapter 16)
<netinet/in.h>	•	•	•	•	Internet address family (Section 16.3)
<netinet/tcp.h>	•	•	•	•	Transmission Control Protocol definitions
<sys/mman.h>	•	•	•	•	memory management declarations
<sys/select.h>	•	•	•	•	select function (Section 14.4.1)
<sys/socket.h>	•	•	•	•	sockets interface (Chapter 16)
<sys/stat.h>	•	•	•	•	file status (Chapter 4)
<sys/statvfs.h>	•	•	•	•	file system information
<sys/times.h>	•	•	•	•	process times (Section 8.17)
<sys/types.h>	•	•	•	•	primitive system data types (Section 2.8)
<sys/un.h>	•	•	•	•	UNIX domain socket definitions (Section 17.2)
<sys/utsname.h>	•	•	•	•	system name (Section 6.9)
<sys/wait.h>	•	•	•	•	process control (Section 8.6)

Figure 2.2 Required headers defined by the POSIX standard

each functional area and highlight text describing aspects of the standard that depend on the support of a particular option. Many options deal with real-time extensions.

POSIX.1 does not include the notion of a superuser. Instead, certain operations require "appropriate privileges," although POSIX.1 leaves the definition of this term up to the implementation. UNIX systems that conform to the Department of Defense's security guidelines have many levels of security. In this text, however, we use the traditional terminology and refer to operations that require superuser privilege.

Header	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
<fmtmsg.h>	•	•	•	•	message display structures
<ftw.h>	•	•	•	•	file tree walking (Section 4.22)
<libgen.h>	•	•	•	•	pathname management functions
<ndbm.h>	•		•	•	database operations
<search.h>	•	•	•	•	search tables
<syslog.h>	•	•	•	•	system error logging (Section 13.4)
<utmpx.h>		•	•	•	user accounting database
<sys/ipc.h>	•	•	•	•	IPC (Section 15.6)
<sys/msg.h>	•	•	•	•	XSI message queues (Section 15.7)
<sys/resource.h>	•	•	•	•	resource operations (Section 7.11)
<sys/sem.h>	•	•	•	•	XSI semaphores (Section 15.8)
<sys/shm.h>	•	•	•	•	XSI shared memory (Section 15.9)
<sys/time.h>	•	•	•	•	time types
<sys/uio.h>	•	•	•	•	vector I/O operations (Section 14.6)

Figure 2.3 XSI option headers defined by the POSIX standard

After more than twenty years of work, the standards are mature and stable. The POSIX.1 standard is maintained by an open working group known as the Austin Group (`http://www.opengroup.org/austin`). To ensure that they are still relevant, the standards need to be either updated or reaffirmed every so often.

2.2.3 The Single UNIX Specification

The Single UNIX Specification, a superset of the POSIX.1 standard, specifies additional interfaces that extend the functionality provided by the POSIX.1 specification. POSIX.1 is equivalent to the Base Specifications portion of the Single UNIX Specification.

The *X/Open System Interfaces* (XSI) option in POSIX.1 describes optional interfaces and defines which optional portions of POSIX.1 must be supported for an implementation to be deemed *XSI conforming*. These include file synchronization, thread stack address and size attributes, thread process-shared synchronization, and the `_XOPEN_UNIX` symbolic constant (marked "SUS mandatory" in Figure 2.5). Only XSI-conforming implementations can be called UNIX systems.

> The Open Group owns the UNIX trademark and uses the Single UNIX Specification to define the interfaces an implementation must support to call itself a UNIX system. Vendors must file conformance statements, pass test suites to verify conformance, and license the right to use the UNIX trademark.

Header	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
<mqueue.h>	•	•		•	message queues
<spawn.h>	•	•	•	•	real-time spawn interface

Figure 2.4 Optional headers defined by the POSIX standard

Code	SUS mandatory	Symbolic constant	Description
ADV		_POSIX_ADVISORY_INFO	advisory information (real-time)
CPT		_POSIX_CPUTIME	process CPU time clocks (real-time)
FSC	•	_POSIX_FSYNC	file synchronization
IP6		_POSIX_IPV6	IPv6 interfaces
ML		_POSIX_MEMLOCK	process memory locking (real-time)
MLR		_POSIX_MEMLOCK_RANGE	memory range locking (real-time)
MON		_POSIX_MONOTONIC_CLOCK	monotonic clock (real-time)
MSG		_POSIX_MESSAGE_PASSING	message passing (real-time)
MX		__STDC_IEC_559__	IEC 60559 floating-point option
PIO		_POSIX_PRIORITIZED_IO	prioritized input and output
PS		_POSIX_PRIORITY_SCHEDULING	process scheduling (real-time)
RPI		_POSIX_THREAD_ROBUST_PRIO_INHERIT	robust mutex priority inheritance (real-time)
RPP		_POSIX_THREAD_ROBUST_PRIO_PROTECT	robust mutex priority protection (real-time)
RS		_POSIX_RAW_SOCKETS	raw sockets
SHM		_POSIX_SHARED_MEMORY_OBJECTS	shared memory objects (real-time)
SIO		_POSIX_SYNCHRONIZED_IO	synchronized input and output (real-time)
SPN		_POSIX_SPAWN	spawn (real-time)
SS		_POSIX_SPORADIC_SERVER	process sporadic server (real-time)
TCT		_POSIX_THREAD_CPUTIME	thread CPU time clocks (real-time)
TPI		_POSIX_THREAD_PRIO_INHERIT	nonrobust mutex priority inheritance (real-time)
TPP		_POSIX_THREAD_PRIO_PROTECT	nonrobust mutex priority protection (real-time)
TPS		_POSIX_THREAD_PRIORITY_SCHEDULING	thread execution scheduling (real-time)
TSA	•	_POSIX_THREAD_ATTR_STACKADDR	thread stack address attribute
TSH	•	_POSIX_THREAD_PROCESS_SHARED	thread process-shared synchronization
TSP		_POSIX_THREAD_SPORADIC_SERVER	thread sporadic server (real-time)
TSS	•	_POSIX_THREAD_ATTR_STACKSIZE	thread stack size address
TYM		_POSIX_TYPED_MEMORY_OBJECTS	typed memory objects (real-time)
XSI	•	_XOPEN_UNIX	X/Open interfaces

Figure 2.5 POSIX.1 optional interface groups and codes

Several of the interfaces that are optional for XSI-conforming systems are divided into *option groups* based on common functionality, as follows:

- Encryption: denoted by the `_XOPEN_CRYPT` symbolic constant
- Real-time: denoted by the `_XOPEN_REALTIME` symbolic constant
- Advanced real-time
- Real-time threads: denoted by `_XOPEN_REALTIME_THREADS`
- Advanced real-time threads

The Single UNIX Specification is a publication of The Open Group, which was formed in 1996 as a merger of X/Open and the Open Software Foundation (OSF), both industry consortia. X/Open used to publish the *X/Open Portability Guide*, which adopted specific standards and filled in the gaps where functionality was missing. The goal of these guides was to improve application portability beyond what was possible by merely conforming to published standards.

The first version of the Single UNIX Specification was published by X/Open in 1994. It was also known as "Spec 1170," because it contained roughly 1,170 interfaces. It grew out of the Common Open Software Environment (COSE) initiative, whose goal was to improve application portability across all implementations of the UNIX operating system. The COSE group—Sun, IBM, HP, Novell/USL, and OSF—went further than endorsing standards by including interfaces used by common commercial applications. The resulting 1,170 interfaces were selected from these applications, and also included the X/Open Common Application Environment (CAE), Issue 4 (known as "XPG4" as a historical reference to its predecessor, the X/Open Portability Guide), the System V Interface Definition (SVID), Edition 3, Level 1 interfaces, and the OSF Application Environment Specification (AES) Full Use interfaces.

The second version of the Single UNIX Specification was published by The Open Group in 1997. The new version added support for threads, real-time interfaces, 64-bit processing, large files, and enhanced multibyte character processing.

The third version of the Single UNIX Specification (SUSv3) was published by The Open Group in 2001. The Base Specifications of SUSv3 are the same as IEEE Standard 1003.1-2001 and are divided into four sections: Base Definitions, System Interfaces, Shell and Utilities, and Rationale. SUSv3 also includes X/Open Curses Issue 4, Version 2, but this specification is not part of POSIX.1.

In 2002, ISO approved the IEEE Standard 1003.1-2001 as International Standard ISO/IEC 9945:2002. The Open Group updated the 1003.1 standard again in 2003 to include technical corrections, and ISO approved this as International Standard ISO/IEC 9945:2003. In April 2004, The Open Group published the Single UNIX Specification, Version 3, 2004 Edition. It merged more technical corrections into the main text of the standard.

In 2008, the Single UNIX Specification was updated, including corrections and new interfaces, removing obsolete interfaces, and marking other interfaces as being obsolescent in preparation for future removal. Additionally, some previously optional interfaces were promoted to nonoptional status, including asynchronous I/O, barriers, clock selection, memory-mapped files, memory protection, reader–writer locks, real-time signals, POSIX semaphores, spin locks, thread-safe functions, threads, timeouts, and timers. The resulting standard is known as Issue 7 of the Base Specifications, and is the same as POSIX.1-2008. The Open Group bundled this version with an updated X/Open Curses specification and released them as version 4 of the Single UNIX Specification in 2010. We'll refer to this as SUSv4.

2.2.4 FIPS

FIPS stands for Federal Information Processing Standard. It was published by the U.S. government, which used it for the procurement of computer systems. FIPS 151-1 (April 1989) was based on the IEEE Standard 1003.1-1988 and a draft of the ANSI C standard. This was followed by FIPS 151-2 (May 1993), which was based on the IEEE Standard 1003.1-1990. FIPS 151-2 required some features that POSIX.1 listed as optional. All these options were included as mandatory in POSIX.1-2001.

The effect of the POSIX.1 FIPS was to require any vendor that wished to sell POSIX.1-compliant computer systems to the U.S. government to support some of the optional features of POSIX.1. The POSIX.1 FIPS has since been withdrawn, so we won't consider it further in this text.

2.3 UNIX System Implementations

The previous section described ISO C, IEEE POSIX, and the Single UNIX Specification—three standards originally created by independent organizations. Standards, however, are interface specifications. How do these standards relate to the real world? These standards are taken by vendors and turned into actual implementations. In this book, we are interested in both these standards and their implementation.

Section 1.1 of McKusick et al. [1996] gives a detailed history (and a nice picture) of the UNIX System family tree. Everything starts from the Sixth Edition (1976) and Seventh Edition (1979) of the UNIX Time-Sharing System on the PDP-11 (usually called Version 6 and Version 7, respectively). These were the first releases widely distributed outside of Bell Laboratories. Three branches of the tree evolved.

1. One at AT&T that led to System III and System V, the so-called commercial versions of the UNIX System.
2. One at the University of California at Berkeley that led to the 4.xBSD implementations.
3. The research version of the UNIX System, developed at the Computing Science Research Center of AT&T Bell Laboratories, that led to the UNIX Time-Sharing System 8th Edition, 9th Edition, and ended with the 10th Edition in 1990.

2.3.1 UNIX System V Release 4

UNIX System V Release 4 (SVR4) was a product of AT&T's UNIX System Laboratories (USL, formerly AT&T's UNIX Software Operation). SVR4 merged functionality from AT&T UNIX System V Release 3.2 (SVR3.2), the SunOS operating system from Sun Microsystems, the 4.3BSD release from the University of California, and the Xenix system from Microsoft into one coherent operating system. (Xenix was originally developed from Version 7, with many features later taken from System V.) The SVR4 source code was released in late 1989, with the first end-user copies becoming available during 1990. SVR4 conformed to both the POSIX 1003.1 standard and the X/Open Portability Guide, Issue 3 (XPG3).

AT&T also published the System V Interface Definition (SVID) [AT&T 1989]. Issue 3 of the SVID specified the functionality that an operating system must offer to qualify as a conforming implementation of UNIX System V Release 4. As with POSIX.1, the SVID specified an interface, not an implementation. No distinction was made in the SVID between system calls and library functions. The reference manual for an actual implementation of SVR4 must be consulted to see this distinction [AT&T 1990e].

2.3.2 4.4BSD

The Berkeley Software Distribution (BSD) releases were produced and distributed by the Computer Systems Research Group (CSRG) at the University of California at Berkeley; 4.2BSD was released in 1983 and 4.3BSD in 1986. Both of these releases ran on the VAX minicomputer. The next release, 4.3BSD Tahoe in 1988, also ran on a particular minicomputer called the Tahoe. (The book by Leffler et al. [1989] describes the 4.3BSD Tahoe release.) This was followed in 1990 with the 4.3BSD Reno release; 4.3BSD Reno supported many of the POSIX.1 features.

The original BSD systems contained proprietary AT&T source code and were covered by AT&T licenses. To obtain the source code to the BSD system you had to have a UNIX source license from AT&T. This changed as more and more of the AT&T source code was replaced over the years with non-AT&T source code and as many of the new features added to the Berkeley system were derived from non-AT&T sources.

In 1989, Berkeley identified much of the non-AT&T source code in the 4.3BSD Tahoe release and made it publicly available as the BSD Networking Software, Release 1.0. Release 2.0 of the BSD Networking Software followed in 1991, which was derived from the 4.3BSD Reno release. The intent was that most, if not all, of the 4.4BSD system would be free of AT&T license restrictions, thus making the source code available to all.

4.4BSD-Lite was intended to be the final release from the CSRG. Its introduction was delayed, however, because of legal battles with USL. Once the legal differences were resolved, 4.4BSD-Lite was released in 1994, fully unencumbered, so no UNIX source license was needed to receive it. The CSRG followed this with a bug-fix release in 1995. This release, 4.4BSD-Lite, release 2, was the final version of BSD from the CSRG. (This version of BSD is described in the book by McKusick et al. [1996].)

The UNIX system development done at Berkeley started with PDP-11s, then moved to the VAX minicomputer, and then to other so-called workstations. During the early 1990s, support was provided to Berkeley for the popular 80386-based personal computers, leading to what is called 386BSD. This support was provided by Bill Jolitz and was documented in a series of monthly articles in *Dr. Dobb's Journal* throughout 1991. Much of this code appeared in the BSD Networking Software, Release 2.0.

2.3.3 FreeBSD

FreeBSD is based on the 4.4BSD-Lite operating system. The FreeBSD project was formed to carry on the BSD line after the Computing Science Research Group at the University of California at Berkeley decided to end its work on the BSD versions of the UNIX operating system, and the 386BSD project seemed to be neglected for too long.

All software produced by the FreeBSD project is freely available in both binary and source forms. The FreeBSD 8.0 operating system was one of the four operating systems used to test the examples in this book.

> Several other BSD-based free operating systems are available. The NetBSD project (`http://www.netbsd.org`) is similar to the FreeBSD project, but emphasizes portability between hardware platforms. The OpenBSD project (`http://www.openbsd.org`) is similar to FreeBSD but places a greater emphasis on security.

2.3.4 Linux

Linux is an operating system that provides a rich programming environment similar to that of a UNIX System; it is freely available under the GNU Public License. The popularity of Linux is somewhat of a phenomenon in the computer industry. Linux is distinguished by often being the first operating system to support new hardware.

Linux was created in 1991 by Linus Torvalds as a replacement for MINIX. A grass-roots effort then sprang up, whereby many developers across the world volunteered their time to use and enhance it.

The Ubuntu 12.04 distribution of Linux was one of the operating systems used to test the examples in this book. That distribution uses the 3.2.0 version of the Linux operating system kernel.

2.3.5 Mac OS X

Mac OS X is based on entirely different technology than prior versions. The core operating system is called "Darwin," and is based on a combination of the Mach kernel (Accetta et al. [1986]), the FreeBSD operating system, and an object-oriented framework for drivers and other kernel extensions. As of version 10.5, the Intel port of Mac OS X has been certified to be a UNIX system. (For more information on UNIX certification, see `http://www.opengroup.org/certification/idx/unix.html`.)

Mac OS X version 10.6.8 (Darwin 10.8.0) was used as one of the operating systems to test the examples in this book.

2.3.6 Solaris

Solaris is the version of the UNIX System developed by Sun Microsystems (now Oracle). Solaris is based on System V Release 4, but includes more than fifteen years of enhancements from the engineers at Sun Microsystems. It is arguably the only commercially successful SVR4 descendant, and is formally certified to be a UNIX system.

In 2005, Sun Microsystems released most of the Solaris operating system source code to the public as part of the OpenSolaris open source operating system in an attempt to build an external developer community around Solaris.

The Solaris 10 UNIX system was one of the operating systems used to test the examples in this book.

2.3.7 Other UNIX Systems

Other versions of the UNIX system that have been certified in the past include

- AIX, IBM's version of the UNIX System
- HP-UX, Hewlett-Packard's version of the UNIX System
- IRIX, the UNIX System version shipped by Silicon Graphics
- UnixWare, the UNIX System descended from SVR4 sold by SCO

2.4 Relationship of Standards and Implementations

The standards that we've mentioned define a subset of any actual system. The focus of this book is on four real systems: FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10. Although only Mac OS X and Solaris can call themselves UNIX systems, all four provide a similar programming environment. Because all four are POSIX compliant to varying degrees, we will also concentrate on the features required by the POSIX.1 standard, noting any differences between POSIX and the actual implementations of these four systems. Those features and routines that are specific to only a particular implementation are clearly marked. We'll also note any features that are required on UNIX systems but are optional on other POSIX-conforming systems.

Be aware that the implementations provide backward compatibility for features in earlier releases, such as SVR3.2 and 4.3BSD. For example, Solaris supports both the POSIX.1 specification for nonblocking I/O (`O_NONBLOCK`) and the traditional System V method (`O_NDELAY`). In this text, we'll use only the POSIX.1 feature, although we'll mention the nonstandard feature that it replaces. Similarly, both SVR3.2 and 4.3BSD provided reliable signals in a way that differs from the POSIX.1 standard. In Chapter 10 we describe only the POSIX.1 signal mechanism.

2.5 Limits

The implementations define many magic numbers and constants. Many of these have been hard coded into programs or were determined using ad hoc techniques. With the various standardization efforts that we've described, more portable methods are now provided to determine these magic numbers and implementation-defined limits, greatly improving the portability of software written for the UNIX environment.

Two types of limits are needed:

1. Compile-time limits (e.g., what's the largest value of a short integer?)
2. Runtime limits (e.g., how many bytes in a filename?)

Compile-time limits can be defined in headers that any program can include at compile time. But runtime limits require the process to call a function to obtain the limit's value.

Additionally, some limits can be fixed on a given implementation—and could therefore be defined statically in a header—yet vary on another implementation and would require a runtime function call. An example of this type of limit is the maximum number of bytes in a filename. Before SVR4, System V historically allowed only 14 bytes in a filename, whereas BSD-derived systems increased this number to 255. Most UNIX System implementations these days support multiple file system types, and each type has its own limit. This is the case of a runtime limit that depends on where in the file system the file in question is located. A filename in the root file system, for example, could have a 14-byte limit, whereas a filename in another file system could have a 255-byte limit.

To solve these problems, three types of limits are provided:

1. Compile-time limits (headers)

2. Runtime limits not associated with a file or directory (the `sysconf` function)
3. Runtime limits that are associated with a file or a directory (the `pathconf` and `fpathconf` functions)

To further confuse things, if a particular runtime limit does not vary on a given system, it can be defined statically in a header. If it is not defined in a header, however, the application must call one of the three `conf` functions (which we describe shortly) to determine its value at runtime.

Name	Description	Minimum acceptable value	Typical value
`CHAR_BIT`	bits in a `char`	8	8
`CHAR_MAX`	max value of `char`	(see later)	127
`CHAR_MIN`	min value of `char`	(see later)	–128
`SCHAR_MAX`	max value of `signed char`	127	127
`SCHAR_MIN`	min value of `signed char`	–127	–128
`UCHAR_MAX`	max value of `unsigned char`	255	255
`INT_MAX`	max value of `int`	32,767	2,147,483,647
`INT_MIN`	min value of `int`	–32,767	–2,147,483,648
`UINT_MAX`	max value of `unsigned int`	65,535	4,294,967,295
`SHRT_MAX`	max value of `short`	32,767	32,767
`SHRT_MIN`	min value of `short`	–32,767	–32,768
`USHRT_MAX`	max value of `unsigned short`	65,535	65,535
`LONG_MAX`	max value of `long`	2,147,483,647	2,147,483,647
`LONG_MIN`	min value of `long`	–2,147,483,647	–2,147,483,648
`ULONG_MAX`	max value of `unsigned long`	4,294,967,295	4,294,967,295
`LLONG_MAX`	max value of `long long`	9,223,372,036,854,775,807	9,223,372,036,854,775,807
`LLONG_MIN`	min value of `long long`	–9,223,372,036,854,775,807	–9,223,372,036,854,775,808
`ULLONG_MAX`	max value of `unsigned long long`	18,446,744,073,709,551,615	18,446,744,073,709,551,615
`MB_LEN_MAX`	max number of bytes in a multibyte character constant	1	6

Figure 2.6 Sizes of integral values from `<limits.h>`

2.5.1 ISO C Limits

All of the compile-time limits defined by ISO C are defined in the file `<limits.h>` (see Figure 2.6). These constants don't change in a given system. The third column in Figure 2.6 shows the minimum acceptable values from the ISO C standard. This allows for a system with 16-bit integers using one's-complement arithmetic. The fourth column shows the values from a Linux system with 32-bit integers using two's-complement arithmetic. Note that none of the unsigned data types has a minimum value, as this value must be 0 for an unsigned data type. On a 64-bit system, the values for `long` integer maximums match the maximum values for `long long` integers.

One difference that we will encounter is whether a system provides signed or unsigned character values. From the fourth column in Figure 2.6, we see that this

particular system uses signed characters. We see that CHAR_MIN equals SCHAR_MIN and that CHAR_MAX equals SCHAR_MAX. If the system uses unsigned characters, we would have CHAR_MIN equal to 0 and CHAR_MAX equal to UCHAR_MAX.

The floating-point data types in the header <float.h> have a similar set of definitions. Anyone doing serious floating-point work should examine this file.

Although the ISO C standard specifies minimum acceptable values for integral data types, POSIX.1 makes extensions to the C standard. To conform to POSIX.1, an implementation must support a minimum value of 2,147,483,647 for INT_MAX, –2,147,483,647 for INT_MIN, and 4,294,967,295 for UINT_MAX. Because POSIX.1 requires implementations to support an 8-bit char, CHAR_BIT must be 8, SCHAR_MIN must be –128, SCHAR_MAX must be 127, and UCHAR_MAX must be 255.

Another ISO C constant that we'll encounter is FOPEN_MAX, the minimum number of standard I/O streams that the implementation guarantees can be open at once. This constant is found in the <stdio.h> header, and its minimum value is 8. The POSIX.1 value STREAM_MAX, if defined, must have the same value as FOPEN_MAX.

ISO C also defines the constant TMP_MAX in <stdio.h>. It is the maximum number of unique filenames generated by the tmpnam function. We'll have more to say about this constant in Section 5.13.

Although ISO C defines the constant FILENAME_MAX, we avoid using it, because POSIX.1 provides better alternatives (NAME_MAX and PATH_MAX). We'll see these constants shortly.

Figure 2.7 shows the values of FILENAME_MAX, FOPEN_MAX, and TMP_MAX on the four platforms we discuss in this book.

Limit	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
FOPEN_MAX	20	16	20	20
TMP_MAX	308,915,776	238,328	308,915,776	17,576
FILENAME_MAX	1024	4096	1024	1024

Figure 2.7 ISO limits on various platforms

2.5.2 POSIX Limits

POSIX.1 defines numerous constants that deal with implementation limits of the operating system. Unfortunately, this is one of the more confusing aspects of POSIX.1. Although POSIX.1 defines numerous limits and constants, we'll concern ourselves with only the ones that affect the base POSIX.1 interfaces. These limits and constants are divided into the following seven categories:

1. Numerical limits: LONG_BIT, SSIZE_MAX, and WORD_BIT
2. Minimum values: the 25 constants in Figure 2.8
3. Maximum value: _POSIX_CLOCKRES_MIN

4. Runtime increasable values: CHARCLASS_NAME_MAX, COLL_WEIGHTS_MAX, LINE_MAX, NGROUPS_MAX, and RE_DUP_MAX
5. Runtime invariant values, possibly indeterminate: the 17 constants in Figure 2.9 (plus an additional four constants introduced in Section 12.2 and three constants introduced in Section 14.5)
6. Other invariant values: NL_ARGMAX, NL_MSGMAX, NL_SETMAX, and NL_TEXTMAX
7. Pathname variable values: FILESIZEBITS, LINK_MAX, MAX_CANON, MAX_INPUT, NAME_MAX, PATH_MAX, PIPE_BUF, and SYMLINK_MAX

Name	Description: minimum acceptable value for maximum ...	Value
_POSIX_ARG_MAX	length of arguments to exec functions	4,096
_POSIX_CHILD_MAX	number of child processes at a time per real user ID	25
_POSIX_DELAYTIMER_MAX	number of timer expiration overruns	32
_POSIX_HOST_NAME_MAX	length of a host name as returned by gethostname	255
_POSIX_LINK_MAX	number of links to a file	8
_POSIX_LOGIN_NAME_MAX	length of a login name	9
_POSIX_MAX_CANON	number of bytes on a terminal's canonical input queue	255
_POSIX_MAX_INPUT	space available on a terminal's input queue	255
_POSIX_NAME_MAX	number of bytes in a filename, not including the terminating null	14
_POSIX_NGROUPS_MAX	number of simultaneous supplementary group IDs per process	8
_POSIX_OPEN_MAX	maximum number of open files per process	20
_POSIX_PATH_MAX	number of bytes in a pathname, including the terminating null	256
_POSIX_PIPE_BUF	number of bytes that can be written atomically to a pipe	512
_POSIX_RE_DUP_MAX	number of repeated occurrences of a basic regular expression permitted by the regexec and regcomp functions when using the interval notation \{m,n\}	255
_POSIX_RTSIG_MAX	number of real-time signal numbers reserved for applications	8
_POSIX_SEM_NSEMS_MAX	number of semaphores a process can have in use at one time	256
_POSIX_SEM_VALUE_MAX	value a semaphore can hold	32,767
_POSIX_SIGQUEUE_MAX	number of queued signals a process can send and have pending	32
_POSIX_SSIZE_MAX	value that can be stored in ssize_t object	32,767
_POSIX_STREAM_MAX	number of standard I/O streams a process can have open at once	8
_POSIX_SYMLINK_MAX	number of bytes in a symbolic link	255
_POSIX_SYMLOOP_MAX	number of symbolic links that can be traversed during pathname resolution	8
_POSIX_TIMER_MAX	number of timers per process	32
_POSIX_TTY_NAME_MAX	length of a terminal device name, including the terminating null	9
_POSIX_TZNAME_MAX	number of bytes for the name of a time zone	6

Figure 2.8 POSIX.1 minimum values from <limits.h>

Of these limits and constants, some may be defined in <limits.h>, and others may or may not be defined, depending on certain conditions. We describe the limits and constants that may or may not be defined in Section 2.5.4, when we describe the sysconf, pathconf, and fpathconf functions. The 25 minimum values are shown in Figure 2.8.

These minimum values do not change from one system to another. They specify the most restrictive values for these features. A conforming POSIX.1 implementation must provide values that are at least this large. This is why they are called minimums, although their names all contain `MAX`. Also, to ensure portability, a strictly conforming application must not require a larger value. We describe what each of these constants refers to as we proceed through the text.

> A strictly conforming POSIX application is different from an application that is merely POSIX conforming. A POSIX-conforming application uses only interfaces defined in IEEE Standard 1003.1-2008. A strictly conforming POSIX application must meet further restrictions, such as not relying on any undefined behavior, not using any obsolescent interfaces, and not requiring values of constants larger than the minimums shown in Figure 2.8.

Name	Description	Minimum acceptable value
`ARG_MAX`	maximum length of arguments to `exec` functions	`_POSIX_ARG_MAX`
`ATEXIT_MAX`	maximum number of functions that can be registered with the `atexit` function	32
`CHILD_MAX`	maximum number of child processes per real user ID	`_POSIX_CHILD_MAX`
`DELAYTIMER_MAX`	maximum number of timer expiration overruns	`_POSIX_DELAYTIMER_MAX`
`HOST_NAME_MAX`	maximum length of a host name as returned by `gethostname`	`_POSIX_HOST_NAME_MAX`
`LOGIN_NAME_MAX`	maximum length of a login name	`_POSIX_LOGIN_NAME_MAX`
`OPEN_MAX`	one more than the maximum value assigned to a newly created file descriptor	`_POSIX_OPEN_MAX`
`PAGESIZE`	system memory page size, in bytes	1
`RTSIG_MAX`	maximum number of real-time signals reserved for application use	`_POSIX_RTSIG_MAX`
`SEM_NSEMS_MAX`	maximum number of semaphores a process can use	`_POSIX_SEM_NSEMS_MAX`
`SEM_VALUE_MAX`	maximum value of a semaphore	`_POSIX_SEM_VALUE_MAX`
`SIGQUEUE_MAX`	maximum number of signals that can be queued for a process	`_POSIX_SIGQUEUE_MAX`
`STREAM_MAX`	maximum number of standard I/O streams a process can have open at once	`_POSIX_STREAM_MAX`
`SYMLOOP_MAX`	number of symbolic links that can be traversed during pathname resolution	`_POSIX_SYMLOOP_MAX`
`TIMER_MAX`	maximum number of timers per process	`_POSIX_TIMER_MAX`
`TTY_NAME_MAX`	length of a terminal device name, including the terminating null	`_POSIX_TTY_NAME_MAX`
`TZNAME_MAX`	number of bytes for the name of a time zone	`_POSIX_TZNAME_MAX`

Figure 2.9 POSIX.1 runtime invariant values from `<limits.h>`

Unfortunately, some of these invariant minimum values are too small to be of practical use. For example, most UNIX systems today provide far more than 20 open files per process. Also, the minimum limit of 256 for `_POSIX_PATH_MAX` is too small. Pathnames can exceed this limit. This means that we can't use the two constants `_POSIX_OPEN_MAX` and `_POSIX_PATH_MAX` as array sizes at compile time.

Each of the 25 invariant minimum values in Figure 2.8 has an associated implementation value whose name is formed by removing the `_POSIX_` prefix from the name in Figure 2.8. The names without the leading `_POSIX_` were intended to be the actual values that a given implementation supports. (These 25 implementation values are from items 1, 4, 5, and 7 from our list earlier in this section: 2 of the runtime increasable values, 15 of the runtime invariant values, and 7 of the pathname variable values, along with `SSIZE_MAX` from the numeric values.) The problem is that not all of the 25 implementation values are guaranteed to be defined in the `<limits.h>` header.

For example, a particular value may not be included in the header if its actual value for a given process depends on the amount of memory on the system. If the values are not defined in the header, we can't use them as array bounds at compile time. To determine the actual implementation value at runtime, POSIX.1 decided to provide three functions for us to call—`sysconf`, `pathconf`, and `fpathconf`. There is still a problem, however, because some of the values are defined by POSIX.1 as being possibly "indeterminate" (logically infinite). This means that the value has no practical upper bound. On Solaris, for example, the number of functions you can register with `atexit` to be run when a process ends is limited only by the amount of memory on the system. Thus `ATEXIT_MAX` is considered indeterminate on Solaris. We'll return to this problem of indeterminate runtime limits in Section 2.5.5.

2.5.3 XSI Limits

The XSI option also defines constants representing implementation limits. They include:

1. Minimum values: the five constants in Figure 2.10
2. Runtime invariant values, possibly indeterminate: `IOV_MAX` and `PAGE_SIZE`

The minimum values are listed in Figure 2.10. The last two illustrate the situation in which the POSIX.1 minimums were too small—presumably to allow for embedded POSIX.1 implementations—so symbols with larger minimum values were added for XSI-conforming systems.

Name	Description	Minimum acceptable value	Typical value
`NL_LANGMAX`	maximum number of bytes in `LANG` environment variable	14	14
`NZERO`	default process priority	20	20
`_XOPEN_IOV_MAX`	maximum number of `iovec` structures that can be used with `readv` or `writev`	16	16
`_XOPEN_NAME_MAX`	number of bytes in a filename	255	255
`_XOPEN_PATH_MAX`	number of bytes in a pathname	1,024	1,024

Figure 2.10 XSI minimum values from `<limits.h>`

2.5.4 sysconf, pathconf, and fpathconf Functions

We've listed various minimum values that an implementation must support, but how do we find out the limits that a particular system actually supports? As we mentioned earlier, some of these limits might be available at compile time; others must be determined at runtime. We've also mentioned that some limits don't change in a given system, whereas others can change because they are associated with a file or directory. The runtime limits are obtained by calling one of the following three functions.

```
#include <unistd.h>

long sysconf(int name);

long pathconf(const char *pathname, int name);

long fpathconf(int fd, int name);
```

All three return: corresponding value if OK, –1 on error (see later)

The difference between the last two functions is that one takes a pathname as its argument and the other takes a file descriptor argument.

Figure 2.11 lists the *name* arguments that `sysconf` uses to identify system limits. Constants beginning with `_SC_` are used as arguments to `sysconf` to identify the runtime limit. Figure 2.12 lists the *name* arguments that are used by `pathconf` and `fpathconf` to identify system limits. Constants beginning with `_PC_` are used as arguments to `pathconf` and `fpathconf` to identify the runtime limit.

We need to look in more detail at the different return values from these three functions.

1. All three functions return –1 and set `errno` to `EINVAL` if the *name* isn't one of the appropriate constants. The third column in Figures 2.11 and 2.12 lists the limit constants we'll deal with throughout the rest of this book.
2. Some *name*s can return either the value of the variable (a return value ≥ 0) or an indication that the value is indeterminate. An indeterminate value is indicated by returning –1 and not changing the value of `errno`.
3. The value returned for `_SC_CLK_TCK` is the number of clock ticks per second, for use with the return values from the `times` function (Section 8.17).

Some restrictions apply to the `pathconf` *pathname* argument and the `fpathconf` *fd* argument. If any of these restrictions isn't met, the results are undefined.

1. The referenced file for `_PC_MAX_CANON` and `_PC_MAX_INPUT` must be a terminal file.
2. The referenced file for `_PC_LINK_MAX` and `_PC_TIMESTAMP_RESOLUTION` can be either a file or a directory. If the referenced file is a directory, the return value applies to the directory itself, not to the filename entries within the directory.
3. The referenced file for `_PC_FILESIZEBITS` and `_PC_NAME_MAX` must be a directory. The return value applies to filenames within the directory.

Name of limit	Description	*name* argument
ARG_MAX	maximum length, in bytes, of arguments to the exec functions	_SC_ARG_MAX
ATEXIT_MAX	maximum number of functions that can be registered with the atexit function	_SC_ATEXIT_MAX
CHILD_MAX	maximum number of processes per real user ID	_SC_CHILD_MAX
clock ticks/second	number of clock ticks per second	_SC_CLK_TCK
COLL_WEIGHTS_MAX	maximum number of weights that can be assigned to an entry of the LC_COLLATE order keyword in the locale definition file	_SC_COLL_WEIGHTS_MAX
DELAYTIMER_MAX	maximum number of timer expiration overruns	_SC_DELAYTIMER_MAX
HOST_NAME_MAX	maximum length of a host name as returned by gethostname	_SC_HOST_NAME_MAX
IOV_MAX	maximum number of iovec structures that can be used with readv or writev	_SC_IOV_MAX
LINE_MAX	maximum length of a utility's input line	_SC_LINE_MAX
LOGIN_NAME_MAX	maximum length of a login name	_SC_LOGIN_NAME_MAX
NGROUPS_MAX	maximum number of simultaneous supplementary process group IDs per process	_SC_NGROUPS_MAX
OPEN_MAX	one more than the maximum value assigned to a newly created file descriptor	_SC_OPEN_MAX
PAGESIZE	system memory page size, in bytes	_SC_PAGESIZE
PAGE_SIZE	system memory page size, in bytes	_SC_PAGE_SIZE
RE_DUP_MAX	number of repeated occurrences of a basic regular expression permitted by the regexec and regcomp functions when using the interval notation \{m,n\}	_SC_RE_DUP_MAX
RTSIG_MAX	maximum number of real-time signals reserved for application use	_SC_RTSIG_MAX
SEM_NSEMS_MAX	maximum number of semaphores a process can use at one time	_SC_SEM_NSEMS_MAX
SEM_VALUE_MAX	maximum value of a semaphore	_SC_SEM_VALUE_MAX
SIGQUEUE_MAX	maximum number of signals that can be queued for a process	_SC_SIGQUEUE_MAX
STREAM_MAX	maximum number of standard I/O streams per process at any given time; if defined, it must have the same value as FOPEN_MAX	_SC_STREAM_MAX
SYMLOOP_MAX	number of symbolic links that can be traversed during pathname resolution	_SC_SYMLOOP_MAX
TIMER_MAX	maximum number of timers per process	_SC_TIMER_MAX
TTY_NAME_MAX	length of a terminal device name, including the terminating null	_SC_TTY_NAME_MAX
TZNAME_MAX	maximum number of bytes for a time zone name	_SC_TZNAME_MAX

Figure 2.11 Limits and *name* arguments to sysconf

4. The referenced file for _PC_PATH_MAX must be a directory. The value returned is the maximum length of a relative pathname when the specified directory is the working directory. (Unfortunately, this isn't the real maximum length of an absolute pathname, which is what we want to know. We'll return to this problem in Section 2.5.5.)

Name of limit	Description	*name* argument
FILESIZEBITS	minimum number of bits needed to represent, as a signed integer value, the maximum size of a regular file allowed in the specified directory	_PC_FILESIZEBITS
LINK_MAX	maximum value of a file's link count	_PC_LINK_MAX
MAX_CANON	maximum number of bytes on a terminal's canonical input queue	_PC_MAX_CANON
MAX_INPUT	number of bytes for which space is available on terminal's input queue	_PC_MAX_INPUT
NAME_MAX	maximum number of bytes in a filename (does not include a null at end)	_PC_NAME_MAX
PATH_MAX	maximum number of bytes in a relative pathname, including the terminating null	_PC_PATH_MAX
PIPE_BUF	maximum number of bytes that can be written atomically to a pipe	_PC_PIPE_BUF
_POSIX_TIMESTAMP_RESOLUTION	resolution in nanoseconds for file timestamps	_PC_TIMESTAMP_RESOLUTION
SYMLINK_MAX	number of bytes in a symbolic link	_PC_SYMLINK_MAX

Figure 2.12 Limits and *name* arguments to `pathconf` and `fpathconf`

5. The referenced file for `_PC_PIPE_BUF` must be a pipe, FIFO, or directory. In the first two cases (pipe or FIFO), the return value is the limit for the referenced pipe or FIFO. For the other case (a directory), the return value is the limit for any FIFO created in that directory.
6. The referenced file for `_PC_SYMLINK_MAX` must be a directory. The value returned is the maximum length of the string that a symbolic link in that directory can contain.

Example

The `awk`(1) program shown in Figure 2.13 builds a C program that prints the value of each `pathconf` and `sysconf` symbol.

```
#!/usr/bin/awk -f
BEGIN   {
    printf("#include \"apue.h\"\n")
    printf("#include <errno.h>\n")
    printf("#include <limits.h>\n")
    printf("\n")
    printf("static void pr_sysconf(char *, int);\n")
    printf("static void pr_pathconf(char *, char *, int);\n")
    printf("\n")
    printf("int\n")
    printf("main(int argc, char *argv[])\n")
```

```
        printf("{\n")
        printf("\tif (argc != 2)\n")
        printf("\t\terr_quit(\"usage: a.out <dirname>\");\n\n")
        FS="\t+"
        while (getline <"sysconf.sym" > 0) {
            printf("#ifdef %s\n", $1)
            printf("\tprintf(\"%s defined to be %%ld\\n\", (long)%s+0);\n",
                 $1, $1)
            printf("#else\n")
            printf("\tprintf(\"no symbol for %s\\n\");\n", $1)
            printf("#endif\n")
            printf("#ifdef %s\n", $2)
            printf("\tpr_sysconf(\"%s =\", %s);\n", $1, $2)
            printf("#else\n")
            printf("\tprintf(\"no symbol for %s\\n\");\n", $2)
            printf("#endif\n")
        }
        close("sysconf.sym")
        while (getline <"pathconf.sym" > 0) {
            printf("#ifdef %s\n", $1)
            printf("\tprintf(\"%s defined to be %%ld\\n\", (long)%s+0);\n",
                $1, $1)
            printf("#else\n")
            printf("\tprintf(\"no symbol for %s\\n\");\n", $1)
            printf("#endif\n")
            printf("#ifdef %s\n", $2)
            printf("\tpr_pathconf(\"%s =\", argv[1], %s);\n", $1, $2)
            printf("#else\n")
            printf("\tprintf(\"no symbol for %s\\n\");\n", $2)
            printf("#endif\n")
        }
        close("pathconf.sym")
        exit
    }
    END {
        printf("\texit(0);\n")
        printf("}\n\n")
        printf("static void\n")
        printf("pr_sysconf(char *mesg, int name)\n")
        printf("{\n")
        printf("\tlong  val;\n\n")
        printf("\tfputs(mesg, stdout);\n")
        printf("\terrno = 0;\n")
        printf("\tif ((val = sysconf(name)) < 0) {\n")
        printf("\t\tif (errno != 0) {\n")
        printf("\t\t\tif (errno == EINVAL)\n")
        printf("\t\t\t\tfputs(\" (not supported)\\n\", stdout);\n")
        printf("\t\t\telse\n")
        printf("\t\t\t\terr_sys(\"sysconf error\");\n")
        printf("\t\t} else {\n")
        printf("\t\t\tfputs(\" (no limit)\\n\", stdout);\n")
```

```
    printf("\t\t}\n")
    printf("\t} else {\n")
    printf("\t\tprintf(\" %%ld\\n\", val);\n")
    printf("\t}\n")
    printf("}\n\n")
    printf("static void\n")
    printf("pr_pathconf(char *mesg, char *path, int name)\n")
    printf("{\n")
    printf("\tlong  val;\n")
    printf("\n")
    printf("\tfputs(mesg, stdout);\n")
    printf("\terrno = 0;\n")
    printf("\tif ((val = pathconf(path, name)) < 0) {\n")
    printf("\t\tif (errno != 0) {\n")
    printf("\t\t\tif (errno == EINVAL)\n")
    printf("\t\t\t\tfputs(\" (not supported)\\n\", stdout);\n")
    printf("\t\t\telse\n")
    printf("\t\t\t\terr_sys(\"pathconf error, path = %%s\", path);\n")
    printf("\t\t} else {\n")
    printf("\t\t\tfputs(\" (no limit)\\n\", stdout);\n")
    printf("\t\t}\n")
    printf("\t} else {\n")
    printf("\t\tprintf(\" %%ld\\n\", val);\n")
    printf("\t}\n")
    printf("}\n")
}
```

Figure 2.13 Build C program to print all supported configuration limits

The `awk` program reads two input files—`pathconf.sym` and `sysconf.sym`—that contain lists of the limit name and symbol, separated by tabs. All symbols are not defined on every platform, so the `awk` program surrounds each call to `pathconf` and `sysconf` with the necessary `#ifdef` statements.

For example, the `awk` program transforms a line in the input file that looks like

```
NAME_MAX        _PC_NAME_MAX
```

into the following C code:

```
#ifdef NAME_MAX
    printf("NAME_MAX is defined to be %d\n", NAME_MAX+0);
#else
    printf("no symbol for NAME_MAX\n");
#endif
#ifdef _PC_NAME_MAX
    pr_pathconf("NAME_MAX =", argv[1], _PC_NAME_MAX);
#else
    printf("no symbol for _PC_NAME_MAX\n");
#endif
```

The program in Figure 2.14, generated by the `awk` program, prints all these limits, handling the case in which a limit is not defined.

```
#include "apue.h"
#include <errno.h>
#include <limits.h>

static void pr_sysconf(char *, int);
static void pr_pathconf(char *, char *, int);

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <dirname>");

#ifdef ARG_MAX
    printf("ARG_MAX defined to be %ld\n", (long)ARG_MAX+0);
#else
    printf("no symbol for ARG_MAX\n");
#endif
#ifdef _SC_ARG_MAX
    pr_sysconf("ARG_MAX =", _SC_ARG_MAX);
#else
    printf("no symbol for _SC_ARG_MAX\n");
#endif

/* similar processing for all the rest of the sysconf symbols... */

#ifdef MAX_CANON
    printf("MAX_CANON defined to be %ld\n", (long)MAX_CANON+0);
#else
    printf("no symbol for MAX_CANON\n");
#endif
#ifdef _PC_MAX_CANON
    pr_pathconf("MAX_CANON =", argv[1], _PC_MAX_CANON);
#else
    printf("no symbol for _PC_MAX_CANON\n");
#endif

/* similar processing for all the rest of the pathconf symbols... */

    exit(0);
}

static void
pr_sysconf(char *mesg, int name)
{
    long    val;

    fputs(mesg, stdout);
    errno = 0;
    if ((val = sysconf(name)) < 0) {
        if (errno != 0) {
            if (errno == EINVAL)
                fputs(" (not supported)\n", stdout);
            else
```

```
                err_sys("sysconf error");
        } else {
            fputs(" (no limit)\n", stdout);
        }
    } else {
        printf(" %ld\n", val);
    }
}

static void
pr_pathconf(char *mesg, char *path, int name)
{
    long    val;

    fputs(mesg, stdout);
    errno = 0;
    if ((val = pathconf(path, name)) < 0) {
        if (errno != 0) {
            if (errno == EINVAL)
                fputs(" (not supported)\n", stdout);
            else
                err_sys("pathconf error, path = %s", path);
        } else {
            fputs(" (no limit)\n", stdout);
        }
    } else {
        printf(" %ld\n", val);
    }
}
```

Figure 2.14 Print all possible `sysconf` and `pathconf` values

Figure 2.15 summarizes the results from Figure 2.14 for the four systems we discuss in this book. The entry "no symbol" means that the system doesn't provide a corresponding `_SC` or `_PC` symbol to query the value of the constant. Thus the limit is undefined in this case. In contrast, the entry "unsupported" means that the symbol is defined by the system but unrecognized by the `sysconf` or `pathconf` functions. The entry "no limit" means that the system defines no limit for the constant, but this doesn't mean that the limit is infinite; it just means that the limit is indeterminite.

> Beware that some limits are reported incorrectly. For example, on Linux, `SYMLOOP_MAX` is reportedly unlimited, but an examination of the source code reveals that there is actually a hard-coded limit of 40 for the number of consecutive symbolic links traversed in the absence of a loop (see the `follow_link` function in `fs/namei.c`).
>
> Another potential source of inaccuracy in Linux is that the `pathconf` and `fpathconf` functions are implemented in the C library. The configuration limits returned by these functions depend on the underlying file system type, so if your file system is unknown to the C library, the functions return an educated guess.

We'll see in Section 4.14 that `UFS` is the SVR4 implementation of the Berkeley fast file system. PCFS is the MS-DOS FAT file system implementation for Solaris. □

Limit	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	
				UFS file system	PCFS file system
ARG_MAX	262,144	2,097,152	262,144	2,096,640	2,096,640
ATEXIT_MAX	32	2,147,483,647	2,147,483,647	no limit	no limit
CHARCLASS_NAME_MAX	no symbol	2,048	14	14	14
CHILD_MAX	1,760	47,211	266	8,021	8,021
clock ticks/second	128	100	100	100	100
COLL_WEIGHTS_MAX	0	255	2	10	10
FILESIZEBITS	64	64	64	41	unsupported
HOST_NAME_MAX	255	64	255	255	255
IOV_MAX	1,024	1,024	1024	16	16
LINE_MAX	2,048	2,048	2,048	2,048	2,048
LINK_MAX	32,767	65,000	32,767	32,767	1
LOGIN_NAME_MAX	17	256	255	9	9
MAX_CANON	255	255	1,024	256	256
MAX_INPUT	255	255	1,024	512	512
NAME_MAX	255	255	255	255	8
NGROUPS_MAX	1,023	65,536	16	16	16
OPEN_MAX	3,520	1,024	256	256	256
PAGESIZE	4,096	4,096	4,096	8,192	8,192
PAGE_SIZE	4,096	4,096	4,096	8,192	8,192
PATH_MAX	1,024	4,096	1,024	1,024	1,024
PIPE_BUF	512	4,096	512	5,120	5,120
RE_DUP_MAX	255	32,767	255	255	255
STREAM_MAX	3,520	16	20	256	256
SYMLINK_MAX	1,024	no limit	255	1,024	1,024
SYMLOOP_MAX	32	no limit	32	20	20
TTY_NAME_MAX	255	32	255	128	128
TZNAME_MAX	255	6	255	no limit	no limit

Figure 2.15 Examples of configuration limits

2.5.5 Indeterminate Runtime Limits

We mentioned that some of the limits can be indeterminate. The problem we encounter is that if these limits aren't defined in the `<limits.h>` header, we can't use them at compile time. But they might not be defined at runtime if their value is indeterminate! Let's look at two specific cases: allocating storage for a pathname and determining the number of file descriptors.

Pathnames

Many programs need to allocate storage for a pathname. Typically, the storage has been allocated at compile time, and various magic numbers—few of which are the correct value—have been used by different programs as the array size: 256, 512, 1024, or the standard I/O constant `BUFSIZ`. The 4.3BSD constant `MAXPATHLEN` in the header `<sys/param.h>` is the correct value, but many 4.3BSD applications didn't use it.

POSIX.1 tries to help with the PATH_MAX value, but if this value is indeterminate, we're still out of luck. Figure 2.16 shows a function that we'll use throughout this text to allocate storage dynamically for a pathname.

If the constant PATH_MAX is defined in <limits.h>, then we're all set. If it's not, then we need to call pathconf. The value returned by pathconf is the maximum size of a relative pathname when the first argument is the working directory, so we specify the root as the first argument and add 1 to the result. If pathconf indicates that PATH_MAX is indeterminate, we have to punt and just guess a value.

Versions of POSIX.1 prior to 2001 were unclear as to whether PATH_MAX included a null byte at the end of the pathname. If the operating system implementation conforms to one of these prior versions and doesn't conform to any version of the Single UNIX Specification (which *does* require the terminating null byte to be included), we need to add 1 to the amount of memory we allocate for a pathname, just to be on the safe side.

The correct way to handle the case of an indeterminate result depends on how the allocated space is being used. If we are allocating space for a call to getcwd, for example—to return the absolute pathname of the current working directory; see Section 4.23—and if the allocated space is too small, an error is returned and errno is set to ERANGE. We could then increase the allocated space by calling realloc (see Section 7.8 and Exercise 4.16) and try again. We could keep doing this until the call to getcwd succeeded.

```
#include "apue.h"
#include <errno.h>
#include <limits.h>

#ifdef  PATH_MAX
static long pathmax = PATH_MAX;
#else
static long pathmax = 0;
#endif

static long posix_version = 0;
static long xsi_version = 0;

/* If PATH_MAX is indeterminate, no guarantee this is adequate */
#define PATH_MAX_GUESS  1024

char *
path_alloc(size_t *sizep) /* also return allocated size, if nonnull */
{
    char    *ptr;
    size_t  size;

    if (posix_version == 0)
        posix_version = sysconf(_SC_VERSION);

    if (xsi_version == 0)
        xsi_version = sysconf(_SC_XOPEN_VERSION);

    if (pathmax == 0) {     /* first time through */
```

```
        errno = 0;
        if ((pathmax = pathconf("/", _PC_PATH_MAX)) < 0) {
            if (errno == 0)
                pathmax = PATH_MAX_GUESS;   /* it's indeterminate */
            else
                err_sys("pathconf error for _PC_PATH_MAX");
        } else {
            pathmax++;      /* add one since it's relative to root */
        }
    }

    /*
     * Before POSIX.1-2001, we aren't guaranteed that PATH_MAX includes
     * the terminating null byte.  Same goes for XPG3.
     */
    if ((posix_version < 200112L) && (xsi_version < 4))
        size = pathmax + 1;
    else
        size = pathmax;

    if ((ptr = malloc(size)) == NULL)
        err_sys("malloc error for pathname");

    if (sizep != NULL)
        *sizep = size;
    return(ptr);
}
```

Figure 2.16 Dynamically allocate space for a pathname

Maximum Number of Open Files

A common sequence of code in a daemon process—a process that runs in the background, not connected to a terminal—is one that closes all open files. Some programs have the following code sequence, assuming the constant `NOFILE` was defined in the `<sys/param.h>` header:

```
#include  <sys/param.h>

for (i = 0; i < NOFILE; i++)
    close(i);
```

Other programs use the constant `_NFILE` that some versions of `<stdio.h>` provide as the upper limit. Some hard code the upper limit as 20. However, none of these approaches is portable.

We would hope to use the POSIX.1 value `OPEN_MAX` to determine this value portably, but if the value is indeterminate, we still have a problem. If we wrote the following code and if `OPEN_MAX` was indeterminate, the loop would never execute, since `sysconf` would return –1:

```
#include  <unistd.h>

for (i = 0; i < sysconf(_SC_OPEN_MAX); i++)
    close(i);
```

Our best option in this case is just to close all descriptors up to some arbitrary limit—say, 256. We show this technique in Figure 2.17. As with our pathname example, this strategy is not guaranteed to work for all cases, but it's the best we can do without using a more exotic approach.

```
#include "apue.h"
#include <errno.h>
#include <limits.h>

#ifdef  OPEN_MAX
static long openmax = OPEN_MAX;
#else
static long openmax = 0;
#endif

/*
 * If OPEN_MAX is indeterminate, this might be inadequate.
 */
#define OPEN_MAX_GUESS  256

long
open_max(void)
{
    if (openmax == 0) {     /* first time through */
        errno = 0;
        if ((openmax = sysconf(_SC_OPEN_MAX)) < 0) {
            if (errno == 0)
                openmax = OPEN_MAX_GUESS;   /* it's indeterminate */
            else
                err_sys("sysconf error for _SC_OPEN_MAX");
        }
    }
    return(openmax);
}
```

Figure 2.17 Determine the number of file descriptors

We might be tempted to call `close` until we get an error return, but the error return from `close` (`EBADF`) doesn't distinguish between an invalid descriptor and a descriptor that wasn't open. If we tried this technique and descriptor 9 was not open but descriptor 10 was, we would stop on 9 and never close 10. The `dup` function (Section 3.12) does return a specific error when `OPEN_MAX` is exceeded, but duplicating a descriptor a couple of hundred times is an extreme way to determine this value.

Some implementations will return `LONG_MAX` for limit values that are effectively unlimited. Such is the case with the Linux limit for `ATEXIT_MAX` (see Figure 2.15). This isn't a good idea, because it can cause programs to behave badly.

For example, we can use the `ulimit` command built into the Bourne-again shell to change the maximum number of files our processes can have open at one time. This generally requires special (superuser) privileges if the limit is to be effectively unlimited. But once set to infinite, `sysconf` will report `LONG_MAX` as the limit for `OPEN_MAX`. A program that relies on this value as the upper bound of file descriptors to close, as shown in Figure 2.17, will waste a lot of time trying to close 2,147,483,647 file descriptors, most of which aren't even in use.

Systems that support the XSI option in the Single UNIX Specification will provide the `getrlimit`(2) function (Section 7.11). It can be used to return the maximum number of descriptors that a process can have open. With it, we can detect that there is no configured upper bound to the number of open files our processes can open, so we can avoid this problem.

> The `OPEN_MAX` value is called runtime invariant by POSIX, meaning that its value should not change during the lifetime of a process. But on systems that support the XSI option, we can call the `setrlimit`(2) function (Section 7.11) to change this value for a running process. (This value can also be changed from the C shell with the `limit` command, and from the Bourne, Bourne-again, Debian Almquist, and Korn shells with the `ulimit` command.) If our system supports this functionality, we could change the function in Figure 2.17 to call `sysconf` every time it is called, not just the first time.

2.6 Options

We saw the list of POSIX.1 options in Figure 2.5 and discussed XSI option groups in Section 2.2.3. If we are to write portable applications that depend on any of these optionally supported features, we need a portable way to determine whether an implementation supports a given option.

Just as with limits (Section 2.5), POSIX.1 defines three ways to do this.

1. Compile-time options are defined in `<unistd.h>`.
2. Runtime options that are not associated with a file or a directory are identified with the `sysconf` function.
3. Runtime options that are associated with a file or a directory are discovered by calling either the `pathconf` or the `fpathconf` function.

The options include the symbols listed in the third column of Figure 2.5, as well as the symbols listed in Figures 2.19 and 2.18. If the symbolic constant is not defined, we must use `sysconf`, `pathconf`, or `fpathconf` to determine whether the option is supported. In this case, the *name* argument to the function is formed by replacing the `_POSIX` at the beginning of the symbol with `_SC` or `_PC`. For constants that begin with `_XOPEN`, the *name* argument is formed by prepending the string with `_SC` or `_PC`. For example, if the constant `_POSIX_RAW_SOCKETS` is undefined, we can call `sysconf` with the *name* argument set to `_SC_RAW_SOCKETS` to determine whether the platform supports the raw sockets option. If the constant `_XOPEN_UNIX` is undefined, we can call `sysconf` with the *name* argument set to `_SC_XOPEN_UNIX` to determine whether the platform supports the XSI option interfaces.

For each option, we have three possibilities for a platform's support status.

1. If the symbolic constant is either undefined or defined to have the value –1, then the corresponding option is unsupported by the platform at compile time. It is possible to run an old application on a newer system where the option *is* supported, so a runtime check might indicate the option is supported even though the option wasn't supported at the time the application was compiled.
2. If the symbolic constant is defined to be greater than zero, then the corresponding option is supported.
3. If the symbolic constant is defined to be equal to zero, then we must call `sysconf`, `pathconf`, or `fpathconf` to determine whether the option is supported.

The symbolic constants used with `pathconf` and `fpathconf` are summarized in Figure 2.18. Figure 2.19 summarizes the nonobsolete options and their symbolic constants that can be used with `sysconf`, in addition to those listed in Figure 2.5. Note that we omit options associated with utility commands.

As with the system limits, there are several points to note regarding how options are treated by `sysconf`, `pathconf`, and `fpathconf`.

1. The value returned for `_SC_VERSION` indicates the four-digit year and two-digit month of the standard. This value can be 198808L, 199009L, 199506L, or some other value for a later version of the standard. The value associated with Version 3 of the Single UNIX Specification is 200112L (the 2001 edition of POSIX.1). The value associated with Version 4 of the Single UNIX Specification (the 2008 edition of POSIX.1) is 200809L.
2. The value returned for `_SC_XOPEN_VERSION` indicates the version of the XSI that the system supports. The value associated with Version 3 of the Single UNIX Specification is 600. The value associated with Version 4 of the Single UNIX Specification (the 2008 edition of POSIX.1) is 700.
3. The values `_SC_JOB_CONTROL`, `_SC_SAVED_IDS`, and `_PC_VDISABLE` no longer represent optional features. Although XPG4 and prior versions of the Single UNIX Specification required that these features be supported, Version 3 of the Single UNIX Specification is the earliest version where these features are no longer optional in POSIX.1. These symbols are retained for backward compatibility.
4. Platforms conforming to POSIX.1-2008 are also required to support the following options:
   - `_POSIX_ASYNCHRONOUS_IO`
   - `_POSIX_BARRIERS`
   - `_POSIX_CLOCK_SELECTION`
   - `_POSIX_MAPPED_FILES`
   - `_POSIX_MEMORY_PROTECTION`

- _POSIX_READER_WRITER_LOCKS
- _POSIX_REALTIME_SIGNALS
- _POSIX_SEMAPHORES
- _POSIX_SPIN_LOCKS
- _POSIX_THREAD_SAFE_FUNCTIONS
- _POSIX_THREADS
- _POSIX_TIMEOUTS
- _POSIX_TIMERS

These constants are defined to have the value 200809L. Their corresponding _SC symbols are also retained for backward compatibility.

5. _PC_CHOWN_RESTRICTED and _PC_NO_TRUNC return –1 without changing errno if the feature is not supported for the specified *pathname* or *fd*. On all POSIX-conforming systems, the return value will be greater than zero (indicating that the feature is supported).
6. The referenced file for _PC_CHOWN_RESTRICTED must be either a file or a directory. If it is a directory, the return value indicates whether this option applies to files within that directory.
7. The referenced file for _PC_NO_TRUNC and _PC_2_SYMLINKS must be a directory.
8. For _PC_NO_TRUNC, the return value applies to filenames within the directory.
9. The referenced file for _PC_VDISABLE must be a terminal file.
10. For _PC_ASYNC_IO, _PC_PRIO_IO, and _PC_SYNC_IO, the referenced file must not be a directory.

Name of option	Indicates ...	*name* argument
_POSIX_CHOWN_RESTRICTED	whether use of chown is restricted	_PC_CHOWN_RESTRICTED
_POSIX_NO_TRUNC	whether filenames longer than NAME_MAX generate an error	_PC_NO_TRUNC
_POSIX_VDISABLE	if defined, terminal special characters can be disabled with this value	_PC_VDISABLE
_POSIX_ASYNC_IO	whether asynchronous I/O can be used with the associated file	_PC_ASYNC_IO
_POSIX_PRIO_IO	whether prioritized I/O can be used with the associated file	_PC_PRIO_IO
_POSIX_SYNC_IO	whether synchronized I/O can be used with the associated file	_PC_SYNC_IO
_POSIX2_SYMLINKS	whether symbolic links are supported in the directory	_PC_2_SYMLINKS

Figure 2.18 Options and *name* arguments to pathconf and fpathconf

Name of option	Indicates ...	*name* argument
_POSIX_ASYNCHRONOUS_IO	whether the implementation supports POSIX asynchronous I/O	_SC_ASYNCHRONOUS_IO
_POSIX_BARRIERS	whether the implementation supports barriers	_SC_BARRIERS
_POSIX_CLOCK_SELECTION	whether the implementation supports clock selection	_SC_CLOCK_SELECTION
_POSIX_JOB_CONTROL	whether the implementation supports job control	_SC_JOB_CONTROL
_POSIX_MAPPED_FILES	whether the implementation supports memory-mapped files	_SC_MAPPED_FILES
_POSIX_MEMORY_PROTECTION	whether the implementation supports memory protection	_SC_MEMORY_PROTECTION
_POSIX_READER_WRITER_LOCKS	whether the implementation supports reader–writer locks	_SC_READER_WRITER_LOCKS
_POSIX_REALTIME_SIGNALS	whether the implementation supports real-time signals	_SC_REALTIME_SIGNALS
_POSIX_SAVED_IDS	whether the implementation supports the saved set-user-ID and the saved set-group-ID	_SC_SAVED_IDS
_POSIX_SEMAPHORES	whether the implementation supports POSIX semaphores	_SC_SEMAPHORES
_POSIX_SHELL	whether the implementation supports the POSIX shell	_SC_SHELL
_POSIX_SPIN_LOCKS	whether the implementation supports spin locks	_SC_SPIN_LOCKS
_POSIX_THREAD_SAFE_FUNCTIONS	whether the implementation supports thread-safe functions	_SC_THREAD_SAFE_FUNCTIONS
_POSIX_THREADS	whether the implementation supports threads	_SC_THREADS
_POSIX_TIMEOUTS	whether the implementation supports timeout-based variants of selected functions	_SC_TIMEOUTS
_POSIX_TIMERS	whether the implementation supports timers	_SC_TIMERS
_POSIX_VERSION	the POSIX.1 version	_SC_VERSION
_XOPEN_CRYPT	whether the implementation supports the XSI encryption option group	_SC_XOPEN_CRYPT
_XOPEN_REALTIME	whether the implementation supports the XSI real-time option group	_SC_XOPEN_REALTIME
_XOPEN_REALTIME_THREADS	whether the implementation supports the XSI real-time threads option group	_SC_XOPEN_REALTIME_THREADS
_XOPEN_SHM	whether the implementation supports the XSI shared memory option group	_SC_XOPEN_SHM
_XOPEN_VERSION	the XSI version	_SC_XOPEN_VERSION

Figure 2.19 Options and *name* arguments to `sysconf`

Figure 2.20 shows several configuration options and their corresponding values on the four sample systems we discuss in this text. An entry is "unsupported" if the system defines the symbolic constant but it has a value of –1, or if it has a value of 0 but the corresponding `sysconf` or `pathconf` call returned –1. It is interesting to see that some system implementations haven't yet caught up to the latest version of the Single UNIX Specification.

Limit	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	
				UFS file system	PCFS file system
_POSIX_CHOWN_RESTRICTED	1	1	200112	1	1
_POSIX_JOB_CONTROL	1	1	200112	1	1
_POSIX_NO_TRUNC	1	1	200112	1	unsupported
_POSIX_SAVED_IDS	unsupported	1	200112	1	1
_POSIX_THREADS	200112	200809	200112	200112	200112
_POSIX_VDISABLE	255	0	255	0	0
_POSIX_VERSION	200112	200809	200112	200112	200112
_XOPEN_UNIX	unsupported	1	1	1	1
_XOPEN_VERSION	unsupported	700	600	600	600

Figure 2.20 Examples of configuration options

Note that `pathconf` returns a value of –1 for `_PC_NO_TRUNC` when used with a file from a PCFS file system on Solaris. The PCFS file system supports the DOS format (for floppy disks), and DOS filenames are silently truncated to the 8.3 format limit that the DOS file system requires.

2.7 Feature Test Macros

The headers define numerous POSIX.1 and XSI symbols, as we've described. Even so, most implementations can add their own definitions to these headers, in addition to the POSIX.1 and XSI definitions. If we want to compile a program so that it depends only on the POSIX definitions and doesn't conflict with any implementation-defined constants, we need to define the constant `_POSIX_C_SOURCE`. All the POSIX.1 headers use this constant to exclude any implementation-defined definitions when `_POSIX_C_SOURCE` is defined.

> Older versions of the POSIX.1 standard defined the `_POSIX_SOURCE` constant. This was superseded by the `_POSIX_C_SOURCE` constant in the 2001 version of POSIX.1.

The constants `_POSIX_C_SOURCE` and `_XOPEN_SOURCE` are called *feature test macros*. All feature test macros begin with an underscore. When used, they are typically defined in the `cc` command, as in

```
cc -D_POSIX_C_SOURCE=200809L file.c
```

This causes the feature test macro to be defined before any header files are included by the C program. If we want to use only the POSIX.1 definitions, we can also set the first line of a source file to

```
#define _POSIX_C_SOURCE  200809L
```

To enable the XSI option of Version 4 of the Single UNIX Specification, we need to define the constant `_XOPEN_SOURCE` to be 700. Besides enabling the XSI option, this has the same effect as defining `_POSIX_C_SOURCE` to be 200809L as far as POSIX.1 functionality is concerned.

The Single UNIX Specification defines the `c99` utility as the interface to the C compilation environment. With it we can compile a file as follows:

```
c99 -D_XOPEN_SOURCE=700 file.c -o file
```

To enable the 1999 ISO C extensions in the `gcc` C compiler, we use the `-std=c99` option, as in

```
gcc -D_XOPEN_SOURCE=700 -std=c99 file.c -o file
```

2.8 Primitive System Data Types

Historically, certain C data types have been associated with certain UNIX system variables. For example, major and minor device numbers have historically been stored in a 16-bit short integer, with 8 bits for the major device number and 8 bits for the minor device number. But many larger systems need more than 256 values for these device numbers, so a different technique is needed. (Indeed, the 32-bit version of Solaris uses 32 bits for the device number: 14 bits for the major and 18 bits for the minor.)

The header `<sys/types.h>` defines some implementation-dependent data types, called the *primitive system data types*. More of these data types are defined in other headers as well. These data types are defined in the headers with the C `typedef` facility. Most end in `_t`. Figure 2.21 lists many of the primitive system data types that we'll encounter in this text.

By defining these data types this way, we do not build into our programs implementation details that can change from one system to another. We describe what each of these data types is used for when we encounter them later in the text.

2.9 Differences Between Standards

All in all, these various standards fit together nicely. Our main concern is any differences between the ISO C standard and POSIX.1, since the Base Specifications of the Single UNIX Specification and POSIX.1 are one and the same. Conflicts are unintended, but if they should arise, POSIX.1 defers to the ISO C standard. However, there are some differences.

ISO C defines the function `clock` to return the amount of CPU time used by a process. The value returned is a `clock_t` value, but ISO C doesn't specify its units. To

Type	Description
clock_t	counter of clock ticks (process time) (Section 1.10)
comp_t	compressed clock ticks (not defined by POSIX.1; see Section 8.14)
dev_t	device numbers (major and minor) (Section 4.24)
fd_set	file descriptor sets (Section 14.4.1)
fpos_t	file position (Section 5.10)
gid_t	numeric group IDs
ino_t	i-node numbers (Section 4.14)
mode_t	file type, file creation mode (Section 4.5)
nlink_t	link counts for directory entries (Section 4.14)
off_t	file sizes and offsets (signed) (lseek, Section 3.6)
pid_t	process IDs and process group IDs (signed) (Sections 8.2 and 9.4)
pthread_t	thread IDs (Section 11.3)
ptrdiff_t	result of subtracting two pointers (signed)
rlim_t	resource limits (Section 7.11)
sig_atomic_t	data type that can be accessed atomically (Section 10.15)
sigset_t	signal set (Section 10.11)
size_t	sizes of objects (such as strings) (unsigned) (Section 3.7)
ssize_t	functions that return a count of bytes (signed) (read, write, Section 3.7)
time_t	counter of seconds of calendar time (Section 1.10)
uid_t	numeric user IDs
wchar_t	can represent all distinct character codes

Figure 2.21 Some common primitive system data types

convert this value to seconds, we divide it by CLOCKS_PER_SEC, which is defined in the <time.h> header. POSIX.1 defines the function times that returns both the CPU time (for the caller and all its terminated children) and the clock time. All these time values are clock_t values. The sysconf function is used to obtain the number of clock ticks per second for use with the return values from the times function. What we have is the same data type (clock_t) used to hold measurements of time defined with different units by ISO C and POSIX.1. The difference can be seen in Solaris, where clock returns microseconds (hence CLOCKS_PER_SEC is 1 million), whereas sysconf returns the value 100 for clock ticks per second. Thus we must take care when using variables of type clock_t so that we don't mix variables with different units.

Another area of potential conflict is when the ISO C standard specifies a function, but doesn't specify it as strongly as POSIX.1 does. This is the case for functions that require a different implementation in a POSIX environment (with multiple processes) than in an ISO C environment (where very little can be assumed about the host operating system). Nevertheless, POSIX-compliant systems implement the ISO C function for compatibility. The signal function is an example. If we unknowingly use the signal function provided by Solaris (hoping to write portable code that can be run in ISO C environments and under older UNIX systems), it will provide semantics different from the POSIX.1 sigaction function. We'll have more to say about the signal function in Chapter 10.

2.10 Summary

Much has happened with the standardization of the UNIX programming environment over the past two and a half decades. We've described the dominant standards—ISO C, POSIX, and the Single UNIX Specification—and their effect on the four platforms that we'll examine in this text—FreeBSD, Linux, Mac OS X, and Solaris. These standards try to define certain parameters that can change with each implementation, but we've seen that these limits are imperfect. We'll encounter many of these limits and magic constants as we proceed through the text.

The bibliography specifies how to obtain copies of the standards discussed in this chapter.

Exercises

2.1 We mentioned in Section 2.8 that some of the primitive system data types are defined in more than one header. For example, in FreeBSD 8.0, `size_t` is defined in 29 different headers. Because all 29 headers could be included in a program and because ISO C does not allow multiple `typedef`s for the same name, how must the headers be written?

2.2 Examine your system's headers and list the actual data types used to implement the primitive system data types.

2.3 Update the program in Figure 2.17 to avoid the needless processing that occurs when `sysconf` returns `LONG_MAX` as the limit for `OPEN_MAX`.

3

File I/O

3.1 Introduction

We'll start our discussion of the UNIX System by describing the functions available for file I/O—open a file, read a file, write a file, and so on. Most file I/O on a UNIX system can be performed using only five functions: `open`, `read`, `write`, `lseek`, and `close`. We then examine the effect of various buffer sizes on the `read` and `write` functions.

The functions described in this chapter are often referred to as *unbuffered I/O*, in contrast to the standard I/O routines, which we describe in Chapter 5. The term *unbuffered* means that each `read` or `write` invokes a system call in the kernel. These unbuffered I/O functions are not part of ISO C, but are part of POSIX.1 and the Single UNIX Specification.

Whenever we describe the sharing of resources among multiple processes, the concept of an atomic operation becomes important. We examine this concept with regard to file I/O and the arguments to the `open` function. This leads to a discussion of how files are shared among multiple processes and which kernel data structures are involved. After describing these features, we describe the `dup`, `fcntl`, `sync`, `fsync`, and `ioctl` functions.

3.2 File Descriptors

To the kernel, all open files are referred to by file descriptors. A file descriptor is a non-negative integer. When we open an existing file or create a new file, the kernel returns a file descriptor to the process. When we want to read or write a file, we identify the file with the file descriptor that was returned by `open` or `creat` as an argument to either `read` or `write`.

By convention, UNIX System shells associate file descriptor 0 with the standard input of a process, file descriptor 1 with the standard output, and file descriptor 2 with the standard error. This convention is used by the shells and many applications; it is not a feature of the UNIX kernel. Nevertheless, many applications would break if these associations weren't followed.

Although their values are standardized by POSIX.1, the magic numbers 0, 1, and 2 should be replaced in POSIX-compliant applications with the symbolic constants `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO` to improve readability. These constants are defined in the `<unistd.h>` header.

File descriptors range from 0 through `OPEN_MAX–1`. (Recall Figure 2.11.) Early historical implementations of the UNIX System had an upper limit of 19, allowing a maximum of 20 open files per process, but many systems subsequently increased this limit to 63.

> With FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10, the limit is essentially infinite, bounded by the amount of memory on the system, the size of an integer, and any hard and soft limits configured by the system administrator.

3.3 `open` and `openat` Functions

A file is opened or created by calling either the `open` function or the `openat` function.

```
#include <fcntl.h>

int open(const char *path, int oflag, ... /* mode_t mode */ );

int openat(int fd, const char *path, int oflag, ... /* mode_t mode */ );
```

Both return: file descriptor if OK, –1 on error

We show the last argument as `...`, which is the ISO C way to specify that the number and types of the remaining arguments may vary. For these functions, the last argument is used only when a new file is being created, as we describe later. We show this argument as a comment in the prototype.

The *path* parameter is the name of the file to open or create. This function has a multitude of options, which are specified by the *oflag* argument. This argument is formed by ORing together one or more of the following constants from the `<fcntl.h>` header:

`O_RDONLY` Open for reading only.

`O_WRONLY` Open for writing only.

`O_RDWR` Open for reading and writing.

> Most implementations define `O_RDONLY` as 0, `O_WRONLY` as 1, and `O_RDWR` as 2, for compatibility with older programs.

`O_EXEC` Open for execute only.

`O_SEARCH` Open for search only (applies to directories).

> The purpose of the O_SEARCH constant is to evaluate search permissions at the time a directory is opened. Further operations using the directory's file descriptor will not reevaluate permission to search the directory. None of the versions of the operating systems covered in this book support O_SEARCH yet.

One and only one of the previous five constants must be specified. The following constants are optional:

O_APPEND Append to the end of file on each write. We describe this option in detail in Section 3.11.

O_CLOEXEC Set the FD_CLOEXEC file descriptor flag. We discuss file descriptor flags in Section 3.14.

O_CREAT Create the file if it doesn't exist. This option requires a third argument to the open function (a fourth argument to the openat function)—the *mode*, which specifies the access permission bits of the new file. (When we describe a file's access permission bits in Section 4.5, we'll see how to specify the *mode* and how it can be modified by the umask value of a process.)

O_DIRECTORY Generate an error if *path* doesn't refer to a directory.

O_EXCL Generate an error if O_CREAT is also specified and the file already exists. This test for whether the file already exists and the creation of the file if it doesn't exist is an atomic operation. We describe atomic operations in more detail in Section 3.11.

O_NOCTTY If *path* refers to a terminal device, do not allocate the device as the controlling terminal for this process. We talk about controlling terminals in Section 9.6.

O_NOFOLLOW Generate an error if *path* refers to a symbolic link. We discuss symbolic links in Section 4.17.

O_NONBLOCK If *path* refers to a FIFO, a block special file, or a character special file, this option sets the nonblocking mode for both the opening of the file and subsequent I/O. We describe this mode in Section 14.2.

> In earlier releases of System V, the O_NDELAY (no delay) flag was introduced. This option is similar to the O_NONBLOCK (nonblocking) option, but an ambiguity was introduced in the return value from a read operation. The no-delay option causes a read operation to return 0 if there is no data to be read from a pipe, FIFO, or device, but this conflicts with a return value of 0, indicating an end of file. SVR4-based systems still support the no-delay option, with the old semantics, but new applications should use the nonblocking option instead.

O_SYNC Have each write wait for physical I/O to complete, including I/O necessary to update file attributes modified as a result of the write. We use this option in Section 3.14.

O_TRUNC If the file exists and if it is successfully opened for either write-only or read–write, truncate its length to 0.

`O_TTY_INIT` When opening a terminal device that is not already open, set the nonstandard `termios` parameters to values that result in behavior that conforms to the Single UNIX Specification. We discuss the `termios` structure when we discuss terminal I/O in Chapter 18.

The following two flags are also optional. They are part of the synchronized input and output option of the Single UNIX Specification (and thus POSIX.1).

`O_DSYNC` Have each `write` wait for physical I/O to complete, but don't wait for file attributes to be updated if they don't affect the ability to read the data just written.

> The `O_DSYNC` and `O_SYNC` flags are similar, but subtly different. The `O_DSYNC` flag affects a file's attributes only when they need to be updated to reflect a change in the file's data (for example, update the file's size to reflect more data). With the `O_SYNC` flag, data and attributes are always updated synchronously. When overwriting an existing part of a file opened with the `O_DSYNC` flag, the file times wouldn't be updated synchronously. In contrast, if we had opened the file with the `O_SYNC` flag, every `write` to the file would update the file's times before the `write` returns, regardless of whether we were writing over existing bytes or appending to the file.

`O_RSYNC` Have each `read` operation on the file descriptor wait until any pending writes for the same portion of the file are complete.

> Solaris 10 supports all three synchronization flags. Historically, FreeBSD (and thus Mac OS X) have used the `O_FSYNC` flag, which has the same behavior as `O_SYNC`. Because the two flags are equivalent, they define the flags to have the same value. FreeBSD 8.0 doesn't support the `O_DSYNC` or `O_RSYNC` flags. Mac OS X doesn't support the `O_RSYNC` flag, but defines the `O_DSYNC` flag, treating it the same as the `O_SYNC` flag. Linux 3.2.0 supports the `O_DSYNC` flag, but treats the `O_RSYNC` flag the same as `O_SYNC`.

The file descriptor returned by `open` and `openat` is guaranteed to be the lowest-numbered unused descriptor. This fact is used by some applications to open a new file on standard input, standard output, or standard error. For example, an application might close standard output—normally, file descriptor 1—and then open another file, knowing that it will be opened on file descriptor 1. We'll see a better way to guarantee that a file is open on a given descriptor in Section 3.12, when we explore the `dup2` function.

The *fd* parameter distinguishes the `openat` function from the `open` function. There are three possibilities:

1. The *path* parameter specifies an absolute pathname. In this case, the *fd* parameter is ignored and the `openat` function behaves like the `open` function.
2. The *path* parameter specifies a relative pathname and the *fd* parameter is a file descriptor that specifies the starting location in the file system where the relative pathname is to be evaluated. The *fd* parameter is obtained by opening the directory where the relative pathname is to be evaluated.

3. The *path* parameter specifies a relative pathname and the *fd* parameter has the special value AT_FDCWD. In this case, the pathname is evaluated starting in the current working directory and the openat function behaves like the open function.

The openat function is one of a class of functions added to the latest version of POSIX.1 to address two problems. First, it gives threads a way to use relative pathnames to open files in directories other than the current working directory. As we'll see in Chapter 11, all threads in the same process share the same current working directory, so this makes it difficult for multiple threads in the same process to work in different directories at the same time. Second, it provides a way to avoid time-of-check-to-time-of-use (TOCTTOU) errors.

The basic idea behind TOCTTOU errors is that a program is vulnerable if it makes two file-based function calls where the second call depends on the results of the first call. Because the two calls are not atomic, the file can change between the two calls, thereby invalidating the results of the first call, leading to a program error. TOCTTOU errors in the file system namespace generally deal with attempts to subvert file system permissions by tricking a privileged program into either reducing permissions on a privileged file or modifying a privileged file to open up a security hole. Wei and Pu [2005] discuss TOCTTOU weaknesses in the UNIX file system interface.

Filename and Pathname Truncation

What happens if NAME_MAX is 14 and we try to create a new file in the current directory with a filename containing 15 characters? Traditionally, early releases of System V, such as SVR2, allowed this to happen, silently truncating the filename beyond the 14th character. BSD-derived systems, in contrast, returned an error status, with errno set to ENAMETOOLONG. Silently truncating the filename presents a problem that affects more than simply the creation of new files. If NAME_MAX is 14 and a file exists whose name is exactly 14 characters, any function that accepts a pathname argument, such as open or stat, has no way to determine what the original name of the file was, as the original name might have been truncated.

With POSIX.1, the constant _POSIX_NO_TRUNC determines whether long filenames and long components of pathnames are truncated or an error is returned. As we saw in Chapter 2, this value can vary based on the type of the file system, and we can use fpathconf or pathconf to query a directory to see which behavior is supported.

> Whether an error is returned is largely historical. For example, SVR4-based systems do not generate an error for the traditional System V file system, S5. For the BSD-style file system (known as UFS), however, SVR4-based systems do generate an error. Figure 2.20 illustrates another example: Solaris will return an error for UFS, but not for PCFS, the DOS-compatible file system, as DOS silently truncates filenames that don't fit in an 8.3 format. BSD-derived systems and Linux always return an error.

If _POSIX_NO_TRUNC is in effect, errno is set to ENAMETOOLONG, and an error status is returned if any filename component of the pathname exceeds NAME_MAX.

> Most modern file systems support a maximum of 255 characters for filenames. Because filenames are usually shorter than this limit, this constraint tends to not present problems for most applications.

3.4 creat Function

A new file can also be created by calling the creat function.

```
#include <fcntl.h>

int creat(const char *path, mode_t mode);
```

Returns: file descriptor opened for write-only if OK, –1 on error

Note that this function is equivalent to

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

> Historically, in early versions of the UNIX System, the second argument to open could be only 0, 1, or 2. There was no way to open a file that didn't already exist. Therefore, a separate system call, creat, was needed to create new files. With the O_CREAT and O_TRUNC options now provided by open, a separate creat function is no longer needed.

We'll show how to specify *mode* in Section 4.5 when we describe a file's access permissions in detail.

One deficiency with creat is that the file is opened only for writing. Before the new version of open was provided, if we were creating a temporary file that we wanted to write and then read back, we had to call creat, close, and then open. A better way is to use the open function, as in

```
open(path, O_RDWR | O_CREAT | O_TRUNC, mode);
```

3.5 close Function

An open file is closed by calling the close function.

```
#include <unistd.h>

int close(int fd);
```

Returns: 0 if OK, –1 on error

Closing a file also releases any record locks that the process may have on the file. We'll discuss this point further in Section 14.3.

When a process terminates, all of its open files are closed automatically by the kernel. Many programs take advantage of this fact and don't explicitly close open files. See the program in Figure 1.4, for example.

3.6 lseek Function

Every open file has an associated "current file offset," normally a non-negative integer that measures the number of bytes from the beginning of the file. (We describe some exceptions to the "non-negative" qualifier later in this section.) Read and write operations normally start at the current file offset and cause the offset to be incremented by the number of bytes read or written. By default, this offset is initialized to 0 when a file is opened, unless the O_APPEND option is specified.

An open file's offset can be set explicitly by calling `lseek`.

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
                                    Returns: new file offset if OK, –1 on error
```

The interpretation of the *offset* depends on the value of the *whence* argument.

- If *whence* is `SEEK_SET`, the file's offset is set to *offset* bytes from the beginning of the file.
- If *whence* is `SEEK_CUR`, the file's offset is set to its current value plus the *offset*. The *offset* can be positive or negative.
- If *whence* is `SEEK_END`, the file's offset is set to the size of the file plus the *offset*. The *offset* can be positive or negative.

Because a successful call to `lseek` returns the new file offset, we can seek zero bytes from the current position to determine the current offset:

```
off_t    currpos;

currpos = lseek(fd, 0, SEEK_CUR);
```

This technique can also be used to determine if a file is capable of seeking. If the file descriptor refers to a pipe, FIFO, or socket, `lseek` sets `errno` to `ESPIPE` and returns –1.

> The three symbolic constants—`SEEK_SET`, `SEEK_CUR`, and `SEEK_END`—were introduced with System V. Prior to this, *whence* was specified as 0 (absolute), 1 (relative to the current offset), or 2 (relative to the end of file). Much software still exists with these numbers hard coded.
>
> The character `l` in the name `lseek` means "long integer." Before the introduction of the `off_t` data type, the *offset* argument and the return value were long integers. `lseek` was introduced with Version 7 when long integers were added to C. (Similar functionality was provided in Version 6 by the functions `seek` and `tell`.)

Example

The program in Figure 3.1 tests its standard input to see whether it is capable of seeking.

```
#include "apue.h"

int
main(void)
{
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("cannot seek\n");
    else
        printf("seek OK\n");
    exit(0);
}
```

Figure 3.1 Test whether standard input is capable of seeking

If we invoke this program interactively, we get

```
$ ./a.out < /etc/passwd
seek OK
$ cat < /etc/passwd | ./a.out
cannot seek
$ ./a.out < /var/spool/cron/FIFO
cannot seek
```

□

Normally, a file's current offset must be a non-negative integer. It is possible, however, that certain devices could allow negative offsets. But for regular files, the offset must be non-negative. Because negative offsets are possible, we should be careful to compare the return value from `lseek` as being equal to or not equal to –1, rather than testing whether it is less than 0.

> The `/dev/kmem` device on FreeBSD for the Intel x86 processor supports negative offsets.
>
> Because the offset (`off_t`) is a signed data type (Figure 2.21), we lose a factor of 2 in the maximum file size. If `off_t` is a 32-bit integer, the maximum file size is 2^{31}–1 bytes.

`lseek` only records the current file offset within the kernel—it does not cause any I/O to take place. This offset is then used by the next read or write operation.

The file's offset can be greater than the file's current size, in which case the next `write` to the file will extend the file. This is referred to as creating a hole in a file and is allowed. Any bytes in a file that have not been written are read back as 0.

A hole in a file isn't required to have storage backing it on disk. Depending on the file system implementation, when you write after seeking past the end of a file, new disk blocks might be allocated to store the data, but there is no need to allocate disk blocks for the data between the old end of file and the location where you start writing.

Example

The program shown in Figure 3.2 creates a file with a hole in it.

```
#include "apue.h"
#include <fcntl.h>

char    buf1[] = "abcdefghij";
char    buf2[] = "ABCDEFGHIJ";

int
main(void)
{
    int     fd;

    if ((fd = creat("file.hole", FILE_MODE)) < 0)
        err_sys("creat error");

    if (write(fd, buf1, 10) != 10)
        err_sys("buf1 write error");
    /* offset now = 10 */
```

```
    if (lseek(fd, 16384, SEEK_SET) == -1)
        err_sys("lseek error");
    /* offset now = 16384 */

    if (write(fd, buf2, 10) != 10)
        err_sys("buf2 write error");
    /* offset now = 16394 */

    exit(0);
}
```

Figure 3.2 Create a file with a hole in it

Running this program gives us

```
$ ./a.out
$ ls -l file.hole                    check its size
-rw-r--r--  1 sar       16394 Nov 25 01:01 file.hole
$ od -c file.hole                    let's look at the actual contents
0000000   a  b  c  d  e  f  g  h  i  j \0 \0 \0 \0 \0 \0
0000020  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
*
0040000   A  B  C  D  E  F  G  H  I  J
0040012
```

We use the od(1) command to look at the contents of the file. The `-c` flag tells it to print the contents as characters. We can see that the unwritten bytes in the middle are read back as zero. The seven-digit number at the beginning of each line is the byte offset in octal.

To prove that there is really a hole in the file, let's compare the file we just created with a file of the same size, but without holes:

```
$ ls -ls file.hole file.nohole     compare sizes
  8 -rw-r--r--  1 sar       16394 Nov 25 01:01 file.hole
 20 -rw-r--r--  1 sar       16394 Nov 25 01:03 file.nohole
```

Although both files are the same size, the file without holes consumes 20 disk blocks, whereas the file with holes consumes only 8 blocks.

In this example, we call the `write` function (Section 3.8). We'll have more to say about files with holes in Section 4.12. □

Because the offset address that `lseek` uses is represented by an `off_t`, implementations are allowed to support whatever size is appropriate on their particular platform. Most platforms today provide two sets of interfaces to manipulate file offsets: one set that uses 32-bit file offsets and another set that uses 64-bit file offsets.

The Single UNIX Specification provides a way for applications to determine which environments are supported through the `sysconf` function (Section 2.5.4). Figure 3.3 summarizes the `sysconf` constants that are defined.

Name of option	Description	*name* argument
`_POSIX_V7_ILP32_OFF32`	`int`, `long`, pointer, and `off_t` types are 32 bits.	`_SC_V7_ILP32_OFF32`
`_POSIX_V7_ILP32_OFFBIG`	`int`, `long`, and pointer types are 32 bits; `off_t` types are at least 64 bits.	`_SC_V7_ILP32_OFFBIG`
`_POSIX_V7_LP64_OFF64`	`int` types are 32 bits; `long`, pointer, and `off_t` types are 64 bits.	`_SC_V7_LP64_OFF64`
`_POSIX_V7_LP64_OFFBIG`	`int` types are at least 32 bits; `long`, pointer, and `off_t` types are at least 64 bits.	`_SC_V7_LP64_OFFBIG`

Figure 3.3 Data size options and *name* arguments to `sysconf`

The `c99` compiler requires that we use the `getconf`(1) command to map the desired data size model to the flags necessary to compile and link our programs. Different flags and libraries might be needed, depending on the environments supported by each platform.

Unfortunately, this is one area in which implementations haven't caught up to the standards. If your system does not match the latest version of the standard, the system might support the option names from the previous version of the Single UNIX Specification: `_POSIX_V6_ILP32_OFF32`, `_POSIX_V6_ILP32_OFFBIG`, `_POSIX_V6_LP64_OFF64`, and `_POSIX_V6_LP64_OFFBIG`.

To get around this, applications can set the `_FILE_OFFSET_BITS` constant to 64 to enable 64-bit offsets. Doing so changes the definition of `off_t` to be a 64-bit signed integer. Setting `_FILE_OFFSET_BITS` to 32 enables 32-bit file offsets. Be aware, however, that although all four platforms discussed in this text support both 32-bit and 64-bit file offsets, setting `_FILE_OFFSET_BITS` is not guaranteed to be portable and might not have the desired effect.

Figure 3.4 summarizes the size in bytes of the `off_t` data type for the platforms covered in this book when an application doesn't define `_FILE_OFFSET_BITS`, as well as the size when an application defines `_FILE_OFFSET_BITS` to have a value of either 32 or 64.

Operating system	CPU architecture	`_FILE_OFFSET_BITS` value		
		Undefined	32	64
FreeBSD 8.0	x86 32-bit	8	8	8
Linux 3.2.0	x86 64-bit	8	8	8
Mac OS X 10.6.8	x86 64-bit	8	8	8
Solaris 10	SPARC 64-bit	8	4	8

Figure 3.4 Size in bytes of `off_t` for different platforms

Note that even though you might enable 64-bit file offsets, your ability to create a file larger than 2 GB (2^{31}–1 bytes) depends on the underlying file system type.

3.7 read Function

Data is read from an open file with the read function.

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t nbytes);
```

Returns: number of bytes read, 0 if end of file, –1 on error

If the read is successful, the number of bytes read is returned. If the end of file is encountered, 0 is returned.

There are several cases in which the number of bytes actually read is less than the amount requested:

- When reading from a regular file, if the end of file is reached before the requested number of bytes has been read. For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).
- When reading from a terminal device. Normally, up to one line is read at a time. (We'll see how to change this default in Chapter 18.)
- When reading from a network. Buffering within the network may cause less than the requested amount to be returned.
- When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, read will return only what is available.
- When reading from a record-oriented device. Some record-oriented devices, such as magnetic tape, can return up to a single record at a time.
- When interrupted by a signal and a partial amount of data has already been read. We discuss this further in Section 10.5.

The read operation starts at the file's current offset. Before a successful return, the offset is incremented by the number of bytes actually read.

POSIX.1 changed the prototype for this function in several ways. The classic definition is

```
int read(int fd, char *buf, unsigned nbytes);
```

- First, the second argument was changed from char * to void * to be consistent with ISO C: the type void * is used for generic pointers.
- Next, the return value was required to be a signed integer (ssize_t) to return a positive byte count, 0 (for end of file), or –1 (for an error).
- Finally, the third argument historically has been an unsigned integer, to allow a 16-bit implementation to read or write up to 65,534 bytes at a time. With the 1990 POSIX.1 standard, the primitive system data type ssize_t was introduced to provide the signed return value, and the unsigned size_t was used for the third argument. (Recall the SSIZE_MAX constant from Section 2.5.2.)

3.8 write Function

Data is written to an open file with the write function.

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t nbytes);
```

Returns: number of bytes written if OK, –1 on error

The return value is usually equal to the *nbytes* argument; otherwise, an error has occurred. A common cause for a write error is either filling up a disk or exceeding the file size limit for a given process (Section 7.11 and Exercise 10.11).

For a regular file, the write operation starts at the file's current offset. If the O_APPEND option was specified when the file was opened, the file's offset is set to the current end of file before each write operation. After a successful write, the file's offset is incremented by the number of bytes actually written.

3.9 I/O Efficiency

The program in Figure 3.5 copies a file, using only the read and write functions.

```
#include "apue.h"

#define BUFFSIZE    4096

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

Figure 3.5 Copy standard input to standard output

The following caveats apply to this program.

- It reads from standard input and writes to standard output, assuming that these have been set up by the shell before this program is executed. Indeed, all normal UNIX system shells provide a way to open a file for reading on standard input and to create (or rewrite) a file on standard output. This prevents the program from having to open the input and output files, and allows the user to take advantage of the shell's I/O redirection facilities.

- The program doesn't close the input file or output file. Instead, the program uses the feature of the UNIX kernel that closes all open file descriptors in a process when that process terminates.
- This example works for both text files and binary files, since there is no difference between the two to the UNIX kernel.

One question we haven't answered, however, is how we chose the `BUFFSIZE` value. Before answering that, let's run the program using different values for `BUFFSIZE`. Figure 3.6 shows the results for reading a 516,581,760-byte file, using 20 different buffer sizes.

`BUFFSIZE`	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Number of loops
1	20.03	117.50	138.73	516,581,760
2	9.69	58.76	68.60	258,290,880
4	4.60	36.47	41.27	129,145,440
8	2.47	15.44	18.38	64,572,720
16	1.07	7.93	9.38	32,286,360
32	0.56	4.51	8.82	16,143,180
64	0.34	2.72	8.66	8,071,590
128	0.34	1.84	8.69	4,035,795
256	0.15	1.30	8.69	2,017,898
512	0.09	0.95	8.63	1,008,949
1,024	0.02	0.78	8.58	504,475
2,048	0.04	0.66	8.68	252,238
4,096	0.03	0.58	8.62	126,119
8,192	0.00	0.54	8.52	63,060
16,384	0.01	0.56	8.69	31,530
32,768	0.00	0.56	8.51	15,765
65,536	0.01	0.56	9.12	7,883
131,072	0.00	0.58	9.08	3,942
262,144	0.00	0.60	8.70	1,971
524,288	0.01	0.58	8.58	986

Figure 3.6 Timing results for reading with different buffer sizes on Linux

The file was read using the program shown in Figure 3.5, with standard output redirected to `/dev/null`. The file system used for this test was the Linux `ext4` file system with 4,096-byte blocks. (The `st_blksize` value, which we describe in Section 4.12, is 4,096.) This accounts for the minimum in the system time occurring at the few timing measurements starting around a `BUFFSIZE` of 4,096. Increasing the buffer size beyond this limit has little positive effect.

Most file systems support some kind of read-ahead to improve performance. When sequential reads are detected, the system tries to read in more data than an application requests, assuming that the application will read it shortly. The effect of read-ahead can be seen in Figure 3.6, where the elapsed time for buffer sizes as small as 32 bytes is as good as the elapsed time for larger buffer sizes.

We'll return to this timing example later in the text. In Section 3.14, we show the effect of synchronous writes; in Section 5.8, we compare these unbuffered I/O times with the standard I/O library.

> Beware when trying to measure the performance of programs that read and write files. The operating system will try to cache the file incore, so if you measure the performance of the program repeatedly, the successive timings will likely be better than the first. This improvement occurs because the first run causes the file to be entered into the system's cache, and successive runs access the file from the system's cache instead of from the disk. (The term *incore* means *in main memory*. Back in the day, a computer's main memory was built out of ferrite core. This is where the phrase "core dump" comes from: the main memory image of a program stored in a file on disk for diagnosis.)
>
> In the tests reported in Figure 3.6, each run with a different buffer size was made using a different copy of the file so that the current run didn't find the data in the cache from the previous run. The files are large enough that they all don't remain in the cache (the test system was configured with 6 GB of RAM).

3.10 File Sharing

The UNIX System supports the sharing of open files among different processes. Before describing the dup function, we need to describe this sharing. To do this, we'll examine the data structures used by the kernel for all I/O.

> The following description is conceptual; it may or may not match a particular implementation. Refer to Bach [1986] for a discussion of these structures in System V. McKusick et al. [1996] describe these structures in 4.4BSD. McKusick and Neville-Neil [2005] cover FreeBSD 5.2. For a similar discussion of Solaris, see McDougall and Mauro [2007]. The Linux 2.6 kernel architecture is discussed in Bovet and Cesati [2006].

The kernel uses three data structures to represent an open file, and the relationships among them determine the effect one process has on another with regard to file sharing.

1. Every process has an entry in the process table. Within each process table entry is a table of open file descriptors, which we can think of as a vector, with one entry per descriptor. Associated with each file descriptor are
   (a) The file descriptor flags (close-on-exec; refer to Figure 3.7 and Section 3.14)
   (b) A pointer to a file table entry
2. The kernel maintains a file table for all open files. Each file table entry contains
   (a) The file status flags for the file, such as read, write, append, sync, and nonblocking; more on these in Section 3.14
   (b) The current file offset
   (c) A pointer to the v-node table entry for the file
3. Each open file (or device) has a v-node structure that contains information about the type of file and pointers to functions that operate on the file. For most files, the

v-node also contains the i-node for the file. This information is read from disk when the file is opened, so that all the pertinent information about the file is readily available. For example, the i-node contains the owner of the file, the size of the file, pointers to where the actual data blocks for the file are located on disk, and so on. (We talk more about i-nodes in Section 4.14 when we describe the typical UNIX file system in more detail.)

> Linux has no v-node. Instead, a generic i-node structure is used. Although the implementations differ, the v-node is conceptually the same as a generic i-node. Both point to an i-node structure specific to the file system.

We're ignoring some implementation details that don't affect our discussion. For example, the table of open file descriptors can be stored in the user area (a separate per-process structure that can be paged out) instead of the process table. Also, these tables can be implemented in numerous ways—they need not be arrays; one alternate implementation is a linked lists of structures. Regardless of the implementation details, the general concepts remain the same.

Figure 3.7 shows a pictorial arrangement of these three tables for a single process that has two different files open: one file is open on standard input (file descriptor 0), and the other is open on standard output (file descriptor 1).

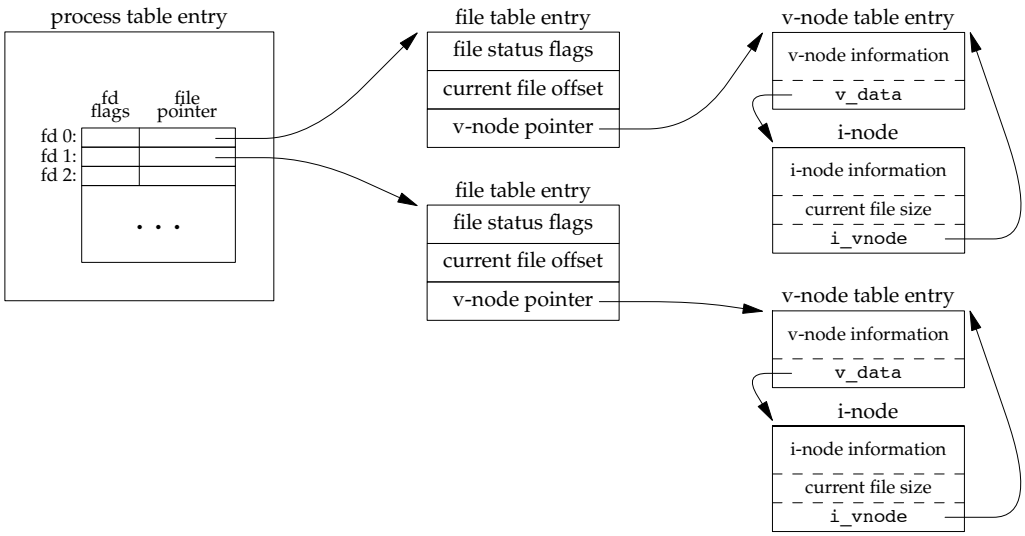


Figure 3.7 Kernel data structures for open files

The arrangement of these three tables has existed since the early versions of the UNIX System [Thompson 1978]. This arrangement is critical to the way files are shared among processes. We'll return to this figure in later chapters, when we describe additional ways that files are shared.

The v-node was invented to provide support for multiple file system types on a single computer system. This work was done independently by Peter Weinberger (Bell Laboratories) and Bill Joy (Sun Microsystems). Sun called this the Virtual File System and called the file system–independent portion of the i-node the v-node [Kleiman 1986]. The v-node propagated through various vendor implementations as support for Sun's Network File System (NFS) was added. The first release from Berkeley to provide v-nodes was the 4.3BSD Reno release, when NFS was added.

In SVR4, the v-node replaced the file system–independent i-node of SVR3. Solaris is derived from SVR4 and, therefore, uses v-nodes.

Instead of splitting the data structures into a v-node and an i-node, Linux uses a file system–independent i-node and a file system–dependent i-node.

If two independent processes have the same file open, we could have the arrangement shown in Figure 3.8.

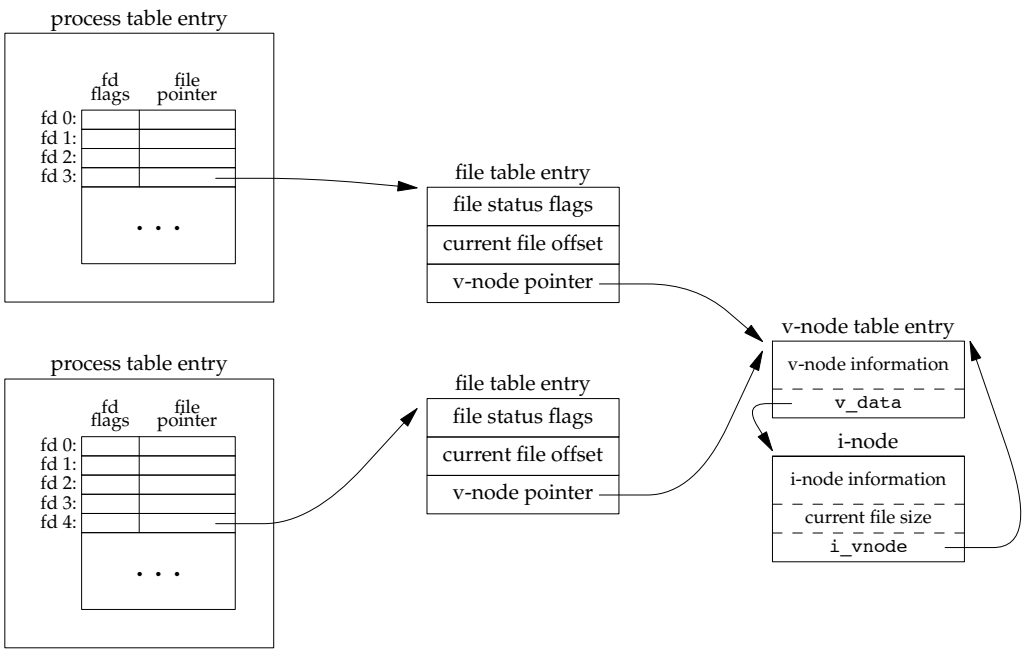


Figure 3.8 Two independent processes with the same file open

We assume here that the first process has the file open on descriptor 3 and that the second process has that same file open on descriptor 4. Each process that opens the file gets its own file table entry, but only a single v-node table entry is required for a given file. One reason each process gets its own file table entry is so that each process has its own current offset for the file.

Given these data structures, we now need to be more specific about what happens with certain operations that we've already described.

- After each `write` is complete, the current file offset in the file table entry is incremented by the number of bytes written. If this causes the current file offset to exceed the current file size, the current file size in the i-node table entry is set to the current file offset (for example, the file is extended).
- If a file is opened with the `O_APPEND` flag, a corresponding flag is set in the file status flags of the file table entry. Each time a `write` is performed for a file with this append flag set, the current file offset in the file table entry is first set to the current file size from the i-node table entry. This forces every `write` to be appended to the current end of file.
- If a file is positioned to its current end of file using `lseek`, all that happens is the current file offset in the file table entry is set to the current file size from the i-node table entry. (Note that this is not the same as if the file was opened with the `O_APPEND` flag, as we will see in Section 3.11.)
- The `lseek` function modifies only the current file offset in the file table entry. No I/O takes place.

It is possible for more than one file descriptor entry to point to the same file table entry, as we'll see when we discuss the `dup` function in Section 3.12. This also happens after a `fork` when the parent and the child share the same file table entry for each open descriptor (Section 8.3).

Note the difference in scope between the file descriptor flags and the file status flags. The former apply only to a single descriptor in a single process, whereas the latter apply to all descriptors in any process that point to the given file table entry. When we describe the `fcntl` function in Section 3.14, we'll see how to fetch and modify both the file descriptor flags and the file status flags.

Everything that we've described so far in this section works fine for multiple processes that are reading the same file. Each process has its own file table entry with its own current file offset. Unexpected results can arise, however, when multiple processes write to the same file. To see how to avoid some surprises, we need to understand the concept of atomic operations.

3.11 Atomic Operations

Appending to a File

Consider a single process that wants to append to the end of a file. Older versions of the UNIX System didn't support the `O_APPEND` option to `open`, so the program was coded as follows:

```
if (lseek(fd, 0L, 2) < 0)          /* position to EOF */
   err_sys("lseek error");
if (write(fd, buf, 100) != 100)    /* and write */
   err_sys("write error");
```

This works fine for a single process, but problems arise if multiple processes use this technique to append to the same file. (This scenario can arise if multiple instances of the same program are appending messages to a log file, for example.)

Assume that two independent processes, A and B, are appending to the same file. Each has opened the file but *without* the `O_APPEND` flag. This gives us the same picture as Figure 3.8. Each process has its own file table entry, but they share a single v-node table entry. Assume that process A does the `lseek` and that this sets the current offset for the file for process A to byte offset 1,500 (the current end of file). Then the kernel switches processes, and B continues running. Process B then does the `lseek`, which sets the current offset for the file for process B to byte offset 1,500 also (the current end of file). Then B calls `write`, which increments B's current file offset for the file to 1,600. Because the file's size has been extended, the kernel also updates the current file size in the v-node to 1,600. Then the kernel switches processes and A resumes. When A calls `write`, the data is written starting at the current file offset for A, which is byte offset 1,500. This overwrites the data that B wrote to the file.

The problem here is that our logical operation of "position to the end of file and write" requires two separate function calls (as we've shown it). The solution is to have the positioning to the current end of file and the write be an atomic operation with regard to other processes. Any operation that requires more than one function call cannot be atomic, as there is always the possibility that the kernel might temporarily suspend the process between the two function calls (as we assumed previously).

The UNIX System provides an atomic way to do this operation if we set the `O_APPEND` flag when a file is opened. As we described in the previous section, this causes the kernel to position the file to its current end of file before each `write`. We no longer have to call `lseek` before each `write`.

`pread` and `pwrite` Functions

The Single UNIX Specification includes two functions that allow applications to seek and perform I/O atomically: `pread` and `pwrite`.

```
#include <unistd.h>

ssize_t pread(int fd, void *buf, size_t nbytes, off_t offset);
                                Returns: number of bytes read, 0 if end of file, –1 on error

ssize_t pwrite(int fd, const void *buf, size_t nbytes, off_t offset);
                                Returns: number of bytes written if OK, –1 on error
```

Calling `pread` is equivalent to calling `lseek` followed by a call to `read`, with the following exceptions.

- There is no way to interrupt the two operations that occur when we call `pread`.
- The current file offset is not updated.

Calling pwrite is equivalent to calling lseek followed by a call to write, with similar exceptions.

Creating a File

We saw another example of an atomic operation when we described the O_CREAT and O_EXCL options for the open function. When both of these options are specified, the open will fail if the file already exists. We also said that the check for the existence of the file and the creation of the file was performed as an atomic operation. If we didn't have this atomic operation, we might try

```
if ((fd = open(path, O_WRONLY)) < 0) {
    if (errno == ENOENT) {
        if ((fd = creat(path, mode)) < 0)
            err_sys("creat error");
    } else {
        err_sys("open error");
    }
}
```

The problem occurs if the file is created by another process between the open and the creat. If the file is created by another process between these two function calls, and if that other process writes something to the file, that data is erased when this creat is executed. Combining the test for existence and the creation into a single atomic operation avoids this problem.

In general, the term *atomic operation* refers to an operation that might be composed of multiple steps. If the operation is performed atomically, either all the steps are performed (on success) or none are performed (on failure). It must not be possible for only a subset of the steps to be performed. We'll return to the topic of atomic operations when we describe the link function (Section 4.15) and record locking (Section 14.3).

3.12 dup and dup2 Functions

An existing file descriptor is duplicated by either of the following functions:

```
#include <unistd.h>

int dup(int fd);

int dup2(int fd, int fd2);
```

Both return: new file descriptor if OK, –1 on error

The new file descriptor returned by dup is guaranteed to be the lowest-numbered available file descriptor. With dup2, we specify the value of the new descriptor with the *fd2* argument. If *fd2* is already open, it is first closed. If *fd* equals *fd2*, then dup2 returns *fd2* without closing it. Otherwise, the FD_CLOEXEC file descriptor flag is cleared for *fd2*, so that *fd2* is left open if the process calls exec.

The new file descriptor that is returned as the value of the functions shares the same file table entry as the *fd* argument. We show this in Figure 3.9.

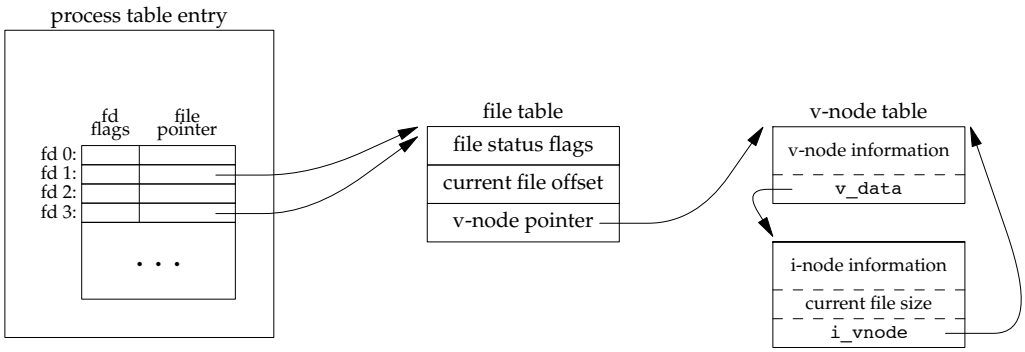


Figure 3.9 Kernel data structures after `dup(1)`

In this figure, we assume that when it's started, the process executes

```
newfd = dup(1);
```

We assume that the next available descriptor is 3 (which it probably is, since 0, 1, and 2 are opened by the shell). Because both descriptors point to the same file table entry, they share the same file status flags—read, write, append, and so on—and the same current file offset.

Each descriptor has its own set of file descriptor flags. As we describe in Section 3.14, the close-on-exec file descriptor flag for the new descriptor is always cleared by the dup functions.

Another way to duplicate a descriptor is with the `fcntl` function, which we describe in Section 3.14. Indeed, the call

```
dup(fd);
```

is equivalent to

```
fcntl(fd, F_DUPFD, 0);
```

Similarly, the call

```
dup2(fd, fd2);
```

is equivalent to

```
close(fd2);
fcntl(fd, F_DUPFD, fd2);
```

In this last case, the `dup2` is not exactly the same as a `close` followed by an `fcntl`. The differences are as follows:

1. dup2 is an atomic operation, whereas the alternate form involves two function calls. It is possible in the latter case to have a signal catcher called between the close and the fcntl that could modify the file descriptors. (We describe signals in Chapter 10.) The same problem could occur if a different thread changes the file descriptors. (We describe threads in Chapter 11.)

2. There are some errno differences between dup2 and fcntl.

The dup2 system call originated with Version 7 and propagated through the BSD releases. The fcntl method for duplicating file descriptors appeared with System III and continued with System V. SVR3.2 picked up the dup2 function, and 4.2BSD picked up the fcntl function and the F_DUPFD functionality. POSIX.1 requires both dup2 and the F_DUPFD feature of fcntl.

3.13 sync, fsync, and fdatasync Functions

Traditional implementations of the UNIX System have a buffer cache or page cache in the kernel through which most disk I/O passes. When we write data to a file, the data is normally copied by the kernel into one of its buffers and queued for writing to disk at some later time. This is called *delayed write*. (Chapter 3 of Bach [1986] discusses this buffer cache in detail.)

The kernel eventually writes all the delayed-write blocks to disk, normally when it needs to reuse the buffer for some other disk block. To ensure consistency of the file system on disk with the contents of the buffer cache, the sync, fsync, and fdatasync functions are provided.

```
#include <unistd.h>

int fsync(int fd);

int fdatasync(int fd);
                                                    Returns: 0 if OK, –1 on error

void sync(void);
```

The sync function simply queues all the modified block buffers for writing and returns; it does not wait for the disk writes to take place.

The function sync is normally called periodically (usually every 30 seconds) from a system daemon, often called update. This guarantees regular flushing of the kernel's block buffers. The command sync(1) also calls the sync function.

The function fsync refers only to a single file, specified by the file descriptor *fd*, and waits for the disk writes to complete before returning. This function is used when an application, such as a database, needs to be sure that the modified blocks have been written to the disk.

The fdatasync function is similar to fsync, but it affects only the data portions of a file. With fsync, the file's attributes are also updated synchronously.

All four of the platforms described in this book support sync and fsync. However, FreeBSD 8.0 does not support fdatasync.

3.14 fcntl Function

The fcntl function can change the properties of a file that is already open.

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* int arg */ );
```

Returns: depends on *cmd* if OK (see following), –1 on error

In the examples in this section, the third argument is always an integer, corresponding to the comment in the function prototype just shown. When we describe record locking in Section 14.3, however, the third argument becomes a pointer to a structure.

The fcntl function is used for five different purposes.

1. Duplicate an existing descriptor (*cmd* = F_DUPFD or F_DUPFD_CLOEXEC)
2. Get/set file descriptor flags (*cmd* = F_GETFD or F_SETFD)
3. Get/set file status flags (*cmd* = F_GETFL or F_SETFL)
4. Get/set asynchronous I/O ownership (*cmd* = F_GETOWN or F_SETOWN)
5. Get/set record locks (*cmd* = F_GETLK, F_SETLK, or F_SETLKW)

We'll now describe the first 8 of these 11 *cmd* values. (We'll wait until Section 14.3 to describe the last 3, which deal with record locking.) Refer to Figure 3.7, as we'll discuss both the file descriptor flags associated with each file descriptor in the process table entry and the file status flags associated with each file table entry.

F_DUPFD — Duplicate the file descriptor *fd*. The new file descriptor is returned as the value of the function. It is the lowest-numbered descriptor that is not already open, and that is greater than or equal to the third argument (taken as an integer). The new descriptor shares the same file table entry as *fd*. (Refer to Figure 3.9.) But the new descriptor has its own set of file descriptor flags, and its FD_CLOEXEC file descriptor flag is cleared. (This means that the descriptor is left open across an exec, which we discuss in Chapter 8.)

F_DUPFD_CLOEXEC — Duplicate the file descriptor and set the FD_CLOEXEC file descriptor flag associated with the new descriptor. Returns the new file descriptor.

F_GETFD — Return the file descriptor flags for *fd* as the value of the function. Currently, only one file descriptor flag is defined: the FD_CLOEXEC flag.

F_SETFD — Set the file descriptor flags for *fd*. The new flag value is set from the third argument (taken as an integer).

Be aware that some existing programs that deal with the file descriptor flags don't use the constant FD_CLOEXEC. Instead, these programs set the flag to either 0 (don't close-on-exec, the default) or 1 (do close-on-exec).

F_GETFL Return the file status flags for *fd* as the value of the function. We described the file status flags when we described the open function. They are listed in Figure 3.10.

File status flag	Description
O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_EXEC	open for execute only
O_SEARCH	open directory for searching only
O_APPEND	append on each write
O_NONBLOCK	nonblocking mode
O_SYNC	wait for writes to complete (data and attributes)
O_DSYNC	wait for writes to complete (data only)
O_RSYNC	synchronize reads and writes
O_FSYNC	wait for writes to complete (FreeBSD and Mac OS X only)
O_ASYNC	asynchronous I/O (FreeBSD and Mac OS X only)

Figure 3.10 File status flags for fcntl

Unfortunately, the five access-mode flags—O_RDONLY, O_WRONLY, O_RDWR, O_EXEC, and O_SEARCH—are not separate bits that can be tested. (As we mentioned earlier, the first three often have the values 0, 1, and 2, respectively, for historical reasons. Also, these five values are mutually exclusive; a file can have only one of them enabled.) Therefore, we must first use the O_ACCMODE mask to obtain the access-mode bits and then compare the result against any of the five values.

F_SETFL Set the file status flags to the value of the third argument (taken as an integer). The only flags that can be changed are O_APPEND, O_NONBLOCK, O_SYNC, O_DSYNC, O_RSYNC, O_FSYNC, and O_ASYNC.

F_GETOWN Get the process ID or process group ID currently receiving the SIGIO and SIGURG signals. We describe these asynchronous I/O signals in Section 14.5.2.

F_SETOWN Set the process ID or process group ID to receive the SIGIO and SIGURG signals. A positive *arg* specifies a process ID. A negative *arg* implies a process group ID equal to the absolute value of *arg*.

The return value from fcntl depends on the command. All commands return –1 on an error or some other value if OK. The following four commands have special return values: F_DUPFD, F_GETFD, F_GETFL, and F_GETOWN. The first command returns the new file descriptor, the next two return the corresponding flags, and the final command returns a positive process ID or a negative process group ID.

Example

The program in Figure 3.11 takes a single command-line argument that specifies a file descriptor and prints a description of selected file flags for that descriptor.

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
{
    int     val;

    if (argc != 2)
        err_quit("usage: a.out <descriptor#>");

    if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
        err_sys("fcntl error for fd %d", atoi(argv[1]));

    switch (val & O_ACCMODE) {
    case O_RDONLY:
        printf("read only");
        break;

    case O_WRONLY:
        printf("write only");
        break;

    case O_RDWR:
        printf("read write");
        break;

    default:
        err_dump("unknown access mode");
    }

    if (val & O_APPEND)
        printf(", append");
    if (val & O_NONBLOCK)
        printf(", nonblocking");
    if (val & O_SYNC)
        printf(", synchronous writes");

#if !defined(_POSIX_C_SOURCE) && defined(O_FSYNC) && (O_FSYNC != O_SYNC)
    if (val & O_FSYNC)
        printf(", synchronous writes");
#endif

    putchar('\n');
    exit(0);
}
```

Figure 3.11 Print file flags for specified descriptor

Note that we use the feature test macro `_POSIX_C_SOURCE` and conditionally compile the file access flags that are not part of POSIX.1. The following script shows the

operation of the program, when invoked from bash (the Bourne-again shell). Results will vary, depending on which shell you use.

```
$ ./a.out 0 < /dev/tty
read only
$ ./a.out 1 > temp.foo
$ cat temp.foo
write only
$ ./a.out 2 2>>temp.foo
write only, append
$ ./a.out 5 5<>temp.foo
read write
```

The clause 5<>temp.foo opens the file temp.foo for reading and writing on file descriptor 5. □

Example

When we modify either the file descriptor flags or the file status flags, we must be careful to fetch the existing flag value, modify it as desired, and then set the new flag value. We can't simply issue an F_SETFD or an F_SETFL command, as this could turn off flag bits that were previously set.

Figure 3.12 shows a function that sets one or more of the file status flags for a descriptor.

```
#include "apue.h"
#include <fcntl.h>

void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
    int     val;

    if ((val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("fcntl F_GETFL error");

    val |= flags;       /* turn on flags */

    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("fcntl F_SETFL error");
}
```

Figure 3.12 Turn on one or more of the file status flags for a descriptor

If we change the middle statement to

```
val &= ~flags;      /* turn flags off */
```

we have a function named clr_fl, which we'll use in some later examples. This statement logically ANDs the one's complement of flags with the current val.

If we add the line

```
set_fl(STDOUT_FILENO, O_SYNC);
```

to the beginning of the program shown in Figure 3.5, we'll turn on the synchronous-write flag. This causes each `write` to wait for the data to be written to disk before returning. Normally in the UNIX System, a `write` only queues the data for writing; the actual disk write operation can take place sometime later. A database system is a likely candidate for using `O_SYNC`, so that it knows on return from a `write` that the data is actually on the disk, in case of an abnormal system failure.

We expect the `O_SYNC` flag to increase the system and clock times when the program runs. To test this, we can run the program in Figure 3.5, copying 492.6 MB of data from one file on disk to another and compare this with a version that does the same thing with the `O_SYNC` flag set. The results from a Linux system using the `ext4` file system are shown in Figure 3.13.

Operation	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)
read time from Figure 3.6 for `BUFFSIZE` = 4,096	0.03	0.58	8.62
normal `write` to disk file	0.00	1.05	9.70
`write` to disk file with `O_SYNC` set	0.02	1.09	10.28
`write` to disk followed by `fdatasync`	0.02	1.14	17.93
`write` to disk followed by `fsync`	0.00	1.19	18.17
`write` to disk with `O_SYNC` set followed by `fsync`	0.02	1.15	17.88

Figure 3.13 Linux `ext4` timing results using various synchronization mechanisms

The six rows in Figure 3.13 were all measured with a `BUFFSIZE` of 4,096 bytes. The results in Figure 3.6 were measured while reading a disk file and writing to `/dev/null`, so there was no disk output. The second row in Figure 3.13 corresponds to reading a disk file and writing to another disk file. This is why the first and second rows in Figure 3.13 are different. The system time increases when we write to a disk file, because the kernel now copies the data from our process and queues the data for writing by the disk driver. We expect the clock time to increase as well when we write to a disk file.

When we enable synchronous writes, the system and clock times should increase significantly. As the third row shows, the system time for writing synchronously is not much more expensive than when we used delayed writes. This implies that the Linux operating system is doing the same amount of work for delayed and synchronous writes (which is unlikely), or else the `O_SYNC` flag isn't having the desired effect. In this case, the Linux operating system isn't allowing us to set the `O_SYNC` flag using `fcntl`, instead failing without returning an error (but it would have honored the flag if we were able to specify it when the file was opened).

The clock time in the last three rows reflects the extra time needed to wait for all of the writes to be committed to disk. After writing a file synchronously, we expect that a call to `fsync` will have no effect. This case is supposed to be represented by the last

row in Figure 3.13, but since the O_SYNC flag isn't having the intended effect, the last row behaves the same way as the fifth row.

Figure 3.14 shows timing results for the same tests run on Mac OS X 10.6.8, which uses the HFS file system. Note that the times match our expectations: synchronous writes are far more expensive than delayed writes, and using fsync with synchronous writes makes very little difference. Note also that adding a call to fsync at the end of the delayed writes makes little measurable difference. It is likely that the operating system flushed previously written data to disk as we were writing new data to the file, so by the time that we called fsync, very little work was left to be done.

Operation	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)
write to /dev/null	0.14	1.02	5.28
normal write to disk file	0.14	3.21	17.04
write to disk file with O_SYNC set	0.39	16.89	60.82
write to disk followed by fsync	0.13	3.07	17.10
write to disk with O_SYNC set followed by fsync	0.39	18.18	62.39

Figure 3.14 Mac OS X HFS timing results using various synchronization mechanisms

Compare fsync and fdatasync, both of which update a file's contents when we say so, with the O_SYNC flag, which updates a file's contents every time we write to the file. The performance of each alternative will depend on many factors, including the underlying operating system implementation, the speed of the disk drive, and the type of the file system. □

With this example, we see the need for fcntl. Our program operates on a descriptor (standard output), never knowing the name of the file that was opened on that descriptor. We can't set the O_SYNC flag when the file is opened, since the shell opened the file. With fcntl, we can modify the properties of a descriptor, knowing only the descriptor for the open file. We'll see another need for fcntl when we describe nonblocking pipes (Section 15.2), since all we have with a pipe is a descriptor.

3.15 ioctl Function

The ioctl function has always been the catchall for I/O operations. Anything that couldn't be expressed using one of the other functions in this chapter usually ended up being specified with an ioctl. Terminal I/O was the biggest user of this function. (When we get to Chapter 18, we'll see that POSIX.1 has replaced the terminal I/O operations with separate functions.)

```
#include <unistd.h>     /* System V */
#include <sys/ioctl.h>  /* BSD and Linux */

int ioctl(int fd, int request, ...);
```

Returns: –1 on error, something else if OK

The `ioctl` function was included in the Single UNIX Specification only as an extension for dealing with STREAMS devices [Rago 1993], but it was moved to obsolescent status in SUSv4. UNIX System implementations use `ioctl` for many miscellaneous device operations. Some implementations have even extended it for use with regular files.

The prototype that we show corresponds to POSIX.1. FreeBSD 8.0 and Mac OS X 10.6.8 declare the second argument as an `unsigned long`. This detail doesn't matter, since the second argument is always a `#defined` name from a header.

For the ISO C prototype, an ellipsis is used for the remaining arguments. Normally, however, there is only one more argument, and it's usually a pointer to a variable or a structure.

In this prototype, we show only the headers required for the function itself. Normally, additional device-specific headers are required. For example, the `ioctl` commands for terminal I/O, beyond the basic operations specified by POSIX.1, all require the `<termios.h>` header.

Each device driver can define its own set of `ioctl` commands. The system, however, provides generic `ioctl` commands for different classes of devices. Examples of some of the categories for these generic `ioctl` commands supported in FreeBSD are summarized in Figure 3.15.

Category	Constant names	Header	Number of `ioctl`s
disk labels	`DIOxxx`	`<sys/disklabel.h>`	4
file I/O	`FIOxxx`	`<sys/filio.h>`	14
mag tape I/O	`MTIOxxx`	`<sys/mtio.h>`	11
socket I/O	`SIOxxx`	`<sys/sockio.h>`	73
terminal I/O	`TIOxxx`	`<sys/ttycom.h>`	43

Figure 3.15 Common FreeBSD `ioctl` operations

The mag tape operations allow us to write end-of-file marks on a tape, rewind a tape, space forward over a specified number of files or records, and the like. None of these operations is easily expressed in terms of the other functions in the chapter (`read`, `write`, `lseek`, and so on), so the easiest way to handle these devices has always been to access their operations using `ioctl`.

We use the `ioctl` function in Section 18.12 to fetch and set the size of a terminal's window, and in Section 19.7 when we access the advanced features of pseudo terminals.

3.16 `/dev/fd`

Newer systems provide a directory named `/dev/fd` whose entries are files named `0`, `1`, `2`, and so on. Opening the file `/dev/fd/`*n* is equivalent to duplicating descriptor *n*, assuming that descriptor *n* is open.

The `/dev/fd` feature was developed by Tom Duff and appeared in the 8th Edition of the Research UNIX System. It is supported by all of the systems described in this book: FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10. It is not part of POSIX.1.

In the function call

```
fd = open("/dev/fd/0", mode);
```

most systems ignore the specified `mode`, whereas others require that it be a subset of the mode used when the referenced file (standard input, in this case) was originally opened. Because the previous `open` is equivalent to

```
fd = dup(0);
```

the descriptors 0 and `fd` share the same file table entry (Figure 3.9). For example, if descriptor 0 was opened read-only, we can only read on `fd`. Even if the system ignores the open mode and the call

```
fd = open("/dev/fd/0", O_RDWR);
```

succeeds, we still can't write to `fd`.

> The Linux implementation of `/dev/fd` is an exception. It maps file descriptors into symbolic links pointing to the underlying physical files. When you open `/dev/fd/0`, for example, you are really opening the file associated with your standard input. Thus the mode of the new file descriptor returned is unrelated to the mode of the `/dev/fd` file descriptor.

We can also call `creat` with a `/dev/fd` pathname argument as well as specify `O_CREAT` in a call to `open`. This allows a program that calls `creat` to still work if the pathname argument is `/dev/fd/1`, for example.

> Beware of doing this on Linux. Because the Linux implementation uses symbolic links to the real files, using `creat` on a `/dev/fd` file will result in the underlying file being truncated.

Some systems provide the pathnames `/dev/stdin`, `/dev/stdout`, and `/dev/stderr`. These pathnames are equivalent to `/dev/fd/0`, `/dev/fd/1`, and `/dev/fd/2`, respectively.

The main use of the `/dev/fd` files is from the shell. It allows programs that use pathname arguments to handle standard input and standard output in the same manner as other pathnames. For example, the `cat`(1) program specifically looks for an input filename of `-` and uses it to mean standard input. The command

```
filter file2 | cat file1 - file3 | lpr
```

is an example. First, `cat` reads `file1`, then its standard input (the output of the `filter` program on `file2`), and then `file3`. If `/dev/fd` is supported, the special handling of `-` can be removed from `cat`, and we can enter

```
filter file2 | cat file1 /dev/fd/0 file3 | lpr
```

The special meaning of `-` as a command-line argument to refer to the standard input or the standard output is a kludge that has crept into many programs. There are also problems if we specify `-` as the first file, as it looks like the start of another command-line option. Using `/dev/fd` is a step toward uniformity and cleanliness.

3.17 Summary

This chapter has described the basic I/O functions provided by the UNIX System. These are often called the unbuffered I/O functions because each `read` or `write` invokes a system call into the kernel. Using only `read` and `write`, we looked at the effect of various I/O sizes on the amount of time required to read a file. We also looked at several ways to flush written data to disk and their effect on application performance.

Atomic operations were introduced when multiple processes append to the same file and when multiple processes create the same file. We also looked at the data structures used by the kernel to share information about open files. We'll return to these data structures later in the text.

We also described the `ioctl` and `fcntl` functions. We return to both of these functions later in the book. In Chapter 14, we'll use `fcntl` for record locking. In Chapter 18 and Chapter 19, we'll use `ioctl` when we deal with terminal devices.

Exercises

3.1 When reading or writing a disk file, are the functions described in this chapter really unbuffered? Explain.

3.2 Write your own `dup2` function that behaves the same way as the `dup2` function described in Section 3.12, without calling the `fcntl` function. Be sure to handle errors correctly.

3.3 Assume that a process executes the following three function calls:

```
fd1 = open(path, oflags);
fd2 = dup(fd1);
fd3 = open(path, oflags);
```

Draw the resulting picture, similar to Figure 3.9. Which descriptors are affected by an `fcntl` on `fd1` with a command of `F_SETFD`? Which descriptors are affected by an `fcntl` on `fd1` with a command of `F_SETFL`?

3.4 The following sequence of code has been observed in various programs:

```
dup2(fd, 0);
dup2(fd, 1);
dup2(fd, 2);
if (fd > 2)
    close(fd);
```

To see why the `if` test is needed, assume that `fd` is 1 and draw a picture of what happens to the three descriptor entries and the corresponding file table entry with each call to `dup2`. Then assume that `fd` is 3 and draw the same picture.

3.5 The Bourne shell, Bourne-again shell, and Korn shell notation

digit1`>&`*digit2*

says to redirect descriptor *digit1* to the same file as descriptor *digit2*. What is the difference between the two commands shown below? (Hint: The shells process their command lines from left to right.)

```
./a.out > outfile 2>&1
./a.out 2>&1 > outfile
```

3.6 If you `open` a file for read–write with the append flag, can you still `read` from anywhere in the file using `lseek`? Can you use `lseek` to replace existing data in the file? Write a program to verify this.

This page intentionally left blank

4

Files and Directories

4.1 Introduction

In the previous chapter we covered the basic functions that perform I/O. The discussion centered on I/O for regular files—opening a file, and reading or writing a file. We'll now look at additional features of the file system and the properties of a file. We'll start with the `stat` functions and go through each member of the `stat` structure, looking at all the attributes of a file. In this process, we'll also describe each of the functions that modify these attributes: change the owner, change the permissions, and so on. We'll also look in more detail at the structure of a UNIX file system and symbolic links. We finish this chapter with the functions that operate on directories, and we develop a function that descends through a directory hierarchy.

4.2 `stat`, `fstat`, `fstatat`, and `lstat` Functions

The discussion in this chapter centers on the four `stat` functions and the information they return.

```
#include <sys/stat.h>

int stat(const char *restrict pathname, struct stat *restrict buf);

int fstat(int fd, struct stat *buf);

int lstat(const char *restrict pathname, struct stat *restrict buf);

int fstatat(int fd, const char *restrict pathname,
            struct stat *restrict buf, int flag);
```

All four return: 0 if OK, –1 on error

Given a *pathname*, the `stat` function returns a structure of information about the named file. The `fstat` function obtains information about the file that is already open on the descriptor *fd*. The `lstat` function is similar to `stat`, but when the named file is a symbolic link, `lstat` returns information about the symbolic link, not the file referenced by the symbolic link. (We'll need `lstat` in Section 4.22 when we walk down a directory hierarchy. We describe symbolic links in more detail in Section 4.17.)

The `fstatat` function provides a way to return the file statistics for a pathname relative to an open directory represented by the *fd* argument. The *flag* argument controls whether symbolic links are followed; when the `AT_SYMLINK_NOFOLLOW` flag is set, `fstatat` will not follow symbolic links, but rather returns information about the link itself. Otherwise, the default is to follow symbolic links, returning information about the file to which the symbolic link points. If the *fd* argument has the value `AT_FDCWD` and the *pathname* argument is a relative pathname, then `fstatat` evaluates the *pathname* argument relative to the current directory. If the *pathname* argument is an absolute pathname, then the *fd* argument is ignored. In these two cases, `fstatat` behaves like either `stat` or `lstat`, depending on the value of *flag*.

The *buf* argument is a pointer to a structure that we must supply. The functions fill in the structure. The definition of the structure can differ among implementations, but it could look like

```
struct stat {
  mode_t          st_mode;    /* file type & mode (permissions) */
  ino_t           st_ino;     /* i-node number (serial number) */
  dev_t           st_dev;     /* device number (file system) */
  dev_t           st_rdev;    /* device number for special files */
  nlink_t         st_nlink;   /* number of links */
  uid_t           st_uid;     /* user ID of owner */
  gid_t           st_gid;     /* group ID of owner */
  off_t           st_size;    /* size in bytes, for regular files */
  struct timespec st_atim;    /* time of last access */
  struct timespec st_mtim;    /* time of last modification */
  struct timespec st_ctim;    /* time of last file status change */
  blksize_t       st_blksize; /* best I/O block size */
  blkcnt_t        st_blocks;  /* number of disk blocks allocated */
};
```

The `st_rdev`, `st_blksize`, and `st_blocks` fields are not required by POSIX.1. They are defined as part of the XSI option in the Single UNIX Specification.

The `timespec` structure type defines time in terms of seconds and nanoseconds. It includes at least the following fields:

```
time_t tv_sec;
long   tv_nsec;
```

Prior to the 2008 version of the standard, the time fields were named `st_atime`, `st_mtime`, and `st_ctime`, and were of type `time_t` (expressed in seconds). The `timespec` structure enables higher-resolution timestamps. The old names can be defined in terms of the `tv_sec` members for compatibility. For example, `st_atime` can be defined as `st_atim.tv_sec`.

Note that most members of the `stat` structure are specified by a primitive system data type (see Section 2.8). We'll go through each member of this structure to examine the attributes of a file.

The biggest user of the `stat` functions is probably the `ls -l` command, to learn all the information about a file.

4.3 File Types

We've talked about two different types of files so far: regular files and directories. Most files on a UNIX system are either regular files or directories, but there are additional types of files. The types are

1. Regular file. The most common type of file, which contains data of some form. There is no distinction to the UNIX kernel whether this data is text or binary. Any interpretation of the contents of a regular file is left to the application processing the file.

   > One notable exception to this is with binary executable files. To execute a program, the kernel must understand its format. All binary executable files conform to a format that allows the kernel to identify where to load a program's text and data.

2. Directory file. A file that contains the names of other files and pointers to information on these files. Any process that has read permission for a directory file can read the contents of the directory, but only the kernel can write directly to a directory file. Processes must use the functions described in this chapter to make changes to a directory.
3. Block special file. A type of file providing buffered I/O access in fixed-size units to devices such as disk drives.

   > Note that FreeBSD no longer supports block special files. All access to devices is through the character special interface.

4. Character special file. A type of file providing unbuffered I/O access in variable-sized units to devices. All devices on a system are either block special files or character special files.
5. FIFO. A type of file used for communication between processes. It's sometimes called a named pipe. We describe FIFOs in Section 15.5.
6. Socket. A type of file used for network communication between processes. A socket can also be used for non-network communication between processes on a single host. We use sockets for interprocess communication in Chapter 16.
7. Symbolic link. A type of file that points to another file. We talk more about symbolic links in Section 4.17.

The type of a file is encoded in the `st_mode` member of the `stat` structure. We can determine the file type with the macros shown in Figure 4.1. The argument to each of these macros is the `st_mode` member from the `stat` structure.

Macro	Type of file
S_ISREG()	regular file
S_ISDIR()	directory file
S_ISCHR()	character special file
S_ISBLK()	block special file
S_ISFIFO()	pipe or FIFO
S_ISLNK()	symbolic link
S_ISSOCK()	socket

Figure 4.1 File type macros in <sys/stat.h>

POSIX.1 allows implementations to represent interprocess communication (IPC) objects, such as message queues and semaphores, as files. The macros shown in Figure 4.2 allow us to determine the type of IPC object from the stat structure. Instead of taking the st_mode member as an argument, these macros differ from those in Figure 4.1 in that their argument is a pointer to the stat structure.

Macro	Type of object
S_TYPEISMQ()	message queue
S_TYPEISSEM()	semaphore
S_TYPEISSHM()	shared memory object

Figure 4.2 IPC type macros in <sys/stat.h>

Message queues, semaphores, and shared memory objects are discussed in Chapter 15. However, none of the various implementations of the UNIX System discussed in this book represent these objects as files.

Example

The program in Figure 4.3 prints the type of file for each command-line argument.

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int         i;
    struct stat buf;
    char        *ptr;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            err_ret("lstat error");
            continue;
        }
        if (S_ISREG(buf.st_mode))
            ptr = "regular";
        else if (S_ISDIR(buf.st_mode))
```

```
            ptr = "directory";
        else if (S_ISCHR(buf.st_mode))
            ptr = "character special";
        else if (S_ISBLK(buf.st_mode))
            ptr = "block special";
        else if (S_ISFIFO(buf.st_mode))
            ptr = "fifo";
        else if (S_ISLNK(buf.st_mode))
            ptr = "symbolic link";
        else if (S_ISSOCK(buf.st_mode))
            ptr = "socket";
        else
            ptr = "** unknown mode **";
        printf("%s\n", ptr);
    }
    exit(0);
}
```

Figure 4.3 Print type of file for each command-line argument

Sample output from Figure 4.3 is

```
$ ./a.out /etc/passwd /etc /dev/log /dev/tty \
> /var/lib/oprofile/opd_pipe /dev/sr0 /dev/cdrom
/etc/passwd: regular
/etc: directory
/dev/log: socket
/dev/tty: character special
/var/lib/oprofile/opd_pipe: fifo
/dev/sr0: block special
/dev/cdrom: symbolic link
```

(Here, we have explicitly entered a backslash at the end of the first command line, telling the shell that we want to continue entering the command on another line. The shell then prompted us with its secondary prompt, >, on the next line.) We have specifically used the `lstat` function instead of the `stat` function to detect symbolic links. If we used the `stat` function, we would never see symbolic links. □

Historically, early versions of the UNIX System didn't provide the `S_ISxxx` macros. Instead, we had to logically AND the `st_mode` value with the mask `S_IFMT` and then compare the result with the constants whose names are `S_IFxxx`. Most systems define this mask and the related constants in the file `<sys/stat.h>`. If we examine this file, we'll find the `S_ISDIR` macro defined something like

```
#define  S_ISDIR(mode)  (((mode) & S_IFMT) == S_IFDIR)
```

We've said that regular files are predominant, but it is interesting to see what percentage of the files on a given system are of each file type. Figure 4.4 shows the counts and percentages for a Linux system that is used as a single-user workstation. This data was obtained from the program shown in Section 4.22.

File type	Count	Percentage
regular file	415,803	79.77 %
directory	62,197	11.93
symbolic link	40,018	8.25
character special	155	0.03
block special	47	0.01
socket	45	0.01
FIFO	0	0.00

Figure 4.4 Counts and percentages of different file types

4.4 Set-User-ID and Set-Group-ID

Every process has six or more IDs associated with it. These are shown in Figure 4.5.

real user ID real group ID	who we really are
effective user ID effective group ID supplementary group IDs	used for file access permission checks
saved set-user-ID saved set-group-ID	saved by `exec` functions

Figure 4.5 User IDs and group IDs associated with each process

- The real user ID and real group ID identify who we really are. These two fields are taken from our entry in the password file when we log in. Normally, these values don't change during a login session, although there are ways for a superuser process to change them, which we describe in Section 8.11.
- The effective user ID, effective group ID, and supplementary group IDs determine our file access permissions, as we describe in the next section. (We defined supplementary group IDs in Section 1.8.)
- The saved set-user-ID and saved set-group-ID contain copies of the effective user ID and the effective group ID, respectively, when a program is executed. We describe the function of these two saved values when we describe the `setuid` function in Section 8.11.

> The saved IDs are required as of the 2001 version of POSIX.1. They were optional in older versions of POSIX. An application can test for the constant `_POSIX_SAVED_IDS` at compile time or can call `sysconf` with the `_SC_SAVED_IDS` argument at runtime, to see whether the implementation supports this feature.

Normally, the effective user ID equals the real user ID, and the effective group ID equals the real group ID.

Every file has an owner and a group owner. The owner is specified by the `st_uid` member of the `stat` structure; the group owner, by the `st_gid` member.

When we execute a program file, the effective user ID of the process is usually the real user ID, and the effective group ID is usually the real group ID. However, we can also set a special flag in the file's mode word (`st_mode`) that says, "When this file is executed, set the effective user ID of the process to be the owner of the file (`st_uid`)." Similarly, we can set another bit in the file's mode word that causes the effective group ID to be the group owner of the file (`st_gid`). These two bits in the file's mode word are called the *set-user-ID* bit and the *set-group-ID* bit.

For example, if the owner of the file is the superuser and if the file's set-user-ID bit is set, then while that program file is running as a process, it has superuser privileges. This happens regardless of the real user ID of the process that executes the file. As an example, the UNIX System program that allows anyone to change his or her password, `passwd`(1), is a set-user-ID program. This is required so that the program can write the new password to the password file, typically either `/etc/passwd` or `/etc/shadow`, files that should be writable only by the superuser. Because a process that is running set-user-ID to some other user usually assumes extra permissions, it must be written carefully. We'll discuss these types of programs in more detail in Chapter 8.

Returning to the `stat` function, the set-user-ID bit and the set-group-ID bit are contained in the file's `st_mode` value. These two bits can be tested against the constants `S_ISUID` and `S_ISGID`, respectively.

4.5 File Access Permissions

The `st_mode` value also encodes the access permission bits for the file. When we say *file*, we mean any of the file types that we described earlier. All the file types—directories, character special files, and so on—have permissions. Many people think of only regular files as having access permissions.

There are nine permission bits for each file, divided into three categories. They are shown in Figure 4.6.

`st_mode` mask	Meaning
`S_IRUSR`	user-read
`S_IWUSR`	user-write
`S_IXUSR`	user-execute
`S_IRGRP`	group-read
`S_IWGRP`	group-write
`S_IXGRP`	group-execute
`S_IROTH`	other-read
`S_IWOTH`	other-write
`S_IXOTH`	other-execute

Figure 4.6 The nine file access permission bits, from `<sys/stat.h>`

The term *user* in the first three rows in Figure 4.6 refers to the owner of the file. The `chmod`(1) command, which is typically used to modify these nine permission bits, allows us to specify `u` for user (owner), `g` for group, and `o` for other. Some books refer to these three as owner, group, and world; this is confusing, as the `chmod` command

uses o to mean other, not owner. We'll use the terms *user*, *group*, and *other*, to be consistent with the chmod command.

The three categories in Figure 4.6—read, write, and execute—are used in various ways by different functions. We'll summarize them here, and return to them when we describe the actual functions.

- The first rule is that *whenever* we want to open any type of file by name, we must have execute permission in each directory mentioned in the name, including the current directory, if it is implied. This is why the execute permission bit for a directory is often called the search bit.

  For example, to open the file /usr/include/stdio.h, we need execute permission in the directory /, execute permission in the directory /usr, and execute permission in the directory /usr/include. We then need appropriate permission for the file itself, depending on how we're trying to open it: read-only, read–write, and so on.

  If the current directory is /usr/include, then we need execute permission in the current directory to open the file stdio.h. This is an example of the current directory being implied, not specifically mentioned. It is identical to our opening the file ./stdio.h.

  Note that read permission for a directory and execute permission for a directory mean different things. Read permission lets us read the directory, obtaining a list of all the filenames in the directory. Execute permission lets us pass through the directory when it is a component of a pathname that we are trying to access. (We need to search the directory to look for a specific filename.)

  Another example of an implicit directory reference is if the PATH environment variable, described in Section 8.10, specifies a directory that does not have execute permission enabled. In this case, the shell will never find executable files in that directory.
- The read permission for a file determines whether we can open an existing file for reading: the O_RDONLY and O_RDWR flags for the open function.
- The write permission for a file determines whether we can open an existing file for writing: the O_WRONLY and O_RDWR flags for the open function.
- We must have write permission for a file to specify the O_TRUNC flag in the open function.
- We cannot create a new file in a directory unless we have write permission and execute permission in the directory.
- To delete an existing file, we need write permission and execute permission in the directory containing the file. We do not need read permission or write permission for the file itself.
- Execute permission for a file must be on if we want to execute the file using any of the seven exec functions (Section 8.10). The file also has to be a regular file.

The file access tests that the kernel performs each time a process opens, creates, or deletes a file depend on the owners of the file (`st_uid` and `st_gid`), the effective IDs of the process (effective user ID and effective group ID), and the supplementary group IDs of the process, if supported. The two owner IDs are properties of the file, whereas the two effective IDs and the supplementary group IDs are properties of the process. The tests performed by the kernel are as follows:

1. If the effective user ID of the process is 0 (the superuser), access is allowed. This gives the superuser free rein throughout the entire file system.
2. If the effective user ID of the process equals the owner ID of the file (i.e., the process owns the file), access is allowed if the appropriate user access permission bit is set. Otherwise, permission is denied. By *appropriate access permission bit*, we mean that if the process is opening the file for reading, the user-read bit must be on. If the process is opening the file for writing, the user-write bit must be on. If the process is executing the file, the user-execute bit must be on.
3. If the effective group ID of the process or one of the supplementary group IDs of the process equals the group ID of the file, access is allowed if the appropriate group access permission bit is set. Otherwise, permission is denied.
4. If the appropriate other access permission bit is set, access is allowed. Otherwise, permission is denied.

These four steps are tried in sequence. Note that if the process owns the file (step 2), access is granted or denied based only on the user access permissions; the group permissions are never looked at. Similarly, if the process does not own the file but belongs to an appropriate group, access is granted or denied based only on the group access permissions; the other permissions are not looked at.

4.6 Ownership of New Files and Directories

When we described the creation of a new file in Chapter 3 using either `open` or `creat`, we never said which values were assigned to the user ID and group ID of the new file. We'll see how to create a new directory in Section 4.21 when we describe the `mkdir` function. The rules for the ownership of a new directory are identical to the rules in this section for the ownership of a new file.

The user ID of a new file is set to the effective user ID of the process. POSIX.1 allows an implementation to choose one of the following options to determine the group ID of a new file:

1. The group ID of a new file can be the effective group ID of the process.
2. The group ID of a new file can be the group ID of the directory in which the file is being created.

FreeBSD 8.0 and Mac OS X 10.6.8 always copy the new file's group ID from the directory. Several Linux file systems allow the choice between the two options to be selected using a `mount`(1) command option. The default behavior for Linux 3.2.0 and Solaris 10 is to determine the group ID of a new file depending on whether the set-group-ID bit is set for the directory in which the file is created. If this bit is set, the new file's group ID is copied from the directory; otherwise, the new file's group ID is set to the effective group ID of the process.

Using the second option—inheriting the directory's group ID—assures us that all files and directories created in that directory will have the same group ID as the directory. This group ownership of files and directories will then propagate down the hierarchy from that point. This is used in the Linux directory `/var/mail`, for example.

As we mentioned earlier, this option for group ownership is the default for FreeBSD 8.0 and Mac OS X 10.6.8, but an option for Linux and Solaris. Under Solaris 10, and by default under Linux 3.2.0, we have to enable the set-group-ID bit, and the `mkdir` function has to propagate a directory's set-group-ID bit automatically for this to work. (This is described in Section 4.21.)

4.7 `access` and `faccessat` Functions

As we described earlier, when we open a file, the kernel performs its access tests based on the effective user and group IDs. Sometimes, however, a process wants to test accessibility based on the real user and group IDs. This is useful when a process is running as someone else, using either the set-user-ID or the set-group-ID feature. Even though a process might be set-user-ID to root, it might still want to verify that the real user can access a given file. The `access` and `faccessat` functions base their tests on the real user and group IDs. (Replace *effective* with *real* in the four steps at the end of Section 4.5.)

```
#include <unistd.h>

int access(const char *pathname, int mode);

int faccessat(int fd, const char *pathname, int mode, int flag);
                                                    Both return: 0 if OK, –1 on error
```

The *mode* is either the value `F_OK` to test if a file exists, or the bitwise OR of any of the flags shown in Figure 4.7.

mode	Description
`R_OK`	test for read permission
`W_OK`	test for write permission
`X_OK`	test for execute permission

Figure 4.7 The *mode* flags for `access` function, from `<unistd.h>`

The `faccessat` function behaves like `access` when the *pathname* argument is absolute or when the *fd* argument has the value `AT_FDCWD` and the *pathname* argument is relative. Otherwise, `faccessat` evaluates the *pathname* relative to the open directory referenced by the *fd* argument.

The *flag* argument can be used to change the behavior of faccessat. If the AT_EACCESS flag is set, the access checks are made using the effective user and group IDs of the calling process instead of the real user and group IDs.

Example

Figure 4.8 shows the use of the access function.

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if (access(argv[1], R_OK) < 0)
        err_ret("access error for %s", argv[1]);
    else
        printf("read access OK\n");
    if (open(argv[1], O_RDONLY) < 0)
        err_ret("open error for %s", argv[1]);
    else
        printf("open for reading OK\n");
    exit(0);
}
```

Figure 4.8 Example of access function

Here is a sample session with this program:

```
$ ls -l a.out
-rwxrwxr-x  1 sar         15945 Nov 30 12:10 a.out
$ ./a.out a.out
read access OK
open for reading OK
$ ls -l /etc/shadow
-r--------  1 root         1315 Jul 17  2002 /etc/shadow
$ ./a.out /etc/shadow
access error for /etc/shadow: Permission denied
open error for /etc/shadow: Permission denied
$ su                                  become superuser
Password:                             enter superuser password
# chown root a.out                    change file's user ID to root
# chmod u+s a.out                     and turn on set-user-ID bit
# ls -l a.out                         check owner and SUID bit
-rwsrwxr-x  1 root        15945 Nov 30 12:10 a.out
# exit                                go back to normal user
$ ./a.out /etc/shadow
access error for /etc/shadow: Permission denied
open for reading OK
```

In this example, the set-user-ID program can determine that the real user cannot normally read the file, even though the open function will succeed. □

> In the preceding example and in Chapter 8, we'll sometimes switch to become the superuser to demonstrate how something works. If you're on a multiuser system and do not have superuser permission, you won't be able to duplicate these examples completely.

4.8 umask Function

Now that we've described the nine permission bits associated with every file, we can describe the file mode creation mask that is associated with every process.

The umask function sets the file mode creation mask for the process and returns the previous value. (This is one of the few functions that doesn't have an error return.)

```
#include <sys/stat.h>

mode_t umask(mode_t cmask);
```

Returns: previous file mode creation mask

The *cmask* argument is formed as the bitwise OR of any of the nine constants from Figure 4.6: S_IRUSR, S_IWUSR, and so on.

The file mode creation mask is used whenever the process creates a new file or a new directory. (Recall from Sections 3.3 and 3.4 our description of the open and creat functions. Both accept a *mode* argument that specifies the new file's access permission bits.) We describe how to create a new directory in Section 4.21. Any bits that are *on* in the file mode creation mask are turned *off* in the file's *mode*.

Example

The program in Figure 4.9 creates two files: one with a umask of 0 and one with a umask that disables all the group and other permission bits.

```
#include "apue.h"
#include <fcntl.h>

#define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

int
main(void)
{
    umask(0);
    if (creat("foo", RWRWRW) < 0)
        err_sys("creat error for foo");
    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    if (creat("bar", RWRWRW) < 0)
        err_sys("creat error for bar");
    exit(0);
}
```

Figure 4.9 Example of umask function

If we run this program, we can see how the permission bits have been set.

```
$ umask                          first print the current file mode creation mask
002
$ ./a.out
$ ls -l foo bar
-rw-------  1 sar              0 Dec  7 21:20 bar
-rw-rw-rw-  1 sar              0 Dec  7 21:20 foo
$ umask                          see if the file mode creation mask changed
002
```

□

Most users of UNIX systems never deal with their umask value. It is usually set once, on login, by the shell's start-up file, and never changed. Nevertheless, when writing programs that create new files, if we want to ensure that specific access permission bits are enabled, we must modify the umask value while the process is running. For example, if we want to ensure that anyone can read a file, we should set the umask to 0. Otherwise, the umask value that is in effect when our process is running can cause permission bits to be turned off.

In the preceding example, we use the shell's umask command to print the file mode creation mask both before we run the program and after it completes. This shows us that changing the file mode creation mask of a process doesn't affect the mask of its parent (often a shell). All of the shells have a built-in umask command that we can use to set or print the current file mode creation mask.

Users can set the umask value to control the default permissions on the files they create. This value is expressed in octal, with one bit representing one permission to be masked off, as shown in Figure 4.10. Permissions can be denied by setting the corresponding bits. Some common umask values are 002 to prevent others from writing your files, 022 to prevent group members and others from writing your files, and 027 to prevent group members from writing your files and others from reading, writing, or executing your files.

Mask bit	Meaning
0400	user-read
0200	user-write
0100	user-execute
0040	group-read
0020	group-write
0010	group-execute
0004	other-read
0002	other-write
0001	other-execute

Figure 4.10 The umask file access permission bits

The Single UNIX Specification requires that the umask command support a symbolic mode of operation. Unlike the octal format, the symbolic format specifies which permissions are to be allowed (i.e., clear in the file creation mask) instead of which ones are to be denied (i.e., set in the file creation mask). Compare both forms of the command, shown below.

```
$ umask                          first print the current file mode creation mask
002
$ umask -S                       print the symbolic form
u=rwx,g=rwx,o=rx
$ umask 027                      change the file mode creation mask
$ umask -S                       print the symbolic form
u=rwx,g=rx,o=
```

4.9 chmod, fchmod, and fchmodat Functions

The chmod, fchmod, and fchmodat functions allow us to change the file access permissions for an existing file.

```
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);

int fchmod(int fd, mode_t mode);

int fchmodat(int fd, const char *pathname, mode_t mode, int flag);
```

All three return: 0 if OK, –1 on error

The chmod function operates on the specified file, whereas the fchmod function operates on a file that has already been opened. The fchmodat function behaves like chmod when the *pathname* argument is absolute or when the *fd* argument has the value AT_FDCWD and the *pathname* argument is relative. Otherwise, fchmodat evaluates the *pathname* relative to the open directory referenced by the *fd* argument. The *flag* argument can be used to change the behavior of fchmodat—when the AT_SYMLINK_NOFOLLOW flag is set, fchmodat doesn't follow symbolic links.

To change the permission bits of a file, the effective user ID of the process must be equal to the owner ID of the file, or the process must have superuser permissions.

The *mode* is specified as the bitwise OR of the constants shown in Figure 4.11.

mode	Description
S_ISUID	set-user-ID on execution
S_ISGID	set-group-ID on execution
S_ISVTX	saved-text (sticky bit)
S_IRWXU	read, write, and execute by user (owner)
S_IRUSR	read by user (owner)
S_IWUSR	write by user (owner)
S_IXUSR	execute by user (owner)
S_IRWXG	read, write, and execute by group
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXO	read, write, and execute by other (world)
S_IROTH	read by other (world)
S_IWOTH	write by other (world)
S_IXOTH	execute by other (world)

Figure 4.11 The *mode* constants for chmod functions, from <sys/stat.h>

Note that nine of the entries in Figure 4.11 are the nine file access permission bits from Figure 4.6. We've added the two set-ID constants (S_ISUID and S_ISGID), the saved-text constant (S_ISVTX), and the three combined constants (S_IRWXU, S_IRWXG, and S_IRWXO).

> The saved-text bit (S_ISVTX) is not part of POSIX.1. It is defined in the XSI option in the Single UNIX Specification. We describe its purpose in the next section.

Example

Recall the final state of the files foo and bar when we ran the program in Figure 4.9 to demonstrate the umask function:

```
$ ls -l foo bar
-rw-------  1 sar             0 Dec  7 21:20 bar
-rw-rw-rw-  1 sar             0 Dec  7 21:20 foo
```

The program shown in Figure 4.12 modifies the mode of these two files.

```
#include "apue.h"

int
main(void)
{
    struct stat     statbuf;

    /* turn on set-group-ID and turn off group-execute */

    if (stat("foo", &statbuf) < 0)
        err_sys("stat error for foo");
    if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("chmod error for foo");

    /* set absolute mode to "rw-r--r--" */

    if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
        err_sys("chmod error for bar");

    exit(0);
}
```

Figure 4.12 Example of chmod function

After running the program in Figure 4.12, we see that the final state of the two files is

```
$ ls -l foo bar
-rw-r--r--  1 sar             0 Dec  7 21:20 bar
-rw-rwSrw-  1 sar             0 Dec  7 21:20 foo
```

In this example, we have set the permissions of the file bar to an absolute value, regardless of the current permission bits. For the file foo, we set the permissions relative to their current state. To do this, we first call stat to obtain the current permissions and then modify them. We have explicitly turned on the set-group-ID bit and turned off the group-execute bit. Note that the ls command lists the group-execute permission as S to signify that the set-group-ID bit is set without the group-execute bit being set.

On Solaris, the `ls` command displays an `l` instead of an `S` to indicate that mandatory file and record locking has been enabled for this file. This behavior applies only to regular files, but we'll discuss this more in Section 14.3.

Finally, note that the time and date listed by the `ls` command did not change after we ran the program in Figure 4.12. We'll see in Section 4.19 that the `chmod` function updates only the time that the i-node was last changed. By default, the `ls -l` lists the time when the contents of the file were last modified. □

The `chmod` functions automatically clear two of the permission bits under the following conditions:

- On systems, such as Solaris, that place special meaning on the sticky bit when used with regular files, if we try to set the sticky bit (`S_ISVTX`) on a regular file and do not have superuser privileges, the sticky bit in the *mode* is automatically turned off. (We describe the sticky bit in the next section.) To prevent malicious users from setting the sticky bit and adversely affecting system performance, only the superuser can set the sticky bit of a regular file.

  In FreeBSD 8.0 and Solaris 10, only the superuser can set the sticky bit on a regular file. Linux 3.2.0 and Mac OS X 10.6.8 place no such restriction on the setting of the sticky bit, because the bit has no meaning when applied to regular files on these systems. Although the bit also has no meaning when applied to regular files on FreeBSD, everyone except the superuser is prevented from setting it on a regular file.

- The group ID of a newly created file might potentially be a group that the calling process does not belong to. Recall from Section 4.6 that it's possible for the group ID of the new file to be the group ID of the parent directory. Specifically, if the group ID of the new file does not equal either the effective group ID of the process or one of the process's supplementary group IDs and if the process does not have superuser privileges, then the set-group-ID bit is automatically turned off. This prevents a user from creating a set-group-ID file owned by a group that the user doesn't belong to.

  FreeBSD 8.0 fails an attempt to set the set-group-ID in this case. The other systems silently turn the bit off, but don't fail the attempt to change the file access permissions.

  FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10 add another security feature to try to prevent misuse of some of the protection bits. If a process that does not have superuser privileges writes to a file, the set-user-ID and set-group-ID bits are automatically turned off. If malicious users find a set-group-ID or a set-user-ID file they can write to, even though they can modify the file, they lose the special privileges of the file.

4.10 Sticky Bit

The `S_ISVTX` bit has an interesting history. On versions of the UNIX System that predated demand paging, this bit was known as the *sticky bit*. If it was set for an executable program file, then the first time the program was executed, a copy of the program's text was saved in the swap area when the process terminated. (The text

portion of a program is the machine instructions.) The program would then load into memory more quickly the next time it was executed, because the swap area was handled as a contiguous file, as compared to the possibly random location of data blocks in a normal UNIX file system. The sticky bit was often set for common application programs, such as the text editor and the passes of the C compiler. Naturally, there was a limit to the number of sticky files that could be contained in the swap area before running out of swap space, but it was a useful technique. The name *sticky* came about because the text portion of the file stuck around in the swap area until the system was rebooted. Later versions of the UNIX System referred to this as the *saved-text* bit; hence the constant `S_ISVTX`. With today's newer UNIX systems, most of which have a virtual memory system and a faster file system, the need for this technique has disappeared.

On contemporary systems, the use of the sticky bit has been extended. The Single UNIX Specification allows the sticky bit to be set for a directory. If the bit is set for a directory, a file in the directory can be removed or renamed only if the user has write permission for the directory and meets one of the following criteria:

- Owns the file
- Owns the directory
- Is the superuser

The directories `/tmp` and `/var/tmp` are typical candidates for the sticky bit—they are directories in which any user can typically create files. The permissions for these two directories are often read, write, and execute for everyone (user, group, and other). But users should not be able to delete or rename files owned by others.

> The saved-text bit is not part of POSIX.1. It is part of the XSI option defined in the Single UNIX Specification, and is supported by FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10.
>
> Solaris 10 places special meaning on the sticky bit if it is set on a regular file. In this case, if none of the execute bits is set, the operating system will not cache the contents of the file.

4.11 chown, fchown, fchownat, and lchown Functions

The `chown` functions allow us to change a file's user ID and group ID, but if either of the arguments *owner* or *group* is –1, the corresponding ID is left unchanged.

```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);

int fchown(int fd, uid_t owner, gid_t group);

int fchownat(int fd, const char *pathname, uid_t owner, gid_t group,
             int flag);

int lchown(const char *pathname, uid_t owner, gid_t group);
```

All four return: 0 if OK, –1 on error

These four functions operate similarly unless the referenced file is a symbolic link. In that case, `lchown` and `fchownat` (with the `AT_SYMLINK_NOFOLLOW` flag set) change the owners of the symbolic link itself, not the file pointed to by the symbolic link.

The `fchown` function changes the ownership of the open file referenced by the *fd* argument. Since it operates on a file that is already open, it can't be used to change the ownership of a symbolic link.

The `fchownat` function behaves like either `chown` or `lchown` when the *pathname* argument is absolute or when the *fd* argument has the value `AT_FDCWD` and the *pathname* argument is relative. In these cases, `fchownat` acts like `lchown` if the `AT_SYMLINK_NOFOLLOW` flag is set in the *flag* argument, or it acts like `chown` if the `AT_SYMLINK_NOFOLLOW` flag is clear. When the *fd* argument is set to the file descriptor of an open directory and the *pathname* argument is a relative pathname, `fchownat` evaluates the *pathname* relative to the open directory.

Historically, BSD-based systems have enforced the restriction that only the superuser can change the ownership of a file. This is to prevent users from giving away their files to others, thereby defeating any disk space quota restrictions. System V, however, has allowed all users to change the ownership of any files they own.

> POSIX.1 allows either form of operation, depending on the value of `_POSIX_CHOWN_RESTRICTED`.
>
> With Solaris 10, this functionality is a configuration option, whose default value is to enforce the restriction. FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8 always enforce the `chown` restriction.

Recall from Section 2.6 that the `_POSIX_CHOWN_RESTRICTED` constant can optionally be defined in the header `<unistd.h>`, and can always be queried using either the `pathconf` function or the `fpathconf` function. Also recall that this option can depend on the referenced file; it can be enabled or disabled on a per file system basis. We'll use the phrase "if `_POSIX_CHOWN_RESTRICTED` is in effect," to mean "if it applies to the particular file that we're talking about," regardless of whether this actual constant is defined in the header.

If `_POSIX_CHOWN_RESTRICTED` is in effect for the specified file, then

1. Only a superuser process can change the user ID of the file.
2. A nonsuperuser process can change the group ID of the file if the process owns the file (the effective user ID equals the user ID of the file), *owner* is specified as –1 or equals the user ID of the file, and *group* equals either the effective group ID of the process or one of the process's supplementary group IDs.

This means that when `_POSIX_CHOWN_RESTRICTED` is in effect, you can't change the user ID of your files. You can change the group ID of files that you own, but only to groups that you belong to.

If these functions are called by a process other than a superuser process, on successful return, both the set-user-ID and the set-group-ID bits are cleared.

4.12 File Size

The `st_size` member of the `stat` structure contains the size of the file in bytes. This field is meaningful only for regular files, directories, and symbolic links.

> FreeBSD 8.0, Mac OS X 10.6.8, and Solaris 10 also define the file size for a pipe as the number of bytes that are available for reading from the pipe. We'll discuss pipes in Section 15.2.

For a regular file, a file size of 0 is allowed. We'll get an end-of-file indication on the first read of the file. For a directory, the file size is usually a multiple of a number, such as 16 or 512. We talk about reading directories in Section 4.22.

For a symbolic link, the file size is the number of bytes in the filename. For example, in the following case, the file size of 7 is the length of the pathname `usr/lib`:

```
lrwxrwxrwx  1 root            7 Sep 25 07:14 lib -> usr/lib
```

(Note that symbolic links do not contain the normal C null byte at the end of the name, as the length is always specified by `st_size`.)

Most contemporary UNIX systems provide the fields `st_blksize` and `st_blocks`. The first is the preferred block size for I/O for the file, and the latter is the actual number of 512-byte blocks that are allocated. Recall from Section 3.9 that we encountered the minimum amount of time required to read a file when we used `st_blksize` for the `read` operations. The standard I/O library, which we describe in Chapter 5, also tries to read or write `st_blksize` bytes at a time, for efficiency.

> Be aware that different versions of the UNIX System use units other than 512-byte blocks for `st_blocks`. Use of this value is nonportable.

Holes in a File

In Section 3.6, we mentioned that a regular file can contain "holes." We showed an example of this in Figure 3.2. Holes are created by seeking past the current end of file and writing some data. As an example, consider the following:

```
$ ls -l core
-rw-r--r--  1 sar   8483248 Nov 18 12:18 core
$ du -s core
272     core
```

The size of the file `core` is slightly more than 8 MB, yet the `du` command reports that the amount of disk space used by the file is 272 512-byte blocks (139,264 bytes). Obviously, this file has many holes.

> The `du` command on many BSD-derived systems reports the number of 1,024-byte blocks. Solaris reports the number of 512-byte blocks. On Linux, the units reported depend on the whether the `POSIXLY_CORRECT` environment is set. When it is set, the `du` command reports 1,024-byte block units; when it is not set, the command reports 512-byte block units.

As we mentioned in Section 3.6, the `read` function returns data bytes of 0 for any byte positions that have not been written. If we execute the following command, we can see that the normal I/O operations read up through the size of the file:

```
$ wc -c core
 8483248 core
```

The wc(1) command with the -c option counts the number of characters (bytes) in the file.

If we make a copy of this file, using a utility such as `cat`(1), all these holes are written out as actual data bytes of 0:

```
$ cat core > core.copy
$ ls -l core*
-rw-r--r--  1 sar   8483248 Nov 18 12:18 core
-rw-rw-r--  1 sar   8483248 Nov 18 12:27 core.copy
$ du -s core*
272     core
16592   core.copy
```

Here, the actual number of bytes used by the new file is 8,495,104 (512 × 16,592). The difference between this size and the size reported by `ls` is caused by the number of blocks used by the file system to hold pointers to the actual data blocks.

Interested readers should refer to Section 4.2 of Bach [1986], Sections 7.2 and 7.3 of McKusick et al. [1996] (or Sections 8.2 and 8.3 in McKusick and Neville-Neil [2005]), Section 15.2 of McDougall and Mauro [2007], and Chapter 12 in Singh [2006] for additional details on the physical layout of files.

4.13 File Truncation

Sometimes we would like to truncate a file by chopping off data at the end of the file. Emptying a file, which we can do with the `O_TRUNC` flag to `open`, is a special case of truncation.

```
#include <unistd.h>

int truncate(const char *pathname, off_t length);

int ftruncate(int fd, off_t length);
```

Both return: 0 if OK, –1 on error

These two functions truncate an existing file to *length* bytes. If the previous size of the file was greater than *length*, the data beyond *length* is no longer accessible. Otherwise, if the previous size was less than *length*, the file size will increase and the data between the old end of file and the new end of file will read as 0 (i.e., a hole is probably created in the file).

BSD releases prior to 4.4BSD could only make a file smaller with `truncate`.

Solaris also includes an extension to `fcntl` (`F_FREESP`) that allows us to free any part of a file, not just a chunk at the end of the file.

We use `ftruncate` in the program shown in Figure 13.6 when we need to empty a file after obtaining a lock on the file.

4.14 File Systems

To appreciate the concept of links to a file, we need a conceptual understanding of the structure of the UNIX file system. Understanding the difference between an i-node and a directory entry that points to an i-node is also useful.

Various implementations of the UNIX file system are in use today. Solaris, for example, supports several types of disk file systems: the traditional BSD-derived UNIX file system (called `UFS`), a file system (called `PCFS`) to read and write DOS-formatted diskettes, and a file system (called `HSFS`) to read CD file systems. We saw one difference between file system types in Figure 2.20. `UFS` is based on the Berkeley fast file system, which we describe in this section.

> Each file system type has its own characteristic features—and some of these features can be confusing. For example, most UNIX file systems support case-sensitive filenames. Thus, if you create one file named `file.txt` and another named `file.TXT`, then two distinct files are created. On Mac OS X, however, the `HFS` file system is case-preserving with case-insensitive comparisons. Thus, if you create `file.txt`, when you try to create `file.TXT`, you will overwrite `file.txt`. However, only the name used when the file was created is stored in the file system (the case-preserving aspect). In fact, any permutation of uppercase and lowercase letters in the sequence `f`, `i`, `l`, `e`, `.`, `t`, `x`, `t` will match when searching for the file (the case-insensitive comparison aspect). As a consequence, besides `file.txt` and `file.TXT`, we can access the file with the names `File.txt`, `fILE.tXt`, and `FiLe.TxT`.

We can think of a disk drive being divided into one or more partitions. Each partition can contain a file system, as shown in Figure 4.13. The i-nodes are fixed-length entries that contain most of the information about a file.

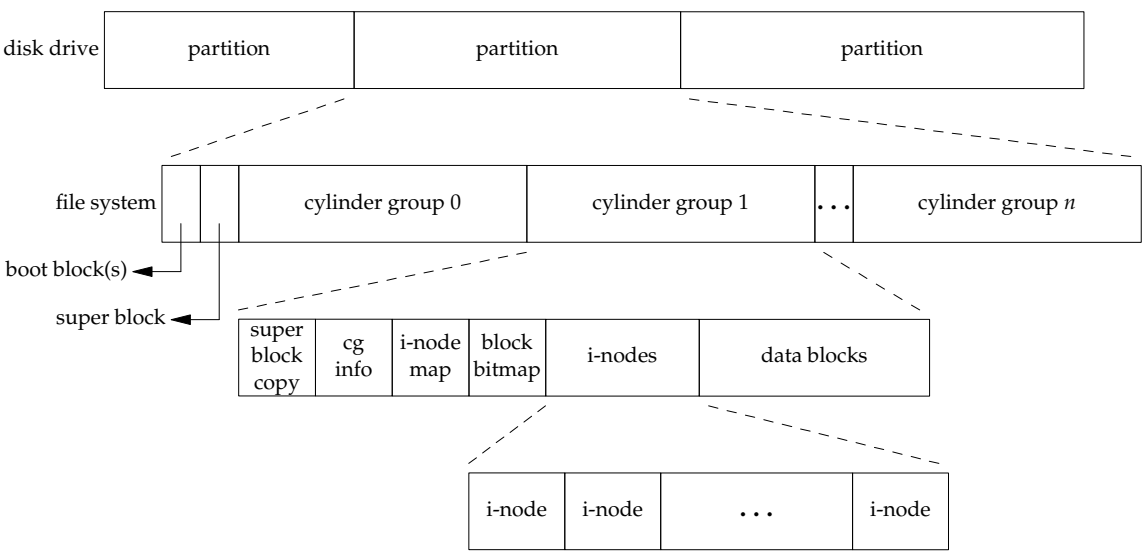


Figure 4.13 Disk drive, partitions, and a file system

If we examine the i-node and data block portion of a cylinder group in more detail, we could have the arrangement shown in Figure 4.14.

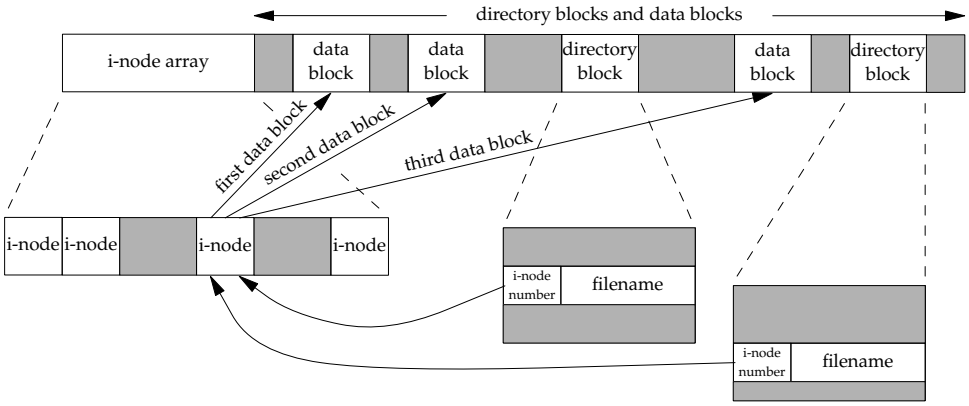


Figure 4.14 Cylinder group's i-nodes and data blocks in more detail

Note the following points from Figure 4.14.

- Two directory entries point to the same i-node entry. Every i-node has a link count that contains the number of directory entries that point to it. Only when the link count goes to 0 can the file be deleted (thereby releasing the data blocks associated with the file). This is why the operation of "unlinking a file" does not always mean "deleting the blocks associated with the file." This is why the function that removes a directory entry is called `unlink`, not delete. In the `stat` structure, the link count is contained in the `st_nlink` member. Its primitive system data type is `nlink_t`. These types of links are called hard links. Recall from Section 2.5.2 that the POSIX.1 constant `LINK_MAX` specifies the maximum value for a file's link count.
- The other type of link is called a *symbolic link*. With a symbolic link, the actual contents of the file—the data blocks—store the name of the file that the symbolic link points to. In the following example, the filename in the directory entry is the three-character string `lib` and the 7 bytes of data in the file are `usr/lib`:

  ```
  lrwxrwxrwx  1 root      7 Sep 25 07:14 lib -> usr/lib
  ```

  The file type in the i-node would be `S_IFLNK` so that the system knows that this is a symbolic link.
- The i-node contains all the information about the file: the file type, the file's access permission bits, the size of the file, pointers to the file's data blocks, and so on. Most of the information in the `stat` structure is obtained from the i-node. Only two items of interest are stored in the directory entry: the filename and the i-node number. The other items—the length of the filename and the length of the directory record—are not of interest to this discussion. The data type for the i-node number is `ino_t`.

- Because the i-node number in the directory entry points to an i-node in the same file system, a directory entry can't refer to an i-node in a different file system. This is why the `ln`(1) command (make a new directory entry that points to an existing file) can't cross file systems. We describe the `link` function in the next section.
- When renaming a file without changing file systems, the actual contents of the file need not be moved—all that needs to be done is to add a new directory entry that points to the existing i-node and then unlink the old directory entry. The link count will remain the same. For example, to rename the file `/usr/lib/foo` to `/usr/foo`, the contents of the file `foo` need not be moved if the directories `/usr/lib` and `/usr` are on the same file system. This is how the `mv`(1) command usually operates.

We've talked about the concept of a link count for a regular file, but what about the link count field for a directory? Assume that we make a new directory in the working directory, as in

```
$ mkdir testdir
```

Figure 4.15 shows the result. Note that in this figure, we explicitly show the entries for dot and dot-dot.

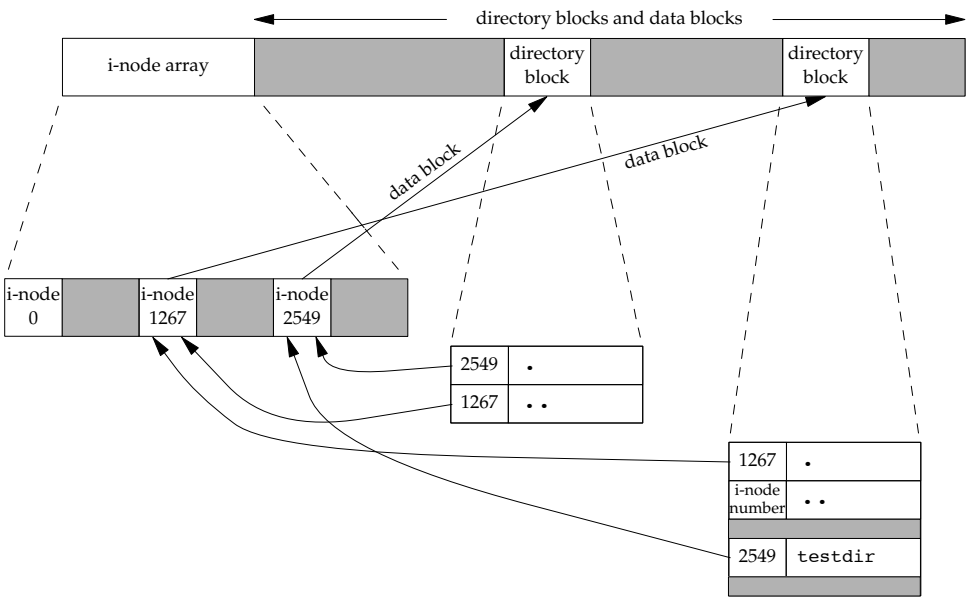


Figure 4.15 Sample cylinder group after creating the directory `testdir`

The i-node whose number is 2549 has a type field of "directory" and a link count equal to 2. Any leaf directory (a directory that does not contain any other directories) always has a link count of 2. The value of 2 comes from the directory entry that names the directory (`testdir`) and from the entry for dot in that directory. The i-node whose

number is 1267 has a type field of "directory" and a link count that is greater than or equal to 3. We know that this link count is greater than or equal to 3 because, at a minimum, the i-node is pointed to from the directory entry that names it (which we don't show in Figure 4.15), from dot, and from dot-dot in the `testdir` directory. Note that every subdirectory in a parent directory causes the parent directory's link count to be increased by 1.

This format is similar to the classic format of the UNIX file system, which is described in detail in Chapter 4 of Bach [1986]. Refer to Chapter 7 of McKusick et al. [1996] or Chapter 8 of McKusick and Neville-Neil [2005] for additional information on the changes made with the Berkeley fast file system. See Chapter 15 of McDougall and Mauro [2007] for details on `UFS`, the Solaris version of the Berkeley fast file system. For information on the `HFS` file system format used in Mac OS X, see Chapter 12 of Singh [2006].

4.15 `link`, `linkat`, `unlink`, `unlinkat`, and `remove` Functions

As we saw in the previous section, a file can have multiple directory entries pointing to its i-node. We can use either the `link` function or the `linkat` function to create a link to an existing file.

```
#include <unistd.h>

int link(const char *existingpath, const char *newpath);

int linkat(int efd, const char *existingpath, int nfd, const char *newpath,
           int flag);
```

Both return: 0 if OK, –1 on error

These functions create a new directory entry, *newpath*, that references the existing file *existingpath*. If the *newpath* already exists, an error is returned. Only the last component of the *newpath* is created. The rest of the path must already exist.

With the `linkat` function, the existing file is specified by both the *efd* and *existingpath* arguments, and the new pathname is specified by both the *nfd* and *newpath* arguments. By default, if either pathname is relative, it is evaluated relative to the corresponding file descriptor. If either file descriptor is set to `AT_FDCWD`, then the corresponding pathname, if it is a relative pathname, is evaluated relative to the current directory. If either pathname is absolute, then the corresponding file descriptor argument is ignored.

When the existing file is a symbolic link, the *flag* argument controls whether the `linkat` function creates a link to the symbolic link or to the file to which the symbolic link points. If the `AT_SYMLINK_FOLLOW` flag is set in the *flag* argument, then a link is created to the target of the symbolic link. If this flag is clear, then a link is created to the symbolic link itself.

The creation of the new directory entry and the increment of the link count must be an atomic operation. (Recall the discussion of atomic operations in Section 3.11.)

Most implementations require that both pathnames be on the same file system, although POSIX.1 allows an implementation to support linking across file systems. If an implementation supports the creation of hard links to directories, it is restricted to only the superuser. This constraint exists because such hard links can cause loops in the file system, which most utilities that process the file system aren't capable of handling. (We show an example of a loop introduced by a symbolic link in Section 4.17.) Many file system implementations disallow hard links to directories for this reason.

To remove an existing directory entry, we call the `unlink` function.

```
#include <unistd.h>

int unlink(const char *pathname);

int unlinkat(int fd, const char *pathname, int flag);
```

Both return: 0 if OK, –1 on error

These functions remove the directory entry and decrement the link count of the file referenced by *pathname*. If there are other links to the file, the data in the file is still accessible through the other links. The file is not changed if an error occurs.

As mentioned earlier, to unlink a file, we must have write permission and execute permission in the directory containing the directory entry, as it is the directory entry that we will be removing. Also, as mentioned in Section 4.10, if the sticky bit is set in this directory we must have write permission for the directory and meet one of the following criteria:

- Own the file
- Own the directory
- Have superuser privileges

Only when the link count reaches 0 can the contents of the file be deleted. One other condition prevents the contents of a file from being deleted: as long as some process has the file open, its contents will not be deleted. When a file is closed, the kernel first checks the count of the number of processes that have the file open. If this count has reached 0, the kernel then checks the link count; if it is 0, the file's contents are deleted.

If the *pathname* argument is a relative pathname, then the `unlinkat` function evaluates the pathname relative to the directory represented by the *fd* file descriptor argument. If the *fd* argument is set to the value `AT_FDCWD`, then the pathname is evaluated relative to the current working directory of the calling process. If the *pathname* argument is an absolute pathname, then the *fd* argument is ignored.

The *flag* argument gives callers a way to change the default behavior of the `unlinkat` function. When the `AT_REMOVEDIR` flag is set, then the `unlinkat` function can be used to remove a directory, similar to using `rmdir`. If this flag is clear, then `unlinkat` operates like `unlink`.

Example

The program shown in Figure 4.16 opens a file and then unlinks it. The program then goes to sleep for 15 seconds before terminating.

```
#include "apue.h"
#include <fcntl.h>

int
main(void)
{
    if (open("tempfile", O_RDWR) < 0)
        err_sys("open error");
    if (unlink("tempfile") < 0)
        err_sys("unlink error");
    printf("file unlinked\n");
    sleep(15);
    printf("done\n");
    exit(0);
}
```

Figure 4.16 Open a file and then `unlink` it

Running this program gives us

```
$ ls -l tempfile                look at how big the file is
-rw-r-----  1 sar     413265408 Jan 21 07:14 tempfile
$ df /home                      check how much free space is available
Filesystem   1K-blocks     Used  Available  Use%  Mounted on
/dev/hda4     11021440  1956332    9065108   18%  /home
$ ./a.out &                     run the program in Figure 4.16 in the background
1364                            the shell prints its process ID
$ file unlinked                 the file is unlinked
ls -l tempfile                  see if the filename is still there
ls: tempfile: No such file or directory     the directory entry is gone
$ df /home                      see if the space is available yet
Filesystem   1K-blocks     Used  Available  Use%  Mounted on
/dev/hda4     11021440  1956332    9065108   18%  /home
$ done                          the program is done, all open files are closed
df /home                        now the disk space should be available
Filesystem   1K-blocks     Used  Available  Use%  Mounted on
/dev/hda4     11021440  1552352    9469088   15%  /home
                                now the 394.1 MB of disk space are available
```

□

This property of `unlink` is often used by a program to ensure that a temporary file it creates won't be left around in case the program crashes. The process creates a file using either `open` or `creat` and then immediately calls `unlink`. The file is not deleted, however, because it is still open. Only when the process either closes the file or terminates, which causes the kernel to close all its open files, is the file deleted.

If *pathname* is a symbolic link, `unlink` removes the symbolic link, not the file referenced by the link. There is no function to remove the file referenced by a symbolic link given the name of the link.

The superuser can call unlink with *pathname* specifying a directory if the file system supports it, but the function rmdir should be used instead to unlink a directory. We describe the rmdir function in Section 4.21.

We can also unlink a file or a directory with the remove function. For a file, remove is identical to unlink. For a directory, remove is identical to rmdir.

```
#include <stdio.h>

int remove(const char *pathname);
```

Returns: 0 if OK, –1 on error

ISO C specifies the remove function to delete a file. The name was changed from the historical UNIX name of unlink because most non-UNIX systems that implement the C standard didn't support the concept of links to a file at the time.

4.16 rename and renameat Functions

A file or a directory is renamed with either the rename or renameat function.

```
#include <stdio.h>

int rename(const char *oldname, const char *newname);

int renameat(int oldfd, const char *oldname, int newfd,
             const char *newname);
```

Both return: 0 if OK, –1 on error

The rename function is defined by ISO C for files. (The C standard doesn't deal with directories.) POSIX.1 expanded the definition to include directories and symbolic links.

There are several conditions to describe for these functions, depending on whether *oldname* refers to a file, a directory, or a symbolic link. We must also describe what happens if *newname* already exists.

1. If *oldname* specifies a file that is not a directory, then we are renaming a file or a symbolic link. In this case, if *newname* exists, it cannot refer to a directory. If *newname* exists and is not a directory, it is removed, and *oldname* is renamed to *newname*. We must have write permission for the directory containing *oldname* and the directory containing *newname*, since we are changing both directories.

2. If *oldname* specifies a directory, then we are renaming a directory. If *newname* exists, it must refer to a directory, and that directory must be empty. (When we say that a directory is empty, we mean that the only entries in the directory are dot and dot-dot.) If *newname* exists and is an empty directory, it is removed, and *oldname* is renamed to *newname*. Additionally, when we're renaming a directory, *newname* cannot contain a path prefix that names *oldname*. For example, we can't rename /usr/foo to /usr/foo/testdir, because the old name (/usr/foo) is a path prefix of the new name and cannot be removed.

3. If either *oldname* or *newname* refers to a symbolic link, then the link itself is processed, not the file to which it resolves.
4. We can't rename dot or dot-dot. More precisely, neither dot nor dot-dot can appear as the last component of *oldname* or *newname*.
5. As a special case, if *oldname* and *newname* refer to the same file, the function returns successfully without changing anything.

If *newname* already exists, we need permissions as if we were deleting it. Also, because we're removing the directory entry for *oldname* and possibly creating a directory entry for *newname*, we need write permission and execute permission in the directory containing *oldname* and in the directory containing *newname*.

The `renameat` function provides the same functionality as the `rename` function, except when either *oldname* or *newname* refers to a relative pathname. If *oldname* specifies a relative pathname, it is evaluated relative to the directory referenced by *oldfd*. Similarly, *newname* is evaluated relative to the directory referenced by *newfd* if *newname* specifies a relative pathname. Either the *oldfd* or *newfd* arguments (or both) can be set to `AT_FDCWD` to evaluate the corresponding pathname relative to the current directory.

4.17 Symbolic Links

A symbolic link is an indirect pointer to a file, unlike the hard links described in the previous section, which pointed directly to the i-node of the file. Symbolic links were introduced to get around the limitations of hard links.

- Hard links normally require that the link and the file reside in the same file system.
- Only the superuser can create a hard link to a directory (when supported by the underlying file system).

There are no file system limitations on a symbolic link and what it points to, and anyone can create a symbolic link to a directory. Symbolic links are typically used to "move" a file or an entire directory hierarchy to another location on a system.

When using functions that refer to a file by name, we always need to know whether the function follows a symbolic link. If the function follows a symbolic link, a pathname argument to the function refers to the file pointed to by the symbolic link. Otherwise, a pathname argument refers to the link itself, not the file pointed to by the link. Figure 4.17 summarizes whether the functions described in this chapter follow a symbolic link. The functions `mkdir`, `mkfifo`, `mknod`, and `rmdir` do not appear in this figure, as they return an error when the pathname is a symbolic link. Also, the functions that take a file descriptor argument, such as `fstat` and `fchmod`, are not listed, as the function that returns the file descriptor (usually `open`) handles the symbolic link. Historically, implementations have differed in whether `chown` follows symbolic links. In all modern systems, however, `chown` does follow symbolic links.

Symbolic links were introduced with 4.2BSD. Initially, `chown` didn't follow symbolic links, but this behavior was changed in 4.4BSD. System V included support for symbolic links in SVR4, but diverged from the original BSD behavior by implementing `chown` to follow symbolic links. In older versions of Linux (those before version 2.1.81), `chown` didn't follow symbolic links. From version 2.1.81 onward, `chown` follows symbolic links. With FreeBSD 8.0, Mac OS X 10.6.8, and Solaris 10, `chown` follows symbolic links. All of these platforms provide implementations of `lchown` to change the ownership of symbolic links themselves.

Function	Does not follow symbolic link	Follows symbolic link
access		•
chdir		•
chmod		•
chown		•
creat		•
exec		•
lchown	•	
link		•
lstat	•	
open		•
opendir		•
pathconf		•
readlink	•	
remove	•	
rename	•	
stat		•
truncate		•
unlink	•	

Figure 4.17 Treatment of symbolic links by various functions

One exception to the behavior summarized in Figure 4.17 occurs when the `open` function is called with both `O_CREAT` and `O_EXCL` set. In this case, if the pathname refers to a symbolic link, `open` will fail with `errno` set to `EEXIST`. This behavior is intended to close a security hole so that privileged processes can't be fooled into writing to the wrong files.

Example

It is possible to introduce loops into the file system by using symbolic links. Most functions that look up a pathname return an `errno` of `ELOOP` when this occurs. Consider the following commands:

```
$ mkdir foo                          make a new directory
$ touch foo/a                        create a 0-length file
$ ln -s ../foo foo/testdir           create a symbolic link
$ ls -l foo
total 0
-rw-r-----  1 sar           0 Jan 22 00:16 a
lrwxrwxrwx  1 sar           6 Jan 22 00:16 testdir -> ../foo
```

This creates a directory `foo` that contains the file `a` and a symbolic link that points to `foo`. We show this arrangement in Figure 4.18, drawing a directory as a circle and a file as a square.

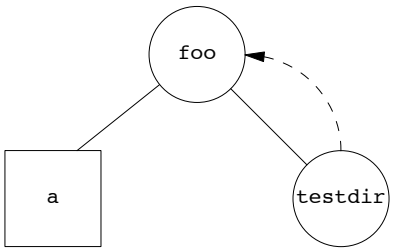


Figure 4.18 Symbolic link `testdir` that creates a loop

If we write a simple program that uses the standard function `ftw`(3) on Solaris to descend through a file hierarchy, printing each pathname encountered, the output is

```
foo
foo/a
foo/testdir
foo/testdir/a
foo/testdir/testdir
foo/testdir/testdir/a
foo/testdir/testdir/testdir
foo/testdir/testdir/testdir/a
    (many more lines until we encounter an ELOOP error)
```

In Section 4.22, we provide our own version of the `ftw` function that uses `lstat` instead of `stat`, to prevent it from following symbolic links.

> Note that on Linux, the `ftw` and `nftw` functions record all directories seen and avoid processing a directory more than once, so they don't display this behavior.

A loop of this form is easy to remove. We can `unlink` the file `foo/testdir`, as `unlink` does not follow a symbolic link. But if we create a hard link that forms a loop of this type, its removal is much more difficult. This is why the `link` function will not form a hard link to a directory unless the process has superuser privileges.

> Indeed, Rich Stevens did this on his own system as an experiment while writing the original version of this section. The file system got corrupted and the normal `fsck`(1) utility couldn't fix things. The deprecated tools `clri`(8) and `dcheck`(8) were needed to repair the file system.
>
> The need for hard links to directories has long since passed. With symbolic links and the `mkdir` function, there is no longer any need for users to create hard links to directories.

When we open a file, if the pathname passed to `open` specifies a symbolic link, `open` follows the link to the specified file. If the file pointed to by the symbolic link doesn't exist, `open` returns an error saying that it can't open the file. This response can confuse users who aren't familiar with symbolic links. For example,

```
$ ln -s /no/such/file myfile              create a symbolic link
$ ls myfile
myfile                                    ls says it's there
$ cat myfile                              so we try to look at it
cat: myfile: No such file or directory
$ ls -l myfile                            try -l option
lrwxrwxrwx  1 sar       13 Jan 22 00:26 myfile -> /no/such/file
```

The file `myfile` does exist, yet `cat` says there is no such file, because `myfile` is a symbolic link and the file pointed to by the symbolic link doesn't exist. The `-l` option to `ls` gives us two hints: the first character is an `l`, which means a symbolic link, and the sequence `->` also indicates a symbolic link. The `ls` command has another option (`-F`) that appends an at-sign (@) to filenames that are symbolic links, which can help us spot symbolic links in a directory listing without the `-l` option. □

4.18 Creating and Reading Symbolic Links

A symbolic link is created with either the `symlink` or `symlinkat` function.

```
#include <unistd.h>

int symlink(const char *actualpath, const char *sympath);

int symlinkat(const char *actualpath, int fd, const char *sympath);
```

Both return: 0 if OK, –1 on error

A new directory entry, *sympath*, is created that points to *actualpath*. It is not required that *actualpath* exist when the symbolic link is created. (We saw this in the example at the end of the previous section.) Also, *actualpath* and *sympath* need not reside in the same file system.

The `symlinkat` function is similar to `symlink`, but the *sympath* argument is evaluated relative to the directory referenced by the open file descriptor for that directory (specified by the *fd* argument). If the *sympath* argument specifies an absolute pathname or if the *fd* argument has the special value `AT_FDCWD`, then `symlinkat` behaves the same way as `symlink`.

Because the `open` function follows a symbolic link, we need a way to open the link itself and read the name in the link. The `readlink` and `readlinkat` functions do this.

```
#include <unistd.h>

ssize_t readlink(const char* restrict pathname, char *restrict buf,
                 size_t bufsize);

ssize_t readlinkat(int fd, const char* restrict pathname,
                   char *restrict buf, size_t bufsize);
```

Both return: number of bytes read if OK, –1 on error

These functions combine the actions of `open`, `read`, and `close`. If successful, they return the number of bytes placed into *buf*. The contents of the symbolic link that are returned in *buf* are not null terminated.

The `readlinkat` function behaves the same way as the `readlink` function when the *pathname* argument specifies an absolute pathname or when the *fd* argument has the special value `AT_FDCWD`. However, when the *fd* argument is a valid file descriptor of an open directory and the *pathname* argument is a relative pathname, then `readlinkat` evaluates the pathname relative to the open directory represented by *fd*.

4.19 File Times

In Section 4.2, we discussed how the 2008 version of the Single UNIX Specification increased the resolution of the time fields in the `stat` structure from seconds to seconds plus nanoseconds. The actual resolution stored with each file's attributes depends on the file system implementation. For file systems that store timestamps in second granularity, the nanoseconds fields will be filled with zeros. For file systems that store timestamps in a resolution higher than seconds, the partial seconds value will be converted into nanoseconds and returned in the nanoseconds fields.

Three time fields are maintained for each file. Their purpose is summarized in Figure 4.19.

Field	Description	Example	`ls`(1) option
`st_atim`	last-access time of file data	`read`	`-u`
`st_mtim`	last-modification time of file data	`write`	default
`st_ctim`	last-change time of i-node status	`chmod, chown`	`-c`

Figure 4.19 The three time values associated with each file

Note the difference between the modification time (`st_mtim`) and the changed-status time (`st_ctim`). The modification time indicates when the contents of the file were last modified. The changed-status time indicates when the i-node of the file was last modified. In this chapter, we've described many operations that affect the i-node without changing the actual contents of the file: changing the file access permissions, changing the user ID, changing the number of links, and so on. Because all the information in the i-node is stored separately from the actual contents of the file, we need the changed-status time, in addition to the modification time.

Note that the system does not maintain the last-access time for an i-node. This is why the functions `access` and `stat`, for example, don't change any of the three times.

The access time is often used by system administrators to delete files that have not been accessed for a certain amount of time. The classic example is the removal of files named `a.out` or `core` that haven't been accessed in the past week. The `find`(1) command is often used for this type of operation.

The modification time and the changed-status time can be used to archive only those files that have had their contents modified or their i-node modified.

The `ls` command displays or sorts only on one of the three time values. By default, when invoked with either the `-l` or the `-t` option, it uses the modification time of a file. The `-u` option causes the `ls` command to use the access time, and the `-c` option causes it to use the changed-status time.

Figure 4.20 summarizes the effects of the various functions that we've described on these three times. Recall from Section 4.14 that a directory is simply a file containing directory entries: filenames and associated i-node numbers. Adding, deleting, or modifying these directory entries can affect the three times associated with that directory. This is why Figure 4.20 contains one column for the three times associated with the file or directory and another column for the three times associated with the parent directory of the referenced file or directory. For example, creating a new file affects the directory that contains the new file, and it affects the i-node for the new file. Reading or writing a file, however, affects only the i-node of the file and has no effect on the directory.

Function	Referenced file or directory			Parent directory of referenced file or directory			Section	Note
	a	m	c	a	m	c		
`chmod`, `fchmod`			•				4.9	
`chown`, `fchown`			•				4.11	
`creat`	•	•	•		•	•	3.4	`O_CREAT` new file
`creat`		•	•				3.4	`O_TRUNC` existing file
`exec`	•						8.10	
`lchown`			•				4.11	
`link`			•		•	•	4.15	parent of second argument
`mkdir`	•	•	•		•	•	4.21	
`mkfifo`	•	•	•		•	•	15.5	
`open`	•	•	•		•	•	3.3	`O_CREAT` new file
`open`		•	•				3.3	`O_TRUNC` existing file
`pipe`	•	•	•				15.2	
`read`	•						3.7	
`remove`			•		•	•	4.15	remove file = `unlink`
`remove`					•	•	4.15	remove directory = `rmdir`
`rename`			•		•	•	4.16	for both arguments
`rmdir`					•	•	4.21	
`truncate`, `ftruncate`		•	•				4.13	
`unlink`			•		•	•	4.15	
`utimes`, `utimensat`, `futimens`	•	•	•				4.20	
`write`		•	•				3.8	

Figure 4.20 Effect of various functions on the access, modification, and changed-status times

(The `mkdir` and `rmdir` functions are covered in Section 4.21. The `utimes`, `utimensat`, and `futimens` functions are covered in the next section. The seven `exec` functions are described in Section 8.10. We describe the `mkfifo` and `pipe` functions in Chapter 15.)

4.20 `futimens`, `utimensat`, and `utimes` Functions

Several functions are available to change the access time and the modification time of a file. The `futimens` and `utimensat` functions provide nanosecond granularity for specifying timestamps, using the `timespec` structure (the same structure used by the `stat` family of functions; see Section 4.2).

```
#include <sys/stat.h>

int futimens(int fd, const struct timespec times[2]);

int utimensat(int fd, const char *path, const struct timespec times[2],
              int flag);
```

Both return: 0 if OK, –1 on error

In both functions, the first element of the *times* array argument contains the access time, and the second element contains the modification time. The two time values are calendar times, which count seconds since the Epoch, as described in Section 1.10. Partial seconds are expressed in nanoseconds.

Timestamps can be specified in one of four ways:

1. The *times* argument is a null pointer. In this case, both timestamps are set to the current time.
2. The *times* argument points to an array of two `timespec` structures. If either `tv_nsec` field has the special value `UTIME_NOW`, the corresponding timestamp is set to the current time. The corresponding `tv_sec` field is ignored.
3. The *times* argument points to an array of two `timespec` structures. If either `tv_nsec` field has the special value `UTIME_OMIT`, then the corresponding timestamp is unchanged. The corresponding `tv_sec` field is ignored.
4. The *times* argument points to an array of two `timespec` structures and the `tv_nsec` field contains a value other than `UTIME_NOW` or `UTIME_OMIT`. In this case, the corresponding timestamp is set to the value specified by the corresponding `tv_sec` and `tv_nsec` fields.

The privileges required to execute these functions depend on the value of the *times* argument.

- If *times* is a null pointer or if either `tv_nsec` field is set to `UTIME_NOW`, either the effective user ID of the process must equal the owner ID of the file, the process must have write permission for the file, or the process must be a superuser process.
- If *times* is a non-null pointer and either `tv_nsec` field has a value other than `UTIME_NOW` or `UTIME_OMIT`, the effective user ID of the process must equal the owner ID of the file, or the process must be a superuser process. Merely having write permission for the file is not adequate.

- If *times* is a non-null pointer and both tv_nsec fields are set to UTIME_OMIT, no permissions checks are performed.

With futimens, you need to open the file to change its times. The utimensat function provides a way to change a file's times using the file's name. The *pathname* argument is evaluated relative to the *fd* argument, which is either a file descriptor of an open directory or the special value AT_FDCWD to force evaluation relative to the current directory of the calling process. If *pathname* specifies an absolute pathname, then the *fd* argument is ignored.

The *flag* argument to utimensat can be used to further modify the default behavior. If the AT_SYMLINK_NOFOLLOW flag is set, then the times of the symbolic link itself are changed (if the pathname refers to a symbolic link). The default behavior is to follow a symbolic link and modify the times of the file to which the link refers.

Both futimens and utimensat are included in POSIX.1. A third function, utimes, is included in the Single UNIX Specification as part of the XSI option.

```
#include <sys/time.h>

int utimes(const char *pathname, const struct timeval times[2]);
```

Returns: 0 if OK, –1 on error

The utimes function operates on a pathname. The *times* argument is a pointer to an array of two timestamps—access time and modification time—but they are expressed in seconds and microseconds:

```
struct timeval {
        time_t tv_sec;    /* seconds */
        long   tv_usec;   /* microseconds */
};
```

Note that we are unable to specify a value for the changed-status time, st_ctim—the time the i-node was last changed—as this field is automatically updated when the utime function is called.

On some versions of the UNIX System, the touch(1) command uses one of these functions. Also, the standard archive programs, tar(1) and cpio(1), optionally call these functions to set a file's times to the time values saved when the file was archived.

Example

The program shown in Figure 4.21 truncates files to zero length using the O_TRUNC option of the open function, but does not change their access time or modification time. To do this, the program first obtains the times with the stat function, truncates the file, and then resets the times with the futimens function.

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
```

```
{
    int             i, fd;
    struct stat     statbuf;
    struct timespec times[2];

    for (i = 1; i < argc; i++) {
        if (stat(argv[i], &statbuf) < 0) {  /* fetch current times */
            err_ret("%s: stat error", argv[i]);
            continue;
        }
        if ((fd = open(argv[i], O_RDWR | O_TRUNC)) < 0) { /* truncate */
            err_ret("%s: open error", argv[i]);
            continue;
        }
        times[0] = statbuf.st_atim;
        times[1] = statbuf.st_mtim;
        if (futimens(fd, times) < 0)        /* reset times */
            err_ret("%s: futimens error", argv[i]);
        close(fd);
    }
    exit(0);
}
```

Figure 4.21 Example of `futimens` function

We can demonstrate the program in Figure 4.21 on Linux with the following commands:

```
$ ls -l changemod times          look at sizes and last-modification times
-rwxr-xr-x  1 sar   13792 Jan 22 01:26 changemod
-rwxr-xr-x  1 sar   13824 Jan 22 01:26 times
$ ls -lu changemod times         look at last-access times
-rwxr-xr-x  1 sar   13792 Jan 22 22:22 changemod
-rwxr-xr-x  1 sar   13824 Jan 22 22:22 times
$ date                           print today's date
Fri Jan 27 20:53:46 EST 2012
$ ./a.out changemod times        run the program in Figure 4.21
$ ls -l changemod times          and check the results
-rwxr-xr-x  1 sar       0 Jan 22 01:26 changemod
-rwxr-xr-x  1 sar       0 Jan 22 01:26 times
$ ls -lu changemod times         check the last-access times also
-rwxr-xr-x  1 sar       0 Jan 22 22:22 changemod
-rwxr-xr-x  1 sar       0 Jan 22 22:22 times
$ ls -lc changemod times         and the changed-status times
-rwxr-xr-x  1 sar       0 Jan 27 20:53 changemod
-rwxr-xr-x  1 sar       0 Jan 27 20:53 times
```

As we would expect, the last-modification times and the last-access times have not changed. The changed-status times, however, have changed to the time that the program was run. □

4.21 mkdir, mkdirat, and rmdir Functions

Directories are created with the mkdir and mkdirat functions, and deleted with the rmdir function.

```
#include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);

int mkdirat(int fd, const char *pathname, mode_t mode);
```

Both return: 0 if OK, –1 on error

These functions create a new, empty directory. The entries for dot and dot-dot are created automatically. The specified file access permissions, *mode*, are modified by the file mode creation mask of the process.

A common mistake is to specify the same *mode* as for a file: read and write permissions only. But for a directory, we normally want at least one of the execute bits enabled, to allow access to filenames within the directory. (See Exercise 4.16.)

The user ID and group ID of the new directory are established according to the rules we described in Section 4.6.

> Solaris 10 and Linux 3.2.0 also have the new directory inherit the set-group-ID bit from the parent directory. Files created in the new directory will then inherit the group ID of that directory. With Linux, the file system implementation determines whether this behavior is supported. For example, the ext2, ext3, and ext4 file systems allow this behavior to be controlled by an option to the mount(1) command. With the Linux implementation of the UFS file system, however, the behavior is not selectable; it inherits the set-group-ID bit to mimic the historical BSD implementation, where the group ID of a directory is inherited from the parent directory.
>
> BSD-based implementations don't propagate the set-group-ID bit; they simply inherit the group ID as a matter of policy. Because FreeBSD 8.0 and Mac OS X 10.6.8 are based on 4.4BSD, they do not require inheriting the set-group-ID bit. On these platforms, newly created files and directories always inherit the group ID of the parent directory, regardless of whether the set-group-ID bit is set.
>
> Earlier versions of the UNIX System did not have the mkdir function; it was introduced with 4.2BSD and SVR3. In the earlier versions, a process had to call the mknod function to create a new directory—but use of the mknod function was restricted to superuser processes. To circumvent this constraint, the normal command that created a directory, mkdir(1), had to be owned by root with the set-user-ID bit on. To create a directory from a process, the mkdir(1) command had to be invoked with the system(3) function.

The mkdirat function is similar to the mkdir function. When the *fd* argument has the special value AT_FDCWD, or when the *pathname* argument specifies an absolute pathname, mkdirat behaves exactly like mkdir. Otherwise, the *fd* argument is an open directory from which relative pathnames will be evaluated.

An empty directory is deleted with the rmdir function. Recall that an empty directory is one that contains entries only for dot and dot-dot.

```
#include <unistd.h>
int rmdir(const char *pathname);
                                                    Returns: 0 if OK, –1 on error
```

If the link count of the directory becomes 0 with this call, and if no other process has the directory open, then the space occupied by the directory is freed. If one or more processes have the directory open when the link count reaches 0, the last link is removed and the dot and dot-dot entries are removed before this function returns. Additionally, no new files can be created in the directory. The directory is not freed, however, until the last process closes it. (Even though some other process has the directory open, it can't be doing much in the directory, as the directory had to be empty for the `rmdir` function to succeed.)

4.22 Reading Directories

Directories can be read by anyone who has access permission to read the directory. But only the kernel can write to a directory, to preserve file system sanity. Recall from Section 4.5 that the write permission bits and execute permission bits for a directory determine if we can create new files in the directory and remove files from the directory—they don't specify if we can write to the directory itself.

The actual format of a directory depends on the UNIX System implementation and the design of the file system. Earlier systems, such as Version 7, had a simple structure: each directory entry was 16 bytes, with 14 bytes for the filename and 2 bytes for the i-node number. When longer filenames were added to 4.2BSD, each entry became variable length, which means that any program that reads a directory is now system dependent. To simplify the process of reading a directory, a set of directory routines were developed and are part of POSIX.1. Many implementations prevent applications from using the `read` function to access the contents of directories, thereby further isolating applications from the implementation-specific details of directory formats.

```
#include <dirent.h>
DIR *opendir(const char *pathname);
DIR *fdopendir(int fd);
                                        Both return: pointer if OK, NULL on error
struct dirent *readdir(DIR *dp);
                          Returns: pointer if OK, NULL at end of directory or error
void rewinddir(DIR *dp);
int closedir(DIR *dp);
                                                    Returns: 0 if OK, –1 on error
long telldir(DIR *dp);
                          Returns: current location in directory associated with dp
void seekdir(DIR *dp, long loc);
```

The fdopendir function first appeared in version 4 of the Single UNIX Specification. It provides a way to convert an open file descriptor into a DIR structure for use by the directory handling functions.

The telldir and seekdir functions are not part of the base POSIX.1 standard. They are included in the XSI option in the Single UNIX Specification, so all conforming UNIX System implementations are expected to provide them.

Recall our use of several of these functions in the program shown in Figure 1.3, our bare-bones implementation of the ls command.

The dirent structure defined in <dirent.h> is implementation dependent. Implementations define the structure to contain at least the following two members:

```
ino_t  d_ino;                  /* i-node number */
char   d_name[];               /* null-terminated filename */
```

> The d_ino entry is not defined by POSIX.1, because it is an implementation feature, but it is defined as part of the XSI option in POSIX.1. POSIX.1 defines only the d_name entry in this structure.

Note that the size of the d_name entry isn't specified, but it is guaranteed to hold at least NAME_MAX characters, not including the terminating null byte (recall Figure 2.15.) Since the filename is null terminated, however, it doesn't matter how d_name is defined in the header, because the array size doesn't indicate the length of the filename.

The DIR structure is an internal structure used by these seven functions to maintain information about the directory being read. The purpose of the DIR structure is similar to that of the FILE structure maintained by the standard I/O library, which we describe in Chapter 5.

The pointer to a DIR structure returned by opendir and fdopendir is then used with the other five functions. The opendir function initializes things so that the first readdir returns the first entry in the directory. When the DIR structure is created by fdopendir, the first entry returned by readdir depends on the file offset associated with the file descriptor passed to fdopendir. Note that the ordering of entries within the directory is implementation dependent and is usually not alphabetical.

Example

We'll use these directory routines to write a program that traverses a file hierarchy. The goal is to produce a count of the various types of files shown in Figure 4.4. The program shown in Figure 4.22 takes a single argument—the starting pathname—and recursively descends the hierarchy from that point. Solaris provides a function, ftw(3), that performs the actual traversal of the hierarchy, calling a user-defined function for each file. The problem with this function is that it calls the stat function for each file, which causes the program to follow symbolic links. For example, if we start at the root and have a symbolic link named /lib that points to /usr/lib, all the files in the directory /usr/lib are counted twice. To correct this problem, Solaris provides an additional function, nftw(3), with an option that stops it from following symbolic links. Although we could use nftw, we'll write our own simple file walker to show the use of the directory routines.

In SUSv4, `nftw` is included as part of the XSI option. Implementations are included in FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10. (In SUSv4, the `ftw` function has been marked as obsolescent.) BSD-based systems have a different function, `fts`(3), that provides similar functionality. It is available in FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8.

```
#include "apue.h"
#include <dirent.h>
#include <limits.h>

/* function type that is called for each filename */
typedef int Myfunc(const char *, const struct stat *, int);

static Myfunc   myfunc;
static int      myftw(char *, Myfunc *);
static int      dopath(Myfunc *);

static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;

int
main(int argc, char *argv[])
{
    int     ret;

    if (argc != 2)
        err_quit("usage:  ftw  <starting-pathname>");

    ret = myftw(argv[1], myfunc);       /* does it all */

    ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock;
    if (ntot == 0)
        ntot = 1;       /* avoid divide by 0; print 0 for all counts */
    printf("regular files  = %7ld, %5.2f %%\n", nreg,
      nreg*100.0/ntot);
    printf("directories    = %7ld, %5.2f %%\n", ndir,
      ndir*100.0/ntot);
    printf("block special  = %7ld, %5.2f %%\n", nblk,
      nblk*100.0/ntot);
    printf("char special   = %7ld, %5.2f %%\n", nchr,
      nchr*100.0/ntot);
    printf("FIFOs          = %7ld, %5.2f %%\n", nfifo,
      nfifo*100.0/ntot);
    printf("symbolic links = %7ld, %5.2f %%\n", nslink,
      nslink*100.0/ntot);
    printf("sockets        = %7ld, %5.2f %%\n", nsock,
      nsock*100.0/ntot);
    exit(ret);
}

/*
 * Descend through the hierarchy, starting at "pathname".
 * The caller's func() is called for every file.
 */
#define FTW_F   1       /* file other than directory */
#define FTW_D   2       /* directory */
```

```
#define FTW_DNR 3       /* directory that can't be read */
#define FTW_NS  4       /* file that we can't stat */

static char *fullpath;      /* contains full pathname for every file */
static size_t pathlen;

static int                  /* we return whatever func() returns */
myftw(char *pathname, Myfunc *func)
{
    fullpath = path_alloc(&pathlen);    /* malloc PATH_MAX+1 bytes */
                                        /* (Figure 2.16) */
    if (pathlen <= strlen(pathname)) {
        pathlen = strlen(pathname) * 2;
        if ((fullpath = realloc(fullpath, pathlen)) == NULL)
            err_sys("realloc failed");
    }
    strcpy(fullpath, pathname);
    return(dopath(func));
}

/*
 * Descend through the hierarchy, starting at "fullpath".
 * If "fullpath" is anything other than a directory, we lstat() it,
 * call func(), and return.  For a directory, we call ourself
 * recursively for each name in the directory.
 */
static int                  /* we return whatever func() returns */
dopath(Myfunc* func)
{
    struct stat     statbuf;
    struct dirent   *dirp;
    DIR             *dp;
    int             ret, n;

    if (lstat(fullpath, &statbuf) < 0)  /* stat error */
        return(func(fullpath, &statbuf, FTW_NS));
    if (S_ISDIR(statbuf.st_mode) == 0)  /* not a directory */
        return(func(fullpath, &statbuf, FTW_F));

    /*
     * It's a directory.  First call func() for the directory,
     * then process each filename in the directory.
     */
    if ((ret = func(fullpath, &statbuf, FTW_D)) != 0)
        return(ret);

    n = strlen(fullpath);
    if (n + NAME_MAX + 2 > pathlen) {   /* expand path buffer */
        pathlen *= 2;
        if ((fullpath = realloc(fullpath, pathlen)) == NULL)
            err_sys("realloc failed");
    }
    fullpath[n++] = '/';
```

```
    fullpath[n] = 0;

    if ((dp = opendir(fullpath)) == NULL)   /* can't read directory */
        return(func(fullpath, &statbuf, FTW_DNR));

    while ((dirp = readdir(dp)) != NULL) {
        if (strcmp(dirp->d_name, ".") == 0  ||
            strcmp(dirp->d_name, "..") == 0)
                continue;       /* ignore dot and dot-dot */
        strcpy(&fullpath[n], dirp->d_name); /* append name after "/" */
        if ((ret = dopath(func)) != 0)      /* recursive */
            break;  /* time to leave */
    }
    fullpath[n-1] = 0;  /* erase everything from slash onward */

    if (closedir(dp) < 0)
        err_ret("can't close directory %s", fullpath);
    return(ret);
}

static int
myfunc(const char *pathname, const struct stat *statptr, int type)
{
    switch (type) {
    case FTW_F:
        switch (statptr->st_mode & S_IFMT) {
        case S_IFREG:   nreg++;     break;
        case S_IFBLK:   nblk++;     break;
        case S_IFCHR:   nchr++;     break;
        case S_IFIFO:   nfifo++;    break;
        case S_IFLNK:   nslink++;   break;
        case S_IFSOCK:  nsock++;    break;
        case S_IFDIR:   /* directories should have type = FTW_D */
            err_dump("for S_IFDIR for %s", pathname);
        }
        break;
    case FTW_D:
        ndir++;
        break;
    case FTW_DNR:
        err_ret("can't read directory %s", pathname);
        break;
    case FTW_NS:
        err_ret("stat error for %s", pathname);
        break;
    default:
        err_dump("unknown type %d for pathname %s", type, pathname);
    }
    return(0);
}
```

Figure 4.22 Recursively descend a directory hierarchy, counting file types

To illustrate the ftw and nftw functions, we have provided more generality in this program than needed. For example, the function myfunc always returns 0, even though the function that calls it is prepared to handle a nonzero return. □

For additional information on descending through a file system and using this technique in many standard UNIX System commands—find, ls, tar, and so on—refer to Fowler, Korn, and Vo [1989].

4.23 chdir, fchdir, and getcwd Functions

Every process has a current working directory. This directory is where the search for all relative pathnames starts (i.e., with all pathnames that do not begin with a slash). When a user logs in to a UNIX system, the current working directory normally starts at the directory specified by the sixth field in the /etc/passwd file—the user's home directory. The current working directory is an attribute of a process; the home directory is an attribute of a login name.

We can change the current working directory of the calling process by calling the chdir or fchdir function.

```
#include <unistd.h>

int chdir(const char *pathname);

int fchdir(int fd);
```

Both return: 0 if OK, –1 on error

We can specify the new current working directory either as a *pathname* or through an open file descriptor.

Example

Because it is an attribute of a process, the current working directory cannot affect processes that invoke the process that executes the chdir. (We describe the relationship between processes in more detail in Chapter 8.) As a result, the program in Figure 4.23 doesn't do what we might expect.

```
#include "apue.h"

int
main(void)
{
    if (chdir("/tmp") < 0)
        err_sys("chdir failed");
    printf("chdir to /tmp succeeded\n");
    exit(0);
}
```

Figure 4.23 Example of chdir function

If we compile this program, call the executable `mycd`, and run it, we get the following:

```
$ pwd
/usr/lib
$ mycd
chdir to /tmp succeeded
$ pwd
/usr/lib
```

The current working directory for the shell that executed the `mycd` program didn't change. This is a side effect of the way that the shell executes programs. Each program is run in a separate process, so the current working directory of the shell is unaffected by the call to `chdir` in the program. For this reason, the `chdir` function has to be called directly from the shell, so the `cd` command is built into the shells. □

Because the kernel must maintain knowledge of the current working directory, we should be able to fetch its current value. Unfortunately, the kernel doesn't maintain the full pathname of the directory. Instead, the kernel keeps information about the directory, such as a pointer to the directory's v-node.

> The Linux kernel can determine the full pathname. Its components are distributed throughout the mount table and the dcache table, and are reassembled, for example, when you read the `/proc/self/cwd` symbolic link.

What we need is a function that starts at the current working directory (dot) and works its way up the directory hierarchy, using dot-dot to move up one level. At each level, the function reads the directory entries until it finds the name that corresponds to the i-node of the directory that it just came from. Repeating this procedure until the root is encountered yields the entire absolute pathname of the current working directory. Fortunately, a function already exists that does this work for us.

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

Returns: *buf* if OK, `NULL` on error

We must pass to this function the address of a buffer, *buf*, and its *size* (in bytes). The buffer must be large enough to accommodate the absolute pathname plus a terminating null byte, or else an error will be returned. (Recall the discussion of allocating space for a maximum-sized pathname in Section 2.5.5.)

> Some older implementations of `getcwd` allow the first argument *buf* to be `NULL`. In this case, the function calls `malloc` to allocate *size* number of bytes dynamically. This is not part of POSIX.1 or the Single UNIX Specification and should be avoided.

Example

The program in Figure 4.24 changes to a specific directory and then calls `getcwd` to print the working directory. If we run the program, we get

```
$ ./a.out
cwd = /var/spool/uucppublic
$ ls -l /usr/spool
lrwxrwxrwx  1 root  12 Jan 31 07:57 /usr/spool -> ../var/spool
```

```
#include "apue.h"

int
main(void)
{
    char    *ptr;
    size_t      size;

    if (chdir("/usr/spool/uucppublic") < 0)
        err_sys("chdir failed");

    ptr = path_alloc(&size);    /* our own function */
    if (getcwd(ptr, size) == NULL)
        err_sys("getcwd failed");

    printf("cwd = %s\n", ptr);
    exit(0);
}
```

Figure 4.24 Example of `getcwd` function

Note that `chdir` follows the symbolic link—as we expect it to, from Figure 4.17—but when it goes up the directory tree, `getcwd` has no idea when it hits the `/var/spool` directory that it is pointed to by the symbolic link `/usr/spool`. This is a characteristic of symbolic links. □

The `getcwd` function is useful when we have an application that needs to return to the location in the file system where it started out. We can save the starting location by calling `getcwd` before we change our working directory. After we complete our processing, we can pass the pathname obtained from `getcwd` to `chdir` to return to our starting location in the file system.

The `fchdir` function provides us with an easy way to accomplish this task. Instead of calling `getcwd`, we can `open` the current directory and save the file descriptor before we change to a different location in the file system. When we want to return to where we started, we can simply pass the file descriptor to `fchdir`.

4.24 Device Special Files

The two fields `st_dev` and `st_rdev` are often confused. We'll need to use these fields in Section 18.9 when we write the `ttyname` function. The rules for their use are simple.

- Every file system is known by its major and minor device numbers, which are encoded in the primitive system data type `dev_t`. The major number identifies the device driver and sometimes encodes which peripheral board to communicate with; the minor number identifies the specific subdevice. Recall from Figure 4.13 that a disk drive often contains several file systems. Each file system on the same disk drive would usually have the same major number, but a different minor number.

- We can usually access the major and minor device numbers through two macros defined by most implementations: `major` and `minor`. Consequently, we don't care how the two numbers are stored in a `dev_t` object.

  Early systems stored the device number in a 16-bit integer, with 8 bits for the major number and 8 bits for the minor number. FreeBSD 8.0 and Mac OS X 10.6.8 use a 32-bit integer, with 8 bits for the major number and 24 bits for the minor number. On 32-bit systems, Solaris 10 uses a 32-bit integer for `dev_t`, with 14 bits designated as the major number and 18 bits designated as the minor number. On 64-bit systems, Solaris 10 represents `dev_t` as a 64-bit integer, with 32 bits for each number. On Linux 3.2.0, although `dev_t` is a 64-bit integer, only 12 bits are used for the major number and 20 bits are used for the minor number.

  POSIX.1 states that the `dev_t` type exists, but doesn't define what it contains or how to get at its contents. The macros `major` and `minor` are defined by most implementations. Which header they are defined in depends on the system. They can be found in `<sys/types.h>` on BSD-based systems. Solaris defines their function prototypes in `<sys/mkdev.h>`, because the macro definitions in `<sys/sysmacros.h>` are considered obsolete in Solaris. Linux defines these macros in `<sys/sysmacros.h>`, which is included by `<sys/types.h>`.

- The `st_dev` value for every filename on a system is the device number of the file system containing that filename and its corresponding i-node.
- Only character special files and block special files have an `st_rdev` value. This value contains the device number for the actual device.

Example

The program in Figure 4.25 prints the device number for each command-line argument. Additionally, if the argument refers to a character special file or a block special file, the `st_rdev` value for the special file is printed.

```
#include "apue.h"
#ifdef SOLARIS
#include <sys/mkdev.h>
#endif

int
main(int argc, char *argv[])
{
    int         i;
    struct stat buf;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (stat(argv[i], &buf) < 0) {
            err_ret("stat error");
            continue;
        }

        printf("dev = %d/%d", major(buf.st_dev),  minor(buf.st_dev));
```

```
        if (S_ISCHR(buf.st_mode) || S_ISBLK(buf.st_mode)) {
            printf(" (%s) rdev = %d/%d",
                    (S_ISCHR(buf.st_mode)) ? "character" : "block",
                    major(buf.st_rdev), minor(buf.st_rdev));
        }
        printf("\n");
    }

    exit(0);
}
```

Figure 4.25 Print `st_dev` and `st_rdev` values

Running this program on Linux gives us the following output:

```
$ ./a.out / /home/sar /dev/tty[01]
/: dev = 8/3
/home/sar: dev = 8/4
/dev/tty0: dev = 0/5 (character) rdev = 4/0
/dev/tty1: dev = 0/5 (character) rdev = 4/1
$ mount                          which directories are mounted on which devices?
/dev/sda3 on / type ext3 (rw,errors=remount-ro,commit=0)
/dev/sda4 on /home type ext3 (rw,commit=0)
$ ls -l /dev/tty[01] /dev/sda[34]
brw-rw----  1 root       8, 3 2011-07-01 11:08 /dev/sda3
brw-rw----  1 root       8, 4 2011-07-01 11:08 /dev/sda4
crw--w----  1 root       4, 0 2011-07-01 11:08 /dev/tty0
crw-------  1 root       4, 1 2011-07-01 11:08 /dev/tty1
```

The first two arguments to the program are directories (`/` and `/home/sar`), and the next two are the device names `/dev/tty[01]`. (We use the shell's regular expression language to shorten the amount of typing we need to do. The shell will expand the string `/dev/tty[01]` to `/dev/tty0 /dev/tty1`.)

We expect the devices to be character special files. The output from the program shows that the root directory has a different device number than does the `/home/sar` directory, which indicates that they are on different file systems. Running the `mount`(1) command verifies this.

We then use `ls` to look at the two disk devices reported by `mount` and the two terminal devices. The two disk devices are block special files, and the two terminal devices are character special files. (Normally, the only types of devices that are block special files are those that can contain random-access file systems—disk drives, floppy disk drives, and CD-ROMs, for example. Some older UNIX systems supported magnetic tapes for file systems, but this was never widely used.)

Note that the filenames and i-nodes for the two terminal devices (`st_dev`) are on device 0/5—the `devtmpfs` pseudo file system, which implements the `/dev`—but that their actual device numbers are 4/0 and 4/1. □

4.25 Summary of File Access Permission Bits

We've covered all the file access permission bits, some of which serve multiple purposes. Figure 4.26 summarizes these permission bits and their interpretation when applied to a regular file and a directory.

Constant	Description	Effect on regular file	Effect on directory
S_ISUID	set-user-ID	set effective user ID on execution	(not used)
S_ISGID	set-group-ID	if group-execute set, then set effective group ID on execution; otherwise, enable mandatory record locking (if supported)	set group ID of new files created in directory to group ID of directory
S_ISVTX	sticky bit	control caching of file contents (if supported)	restrict removal and renaming of files in directory
S_IRUSR	user-read	user permission to read file	user permission to read directory entries
S_IWUSR	user-write	user permission to write file	user permission to remove and create files in directory
S_IXUSR	user-execute	user permission to execute file	user permission to search for given pathname in directory
S_IRGRP	group-read	group permission to read file	group permission to read directory entries
S_IWGRP	group-write	group permission to write file	group permission to remove and create files in directory
S_IXGRP	group-execute	group permission to execute file	group permission to search for given pathname in directory
S_IROTH	other-read	other permission to read file	other permission to read directory entries
S_IWOTH	other-write	other permission to write file	other permission to remove and create files in directory
S_IXOTH	other-execute	other permission to execute file	other permission to search for given pathname in directory

Figure 4.26 Summary of file access permission bits

The final nine constants can also be grouped into threes, as follows:

```
S_IRWXU = S_IRUSR | S_IWUSR | S_IXUSR
S_IRWXG = S_IRGRP | S_IWGRP | S_IXGRP
S_IRWXO = S_IROTH | S_IWOTH | S_IXOTH
```

4.26 Summary

This chapter has centered on the `stat` function. We've gone through each member in the `stat` structure in detail. This, in turn, led us to examine all the attributes of UNIX files and directories. We've looked at how files and directories might be laid out in a file

system, and we've seen how to navigate the file system namespace. A thorough understanding of all the properties of files and directories and all the functions that operate on them is essential to UNIX programming.

Exercises

4.1 Modify the program in Figure 4.3 to use `stat` instead of `lstat`. What changes if one of the command-line arguments is a symbolic link?

4.2 What happens if the file mode creation mask is set to 777 (octal)? Verify the results using your shell's `umask` command.

4.3 Verify that turning off user-read permission for a file that you own denies your access to the file.

4.4 Run the program in Figure 4.9 *after* creating the files `foo` and `bar`. What happens?

4.5 In Section 4.12, we said that a file size of 0 is valid for a regular file. We also said that the `st_size` field is defined for directories and symbolic links. Should we ever see a file size of 0 for a directory or a symbolic link?

4.6 Write a utility like `cp`(1) that copies a file containing holes, without writing the bytes of 0 to the output file.

4.7 Note in the output from the `ls` command in Section 4.12 that the files `core` and `core.copy` have different access permissions. If the `umask` value didn't change between the creation of the two files, explain how the difference could have occurred.

4.8 When running the program in Figure 4.16, we check the available disk space with the `df`(1) command. Why didn't we use the `du`(1) command?

4.9 In Figure 4.20, we show the `unlink` function as modifying the changed-status time of the file itself. How can this happen?

4.10 In Section 4.22, how does the system's limit on the number of open files affect the `myftw` function?

4.11 In Section 4.22, our version of `ftw` never changes its directory. Modify this routine so that each time it encounters a directory, it uses the `chdir` function to change to that directory, allowing it to use the filename and not the pathname for each call to `lstat`. When all the entries in a directory have been processed, execute `chdir("..")`. Compare the time used by this version and the version in the text.

4.12 Each process also has a root directory that is used for resolution of absolute pathnames. This root directory can be changed with the `chroot` function. Look up the description for this function in your manuals. When might this function be useful?

4.13 How can you set only one of the two time values with the `utimes` function?

4.14 Some versions of the `finger`(1) command output "New mail received ..." and "unread since ..." where ... are the corresponding times and dates. How can the program determine these two times and dates?

4.15 Examine the archive formats used by the `cpio`(1) and `tar`(1) commands. (These descriptions are usually found in Section 5 of the *UNIX Programmer's Manual*.) How many of the three possible time values are saved for each file? When a file is restored, what value do you think the access time is set to, and why?

4.16 Does the UNIX System have a fundamental limitation on the depth of a directory tree? To find out, write a program that creates a directory and then changes to that directory, in a loop. Make certain that the length of the absolute pathname of the leaf of this directory is greater than your system's `PATH_MAX` limit. Can you call `getcwd` to fetch the directory's pathname? How do the standard UNIX System tools deal with this long pathname? Can you archive the directory using either `tar` or `cpio`?

4.17 In Section 3.16, we described the `/dev/fd` feature. For any user to be able to access these files, their permissions must be `rw-rw-rw-`. Some programs that create an output file delete the file first, in case it already exists, ignoring the return code:

```
unlink(path);
if ((fd = creat(path, FILE_MODE)) < 0)
    err_sys(...);
```

What happens if `path` is `/dev/fd/1`?

5

Standard I/O Library

5.1 Introduction

In this chapter, we describe the standard I/O library. This library is specified by the ISO C standard because it has been implemented on many operating systems other than the UNIX System. Additional interfaces are defined as extensions to the ISO C standard by the Single UNIX Specification.

The standard I/O library handles such details as buffer allocation and performing I/O in optimal-sized chunks, obviating our need to worry about using the correct block size (as in Section 3.9). This makes the library easy to use, but at the same time introduces another set of problems if we're not cognizant of what's going on.

> The standard I/O library was written by Dennis Ritchie around 1975. It was a major revision of the Portable I/O library written by Mike Lesk. Surprisingly little has changed in the standard I/O library after more than 35 years.

5.2 Streams and `FILE` Objects

In Chapter 3, all the I/O routines centered on file descriptors. When a file is opened, a file descriptor is returned, and that descriptor is then used for all subsequent I/O operations. With the standard I/O library, the discussion centers on *streams*. (Do not confuse the standard I/O term *stream* with the STREAMS I/O system that is part of System V and was standardized in the XSI STREAMS option in the Single UNIX Specification, but is now marked obsolescent in SUSv4.) When we open or create a file with the standard I/O library, we say that we have associated a stream with the file.

With the ASCII character set, a single character is represented by a single byte. With international character sets, a character can be represented by more than one byte.

Standard I/O file streams can be used with both single-byte and multibyte (“wide”) character sets. A stream’s orientation determines whether the characters that are read and written are single byte or multibyte. Initially, when a stream is created, it has no orientation. If a multibyte I/O function (see <wchar.h>) is used on a stream without orientation, the stream’s orientation is set to wide oriented. If a byte I/O function is used on a stream without orientation, the stream’s orientation is set to byte oriented. Only two functions can change the orientation once set. The freopen function (discussed shortly) will clear a stream’s orientation; the fwide function can be used to set a stream’s orientation.

```
#include <stdio.h>
#include <wchar.h>

int fwide(FILE *fp, int mode);
```

Returns: positive if stream is wide oriented,
negative if stream is byte oriented,
or 0 if stream has no orientation

The fwide function performs different tasks, depending on the value of the *mode* argument.

- If the *mode* argument is negative, fwide will try to make the specified stream byte oriented.
- If the *mode* argument is positive, fwide will try to make the specified stream wide oriented.
- If the *mode* argument is zero, fwide will not try to set the orientation, but will still return a value identifying the stream’s orientation.

Note that fwide will not change the orientation of a stream that is already oriented. Also note that there is no error return. Consider what would happen if the stream is invalid. The only recourse we have is to clear errno before calling fwide and check the value of errno when we return. Throughout the rest of this book, we will deal only with byte-oriented streams.

When we open a stream, the standard I/O function fopen (Section 5.5) returns a pointer to a FILE object. This object is normally a structure that contains all the information required by the standard I/O library to manage the stream: the file descriptor used for actual I/O, a pointer to a buffer for the stream, the size of the buffer, a count of the number of characters currently in the buffer, an error flag, and the like.

Application software should never need to examine a FILE object. To reference the stream, we pass its FILE pointer as an argument to each standard I/O function. Throughout this text, we’ll refer to a pointer to a FILE object, the type FILE *, as a *file pointer*.

Throughout this chapter, we describe the standard I/O library in the context of a UNIX system. As we mentioned, this library has been ported to a wide variety of other operating systems. To provide some insight about how this library can be implemented, we will talk about its typical implementation on a UNIX system.

5.3 Standard Input, Standard Output, and Standard Error

Three streams are predefined and automatically available to a process: standard input, standard output, and standard error. These streams refer to the same files as the file descriptors `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`, respectively, which we mentioned in Section 3.2.

These three standard I/O streams are referenced through the predefined file pointers `stdin`, `stdout`, and `stderr`. The file pointers are defined in the `<stdio.h>` header.

5.4 Buffering

The goal of the buffering provided by the standard I/O library is to use the minimum number of `read` and `write` calls. (Recall Figure 3.6, which showed the amount of CPU time required to perform I/O using various buffer sizes.) Also, this library tries to do its buffering automatically for each I/O stream, obviating the need for the application to worry about it. Unfortunately, the single aspect of the standard I/O library that generates the most confusion is its buffering.

Three types of buffering are provided:

1. Fully buffered. In this case, actual I/O takes place when the standard I/O buffer is filled. Files residing on disk are normally fully buffered by the standard I/O library. The buffer used is usually obtained by one of the standard I/O functions calling `malloc` (Section 7.8) the first time I/O is performed on a stream.

   The term *flush* describes the writing of a standard I/O buffer. A buffer can be flushed automatically by the standard I/O routines, such as when a buffer fills, or we can call the function `fflush` to flush a stream. Unfortunately, in the UNIX environment, *flush* means two different things. In terms of the standard I/O library, it means writing out the contents of a buffer, which may be partially filled. In terms of the terminal driver, such as the `tcflush` function in Chapter 18, it means to discard the data that's already stored in a buffer.

2. Line buffered. In this case, the standard I/O library performs I/O when a newline character is encountered on input or output. This allows us to output a single character at a time (with the standard I/O `fputc` function), knowing that actual I/O will take place only when we finish writing each line. Line buffering is typically used on a stream when it refers to a terminal—standard input and standard output, for example.

   Line buffering comes with two caveats. First, the size of the buffer that the standard I/O library uses to collect each line is fixed, so I/O might take place if we fill this buffer before writing a newline. Second, whenever input is requested through the standard I/O library from either (a) an unbuffered stream

or (b) a line-buffered stream (that requires data to be requested from the kernel), *all* line-buffered output streams are flushed. The reason for the qualifier on (b) is that the requested data may already be in the buffer, which doesn't require data to be read from the kernel. Obviously, any input from an unbuffered stream, item (a), requires data to be obtained from the kernel.

3. Unbuffered. The standard I/O library does not buffer the characters. If we write 15 characters with the standard I/O `fputs` function, for example, we expect these 15 characters to be output as soon as possible, probably with the `write` function from Section 3.8.

   The standard error stream, for example, is normally unbuffered so that any error messages are displayed as quickly as possible, regardless of whether they contain a newline.

ISO C requires the following buffering characteristics:

- Standard input and standard output are fully buffered, if and only if they do not refer to an interactive device.
- Standard error is never fully buffered.

This, however, doesn't tell us whether standard input and standard output are unbuffered or line buffered if they refer to an interactive device and whether standard error should be unbuffered or line buffered. Most implementations default to the following types of buffering:

- Standard error is always unbuffered.
- All other streams are line buffered if they refer to a terminal device; otherwise, they are fully buffered.

> The four platforms discussed in this book follow these conventions for standard I/O buffering: standard error is unbuffered, streams open to terminal devices are line buffered, and all other streams are fully buffered.

We explore standard I/O buffering in more detail in Section 5.12 and Figure 5.11.

If we don't like these defaults for any given stream, we can change the buffering by calling either the `setbuf` or `setvbuf` function.

```
#include <stdio.h>

void setbuf(FILE *restrict fp, char *restrict buf);

int setvbuf(FILE *restrict fp, char *restrict buf, int mode,
            size_t size);
```

Returns: 0 if OK, nonzero on error

These functions must be called *after* the stream has been opened (obviously, since each requires a valid file pointer as its first argument) but *before* any other operation is performed on the stream.

With `setbuf`, we can turn buffering on or off. To enable buffering, *buf* must point to a buffer of length `BUFSIZ`, a constant defined in `<stdio.h>`. Normally, the stream is then fully buffered, but some systems may set line buffering if the stream is associated with a terminal device. To disable buffering, we set *buf* to `NULL`.

With `setvbuf`, we specify exactly which type of buffering we want. This is done with the *mode* argument:

`_IOFBF`	fully buffered
`_IOLBF`	line buffered
`_IONBF`	unbuffered

If we specify an unbuffered stream, the *buf* and *size* arguments are ignored. If we specify fully buffered or line buffered, *buf* and *size* can optionally specify a buffer and its size. If the stream is buffered and *buf* is `NULL`, the standard I/O library will automatically allocate its own buffer of the appropriate size for the stream. By appropriate size, we mean the value specified by the constant `BUFSIZ`.

> Some C library implementations use the value from the `st_blksize` member of the `stat` structure (see Section 4.2) to determine the optimal standard I/O buffer size. As we will see later in this chapter, the GNU C library uses this method.

Figure 5.1 summarizes the actions of these two functions and their various options.

Function	*mode*	*buf*	Buffer and length	Type of buffering
`setbuf`		non-null	user *buf* of length `BUFSIZ`	fully buffered or line buffered
		`NULL`	(no buffer)	unbuffered
`setvbuf`	`_IOFBF`	non-null	user *buf* of length *size*	fully buffered
		`NULL`	system buffer of appropriate length	
	`_IOLBF`	non-null	user *buf* of length *size*	line buffered
		`NULL`	system buffer of appropriate length	
	`_IONBF`	(ignored)	(no buffer)	unbuffered

Figure 5.1 Summary of the `setbuf` and `setvbuf` functions

Be aware that if we allocate a standard I/O buffer as an automatic variable within a function, we have to close the stream before returning from the function. (We'll discuss this point further in Section 7.8.) Also, some implementations use part of the buffer for internal bookkeeping, so the actual number of bytes of data that can be stored in the buffer can be less than *size*. In general, we should let the system choose the buffer size and automatically allocate the buffer. When we do this, the standard I/O library automatically releases the buffer when we close the stream.

At any time, we can force a stream to be flushed.

```
#include <stdio.h>

int fflush(FILE *fp);
```

Returns: 0 if OK, `EOF` on error

The `fflush` function causes any unwritten data for the stream to be passed to the kernel. As a special case, if *fp* is `NULL`, `fflush` causes all output streams to be flushed.

5.5 Opening a Stream

The fopen, freopen, and fdopen functions open a standard I/O stream.

```
#include <stdio.h>

FILE *fopen(const char *restrict pathname, const char *restrict type);

FILE *freopen(const char *restrict pathname, const char *restrict type,
              FILE *restrict fp);

FILE *fdopen(int fd, const char *type);
```

All three return: file pointer if OK, NULL on error

The differences in these three functions are as follows:

1. The fopen function opens a specified file.
2. The freopen function opens a specified file on a specified stream, closing the stream first if it is already open. If the stream previously had an orientation, freopen clears it. This function is typically used to open a specified file as one of the predefined streams: standard input, standard output, or standard error.
3. The fdopen function takes an existing file descriptor, which we could obtain from the open, dup, dup2, fcntl, pipe, socket, socketpair, or accept functions, and associates a standard I/O stream with the descriptor. This function is often used with descriptors that are returned by the functions that create pipes and network communication channels. Because these special types of files cannot be opened with the standard I/O fopen function, we have to call the device-specific function to obtain a file descriptor, and then associate this descriptor with a standard I/O stream using fdopen.

Both fopen and freopen are part of ISO C; fdopen is part of POSIX.1, since ISO C doesn't deal with file descriptors.

type	Description	open(2) Flags
r or rb	open for reading	O_RDONLY
w or wb	truncate to 0 length or create for writing	O_WRONLY|O_CREAT|O_TRUNC
a or ab	append; open for writing at end of file, or create for writing	O_WRONLY|O_CREAT|O_APPEND
r+ or r+b or rb+	open for reading and writing	O_RDWR
w+ or w+b or wb+	truncate to 0 length or create for reading and writing	O_RDWR|O_CREAT|O_TRUNC
a+ or a+b or ab+	open or create for reading and writing at end of file	O_RDWR|O_CREAT|O_APPEND

Figure 5.2 The *type* argument for opening a standard I/O stream

ISO C specifies 15 values for the *type* argument, shown in Figure 5.3. Using the character b as part of the *type* allows the standard I/O system to differentiate between a

text file and a binary file. Since the UNIX kernel doesn't differentiate between these types of files, specifying the character `b` as part of the *type* has no effect.

With `fdopen`, the meanings of the *type* argument differ slightly. The descriptor has already been opened, so opening for writing does not truncate the file. (If the descriptor was created by the `open` function, for example, and the file already existed, the `O_TRUNC` flag would control whether the file was truncated. The `fdopen` function cannot simply truncate any file it opens for writing.) Also, the standard I/O append mode cannot create the file (since the file has to exist if a descriptor refers to it).

When a file is opened with a type of append, each write will take place at the then current end of file. If multiple processes open the same file with the standard I/O append mode, the data from each process will be correctly written to the file.

> Versions of `fopen` from Berkeley before 4.4BSD and the simple version shown on page 177 of Kernighan and Ritchie [1988] do not handle the append mode correctly. These versions do an `lseek` to the end of file when the stream is opened. To correctly support the append mode when multiple processes are involved, the file must be opened with the `O_APPEND` flag, which we discussed in Section 3.3. Doing an `lseek` before each write won't work either, as we discussed in Section 3.11.

When a file is opened for reading and writing (the plus sign in the *type*), two restrictions apply.

- Output cannot be directly followed by input without an intervening `fflush`, `fseek`, `fsetpos`, or `rewind`.
- Input cannot be directly followed by output without an intervening `fseek`, `fsetpos`, or `rewind`, or an input operation that encounters an end of file.

We can summarize the six ways to open a stream from Figure 5.2 in Figure 5.3.

Restriction	`r`	`w`	`a`	`r+`	`w+`	`a+`
file must already exist	•			•		
previous contents of file discarded		•			•	
stream can be read	•			•	•	•
stream can be written		•	•	•	•	•
stream can be written only at end			•			•

Figure 5.3 Six ways to open a standard I/O stream

Note that if a new file is created by specifying a *type* of either `w` or `a`, we are not able to specify the file's access permission bits, as we were able to do with the `open` function and the `creat` function in Chapter 3. POSIX.1 requires implementations to create the file with the following permissions bit set:

```
S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH
```

Recall from Section 4.8, however, that we can restrict these permissions by adjusting our `umask` value.

By default, the stream that is opened is fully buffered, unless it refers to a terminal device, in which case it is line buffered. Once the stream is opened, but before we do

any other operation on the stream, we can change the buffering if we want to, with the `setbuf` or `setvbuf` functions from the previous section.

An open stream is closed by calling `fclose`.

```
#include <stdio.h>
int fclose(FILE *fp);
                                                    Returns: 0 if OK, EOF on error
```

Any buffered output data is flushed before the file is closed. Any input data that may be buffered is discarded. If the standard I/O library had automatically allocated a buffer for the stream, that buffer is released.

When a process terminates normally, either by calling the `exit` function directly or by returning from the `main` function, all standard I/O streams with unwritten buffered data are flushed and all open standard I/O streams are closed.

5.6 Reading and Writing a Stream

Once we open a stream, we can choose from among three types of unformatted I/O:

1. Character-at-a-time I/O. We can read or write one character at a time, with the standard I/O functions handling all the buffering, if the stream is buffered.
2. Line-at-a-time I/O. If we want to read or write a line at a time, we use `fgets` and `fputs`. Each line is terminated with a newline character, and we have to specify the maximum line length that we can handle when we call `fgets`. We describe these two functions in Section 5.7.
3. Direct I/O. This type of I/O is supported by the `fread` and `fwrite` functions. For each I/O operation, we read or write some number of objects, where each object is of a specified size. These two functions are often used for binary files where we read or write a structure with each operation. We describe these two functions in Section 5.9.

   The term *direct I/O*, from the ISO C standard, is known by many names: binary I/O, object-at-a-time I/O, record-oriented I/O, or structure-oriented I/O. Don't confuse this feature with the `O_DIRECT` `open` flag supported by FreeBSD and Linux—they are unrelated.

(We describe the formatted I/O functions, such as `printf` and `scanf`, in Section 5.11.)

Input Functions

Three functions allow us to read one character at a time.

```
#include <stdio.h>
int getc(FILE *fp);
int fgetc(FILE *fp);
int getchar(void);
                    All three return: next character if OK, EOF on end of file or error
```

The function `getchar` is defined to be equivalent to `getc(stdin)`. The difference between `getc` and `fgetc` is that `getc` can be implemented as a macro, whereas `fgetc` cannot be implemented as a macro. This means three things.

1. The argument to `getc` should not be an expression with side effects, because it could be evaluated more than once.
2. Since `fgetc` is guaranteed to be a function, we can take its address. This allows us to pass the address of `fgetc` as an argument to another function.
3. Calls to `fgetc` probably take longer than calls to `getc`, as it usually takes more time to call a function.

These three functions return the next character as an `unsigned char` converted to an `int`. The reason for specifying unsigned is so that the high-order bit, if set, doesn't cause the return value to be negative. The reason for requiring an integer return value is so that all possible character values can be returned, along with an indication that either an error occurred or the end of file has been encountered. The constant `EOF` in `<stdio.h>` is required to be a negative value. Its value is often –1. This representation also means that we cannot store the return value from these three functions in a character variable and later compare this value with the constant `EOF`.

Note that these functions return the same value whether an error occurs or the end of file is reached. To distinguish between the two, we must call either `ferror` or `feof`.

```
#include <stdio.h>

int ferror(FILE *fp);

int feof(FILE *fp);
```

Both return: nonzero (true) if condition is true, 0 (false) otherwise

```
void clearerr(FILE *fp);
```

In most implementations, two flags are maintained for each stream in the `FILE` object:

- An error flag
- An end-of-file flag

Both flags are cleared by calling `clearerr`.

After reading from a stream, we can push back characters by calling `ungetc`.

```
#include <stdio.h>

int ungetc(int c, FILE *fp);
```

Returns: *c* if OK, `EOF` on error

The characters that are pushed back are returned by subsequent reads on the stream in reverse order of their pushing. Be aware, however, that although ISO C allows an implementation to support any amount of pushback, an implementation is required to provide only a single character of pushback. We should not count on more than a single character.

The character that we push back does not have to be the same character that was read. We are not able to push back `EOF`. When we reach the end of file, however, we can push back a character. The next read will return that character, and the read after that will return `EOF`. This works because a successful call to `ungetc` clears the end-of-file indication for the stream.

Pushback is often used when we're reading an input stream and breaking the input into words or tokens of some form. Sometimes we need to peek at the next character to determine how to handle the current character. It's then easy to push back the character that we peeked at, for the next call to `getc` to return. If the standard I/O library didn't provide this pushback capability, we would have to store the character in a variable of our own, along with a flag telling us to use this character instead of calling `getc` the next time we need a character.

When we push characters back with `ungetc`, they are not written back to the underlying file or device. Instead, they are kept incore in the standard I/O library's buffer for the stream.

Output Functions

Output functions are available that correspond to each of the input functions we've already described.

```
#include <stdio.h>

int putc(int c, FILE *fp);

int fputc(int c, FILE *fp);

int putchar(int c);
```

All three return: *c* if OK, `EOF` on error

As with the input functions, `putchar(c)` is equivalent to `putc(c, stdout)`, and `putc` can be implemented as a macro, whereas `fputc` cannot be implemented as a macro.

5.7 Line-at-a-Time I/O

Line-at-a-time input is provided by the two functions, `fgets` and `gets`.

```
#include <stdio.h>

char *fgets(char *restrict buf, int n, FILE *restrict fp);

char *gets(char *buf);
```

Both return: `buf` if OK, `NULL` on end of file or error

Both specify the address of the buffer to read the line into. The `gets` function reads from standard input, whereas `fgets` reads from the specified stream.

With `fgets`, we have to specify the size of the buffer, *n*. This function reads up through and including the next newline, but no more than $n-1$ characters, into the

buffer. The buffer is terminated with a null byte. If the line, including the terminating newline, is longer than $n-1$, only a partial line is returned, but the buffer is always null terminated. Another call to `fgets` will read what follows on the line.

The `gets` function should never be used. The problem is that it doesn't allow the caller to specify the buffer size. This allows the buffer to overflow if the line is longer than the buffer, writing over whatever happens to follow the buffer in memory. For a description of how this flaw was used as part of the Internet worm of 1988, see the June 1989 issue (vol. 32, no. 6) of *Communications of the ACM*. An additional difference with `gets` is that it doesn't store the newline in the buffer, as `fgets` does.

> This difference in newline handling between the two functions goes way back in the evolution of the UNIX System. Even the Version 7 manual (1979) states "`gets` deletes a newline, `fgets` keeps it, all in the name of backward compatibility."

Even though ISO C requires an implementation to provide `gets`, you should use `fgets` instead. In fact, `gets` is marked as an obsolescent interface in SUSv4 and has been omitted from the latest version of the ISO C standard (ISO/IEC 9899:2011).

Line-at-a-time output is provided by `fputs` and `puts`.

```
#include <stdio.h>

int fputs(const char *restrict str, FILE *restrict fp);

int puts(const char *str);
```

Both return: non-negative value if OK, `EOF` on error

The function `fputs` writes the null-terminated string to the specified stream. The null byte at the end is not written. Note that this need not be line-at-a-time output, since the string need not contain a newline as the last non-null character. Usually, this is the case—the last non-null character is a newline—but it's not required.

The `puts` function writes the null-terminated string to the standard output, without writing the null byte. But `puts` then writes a newline character to the standard output.

The `puts` function is not unsafe, like its counterpart `gets`. Nevertheless, we'll avoid using it, to prevent having to remember whether it appends a newline. If we always use `fgets` and `fputs`, we know that we always have to deal with the newline character at the end of each line.

5.8 Standard I/O Efficiency

Using the functions from the previous section, we can get an idea of the efficiency of the standard I/O system. The program in Figure 5.4 is like the one in Figure 3.5: it simply copies standard input to standard output, using `getc` and `putc`. These two routines can be implemented as macros.

```
#include "apue.h"

int
main(void)
{
    int     c;

    while ((c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

Figure 5.4 Copy standard input to standard output using `getc` and `putc`

We can make another version of this program that uses `fgetc` and `fputc`, which should be functions, not macros. (We don't show this trivial change to the source code.)

Finally, we have a version that reads and writes lines, shown in Figure 5.5.

```
#include "apue.h"

int
main(void)
{
    char    buf[MAXLINE];

    while (fgets(buf, MAXLINE, stdin) != NULL)
        if (fputs(buf, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

Figure 5.5 Copy standard input to standard output using `fgets` and `fputs`

Note that we do not close the standard I/O streams explicitly in either Figure 5.4 or Figure 5.5. Instead, we know that the `exit` function will flush any unwritten data and then close all open streams. (We'll discuss this in Section 8.5.) It is interesting to compare the timing of these three programs with the timing data from Figure 3.6. We show this data when operating on the same file (98.5 MB with 3 million lines) in Figure 5.6.

For each of the three standard I/O versions, the user CPU time is larger than the best `read` version from Figure 3.6, because the character-at-a-time standard I/O versions have a loop that is executed 100 million times, and the loop in the line-at-a-

Function	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Bytes of program text
best time from Figure 3.6	0.05	0.29	3.18	
`fgets, fputs`	2.27	0.30	3.49	143
`getc, putc`	8.45	0.29	10.33	114
`fgetc, fputc`	8.16	0.40	10.18	114
single byte time from Figure 3.6	134.61	249.94	394.95	

Figure 5.6 Timing results using standard I/O routines

time version is executed 3,144,984 times. In the `read` version, its loop is executed only 25,224 times (for a buffer size of 4,096). This difference in clock times stems from the difference in user times and the difference in the times spent waiting for I/O to complete, as the system times are comparable.

The system CPU time is about the same as before, because roughly the same number of kernel requests are being made. One advantage of using the standard I/O routines is that we don't have to worry about buffering or choosing the optimal I/O size. We do have to determine the maximum line size for the version that uses `fgets`, but that's easier than trying to choose the optimal I/O size.

The final column in Figure 5.6 is the number of bytes of text space—the machine instructions generated by the C compiler—for each of the `main` functions. We can see that the version using `getc` and `putc` takes the same amount of space as the one using the `fgetc` and `fputc` functions. Usually, `getc` and `putc` are implemented as macros, but in the GNU C library implementation the macro simply expands to a function call.

The version using line-at-a-time I/O is almost twice as fast as the version using character-at-a-time I/O. If the `fgets` and `fputs` functions are implemented using `getc` and `putc` (see Section 7.7 of Kernighan and Ritchie [1988], for example), then we would expect the timing to be similar to the `getc` version. Actually, we might expect the line-at-a-time version to take longer, since we would be adding the overhead of 200 million extra function calls to the existing 6 million ones. What is happening with this example is that the line-at-a-time functions are implemented using `memccpy`(3). Often, the `memccpy` function is implemented in assembly language instead of C, for efficiency.

The last point of interest with these timing numbers is that the `fgetc` version is so much faster than the `BUFFSIZE=1` version from Figure 3.6. Both involve the same number of function calls—about 200 million—yet the `fgetc` version is more than 16 times faster in terms of user CPU time and almost 39 times faster in terms of clock time. The difference is that the version using `read` executes 200 million function calls, which in turn execute 200 million system calls. With the `fgetc` version, we still execute 200 million function calls, but this translates into only 25,224 system calls. System calls are usually much more expensive than ordinary function calls.

As a disclaimer, you should be aware that these timing results are valid only on the single system they were run on. The results depend on many implementation features that aren't the same on every UNIX system. Nevertheless, having a set of numbers such as these, and explaining why the various versions differ, helps us understand the system better. From this section and Section 3.9, we've learned that the standard I/O

library is not much slower than calling the `read` and `write` functions directly. For most nontrivial applications, the largest amount of user CPU time is taken by the application, not by the standard I/O routines.

5.9 Binary I/O

The functions from Section 5.6 operated with one character at a time, and the functions from Section 5.7 operated with one line at a time. If we're doing binary I/O, we often would like to read or write an entire structure at a time. To do this using `getc` or `putc`, we have to loop through the entire structure, one byte at a time, reading or writing each byte. We can't use the line-at-a-time functions, since `fputs` stops writing when it hits a null byte, and there might be null bytes within the structure. Similarly, `fgets` won't work correctly on input if any of the data bytes are nulls or newlines. Therefore, the following two functions are provided for binary I/O.

```
#include <stdio.h>

size_t fread(void *restrict ptr, size_t size, size_t nobj,
             FILE *restrict fp);

size_t fwrite(const void *restrict ptr, size_t size, size_t nobj,
              FILE *restrict fp);
```

Both return: number of objects read or written

These functions have two common uses:

1. Read or write a binary array. For example, to write elements 2 through 5 of a floating-point array, we could write

```
float   data[10];

if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
    err_sys("fwrite error");
```

   Here, we specify *size* as the size of each element of the array and *nobj* as the number of elements.

2. Read or write a structure. For example, we could write

```
struct {
  short  count;
  long   total;
  char   name[NAMESIZE];
} item;

if (fwrite(&item, sizeof(item), 1, fp) != 1)
    err_sys("fwrite error");
```

   Here, we specify *size* as the size of structure and *nobj* as 1 (the number of objects to write).

The obvious generalization of these two cases is to read or write an array of structures. To do this, *size* would be the `sizeof` the structure, and *nobj* would be the number of elements in the array.

Both `fread` and `fwrite` return the number of objects read or written. For the read case, this number can be less than *nobj* if an error occurs or if the end of file is encountered. In this situation, `ferror` or `feof` must be called. For the write case, if the return value is less than the requested *nobj*, an error has occurred.

A fundamental problem with binary I/O is that it can be used to read only data that has been written on the same system. This was OK many years ago, when all the UNIX systems were PDP-11s, but the norm today is to have heterogeneous systems connected together with networks. It is common to want to write data on one system and process it on another. These two functions won't work, for two reasons.

1. The offset of a member within a structure can differ between compilers and systems because of different alignment requirements. Indeed, some compilers have an option allowing structures to be packed tightly, to save space with a possible runtime performance penalty, or aligned accurately, to optimize runtime access of each member. This means that even on a single system, the binary layout of a structure can differ, depending on compiler options.

2. The binary formats used to store multibyte integers and floating-point values differ among machine architectures.

We'll touch on some of these issues when we discuss sockets in Chapter 16. The real solution for exchanging binary data among different systems is to use an agreed-upon canonical format. Refer to Section 8.2 of Rago [1993] or Section 5.18 of Stevens, Fenner, & Rudoff [2004] for a description of some techniques various network protocols use to exchange binary data.

We'll return to the `fread` function in Section 8.14 when we use it to read a binary structure, the UNIX process accounting records.

5.10 Positioning a Stream

There are three ways to position a standard I/O stream:

1. The two functions `ftell` and `fseek`. They have been around since Version 7, but they assume that a file's position can be stored in a long integer.

2. The two functions `ftello` and `fseeko`. They were introduced in the Single UNIX Specification to allow for file offsets that might not fit in a long integer. They replace the long integer with the `off_t` data type.

3. The two functions `fgetpos` and `fsetpos`. They were introduced by ISO C. They use an abstract data type, `fpos_t`, that records a file's position. This data type can be made as big as necessary to record a file's position.

When porting applications to non-UNIX systems, use `fgetpos` and `fsetpos`.

```
#include <stdio.h>

long ftell(FILE *fp);
                                    Returns: current file position indicator if OK, –1L on error

int fseek(FILE *fp, long offset, int whence);
                                                    Returns: 0 if OK, –1 on error

void rewind(FILE *fp);
```

For a binary file, a file's position indicator is measured in bytes from the beginning of the file. The value returned by `ftell` for a binary file is this byte position. To position a binary file using `fseek`, we must specify a byte *offset* and indicate how that offset is interpreted. The values for *whence* are the same as for the `lseek` function from Section 3.6: `SEEK_SET` means from the beginning of the file, `SEEK_CUR` means from the current file position, and `SEEK_END` means from the end of file. ISO C doesn't require an implementation to support the `SEEK_END` specification for a binary file, as some systems require a binary file to be padded at the end with zeros to make the file size a multiple of some magic number. Under the UNIX System, however, `SEEK_END` is supported for binary files.

For text files, the file's current position may not be measurable as a simple byte offset. Again, this is mainly under non-UNIX systems that might store text files in a different format. To position a text file, *whence* has to be `SEEK_SET`, and only two values for *offset* are allowed: 0—meaning rewind the file to its beginning—or a value that was returned by `ftell` for that file. A stream can also be set to the beginning of the file with the `rewind` function.

The `ftello` function is the same as `ftell`, and the `fseeko` function is the same as `fseek`, except that the type of the offset is `off_t` instead of `long`.

```
#include <stdio.h>

off_t ftello(FILE *fp);
                        Returns: current file position indicator if OK, (off_t)–1 on error

int fseeko(FILE *fp, off_t offset, int whence);
                                                    Returns: 0 if OK, –1 on error
```

Recall the discussion of the `off_t` data type in Section 3.6. Implementations can define the `off_t` type to be larger than 32 bits.

As we mentioned earlier, the `fgetpos` and `fsetpos` functions were introduced by the ISO C standard.

```
#include <stdio.h>

int fgetpos(FILE *restrict fp, fpos_t *restrict pos);

int fsetpos(FILE *fp, const fpos_t *pos);
                                            Both return: 0 if OK, nonzero on error
```

The fgetpos function stores the current value of the file's position indicator in the object pointed to by *pos*. This value can be used in a later call to fsetpos to reposition the stream to that location.

5.11 Formatted I/O

Formatted Output

Formatted output is handled by the five printf functions.

```
#include <stdio.h>

int printf(const char *restrict format, ...);

int fprintf(FILE *restrict fp, const char *restrict format, ...);

int dprintf(int fd, const char *restrict format, ...);
```

All three return: number of characters output if OK, negative value if output error

```
int sprintf(char *restrict buf, const char *restrict format, ...);
```

Returns: number of characters stored in array if OK, negative value if encoding error

```
int snprintf(char *restrict buf, size_t n,
             const char *restrict format, ...);
```

Returns: number of characters that would have been stored in array if buffer was large enough, negative value if encoding error

The printf function writes to the standard output, fprintf writes to the specified stream, dprintf writes to the specified file descriptor, and sprintf places the formatted characters in the array *buf*. The sprintf function automatically appends a null byte at the end of the array, but this null byte is not included in the return value.

Note that it's possible for sprintf to overflow the buffer pointed to by *buf*. The caller is responsible for ensuring that the buffer is large enough. Because buffer overflows can lead to program instability and even security violations, snprintf was introduced. With it, the size of the buffer is an explicit parameter; any characters that would have been written past the end of the buffer are discarded instead. The snprintf function returns the number of characters that would have been written to the buffer had it been big enough. As with sprintf, the return value doesn't include the terminating null byte. If snprintf returns a positive value less than the buffer size *n*, then the output was not truncated. If an encoding error occurs, snprintf returns a negative value.

Although dprintf doesn't deal with a file pointer, we include it with the rest of the related functions that handle formatted output. Note that using dprintf removes the need to call fdopen to convert a file descriptor into a file pointer for use with fprintf.

The format specification controls how the remainder of the arguments will be encoded and ultimately displayed. Each argument is encoded according to a conversion specification that starts with a percent sign (%). Except for the conversion

specifications, other characters in the format are copied unmodified. A conversion specification has four optional components, shown in square brackets below:

```
%[flags][fldwidth][precision][lenmodifier]convtype
```

The flags are summarized in Figure 5.7.

Flag	Description
'	(apostrophe) format integer with thousands grouping characters
-	left-justify the output in the field
+	always display sign of a signed conversion
(space)	prefix by a space if no sign is generated
#	convert using alternative form (include 0x prefix for hexadecimal format, for example)
0	prefix with leading zeros instead of padding with spaces

Figure 5.7 The flags component of a conversion specification

The `fldwidth` component specifies a minimum field width for the conversion. If the conversion results in fewer characters, it is padded with spaces. The field width is a non-negative decimal integer or an asterisk.

The `precision` component specifies the minimum number of digits to appear for integer conversions, the minimum number of digits to appear to the right of the decimal point for floating-point conversions, or the maximum number of bytes for string conversions. The precision is a period (.) followed by a optional non-negative decimal integer or an asterisk.

Either the field width or precision (or both) can be an asterisk. In this case, an integer argument specifies the value to be used. The argument appears directly before the argument to be converted.

The `lenmodifier` component specifies the size of the argument. Possible values are summarized in Figure 5.8.

Length modifier	Description
hh	signed or unsigned `char`
h	signed or unsigned `short`
l	signed or unsigned `long` or wide character
ll	signed or unsigned `long long`
j	`intmax_t` or `uintmax_t`
z	`size_t`
t	`ptrdiff_t`
L	`long double`

Figure 5.8 The length modifier component of a conversion specification

The `convtype` component is not optional. It controls how the argument is interpreted. The various conversion types are summarized in Figure 5.9.

With the normal conversion specification, conversions are applied to the arguments in the order they appear after the *format* argument. An alternative conversion specification syntax allows the arguments to be named explicitly with the sequence *%n$*

Conversion type	Description
d,i	signed decimal
o	unsigned octal
u	unsigned decimal
x,X	unsigned hexadecimal
f,F	double floating-point number
e,E	double floating-point number in exponential format
g,G	interpreted as f, F, e, or E, depending on value converted
a,A	double floating-point number in hexadecimal exponential format
c	character (with l length modifier, wide character)
s	string (with l length modifier, wide character string)
p	pointer to a void
n	pointer to a signed integer into which is written the number of characters written so far
%	a % character
C	wide character (XSI option, equivalent to lc)
S	wide character string (XSI option, equivalent to ls)

Figure 5.9 The conversion type component of a conversion specification

representing the *n*th argument. Note, however, that the two syntaxes can't be mixed in the same format specification. With the alternative syntax, arguments are numbered starting at one. If either the field width or precision is to be supplied by an argument, the asterisk syntax is modified to **m$*, where *m* specifies the position of the argument supplying the value.

The following five variants of the `printf` family are similar to the previous five, but the variable argument list (`...`) is replaced with *arg*.

```
#include <stdarg.h>
#include <stdio.h>

int vprintf(const char *restrict format, va_list arg);

int vfprintf(FILE *restrict fp, const char *restrict format,
             va_list arg);

int vdprintf(int fd, const char *restrict format, va_list arg);
```

All three return: number of characters output if OK, negative value if output error

```
int vsprintf(char *restrict buf, const char *restrict format,
             va_list arg);
```

Returns: number of characters stored in array if OK, negative value if encoding error

```
int vsnprintf(char *restrict buf, size_t n,
              const char *restrict format, va_list arg);
```

Returns: number of characters that would have been stored in array if buffer was large enough, negative value if encoding error

We use the `vsnprintf` function in the error routines in Appendix B.

Refer to Section 7.3 of Kernighan and Ritchie [1988] for additional details on handling variable-length argument lists with ISO Standard C. Be aware that the variable-length argument list routines provided with ISO C—the `<stdarg.h>` header and its associated routines—differ from the `<varargs.h>` routines that were provided with older UNIX systems.

Formatted Input

Formatted input is handled by the three `scanf` functions.

```
#include <stdio.h>

int scanf(const char *restrict format, ...);

int fscanf(FILE *restrict fp, const char *restrict format, ...);

int sscanf(const char *restrict buf, const char *restrict format, ...);
```

All three return: number of input items assigned, `EOF` if input error or end of file before any conversion

The `scanf` family is used to parse an input string and convert character sequences into variables of specified types. The arguments following the format contain the addresses of the variables to initialize with the results of the conversions.

The format specification controls how the arguments are converted for assignment. The percent sign (%) indicates the beginning of a conversion specification. Except for the conversion specifications and white space, other characters in the format have to match the input. If a character doesn't match, processing stops, leaving the remainder of the input unread.

There are three optional components to a conversion specification, shown in square brackets below:

```
%[*][fldwidth][m][lenmodifier]convtype
```

The optional leading asterisk is used to suppress conversion. Input is converted as specified by the rest of the conversion specification, but the result is not stored in an argument.

The `fldwidth` component specifies the maximum field width in characters. The `lenmodifier` component specifies the size of the argument to be initialized with the result of the conversion. The same length modifiers supported by the `printf` family of functions are supported by the `scanf` family of functions (see Figure 5.8 for a list of the length modifiers).

The `convtype` field is similar to the conversion type field used by the `printf` family, but there are some differences. One difference is that results that are stored in unsigned types can optionally be signed on input. For example, –1 will scan as 4294967295 into an unsigned integer. Figure 5.10 summarizes the conversion types supported by the `scanf` family of functions.

The optional `m` character between the field width and the length modifier is called the *assignment-allocation character*. It can be used with the `%c`, `%s`, and `%[` conversion

Conversion type	Description
d	signed decimal, base 10
i	signed decimal, base determined by format of input
o	unsigned octal (input optionally signed)
u	unsigned decimal, base 10 (input optionally signed)
x,X	unsigned hexadecimal (input optionally signed)
a,A,e,E,f,F,g,G	floating-point number
c	character (with l length modifier, wide character)
s	string (with l length modifier, wide character string)
[	matches a sequence of listed characters, ending with]
[^	matches all characters except the ones listed, ending with]
p	pointer to a void
n	pointer to a signed integer into which is written the number of characters read so far
%	a % character
C	wide character (XSI option, equivalent to lc)
S	wide character string (XSI option, equivalent to ls)

Figure 5.10 The conversion type component of a conversion specification

specifiers to force a memory buffer to be allocated to hold the converted string. In this case, the corresponding argument should be the address of a pointer to which the address of the allocated buffer will be copied. If the call succeeds, the caller is responsible for freeing the buffer by calling the `free` function when the buffer is no longer needed.

The `scanf` family of functions also supports the alternative conversion specification syntax allowing the arguments to be named explicitly: the sequence *%n$* represents the *n*th argument. With the `printf` family of functions, the same numbered argument can be referenced in the format string more than once. In this case, however, the Single UNIX Specification states that the behavior is undefined with the `scanf` family of functions.

Like the `printf` family, the `scanf` family supports functions that use variable argument lists as specified by `<stdarg.h>`.

```
#include <stdarg.h>
#include <stdio.h>

int vscanf(const char *restrict format, va_list arg);

int vfscanf(FILE *restrict fp, const char *restrict format,
            va_list arg);

int vsscanf(const char *restrict buf, const char *restrict format,
            va_list arg);
```

All three return: number of input items assigned, `EOF` if input error or end of file before any conversion

Refer to your UNIX system manual for additional details on the `scanf` family of functions.

5.12 Implementation Details

As we've mentioned, under the UNIX System, the standard I/O library ends up calling the I/O routines that we described in Chapter 3. Each standard I/O stream has an associated file descriptor, and we can obtain the descriptor for a stream by calling `fileno`.

> Note that `fileno` is not part of the ISO C standard, but rather an extension supported by POSIX.1.

```
#include <stdio.h>

int fileno(FILE *fp);
```

Returns: the file descriptor associated with the stream

We need this function if we want to call the `dup` or `fcntl` functions, for example.

To look at the implementation of the standard I/O library on your system, start with the header `<stdio.h>`. This will show how the `FILE` object is defined, the definitions of the per-stream flags, and any standard I/O routines, such as `getc`, that are defined as macros. Section 8.5 of Kernighan and Ritchie [1988] has a sample implementation that shows the flavor of many implementations on UNIX systems. Chapter 12 of Plauger [1992] provides the complete source code for an implementation of the standard I/O library. The implementation of the GNU standard I/O library is also publicly available.

Example

The program in Figure 5.11 prints the buffering for the three standard streams and for a stream that is associated with a regular file.

```
#include "apue.h"

void    pr_stdio(const char *, FILE *);
int     is_unbuffered(FILE *);
int     is_linebuffered(FILE *);
int     buffer_size(FILE *);

int
main(void)
{
    FILE    *fp;

    fputs("enter any character\n", stdout);
    if (getchar() == EOF)
        err_sys("getchar error");
    fputs("one line to standard error\n", stderr);

    pr_stdio("stdin",  stdin);
    pr_stdio("stdout", stdout);
    pr_stdio("stderr", stderr);
```

```
    if ((fp = fopen("/etc/passwd", "r")) == NULL)
        err_sys("fopen error");
    if (getc(fp) == EOF)
        err_sys("getc error");
    pr_stdio("/etc/passwd", fp);
    exit(0);
}

void
pr_stdio(const char *name, FILE *fp)
{
    printf("stream = %s, ", name);
    if (is_unbuffered(fp))
        printf("unbuffered");
    else if (is_linebuffered(fp))
        printf("line buffered");
    else /* if neither of above */
        printf("fully buffered");
    printf(", buffer size = %d\n", buffer_size(fp));
}

/*
 * The following is nonportable.
 */

#if defined(_IO_UNBUFFERED)

int
is_unbuffered(FILE *fp)
{
    return(fp->_flags & _IO_UNBUFFERED);
}

int
is_linebuffered(FILE *fp)
{
    return(fp->_flags & _IO_LINE_BUF);
}

int
buffer_size(FILE *fp)
{
    return(fp->_IO_buf_end - fp->_IO_buf_base);
}

#elif defined(__SNBF)

int
is_unbuffered(FILE *fp)
{
    return(fp->_flags & __SNBF);
}
```

```
int
is_linebuffered(FILE *fp)
{
    return(fp->_flags & __SLBF);
}

int
buffer_size(FILE *fp)
{
    return(fp->_bf._size);
}

#elif defined(_IONBF)

#ifdef _LP64
#define _flag __pad[4]
#define _ptr __pad[1]
#define _base __pad[2]
#endif

int
is_unbuffered(FILE *fp)
{
    return(fp->_flag & _IONBF);
}

int
is_linebuffered(FILE *fp)
{
    return(fp->_flag & _IOLBF);
}

int
buffer_size(FILE *fp)
{
#ifdef _LP64
    return(fp->_base - fp->_ptr);
#else
    return(BUFSIZ); /* just a guess */
#endif
}

#else

#error unknown stdio implementation!

#endif
```

Figure 5.11 Print buffering for various standard I/O streams

Note that we perform I/O on each stream before printing its buffering status, since the first I/O operation usually causes the buffers to be allocated for a stream. The structure members and the constants used in this example are defined by the implementations of

the standard I/O library used on the four platforms described in this book. Be aware that implementations of the standard I/O library vary, and programs like this example are nonportable, since they embed knowledge specific to particular implementations.

If we run the program in Figure 5.11 twice, once with the three standard streams connected to the terminal and once with the three standard streams redirected to files, we get the following result:

```
$ ./a.out                              stdin, stdout, and stderr connected to terminal
enter any character
                                       we type a newline
one line to standard error
stream = stdin, line buffered, buffer size = 1024
stream = stdout, line buffered, buffer size = 1024
stream = stderr, unbuffered, buffer size = 1
stream = /etc/passwd, fully buffered, buffer size = 4096
$ ./a.out < /etc/group > std.out 2> std.err
                                       run it again with all three streams redirected
$ cat std.err
one line to standard error
$ cat std.out
enter any character
stream = stdin, fully buffered, buffer size = 4096
stream = stdout, fully buffered, buffer size = 4096
stream = stderr, unbuffered, buffer size = 1
stream = /etc/passwd, fully buffered, buffer size = 4096
```

We can see that the default for this system is to have standard input and standard output line buffered when they're connected to a terminal. The line buffer is 1,024 bytes. Note that this doesn't restrict us to 1,024-byte input and output lines; that's just the size of the buffer. Writing a 2,048-byte line to standard output will require two `write` system calls. When we redirect these two streams to regular files, they become fully buffered, with buffer sizes equal to the preferred I/O size—the `st_blksize` value from the `stat` structure—for the file system. We also see that the standard error is always unbuffered, as it should be, and that a regular file defaults to fully buffered. □

5.13 Temporary Files

The ISO C standard defines two functions that are provided by the standard I/O library to assist in creating temporary files.

```
#include <stdio.h>

char *tmpnam(char *ptr);
                                        Returns: pointer to unique pathname

FILE *tmpfile(void);
                                    Returns: file pointer if OK, NULL on error
```

The tmpnam function generates a string that is a valid pathname and that does not match the name of any existing file. This function generates a different pathname each time it is called, up to TMP_MAX times. TMP_MAX is defined in <stdio.h>.

> Although ISO C defines TMP_MAX, the C standard requires only that its value be at least 25. The Single UNIX Specification, however, requires that XSI-conforming systems support a value of at least 10,000. This minimum value allows an implementation to use four digits (0000–9999), although most implementations on UNIX systems use alphanumeric characters.
>
> The tmpnam function is marked obsolescent in SUSv4, but the ISO C standard continues to support it.

If *ptr* is NULL, the generated pathname is stored in a static area, and a pointer to this area is returned as the value of the function. Subsequent calls to tmpnam can overwrite this static area. (Thus, if we call this function more than once and we want to save the pathname, we have to save a copy of the pathname, not a copy of the pointer.) If *ptr* is not NULL, it is assumed that it points to an array of at least L_tmpnam characters. (The constant L_tmpnam is defined in <stdio.h>.) The generated pathname is stored in this array, and *ptr* is returned as the value of the function.

The tmpfile function creates a temporary binary file (type wb+) that is automatically removed when it is closed or on program termination. Under the UNIX System, it makes no difference that this file is a binary file.

Example

The program in Figure 5.12 demonstrates these two functions.

```
#include "apue.h"

int
main(void)
{
    char    name[L_tmpnam], line[MAXLINE];
    FILE    *fp;

    printf("%s\n", tmpnam(NULL));       /* first temp name */

    tmpnam(name);                       /* second temp name */
    printf("%s\n", name);

    if ((fp = tmpfile()) == NULL)       /* create temp file */
        err_sys("tmpfile error");
    fputs("one line of output\n", fp);  /* write to temp file */
    rewind(fp);                         /* then read it back */
    if (fgets(line, sizeof(line), fp) == NULL)
        err_sys("fgets error");
    fputs(line, stdout);                /* print the line we wrote */

    exit(0);
}
```

Figure 5.12 Demonstrate tmpnam and tmpfile functions

If we execute the program in Figure 5.12, we get

```
$ ./a.out
/tmp/fileT0Hsu6
/tmp/filekmAsYQ
one line of output
```

□

The standard technique often used by the `tmpfile` function is to create a unique pathname by calling `tmpnam`, then create the file, and immediately `unlink` it. Recall from Section 4.15 that unlinking a file does not delete its contents until the file is closed. This way, when the file is closed, either explicitly or on program termination, the contents of the file are deleted.

The Single UNIX Specification defines two additional functions as part of the XSI option for dealing with temporary files: `mkdtemp` and `mkstemp`.

> Older versions of the Single UNIX Specification defined the `tempnam` function as a way to create a temporary file in a caller-specified location. It is marked obsolescent in SUSv4.

```
#include <stdlib.h>

char *mkdtemp(char *template);
                                Returns: pointer to directory name if OK, NULL on error

int mkstemp(char *template);
                                Returns: file descriptor if OK, –1 on error
```

The `mkdtemp` function creates a directory with a unique name, and the `mkstemp` function creates a regular file with a unique name. The name is selected using the *template* string. This string is a pathname whose last six characters are set to `XXXXXX`. The function replaces these placeholders with different characters to create a unique pathname. If successful, these functions modify the *template* string to reflect the name of the temporary file.

The directory created by `mkdtemp` is created with the following access permission bits set: `S_IRUSR | S_IWUSR | S_IXUSR`. Note that the file mode creation mask of the calling process can restrict these permissions further. If directory creation is successful, `mkdtemp` returns the name of the new directory.

The `mkstemp` function creates a regular file with a unique name and opens it. The file descriptor returned by `mkstemp` is open for reading and writing. The file created by `mkstemp` is created with access permissions `S_IRUSR | S_IWUSR`.

Unlike `tmpfile`, the temporary file created by `mkstemp` is not removed automatically for us. If we want to remove it from the file system namespace, we need to unlink it ourselves.

Use of `tmpnam` and `tempnam` does have at least one drawback: a window exists between the time that the unique pathname is returned and the time that an application creates a file with that name. During this timing window, another process can create a file of the same name. The `tmpfile` and `mkstemp` functions should be used instead, as they don't suffer from this problem.

Example

The program in Figure 5.13 shows how to use (and how not to use) the `mkstemp` function.

```
#include "apue.h"
#include <errno.h>

void make_temp(char *template);

int
main()
{
    char    good_template[] = "/tmp/dirXXXXXX"; /* right way */
    char    *bad_template = "/tmp/dirXXXXXX";   /* wrong way*/

    printf("trying to create first temp file...\n");
    make_temp(good_template);
    printf("trying to create second temp file...\n");
    make_temp(bad_template);
    exit(0);
}

void
make_temp(char *template)
{
    int         fd;
    struct stat sbuf;

    if ((fd = mkstemp(template)) < 0)
        err_sys("can't create temp file");
    printf("temp name = %s\n", template);
    close(fd);
    if (stat(template, &sbuf) < 0) {
        if (errno == ENOENT)
            printf("file doesn't exist\n");
        else
            err_sys("stat failed");
    } else {
        printf("file exists\n");
        unlink(template);
    }
}
```

Figure 5.13 Demonstrate `mkstemp` function

If we execute the program in Figure 5.13, we get

```
$ ./a.out
trying to create first temp file...
temp name = /tmp/dirUmBT7h
file exists
```

```
trying to create second temp file...
Segmentation fault
```

The difference in behavior comes from the way the two template strings are declared. For the first template, the name is allocated on the stack, because we use an array variable. For the second name, however, we use a pointer. In this case, only the memory for the pointer itself resides on the stack; the compiler arranges for the string to be stored in the read-only segment of the executable. When the `mkstemp` function tries to modify the string, a segmentation fault occurs. □

5.14 Memory Streams

As we've seen, the standard I/O library buffers data in memory, so operations such as character-at-a-time I/O and line-at-a-time I/O are more efficient. We've also seen that we can provide our own buffer for the library to use by calling `setbuf` or `setvbuf`. In Version 4, the Single UNIX Specification added support for *memory streams*. These are standard I/O streams for which there are no underlying files, although they are still accessed with `FILE` pointers. All I/O is done by transferring bytes to and from buffers in main memory. As we shall see, even though these streams look like file streams, several features make them more suited for manipulating character strings.

Three functions are available to create memory streams. The first is `fmemopen`.

```
#include <stdio.h>

FILE *fmemopen(void *restrict buf, size_t size,
               const char *restrict type);
```

Returns: stream pointer if OK, `NULL` on error

The `fmemopen` function allows the caller to provide a buffer to be used for the memory stream: the *buf* argument points to the beginning of the buffer and the *size* argument specifies the size of the buffer in bytes. If the *buf* argument is null, then the `fmemopen` function allocates a buffer of *size* bytes. In this case, the buffer will be freed when the stream is closed.

The *type* argument controls how the stream can be used. The possible values for *type* are summarized in Figure 5.14.

type	Description
`r` or `rb`	open for reading
`w` or `wb`	open for writing
`a` or `ab`	append; open for writing at first null byte
`r+` or `r+b` or `rb+`	open for reading and writing
`w+` or `w+b` or `wb+`	truncate to 0 length and open for reading and writing
`a+` or `a+b` or `ab+`	append; open for reading and writing at first null byte

Figure 5.14 The *type* argument for opening a memory stream

Note that these values correspond to the ones for file-based standard I/O streams, but there are some subtle differences. First, whenever a memory stream is opened for

append, the current file position is set to the first null byte in the buffer. If the buffer contains no null bytes, then the current position is set to one byte past the end of the buffer. When a stream is not opened for append, the current position is set to the beginning of the buffer. Because the append mode determines the end of the data by the first null byte, memory streams aren't well suited for storing binary data (which might contain null bytes before the end of the data).

Second, if the *buf* argument is a null pointer, it makes no sense to open the stream for only reading or only writing. Because the buffer is allocated by `fmemopen` in this case, there is no way to find the buffer's address, so to open the stream only for writing means we could never read what we've written. Similarly, to open the stream only for reading means we can only read the contents of a buffer into which we can never write.

Third, a null byte is written at the current position in the stream whenever we increase the amount of data in the stream's buffer and call `fclose`, `fflush`, `fseek`, `fseeko`, or `fsetpos`.

Example

It's instructive to look at how writes to a memory stream operate on a buffer we provide. Figure 5.15 shows a sample program that seeds the buffer with a known pattern to see how writes to the stream behave.

```
#include "apue.h"

#define BSZ 48

int
main()
{
    FILE *fp;
    char buf[BSZ];

    memset(buf, 'a', BSZ-2);
    buf[BSZ-2] = '\0';
    buf[BSZ-1] = 'X';
    if ((fp = fmemopen(buf, BSZ, "w+")) == NULL)
        err_sys("fmemopen failed");
    printf("initial buffer contents: %s\n", buf);
    fprintf(fp, "hello, world");
    printf("before flush: %s\n", buf);
    fflush(fp);
    printf("after fflush: %s\n", buf);
    printf("len of string in buf = %ld\n", (long)strlen(buf));

    memset(buf, 'b', BSZ-2);
    buf[BSZ-2] = '\0';
    buf[BSZ-1] = 'X';
    fprintf(fp, "hello, world");
    fseek(fp, 0, SEEK_SET);
    printf("after  fseek: %s\n", buf);
    printf("len of string in buf = %ld\n", (long)strlen(buf));
```

```
    memset(buf, 'c', BSZ-2);
    buf[BSZ-2] = '\0';
    buf[BSZ-1] = 'X';
    fprintf(fp, "hello, world");
    fclose(fp);
    printf("after fclose: %s\n", buf);
    printf("len of string in buf = %ld\n", (long)strlen(buf));

    return(0);
}
```

Figure 5.15 Investigate memory stream write behavior

When we run the program on Linux, we get the following:

```
$ ./a.out
                                    overwrite the buffer with a's
initial buffer contents:            fmemopen places a null byte at beginning of buffer
before flush:                       buffer is unchanged until stream is flushed
after fflush: hello, world
len of string in buf = 12           null byte added to end of string
                                    now overwrite the buffer with b's
after  fseek: bbbbbbbbbbbbhello, world        fseek causes flush
len of string in buf = 24           null byte appended again
                                    now overwrite the buffer with c's
after fclose: hello, worldccccccccccccccccccccccccccccccccc
len of string in buf = 46           no null byte appended
```

This example shows the policy for flushing memory streams and appending null bytes. A null byte is appended automatically whenever we write to a memory stream and advance the stream's notion of the size of the stream's contents (as opposed to the size of the buffer, which is fixed). The size of the stream's contents is determined by how much we write to it.

> Of the four platforms covered in this book, only Linux 3.2.0 provides support for memory streams. This is a case of the implementations not having caught up yet with the latest standards, and will change with time.
>
> □

The other two functions that can be used to create a memory stream are `open_memstream` and `open_wmemstream`.

```
#include <stdio.h>

FILE *open_memstream(char **bufp, size_t *sizep);

#include <wchar.h>

FILE *open_wmemstream(wchar_t **bufp, size_t *sizep);
```

Both return: stream pointer if OK, `NULL` on error

The `open_memstream` function creates a stream that is byte oriented, and the `open_wmemstream` function creates a stream that is wide oriented (recall the discussion of multibyte characters in Section 5.2). These two functions differ from `fmemopen` in several ways:

- The stream created is only open for writing.
- We can't specify our own buffer, but we can get access to the buffer's address and size through the *bufp* and *sizep* arguments, respectively.
- We need to free the buffer ourselves after closing the stream.
- The buffer will grow as we add bytes to the stream.

We must follow some rules, however, regarding the use of the buffer address and its length. First, the buffer address and length are only valid after a call to `fclose` or `fflush`. Second, these values are only valid until the next write to the stream or a call to `fclose`. Because the buffer can grow, it may need to be reallocated. If this happens, then we will find that the value of the buffer's memory address will change the next time we call `fclose` or `fflush`.

Memory streams are well suited for creating strings, because they prevent buffer overflows. They can also provide a performance boost for functions that take standard I/O stream arguments used for temporary files, because memory streams access only main memory instead of a file stored on disk.

5.15 Alternatives to Standard I/O

The standard I/O library is not perfect. Korn and Vo [1991] list numerous defects—some in the basic design, but most in the various implementations.

One inefficiency inherent in the standard I/O library is the amount of data copying that takes place. When we use the line-at-a-time functions, `fgets` and `fputs`, the data is usually copied twice: once between the kernel and the standard I/O buffer (when the corresponding `read` or `write` is issued) and again between the standard I/O buffer and our line buffer. The Fast I/O library [`fio`(3) in AT&T 1990a] gets around this by having the function that reads a line return a pointer to the line instead of copying the line into another buffer. Hume [1988] reports a threefold increase in the speed of a version of the `grep`(1) utility simply by making this change.

Korn and Vo [1991] describe another replacement for the standard I/O library: *sfio*. This package is similar in speed to the *fio* library and normally faster than the standard I/O library. The *sfio* package also provides some new features that aren't found in most other packages: I/O streams generalized to represent both files and regions of memory, processing modules that can be written and stacked on an I/O stream to change the operation of a stream, and better exception handling.

Krieger, Stumm, and Unrau [1992] describe another alternative that uses mapped files—the `mmap` function that we describe in Section 14.8. This new package is called ASI, the Alloc Stream Interface. The programming interface resembles the UNIX System memory allocation functions (`malloc`, `realloc`, and `free`, described in

Section 7.8). As with the *sfio* package, ASI attempts to minimize the amount of data copying by using pointers.

Several implementations of the standard I/O library are available in C libraries that were designed for systems with small memory footprints, such as embedded systems. These implementations emphasize modest memory requirements over portability, speed, or functionality. Two such implementations are the uClibc C library (see `http://www.uclibc.org` for more information) and the Newlib C library (`http://sources.redhat.com/newlib`).

5.16 Summary

The standard I/O library is used by most UNIX applications. In this chapter, we looked at many of the functions provided by this library, as well as at some implementation details and efficiency considerations. Be aware of the buffering that takes place with this library, as this is the area that generates the most problems and confusion.

Exercises

5.1 Implement `setbuf` using `setvbuf`.

5.2 Type in the program that copies a file using line-at-a-time I/O (`fgets` and `fputs`) from Figure 5.5, but use a `MAXLINE` of 4. What happens if you copy lines that exceed this length? Explain what is happening.

5.3 What does a return value of 0 from `printf` mean?

5.4 The following code works correctly on some machines, but not on others. What could be the problem?

```
#include    <stdio.h>

int
main(void)
{
    char    c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

5.5 How would you use the `fsync` function (Section 3.13) with a standard I/O stream?

5.6 In the programs in Figures 1.7 and 1.10, the prompt that is printed does not contain a newline, and we don't call `fflush`. What causes the prompt to be output?

5.7 BSD-based systems provide a function called `funopen` that allows us to intercept read, write, seek, and close calls on a stream. Use this function to implement `fmemopen` for FreeBSD and Mac OS X.

This page intentionally left blank

6

System Data Files and Information

6.1 Introduction

A UNIX system requires numerous data files for normal operation: the password file `/etc/passwd` and the group file `/etc/group` are two files that are frequently used by various programs. For example, the password file is used every time a user logs in to a UNIX system and every time someone executes an `ls -l` command.

Historically, these data files have been ASCII text files and were read with the standard I/O library. But for larger systems, a sequential scan through the password file becomes time consuming. We want to be able to store these data files in a format other than ASCII text, but still provide an interface for an application program that works with any file format. The portable interfaces to these data files are the subject of this chapter. We also cover the system identification functions and the time and date functions.

6.2 Password File

The UNIX System's password file, called the user database by POSIX.1, contains the fields shown in Figure 6.1. These fields are contained in a `passwd` structure that is defined in `<pwd.h>`.

> Note that POSIX.1 specifies only 5 of the 10 fields in the `passwd` structure. Most platforms support at least 7 of the fields. The BSD-derived platforms support all 10.

Description	struct passwd member	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
user name	char *pw_name	•	•	•	•	•
encrypted password	char *pw_passwd		•	•	•	•
numerical user ID	uid_t pw_uid	•	•	•	•	•
numerical group ID	gid_t pw_gid	•	•	•	•	•
comment field	char *pw_gecos		•	•	•	•
initial working directory	char *pw_dir	•	•	•	•	•
initial shell (user program)	char *pw_shell	•	•	•	•	•
user access class	char *pw_class		•		•	
next time to change password	time_t pw_change		•		•	
account expiration time	time_t pw_expire		•		•	

Figure 6.1 Fields in `/etc/passwd` file

Historically, the password file has been stored in `/etc/passwd` and has been an ASCII file. Each line contains the fields described in Figure 6.1, separated by colons. For example, four lines from the `/etc/passwd` file on Linux could be

```
root:x:0:0:root:/root:/bin/bash
squid:x:23:23::/var/spool/squid:/dev/null
nobody:x:65534:65534:Nobody:/home:/bin/sh
sar:x:205:105:Stephen Rago:/home/sar:/bin/bash
```

Note the following points about these entries.

- There is usually an entry with the user name `root`. This entry has a user ID of 0 (the superuser).
- The encrypted password field contains a single character as a placeholder where older versions of the UNIX System used to store the encrypted password. Because it is a security hole to store the encrypted password in a file that is readable by everyone, encrypted passwords are now kept elsewhere. We'll cover this issue in more detail in the next section when we discuss passwords.
- Some fields in a password file entry can be empty. If the encrypted password field is empty, it usually means that the user does not have a password. (This is not recommended.) The entry for `squid` has one blank field: the comment field. An empty comment field has no effect.
- The shell field contains the name of the executable program to be used as the login shell for the user. The default value for an empty shell field is usually `/bin/sh`. Note, however, that the entry for `squid` has `/dev/null` as the login shell. Obviously, this is a device and cannot be executed, so its use here is to prevent anyone from logging in to our system as user `squid`.

  Many services have separate user IDs for the daemon processes (Chapter 13) that help implement the service. The `squid` entry is for the processes implementing the `squid` proxy cache service.

- There are several alternatives to using `/dev/null` to prevent a particular user from logging in to a system. For example, `/bin/false` is often used as the login shell. It simply exits with an unsuccessful (nonzero) status; the shell evaluates the exit status as false. It is also common to see `/bin/true` used to disable an account; it simply exits with a successful (zero) status. Some systems provide the `nologin` command, which prints a customizable error message and exits with a nonzero exit status.
- The `nobody` user name can be used to allow people to log in to a system, but with a user ID (65534) and group ID (65534) that provide no privileges. The only files that this user ID and group ID can access are those that are readable or writable by the world. (This approach assumes that there are no files specifically owned by user ID 65534 or group ID 65534, which should be the case.)
- Some systems that provide the `finger`(1) command support additional information in the comment field. Each of these fields is separated by a comma: the user's name, office location, office phone number, and home phone number. Additionally, an ampersand in the comment field is replaced with the login name (capitalized) by some utilities. For example, we could have

  ```
  sar:x:205:105:Steve Rago, SF 5-121, 555-1111, 555-2222:/home/sar:/bin/sh
  ```

  Then we could use `finger` to print information about Steve Rago.

  ```
  $ finger -p sar
  Login: sar                         Name: Steve Rago
  Directory: /home/sar               Shell: /bin/sh
  Office:  SF 5-121,  555-1111       Home Phone:  555-2222
  On since Mon Jan 19 03:57 (EST) on ttyv0 (messages off)
  No Mail.
  ```

  Even if your system doesn't support the `finger` command, these fields can still go into the comment field, since that field is simply a comment and not interpreted by system utilities.

Some systems provide the `vipw` command to allow administrators to edit the password file. The `vipw` command serializes changes to the password file and makes sure that any additional files are consistent with the changes made. It is also common for systems to provide similar functionality through graphical user interfaces.

POSIX.1 defines two functions to fetch entries from the password file. These functions allow us to look up an entry given a user's login name or numerical user ID.

```
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);

struct passwd *getpwnam(const char *name);
```

Both return: pointer if OK, `NULL` on error

The `getpwuid` function is used by the `ls`(1) program to map the numerical user ID contained in an i-node into a user's login name. The `getpwnam` function is used by the `login`(1) program when we enter our login name.

Both functions return a pointer to a `passwd` structure that the functions fill in. This structure is usually a `static` variable within the function, so its contents are overwritten each time we call either of these functions.

These two POSIX.1 functions are fine if we want to look up either a login name or a user ID, but some programs need to go through the entire password file. Three functions can be used for this purpose: `getpwent`, `setpwent`, and `endpwent`.

```
#include <pwd.h>

struct passwd *getpwent(void);
                                    Returns: pointer if OK, NULL on error or end of file

void setpwent(void);

void endpwent(void);
```

These three functions are not part of the base POSIX.1 standard. They are defined as part of the XSI option in the Single UNIX Specification. As such, all UNIX systems are expected to provide them.

We call `getpwent` to return the next entry in the password file. As with the two POSIX.1 functions, `getpwent` returns a pointer to a structure that it has filled in. This structure is normally overwritten each time we call this function. If this is the first call to this function, it opens whatever files it uses. There is no order implied when we use this function; the entries can be in any order, because some systems use a hashed version of the file `/etc/passwd`.

The function `setpwent` rewinds whatever files it uses, and `endpwent` closes these files. When using `getpwent`, we must always be sure to close these files by calling `endpwent` when we're through. Although `getpwent` is smart enough to know when it has to open its files (the first time we call it), it never knows when we're through.

Example

Figure 6.2 shows an implementation of the function `getpwnam`.

```
#include <pwd.h>
#include <stddef.h>
#include <string.h>

struct passwd *
getpwnam(const char *name)
{
    struct passwd  *ptr;

    setpwent();
    while ((ptr = getpwent()) != NULL)
        if (strcmp(name, ptr->pw_name) == 0)
            break;      /* found a match */
    endpwent();
    return(ptr);    /* ptr is NULL if no match found */
}
```

Figure 6.2 The `getpwnam` function

The call to `setpwent` at the beginning of this function is self-defense: we ensure that the files are rewound, in case the caller has already opened them by calling `getpwent`. We call `endpwent` when we're done, because neither `getpwnam` nor `getpwuid` should leave any of the files open. □

6.3 Shadow Passwords

The encrypted password is a copy of the user's password that has been put through a one-way encryption algorithm. Because this algorithm is one-way, we can't guess the original password from the encrypted version.

Historically, the algorithm used always generated 13 printable characters from the 64-character set `[a-zA-Z0-9./]` (see Morris and Thompson [1979]). Some newer systems use alternative algorithms, such as MD5 or SHA-1, to generate longer encrypted password strings. (The more characters used to store the encrypted password, the more combinations there are, and the harder it will be to guess the password by trying all possible variations.) When we place a single character in the encrypted password field, we ensure that an encrypted password will never match this value.

Given an encrypted password, we can't apply an algorithm that inverts it and returns the plaintext password. (The plaintext password is what we enter at the `Password:` prompt.) But we could guess a password, run it through the one-way algorithm, and compare the result to the encrypted password. If user passwords were randomly chosen, this brute-force approach wouldn't be too successful. Users, however, tend to choose nonrandom passwords, such as spouse's name, street names, or pet names. A common experiment is for someone to obtain a copy of the password file and try guessing the passwords. (Chapter 4 of Garfinkel et al. [2003] contains additional details and history on passwords and the password encryption scheme used on UNIX systems.)

To make it more difficult to obtain the raw materials (the encrypted passwords), systems now store the encrypted password in another file, often called the *shadow password file*. Minimally, this file has to contain the user name and the encrypted password. Other information relating to the password is also stored here (Figure 6.3).

Description	`struct spwd` member
user login name	`char *sp_namp`
encrypted password	`char *sp_pwdp`
days since Epoch of last password change	`int sp_lstchg`
days until change allowed	`int sp_min`
days before change required	`int sp_max`
days warning for expiration	`int sp_warn`
days before account inactive	`int sp_inact`
days since Epoch when account expires	`int sp_expire`
reserved	`unsigned int sp_flag`

Figure 6.3 Fields in `/etc/shadow` file

The only two mandatory fields are the user's login name and encrypted password. The other fields control how often the password is to change—known as "password aging"—and how long an account is allowed to remain active.

The shadow password file should not be readable by the world. Only a few programs need to access encrypted passwords—`login`(1) and `passwd`(1), for example—and these programs are often set-user-ID root. With shadow passwords, the regular password file, `/etc/passwd`, can be left readable by the world.

On Linux 3.2.0 and Solaris 10, a separate set of functions is available to access the shadow password file, similar to the set of functions used to access the password file.

```
#include <shadow.h>

struct spwd *getspnam(const char *name);

struct spwd *getspent(void);
```

Both return: pointer if OK, `NULL` on error

```
void setspent(void);

void endspent(void);
```

On FreeBSD 8.0 and Mac OS X 10.6.8, there is no shadow password structure. The additional account information is stored in the password file (refer back to Figure 6.1).

6.4 Group File

The UNIX System's group file, called the group database by POSIX.1, contains the fields shown in Figure 6.4. These fields are contained in a `group` structure that is defined in `<grp.h>`.

Description	`struct group` member	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
group name	`char *gr_name`	•	•	•	•	•
encrypted password	`char *gr_passwd`		•	•	•	•
numerical group ID	`int gr_gid`	•	•	•	•	•
array of pointers to individual user names	`char **gr_mem`	•	•	•	•	•

Figure 6.4 Fields in `/etc/group` file

The field `gr_mem` is an array of pointers to the user names that belong to this group. This array is terminated by a null pointer.

We can look up either a group name or a numerical group ID with the following two functions, which are defined by POSIX.1.

```
#include <grp.h>

struct group *getgrgid(gid_t gid);

struct group *getgrnam(const char *name);
```

Both return: pointer if OK, `NULL` on error

Like the password file functions, both of these functions normally return pointers to a `static` variable, which is overwritten on each call.

If we want to search the entire group file, we need some additional functions. The following three functions are like their counterparts for the password file.

```
#include <grp.h>

struct group *getgrent(void);
                                    Returns: pointer if OK, NULL on error or end of file

void setgrent(void);

void endgrent(void);
```

These three functions are not part of the base POSIX.1 standard. They are defined as part of the XSI option in the Single UNIX Specification. All UNIX Systems provide them.

The `setgrent` function opens the group file, if it's not already open, and rewinds it. The `getgrent` function reads the next entry from the group file, opening the file first, if it's not already open. The `endgrent` function closes the group file.

6.5 Supplementary Group IDs

The use of groups in the UNIX System has changed over time. With Version 7, each user belonged to a single group at any point in time. When we logged in, we were assigned the real group ID corresponding to the numerical group ID in our password file entry. We could change this at any point by executing `newgrp`(1). If the `newgrp` command succeeded (refer to the manual page for the permission rules), our real group ID was changed to the new group's ID, and this value was used for all subsequent file access permission checks. We could always go back to our original group by executing `newgrp` without any arguments.

This form of group membership persisted until it was changed in 4.2BSD (circa 1983). With 4.2BSD, the concept of supplementary group IDs was introduced. Not only did we belong to the group corresponding to the group ID in our password file entry, but we could also belong to as many as 16 additional groups. The file access permission checks were modified so that in addition to comparing the the file's group ID to the process effective group ID, it was also compared to all the supplementary group IDs.

Supplementary group IDs are a required feature of POSIX.1. (In older versions of POSIX.1, they were optional.) The constant `NGROUPS_MAX` (Figure 2.11) specifies the number of supplementary group IDs. A common value is 16 (Figure 2.15).

The advantage of using supplementary group IDs is that we no longer have to change groups explicitly. It is not uncommon to belong to multiple groups (i.e., participate in multiple projects) at the same time.

Three functions are provided to fetch and set the supplementary group IDs.

```
#include <unistd.h>

int getgroups(int gidsetsize, gid_t grouplist[]);
```

Returns: number of supplementary group IDs if OK, –1 on error

```
#include <grp.h>    /* on Linux */
#include <unistd.h> /* on FreeBSD, Mac OS X, and Solaris */

int setgroups(int ngroups, const gid_t grouplist[]);

#include <grp.h>    /* on Linux and Solaris */
#include <unistd.h> /* on FreeBSD and Mac OS X */

int initgroups(const char *username, gid_t basegid);
```

Both return: 0 if OK, –1 on error

Of these three functions, only `getgroups` is specified by POSIX.1. Because `setgroups` and `initgroups` are privileged operations, they are not part of POSIX.1. All four platforms covered in this book support all three functions, but on Mac OS X 10.6.8, *basegid* is declared to be of type `int`.

The `getgroups` function fills in the array *grouplist* with the supplementary group IDs. Up to *gidsetsize* elements are stored in the array. The number of supplementary group IDs stored in the array is returned by the function.

As a special case, if *gidsetsize* is 0, the function returns only the number of supplementary group IDs. The array *grouplist* is not modified. (This allows the caller to determine the size of the *grouplist* array to allocate.)

The `setgroups` function can be called by the superuser to set the supplementary group ID list for the calling process: *grouplist* contains the array of group IDs, and *ngroups* specifies the number of elements in the array. The value of *ngroups* cannot be larger than `NGROUPS_MAX`.

The `setgroups` function is usually called from the `initgroups` function, which reads the entire group file—with the functions `getgrent`, `setgrent`, and `endgrent`, which we described earlier—and determines the group membership for *username*. It then calls `setgroups` to initialize the supplementary group ID list for the user. One must be superuser to call `initgroups`, since it calls `setgroups`. In addition to finding all the groups that *username* is a member of in the group file, `initgroups` includes *basegid* in the supplementary group ID list; *basegid* is the group ID from the password file for *username*.

The `initgroups` function is called by only a few programs. The `login`(1) program, for example, calls it when we log in.

6.6 Implementation Differences

We've already discussed the shadow password file supported by Linux and Solaris. FreeBSD and Mac OS X store encrypted passwords differently. Figure 6.5 summarizes how the four platforms covered in this book store user and group information.

Information	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
account information	`/etc/passwd`	`/etc/passwd`	Directory Services	`/etc/passwd`
encrypted passwords	`/etc/master.passwd`	`/etc/shadow`	Directory Services	`/etc/shadow`
hashed password files?	yes	no	no	no
group information	`/etc/group`	`/etc/group`	Directory Services	`/etc/group`

Figure 6.5 Account implementation differences

On FreeBSD, the shadow password file is `/etc/master.passwd`. Special commands are used to edit it, which in turn generate a copy of `/etc/passwd` from the shadow password file. In addition, hashed versions of the files are generated: `/etc/pwd.db` is the hashed version of `/etc/passwd`, and `/etc/spwd.db` is the hashed version of `/etc/master.passwd`. These provide better performance for large installations.

On Mac OS X, however, `/etc/passwd` and `/etc/master.passwd` are used only in single-user mode (when the system is undergoing maintenance; single-user mode usually means that no system services are enabled). In multiuser mode—during normal operation—the Directory Services daemon provides access to account information for users and groups.

Although Linux and Solaris support similar shadow password interfaces, there are some subtle differences. For example, the integer fields shown in Figure 6.3 are defined as type `int` on Solaris, but as `long int` on Linux. Another difference is the account-inactive field: Solaris defines it to be the number of days since the user last logged in to the system after which the account will be automatically disabled, whereas Linux defines it to be the number of days after the maximum password age has been reached during which the password will still be accepted.

On many systems, the user and group databases are implemented using the Network Information Service (NIS). This allows administrators to edit a master copy of the databases and distribute them automatically to all servers in an organization. Client systems contact servers to look up information about users and groups. NIS+ and the Lightweight Directory Access Protocol (LDAP) provide similar functionality. Many systems control the method used to administer each type of information through the `/etc/nsswitch.conf` configuration file.

6.7 Other Data Files

We've discussed only two of the system's data files so far: the password file and the group file. Numerous other files are used by UNIX systems in normal day-to-day operation. For example, the BSD networking software has one data file for the services provided by the various network servers (`/etc/services`), one for the protocols (`/etc/protocols`), and one for the networks (`/etc/networks`). Fortunately, the interfaces to these various files are like the ones we've already described for the password and group files.

The general principle is that every data file has at least three functions:

1. A `get` function that reads the next record, opening the file if necessary. These functions normally return a pointer to a structure. A null pointer is returned when the end of file is reached. Most of the `get` functions return a pointer to a `static` structure, so we always have to copy the structure if we want to save it.
2. A `set` function that opens the file, if not already open, and rewinds the file. We use this function when we know we want to start again at the beginning of the file.
3. An `end` entry that closes the data file. As we mentioned earlier, we always have to call this function when we're done, to close all the files.

Additionally, if the data file supports some form of keyed lookup, routines are provided to search for a record with a specific key. For example, two keyed lookup routines are provided for the password file: `getpwnam` looks for a record with a specific user name, and `getpwuid` looks for a record with a specific user ID.

Figure 6.6 shows some of these routines, which are common to UNIX systems. In this figure, we show the functions for the password files and group file, which we discussed earlier in this chapter, and some of the networking functions. There are `get`, `set`, and `end` functions for all the data files in this figure.

Description	Data file	Header	Structure	Additional keyed lookup functions
passwords	/etc/passwd	<pwd.h>	passwd	getpwnam, getpwuid
groups	/etc/group	<grp.h>	group	getgrnam, getgrgid
shadow	/etc/shadow	<shadow.h>	spwd	getspnam
hosts	/etc/hosts	<netdb.h>	hostent	getnameinfo, getaddrinfo
networks	/etc/networks	<netdb.h>	netent	getnetbyname, getnetbyaddr
protocols	/etc/protocols	<netdb.h>	protoent	getprotobyname, getprotobynumber
services	/etc/services	<netdb.h>	servent	getservbyname, getservbyport

Figure 6.6 Similar routines for accessing system data files

Under Solaris, the last four data files in Figure 6.6 are symbolic links to files of the same name in the directory `/etc/inet`. Most UNIX System implementations have additional functions that are like these, but the additional functions tend to deal with system administration files and are specific to each implementation.

6.8 Login Accounting

Two data files provided with most UNIX systems are the `utmp` file, which keeps track of all the users currently logged in, and the `wtmp` file, which keeps track of all logins and logouts. With Version 7, one type of record was written to both files, a binary record consisting of the following structure:

```
struct utmp {
  char  ut_line[8]; /* tty line: "ttyh0", "ttyd0", "ttyp0", ... */
  char  ut_name[8]; /* login name */
  long  ut_time;    /* seconds since Epoch */
};
```

On login, one of these structures was filled in and written to the `utmp` file by the `login` program, and the same structure was appended to the `wtmp` file. On logout, the entry in the `utmp` file was erased—filled with null bytes—by the `init` process, and a new entry was appended to the `wtmp` file. This logout entry in the `wtmp` file had the `ut_name` field zeroed out. Special entries were appended to the `wtmp` file to indicate when the system was rebooted and right before and after the system's time and date was changed. The `who`(1) program read the `utmp` file and printed its contents in a readable form. Later versions of the UNIX System provided the `last`(1) command, which read through the `wtmp` file and printed selected entries.

Most versions of the UNIX System still provide the `utmp` and `wtmp` files, but as expected, the amount of information in these files has grown. The 20-byte structure that was written by Version 7 grew to 36 bytes with SVR2, and the extended `utmp` structure with SVR4 takes more than 350 bytes!

> The detailed format of these records in Solaris is given in the `utmpx`(4) manual page. With Solaris 10, both files are in the `/var/adm` directory. Solaris provides numerous functions described in `getutxent`(3) to read and write these two files.
>
> On FreeBSD 8.0 and Linux 3.2.0, the `utmp`(5) manual page gives the format of their versions of these login records. The pathnames of these two files are `/var/run/utmp` and `/var/log/wtmp`. On Mac OS X 10.6.8, the `utmp` and `wtmp` files do not exist. As of Mac OS X 10.5, the information found in the `wtmp` file can be obtained from the system logging facility, and the `utmpx` file contains information about the active login sessions.

6.9 System Identification

POSIX.1 defines the `uname` function to return information on the current host and operating system.

```
#include <sys/utsname.h>

int uname(struct utsname *name);
```

Returns: non-negative value if OK, –1 on error

We pass the address of a `utsname` structure to this function, and the function then fills it in. POSIX.1 defines only the minimum fields in the structure, which are all character arrays, and it's up to each implementation to set the size of each array. Some implementations provide additional fields in the structure.

```
struct utsname {
  char  sysname[];    /* name of the operating system */
  char  nodename[];   /* name of this node */
  char  release[];    /* current release of operating system */
  char  version[];    /* current version of this release */
  char  machine[];    /* name of hardware type */
};
```

Each string is null terminated. The maximum name lengths, including the terminating null byte, supported by the four platforms discussed in this book are listed in Figure 6.7. The information in the `utsname` structure can usually be printed with the `uname`(1) command.

> POSIX.1 warns that the `nodename` element may not be adequate to reference the host on a communications network. This function is from System V, and in older days, the `nodename` element was adequate for referencing the host on a UUCP network.
>
> Realize also that the information in this structure does not give any information on the POSIX.1 level. This should be obtained using `_POSIX_VERSION`, as described in Section 2.6.
>
> Finally, this function gives us a way only to fetch the information in the structure; there is nothing specified by POSIX.1 about initializing this information.

Historically, BSD-derived systems provided the `gethostname` function to return only the name of the host. This name is usually the name of the host on a TCP/IP network.

```
#include <unistd.h>

int gethostname(char *name, int namelen);
```

Returns: 0 if OK, –1 on error

The *namelen* argument specifies the size of the *name* buffer. If enough space is provided, the string returned through *name* is null terminated. If insufficient room is provided, however, it is unspecified whether the string is null terminated.

The `gethostname` function, which is now defined as part of POSIX.1, specifies that the maximum host name length is `HOST_NAME_MAX`. Figure 6.7 summarizes the maximum name lengths supported by the four implementations covered in this book.

Interface	Maximum name length			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
`uname`	256	65	256	257
`gethostname`	256	64	256	256

Figure 6.7 System identification name limits

If the host is connected to a TCP/IP network, the host name is normally the fully qualified domain name of the host.

There is also a `hostname`(1) command that can fetch or set the host name. (The host name is set by the superuser using a similar function, `sethostname`.) The host name is normally set at bootstrap time from one of the start-up files invoked by `/etc/rc` or `init`.

6.10 Time and Date Routines

The basic time service provided by the UNIX kernel counts the number of seconds that have passed since the Epoch: 00:00:00 January 1, 1970, Coordinated Universal Time (UTC). In Section 1.10, we said that these seconds are represented in a `time_t` data type, and we call them *calendar times*. These calendar times represent both the time and the date. The UNIX System has always differed from other operating systems in (a) keeping time in UTC instead of the local time, (b) automatically handling conversions, such as daylight saving time, and (c) keeping the time and date as a single quantity.

The `time` function returns the current time and date.

```
#include <time.h>

time_t time(time_t *calptr);
```

Returns: value of time if OK, –1 on error

The time value is always returned as the value of the function. If the argument is non-null, the time value is also stored at the location pointed to by *calptr*.

The real-time extensions to POSIX.1 added support for multiple system clocks. In Version 4 of the Single UNIX Specification, the interfaces used to control these clocks were moved from an option group to the base. A clock is identified by the `clockid_t` type. Standard values are summarized in Figure 6.8.

Identifier	Option	Description
CLOCK_REALTIME		real system time
CLOCK_MONOTONIC	_POSIX_MONOTONIC_CLOCK	real system time with no negative jumps
CLOCK_PROCESS_CPUTIME_ID	_POSIX_CPUTIME	CPU time for calling process
CLOCK_THREAD_CPUTIME_ID	_POSIX_THREAD_CPUTIME	CPU time for calling thread

Figure 6.8 Clock type identifiers

The `clock_gettime` function can be used to get the time of the specified clock. The time is returned in a `timespec` structure, introduced in Section 4.2, which expresses time values in terms of seconds and nanoseconds.

```
#include <sys/time.h>

int clock_gettime(clockid_t clock_id, struct timespec *tsp);
```

Returns: 0 if OK, –1 on error

When the clock ID is set to `CLOCK_REALTIME`, the `clock_gettime` function provides similar functionality to the `time` function, except with `clock_gettime`, we might be able to get a higher-resolution time value if the system supports it.

We can use the `clock_getres` function to determine the resolution of a given system clock.

```
#include <sys/time.h>

int clock_getres(clockid_t clock_id, struct timespec *tsp);
```

Returns: 0 if OK, –1 on error

The `clock_getres` function initializes the `timespec` structure pointed to by the *tsp* argument to the resolution of the clock corresponding to the *clock_id* argument. For example, if the resolution is 1 millisecond, then the `tv_sec` field will contain 0 and the `tv_nsec` field will contain the value 1000000.

To set the time for a particular clock, we can call the `clock_settime` function.

```
#include <sys/time.h>

int clock_settime(clockid_t clock_id, const struct timespec *tsp);
```

Returns: 0 if OK, –1 on error

We need the appropriate privileges to change a clock's time. Some clocks, however, can't be modified.

> Historically, on implementations derived from System V, the `stime`(2) function was called to set the system time, whereas BSD-derived systems used `settimeofday`(2).

Version 4 of the Single UNIX Specification specifies that the `gettimeofday` function is now obsolescent. However, a lot of programs still use it, because it provides greater resolution (up to a microsecond) than the `time` function.

```
#include <sys/time.h>

int gettimeofday(struct timeval *restrict tp, void *restrict tzp);
```

Returns: 0 always

The only legal value for *tzp* is `NULL`; other values result in unspecified behavior. Some platforms support the specification of a time zone through the use of *tzp*, but this is implementation specific and not defined by the Single UNIX Specification.

The `gettimeofday` function stores the current time as measured from the Epoch in the memory pointed to by *tp*. This time is represented as a `timeval` structure, which stores seconds and microseconds.

Once we have the integer value that counts the number of seconds since the Epoch, we normally call a function to convert it to a broken-down time structure, and then call another function to generate a human-readable time and date. Figure 6.9 shows the relationships between the various time functions. (The three functions in this figure that are shown with dashed lines—`localtime`, `mktime`, and `strftime`—are all affected by the `TZ` environment variable, which we describe later in this section. The dotted lines show how the calendar time is obtained from time-related structures.)

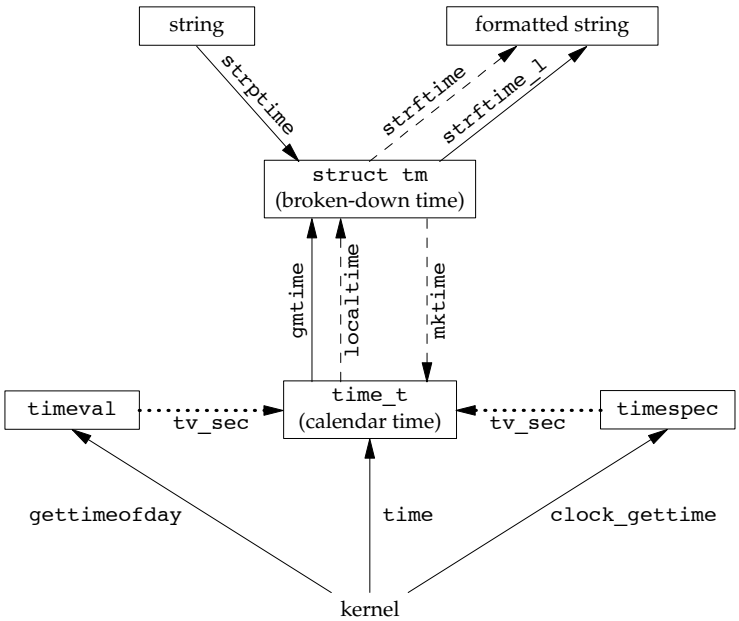


Figure 6.9 Relationship of the various time functions

The two functions `localtime` and `gmtime` convert a calendar time into what's called a broken-down time, a `tm` structure.

```
struct tm {      /* a broken-down time */
  int  tm_sec;   /* seconds after the minute: [0 - 60] */
  int  tm_min;   /* minutes after the hour: [0 - 59] */
  int  tm_hour;  /* hours after midnight: [0 - 23] */
  int  tm_mday;  /* day of the month: [1 - 31] */
  int  tm_mon;   /* months since January: [0 - 11] */
  int  tm_year;  /* years since 1900 */
  int  tm_wday;  /* days since Sunday: [0 - 6] */
  int  tm_yday;  /* days since January 1: [0 - 365] */
  int  tm_isdst; /* daylight saving time flag: <0, 0, >0 */
};
```

The reason that the seconds can be greater than 59 is to allow for a leap second. Note that all the fields except the day of the month are 0-based. The daylight saving time flag is positive if daylight saving time is in effect, 0 if it's not in effect, and negative if the information isn't available.

> In older versions of the Single UNIX Specification, double leap seconds were allowed. Thus the valid range of values for the `tm_sec` member was 0–61. The formal definition of UTC doesn't allow for double leap seconds, so the valid range for seconds is now 0–60.

```
#include <time.h>

struct tm *gmtime(const time_t *calptr);

struct tm *localtime(const time_t *calptr);
```

Both return: pointer to broken-down time, NULL on error

The difference between localtime and gmtime is that the first converts the calendar time to the local time, taking into account the local time zone and daylight saving time flag, whereas the latter converts the calendar time into a broken-down time expressed as UTC.

The function mktime takes a broken-down time, expressed as a local time, and converts it into a time_t value.

```
#include <time.h>

time_t mktime(struct tm *tmptr);
```

Returns: calendar time if OK, –1 on error

The strftime function is a printf-like function for time values. It is complicated by the multitude of arguments available to customize the string it produces.

```
#include <time.h>

size_t strftime(char *restrict buf, size_t maxsize,
                const char *restrict format,
                const struct tm *restrict tmptr);

size_t strftime_l(char *restrict buf, size_t maxsize,
                const char *restrict format,
                const struct tm *restrict tmptr, locale_t locale);
```

Both return: number of characters stored in array if room, 0 otherwise

Two older functions, asctime and ctime, can be used to produce a 26-byte printable string similar to the default output of the date(1) command. However, these functions are now marked obsolescent, because they are susceptible to buffer overflow problems.

The strftime and strftime_l functions are the same, except that the strftime_l function allows the caller to specify the locale as an argument. The strftime function uses the locale specified by the TZ environment variable.

The *tmptr* argument is the time value to format, specified by a pointer to a broken-down time value. The formatted result is stored in the array *buf* whose size is *maxsize* characters. If the size of the result, including the terminating null, fits in the buffer, these functions return the number of characters stored in *buf*, excluding the terminating null. Otherwise, these functions return 0.

The *format* argument controls the formatting of the time value. Like the printf functions, conversion specifiers are given as a percent sign followed by a special character. All other characters in the *format* string are copied to the output. Two percent signs in a row generate a single percent sign in the output. Unlike the printf

functions, each conversion specified generates a different fixed-size output string—there are no field widths in the *format* string. Figure 6.10 describes the 37 ISO C conversion specifiers.

Format	Description	Example
`%a`	abbreviated weekday name	`Thu`
`%A`	full weekday name	`Thursday`
`%b`	abbreviated month name	`Jan`
`%B`	full month name	`January`
`%c`	date and time	`Thu Jan 19 21:24:52 2012`
`%C`	year/100: [00–99]	`20`
`%d`	day of the month: [01–31]	`19`
`%D`	date [MM/DD/YY]	`01/19/12`
`%e`	day of month (single digit preceded by space) [1–31]	`19`
`%F`	ISO 8601 date format [YYYY–MM–DD]	`2012-01-19`
`%g`	last two digits of ISO 8601 week-based year [00–99]	`12`
`%G`	ISO 8601 week-based year	`2012`
`%h`	same as `%b`	`Jan`
`%H`	hour of the day (24-hour format): [00–23]	`21`
`%I`	hour of the day (12-hour format): [01–12]	`09`
`%j`	day of the year: [001–366]	`019`
`%m`	month: [01–12]	`01`
`%M`	minute: [00–59]	`24`
`%n`	newline character	
`%p`	AM/PM	`PM`
`%r`	locale's time (12-hour format)	`09:24:52 PM`
`%R`	same as `%H:%M`	`21:24`
`%S`	second: [00–60]	`52`
`%t`	horizontal tab character	
`%T`	same as `%H:%M:%S`	`21:24:52`
`%u`	ISO 8601 weekday [Monday = 1, 1–7]	`4`
`%U`	Sunday week number: [00–53]	`03`
`%V`	ISO 8601 week number: [01–53]	`03`
`%w`	weekday: [0 = Sunday, 0–6]	`4`
`%W`	Monday week number: [00–53]	`03`
`%x`	locale's date	`01/19/12`
`%X`	locale's time	`21:24:52`
`%y`	last two digits of year: [00–99]	`12`
`%Y`	year	`2012`
`%z`	offset from UTC in ISO 8601 format	`-0500`
`%Z`	time zone name	`EST`
`%%`	translates to a percent sign	`%`

Figure 6.10 Conversion specifiers for `strftime`

The third column of this figure is from the output of `strftime` under Mac OS X, corresponding to the time and date `Thu Jan 19 21:24:52 EST 2012`.

The only specifiers that are not self-evident are `%U`, `%V`, and `%W`. The `%U` specifier represents the week number of the year, where the week containing the first Sunday is week 1. The `%W` specifier represents the week number of the year, where the week containing the first Monday is week 1. The `%V` specifier is different. If the week

containing the first day in January has four or more days in the new year, then this is treated as week 1. Otherwise, it is treated as the last week of the previous year. In both cases, Monday is treated as the first day of the week.

As with `printf`, `strftime` supports modifiers for some of the conversion specifiers. The `E` and `O` modifiers can be used to generate an alternative format if one is supported by the locale.

Some systems support additional, nonstandard extensions to the *format* string for `strftime`.

Example

Figure 6.11 shows how to use several of the time functions discussed in this chapter. In particular, it shows how `strftime` can be used to print a string containing the current date and time.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int
main(void)
{
    time_t t;
    struct tm *tmp;
    char buf1[16];
    char buf2[64];

    time(&t);
    tmp = localtime(&t);
    if (strftime(buf1, 16, "time and date: %r, %a %b %d, %Y", tmp) == 0)
        printf("buffer length 16 is too small\n");
    else
        printf("%s\n", buf1);
    if (strftime(buf2, 64, "time and date: %r, %a %b %d, %Y", tmp) == 0)
        printf("buffer length 64 is too small\n");
    else
        printf("%s\n", buf2);
    exit(0);
}
```

Figure 6.11 Using the `strftime` function

Recall the relationship of the various time functions shown in Figure 6.9. Before we can print the time in a human-readable format, we need to get the time and convert it into a broken-down time structure. Sample output from Figure 6.11 is

```
$ ./a.out
buffer length 16 is too small
time and date: 11:12:35 PM, Thu Jan 19, 2012
```

□

The `strptime` function is the inverse of `strftime`. It takes a string and converts it into a broken-down time.

```
#include <time.h>

char *strptime(const char *restrict buf, const char *restrict format,
               struct tm *restrict tmptr);
```

Returns: pointer to one character past last character parsed, `NULL` otherwise

The *format* argument describes the format of the string in the buffer pointed to by the *buf* argument. The format specification is similar, although it differs slightly from the specification for the `strftime` function. The conversion specifiers for the `strptime` function are summarized in Figure 6.12.

Format	Description
`%a`	abbreviated or full weekday name
`%A`	same as `%a`
`%b`	abbreviated or full month name
`%B`	same as `%b`
`%c`	date and time
`%C`	all but the last two digits of the year
`%d`	day of the month: [01−31]
`%D`	date [MM/DD/YY]
`%e`	same as `%d`
`%h`	same as `%b`
`%H`	hour of the day (24-hour format): [00−23]
`%I`	hour of the day (12-hour format): [01−12]
`%j`	day of the year: [001−366]
`%m`	month: [01−12]
`%M`	minute: [00−59]
`%n`	any white space
`%p`	AM/PM
`%r`	locale's time (12-hour format, AM/PM notation)
`%R`	time as `%H:%M`
`%S`	second: [00−60]
`%t`	any white space
`%T`	time as `%H:%M:%S`
`%U`	Sunday week number: [00−53]
`%w`	weekday: [0 = Sunday, 0−6]
`%W`	Monday week number: [00−53]
`%x`	locale's date
`%X`	locale's time
`%y`	last two digits of year: [00−99]
`%Y`	year
`%%`	translates to a percent sign

Figure 6.12 Conversion specifiers for `strptime`

We mentioned that the three functions in Figure 6.9 with dashed lines were affected by the `TZ` environment variable: `localtime`, `mktime`, and `strftime`. If defined, the value of this environment variable is used by these functions instead of the default time

zone. If the variable is defined to be a null string, such as `TZ=`, then UTC is normally used. The value of this environment variable is often something like `TZ=EST5EDT`, but POSIX.1 allows a much more detailed specification. Refer to the Environment Variables chapter of the Single UNIX Specification [Open Group 2010] for all the details on the `TZ` variable.

More information on the `TZ` environment variable can be found in the `tzset`(3) manual page.

6.11 Summary

The password file and the group file are used on all UNIX systems. We've looked at the various functions that read these files. We've also talked about shadow passwords, which can enhance system security. Supplementary group IDs provide a way to participate in multiple groups at the same time. We also looked at how similar functions are provided by most systems to access other system-related data files. We discussed the POSIX.1 functions that programs can use to identify the system on which they are running. We finished the chapter by looking at the time and date functions provided by ISO C and the Single UNIX Specification.

Exercises

6.1 If the system uses a shadow file and we need to obtain the encrypted password, how do we do so?

6.2 If you have superuser access and your system uses shadow passwords, implement the previous exercise.

6.3 Write a program that calls `uname` and prints all the fields in the `utsname` structure. Compare the output to the output from the `uname`(1) command.

6.4 Calculate the latest time that can be represented by the `time_t` data type. After it wraps around, what happens?

6.5 Write a program to obtain the current time and print it using `strftime`, so that it looks like the default output from `date`(1). Set the `TZ` environment variable to different values and see what happens.

7

Process Environment

7.1 Introduction

Before looking at the process control primitives in the next chapter, we need to examine the environment of a single process. In this chapter, we'll see how the `main` function is called when the program is executed, how command-line arguments are passed to the new program, what the typical memory layout looks like, how to allocate additional memory, how the process can use environment variables, and various ways for the process to terminate. Additionally, we'll look at the `longjmp` and `setjmp` functions and their interaction with the stack. We finish the chapter by examining the resource limits of a process.

7.2 `main` Function

A C program starts execution with a function called `main`. The prototype for the `main` function is

```
int main(int argc, char *argv[]);
```

where *argc* is the number of command-line arguments, and *argv* is an array of pointers to the arguments. We describe these arguments in Section 7.4.

When a C program is executed by the kernel—by one of the `exec` functions, which we describe in Section 8.10—a special start-up routine is called before the `main` function is called. The executable program file specifies this routine as the starting address for the program; this is set up by the link editor when it is invoked by the C compiler. This start-up routine takes values from the kernel—the command-line arguments and the environment—and sets things up so that the `main` function is called as shown earlier.

7.3 Process Termination

There are eight ways for a process to terminate. Normal termination occurs in five ways:

1. Return from `main`
2. Calling `exit`
3. Calling `_exit` or `_Exit`
4. Return of the last thread from its start routine (Section 11.5)
5. Calling `pthread_exit` (Section 11.5) from the last thread

Abnormal termination occurs in three ways:

6. Calling `abort` (Section 10.17)
7. Receipt of a signal (Section 10.2)
8. Response of the last thread to a cancellation request (Sections 11.5 and 12.7)

> For now, we'll ignore the three termination methods specific to threads until we discuss threads in Chapters 11 and 12.

The start-up routine that we mentioned in the previous section is also written so that if the `main` function returns, the `exit` function is called. If the start-up routine were coded in C (it is often coded in assembly language) the call to `main` could look like

```
exit(main(argc, argv));
```

Exit Functions

Three functions terminate a program normally: `_exit` and `_Exit`, which return to the kernel immediately, and `exit`, which performs certain cleanup processing and then returns to the kernel.

```
#include <stdlib.h>

void exit(int status);

void _Exit(int status);

#include <unistd.h>

void _exit(int status);
```

We'll discuss the effect of these three functions on other processes, such as the children and the parent of the terminating process, in Section 8.5.

> The reason for the different headers is that `exit` and `_Exit` are specified by ISO C, whereas `_exit` is specified by POSIX.1.

Historically, the `exit` function has always performed a clean shutdown of the standard I/O library: the `fclose` function is called for all open streams. Recall from Section 5.5 that this causes all buffered output data to be flushed (written to the file).

All three exit functions expect a single integer argument, which we call the *exit status*. Most UNIX System shells provide a way to examine the exit status of a process. If (a) any of these functions is called without an exit status, (b) `main` does a `return` without a return value, or (c) the `main` function is not declared to return an integer, the exit status of the process is undefined. However, if the return type of `main` is an integer and `main` "falls off the end" (an implicit return), the exit status of the process is 0.

> This behavior is new with the 1999 version of the ISO C standard. Historically, the exit status was undefined if the end of the `main` function was reached without an explicit `return` statement or a call to the `exit` function.

Returning an integer value from the `main` function is equivalent to calling `exit` with the same value. Thus

```
exit(0);
```

is the same as

```
return(0);
```

from the `main` function.

Example

The program in Figure 7.1 is the classic "hello, world" example.

```
#include    <stdio.h>

main()
{
    printf("hello, world\n");
}
```

Figure 7.1 Classic C program

When we compile and run the program in Figure 7.1, we see that the exit code is random. If we compile the same program on different systems, we are likely to get different exit codes, depending on the contents of the stack and register contents at the time that the `main` function returns:

```
$ gcc hello.c
$ ./a.out
hello, world
$ echo $?                       print the exit status
13
```

Now if we enable the 1999 ISO C compiler extensions, we see that the exit code changes:

```
$ gcc -std=c99 hello.c          enable gcc's 1999 ISO C extensions
hello.c:4: warning: return type defaults to 'int'
$ ./a.out
hello, world
$ echo $?                       print the exit status
0
```

Note the compiler warning when we enable the 1999 ISO C extensions. This warning is printed because the type of the `main` function is not explicitly declared to be an integer. If we were to add this declaration, the message would go away. However, if we were to enable all recommended warnings from the compiler (with the `-Wall` flag), then we would see a warning message something like "control reaches end of nonvoid function."

The declaration of `main` as returning an integer and the use of `exit` instead of `return` produces needless warnings from some compilers and the `lint`(1) program. The problem is that these compilers don't know that an `exit` from `main` is the same as a `return`. One way around these warnings, which become annoying after a while, is to use `return` instead of `exit` from `main`. But doing this prevents us from using the UNIX System's `grep` utility to locate all calls to `exit` from a program. Another solution is to declare `main` as returning `void`, instead of `int`, and continue calling `exit`. This gets rid of the compiler warning but doesn't look right (especially in a programming text), and can generate other compiler warnings, since the return type of `main` is supposed to be a signed integer. In this text, we show `main` as returning an integer, since that is the definition specified by both ISO C and POSIX.1.

Different compilers vary in the verbosity of their warnings. Note that the GNU C compiler usually doesn't emit these extraneous compiler warnings unless additional warning options are used.

□

In the next chapter, we'll see how any process can cause a program to be executed, wait for the process to complete, and then fetch its exit status.

`atexit` Function

With ISO C, a process can register at least 32 functions that are automatically called by `exit`. These are called *exit handlers* and are registered by calling the `atexit` function.

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

Returns: 0 if OK, nonzero on error

This declaration says that we pass the address of a function as the argument to `atexit`. When this function is called, it is not passed any arguments and is not expected to return a value. The `exit` function calls these functions in reverse order of their registration. Each function is called as many times as it was registered.

These exit handlers first appeared in the ANSI C Standard in 1989. Systems that predate ANSI C, such as SVR3 and 4.3BSD, did not provide these exit handlers.

ISO C requires that systems support at least 32 exit handlers, but implementations often support more (see Figure 2.15). The `sysconf` function can be used to determine the maximum number of exit handlers supported by a given platform, as illustrated in Figure 2.14.

With ISO C and POSIX.1, `exit` first calls the exit handlers and then closes (via `fclose`) all open streams. POSIX.1 extends the ISO C standard by specifying that any exit handlers installed will be cleared if the program calls any of the `exec` family of functions. Figure 7.2 summarizes how a C program is started and the various ways it can terminate.

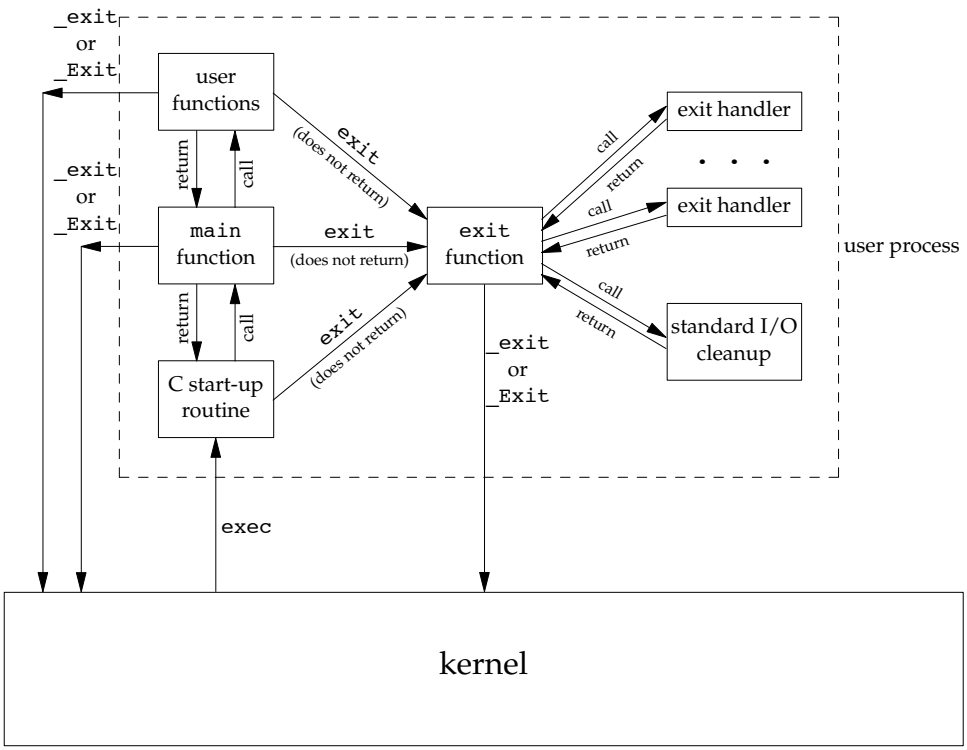


Figure 7.2 How a C program is started and how it terminates

The only way a program can be executed by the kernel is if one of the `exec` functions is called. The only way a process can voluntarily terminate is if `_exit` or `_Exit` is called, either explicitly or implicitly (by calling `exit`). A process can also be involuntarily terminated by a signal (not shown in Figure 7.2).

Example

The program in Figure 7.3 demonstrates the use of the `atexit` function.

```
#include "apue.h"

static void my_exit1(void);
static void my_exit2(void);

int
main(void)
{
    if (atexit(my_exit2) != 0)
        err_sys("can't register my_exit2");

    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");
    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");

    printf("main is done\n");
    return(0);
}

static void
my_exit1(void)
{
    printf("first exit handler\n");
}

static void
my_exit2(void)
{
    printf("second exit handler\n");
}
```

Figure 7.3 Example of exit handlers

Executing the program in Figure 7.3 yields

```
$ ./a.out
main is done
first exit handler
first exit handler
second exit handler
```

An exit handler is called once for each time it is registered. In Figure 7.3, the first exit handler is registered twice, so it is called two times. Note that we don't call `exit`; instead, we return from `main`. □

7.4 Command-Line Arguments

When a program is executed, the process that does the `exec` can pass command-line arguments to the new program. This is part of the normal operation of the UNIX system shells. We have already seen this in many of the examples from earlier chapters.

Example

The program in Figure 7.4 echoes all its command-line arguments to standard output. Note that the normal `echo`(1) program doesn't echo the zeroth argument.

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int     i;

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}
```

Figure 7.4 Echo all command-line arguments to standard output

If we compile this program and name the executable `echoarg`, we have

```
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

We are guaranteed by both ISO C and POSIX.1 that `argv[argc]` is a null pointer. This lets us alternatively code the argument-processing loop as

```
for (i = 0; argv[i] != NULL; i++)
```

□

7.5 Environment List

Each program is also passed an *environment list*. Like the argument list, the environment list is an array of character pointers, with each pointer containing the address of a null-terminated C string. The address of the array of pointers is contained in the global variable `environ`:

```
extern char **environ;
```

For example, if the environment consisted of five strings, it could look like Figure 7.5. Here we explicitly show the null bytes at the end of each string. We'll call `environ` the

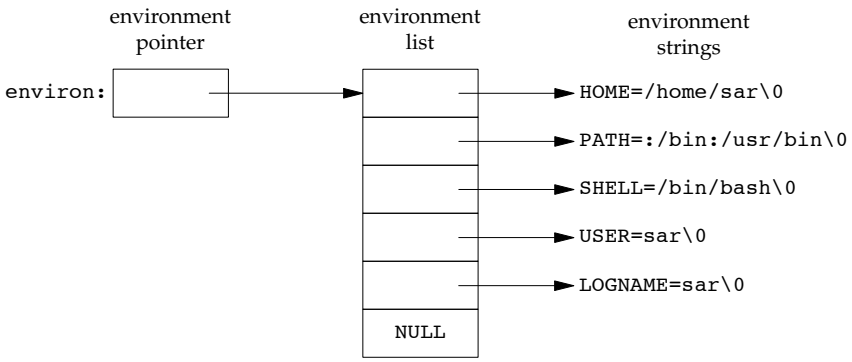


Figure 7.5 Environment consisting of five C character strings

environment pointer, the array of pointers the environment list, and the strings they point to the *environment strings*.

By convention, the environment consists of

name=value

strings, as shown in Figure 7.5. Most predefined names are entirely uppercase, but this is only a convention.

Historically, most UNIX systems have provided a third argument to the `main` function that is the address of the environment list:

```
int main(int argc, char *argv[], char *envp[]);
```

Because ISO C specifies that the `main` function be written with two arguments, and because this third argument provides no benefit over the global variable `environ`, POSIX.1 specifies that `environ` should be used instead of the (possible) third argument. Access to specific environment variables is normally through the `getenv` and `putenv` functions, described in Section 7.9, instead of through the `environ` variable. But to go through the entire environment, the `environ` pointer must be used.

7.6 Memory Layout of a C Program

Historically, a C program has been composed of the following pieces:

- Text segment, consisting of the machine instructions that the CPU executes. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

- Initialized data segment, usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration

```
int   maxcount = 99;
```

  appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.

- Uninitialized data segment, often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing. The C declaration

```
long   sum[1000];
```

  appearing outside any function causes this variable to be stored in the uninitialized data segment.

- Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.

- Heap, where dynamic memory allocation usually takes place. Historically, the heap has been located between the uninitialized data and the stack.

Figure 7.6 shows the typical arrangement of these segments. This is a logical picture of how a program looks; there is no requirement that a given implementation arrange its memory in this fashion. Nevertheless, this gives us a typical arrangement to describe. With Linux on a 32-bit Intel x86 processor, the text segment starts at location `0x08048000`, and the bottom of the stack starts just below `0xC0000000`. (The stack grows from higher-numbered addresses to lower-numbered addresses on this particular architecture.) The unused virtual address space between the top of the heap and the top of the stack is large.

> Several more segment types exist in an `a.out`, containing the symbol table, debugging information, linkage tables for dynamic shared libraries, and the like. These additional sections don't get loaded as part of the program's image executed by a process.

Note from Figure 7.6 that the contents of the uninitialized data segment are not stored in the program file on disk, because the kernel sets the contents to 0 before the program starts running. The only portions of the program that need to be saved in the program file are the text segment and the initialized data.

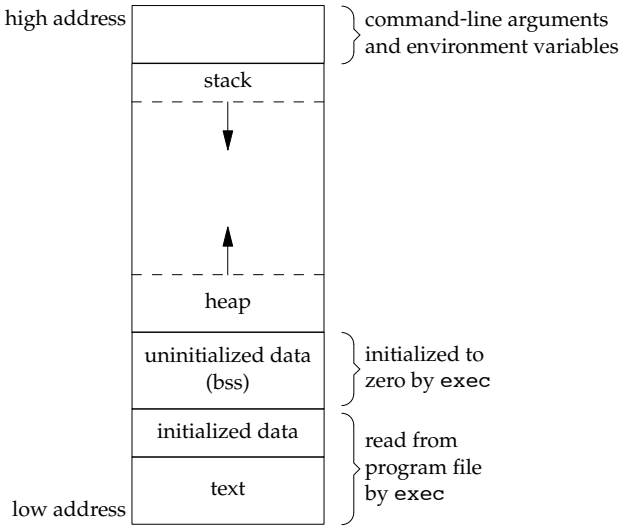


Figure 7.6 Typical memory arrangement

The `size`(1) command reports the sizes (in bytes) of the text, data, and bss segments. For example:

```
$ size /usr/bin/cc /bin/sh
   text    data     bss     dec     hex  filename
 346919    3576    6680  357175  57337  /usr/bin/cc
 102134    1776   11272  115182  1c1ee  /bin/sh
```

The fourth and fifth columns are the total of the three sizes, displayed in decimal and hexadecimal, respectively.

7.7 Shared Libraries

Most UNIX systems today support shared libraries. Arnold [1986] describes an early implementation under System V, and Gingell et al. [1987] describe a different implementation under SunOS. Shared libraries remove the common library routines from the executable file, instead maintaining a single copy of the library routine somewhere in memory that all processes reference. This reduces the size of each executable file but may add some runtime overhead, either when the program is first executed or the first time each shared library function is called. Another advantage of shared libraries is that library functions can be replaced with new versions without having to relink edit every program that uses the library (assuming that the number and type of arguments haven't changed).

Different systems provide different ways for a program to say that it wants to use or not use the shared libraries. Options for the `cc`(1) and `ld`(1) commands are typical. As

an example of the size differences, the following executable file—the classic `hello.c` program—was first created without shared libraries:

```
$ gcc -static hello1.c          prevent gcc from using shared libraries
$ ls -l a.out
-rwxr-xr-x  1 sar     879443 Sep 2 10:39 a.out
$ size a.out
   text    data    bss     dec     hex  filename
 787775    6128  11272  805175   c4937  a.out
```

If we compile this program to use shared libraries, the text and data sizes of the executable file are greatly decreased:

```
$ gcc hello1.c                  gcc defaults to use shared libraries
$ ls -l a.out
-rwxr-xr-x  1 sar       8378 Sep 2 10:39 a.out
$ size a.out
   text    data    bss     dec     hex  filename
   1176     504     16    1696     6a0  a.out
```

7.8 Memory Allocation

ISO C specifies three functions for memory allocation:

1. `malloc`, which allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
2. `calloc`, which allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.
3. `realloc`, which increases or decreases the size of a previously allocated area. When the size increases, it may involve moving the previously allocated area somewhere else, to provide the additional room at the end. Also, when the size increases, the initial value of the space between the old contents and the end of the new area is indeterminate.

```
#include <stdlib.h>

void *malloc(size_t size);

void *calloc(size_t nobj, size_t size);

void *realloc(void *ptr, size_t newsize);

                                All three return: non-null pointer if OK, NULL on error

void free(void *ptr);
```

The pointer returned by the three allocation functions is guaranteed to be suitably aligned so that it can be used for any data object. For example, if the most restrictive alignment requirement on a particular system requires that `doubles` must start at memory locations that are multiples of 8, then all pointers returned by these three functions would be so aligned.

Because the three `alloc` functions return a generic `void *` pointer, if we `#include <stdlib.h>` (to obtain the function prototypes), we do not explicitly have to cast the pointer returned by these functions when we assign it to a pointer of a different type. The default return value for undeclared functions is `int`, so using a cast without the proper function declaration could hide an error on systems where the size of type `int` differs from the size of a function's return value (a pointer in this case).

The function `free` causes the space pointed to by *ptr* to be deallocated. This freed space is usually put into a pool of available memory and can be allocated in a later call to one of the three `alloc` functions.

The `realloc` function lets us change the size of a previously allocated area. (The most common usage is to increase an area's size.) For example, if we allocate room for 512 elements in an array that we fill in at runtime but later find that we need more room, we can call `realloc`. If there is room beyond the end of the existing region for the requested space, then `realloc` simply allocates this additional area at the end and returns the same pointer that we passed it. But if there isn't room, `realloc` allocates another area that is large enough, copies the existing 512-element array to the new area, frees the old area, and returns the pointer to the new area. Because the area may move, we shouldn't have any pointers into this area. Exercise 4.16 and Figure C.3 show the use of `realloc` with `getcwd` to handle a pathname of any length. Figure 17.27 shows an example that uses `realloc` to avoid arrays with fixed, compile-time sizes.

Note that the final argument to `realloc` is the new size of the region, not the difference between the old and new sizes. As a special case, if *ptr* is a null pointer, `realloc` behaves like `malloc` and allocates a region of the specified *newsize*.

> Older versions of these routines allowed us to `realloc` a block that we had `free`d since the last call to `malloc`, `realloc`, or `calloc`. This trick dates back to Version 7 and exploited the search strategy of `malloc` to perform storage compaction. Solaris still supports this feature, but many other platforms do not. This feature is deprecated and should not be used.

The allocation routines are usually implemented with the `sbrk`(2) system call. This system call expands (or contracts) the heap of the process. (Refer to Figure 7.6.) A sample implementation of `malloc` and `free` is given in Section 8.7 of Kernighan and Ritchie [1988].

Although `sbrk` can expand or contract the memory of a process, most versions of `malloc` and `free` never decrease their memory size. The space that we free is available for a later allocation, but the freed space is not usually returned to the kernel; instead, that space is kept in the `malloc` pool.

Most implementations allocate more space than requested and use the additional space for record keeping—the size of the block, a pointer to the next allocated block, and the like. As a consequence, writing past the end or before the start of an allocated area could overwrite this record-keeping information in another block. These types of errors are often catastrophic, but difficult to find, because the error may not show up until much later.

Writing past the end or before the beginning of a dynamically allocated buffer can corrupt more than internal record-keeping information. The memory before and after a dynamically allocated buffer can potentially be used for other dynamically allocated

objects. These objects can be unrelated to the code corrupting them, making it even more difficult to find the source of the corruption.

Other possible errors that can be fatal are freeing a block that was already freed and calling `free` with a pointer that was not obtained from one of the three `alloc` functions. If a process calls `malloc` but forgets to call `free`, its memory usage will continually increase; this is called leakage. If we do not call `free` to return unused space, the size of a process's address space will slowly increase until no free space is left. During this time, performance can degrade from excess paging overhead.

Because memory allocation errors are difficult to track down, some systems provide versions of these functions that do additional error checking every time one of the three `alloc` functions or `free` is called. These versions of the functions are often specified by including a special library for the link editor. There are also publicly available sources that you can compile with special flags to enable additional runtime checking.

> FreeBSD, Mac OS X, and Linux support additional debugging through the setting of environment variables. In addition, options can be passed to the FreeBSD library through the symbolic link `/etc/malloc.conf`.

Alternate Memory Allocators

Many replacements for `malloc` and `free` are available. Some systems already include libraries providing alternative memory allocator implementations. Other systems provide only the standard allocator, leaving it up to software developers to download alternatives, if desired. We discuss some of the alternatives here.

`libmalloc`

SVR4-based systems, such as Solaris, include the `libmalloc` library, which provides a set of interfaces matching the ISO C memory allocation functions. The `libmalloc` library includes `mallopt`, a function that allows a process to set certain variables that control the operation of the storage allocator. A function called `mallinfo` is also available to provide statistics on the memory allocator.

`vmalloc`

Vo [1996] describes a memory allocator that allows processes to allocate memory using different techniques for different regions of memory. In addition to the functions specific to `vmalloc`, the library provides emulations of the ISO C memory allocation functions.

`quick-fit`

Historically, the standard `malloc` algorithm used either a best-fit or a first-fit memory allocation strategy. Quick-fit is faster than either, but tends to use more memory. Weinstock and Wulf [1988] describe the algorithm, which is based on splitting up memory into buffers of various sizes and maintaining unused buffers on different free lists, depending on the buffer sizes. Most modern allocators are based on quick-fit.

jemalloc

The jemalloc implementation of the malloc family of library functions is the default memory allocator in FreeBSD 8.0. It was designed to scale well when used with multithreaded applications running on multiprocessor systems. Evans [2006] describes the implementation and evaluates its performance.

TCMalloc

TCMalloc was designed as a replacement for the malloc family of functions to provide high performance, scalability, and memory efficiency. It uses thread-local caches to avoid locking overhead when allocating buffers from and releasing buffers to the cache. It also has a heap checker and a heap profiler built in to aid in debugging and analyzing dynamic memory usage. The TCMalloc library is available as open source from Google. It is briefly described by Ghemawat and Menage [2005].

alloca Function

One additional function is also worth mentioning. The function alloca has the same calling sequence as malloc; however, instead of allocating memory from the heap, the memory is allocated from the stack frame of the current function. The advantage is that we don't have to free the space; it goes away automatically when the function returns. The alloca function increases the size of the stack frame. The disadvantage is that some systems can't support alloca, if it's impossible to increase the size of the stack frame after the function has been called. Nevertheless, many software packages use it, and implementations exist for a wide variety of systems.

All four platforms discussed in this text provide the alloca function.

7.9 Environment Variables

As we mentioned earlier, the environment strings are usually of the form

name=*value*

The UNIX kernel never looks at these strings; their interpretation is up to the various applications. The shells, for example, use numerous environment variables. Some, such as HOME and USER, are set automatically at login; others are left for us to set. We normally set environment variables in a shell start-up file to control the shell's actions. If we set the environment variable MAILPATH, for example, it tells the Bourne shell, GNU Bourne-again shell, and Korn shell where to look for mail.

ISO C defines a function that we can use to fetch values from the environment, but this standard says that the contents of the environment are implementation defined.

```
#include <stdlib.h>

char *getenv(const char *name);
```

Returns: pointer to *value* associated with *name*, NULL if not found

Note that this function returns a pointer to the *value* of a *name=value* string. We should always use `getenv` to fetch a specific value from the environment, instead of accessing `environ` directly.

Some environment variables are defined by POSIX.1 in the Single UNIX Specification, whereas others are defined only if the XSI option is supported. Figure 7.7 lists the environment variables defined by the Single UNIX Specification and notes which implementations support the variables. Any environment variable defined by POSIX.1 is marked with •; otherwise, it is part of the XSI option. Many additional implementation-dependent environment variables are used in the four implementations described in this book. Note that ISO C doesn't define any environment variables.

Variable	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
COLUMNS	•	•	•	•	•	terminal width
DATEMSK	XSI		•	•	•	`getdate`(3) template file pathname
HOME	•	•	•	•	•	home directory
LANG	•	•	•	•	•	name of locale
LC_ALL	•	•	•	•	•	name of locale
LC_COLLATE	•	•	•	•	•	name of locale for collation
LC_CTYPE	•	•	•	•	•	name of locale for character classification
LC_MESSAGES	•	•	•	•	•	name of locale for messages
LC_MONETARY	•	•	•	•	•	name of locale for monetary editing
LC_NUMERIC	•	•	•	•	•	name of locale for numeric editing
LC_TIME	•	•	•	•	•	name of locale for date/time formatting
LINES	•	•	•	•	•	terminal height
LOGNAME	•	•	•	•	•	login name
MSGVERB	XSI	•	•	•	•	`fmtmsg`(3) message components to process
NLSPATH	•	•	•	•	•	sequence of templates for message catalogs
PATH	•	•	•	•	•	list of path prefixes to search for executable file
PWD	•	•	•	•	•	absolute pathname of current working directory
SHELL	•	•	•	•	•	name of user's preferred shell
TERM	•	•	•	•	•	terminal type
TMPDIR	•	•	•	•	•	pathname of directory for creating temporary files
TZ	•	•	•	•	•	time zone information

Figure 7.7 Environment variables defined in the Single UNIX Specification

In addition to fetching the value of an environment variable, sometimes we may want to set an environment variable. We may want to change the value of an existing variable or add a new variable to the environment. (In the next chapter, we'll see that we can affect the environment of only the current process and any child processes that we invoke. We cannot affect the environment of the parent process, which is often a shell. Nevertheless, it is still useful to be able to modify the environment list.) Unfortunately, not all systems support this capability. Figure 7.8 shows the functions that are supported by the various standards and implementations.

Function	ISO C	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
getenv	•	•	•	•	•	•
putenv		XSI	•	•	•	•
setenv		•	•	•	•	
unsetenv		•	•	•	•	
clearenv				•		

Figure 7.8 Support for various environment list functions

The `clearenv` function is not part of the Single UNIX Specification. It is used to remove all entries from the environment list.

The prototypes for the middle three functions listed in Figure 7.8 are

```
#include <stdlib.h>

int putenv(char *str);
```

Returns: 0 if OK, nonzero on error

```
int setenv(const char *name, const char *value, int rewrite);

int unsetenv(const char *name);
```

Both return: 0 if OK, –1 on error

The operation of these three functions is as follows:

- The `putenv` function takes a string of the form *name=value* and places it in the environment list. If *name* already exists, its old definition is first removed.
- The `setenv` function sets *name* to *value*. If *name* already exists in the environment, then (a) if *rewrite* is nonzero, the existing definition for *name* is first removed; or (b) if *rewrite* is 0, an existing definition for *name* is not removed, *name* is not set to the new *value*, and no error occurs.
- The `unsetenv` function removes any definition of *name*. It is not an error if such a definition does not exist.

Note the difference between `putenv` and `setenv`. Whereas `setenv` must allocate memory to create the *name=value* string from its arguments, `putenv` is free to place the string passed to it directly into the environment. Indeed, many implementations do exactly this, so it would be an error to pass `putenv` a string allocated on the stack, since the memory would be reused after we return from the current function.

It is interesting to examine how these functions must operate when modifying the environment list. Recall Figure 7.6: the environment list—the array of pointers to the actual *name=value* strings—and the environment strings are typically stored at the top of a process's memory space, above the stack. Deleting a string is simple; we just find the pointer in the environment list and move all subsequent pointers down one. But adding a string or modifying an existing string is more difficult. The space at the top of the stack cannot be expanded, because it is often at the top of the address space of the

process and so can't expand upward; it can't be expanded downward, because all the stack frames below it can't be moved.

1. If we're modifying an existing *name:*
   a. If the size of the new *value* is less than or equal to the size of the existing *value*, we can just copy the new string over the old string.
   b. If the size of the new *value* is larger than the old one, however, we must `malloc` to obtain room for the new string, copy the new string to this area, and then replace the old pointer in the environment list for *name* with the pointer to this allocated area.
2. If we're adding a new *name*, it's more complicated. First, we have to call `malloc` to allocate room for the *name=value* string and copy the string to this area.
   a. Then, if it's the first time we've added a new *name*, we have to call `malloc` to obtain room for a new list of pointers. We copy the old environment list to this new area and store a pointer to the *name=value* string at the end of this list of pointers. We also store a null pointer at the end of this list, of course. Finally, we set `environ` to point to this new list of pointers. Note from Figure 7.6 that if the original environment list was contained above the top of the stack, as is common, then we have moved this list of pointers to the heap. But most of the pointers in this list still point to *name=value* strings above the top of the stack.
   b. If this isn't the first time we've added new strings to the environment list, then we know that we've already allocated room for the list on the heap, so we just call `realloc` to allocate room for one more pointer. The pointer to the new *name=value* string is stored at the end of the list (on top of the previous null pointer), followed by a null pointer.

7.10 `setjmp` and `longjmp` Functions

In C, we can't `goto` a label that's in another function. Instead, we must use the `setjmp` and `longjmp` functions to perform this type of branching. As we'll see, these two functions are useful for handling error conditions that occur in a deeply nested function call.

Consider the skeleton in Figure 7.9. It consists of a main loop that reads lines from standard input and calls the function `do_line` to process each line. This function then calls `get_token` to fetch the next token from the input line. The first token of a line is assumed to be a command of some form, and a `switch` statement selects each command. For the single command shown, the function `cmd_add` is called.

The skeleton in Figure 7.9 is typical for programs that read commands, determine the command type, and then call functions to process each command. Figure 7.10 shows what the stack could look like after `cmd_add` has been called.

```
#include "apue.h"

#define TOK_ADD    5

void    do_line(char *);
void    cmd_add(void);
int     get_token(void);

int
main(void)
{
    char    line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

char    *tok_ptr;       /* global pointer for get_token() */

void
do_line(char *ptr)      /* process one line of input */
{
    int     cmd;

    tok_ptr = ptr;
    while ((cmd = get_token()) > 0) {
        switch (cmd) {  /* one case for each command */
        case TOK_ADD:
                cmd_add();
                break;
        }
    }
}

void
cmd_add(void)
{
    int     token;

    token = get_token();
    /* rest of processing for this command */
}

int
get_token(void)
{
    /* fetch next token from line pointed to by tok_ptr */
}
```

Figure 7.9 Typical program skeleton for command processing

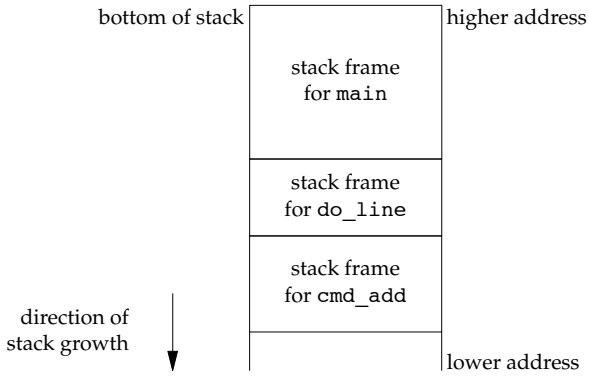


Figure 7.10 Stack frames after `cmd_add` has been called

Storage for the automatic variables is within the stack frame for each function. The array `line` is in the stack frame for `main`, the integer `cmd` is in the stack frame for `do_line`, and the integer `token` is in the stack frame for `cmd_add`.

As we've said, this type of arrangement of the stack is typical, but not required. Stacks do not have to grow toward lower memory addresses. On systems that don't have built-in hardware support for stacks, a C implementation might use a linked list for its stack frames.

The coding problem that's often encountered with programs like the one shown in Figure 7.9 is how to handle nonfatal errors. For example, if the `cmd_add` function encounters an error—say, an invalid number—it might want to print an error message, ignore the rest of the input line, and return to the `main` function to read the next input line. But when we're deeply nested numerous levels down from the `main` function, this is difficult to do in C. (In this example, the `cmd_add` function is only two levels down from `main`, but it's not uncommon to be five or more levels down from the point to which we want to return.) It becomes messy if we have to code each function with a special return value that tells it to return one level.

The solution to this problem is to use a nonlocal `goto`: the `setjmp` and `longjmp` functions. The adjective "nonlocal" indicates that we're not doing a normal C `goto` statement within a function; instead, we're branching back through the call frames to a function that is in the call path of the current function.

```
#include <setjmp.h>

int setjmp(jmp_buf env);

                Returns: 0 if called directly, nonzero if returning from a call to longjmp

void longjmp(jmp_buf env, int val);
```

We call setjmp from the location that we want to return to, which in this example is in the main function. In this case, setjmp returns 0 because we called it directly. In the call to setjmp, the argument *env* is of the special type jmp_buf. This data type is some form of array that is capable of holding all the information required to restore the status of the stack to the state when we call longjmp. Normally, the *env* variable is a global variable, since we'll need to reference it from another function.

When we encounter an error—say, in the cmd_add function—we call longjmp with two arguments. The first is the same *env* that we used in a call to setjmp, and the second, *val*, is a nonzero value that becomes the return value from setjmp. The second argument allows us to use more than one longjmp for each setjmp. For example, we could longjmp from cmd_add with a *val* of 1 and also call longjmp from get_token with a *val* of 2. In the main function, the return value from setjmp is either 1 or 2, and we can test this value, if we want, and determine whether the longjmp was from cmd_add or get_token.

Let's return to the example. Figure 7.11 shows both the main and cmd_add functions. (The other two functions, do_line and get_token, haven't changed.)

```
#include "apue.h"
#include <setjmp.h>

#define TOK_ADD    5

jmp_buf jmpbuffer;

int
main(void)
{
    char    line[MAXLINE];

    if (setjmp(jmpbuffer) != 0)
        printf("error");
    while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
    exit(0);
}

 . . .

void
cmd_add(void)
{
    int     token;

    token = get_token();
    if (token < 0)      /* an error has occurred */
        longjmp(jmpbuffer, 1);
    /* rest of processing for this command */
}
```

Figure 7.11 Example of setjmp and longjmp

When main is executed, we call setjmp, which records whatever information it needs to in the variable jmpbuffer and returns 0. We then call do_line, which calls cmd_add, and assume that an error of some form is detected. Before the call to longjmp in cmd_add, the stack looks like that in Figure 7.10. But longjmp causes the stack to be "unwound" back to the main function, throwing away the stack frames for cmd_add and do_line (Figure 7.12). Calling longjmp causes the setjmp in main to return, but this time it returns with a value of 1 (the second argument for longjmp).

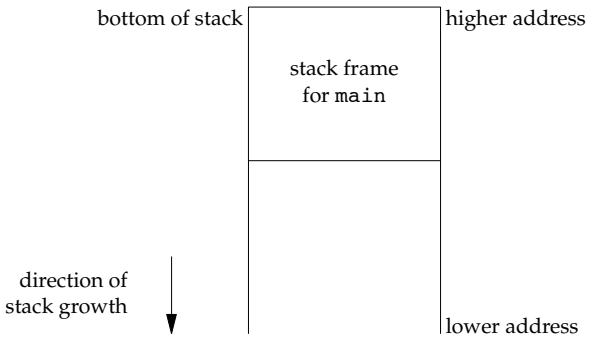


Figure 7.12 Stack frame after longjmp has been called

Automatic, Register, and Volatile Variables

We've seen what the stack looks like after calling longjmp. The next question is, "What are the states of the automatic variables and register variables in the main function?" When we return to main as a result of the longjmp, do these variables have values corresponding to those when the setjmp was previously called (i.e., are their values rolled back), or are their values left alone so that their values are whatever they were when do_line was called (which caused cmd_add to be called, which caused longjmp to be called)? Unfortunately, the answer is "It depends." Most implementations do not try to roll back these automatic variables and register variables, but the standards say only that their values are indeterminate. If you have an automatic variable that you don't want rolled back, define it with the volatile attribute. Variables that are declared as global or static are left alone when longjmp is executed.

Example

The program in Figure 7.13 demonstrates the different behavior that can be seen with automatic, global, register, static, and volatile variables after calling longjmp.

```
#include "apue.h"
#include <setjmp.h>

static void f1(int, int, int, int);
static void f2(void);

static jmp_buf  jmpbuffer;
static int      globval;

int
main(void)
{
    int             autoval;
    register int    regival;
    volatile int    volaval;
    static int      statval;

    globval = 1; autoval = 2; regival = 3; volaval = 4; statval = 5;

    if (setjmp(jmpbuffer) != 0) {
        printf("after longjmp:\n");
        printf("globval = %d, autoval = %d, regival = %d,"
            " volaval = %d, statval = %d\n",
            globval, autoval, regival, volaval, statval);
        exit(0);
    }

    /*
     * Change variables after setjmp, but before longjmp.
     */
    globval = 95; autoval = 96; regival = 97; volaval = 98;
    statval = 99;

    f1(autoval, regival, volaval, statval); /* never returns */
    exit(0);
}

static void
f1(int i, int j, int k, int l)
{
    printf("in f1():\n");
    printf("globval = %d, autoval = %d, regival = %d,"
        " volaval = %d, statval = %d\n", globval, i, j, k, l);
    f2();
}

static void
f2(void)
{
    longjmp(jmpbuffer, 1);
}
```

Figure 7.13 Effect of `longjmp` on various types of variables

If we compile and test the program in Figure 7.13, with and without compiler optimizations, the results are different:

```
$ gcc testjmp.c                compile without any optimization
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
$ gcc -O testjmp.c             compile with full optimization
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 2, regival = 3, volaval = 98, statval = 99
```

Note that the optimizations don't affect the global, static, and volatile variables; their values after the `longjmp` are the last values that they assumed. The `setjmp`(3) manual page on one system states that variables stored in memory will have values as of the time of the `longjmp`, whereas variables in the CPU and floating-point registers are restored to their values when `setjmp` was called. This is indeed what we see when we run the program in Figure 7.13. Without optimization, all five variables are stored in memory (the `register` hint is ignored for `regival`). When we enable optimization, both `autoval` and `regival` go into registers, even though the former wasn't declared `register`, and the `volatile` variable stays in memory. The important thing to realize with this example is that you must use the `volatile` attribute if you're writing portable code that uses nonlocal jumps. Anything else can change from one system to the next.

Some `printf` format strings in Figure 7.13 are longer than will fit comfortably for display in a programming text. Instead of making multiple calls to `printf`, we rely on ISO C's string concatenation feature, where the sequence

```
"string1" "string2"
```

is equivalent to

```
"string1string2"
```

□

We'll return to these two functions, `setjmp` and `longjmp`, in Chapter 10 when we discuss signal handlers and their signal versions: `sigsetjmp` and `siglongjmp`.

Potential Problem with Automatic Variables

Having looked at the way stack frames are usually handled, it is worth looking at a potential error in dealing with automatic variables. The basic rule is that an automatic variable can never be referenced after the function that declared it returns. Numerous warnings about this can be found throughout the UNIX System manuals.

Figure 7.14 shows a function called `open_data` that opens a standard I/O stream and sets the buffering for the stream.

```
#include    <stdio.h>

FILE *
open_data(void)
{
    FILE    *fp;
    char    databuf[BUFSIZ];  /* setvbuf makes this the stdio buffer */

    if ((fp = fopen("datafile", "r")) == NULL)
        return(NULL);
    if (setvbuf(fp, databuf, _IOLBF, BUFSIZ) != 0)
        return(NULL);
    return(fp);     /* error */
}
```

Figure 7.14 Incorrect usage of an automatic variable

The problem is that when `open_data` returns, the space it used on the stack will be used by the stack frame for the next function that is called. But the standard I/O library will still be using that portion of memory for its stream buffer. Chaos is sure to result. To correct this problem, the array `databuf` needs to be allocated from global memory, either statically (`static` or `extern`) or dynamically (one of the `alloc` functions).

7.11 `getrlimit` and `setrlimit` Functions

Every process has a set of resource limits, some of which can be queried and changed by the `getrlimit` and `setrlimit` functions.

```
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlptr);

int setrlimit(int resource, const struct rlimit *rlptr);
```

Both return: 0 if OK, –1 on error

These two functions are defined in the XSI option in the Single UNIX Specification. The resource limits for a process are normally established by process 0 when the system is initialized and then inherited by each successive process. Each implementation has its own way of tuning the various limits.

Each call to these two functions specifies a single *resource* and a pointer to the following structure:

```
struct rlimit {
  rlim_t  rlim_cur;  /* soft limit: current limit */
  rlim_t  rlim_max;  /* hard limit: maximum value for rlim_cur */
};
```

Three rules govern the changing of the resource limits.

1. A process can change its soft limit to a value less than or equal to its hard limit.
2. A process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.
3. Only a superuser process can raise a hard limit.

An infinite limit is specified by the constant RLIM_INFINITY.

The *resource* argument takes on one of the following values. Figure 7.15 shows which limits are defined by the Single UNIX Specification and supported by each implementation.

RLIMIT_AS	The maximum size in bytes of a process's total available memory. This affects the sbrk function (Section 1.11) and the mmap function (Section 14.8).
RLIMIT_CORE	The maximum size in bytes of a core file. A limit of 0 prevents the creation of a core file.
RLIMIT_CPU	The maximum amount of CPU time in seconds. When the soft limit is exceeded, the SIGXCPU signal is sent to the process.
RLIMIT_DATA	The maximum size in bytes of the data segment: the sum of the initialized data, uninitialized data, and heap from Figure 7.6.
RLIMIT_FSIZE	The maximum size in bytes of a file that may be created. When the soft limit is exceeded, the process is sent the SIGXFSZ signal.
RLIMIT_MEMLOCK	The maximum amount of memory in bytes that a process can lock into memory using mlock(2).
RLIMIT_MSGQUEUE	The maximum amount of memory in bytes that a process can allocate for POSIX message queues.
RLIMIT_NICE	The limit to which a process's nice value (Section 8.16) can be raised to affect its scheduling priority.
RLIMIT_NOFILE	The maximum number of open files per process. Changing this limit affects the value returned by the sysconf function for its _SC_OPEN_MAX argument (Section 2.5.4). See Figure 2.17 also.
RLIMIT_NPROC	The maximum number of child processes per real user ID. Changing this limit affects the value returned for _SC_CHILD_MAX by the sysconf function (Section 2.5.4).
RLIMIT_NPTS	The maximum number of pseudo terminals (Chapter 19) that a user can have open at one time.

RLIMIT_RSS	Maximum resident set size (RSS) in bytes. If available physical memory is low, the kernel takes memory from processes that exceed their RSS.
RLIMIT_SBSIZE	The maximum size in bytes of socket buffers that a user can consume at any given time.
RLIMIT_SIGPENDING	The maximum number of signals that can be queued for a process. This limit is enforced by the sigqueue function (Section 10.20).
RLIMIT_STACK	The maximum size in bytes of the stack. See Figure 7.6.
RLIMIT_SWAP	The maximum amount of swap space in bytes that a user can consume.
RLIMIT_VMEM	This is a synonym for RLIMIT_AS.

Limit	XSI	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
RLIMIT_AS	•	•	•		•
RLIMIT_CORE	•	•	•	•	•
RLIMIT_CPU	•	•	•	•	•
RLIMIT_DATA	•	•	•	•	•
RLIMIT_FSIZE	•	•	•	•	•
RLIMIT_MEMLOCK		•	•	•	
RLIMIT_MSGQUEUE			•		
RLIMIT_NICE			•		
RLIMIT_NOFILE	•	•	•	•	•
RLIMIT_NPROC		•	•	•	
RLIMIT_NPTS		•			
RLIMIT_RSS		•	•	•	
RLIMIT_SBSIZE		•			
RLIMIT_SIGPENDING			•		
RLIMIT_STACK	•	•	•	•	•
RLIMIT_SWAP		•			
RLIMIT_VMEM					•

Figure 7.15 Support for resource limits

The resource limits affect the calling process and are inherited by any of its children. This means that the setting of resource limits needs to be built into the shells to affect all our future processes. Indeed, the Bourne shell, the GNU Bourne-again shell, and the Korn shell have the built-in `ulimit` command, and the C shell has the built-in `limit` command. (The `umask` and `chdir` functions also have to be handled as shell built-ins.)

Example

The program in Figure 7.16 prints out the current soft limit and hard limit for all the resource limits supported on the system. To compile this program on all the various

implementations, we have conditionally included the resource names that differ. Note that some systems define `rlim_t` to be an `unsigned long long` instead of an `unsigned long`. This definition can even change on the same system, depending on whether we compile the program to support 64-bit files. Some limits apply to file size, so the `rlim_t` type has to be large enough to represent a file size limit. To avoid compiler warnings that use the wrong format specification, we first copy the limit into a 64-bit integer so that we have to deal with only one format.

```
#include "apue.h"
#include <sys/resource.h>

#define doit(name)  pr_limits(#name, name)

static void pr_limits(char *, int);

int
main(void)
{
#ifdef  RLIMIT_AS
    doit(RLIMIT_AS);
#endif

    doit(RLIMIT_CORE);
    doit(RLIMIT_CPU);
    doit(RLIMIT_DATA);
    doit(RLIMIT_FSIZE);

#ifdef  RLIMIT_MEMLOCK
    doit(RLIMIT_MEMLOCK);
#endif

#ifdef RLIMIT_MSGQUEUE
    doit(RLIMIT_MSGQUEUE);
#endif

#ifdef RLIMIT_NICE
    doit(RLIMIT_NICE);
#endif

    doit(RLIMIT_NOFILE);

#ifdef  RLIMIT_NPROC
    doit(RLIMIT_NPROC);
#endif

#ifdef RLIMIT_NPTS
    doit(RLIMIT_NPTS);
#endif

#ifdef  RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif

#ifdef  RLIMIT_SBSIZE
    doit(RLIMIT_SBSIZE);
```

```
#endif

#ifdef RLIMIT_SIGPENDING
    doit(RLIMIT_SIGPENDING);
#endif

    doit(RLIMIT_STACK);

#ifdef RLIMIT_SWAP
    doit(RLIMIT_SWAP);
#endif

#ifdef  RLIMIT_VMEM
    doit(RLIMIT_VMEM);
#endif

    exit(0);
}

static void
pr_limits(char *name, int resource)
{
    struct rlimit       limit;
    unsigned long long  lim;

    if (getrlimit(resource, &limit) < 0)
        err_sys("getrlimit error for %s", name);
    printf("%-14s  ", name);
    if (limit.rlim_cur == RLIM_INFINITY) {
        printf("(infinite)  ");
    } else {
        lim = limit.rlim_cur;
        printf("%10lld  ", lim);
    }
    if (limit.rlim_max == RLIM_INFINITY) {
        printf("(infinite)");
    } else {
        lim = limit.rlim_max;
        printf("%10lld", lim);
    }
    putchar((int)'\n');
}
```

Figure 7.16 Print the current resource limits

Note that we've used the ISO C string-creation operator (#) in the `doit` macro, to generate the string value for each resource name. When we say

```
doit(RLIMIT_CORE);
```

the C preprocessor expands this into

```
pr_limits("RLIMIT_CORE", RLIMIT_CORE);
```

Running this program under FreeBSD gives us the following output:

```
$ ./a.out
RLIMIT_AS       (infinite)  (infinite)
RLIMIT_CORE     (infinite)  (infinite)
RLIMIT_CPU      (infinite)  (infinite)
RLIMIT_DATA      536870912   536870912
RLIMIT_FSIZE    (infinite)  (infinite)
RLIMIT_MEMLOCK  (infinite)  (infinite)
RLIMIT_NOFILE         3520        3520
RLIMIT_NPROC          1760        1760
RLIMIT_NPTS     (infinite)  (infinite)
RLIMIT_RSS      (infinite)  (infinite)
RLIMIT_SBSIZE   (infinite)  (infinite)
RLIMIT_STACK      67108864    67108864
RLIMIT_SWAP     (infinite)  (infinite)
RLIMIT_VMEM     (infinite)  (infinite)
```

Solaris gives us the following results:

```
$ ./a.out
RLIMIT_AS       (infinite)  (infinite)
RLIMIT_CORE     (infinite)  (infinite)
RLIMIT_CPU      (infinite)  (infinite)
RLIMIT_DATA     (infinite)  (infinite)
RLIMIT_FSIZE    (infinite)  (infinite)
RLIMIT_NOFILE          256       65536
RLIMIT_STACK       8388608  (infinite)
RLIMIT_VMEM     (infinite)  (infinite)
```

□

Exercise 10.11 continues the discussion of resource limits, after we've covered signals.

7.12 Summary

Understanding the environment of a C program within a UNIX system's environment is a prerequisite to understanding the process control features of the UNIX System. In this chapter, we've looked at how a process is started, how it can terminate, and how it's passed an argument list and an environment. Although both the argument list and the environment are uninterpreted by the kernel, it is the kernel that passes both from the caller of `exec` to the new process.

We've also examined the typical memory layout of a C program and seen how a process can dynamically allocate and free memory. It is worthwhile to look in detail at the functions available for manipulating the environment, since they involve memory allocation. The functions `setjmp` and `longjmp` were presented, providing a way to perform nonlocal branching within a process. We finished the chapter by describing the resource limits that various implementations provide.

Exercises

7.1 On an Intel x86 system under Linux, if we execute the program that prints "hello, world" and do not call `exit` or `return`, the termination status of the program—which we can examine with the shell—is 13. Why?

7.2 When is the output from the `printfs` in Figure 7.3 actually output?

7.3 Is there any way for a function that is called by `main` to examine the command-line arguments without (a) passing `argc` and `argv` as arguments from `main` to the function or (b) having `main` copy `argc` and `argv` into global variables?

7.4 Some UNIX system implementations purposely arrange that, when a program is executed, location 0 in the data segment is not accessible. Why?

7.5 Use the `typedef` facility of C to define a new data type `Exitfunc` for an exit handler. Redo the prototype for `atexit` using this data type.

7.6 If we allocate an array of `long`s using `calloc`, is the array initialized to 0? If we allocate an array of pointers using `calloc`, is the array initialized to null pointers?

7.7 In the output from the `size` command at the end of Section 7.6, why aren't any sizes given for the heap and the stack?

7.8 In Section 7.7, the two file sizes (879443 and 8378) don't equal the sums of their respective text and data sizes. Why?

7.9 In Section 7.7, why does the size of the executable file differ so dramatically when we use shared libraries for such a trivial program?

7.10 At the end of Section 7.10, we showed how a function can't return a pointer to an automatic variable. Is the following code correct?

```
int
f1(int val)
{
    int     num = 0;
    int     *ptr = &num;

    if (val == 0) {
        int     val;

        val = 5;
        ptr = &val;
    }
    return(*ptr + 1);
}
```

8

Process Control

8.1 Introduction

We now turn to the process control provided by the UNIX System. This includes the creation of new processes, program execution, and process termination. We also look at the various IDs that are the property of the process—real, effective, and saved; user and group IDs—and how they're affected by the process control primitives. Interpreter files and the `system` function are also covered. We conclude the chapter by looking at the process accounting provided by most UNIX systems. This lets us look at the process control functions from a different perspective.

8.2 Process Identifiers

Every process has a unique process ID, a non-negative integer. Because the process ID is the only well-known identifier of a process that is always unique, it is often used as a piece of other identifiers, to guarantee uniqueness. For example, applications sometimes include the process ID as part of a filename in an attempt to generate unique filenames.

Although unique, process IDs are reused. As processes terminate, their IDs become candidates for reuse. Most UNIX systems implement algorithms to delay reuse, however, so that newly created processes are assigned IDs different from those used by processes that terminated recently. This prevents a new process from being mistaken for the previous process to have used the same ID.

There are some special processes, but the details differ from implementation to implementation. Process ID 0 is usually the scheduler process and is often known as the *swapper*. No program on disk corresponds to this process, which is part of the

kernel and is known as a system process. Process ID 1 is usually the `init` process and is invoked by the kernel at the end of the bootstrap procedure. The program file for this process was `/etc/init` in older versions of the UNIX System and is `/sbin/init` in newer versions. This process is responsible for bringing up a UNIX system after the kernel has been bootstrapped. `init` usually reads the system-dependent initialization files—the `/etc/rc*` files or `/etc/inittab` and the files in `/etc/init.d`—and brings the system to a certain state, such as multiuser. The `init` process never dies. It is a normal user process, not a system process within the kernel, like the swapper, although it does run with superuser privileges. Later in this chapter, we'll see how `init` becomes the parent process of any orphaned child process.

> In Mac OS X 10.4, the `init` process was replaced with the `launchd` process, which performs the same set of tasks as `init`, but has expanded functionality. See Section 5.10 in Singh [2006] for a discussion of how `launchd` operates.

Each UNIX System implementation has its own set of kernel processes that provide operating system services. For example, on some virtual memory implementations of the UNIX System, process ID 2 is the *pagedaemon*. This process is responsible for supporting the paging of the virtual memory system.

In addition to the process ID, there are other identifiers for every process. The following functions return these identifiers.

```
#include <unistd.h>

pid_t getpid(void);
                                        Returns: process ID of calling process

pid_t getppid(void);
                                    Returns: parent process ID of calling process

uid_t getuid(void);
                                        Returns: real user ID of calling process

uid_t geteuid(void);
                                    Returns: effective user ID of calling process

gid_t getgid(void);
                                        Returns: real group ID of calling process

gid_t getegid(void);
                                   Returns: effective group ID of calling process
```

Note that none of these functions has an error return. We'll return to the parent process ID in the next section when we discuss the `fork` function. The real and effective user and group IDs were discussed in Section 4.4.

8.3 fork Function

An existing process can create a new one by calling the fork function.

```
#include <unistd.h>

pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, –1 on error

The new process created by fork is called the *child process*. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child. The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children. The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)

Both the child and the parent continue executing with the instruction that follows the call to fork. The child is a copy of the parent. For example, the child gets a copy of the parent's data space, heap, and stack. Note that this is a copy for the child; the parent and the child do not share these portions of memory. The parent and the child do share the text segment, however (Section 7.6).

Modern implementations don't perform a complete copy of the parent's data, stack, and heap, since a fork is often followed by an exec. Instead, a technique called *copy-on-write* (COW) is used. These regions are shared by the parent and the child and have their protection changed by the kernel to read-only. If either process tries to modify these regions, the kernel then makes a copy of that piece of memory only, typically a "page" in a virtual memory system. Section 9.2 of Bach [1986] and Sections 5.6 and 5.7 of McKusick et al. [1996] provide more detail on this feature.

> Variations of the fork function are provided by some platforms. All four platforms discussed in this book support the vfork(2) variant discussed in the next section.
>
> Linux 3.2.0 also provides new process creation through the clone(2) system call. This is a generalized form of fork that allows the caller to control what is shared between parent and child.
>
> FreeBSD 8.0 provides the rfork(2) system call, which is similar to the Linux clone system call. The rfork call is derived from the Plan 9 operating system (Pike et al. [1995]).
>
> Solaris 10 provides two threads libraries: one for POSIX threads (pthreads) and one for Solaris threads. In previous releases, the behavior of fork differed between the two thread libraries. For POSIX threads, fork created a process containing only the calling thread, but for Solaris threads, fork created a process containing copies of all threads from the process of the calling thread. In Solaris 10, this behavior has changed; fork creates a child containing a copy of the calling thread only, regardless of which thread library is used. Solaris also provides the fork1 function, which can be used to create a process that duplicates only the calling thread, and the forkall function, which can be used to create a process that duplicates all the threads in the process. Threads are discussed in detail in Chapters 11 and 12.

Example

The program in Figure 8.1 demonstrates the `fork` function, showing how changes to variables in a child process do not affect the value of the variables in the parent process.

```
#include "apue.h"

int     globvar = 6;        /* external variable in initialized data */
char    buf[] = "a write to stdout\n";

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");    /* we don't flush stdout */

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {      /* child */
        globvar++;              /* modify variables */
        var++;
    } else {
        sleep(2);               /* parent */
    }

    printf("pid = %ld, glob = %d, var = %d\n", (long)getpid(), globvar,
      var);
    exit(0);
}
```

Figure 8.1 Example of `fork` function

If we execute this program, we get

```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89      child's variables were changed
pid = 429, glob = 6, var = 88      parent's copy was not changed
$ ./a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

In general, we never know whether the child starts executing before the parent, or vice versa. The order depends on the scheduling algorithm used by the kernel. If it's required that the child and parent synchronize their actions, some form of interprocess

communication is required. In the program shown in Figure 8.1, we simply have the parent put itself to sleep for 2 seconds, to let the child execute. There is no guarantee that the length of this delay is adequate, and we talk about this and other types of synchronization in Section 8.9 when we discuss race conditions. In Section 10.16, we show how to use signals to synchronize a parent and a child after a `fork`.

When we write to standard output, we subtract 1 from the size of `buf` to avoid writing the terminating null byte. Although `strlen` will calculate the length of a string not including the terminating null byte, `sizeof` calculates the size of the buffer, which does include the terminating null byte. Another difference is that using `strlen` requires a function call, whereas `sizeof` calculates the buffer length at compile time, as the buffer is initialized with a known string and its size is fixed.

Note the interaction of `fork` with the I/O functions in the program in Figure 8.1. Recall from Chapter 3 that the `write` function is not buffered. Because `write` is called before the `fork`, its data is written once to standard output. The standard I/O library, however, is buffered. Recall from Section 5.12 that standard output is line buffered if it's connected to a terminal device; otherwise, it's fully buffered. When we run the program interactively, we get only a single copy of the first `printf` line, because the standard output buffer is flushed by the newline. When we redirect standard output to a file, however, we get two copies of the `printf` line. In this second case, the `printf` before the `fork` is called once, but the line remains in the buffer when `fork` is called. This buffer is then copied into the child when the parent's data space is copied to the child. Both the parent and the child now have a standard I/O buffer with this line in it. The second `printf`, right before the `exit`, just appends its data to the existing buffer. When each process terminates, its copy of the buffer is finally flushed. □

File Sharing

When we redirect the standard output of the parent from the program in Figure 8.1, the child's standard output is also redirected. Indeed, one characteristic of `fork` is that all file descriptors that are open in the parent are duplicated in the child. We say "duplicated" because it's as if the `dup` function had been called for each descriptor. The parent and the child share a file table entry for every open descriptor (recall Figure 3.9).

Consider a process that has three different files opened for standard input, standard output, and standard error. On return from `fork`, we have the arrangement shown in Figure 8.2.

It is important that the parent and the child share the same file offset. Consider a process that `fork`s a child, then `wait`s for the child to complete. Assume that both processes write to standard output as part of their normal processing. If the parent has its standard output redirected (by a shell, perhaps), it is essential that the parent's file offset be updated by the child when the child writes to standard output. In this case, the child can write to standard output while the parent is `wait`ing for it; on completion of the child, the parent can continue writing to standard output, knowing that its output will be appended to whatever the child wrote. If the parent and the child did not share the same file offset, this type of interaction would be more difficult to accomplish and would require explicit actions by the parent.

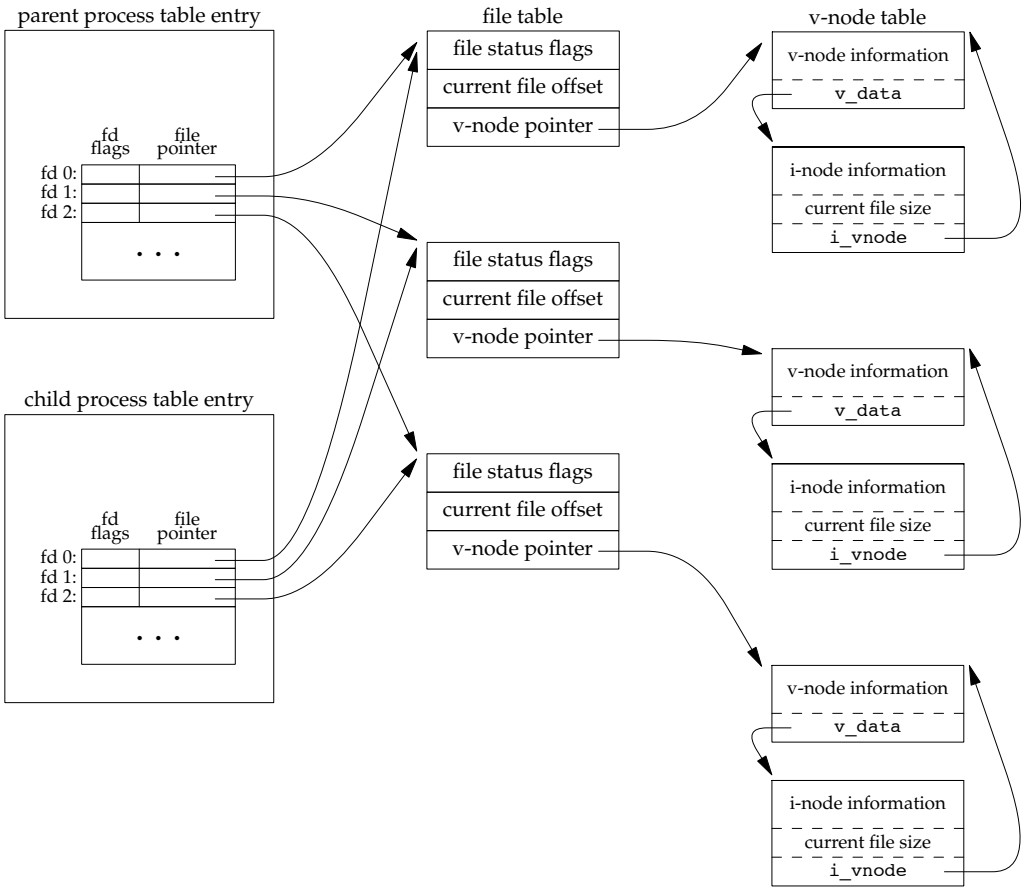


Figure 8.2 Sharing of open files between parent and child after `fork`

If both parent and child write to the same descriptor, without any form of synchronization, such as having the parent `wait` for the child, their output will be intermixed (assuming it's a descriptor that was open before the `fork`). Although this is possible—we saw it in Figure 8.2—it's not the normal mode of operation.

There are two normal cases for handling the descriptors after a `fork`.

1. The parent waits for the child to complete. In this case, the parent does not need to do anything with its descriptors. When the child terminates, any of the shared descriptors that the child read from or wrote to will have their file offsets updated accordingly.
2. Both the parent and the child go their own ways. Here, after the `fork`, the parent closes the descriptors that it doesn't need, and the child does the same thing. This way, neither interferes with the other's open descriptors. This scenario is often found with network servers.

Besides the open files, numerous other properties of the parent are inherited by the child:

- Real user ID, real group ID, effective user ID, and effective group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- The set-user-ID and set-group-ID flags
- Current working directory
- Root directory
- File mode creation mask
- Signal mask and dispositions
- The close-on-exec flag for any open file descriptors
- Environment
- Attached shared memory segments
- Memory mappings
- Resource limits

The differences between the parent and child are

- The return values from `fork` are different.
- The process IDs are different.
- The two processes have different parent process IDs: the parent process ID of the child is the parent; the parent process ID of the parent doesn't change.
- The child's `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` values are set to 0 (these times are discussed in Section 8.17).
- File locks set by the parent are not inherited by the child.
- Pending alarms are cleared for the child.
- The set of pending signals for the child is set to the empty set.

Many of these features haven't been discussed yet—we'll cover them in later chapters.

The two main reasons for `fork` to fail are (a) if too many processes are already in the system, which usually means that something else is wrong, or (b) if the total number of processes for this real user ID exceeds the system's limit. Recall from Figure 2.11 that `CHILD_MAX` specifies the maximum number of simultaneous processes per real user ID.

There are two uses for `fork`:

1. When a process wants to duplicate itself so that the parent and the child can each execute different sections of code at the same time. This is common for network servers—the parent waits for a service request from a client. When the request arrives, the parent calls `fork` and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.
2. When a process wants to execute a different program. This is common for shells. In this case, the child does an `exec` (which we describe in Section 8.10) right after it returns from the `fork`.

Some operating systems combine the operations from step 2—a `fork` followed by an `exec`—into a single operation called a *spawn*. The UNIX System separates the two, as there are numerous cases where it is useful to `fork` without doing an `exec`. Also, separating the two operations allows the child to change the per-process attributes between the `fork` and the `exec`, such as I/O redirection, user ID, signal disposition, and so on. We'll see numerous examples of this in Chapter 15.

> The Single UNIX Specification does include `spawn` interfaces in the advanced real-time option group. These interfaces are not intended to be replacements for `fork` and `exec`, however. They are intended to support systems that have difficulty implementing `fork` efficiently, especially systems without hardware support for memory management.

8.4 `vfork` Function

The function `vfork` has the same calling sequence and same return values as `fork`, but the semantics of the two functions differ.

> The `vfork` function originated with 2.9BSD. Some consider the function a blemish, but all the platforms covered in this book support it. In fact, the BSD developers removed it from the 4.4BSD release, but all the open source BSD distributions that derive from 4.4BSD added support for it back into their own releases. The `vfork` function was marked as an obsolescent interface in Version 3 of the Single UNIX Specification and was removed entirely in Version 4. We include it here for historical reasons only. Portable applications should not use it.

The `vfork` function was intended to create a new process for the purpose of executing a new program (step 2 at the end of the previous section), similar to the method used by the bare-bones shell from Figure 1.7. The `vfork` function creates the new process, just like `fork`, without copying the address space of the parent into the child, as the child won't reference that address space; the child simply calls `exec` (or `exit`) right after the `vfork`. Instead, the child runs in the address space of the parent until it calls either `exec` or `exit`. This optimization is more efficient on some implementations of the UNIX System, but leads to undefined results if the child modifies any data (except the variable used to hold the return value from `vfork`), makes function calls, or returns without calling `exec` or `exit`. (As we mentioned in the previous section, implementations use copy-on-write to improve the efficiency of a `fork` followed by an `exec`, but no copying is still faster than some copying.)

Another difference between the two functions is that `vfork` guarantees that the child runs first, until the child calls `exec` or `exit`. When the child calls either of these functions, the parent resumes. (This can lead to deadlock if the child depends on further actions of the parent before calling either of these two functions.)

Example

The program in Figure 8.3 is a modified version of the program from Figure 8.1. We've replaced the call to `fork` with `vfork` and removed the `write` to standard output. Also, we don't need to have the parent call `sleep`, as we're guaranteed that it is put to sleep by the kernel until the child calls either `exec` or `exit`.

```
#include "apue.h"

int     globvar = 6;        /* external variable in initialized data */

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    printf("before vfork\n");   /* we don't flush stdio */
    if ((pid = vfork()) < 0) {
        err_sys("vfork error");
    } else if (pid == 0) {      /* child */
        globvar++;              /* modify parent's variables */
        var++;
        _exit(0);               /* child terminates */
    }

    /* parent continues here */
    printf("pid = %ld, glob = %d, var = %d\n", (long)getpid(), globvar,
      var);
    exit(0);
}
```

Figure 8.3 Example of `vfork` function

Running this program gives us

```
$ ./a.out
before vfork
pid = 29039, glob = 7, var = 89
```

Here, the incrementing of the variables done by the child changes the values in the parent. Because the child runs in the address space of the parent, this doesn't surprise us. This behavior, however, differs from the behavior of `fork`.

Note in Figure 8.3 that we call `_exit` instead of `exit`. As we described in Section 7.3, `_exit` does not perform any flushing of standard I/O buffers. If we call `exit` instead, the results are indeterminate. Depending on the implementation of the standard I/O library, we might see no difference in the output, or we might find that the output from the first `printf` in the parent has disappeared.

If the child calls `exit`, the implementation flushes the standard I/O streams. If this is the only action taken by the library, then we will see no difference from the output generated if the child called `_exit`. If the implementation also closes the standard I/O streams, however, the memory representing the `FILE` object for the standard output will be cleared out. Because the child is borrowing the parent's address space, when the parent resumes and calls `printf`, no output will appear and `printf` will return –1. Note that the parent's `STDOUT_FILENO` is still valid, as the child gets a copy of the parent's file descriptor array (refer back to Figure 8.2).

Most modern implementations of `exit` do not bother to close the streams. Because the process is about to exit, the kernel will close all the file descriptors open in the process. Closing them in the library simply adds overhead without any benefit.

□

Section 5.6 of McKusick et al. [1996] contains additional information on the implementation issues of `fork` and `vfork`. Exercises 8.1 and 8.2 continue the discussion of `vfork`.

8.5 `exit` Functions

As we described in Section 7.3, a process can terminate normally in five ways:

1. Executing a `return` from the `main` function. As we saw in Section 7.3, this is equivalent to calling `exit`.
2. Calling the `exit` function. This function is defined by ISO C and includes the calling of all exit handlers that have been registered by calling `atexit` and closing all standard I/O streams. Because ISO C does not deal with file descriptors, multiple processes (parents and children), and job control, the definition of this function is incomplete for a UNIX system.
3. Calling the `_exit` or `_Exit` function. ISO C defines `_Exit` to provide a way for a process to terminate without running exit handlers or signal handlers. Whether standard I/O streams are flushed depends on the implementation. On UNIX systems, `_Exit` and `_exit` are synonymous and do not flush standard I/O streams. The `_exit` function is called by `exit` and handles the UNIX system-specific details; `_exit` is specified by POSIX.1.

   In most UNIX system implementations, `exit`(3) is a function in the standard C library, whereas `_exit`(2) is a system call.

4. Executing a `return` from the start routine of the last thread in the process. The return value of the thread is not used as the return value of the process, however. When the last thread returns from its start routine, the process exits with a termination status of 0.
5. Calling the `pthread_exit` function from the last thread in the process. As with the previous case, the exit status of the process in this situation is always 0, regardless of the argument passed to `pthread_exit`. We'll say more about `pthread_exit` in Section 11.5.

The three forms of abnormal termination are as follows:

1. Calling `abort`. This is a special case of the next item, as it generates the `SIGABRT` signal.
2. When the process receives certain signals. (We describe signals in more detail in Chapter 10.) The signal can be generated by the process itself (e.g., by calling the `abort` function), by some other process, or by the kernel. Examples of

signals generated by the kernel include the process referencing a memory location not within its address space or trying to divide by 0.

3. The last thread responds to a cancellation request. By default, cancellation occurs in a deferred manner: one thread requests that another be canceled, and sometime later the target thread terminates. We discuss cancellation requests in detail in Sections 11.5 and 12.7.

Regardless of how a process terminates, the same code in the kernel is eventually executed. This kernel code closes all the open descriptors for the process, releases the memory that it was using, and so on.

For any of the preceding cases, we want the terminating process to be able to notify its parent how it terminated. For the three exit functions (`exit`, `_exit`, and `_Exit`), this is done by passing an exit status as the argument to the function. In the case of an abnormal termination, however, the kernel—not the process—generates a termination status to indicate the reason for the abnormal termination. In any case, the parent of the process can obtain the termination status from either the `wait` or the `waitpid` function (described in the next section).

Note that we differentiate between the exit status, which is the argument to one of the three exit functions or the return value from `main`, and the termination status. The exit status is converted into a termination status by the kernel when `_exit` is finally called (recall Figure 7.2). Figure 8.4 describes the various ways the parent can examine the termination status of a child. If the child terminated normally, the parent can obtain the exit status of the child.

When we described the `fork` function, it was obvious that the child has a parent process after the call to `fork`. Now we're talking about returning a termination status to the parent. But what happens if the parent terminates before the child? The answer is that the `init` process becomes the parent process of any process whose parent terminates. In such a case, we say that the process has been inherited by `init`. What normally happens is that whenever a process terminates, the kernel goes through all active processes to see whether the terminating process is the parent of any process that still exists. If so, the parent process ID of the surviving process is changed to be 1 (the process ID of `init`). This way, we're guaranteed that every process has a parent.

Another condition we have to worry about is when a child terminates before its parent. If the child completely disappeared, the parent wouldn't be able to fetch its termination status when and if the parent was finally ready to check if the child had terminated. The kernel keeps a small amount of information for every terminating process, so that the information is available when the parent of the terminating process calls `wait` or `waitpid`. Minimally, this information consists of the process ID, the termination status of the process, and the amount of CPU time taken by the process. The kernel can discard all the memory used by the process and close its open files. In UNIX System terminology, a process that has terminated, but whose parent has not yet waited for it, is called a *zombie*. The `ps`(1) command prints the state of a zombie process as *Z*. If we write a long-running program that `fork`s many child processes, they become zombies unless we wait for them and fetch their termination status.

Some systems provide ways to prevent the creation of zombies, as we describe in Section 10.7.

The final condition to consider is this: What happens when a process that has been inherited by `init` terminates? Does it become a zombie? The answer is "no," because `init` is written so that whenever one of its children terminates, `init` calls one of the `wait` functions to fetch the termination status. By doing this, `init` prevents the system from being clogged by zombies. When we say "one of `init`'s children," we mean either a process that `init` generates directly (such as `getty`, which we describe in Section 9.2) or a process whose parent has terminated and has been subsequently inherited by `init`.

8.6 `wait` and `waitpid` Functions

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the `SIGCHLD` signal to the parent. Because the termination of a child is an asynchronous event—it can happen at any time while the parent is running—this signal is the asynchronous notification from the kernel to the parent. The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler. The default action for this signal is to be ignored. We describe these options in Chapter 10. For now, we need to be aware that a process that calls `wait` or `waitpid` can

- Block, if all of its children are still running
- Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
- Return immediately with an error, if it doesn't have any child processes

If the process is calling `wait` because it received the `SIGCHLD` signal, we expect `wait` to return immediately. But if we call it at any random point in time, it can block.

```
#include <sys/wait.h>

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both return: process ID if OK, 0 (see later), or –1 on error

The differences between these two functions are as follows:

- The `wait` function can block the caller until a child process terminates, whereas `waitpid` has an option that prevents it from blocking.
- The `waitpid` function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

If a child has already terminated and is a zombie, `wait` returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates. If the caller blocks and has multiple children, `wait` returns when one terminates. We can always tell which child terminated, because the process ID is returned by the function.

For both functions, the argument *statloc* is a pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument. If we don't care about the termination status, we simply pass a null pointer as this argument.

Traditionally, the integer status that these two functions return has been defined by the implementation, with certain bits indicating the exit status (for a normal return), other bits indicating the signal number (for an abnormal return), one bit indicating whether a core file was generated, and so on. POSIX.1 specifies that the termination status is to be looked at using various macros that are defined in `<sys/wait.h>`. Four mutually exclusive macros tell us how the process terminated, and they all begin with `WIF`. Based on which of these four macros is true, other macros are used to obtain the exit status, signal number, and the like. The four mutually exclusive macros are shown in Figure 8.4.

Macro	Description
`WIFEXITED(`*status*`)`	True if status was returned for a child that terminated normally. In this case, we can execute `WEXITSTATUS(`*status*`)` to fetch the low-order 8 bits of the argument that the child passed to `exit`, `_exit`, or `_Exit`.

`WIFSIGNALED(`*status*`)`	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute
	`WTERMSIG(`*status*`)`
	to fetch the signal number that caused the termination.
	Additionally, some implementations (but not the Single UNIX Specification) define the macro
	`WCOREDUMP(`*status*`)`
	that returns true if a core file of the terminated process was generated.

`WIFSTOPPED(`*status*`)`	True if status was returned for a child that is currently stopped. In this case, we can execute `WSTOPSIG(`*status*`)` to fetch the signal number that caused the child to stop.
`WIFCONTINUED(`*status*`)`	True if status was returned for a child that has been continued after a job control stop (XSI option; `waitpid` only).

Figure 8.4 Macros to examine the termination status returned by `wait` and `waitpid`

We'll discuss how a process can be stopped in Section 9.8 when we discuss job control.

Example

The function `pr_exit` in Figure 8.5 uses the macros from Figure 8.4 to print a description of the termination status. We'll call this function from numerous programs in the text. Note that this function handles the `WCOREDUMP` macro, if it is defined.

```
#include "apue.h"
#include <sys/wait.h>

void
pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
                WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
                WTERMSIG(status),
#ifdef  WCOREDUMP
                WCOREDUMP(status) ? " (core file generated)" : "");
#else
                "");
#endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
                WSTOPSIG(status));
}
```

Figure 8.5 Print a description of the `exit` status

FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10 all support the `WCOREDUMP` macro. However, some platforms hide its definition if the `_POSIX_C_SOURCE` constant is defined (recall Section 2.7).

The program shown in Figure 8.6 calls the `pr_exit` function, demonstrating the various values for the termination status. If we run the program in Figure 8.6, we get

```
$ ./a.out
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
abnormal termination, signal number = 8 (core file generated)
```

For now, we print the signal number from `WTERMSIG`. We can look at the `<signal.h>` header to verify that `SIGABRT` has a value of 6 and that `SIGFPE` has a value of 8. We'll see a portable way to map a signal number to a descriptive name in Section 10.22. □

As we mentioned, if we have more than one child, `wait` returns on termination of any of the children. But what if we want to wait for a specific process to terminate (assuming we know which process ID we want to wait for)? In older versions of the UNIX System, we would have to call `wait` and compare the returned process ID with the one we're interested in. If the terminated process wasn't the one we wanted, we would have to save the process ID and termination status and call `wait` again. We would need to continue doing this until the desired process terminated. The next time we wanted to wait for a specific process, we would go through the list of already terminated processes to see whether we had already waited for it, and if not, call `wait`

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    pid_t   pid;
    int     status;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        exit(7);

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        abort();                    /* generates SIGABRT */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        status /= 0;                /* divide by 0 generates SIGFPE */

    if (wait(&status) != pid)       /* wait for child */
        err_sys("wait error");
    pr_exit(status);                /* and print its status */

    exit(0);
}
```

Figure 8.6 Demonstrate various `exit` statuses

again. What we need is a function that waits for a specific process. This functionality (and more) is provided by the POSIX.1 `waitpid` function.

The interpretation of the *pid* argument for `waitpid` depends on its value:

pid == –1	Waits for any child process. In this respect, `waitpid` is equivalent to `wait`.
pid > 0	Waits for the child whose process ID equals *pid*.
pid == 0	Waits for any child whose process group ID equals that of the calling process. (We discuss process groups in Section 9.4.)
pid < –1	Waits for any child whose process group ID equals the absolute value of *pid*.

The `waitpid` function returns the process ID of the child that terminated and stores the child's termination status in the memory location pointed to by *statloc*. With `wait`, the only real error is if the calling process has no children. (Another error return is possible, in case the function call is interrupted by a signal. We'll discuss this in Chapter 10.) With `waitpid`, however, it's also possible to get an error if the specified process or process group does not exist or is not a child of the calling process.

The *options* argument lets us further control the operation of `waitpid`. This argument either is 0 or is constructed from the bitwise OR of the constants in Figure 8.7.

> FreeBSD 8.0 and Solaris 10 support one additional, but nonstandard, *option* constant. `WNOWAIT` has the system keep the process whose termination status is returned by `waitpid` in a wait state, so that it may be waited for again.

Constant	Description
`WCONTINUED`	If the implementation supports job control, the status of any child specified by *pid* that has been continued after being stopped, but whose status has not yet been reported, is returned (XSI option).
`WNOHANG`	The `waitpid` function will not block if a child specified by *pid* is not immediately available. In this case, the return value is 0.
`WUNTRACED`	If the implementation supports job control, the status of any child specified by *pid* that has stopped, and whose status has not been reported since it has stopped, is returned. The `WIFSTOPPED` macro determines whether the return value corresponds to a stopped child process.

Figure 8.7 The *options* constants for `waitpid`

The `waitpid` function provides three features that aren't provided by the `wait` function.

1. The `waitpid` function lets us wait for one particular process, whereas the `wait` function returns the status of any terminated child. We'll return to this feature when we discuss the `popen` function.
2. The `waitpid` function provides a nonblocking version of `wait`. There are times when we want to fetch a child's status, but we don't want to block.
3. The `waitpid` function provides support for job control with the `WUNTRACED` and `WCONTINUED` options.

Example

Recall our discussion in Section 8.5 about zombie processes. If we want to write a process so that it `fork`s a child but we don't want to wait for the child to complete and we don't want the child to become a zombie until we terminate, the trick is to call `fork` twice. The program in Figure 8.8 does this.

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {      /* first child */
        if ((pid = fork()) < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit(0);    /* parent from second fork == first child */

        /*
         * We're the second child; our parent becomes init as soon
         * as our real parent calls exit() in the statement above.
         * Here's where we'd continue executing, knowing that when
         * we're done, init will reap our status.
         */
        sleep(2);
        printf("second child, parent pid = %ld\n", (long)getppid());
        exit(0);
    }

    if (waitpid(pid, NULL, 0) != pid)   /* wait for first child */
        err_sys("waitpid error");

    /*
     * We're the parent (the original process); we continue executing,
     * knowing that we're not the parent of the second child.
     */
    exit(0);
}
```

Figure 8.8 Avoid zombie processes by calling `fork` twice

We call `sleep` in the second child to ensure that the first child terminates before printing the parent process ID. After a `fork`, either the parent or the child can continue executing; we never know which will resume execution first. If we didn't put the second child to sleep, and if it resumed execution after the `fork` before its parent, the parent process ID that it printed would be that of its parent, not process ID 1.

Executing the program in Figure 8.8 gives us

```
$ ./a.out
$ second child, parent pid = 1
```

Note that the shell prints its prompt when the original process terminates, which is before the second child prints its parent process ID. □

8.7 waitid Function

The Single UNIX Specification includes an additional function to retrieve the exit status of a process. The `waitid` function is similar to `waitpid`, but provides extra flexibility.

```
#include <sys/wait.h>

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Returns: 0 if OK, –1 on error

Like `waitpid`, `waitid` allows a process to specify which children to wait for. Instead of encoding this information in a single argument combined with the process ID or process group ID, two separate arguments are used. The *id* parameter is interpreted based on the value of *idtype*. The types supported are summarized in Figure 8.9.

Constant	Description
`P_PID`	Wait for a particular process: *id* contains the process ID of the child to wait for.
`P_PGID`	Wait for any child process in a particular process group: *id* contains the process group ID of the children to wait for.
`P_ALL`	Wait for any child process: *id* is ignored.

Figure 8.9 The *idtype* constants for `waitid`

The *options* argument is a bitwise OR of the flags shown in Figure 8.10. These flags indicate which state changes the caller is interested in.

Constant	Description
`WCONTINUED`	Wait for a process that has previously stopped and has been continued, and whose status has not yet been reported.
`WEXITED`	Wait for processes that have exited.
`WNOHANG`	Return immediately instead of blocking if there is no child exit status available.
`WNOWAIT`	Don't destroy the child exit status. The child's exit status can be retrieved by a subsequent call to `wait`, `waitid`, or `waitpid`.
`WSTOPPED`	Wait for a process that has stopped and whose status has not yet been reported.

Figure 8.10 The *options* constants for `waitid`

At least one of `WCONTINUED`, `WEXITED`, or `WSTOPPED` must be specified in the *options* argument.

The *infop* argument is a pointer to a `siginfo` structure. This structure contains detailed information about the signal generated that caused the state change in the child process. The `siginfo` structure is discussed further in Section 10.14.

> Of the four platforms covered in this book, only Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10 provide support for `waitid`. Note, however, that Mac OS X 10.6.8 doesn't set all the information we expect in the `siginfo` structure.

8.8 wait3 and wait4 Functions

Most UNIX system implementations provide two additional functions: wait3 and wait4. Historically, these two variants descend from the BSD branch of the UNIX System. The only feature provided by these two functions that isn't provided by the wait, waitid, and waitpid functions is an additional argument that allows the kernel to return a summary of the resources used by the terminated process and all its child processes.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

pid_t wait3(int *statloc, int options, struct rusage *rusage);

pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```

Both return: process ID if OK, 0, or –1 on error

The resource information includes such statistics as the amount of user CPU time, amount of system CPU time, number of page faults, number of signals received, and the like. Refer to the getrusage(2) manual page for additional details. (This resource information differs from the resource limits we described in Section 7.11.) Figure 8.11 details the various arguments supported by the wait functions.

Function	*pid*	*options*	*rusage*	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
wait				•	•	•	•	•
waitid	•	•		•		•	•	•
waitpid	•	•		•	•	•	•	•
wait3		•	•		•	•	•	•
wait4	•	•	•		•	•	•	•

Figure 8.11 Arguments supported by wait functions on various systems

The wait3 function was included in earlier versions of the Single UNIX Specification. In Version 2, wait3 was moved to the legacy category; wait3 was removed from the specification in Version 3.

8.9 Race Conditions

For our purposes, a race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run. The fork function is a lively breeding ground for race conditions, if any of the logic after the fork either explicitly or implicitly depends on whether the parent or child runs first after the fork. In general, we cannot predict which process runs first. Even if we knew which process would run first, what happens after that process starts running depends on the system load and the kernel's scheduling algorithm.

We saw a potential race condition in the program in Figure 8.8 when the second child printed its parent process ID. If the second child runs before the first child, then its parent process will be the first child. But if the first child runs first and has enough time to `exit`, then the parent process of the second child is `init`. Even calling `sleep`, as we did, guarantees nothing. If the system was heavily loaded, the second child could resume after `sleep` returns, before the first child has a chance to run. Problems of this form can be difficult to debug because they tend to work "most of the time."

A process that wants to wait for a child to terminate must call one of the `wait` functions. If a process wants to wait for its parent to terminate, as in the program from Figure 8.8, a loop of the following form could be used:

```
while (getppid() != 1)
    sleep(1);
```

The problem with this type of loop, called *polling*, is that it wastes CPU time, as the caller is awakened every second to test the condition.

To avoid race conditions and to avoid polling, some form of signaling is required between multiple processes. Signals can be used for this purpose, and we describe one way to do this in Section 10.16. Various forms of interprocess communication (IPC) can also be used. We'll discuss some of these options in Chapters 15 and 17.

For a parent and child relationship, we often have the following scenario. After the `fork`, both the parent and the child have something to do. For example, the parent could update a record in a log file with the child's process ID, and the child might have to create a file for the parent. In this example, we require that each process tell the other when it has finished its initial set of operations, and that each wait for the other to complete, before heading off on its own. The following code illustrates this scenario:

```
#include  "apue.h"

TELL_WAIT();    /* set things up for TELL_xxx & WAIT_xxx */

if ((pid = fork()) < 0) {
    err_sys("fork error");
} else if (pid == 0) {            /* child */

    /* child does whatever is necessary ... */

    TELL_PARENT(getppid());     /* tell parent we're done */
    WAIT_PARENT();              /* and wait for parent */

    /* and the child continues on its way ... */

    exit(0);
}

/* parent does whatever is necessary ... */

TELL_CHILD(pid);            /* tell child we're done */
WAIT_CHILD();               /* and wait for child */

/* and the parent continues on its way ... */

exit(0);
```

We assume that the header `apue.h` defines whatever variables are required. The five routines `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT`, and `WAIT_CHILD` can be either macros or functions.

We'll show various ways to implement these `TELL` and `WAIT` routines in later chapters: Section 10.16 shows an implementation using signals; Figure 15.7 shows an implementation using pipes. Let's look at an example that uses these five routines.

Example

The program in Figure 8.12 outputs two strings: one from the child and one from the parent. The program contains a race condition because the output depends on the order in which the processes are run by the kernel and the length of time for which each process runs.

```
#include "apue.h"

static void charatatime(char *);

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
    }
    exit(0);
}

static void
charatatime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

Figure 8.12 Program with a race condition

We set the standard output unbuffered, so every character output generates a `write`. The goal in this example is to allow the kernel to switch between the two processes as often as possible to demonstrate the race condition. (If we didn't do this, we might never see the type of output that follows. Not seeing the erroneous output doesn't

mean that the race condition doesn't exist; it simply means that we can't see it on this particular system.) The following actual output shows how the results can vary:

```
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
output from child
output from parent
```

We need to change the program in Figure 8.12 to use the `TELL` and `WAIT` functions. The program in Figure 8.13 does this. The lines preceded by a plus sign are new lines.

```
  #include "apue.h"

  static void charatatime(char *);

  int
  main(void)
  {
      pid_t   pid;

+     TELL_WAIT();
+
      if ((pid = fork()) < 0) {
          err_sys("fork error");
      } else if (pid == 0) {
+         WAIT_PARENT();      /* parent goes first */
          charatatime("output from child\n");
      } else {
          charatatime("output from parent\n");
+         TELL_CHILD(pid);
      }
      exit(0);
  }

  static void
  charatatime(char *str)
  {
      char    *ptr;
      int     c;

      setbuf(stdout, NULL);           /* set unbuffered */
      for (ptr = str; (c = *ptr++) != 0; )
          putc(c, stdout);
  }
```

Figure 8.13 Modification of Figure 8.12 to avoid race condition

When we run this program, the output is as we expect; there is no intermixing of output from the two processes.

In the program shown in Figure 8.13, the parent goes first. The child goes first if we change the lines following the fork to be

```
} else if (pid == 0) {
    charatatime("output from child\n");
    TELL_PARENT(getppid());
} else {
    WAIT_CHILD();        /* child goes first */
    charatatime("output from parent\n");
}
```

Exercise 8.4 continues this example. □

8.10 exec Functions

We mentioned in Section 8.3 that one use of the fork function is to create a new process (the child) that then causes another program to be executed by calling one of the exec functions. When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function. The process ID does not change across an exec, because a new process is not created; exec merely replaces the current process—its text, data, heap, and stack segments—with a brand-new program from disk.

There are seven different exec functions, but we'll often simply refer to "the exec function," which means that we could use any of the seven functions. These seven functions round out the UNIX System process control primitives. With fork, we can create new processes; and with the exec functions, we can initiate new programs. The exit function and the wait functions handle termination and waiting for termination. These are the only process control primitives we need. We'll use these primitives in later sections to build additional functions, such as popen and system.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv[]);

int fexecve(int fd, char *const argv[], char *const envp[]);
```

All seven return: –1 on error, no return on success

The first difference in these functions is that the first four take a pathname argument, the next two take a filename argument, and the last one takes a file descriptor argument. When a *filename* argument is specified,

- If *filename* contains a slash, it is taken as a pathname.
- Otherwise, the executable file is searched for in the directories specified by the `PATH` environment variable.

The `PATH` variable contains a list of directories, called path prefixes, that are separated by colons. For example, the *name=value* environment string

```
PATH=/bin:/usr/bin:/usr/local/bin/:.
```

specifies four directories to search. The last path prefix specifies the current directory. (A zero-length prefix also means the current directory. It can be specified as a colon at the beginning of the *value*, two colons in a row, or a colon at the end of the *value*.)

> There are security reasons for *never* including the current directory in the search path. See Garfinkel et al. [2003].

If either `execlp` or `execvp` finds an executable file using one of the path prefixes, but the file isn't a machine executable that was generated by the link editor, the function assumes that the file is a shell script and tries to invoke `/bin/sh` with the *filename* as input to the shell.

With `fexecve`, we avoid the issue of finding the correct executable file altogether and rely on the caller to do this. By using a file descriptor, the caller can verify the file is in fact the intended file and execute it without a race. Otherwise, a malicious user with appropriate privileges could replace the executable file (or a portion of the path to the executable file) after it has been located and verified, but before the caller can execute it (recall the discussion of TOCTTOU errors in Section 3.3).

The next difference concerns the passing of the argument list (`l` stands for list and `v` stands for vector). The functions `execl`, `execlp`, and `execle` require each of the command-line arguments to the new program to be specified as separate arguments. We mark the end of the arguments with a null pointer. For the other four functions (`execv`, `execvp`, `execve`, and `fexecve`), we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.

Before using ISO C prototypes, the normal way to show the command-line arguments for the three functions `execl`, `execle`, and `execlp` was

```
char *arg0, char *arg1, ..., char *argn, (char *)0
```

This syntax explicitly shows that the final command-line argument is followed by a null pointer. If this null pointer is specified by the constant `0`, we must cast it to a pointer; if we don't, it's interpreted as an integer argument. If the size of an integer is different from the size of a `char *`, the actual arguments to the `exec` function will be wrong.

The final difference is the passing of the environment list to the new program. The three functions whose names end in an `e` (`execle`, `execve`, and `fexecve`) allow us to pass a pointer to an array of pointers to the environment strings. The other four

functions, however, use the environ variable in the calling process to copy the existing environment for the new program. (Recall our discussion of the environment strings in Section 7.9 and Figure 7.8. We mentioned that if the system supported such functions as setenv and putenv, we could change the current environment and the environment of any subsequent child processes, but we couldn't affect the environment of the parent process.) Normally, a process allows its environment to be propagated to its children, but in some cases, a process wants to specify a certain environment for a child. One example of the latter is the login program when a new login shell is initiated. Normally, login creates a specific environment with only a few variables defined and lets us, through the shell start-up file, add variables to the environment when we log in.

Before using ISO C prototypes, the arguments to execle were shown as

```
char *pathname, char *arg0, ..., char *argn, (char *)0, char *envp[]
```

This syntax specifically shows that the final argument is the address of the array of character pointers to the environment strings. The ISO C prototype doesn't show this, as all the command-line arguments, the null pointer, and the *envp* pointer are shown with the ellipsis notation (...).

The arguments for these seven exec functions are difficult to remember. The letters in the function names help somewhat. The letter p means that the function takes a *filename* argument and uses the PATH environment variable to find the executable file. The letter l means that the function takes a list of arguments and is mutually exclusive with the letter v, which means that it takes an *argv*[] vector. Finally, the letter e means that the function takes an *envp*[] array instead of using the current environment. Figure 8.14 shows the differences among these seven functions.

Function	*pathname*	*filename*	*fd*	Arg list	*argv*[]	environ	*envp*[]
execl	•			•		•	
execlp		•		•		•	
execle	•			•			•
execv	•				•	•	
execvp		•			•	•	
execve	•				•		•
fexecve			•		•		•
(letter in name)		p	f	l	v		e

Figure 8.14 Differences among the seven exec functions

Every system has a limit on the total size of the argument list and the environment list. From Section 2.5.2 and Figure 2.8, this limit is given by ARG_MAX. This value must be at least 4,096 bytes on a POSIX.1 system. We sometimes encounter this limit when using the shell's filename expansion feature to generate a list of filenames. On some systems, for example, the command

```
grep getrlimit /usr/share/man/*/*
```

can generate a shell error of the form

```
Argument list too long
```

> Historically, the limit in older System V implementations was 5,120 bytes. Older BSD systems had a limit of 20,480 bytes. The limit in current systems is much higher. (See the output from the program in Figure 2.14, which is summarized in Figure 2.15.)

To get around the limitation in argument list size, we can use the `xargs`(1) command to break up long argument lists. To look for all the occurrences of `getrlimit` in the man pages on our system, we could use

```
find /usr/share/man -type f -print | xargs grep getrlimit
```

If the man pages on our system are compressed, however, we could try

```
find /usr/share/man -type f -print | xargs bzgrep getrlimit
```

We use the `type -f` option to the `find` command to restrict the list so that it contains only regular files, because the `grep` commands can't search for patterns in directories, and we want to avoid unnecessary error messages.

We've mentioned that the process ID does not change after an `exec`, but the new program inherits additional properties from the calling process:

- Process ID and parent process ID
- Real user ID and real group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- Time left until alarm clock
- Current working directory
- Root directory
- File mode creation mask
- File locks
- Process signal mask
- Pending signals
- Resource limits
- Nice value (on XSI-conformant systems; see Section 8.16)
- Values for `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime`

The handling of open files depends on the value of the close-on-exec flag for each descriptor. Recall from Figure 3.7 and our mention of the `FD_CLOEXEC` flag in Section 3.14 that every open descriptor in a process has a close-on-exec flag. If this flag is set, the descriptor is closed across an `exec`. Otherwise, the descriptor is left open across the `exec`. The default is to leave the descriptor open across the `exec` unless we specifically set the close-on-exec flag using `fcntl`.

POSIX.1 specifically requires that open directory streams (recall the `opendir`

function from Section 4.22) be closed across an exec. This is normally done by the opendir function calling fcntl to set the close-on-exec flag for the descriptor corresponding to the open directory stream.

Note that the real user ID and the real group ID remain the same across the exec, but the effective IDs can change, depending on the status of the set-user-ID and the set-group-ID bits for the program file that is executed. If the set-user-ID bit is set for the new program, the effective user ID becomes the owner ID of the program file. Otherwise, the effective user ID is not changed (it's not set to the real user ID). The group ID is handled in the same way.

In many UNIX system implementations, only one of these seven functions, execve, is a system call within the kernel. The other six are just library functions that eventually invoke this system call. We can illustrate the relationship among these seven functions as shown in Figure 8.15.

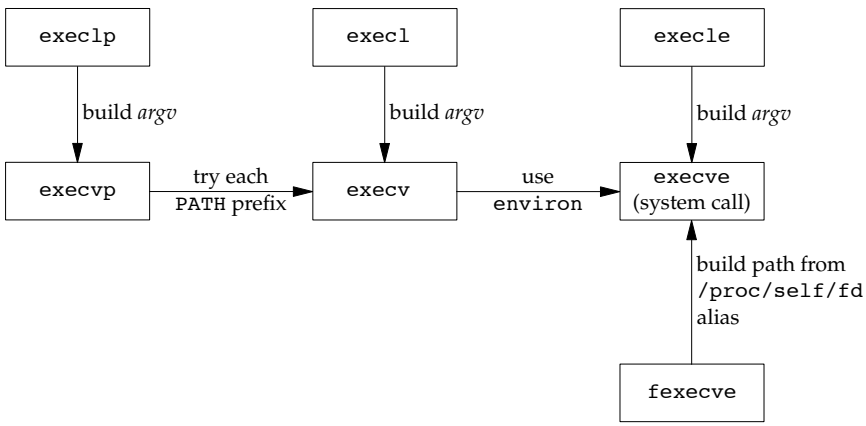


Figure 8.15 Relationship of the seven exec functions

In this arrangement, the library functions execlp and execvp process the PATH environment variable, looking for the first path prefix that contains an executable file named *filename*. The fexecve library function uses /proc to convert the file descriptor argument into a pathname that can be used by execve to execute the program.

> This describes how fexecve is implemented in FreeBSD 8.0 and Linux 3.2.0. Other systems might take a different approach. For example, a system without /proc or /dev/fd could implement fexecve as a system call veneer that translates the file descriptor argument into an i-node pointer, implement execve as a system call veneer that translates the pathname argument into an i-node pointer, and place all the rest of the exec code common to both execve and fexecve in a separate function to be called with an i-node pointer for the file to be executed.

Example

The program in Figure 8.16 demonstrates the exec functions.

```
#include "apue.h"
#include <sys/wait.h>

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify pathname, specify environment */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                "MY ARG2", (char *)0, env_init) < 0)
            err_sys("execle error");
    }

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
            err_sys("execlp error");
    }

    exit(0);
}
```

Figure 8.16 Example of exec functions

We first call `execle`, which requires a pathname and a specific environment. The next call is to `execlp`, which uses a filename and passes the caller's environment to the new program. The only reason the call to `execlp` works is that the directory `/home/sar/bin` is one of the current path prefixes. Note also that we set the first argument, `argv[0]` in the new program, to be the filename component of the pathname. Some shells set this argument to be the complete pathname. This is a convention only; we can set `argv[0]` to any string we like. The `login` command does this when it executes the shell. Before executing the shell, `login` adds a dash as a prefix to `argv[0]` to indicate to the shell that it is being invoked as a login shell. A login shell will execute the start-up profile commands, whereas a nonlogin shell will not.

The program `echoall` that is executed twice in the program in Figure 8.16 is shown in Figure 8.17. It is a trivial program that echoes all its command-line arguments and its entire environment list.

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int         i;
    char        **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++)   /* and all env strings */
        printf("%s\n", *ptr);

    exit(0);
}
```

Figure 8.17 Echo all command-line arguments and all environment strings

When we execute the program from Figure 8.16, we get

```
$ ./a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
$ argv[0]: echoall
argv[1]: only 1 arg
USER=sar
LOGNAME=sar
SHELL=/bin/bash
                                    47 more lines that aren't shown
HOME=/home/sar
```

Note that the shell prompt appeared before the printing of `argv[0]` from the second `exec`. This occurred because the parent did not `wait` for this child process to finish. □

8.11 Changing User IDs and Group IDs

In the UNIX System, privileges, such as being able to change the system's notion of the current date, and access control, such as being able to read or write a particular file, are based on user and group IDs. When our programs need additional privileges or need to gain access to resources that they currently aren't allowed to access, they need to change their user or group ID to an ID that has the appropriate privilege or access. Similarly, when our programs need to lower their privileges or prevent access to certain resources, they do so by changing either their user ID or group ID to an ID without the privilege or ability access to the resource.

In general, we try to use the *least-privilege* model when we design our applications. According to this model, our programs should use the least privilege necessary to accomplish any given task. This reduces the risk that security might be compromised by a malicious user trying to trick our programs into using their privileges in unintended ways.

We can set the real user ID and effective user ID with the `setuid` function. Similarly, we can set the real group ID and the effective group ID with the `setgid` function.

```
#include <unistd.h>

int setuid(uid_t uid);

int setgid(gid_t gid);
```

Both return: 0 if OK, –1 on error

There are rules for who can change the IDs. Let's consider only the user ID for now. (Everything we describe for the user ID also applies to the group ID.)

1. If the process has superuser privileges, the `setuid` function sets the real user ID, effective user ID, and saved set-user-ID to *uid*.
2. If the process does not have superuser privileges, but *uid* equals either the real user ID or the saved set-user-ID, `setuid` sets only the effective user ID to *uid*. The real user ID and the saved set-user-ID are not changed.
3. If neither of these two conditions is true, `errno` is set to `EPERM` and –1 is returned.

Here, we are assuming that `_POSIX_SAVED_IDS` is true. If this feature isn't provided, then delete all preceding references to the saved set-user-ID.

> The saved IDs are a mandatory feature in the 2001 version of POSIX.1. They were optional in older versions of POSIX. To see whether an implementation supports this feature, an application can test for the constant `_POSIX_SAVED_IDS` at compile time or call `sysconf` with the `_SC_SAVED_IDS` argument at runtime.

We can make a few statements about the three user IDs that the kernel maintains.

1. Only a superuser process can change the real user ID. Normally, the real user ID is set by the `login`(1) program when we log in and never changes. Because `login` is a superuser process, it sets all three user IDs when it calls `setuid`.
2. The effective user ID is set by the `exec` functions only if the set-user-ID bit is set for the program file. If the set-user-ID bit is not set, the `exec` functions leave the effective user ID as its current value. We can call `setuid` at any time to set the effective user ID to either the real user ID or the saved set-user-ID. Naturally, we can't set the effective user ID to any random value.
3. The saved set-user-ID is copied from the effective user ID by `exec`. If the file's set-user-ID bit is set, this copy is saved after `exec` stores the effective user ID from the file's user ID.

Figure 8.18 summarizes the various ways these three user IDs can be changed.

ID	exec		setuid(*uid*)	
	set-user-ID bit off	set-user-ID bit on	superuser	unprivileged user
real user ID	unchanged	unchanged	set to *uid*	unchanged
effective user ID	unchanged	set from user ID of program file	set to *uid*	set to *uid*
saved set-user ID	copied from effective user ID	copied from effective user ID	set to *uid*	unchanged

Figure 8.18 Ways to change the three user IDs

Note that we can obtain only the current value of the real user ID and the effective user ID with the functions `getuid` and `geteuid` from Section 8.2. We have no portable way to obtain the current value of the saved set-user-ID.

> FreeBSD 8.0 and LINUX 3.2.0 provide the `getresuid` and `getresgid` functions, which can be used to get the saved set-user-ID and saved set-group-ID, respectively.

`setreuid` and `setregid` Functions

Historically, BSD supported the swapping of the real user ID and the effective user ID with the `setreuid` function.

```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);

int setregid(gid_t rgid, gid_t egid);
```

Both return: 0 if OK, –1 on error

We can supply a value of –1 for any of the arguments to indicate that the corresponding ID should remain unchanged.

The rule is simple: an unprivileged user can always swap between the real user ID and the effective user ID. This allows a set-user-ID program to swap to the user's normal permissions and swap back again later for set-user-ID operations. When the saved set-user-ID feature was introduced with POSIX.1, the rule was enhanced to also allow an unprivileged user to set its effective user ID to its saved set-user-ID.

> Both `setreuid` and `setregid` are included in the XSI option in POSIX.1. As such, all UNIX System implementations are expected to provide support for them.
>
> 4.3BSD didn't have the saved set-user-ID feature described earlier; it used `setreuid` and `setregid` instead. This allowed an unprivileged user to swap back and forth between the two values. Be aware, however, that when programs that used this feature spawned a shell, they had to set the real user ID to the normal user ID before the `exec`. If they didn't do this, the real user ID could be privileged (from the swap done by `setreuid`) and the shell process could call `setreuid` to swap the two and assume the permissions of the more privileged user. As a defensive programming measure to solve this problem, programs set both the real user ID and the effective user ID to the normal user ID before the call to `exec` in the child.

`seteuid` and `setegid` Functions

POSIX.1 includes the two functions `seteuid` and `setegid`. These functions are similar to `setuid` and `setgid`, but only the effective user ID or effective group ID is changed.

```
#include <unistd.h>
int seteuid(uid_t uid);
int setegid(gid_t gid);
                                        Both return: 0 if OK, –1 on error
```

An unprivileged user can set its effective user ID to either its real user ID or its saved set-user-ID. For a privileged user, only the effective user ID is set to *uid*. (This behavior differs from that of the `setuid` function, which changes all three user IDs.)

Figure 8.19 summarizes all the functions that we've described here that modify the three user IDs.

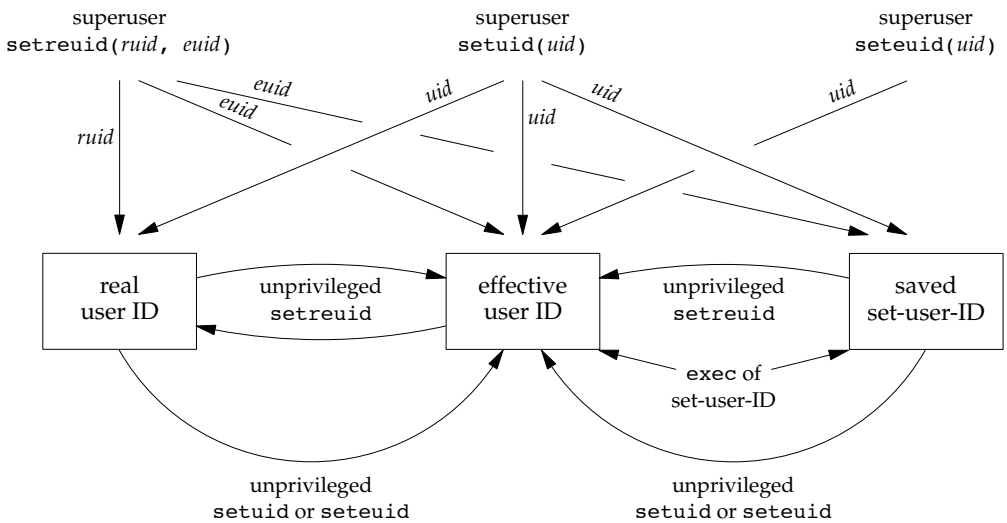


Figure 8.19 Summary of all the functions that set the various user IDs

Group IDs

Everything that we've said so far in this section also applies in a similar fashion to group IDs. The supplementary group IDs are not affected by `setgid`, `setregid`, or `setegid`.

Example

To see the utility of the saved set-user-ID feature, let's examine the operation of a program that uses it. We'll look at the `at`(1) program, which we can use to schedule commands to be run at some time in the future.

> On Linux 3.2.0, the `at` program is installed set-user-ID to user `daemon`. On FreeBSD 8.0, Mac OS X 10.6.8, and Solaris 10, the `at` program is installed set-user-ID to user `root`. This allows the `at` command to write privileged files owned by the daemon that will run the commands on behalf of the user running the `at` command. On Linux 3.2.0, the programs are run by the `atd`(8) daemon. On FreeBSD 8.0 and Solaris 10, the programs are run by the `cron`(1M) daemon. On Mac OS X 10.6.8, the programs are run by the `launchd`(8) daemon.

To prevent being tricked into running commands that we aren't allowed to run, or reading or writing files that we aren't allowed to access, the `at` command and the daemon that ultimately runs the commands on our behalf have to switch between sets of privileges: ours and those of the daemon. The following steps take place.

1. Assuming that the `at` program file is owned by `root` and has its set-user-ID bit set, when we run it, we have

   real user ID = our user ID (unchanged)
   effective user ID = `root`
   saved set-user-ID = `root`

2. The first thing the `at` command does is reduce its privileges so that it runs with our privileges. It calls the `seteuid` function to set the effective user ID to our real user ID. After this, we have

   real user ID = our user ID (unchanged)
   effective user ID = our user ID
   saved set-user-ID = `root` (unchanged)

3. The `at` program runs with our privileges until it needs to access the configuration files that control which commands are to be run and the time at which they need to run. These files are owned by the daemon that will run the commands for us. The `at` command calls `seteuid` to set the effective user ID to `root`. This call is allowed because the argument to `seteuid` equals the saved set-user-ID. (This is why we need the saved set-user-ID.) After this, we have

   real user ID = our user ID (unchanged)
   effective user ID = `root`
   saved set-user-ID = `root` (unchanged)

   Because the effective user ID is `root`, file access is allowed.

4. After the files are modified to record the commands to be run and the time at which they are to be run, the `at` command lowers its privileges by calling

`seteuid` to set its effective user ID to our user ID. This prevents any accidental misuse of privilege. At this point, we have

```
real user ID = our user ID (unchanged)
effective user ID = our user ID
saved set-user-ID = root (unchanged)
```

5. The daemon starts out running with `root` privileges. To run commands on our behalf, the daemon calls `fork` and the child calls `setuid` to change its user ID to our user ID. Because the child is running with `root` privileges, this changes all of the IDs. We have

```
real user ID = our user ID
effective user ID = our user ID
saved set-user-ID = our user ID
```

   Now the daemon can safely execute commands on our behalf, because it can access only the files to which we normally have access. We have no additional permissions.

By using the saved set-user-ID in this fashion, we can use the extra privileges granted to us by the set-user-ID of the program file only when we need elevated privileges. Any other time, however, the process runs with our normal permissions. If we weren't able to switch back to the saved set-user-ID at the end, we might be tempted to retain the extra permissions the whole time we were running (which is asking for trouble). □

8.12 Interpreter Files

All contemporary UNIX systems support interpreter files. These files are text files that begin with a line of the form

```
#! pathname [ optional-argument ]
```

The space between the exclamation point and the *pathname* is optional. The most common of these interpreter files begin with the line

```
#!/bin/sh
```

The *pathname* is normally an absolute pathname, since no special operations are performed on it (i.e., `PATH` is not used). The recognition of these files is done within the kernel as part of processing the `exec` system call. The actual file that gets executed by the kernel is not the interpreter file, but rather the file specified by the *pathname* on the first line of the interpreter file. Be sure to differentiate between the interpreter file—a text file that begins with `#!`—and the interpreter, which is specified by the *pathname* on the first line of the interpreter file.

Be aware that systems place a size limit on the first line of an interpreter file. This limit includes the `#!`, the *pathname*, the optional argument, the terminating newline, and any spaces.

> On FreeBSD 8.0, this limit is 4,097 bytes. On Linux 3.2.0, the limit is 128 bytes. Mac OS X 10.6.8 supports a limit of 513 bytes, whereas Solaris 10 places the limit at 1,024 bytes.

Example

Let's look at an example to see what the kernel does with the arguments to the exec function when the file being executed is an interpreter file and the optional argument on the first line of the interpreter file. The program in Figure 8.20 execs an interpreter file.

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {          /* child */
        if (execl("/home/sar/bin/testinterp",
                  "testinterp", "myarg1", "MY ARG2", (char *)0) < 0)
            err_sys("execl error");
    }
    if (waitpid(pid, NULL, 0) < 0)  /* parent */
        err_sys("waitpid error");
    exit(0);
}
```

Figure 8.20 A program that execs an interpreter file

The following shows the contents of the one-line interpreter file that is executed and the result from running the program in Figure 8.20:

```
$ cat /home/sar/bin/testinterp
#!/home/sar/bin/echoarg foo
$ ./a.out
argv[0]: /home/sar/bin/echoarg
argv[1]: foo
argv[2]: /home/sar/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

The program echoarg (the interpreter) just echoes each of its command-line arguments. (This is the program from Figure 7.4.) Note that when the kernel execs the interpreter (/home/sar/bin/echoarg), argv[0] is the *pathname* of the interpreter, argv[1] is the optional argument from the interpreter file, and the remaining arguments are the *pathname* (/home/sar/bin/testinterp) and the second and third arguments from the call to execl in the program shown in Figure 8.20 (myarg1 and MY ARG2). Both argv[1] and argv[2] from the call to execl have been shifted right two positions. Note that the kernel takes the *pathname* from the execl call instead of the first argument (testinterp), on the assumption that the *pathname* might contain more information than the first argument. □

Example

A common use for the optional argument following the interpreter *pathname* is to specify the `-f` option for programs that support this option. For example, an `awk`(1) program can be executed as

```
awk -f myfile
```

which tells `awk` to read the `awk` program from the file `myfile`.

> Systems derived from UNIX System V often include two versions of the `awk` language. On these systems, `awk` is often called "old `awk`" and corresponds to the original version distributed with Version 7. In contrast, `nawk` (new `awk`) contains numerous enhancements and corresponds to the language described in Aho, Kernighan, and Weinberger [1988]. This newer version provides access to the command-line arguments, which we need for the example that follows. Solaris 10 provides both versions.
>
> The `awk` program is one of the utilities included by POSIX in its 1003.2 standard, which is now part of the base POSIX.1 specification in the Single UNIX Specification. This utility is also based on the language described in Aho, Kernighan, and Weinberger [1988].
>
> The version of `awk` in Mac OS X 10.6.8 is based on the Bell Laboratories version, which has been placed in the public domain. FreeBSD 8.0 and some Linux distributions ship with GNU `awk`, called `gawk`, which is linked to the name `awk`. `gawk` conforms to the POSIX standard, but also includes other extensions. Because they are more up-to-date, `gawk` and the version of `awk` from Bell Laboratories are preferred to either `nawk` or old `awk`. (The Bell Labs version of `awk` is available at `http://cm.bell-labs.com/cm/cs/awkbook/index.html`.)

Using the `-f` option with an interpreter file lets us write

```
#!/bin/awk -f
(awk program follows in the interpreter file)
```

For example, Figure 8.21 shows `/usr/local/bin/awkexample` (an interpreter file).

```
#!/usr/bin/awk -f
# Note: on Solaris, use nawk instead
BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}
```

Figure 8.21 An awk program as an interpreter file

If one of the path prefixes is `/usr/local/bin`, we can execute the program in Figure 8.21 (assuming that we've turned on the execute bit for the file) as

```
$ awkexample file1 FILENAME2 f3
ARGV[0] = awk
ARGV[1] = file1
ARGV[2] = FILENAME2
ARGV[3] = f3
```

When `/bin/awk` is executed, its command-line arguments are

```
/bin/awk -f /usr/local/bin/awkexample file1 FILENAME2 f3
```

The pathname of the interpreter file (`/usr/local/bin/awkexample`) is passed to the interpreter. The filename portion of this pathname (what we typed to the shell) isn't adequate, because the interpreter (`/bin/awk` in this example) can't be expected to use the `PATH` variable to locate files. When it reads the interpreter file, `awk` ignores the first line, since the pound sign is `awk`'s comment character.

We can verify these command-line arguments with the following commands:

```
$ /bin/su                                        become superuser
Password:                                        enter superuser password
# mv /usr/bin/awk /usr/bin/awk.save              save the original program
# cp /home/sar/bin/echoarg /usr/bin/awk          and replace it temporarily
# suspend                                        suspend the superuser shell
[1] + Stopped          /bin/su                   using job control
$ awkexample file1 FILENAME2 f3
argv[0]: /bin/awk
argv[1]: -f
argv[2]: /usr/local/bin/awkexample
argv[3]: file1
argv[4]: FILENAME2
argv[5]: f3
$ fg                                             resume superuser shell using job control
/bin/su
# mv /usr/bin/awk.save /usr/bin/awk              restore the original program
# exit                                           and exit the superuser shell
```

In this example, the `-f` option for the interpreter is required. As we said, this tells `awk` where to look for the `awk` program. If we remove the `-f` option from the interpreter file, an error message usually results when we try to run it. The exact text of the message varies, depending on where the interpreter file is stored and whether the remaining arguments represent existing files. This is because the command-line arguments in this case are

```
/bin/awk /usr/local/bin/awkexample file1 FILENAME2 f3
```

and `awk` is trying to interpret the string `/usr/local/bin/awkexample` as an `awk` program. If we couldn't pass at least a single optional argument to the interpreter (`-f` in this case), these interpreter files would be usable only with the shells. □

Are interpreter files required? Not really. They provide an efficiency gain for the user at some expense in the kernel (since it's the kernel that recognizes these files). Interpreter files are useful for the following reasons.

1. They hide that certain programs are scripts in some other language. For example, to execute the program in Figure 8.21, we just say

   `awkexample` *optional-arguments*

instead of needing to know that the program is really an `awk` script that we would otherwise have to execute as

```
awk -f awkexample optional-arguments
```

2. Interpreter scripts provide an efficiency gain. Consider the previous example again. We could still hide that the program is an `awk` script, by wrapping it in a shell script:

```
awk 'BEGIN {
    for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
    exit
}' $*
```

The problem with this solution is that more work is required. First, the shell reads the command and tries to `execlp` the filename. Because the shell script is an executable file but isn't a machine executable, an error is returned and `execlp` assumes that the file is a shell script (which it is). Then `/bin/sh` is executed with the pathname of the shell script as its argument. The shell correctly runs our script, but to run the `awk` program, the shell does a `fork`, `exec`, and `wait`. Thus there is more overhead involved in replacing an interpreter script with a shell script.

3. Interpreter scripts let us write shell scripts using shells other than `/bin/sh`. When it finds an executable file that isn't a machine executable, `execlp` has to choose a shell to invoke, and it always uses `/bin/sh`. Using an interpreter script, however, we can simply write

```
#!/bin/csh
(C shell script follows in the interpreter file)
```

Again, we could wrap all of this in a `/bin/sh` script (that invokes the C shell), as we described earlier, but more overhead is required.

None of this would work as we've shown here if the three shells and `awk` didn't use the pound sign as their comment character.

8.13 `system` Function

It is convenient to execute a command string from within a program. For example, assume that we want to put a time-and-date stamp into a certain file. We could use the functions described in Section 6.10 to do this: call `time` to get the current calendar time, then call `localtime` to convert it to a broken-down time, then call `strftime` to format the result, and finally write the result to the file. It is much easier, however, to say

```
system("date > file");
```

ISO C defines the `system` function, but its operation is strongly system dependent. POSIX.1 includes the `system` interface, expanding on the ISO C definition to describe its behavior in a POSIX environment.

```
#include <stdlib.h>

int system(const char *cmdstring);

                                                    Returns: (see below)
```

If *cmdstring* is a null pointer, `system` returns nonzero only if a command processor is available. This feature determines whether the `system` function is supported on a given operating system. Under the UNIX System, `system` is always available.

Because `system` is implemented by calling `fork`, `exec`, and `waitpid`, there are three types of return values.

1. If either the `fork` fails or `waitpid` returns an error other than `EINTR`, `system` returns –1 with `errno` set to indicate the error.
2. If the `exec` fails, implying that the shell can't be executed, the return value is as if the shell had executed `exit(127)`.
3. Otherwise, all three functions—`fork`, `exec`, and `waitpid`—succeed, and the return value from `system` is the termination status of the shell, in the format specified for `waitpid`.

> Some older implementations of `system` returned an error (`EINTR`) if `waitpid` was interrupted by a caught signal. Because there is no strategy that an application can use to recover from this type of error (the process ID of the child is hidden from the caller), POSIX later added the requirement that `system` not return an error in this case. (We discuss interrupted system calls in Section 10.5.)

Figure 8.22 shows an implementation of the `system` function. The one feature that it doesn't handle is signals. We'll update this function with signal handling in Section 10.18.

The shell's `-c` option tells it to take the next command-line argument—*cmdstring*, in this case—as its command input instead of reading from standard input or from a given file. The shell parses this null-terminated C string and breaks it up into separate command-line arguments for the command. The actual command string that is passed to the shell can contain any valid shell commands. For example, input and output redirection using < and > can be used.

If we didn't use the shell to execute the command, but tried to execute the command ourself, it would be more difficult. First, we would want to call `execlp`, instead of `execl`, to use the `PATH` variable, like the shell. We would also have to break up the null-terminated C string into separate command-line arguments for the call to `execlp`. Finally, we wouldn't be able to use any of the shell metacharacters.

Note that we call `_exit` instead of `exit`. We do this to prevent any standard I/O buffers, which would have been copied from the parent to the child across the `fork`, from being flushed in the child.

```
#include    <sys/wait.h>
#include    <errno.h>
#include    <unistd.h>

int
system(const char *cmdstring)   /* version without signal handling */
{
    pid_t   pid;
    int     status;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with UNIX */

    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {              /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* execl error */
    } else {                            /* parent */
        while (waitpid(pid, &status, 0) < 0) {
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
        }
    }

    return(status);
}
```

Figure 8.22 The `system` function, without signal handling

We can test this version of `system` with the program shown in Figure 8.23. (The `pr_exit` function was defined in Figure 8.5.) Running the program in Figure 8.23 gives us

```
$ ./a.out
Sat Feb 25 19:36:59 EST 2012
normal termination, exit status = 0     for date
sh: nosuchcommand: command not found
normal termination, exit status = 127   for nosuchcommand
sar      console  Jan  1 14:59
sar      ttys000  Feb  7 19:08
sar      ttys001  Jan 15 15:28
sar      ttys002  Jan 15 21:50
sar      ttys003  Jan 21 16:02
normal termination, exit status = 44    for exit
```

The advantage in using `system`, instead of using `fork` and `exec` directly, is that `system` does all the required error handling and (in our next version of this function in Section 10.18) all the required signal handling.

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    int     status;

    if ((status = system("date")) < 0)
        err_sys("system() error");

    pr_exit(status);

    if ((status = system("nosuchcommand")) < 0)
        err_sys("system() error");

    pr_exit(status);

    if ((status = system("who; exit 44")) < 0)
        err_sys("system() error");

    pr_exit(status);

    exit(0);
}
```

Figure 8.23 Calling the system function

Earlier systems, including SVR3.2 and 4.3BSD, didn't have the waitpid function available. Instead, the parent waited for the child, using a statement such as

```
while ((lastpid = wait(&status)) != pid && lastpid != -1)
    ;
```

A problem occurs if the process that calls system has spawned its own children before calling system. Because the while statement above keeps looping until the child that was generated by system terminates, if any children of the process terminate before the process identified by pid, then the process ID and termination status of these other children are discarded by the while statement. Indeed, this inability to wait for a specific child is one of the reasons given in the POSIX.1 Rationale for including the waitpid function. We'll see in Section 15.3 that the same problem occurs with the popen and pclose functions if the system doesn't provide a waitpid function.

Set-User-ID Programs

What happens if we call system from a set-user-ID program? Doing so creates a security hole and should never be attempted. Figure 8.24 shows a simple program that just calls system for its command-line argument.

```
#include "apue.h"

int
main(int argc, char *argv[])
{
    int     status;

    if (argc < 2)
        err_quit("command-line argument required");

    if ((status = system(argv[1])) < 0)
        err_sys("system() error");

    pr_exit(status);

    exit(0);
}
```

Figure 8.24 Execute the command-line argument using `system`

We'll compile this program into the executable file `tsys`.

Figure 8.25 shows another simple program that prints its real and effective user IDs.

```
#include "apue.h"

int
main(void)
{
    printf("real uid = %d, effective uid = %d\n", getuid(), geteuid());
    exit(0);
}
```

Figure 8.25 Print real and effective user IDs

We'll compile this program into the executable file `printuids`. Running both programs gives us the following:

```
$ tsys printuids                        normal execution, no special privileges
real uid = 205, effective uid = 205
normal termination, exit status = 0
$ su                                    become superuser
Password:                               enter superuser password
# chown root tsys                       change owner
# chmod u+s tsys                        make set-user-ID
# ls -l tsys                            verify file's permissions and owner
-rwsrwxr-x  1 root    7888 Feb 25 22:13 tsys
# exit                                  leave superuser shell
$ tsys printuids
real uid = 205, effective uid = 0       oops, this is a security hole
normal termination, exit status = 0
```

The superuser permissions that we gave the `tsys` program are retained across the `fork` and `exec` that are done by `system`.

> Some implementations have closed this security hole by changing `/bin/sh` to reset the effective user ID to the real user ID when they don't match. On these systems, the previous example doesn't work as shown. Instead, the same effective user ID will be printed regardless of the status of the set-user-ID bit on the program calling `system`.

If it is running with special permissions—either set-user-ID or set-group-ID—and wants to spawn another process, a process should use `fork` and `exec` directly, being certain to change back to normal permissions after the `fork`, before calling `exec`. The `system` function should *never* be used from a set-user-ID or a set-group-ID program.

> One reason for this admonition is that `system` invokes the shell to parse the command string, and the shell uses its `IFS` variable as the input field separator. Older versions of the shell didn't reset this variable to a normal set of characters when invoked. As a result, a malicious user could set `IFS` before `system` was called, causing `system` to execute a different program.

8.14 Process Accounting

Most UNIX systems provide an option to do process accounting. When enabled, the kernel writes an accounting record each time a process terminates. These accounting records typically contain a small amount of binary data with the name of the command, the amount of CPU time used, the user ID and group ID, the starting time, and so on. We'll take a closer look at these accounting records in this section, as it gives us a chance to look at processes again and to use the `fread` function from Section 5.9.

> Process accounting is not specified by any of the standards. Thus all the implementations have annoying differences. For example, the I/O counts maintained on Solaris 10 are in units of bytes, whereas FreeBSD 8.0 and Mac OS X 10.6.8 maintain units of blocks, although there is no distinction between different block sizes, making the counter effectively useless. Linux 3.2.0, on the other hand, doesn't try to maintain I/O statistics at all.
>
> Each implementation also has its own set of administrative commands to process raw accounting data. For example, Solaris provides `runacct`(1m) and `acctcom`(1), whereas FreeBSD provides the `sa`(8) command to process and summarize the raw accounting data.

A function we haven't described (`acct`) enables and disables process accounting. The only use of this function is from the `accton`(8) command (which happens to be one of the few similarities among platforms). A superuser executes `accton` with a pathname argument to enable accounting. The accounting records are written to the specified file, which is usually `/var/account/acct` on FreeBSD and Mac OS X, `/var/log/account/pacct` on Linux, and `/var/adm/pacct` on Solaris. Accounting is turned off by executing `accton` without any arguments.

The structure of the accounting records is defined in the header `<sys/acct.h>`. Although the implementation of each system differs, the accounting records look something like

```
typedef  u_short comp_t;  /* 3-bit base 8 exponent; 13-bit fraction */

struct  acct
{
  char   ac_flag;    /* flag (see Figure 8.26) */
  char   ac_stat;    /* termination status (signal & core flag only) */
                     /* (Solaris only) */
  uid_t  ac_uid;     /* real user ID */
  gid_t  ac_gid;     /* real group ID */
  dev_t  ac_tty;     /* controlling terminal */
  time_t ac_btime;   /* starting calendar time */
  comp_t ac_utime;   /* user CPU time */
  comp_t ac_stime;   /* system CPU time */
  comp_t ac_etime;   /* elapsed time */
  comp_t ac_mem;     /* average memory usage */
  comp_t ac_io;      /* bytes transferred (by read and write) */
                     /* "blocks" on BSD systems */
  comp_t ac_rw;      /* blocks read or written */
                     /* (not present on BSD systems) */
  char   ac_comm[8]; /* command name: [8] for Solaris, */
                     /* [10] for Mac OS X, [16] for FreeBSD, and */
                     /* [17] for Linux */
};
```

Times are recorded in units of clock ticks on most platforms, but FreeBSD stores microseconds instead. The `ac_flag` member records certain events during the execution of the process. These events are described in Figure 8.26.

The data required for the accounting record, such as CPU times and number of characters transferred, is kept by the kernel in the process table and initialized whenever a new process is created, as in the child after a `fork`. Each accounting record is written when the process terminates. This has two consequences.

First, we don't get accounting records for processes that never terminate. Processes like `init` that run for the lifetime of the system don't generate accounting records. This also applies to kernel daemons, which normally don't exit.

Second, the order of the records in the accounting file corresponds to the termination order of the processes, not the order in which they were started. To know the starting order, we would have to go through the accounting file and sort by the starting calendar time. But this isn't perfect, since calendar times are in units of seconds (Section 1.10), and it's possible for many processes to be started in any given second. Alternatively, the elapsed time is given in clock ticks, which are usually between 60 and 128 ticks per second. But we don't know the ending time of a process; all we know is its starting time and ending order. Thus, even though the elapsed time is more accurate than the starting time, we still can't reconstruct the exact starting order of various processes, given the data in the accounting file.

The accounting records correspond to processes, not programs. A new record is initialized by the kernel for the child after a `fork`, not when a new program is executed. Although `exec` doesn't create a new accounting record, the command name changes, and the `AFORK` flag is cleared. This means that if we have a chain of three programs—A

ac_flag	Description	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
AFORK	process is the result of fork, but never called exec	•	•	•	•
ASU	process used superuser privileges		•	•	•
ACORE	process dumped core	•	•	•	
AXSIG	process was killed by a signal	•	•	•	
AEXPND	expanded accounting entry				•
ANVER	new record format	•			

Figure 8.26 Values for ac_flag from accounting record

execs B, then B execs C, and C exits—only a single accounting record is written. The command name in the record corresponds to program C, but the CPU times, for example, are the sum for programs A, B, and C.

Example

To have some accounting data to examine, we'll create a test program to implement the diagram shown in Figure 8.27.

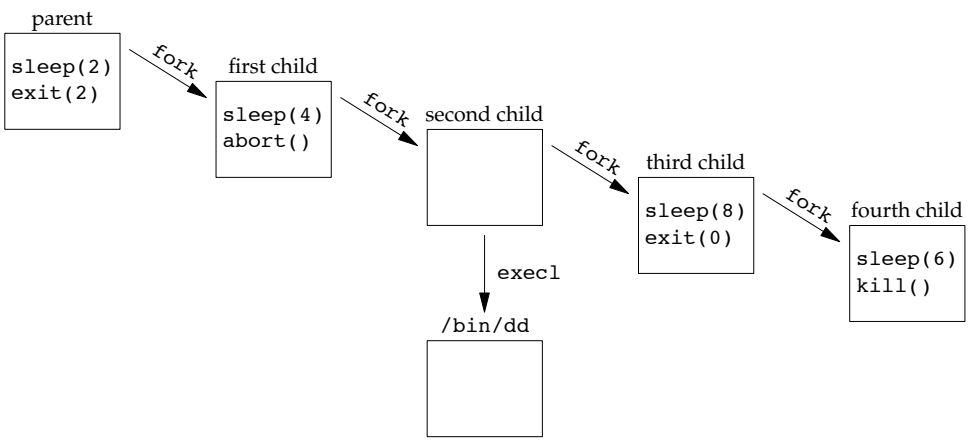


Figure 8.27 Process structure for accounting example

The source for the test program is shown in Figure 8.28. It calls fork four times. Each child does something different and then terminates.

```
#include "apue.h"

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0)
        err_sys("fork error");
```

```
    else if (pid != 0) {        /* parent */
        sleep(2);
        exit(2);                /* terminate with exit status 2 */
    }

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {        /* first child */
        sleep(4);
        abort();                /* terminate with core dump */
    }

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {        /* second child */
        execl("/bin/dd", "dd", "if=/etc/passwd", "of=/dev/null", NULL);
        exit(7);                /* shouldn't get here */
    }

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {        /* third child */
        sleep(8);
        exit(0);                /* normal exit */
    }

    sleep(6);                   /* fourth child */
    kill(getpid(), SIGKILL);    /* terminate w/signal, no core dump */
    exit(6);                    /* shouldn't get here */
}
```

Figure 8.28 Program to generate accounting data

We'll run the test program on Solaris and then use the program in Figure 8.29 to print out selected fields from the accounting records.

```
#include "apue.h"
#include <sys/acct.h>

#if defined(BSD)    /* different structure in FreeBSD */
#define acct acctv2
#define ac_flag ac_trailer.ac_flag
#define FMT "%-*.*s  e = %.0f, chars = %.0f, %c %c %c %c\n"
#elif defined(HAS_AC_STAT)
#define FMT "%-*.*s  e = %6ld, chars = %7ld, stat = %3u: %c %c %c %c\n"
#else
#define FMT "%-*.*s  e = %6ld, chars = %7ld, %c %c %c %c\n"
#endif
#if defined(LINUX)
#define acct acct_v3    /* different structure in Linux */
#endif

#if !defined(HAS_ACORE)
#define ACORE 0
```

```
#endif
#if !defined(HAS_AXSIG)
#define AXSIG 0
#endif

#if !defined(BSD)
static unsigned long
compt2ulong(comp_t comptime)    /* convert comp_t to unsigned long */
{
    unsigned long   val;
    int             exp;

    val = comptime & 0x1fff;    /* 13-bit fraction */
    exp = (comptime >> 13) & 7; /* 3-bit exponent (0-7) */
    while (exp-- > 0)
        val *= 8;
    return(val);
}
#endif

int
main(int argc, char *argv[])
{
    struct acct     acdata;
    FILE            *fp;

    if (argc != 2)
        err_quit("usage: pracct filename");
    if ((fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);
    while (fread(&acdata, sizeof(acdata), 1, fp) == 1) {
        printf(FMT, (int)sizeof(acdata.ac_comm),
            (int)sizeof(acdata.ac_comm), acdata.ac_comm,
#if defined(BSD)
            acdata.ac_etime, acdata.ac_io,
#else
            compt2ulong(acdata.ac_etime), compt2ulong(acdata.ac_io),
#endif
#if defined(HAS_AC_STAT)
            (unsigned char) acdata.ac_stat,
#endif
            acdata.ac_flag & ACORE ? 'D' : ' ',
            acdata.ac_flag & AXSIG ? 'X' : ' ',
            acdata.ac_flag & AFORK ? 'F' : ' ',
            acdata.ac_flag & ASU   ? 'S' : ' ');
    }
    if (ferror(fp))
        err_sys("read error");
    exit(0);
}
```

Figure 8.29 Print selected fields from system's accounting file

BSD-derived platforms don't support the `ac_stat` member, so we define the `HAS_AC_STAT` constant on the platforms that do support this member. Basing the defined symbol on the feature instead of on the platform makes the code read better and allows us to modify the program simply by adding the new definition to our compilation command. The alternative would be to use

```
#if !defined(BSD) && !defined(MACOS)
```

which becomes unwieldy as we port our application to additional platforms.

We define similar constants to determine whether the platform supports the `ACORE` and `AXSIG` accounting flags. We can't use the flag symbols themselves, because on Linux, they are defined as `enum` values, which we can't use in a `#ifdef` expression.

To perform our test, we do the following:

1. Become superuser and enable accounting, with the `accton` command. Note that when this command terminates, accounting should be on; therefore, the first record in the accounting file should be from this command.
2. Exit the superuser shell and run the program in Figure 8.28. This should append six records to the accounting file: one for the superuser shell, one for the test parent, and one for each of the four test children.

   A new process is not created by the `execl` in the second child. There is only a single accounting record for the second child.
3. Become superuser and turn accounting off. Since accounting is off when this `accton` command terminates, it should not appear in the accounting file.
4. Run the program in Figure 8.29 to print the selected fields from the accounting file.

The output from step 4 follows. We have appended the description of the process in italics to selected lines, for the discussion later.

```
accton     e =      1, chars =     336, stat =   0:       S
sh         e =   1550, chars =   20168, stat =   0:       S
dd         e =      2, chars =    1585, stat =   0:           second child
a.out      e =    202, chars =       0, stat =   0:           parent
a.out      e =    420, chars =       0, stat = 134:     F     first child
a.out      e =    600, chars =       0, stat =   9:     F     fourth child
a.out      e =    801, chars =       0, stat =   0:     F     third child
```

For this system, the elapsed time values are measured in units of clock ticks. Figure 2.15 shows that this system generates 100 clock ticks per second. For example, the `sleep(2)` in the parent corresponds to the elapsed time of 202 clock ticks. For the first child, the `sleep(4)` becomes 420 clock ticks. Note that the amount of time a process sleeps is not exact. (We'll return to the `sleep` function in Chapter 10.) Also, the calls to `fork` and `exit` take some amount of time.

Note that the `ac_stat` member is not the true termination status of the process, but rather corresponds to a portion of the termination status that we discussed in

Section 8.6. The only information in this byte is a core-flag bit (usually the high-order bit) and the signal number (usually the seven low-order bits), if the process terminated abnormally. If the process terminated normally, we are not able to obtain the `exit` status from the accounting file. For the first child, this value is 128 + 6. The 128 is the core flag bit, and 6 happens to be the value on this system for `SIGABRT`, which is generated by the call to `abort`. The value 9 for the fourth child corresponds to the value of `SIGKILL`. We can't tell from the accounting data that the parent's argument to `exit` was 2 and that the third child's argument to `exit` was 0.

The size of the file `/etc/passwd` that the `dd` process copies in the second child is 777 bytes. The number of characters of I/O is just over twice this value. It is twice the value, as 777 bytes are read in, then 777 bytes are written out. Even though the output goes to the null device, the bytes are still accounted for. The 31 additional bytes come from the `dd` command reporting the summary of bytes read and written, which it prints to `stdout`.

The `ac_flag` values are are what we would expect. The `F` flag is set for all the child processes except the second child, which does the `execl`. The `F` flag is not set for the parent, because the interactive shell that executed the parent did a `fork` and then an `exec` of the `a.out` file. The first child process calls `abort`, which generates a `SIGABRT` signal to generate the core dump. Note that neither the `X` flag nor the `D` flag is on, as they are not supported on Solaris; the information they represent can be derived from the `ac_stat` field. The fourth child also terminates because of a signal, but the `SIGKILL` signal does not generate a core dump; it just terminates the process.

As a final note, the first child has a 0 count for the number of characters of I/O, yet this process generated a `core` file. It appears that the I/O required to write the `core` file is not charged to the process. □

8.15 User Identification

Any process can find out its real and effective user ID and group ID. Sometimes, however, we want to find out the login name of the user who's running the program. We could call `getpwuid(getuid())`, but what if a single user has multiple login names, each with the same user ID? (A person might have multiple entries in the password file with the same user ID to have a different login shell for each entry.) The system normally keeps track of the name we log in under (Section 6.8), and the `getlogin` function provides a way to fetch that login name.

```
#include <unistd.h>

char *getlogin(void);
```

Returns: pointer to string giving login name if OK, `NULL` on error

This function can fail if the process is not attached to a terminal that a user logged in to. We normally call these processes *daemons*. We discuss them in Chapter 13.

Given the login name, we can then use it to look up the user in the password file—to determine the login shell, for example—using `getpwnam`.

> To find the login name, UNIX systems have historically called the `ttyname` function (Section 18.9) and then tried to find a matching entry in the `utmp` file (Section 6.8). FreeBSD and Mac OS X store the login name in the session structure associated with the process table entry and provide system calls to fetch and store this name.
>
> System V provided the `cuserid` function to return the login name. This function called `getlogin` and, if that failed, did a `getpwuid(getuid())`. The IEEE Standard 1003.1-1988 specified `cuserid`, but it called for the effective user ID to be used, instead of the real user ID. The 1990 version of POSIX.1 dropped the `cuserid` function.
>
> The environment variable `LOGNAME` is usually initialized with the user's login name by `login`(1) and inherited by the login shell. Realize, however, that a user can modify an environment variable, so we shouldn't use `LOGNAME` to validate the user in any way. Instead, we should use `getlogin`.

8.16 Process Scheduling

Historically, the UNIX System provided processes with only coarse control over their scheduling priority. The scheduling policy and priority were determined by the kernel. A process could choose to run with lower priority by adjusting its *nice value* (thus a process could be "nice" and reduce its share of the CPU by adjusting its nice value). Only a privileged process was allowed to increase its scheduling priority.

The real-time extensions in POSIX added interfaces to select among multiple scheduling classes and fine-tune their behavior. We discuss only the interfaces used to adjust the nice value here; they are part of the XSI option in POSIX.1. Refer to Gallmeister [1995] for more information on the real-time scheduling extensions.

In the Single UNIX Specification, nice values range from `0` to `(2*NZERO)-1`, although some implementations support a range from `0` to `2*NZERO`. Lower nice values have higher scheduling priority. Although this might seem backward, it actually makes sense: the more nice you are, the lower your scheduling priority is. `NZERO` is the default nice value of the system.

> Be aware that the header file defining `NZERO` differs among systems. In addition to the header file, Linux 3.2.0 makes the value of `NZERO` accessible through a nonstandard `sysconf` argument (`_SC_NZERO`).

A process can retrieve and change its nice value with the `nice` function. With this function, a process can affect only its own nice value; it can't affect the nice value of any other process.

```
#include <unistd.h>

int nice(int incr);
```

Returns: new nice value – `NZERO` if OK, –1 on error

The *incr* argument is added to the nice value of the calling process. If *incr* is too large, the system silently reduces it to the maximum legal value. Similarly, if *incr* is too small, the system silently increases it to the minimum legal value. Because –1 is a legal successful return value, we need to clear `errno` before calling `nice` and check its value if `nice` returns –1. If the call to `nice` succeeds and the return value is –1, then `errno` will still be zero. If `errno` is nonzero, it means that the call to `nice` failed.

The `getpriority` function can be used to get the nice value for a process, just like the `nice` function. However, `getpriority` can also get the nice value for a group of related processes.

```
#include <sys/resource.h>

int getpriority(int which, id_t who);
```

Returns: nice value between –`NZERO` and `NZERO`–1 if OK, –1 on error

The *which* argument can take on one of three values: `PRIO_PROCESS` to indicate a process, `PRIO_PGRP` to indicate a process group, and `PRIO_USER` to indicate a user ID. The *which* argument controls how the *who* argument is interpreted and the *who* argument selects the process or processes of interest. If the *who* argument is 0, then it indicates the calling process, process group, or user (depending on the value of the *which* argument). When *which* is set to `PRIO_USER` and *who* is 0, the real user ID of the calling process is used. When the *which* argument applies to more than one process, the highest priority (lowest value) of all the applicable processes is returned.

The `setpriority` function can be used to set the priority of a process, a process group, or all the processes belonging to a particular user ID.

```
#include <sys/resource.h>

int setpriority(int which, id_t who, int value);
```

Returns: 0 if OK, –1 on error

The *which* and *who* arguments are the same as in the `getpriority` function. The *value* is added to `NZERO` and this becomes the new nice value.

> The `nice` system call originated with an early PDP-11 version of the Research UNIX System. The `getpriority` and `setpriority` functions originated with 4.2BSD.

The Single UNIX Specification leaves it up to the implementation whether the nice value is inherited by a child process after a `fork`. However, XSI-compliant systems are required to preserve the nice value across a call to `exec`.

> A child process inherits the nice value from its parent process in FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10.

Example

The program in Figure 8.30 measures the effect of adjusting the nice value of a process. Two processes run in parallel, each incrementing its own counter. The parent runs with the default nice value, and the child runs with an adjusted nice value as specified by the

optional command argument. After running for 10 seconds, both processes print the value of their counter and exit. By comparing the counter values for different nice values, we can get an idea how the nice value affects process scheduling.

```
#include "apue.h"
#include <errno.h>
#include <sys/time.h>

#if defined(MACOS)
#include <sys/syslimits.h>
#elif defined(SOLARIS)
#include <limits.h>
#elif defined(BSD)
#include <sys/param.h>
#endif

unsigned long long count;
struct timeval end;

void
checktime(char *str)
{
    struct timeval  tv;

    gettimeofday(&tv, NULL);
    if (tv.tv_sec >= end.tv_sec && tv.tv_usec >= end.tv_usec) {
        printf("%s count = %lld\n", str, count);
        exit(0);
    }
}

int
main(int argc, char *argv[])
{
    pid_t   pid;
    char    *s;
    int     nzero, ret;
    int     adj = 0;

    setbuf(stdout, NULL);
#if defined(NZERO)
    nzero = NZERO;
#elif defined(_SC_NZERO)
    nzero = sysconf(_SC_NZERO);
#else
#error NZERO undefined
#endif
    printf("NZERO = %d\n", nzero);
    if (argc == 2)
        adj = strtol(argv[1], NULL, 10);
    gettimeofday(&end, NULL);
    end.tv_sec += 10;   /* run for 10 seconds */

    if ((pid = fork()) < 0) {
```

```
        err_sys("fork failed");
    } else if (pid == 0) {  /* child */
        s = "child";
        printf("current nice value in child is %d, adjusting by %d\n",
          nice(0)+nzero, adj);
        errno = 0;
        if ((ret = nice(adj)) == -1 && errno != 0)
            err_sys("child set scheduling priority");
        printf("now child nice value is %d\n", ret+nzero);
    } else {        /* parent */
        s = "parent";
        printf("current nice value in parent is %d\n", nice(0)+nzero);
    }
    for(;;) {
        if (++count == 0)
            err_quit("%s counter wrap", s);
        checktime(s);
    }
}
```

Figure 8.30 Evaluate the effect of changing the nice value

We run the program twice: once with the default nice value, and once with the highest valid nice value (the lowest scheduling priority). We run this on a uniprocessor Linux system to show how the scheduler shares the CPU among processes with different nice values. With an otherwise idle system, a multiprocessor system (or a multicore CPU) would allow both processes to run without the need to share a CPU, and we wouldn't see much difference between two processes with different nice values.

```
$ ./a.out
NZERO = 20
current nice value in parent is 20
current nice value in child is 20, adjusting by 0
now child nice value is 20
child count = 1859362
parent count = 1845338
$ ./a.out 20
NZERO = 20
current nice value in parent is 20
current nice value in child is 20, adjusting by 20
now child nice value is 39
parent count = 3595709
child count = 52111
```

When both processes have the same nice value, the parent process gets 50.2% of the CPU and the child gets 49.8% of the CPU. Note that the two processes are effectively treated equally. The percentages aren't exactly equal, because process scheduling isn't exact, and because the child and parent perform different amounts of processing between the time that the end time is calculated and the time that the processing loop begins.

In contrast, when the child has the highest possible nice value (the lowest priority), we see that the parent gets 98.5% of the CPU, while the child gets only 1.5% of the CPU. These values will vary based on how the process scheduler uses the nice value, so a different UNIX system will produce different ratios. □

8.17 Process Times

In Section 1.10, we described three times that we can measure: wall clock time, user CPU time, and system CPU time. Any process can call the `times` function to obtain these values for itself and any terminated children.

```
#include <sys/times.h>

clock_t times(struct tms *buf);
```

Returns: elapsed wall clock time in clock ticks if OK, –1 on error

This function fills in the `tms` structure pointed to by *buf*:

```
struct tms {
  clock_t  tms_utime;  /* user CPU time */
  clock_t  tms_stime;  /* system CPU time */
  clock_t  tms_cutime; /* user CPU time, terminated children */
  clock_t  tms_cstime; /* system CPU time, terminated children */
};
```

Note that the structure does not contain any measurement for the wall clock time. Instead, the function returns the wall clock time as the value of the function, each time it's called. This value is measured from some arbitrary point in the past, so we can't use its absolute value; instead, we use its relative value. For example, we call `times` and save the return value. At some later time, we call `times` again and subtract the earlier return value from the new return value. The difference is the wall clock time. (It is possible, though unlikely, for a long-running process to overflow the wall clock time; see Exercise 1.5.)

The two structure fields for child processes contain values only for children that we have waited for with one of the `wait` functions discussed earlier in this chapter.

All the `clock_t` values returned by this function are converted to seconds using the number of clock ticks per second—the `_SC_CLK_TCK` value returned by `sysconf` (Section 2.5.4).

> Most implementations provide the `getrusage`(2) function. This function returns the CPU times and 14 other values indicating resource usage. Historically, this function originated with the BSD operating system, so BSD-derived implementations generally support more of the fields than do other implementations.

Example

The program in Figure 8.31 executes each command-line argument as a shell command string, timing the command and printing the values from the `tms` structure.

```
#include "apue.h"
#include <sys/times.h>

static void pr_times(clock_t, struct tms *, struct tms *);
static void do_cmd(char *);

int
main(int argc, char *argv[])
{
    int     i;

    setbuf(stdout, NULL);
    for (i = 1; i < argc; i++)
        do_cmd(argv[i]);    /* once for each command-line arg */
    exit(0);
}

static void
do_cmd(char *cmd)       /* execute and time the "cmd" */
{
    struct tms  tmsstart, tmsend;
    clock_t     start, end;
    int         status;

    printf("\ncommand: %s\n", cmd);

    if ((start = times(&tmsstart)) == -1)   /* starting values */
        err_sys("times error");

    if ((status = system(cmd)) < 0)         /* execute command */
        err_sys("system() error");

    if ((end = times(&tmsend)) == -1)       /* ending values */
        err_sys("times error");

    pr_times(end-start, &tmsstart, &tmsend);
    pr_exit(status);
}

static void
pr_times(clock_t real, struct tms *tmsstart, struct tms *tmsend)
{
    static long     clktck = 0;

    if (clktck == 0)    /* fetch clock ticks per second first time */
        if ((clktck = sysconf(_SC_CLK_TCK)) < 0)
            err_sys("sysconf error");

    printf("  real:  %7.2f\n", real / (double) clktck);
    printf("  user:  %7.2f\n",
      (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
    printf("  sys:   %7.2f\n",
      (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
    printf("  child user:  %7.2f\n",
```

```
      (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
    printf("  child sys:   %7.2f\n",
      (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}
```

Figure 8.31 Time and execute all command-line arguments

If we run this program, we get

```
$ ./a.out "sleep 5" "date" "man bash >/dev/null"

command: sleep 5
  real:     5.01
  user:     0.00
  sys:      0.00
  child user:     0.00
  child sys:      0.00
normal termination, exit status = 0

command: date
Sun Feb 26 18:39:23 EST 2012
  real:     0.00
  user:     0.00
  sys:      0.00
  child user:     0.00
  child sys:      0.00
normal termination, exit status = 0

command: man bash >/dev/null
  real:     1.46
  user:     0.00
  sys:      0.00
  child user:     1.32
  child sys:      0.07
normal termination, exit status = 0
```

In the first two commands, execution is fast enough to avoid registering any CPU time at the reported resolution. In the third command, however, we run a command that takes enough processing time to note that all the CPU time appears in the child process, which is where the shell and the command execute. □

8.18 Summary

A thorough understanding of the UNIX System's process control is essential for advanced programming. There are only a few functions to master: `fork`, the `exec` family, `_exit`, `wait`, and `waitpid`. These primitives are used in many applications. The `fork` function also gave us an opportunity to look at race conditions.

Our examination of the `system` function and process accounting gave us another look at all these process control functions. We also looked at another variation of the

exec functions: interpreter files and how they operate. An understanding of the various user IDs and group IDs that are provided—real, effective, and saved—is critical to writing safe set-user-ID programs.

Given an understanding of a single process and its children, in the next chapter we examine the relationship of a process to other processes—sessions and job control. We then complete our discussion of processes in Chapter 10 when we describe signals.

Exercises

8.1 In Figure 8.3, we said that replacing the call to _exit with a call to exit might cause the standard output to be closed and printf to return –1. Modify the program to check whether your implementation behaves this way. If it does not, how can you simulate this behavior?

8.2 Recall the typical arrangement of memory in Figure 7.6. Because the stack frames corresponding to each function call are usually stored in the stack, and because after a vfork the child runs in the address space of the parent, what happens if the call to vfork is from a function other than main and the child does a return from this function after the vfork? Write a test program to verify this, and draw a picture of what's happening.

8.3 Rewrite the program in Figure 8.6 to use waitid instead of wait. Instead of calling pr_exit, determine the equivalent information from the siginfo structure.

8.4 When we execute the program in Figure 8.13 one time, as in

```
$ ./a.out
```

the output is correct. But if we execute the program multiple times, one right after the other, as in

```
$ ./a.out ; ./a.out ; ./a.out
output from parent
ooutput from parent
ouotuptut from child
put from parent
output from child
utput from child
```

the output is not correct. What's happening? How can we correct this? Can this problem happen if we let the child write its output first?

8.5 In the program shown in Figure 8.20, we call execl, specifying the *pathname* of the interpreter file. If we called execlp instead, specifying a *filename* of testinterp, and if the directory /home/sar/bin was a path prefix, what would be printed as argv[2] when the program is run?

8.6 Write a program that creates a zombie, and then call system to execute the ps(1) command to verify that the process is a zombie.

8.7 We mentioned in Section 8.10 that POSIX.1 requires open directory streams to be closed across an exec. Verify this as follows: call opendir for the root directory, peek at your system's implementation of the DIR structure, and print the close-on-exec flag. Then open the same directory for reading, and print the close-on-exec flag.

This page intentionally left blank

9

Process Relationships

9.1 Introduction

We learned in the previous chapter that there are relationships between processes. First, every process has a parent process (the initial kernel-level process is usually its own parent). The parent is notified when the child terminates, and the parent can obtain the child's exit status. We also mentioned process groups when we described the `waitpid` function (Section 8.6) and explained how we can wait for any process in a process group to terminate.

In this chapter, we'll look at process groups in more detail and the concept of sessions that was introduced by POSIX.1. We'll also look at the relationship between the login shell that is invoked for us when we log in and all the processes that we start from our login shell.

It is impossible to describe these relationships without talking about signals, and to talk about signals, we need many of the concepts in this chapter. If you are unfamiliar with the UNIX System signal mechanism, you may want to skim through Chapter 10 at this point.

9.2 Terminal Logins

Let's start by looking at the programs that are executed when we log in to a UNIX system. In early UNIX systems, such as Version 7, users logged in using dumb terminals that were connected to the host with hard-wired connections. The terminals were either local (directly connected) or remote (connected through a modem). In either case, these logins came through a terminal device driver in the kernel. For example, the

common devices on PDP-11s were DH-11s and DZ-11s. A host had a fixed number of these terminal devices, so there was a known upper limit on the number of simultaneous logins.

As bitmapped graphical terminals became available, windowing systems were developed to provide users with new ways to interact with host computers. Applications were developed to create "terminal windows" to emulate character-based terminals, allowing users to interact with hosts in familiar ways (i.e., via the shell command line).

Today, some platforms allow you to start a windowing system after logging in, whereas other platforms automatically start the windowing system for you. In the latter case, you might still have to log in, depending on how the windowing system is configured (some windowing systems can be configured to log you in automatically).

The procedure that we now describe is used to log in to a UNIX system using a terminal. The procedure is similar regardless of the type of terminal we use—it could be a character-based terminal, a graphical terminal emulating a simple character-based terminal, or a graphical terminal running a windowing system.

BSD Terminal Logins

The BSD terminal login procedure has not changed much over the past 35 years. The system administrator creates a file, usually `/etc/ttys`, that has one line per terminal device. Each line specifies the name of the device and other parameters that are passed to the `getty` program. One parameter is the baud rate of the terminal, for example. When the system is bootstrapped, the kernel creates process ID 1, the `init` process, and it is `init` that brings the system up in multiuser mode. The `init` process reads the file `/etc/ttys` and, for every terminal device that allows a login, does a `fork` followed by an `exec` of the program `getty`. This gives us the processes shown in Figure 9.1.

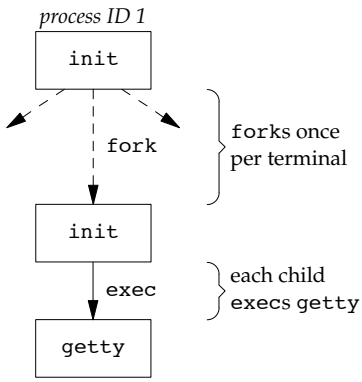


Figure 9.1 Processes invoked by `init` to allow terminal logins

All the processes shown in Figure 9.1 have a real user ID of 0 and an effective user ID of 0 (i.e., they all have superuser privileges). The `init` process also `exec`s the `getty` program with an empty environment.

It is `getty` that calls `open` for the terminal device. The terminal is opened for reading and writing. If the device is a modem, the `open` may delay inside the device driver until the modem is dialed and the call is answered. Once the device is open, file descriptors 0, 1, and 2 are set to the device. Then `getty` outputs something like `login:` and waits for us to enter our user name. If the terminal supports multiple speeds, `getty` can detect special characters that tell it to change the terminal's speed (baud rate). Consult your UNIX system manuals for additional details on the `getty` program and the data files (`gettytab`) that can drive its actions.

When we enter our user name, `getty`'s job is complete, and it then invokes the `login` program, similar to

```
execle("/bin/login", "login", "-p", username, (char *)0, envp);
```

(There can be options in the `gettytab` file to have it invoke other programs, but the default is the `login` program.) `init` invokes `getty` with an empty environment; `getty` creates an environment for `login` (the `envp` argument) with the name of the terminal (something like `TERM=foo`, where the type of terminal `foo` is taken from the `gettytab` file) and any environment strings that are specified in the `gettytab`. The `-p` flag to `login` tells it to preserve the environment that it is passed and to add to that environment, not replace it. Figure 9.2 shows the state of these processes right after `login` has been invoked.

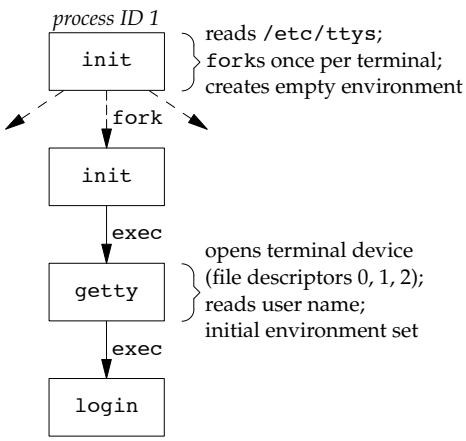


Figure 9.2 State of processes after `login` has been invoked

All the processes shown in Figure 9.2 have superuser privileges, since the original `init` process has superuser privileges. The process ID of the bottom three processes in Figure 9.2 is the same, since the process ID does not change across an `exec`. Also, all the processes other than the original `init` process have a parent process ID of 1.

The `login` program does many things. Since it has our user name, it can call `getpwnam` to fetch our password file entry. Then `login` calls `getpass`(3) to display the prompt `Password:` and read our password (with echoing disabled, of course). It calls `crypt`(3) to encrypt the password that we entered and compares the encrypted

result to the pw_passwd field from our shadow password file entry. If the login attempt fails because of an invalid password (after a few tries), login calls exit with an argument of 1. This termination will be noticed by the parent (init), and it will do another fork followed by an exec of getty, starting the procedure over again for this terminal.

This is the traditional authentication procedure used on UNIX systems. Modern UNIX systems, however, have evolved to support multiple authentication procedures. For example, FreeBSD, Linux, Mac OS X, and Solaris all support a more flexible scheme known as PAM (Pluggable Authentication Modules). PAM allows an administrator to configure the authentication methods to be used to access services that are written to use the PAM library.

If our application needs to verify that a user has the appropriate permission to perform a task, we can either hard code the authentication mechanism in the application or use the PAM library to give us the equivalent functionality. The advantage to using PAM is that administrators can configure different ways to authenticate users for different tasks, based on the local site policies.

If we log in correctly, login will

- Change to our home directory (chdir)
- Change the ownership of our terminal device (chown) so we own it
- Change the access permissions for our terminal device so we have permission to read from and write to it
- Set our group IDs by calling setgid and initgroups
- Initialize the environment with all the information that login has: our home directory (HOME), shell (SHELL), user name (USER and LOGNAME), and a default path (PATH)
- Change to our user ID (setuid) and invoke our login shell, as in

```
execl("/bin/sh", "-sh", (char *)0);
```

  The minus sign as the first character of argv[0] is a flag to all the shells that indicates they are being invoked as a login shell. The shells can look at this character and modify their start-up accordingly.

The login program really does more than we've described here. It optionally prints the message-of-the-day file, checks for new mail, and performs other tasks. In this chapter, we're interested only in the features that we've described.

Recall from our discussion of the setuid function in Section 8.11 that since it is called by a superuser process, setuid changes all three user IDs: the real user ID, effective user ID, and saved set-user-ID. The call to setgid that was done earlier by login has the same effect on all three group IDs.

At this point, our login shell is running. Its parent process ID is the original init process (process ID 1), so when our login shell terminates, init is notified (it is sent a SIGCHLD signal) and it starts the whole procedure over again for this terminal. File descriptors 0, 1, and 2 for our login shell are set to the terminal device. Figure 9.3 shows this arrangement.

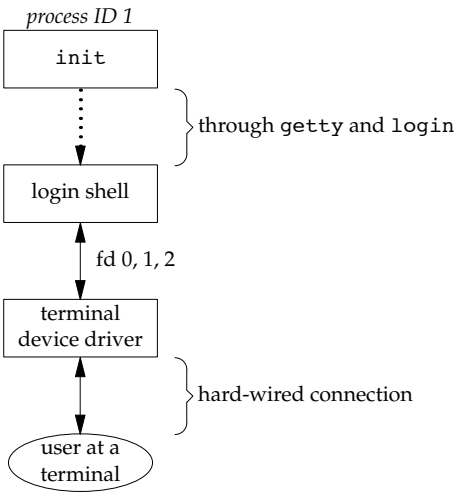


Figure 9.3 Arrangement of processes after everything is set for a terminal login

Our login shell now reads its start-up files (`.profile` for the Bourne shell and Korn shell; `.bash_profile`, `.bash_login`, or `.profile` for the GNU Bourne-again shell; and `.cshrc` and `.login` for the C shell). These start-up files usually change some of the environment variables and add many other variables to the environment. For example, most users set their own `PATH` and often prompt for the actual terminal type (`TERM`). When the start-up files are done, we finally get the shell's prompt and can enter commands.

Mac OS X Terminal Logins

On Mac OS X, the terminal login process follows essentially the same steps as in the BSD login process, since Mac OS X is based in part on FreeBSD. With Mac OS X, however, there are some differences:

- The work of `init` is performed by `launchd`.
- We are presented with a graphical-based login screen from the start.

Linux Terminal Logins

The Linux login procedure is very similar to the BSD procedure. Indeed, the Linux `login` command is derived from the 4.3BSD `login` command. The main difference between the BSD login procedure and the Linux login procedure is in the way the terminal configuration is specified.

Some Linux distributions ship with a version of the `init` program that uses administrative files patterned after System V's `init` file formats. On these systems,

/etc/inittab contains the configuration information specifying the terminal devices for which init should start a getty process.

Other Linux distributions, such as recent Ubuntu distributions, ship with a version of init that is known as "Upstart." It uses configuration files named *.conf that are stored in the /etc/init directory. For example, the specifications for running getty on /dev/tty1 might be found in the file /etc/init/tty1.conf.

Depending on the version of getty in use, the terminal characteristics are specified either on the command line (as with agetty) or in the file /etc/gettydefs (as with mgetty).

Solaris Terminal Logins

Solaris supports two forms of terminal logins: (a) getty style, as described previously for BSD, and (b) ttymon logins, a feature introduced with SVR4. Normally, getty is used for the console, and ttymon is used for other terminal logins.

The ttymon command is part of a larger facility termed SAF, the Service Access Facility. The goal of the SAF was to provide a consistent way to administer services that provide access to a system. (See Chapter 6 of Rago [1993] for more details.) For our purposes, we end up with the same picture as in Figure 9.3, with a different set of steps between init and the login shell. init is the parent of sac (the service access controller), which does a fork and exec of the ttymon program when the system enters multiuser state. The ttymon program monitors all the terminal ports listed in its configuration file and does a fork when we enter our login name. This child of ttymon does an exec of login, and login prompts us for our password. Once this is done, login execs our login shell, and we're at the position shown in Figure 9.3. One difference is that the parent of our login shell is now ttymon, whereas the parent of the login shell from a getty login is init.

9.3 Network Logins

The main (physical) difference between logging in to a system through a serial terminal and logging in to a system through a network is that the connection between the terminal and the computer isn't point-to-point. In this case, login is simply a service available, just like any other network service, such as FTP or SMTP.

With the terminal logins that we described in the previous section, init knows which terminal devices are enabled for logins and spawns a getty process for each device. In the case of network logins, however, all the logins come through the kernel's network interface drivers (e.g., the Ethernet driver), and we don't know ahead of time how many of these will occur. Instead of having a process waiting for each possible login, we now have to wait for a network connection request to arrive.

To allow the same software to process logins over both terminal logins and network logins, a software driver called a *pseudo terminal* is used to emulate the behavior of a serial terminal and map terminal operations to network operations, and vice versa. (In Chapter 19, we'll talk about pseudo terminals in detail.)

BSD Network Logins

In BSD, a single process waits for most network connections: the `inetd` process, sometimes called the *Internet superserver*. In this section, we'll look at the sequence of processes involved in network logins for a BSD system. We are not interested in the detailed network programming aspects of these processes; refer to Stevens, Fenner, and Rudoff [2004] for all the details.

As part of the system start-up, `init` invokes a shell that executes the shell script `/etc/rc`. One of the daemons that is started by this shell script is `inetd`. Once the shell script terminates, the parent process of `inetd` becomes `init`; `inetd` waits for TCP/IP connection requests to arrive at the host. When a connection request arrives for it to handle, `inetd` does a `fork` and `exec` of the appropriate program.

Let's assume that a TCP connection request arrives for the TELNET server. TELNET is a remote login application that uses the TCP protocol. A user on another host (that is connected to the server's host through a network of some form) or on the same host initiates the login by starting the TELNET client:

```
telnet hostname
```

The client opens a TCP connection to *hostname*, and the program that's started on *hostname* is called the TELNET server. The client and the server then exchange data across the TCP connection using the TELNET application protocol. What has happened is that the user who started the client program is now logged in to the server's host. (This assumes, of course, that the user has a valid account on the server's host.) Figure 9.4 shows the sequence of processes involved in executing the TELNET server, called `telnetd`.

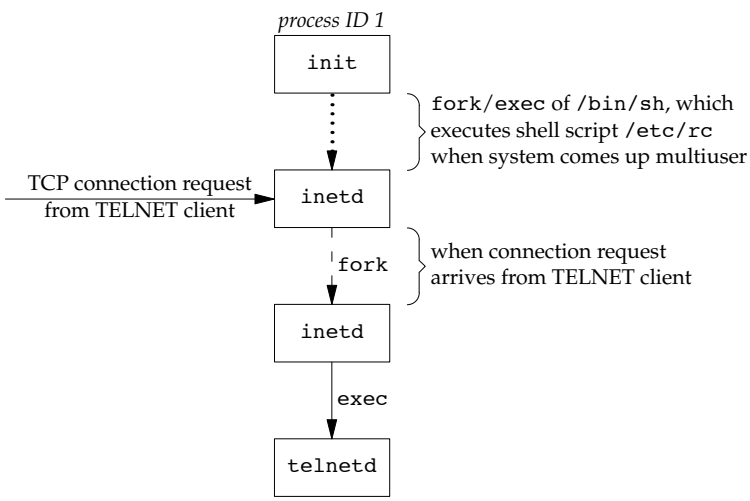


Figure 9.4 Sequence of processes involved in executing TELNET server

The `telnetd` process then opens a pseudo terminal device and splits into two processes using `fork`. The parent handles the communication across the network connection, and the child does an `exec` of the `login` program. The parent and the child are connected through the pseudo terminal. Before doing the `exec`, the child sets up file descriptors 0, 1, and 2 to the pseudo terminal. If we log in correctly, `login` performs the same steps we described in Section 9.2: it changes to our home directory and sets our group IDs, user ID, and our initial environment. Then `login` replaces itself with our login shell by calling `exec`. Figure 9.5 shows the arrangement of the processes at this point.

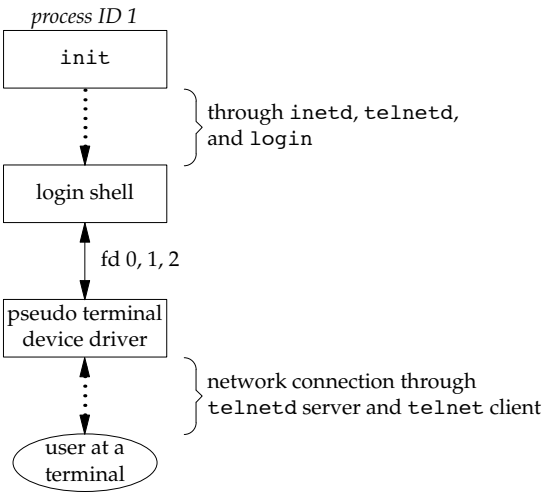


Figure 9.5 Arrangement of processes after everything is set for a network login

Obviously, a lot is going on between the pseudo terminal device driver and the actual user at the terminal. We'll show all the processes involved in this type of arrangement in Chapter 19 when we talk about pseudo terminals in more detail.

The important thing to understand is that whether we log in through a terminal (Figure 9.3) or a network (Figure 9.5), we have a login shell with its standard input, standard output, and standard error connected to either a terminal device or a pseudo terminal device. We'll see in the coming sections that this login shell is the start of a POSIX.1 session, and that the terminal or pseudo terminal is the controlling terminal for the session.

Mac OS X Network Logins

Logging in to a Mac OS X system over a network is identical to logging in to a BSD system, because Mac OS X is based partially on FreeBSD. However, on Mac OS X, the `telnet` daemon is run from `launchd`.

By default, the `telnet` daemon is disabled on Mac OS X (although it can be enabled with the `launchctl`(1) command). The preferred way to perform a network login on Mac OS X is with `ssh`, the secure shell command.

Linux Network Logins

Network logins under Linux are the same as under BSD, except that some distributions use an alternative `inetd` process called the extended Internet services daemon, `xinetd`. The `xinetd` process provides a finer level of control over services it starts compared to `inetd`.

Solaris Network Logins

The scenario for network logins under Solaris is almost identical to the steps under BSD and Linux. An `inetd` server is used that is similar in concept to the BSD version, except that the Solaris version runs as a *restarter* in the Service Management Facility (SMF). A restarter is a daemon that has the responsibility to start and monitor other daemon processes, and restart them if they fail. Although the `inetd` server is started by the master restarter in the SMF, the master restarter is started by `init` and we end up with the same overall picture as in Figure 9.5.

The Solaris Service Management Facility is a framework that manages and monitors system services and provides a way to recover from failures affecting system services. For more details on the Service Management Facility, see Adams [2005] and the Solaris manual pages `smf`(5) and `inetd`(1M).

9.4 Process Groups

In addition to having a process ID, each process belongs to a process group. We'll encounter process groups again when we discuss signals in Chapter 10.

A process group is a collection of one or more processes, usually associated with the same job (job control is discussed in Section 9.8), that can receive signals from the same terminal. Each process group has a unique process group ID. Process group IDs are similar to process IDs: they are positive integers and can be stored in a `pid_t` data type. The function `getpgrp` returns the process group ID of the calling process.

```
#include <unistd.h>

pid_t getpgrp(void);
```

Returns: process group ID of calling process

In older BSD-derived systems, the `getpgrp` function took a *pid* argument and returned the process group for that process. The Single UNIX Specification defines the `getpgid` function that mimics this behavior.

```
#include <unistd.h>

pid_t getpgid(pid_t pid);
```

Returns: process group ID if OK, –1 on error

If *pid* is 0, the process group ID of the calling process is returned. Thus

```
getpgid(0);
```

is equivalent to

```
getpgrp();
```

Each process group can have a process group leader. The leader is identified by its process group ID being equal to its process ID.

It is possible for a process group leader to create a process group, create processes in the group, and then terminate. The process group still exists, as long as at least one process is in the group, regardless of whether the group leader terminates. This is called the process group lifetime—the period of time that begins when the group is created and ends when the last remaining process leaves the group. The last remaining process in the process group can either terminate or enter some other process group.

A process joins an existing process group or creates a new process group by calling `setpgid`. (In the next section, we'll see that `setsid` also creates a new process group.)

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

Returns: 0 if OK, –1 on error

This function sets the process group ID to *pgid* in the process whose process ID equals *pid*. If the two arguments are equal, the process specified by *pid* becomes a process group leader. If *pid* is 0, the process ID of the caller is used. Also, if *pgid* is 0, the process ID specified by *pid* is used as the process group ID.

A process can set the process group ID of only itself or any of its children. Furthermore, it can't change the process group ID of one of its children after that child has called one of the `exec` functions.

In most job-control shells, this function is called after a `fork` to have the parent set the process group ID of the child, and to have the child set its own process group ID. One of these calls is redundant, but by doing both, we are guaranteed that the child is placed into its own process group before either process assumes that this has happened. If we didn't do this, we would have a race condition, since the child's process group membership would depend on which process executes first.

When we discuss signals, we'll see how we can send a signal to either a single process (identified by its process ID) or a process group (identified by its process group ID). Similarly, the `waitpid` function from Section 8.6 lets us wait for either a single process or one process from a specified process group.

9.5 Sessions

A session is a collection of one or more process groups. For example, we could have the arrangement shown in Figure 9.6. Here we have three process groups in a single session.

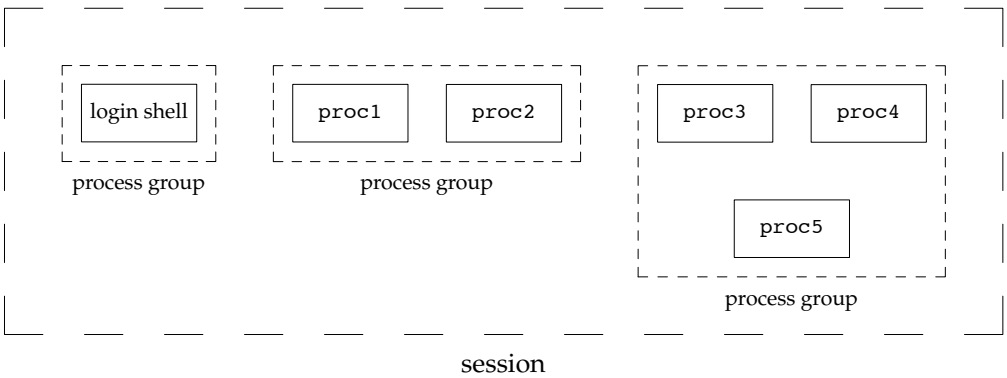


Figure 9.6 Arrangement of processes into process groups and sessions

The processes in a process group are usually placed there by a shell pipeline. For example, the arrangement shown in Figure 9.6 could have been generated by shell commands of the form

```
proc1 | proc2 &
proc3 | proc4 | proc5
```

A process establishes a new session by calling the `setsid` function.

```
#include <unistd.h>

pid_t setsid(void);
```

Returns: process group ID if OK, –1 on error

If the calling process is not a process group leader, this function creates a new session. Three things happen.

1. The process becomes the *session leader* of this new session. (A session leader is the process that creates a session.) The process is the only process in this new session.
2. The process becomes the process group leader of a new process group. The new process group ID is the process ID of the calling process.
3. The process has no controlling terminal. (We'll discuss controlling terminals in the next section.) If the process had a controlling terminal before calling `setsid`, that association is broken.

This function returns an error if the caller is already a process group leader. To ensure this is not the case, the usual practice is to call `fork` and have the parent terminate and the child continue. We are guaranteed that the child is not a process group leader, because the process group ID of the parent is inherited by the child, but the child gets a new process ID. Hence, it is impossible for the child's process ID to equal its inherited process group ID.

The Single UNIX Specification talks only about a "session leader"; there is no "session ID" similar to a process ID or a process group ID. Obviously, a session leader is a single process that has a unique process ID, so we could talk about a session ID that is the process ID of the session leader. This concept of a session ID was introduced in SVR4. Historically, BSD-based systems didn't support this notion, but have since been updated to include it. The `getsid` function returns the process group ID of a process's session leader.

> Some implementations, such as Solaris, join with the Single UNIX Specification in the practice of avoiding the use of the phrase "session ID," opting instead to refer to this as the "process group ID of the session leader." The two are equivalent, since the session leader is always the leader of a process group.

```
#include <unistd.h>

pid_t getsid(pid_t pid);
```

Returns: session leader's process group ID if OK, –1 on error

If *pid* is 0, `getsid` returns the process group ID of the calling process's session leader. For security reasons, some implementations may restrict the calling process from obtaining the process group ID of the session leader if *pid* doesn't belong to the same session as the caller.

9.6 Controlling Terminal

Sessions and process groups have a few other characteristics.

- A session can have a single *controlling terminal*. This is usually the terminal device (in the case of a terminal login) or pseudo terminal device (in the case of a network login) on which we log in.
- The session leader that establishes the connection to the controlling terminal is called the *controlling process*.
- The process groups within a session can be divided into a single *foreground process group* and one or more *background process groups*.
- If a session has a controlling terminal, it has a single foreground process group and all other process groups in the session are background process groups.
- Whenever we press the terminal's interrupt key (often DELETE or Control-C), the interrupt signal is sent to all processes in the foreground process group.

- Whenever we press the terminal's quit key (often Control-backslash), the quit signal is sent to all processes in the foreground process group.
- If a modem (or network) disconnect is detected by the terminal interface, the hang-up signal is sent to the controlling process (the session leader).

These characteristics are shown in Figure 9.7.

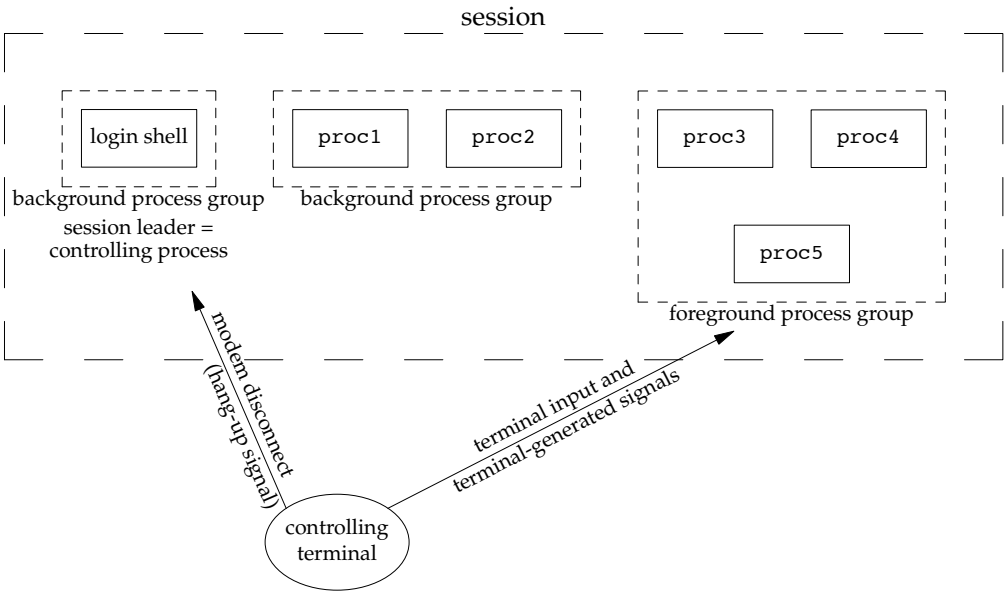


Figure 9.7 Process groups and sessions showing controlling terminal

Usually, we don't have to worry about the controlling terminal; it is established automatically when we log in.

> POSIX.1 leaves the choice of the mechanism used to allocate a controlling terminal up to each individual implementation. We'll show the actual steps in Section 19.4.
>
> Systems derived from UNIX System V allocate the controlling terminal for a session when the session leader opens the first terminal device that is not already associated with a session, as long as the call to `open` does not specify the `O_NOCTTY` flag (Section 3.3).
>
> BSD-based systems allocate the controlling terminal for a session when the session leader calls `ioctl` with a *request* argument of `TIOCSCTTY` (the third argument is a null pointer). The session cannot already have a controlling terminal for this call to succeed. (Normally, this call to `ioctl` follows a call to `setsid`, which guarantees that the process is a session leader without a controlling terminal.) The POSIX.1 `O_NOCTTY` flag to `open` is not used by BSD-based systems, except in compatibility-mode support for other systems.
>
> Figure 9.8 summarizes the way each platform discussed in this book allocates a controlling terminal. Note that although Mac OS X 10.6.8 is derived from BSD, it behaves like System V when allocating a controlling terminal.

Method	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
`open` without `O_NOCTTY`		•	•	•
`TIOCSCTTY ioctl` command	•	•	•	•

Figure 9.8 How various implementations allocate controlling terminals

There are times when a program wants to talk to the controlling terminal, regardless of whether the standard input or standard output is redirected. The way a program guarantees that it is talking to the controlling terminal is to `open` the file `/dev/tty`. This special file is a synonym within the kernel for the controlling terminal. Naturally, if the program doesn't have a controlling terminal, the `open` of this device will fail.

The classic example is the `getpass`(3) function, which reads a password (with terminal echoing turned off, of course). This function is called by the `crypt`(1) program and can be used in a pipeline. For example,

```
crypt < salaries | lpr
```

decrypts the file `salaries` and pipes the output to the print spooler. Because `crypt` reads its input file on its standard input, the standard input can't be used to enter the password. Also, `crypt` is designed so that we have to enter the encryption password each time we run the program, to prevent us from saving the password in a file (which could be a security hole).

There are known ways to break the encoding used by the `crypt` program. See Garfinkel et al. [2003] for more details on encrypting files.

9.7 `tcgetpgrp`, `tcsetpgrp`, and `tcgetsid` Functions

We need a way to tell the kernel which process group is the foreground process group, so that the terminal device driver knows where to send the terminal input and the terminal-generated signals (Figure 9.7).

```
#include <unistd.h>

pid_t tcgetpgrp(int fd);
```

Returns: process group ID of foreground process group if OK, –1 on error

```
int tcsetpgrp(int fd, pid_t pgrpid);
```

Returns: 0 if OK, –1 on error

The function `tcgetpgrp` returns the process group ID of the foreground process group associated with the terminal open on *fd*.

If the process has a controlling terminal, the process can call `tcsetpgrp` to set the foreground process group ID to *pgrpid*. The value of *pgrpid* must be the process group ID of a process group in the same session, and *fd* must refer to the controlling terminal of the session.

Most applications don't call these two functions directly. Instead, the functions are normally called by job-control shells.

The `tcgetsid` function allows an application to obtain the process group ID for the session leader given a file descriptor for the controlling TTY.

```
#include <termios.h>

pid_t tcgetsid(int fd);
```

Returns: session leader's process group ID if OK, –1 on error

Applications that need to manage controlling terminals can use `tcgetsid` to identify the session ID of the controlling terminal's session leader (which is equivalent to the session leader's process group ID).

9.8 Job Control

Job control is a feature that was added to BSD around 1980. This feature allows us to start multiple jobs (groups of processes) from a single terminal and to control which jobs can access the terminal and which jobs are run in the background. Job control requires three forms of support:

1. A shell that supports job control
2. The terminal driver in the kernel must support job control
3. The kernel must support certain job-control signals

> SVR3 provided a different form of job control called *shell layers*. The BSD form of job control, however, was selected by POSIX.1 and is what we describe here. In earlier versions of the standard, job control support was optional, but POSIX.1 now requires platforms to support it.

From our perspective, when using job control from a shell, we can start a job in either the foreground or the background. A job is simply a collection of processes, often a pipeline of processes. For example,

```
vi main.c
```

starts a job consisting of one process in the foreground. The commands

```
pr *.c | lpr &
make all &
```

start two jobs in the background. All the processes invoked by these background jobs are in the background.

As we said, to use the features provided by job control, we need to use a shell that supports job control. With older systems, it was simple to say which shells supported job control and which didn't. The C shell supported job control, the Bourne shell didn't, and it was an option with the Korn shell, depending on whether the host supported job control. But the C shell has been ported to systems (e.g., earlier versions of System V) that don't support job control, and the SVR4 Bourne shell, when invoked by the name `jsh` instead of `sh`, supports job control. The Korn shell continues to support job control

if the host does. The Bourne-again shell also supports job control. We'll just talk generically about a shell that supports job control, versus one that doesn't, when the difference between the various shells doesn't matter.

When we start a background job, the shell assigns it a job identifier and prints one or more of the process IDs. The following script shows how the Korn shell handles this:

```
$ make all > Make.out &
[1]     1475
$ pr *.c | lpr &
[2]     1490
$                                   just press RETURN
[2] +  Done           pr *.c | lpr &
[1] +  Done           make all > Make.out &
```

The `make` is job number 1 and the starting process ID is 1475. The next pipeline is job number 2 and the process ID of the first process is 1490. When the jobs are done and we press RETURN, the shell tells us that the jobs are complete. The reason we have to press RETURN is to have the shell print its prompt. The shell doesn't print the changed status of background jobs at any random time—only right before it prints its prompt, to let us enter a new command line. If the shell didn't do this, it could produce output while we were entering an input line.

The interaction with the terminal driver arises because a special terminal character affects the foreground job: the suspend key (typically Control-Z). Entering this character causes the terminal driver to send the `SIGTSTP` signal to all processes in the foreground process group. The jobs in any background process groups aren't affected. The terminal driver looks for three special characters, which generate signals to the foreground process group.

- The interrupt character (typically DELETE or Control-C) generates `SIGINT`.
- The quit character (typically Control-backslash) generates `SIGQUIT`.
- The suspend character (typically Control-Z) generates `SIGTSTP`.

In Chapter 18, we'll see how we can change these three characters to be any characters we choose and how we can disable the terminal driver's processing of these special characters.

Another job control condition can arise that must be handled by the terminal driver. Since we can have a foreground job and one or more background jobs, which of these receives the characters that we enter at the terminal? Only the foreground job receives terminal input. It is not an error for a background job to try to read from the terminal, but the terminal driver detects this and sends a special signal to the background job: `SIGTTIN`. This signal normally stops the background job; by using the shell, we are notified of this event and can bring the job into the foreground so that it can read from the terminal. The following example demonstrates this:

```
$ cat > temp.foo &          start in background, but it'll read from standard input
[1]     1681
$                           we press RETURN
[1] + Stopped (SIGTTIN)       cat > temp.foo &
```

```
$ fg %1                         bring job number 1 into the foreground
cat > temp.foo                  the shell tells us which job is now in the foreground
hello, world                    enter one line
^D                              type the end-of-file character
$ cat temp.foo                  check that the one line was put into the file
hello, world
```

Note that this example doesn't work on Mac OS X 10.6.8. When we try to bring the `cat` command into the foreground, the `read` fails with `errno` set to `EINTR`. Since Mac OS X is based on FreeBSD, and FreeBSD works as expected, this must be a bug in Mac OS X.

The shell starts the `cat` process in the background, but when `cat` tries to read its standard input (the controlling terminal), the terminal driver, knowing that it is a background job, sends the `SIGTTIN` signal to the background job. The shell detects this change in status of its child (recall our discussion of the `wait` and `waitpid` function in Section 8.6) and tells us that the job has been stopped. We then move the stopped job into the foreground with the shell's `fg` command. (Refer to the manual page for the shell that you are using for all the details on its job control commands, such as `fg` and `bg`, and the various ways to identify the different jobs.) Doing this causes the shell to place the job into the foreground process group (`tcsetpgrp`) and send the continue signal (`SIGCONT`) to the process group. Since it is now in the foreground process group, the job can read from the controlling terminal.

What happens if a background job sends its output to the controlling terminal? This is an option that we can allow or disallow. Normally, we use the `stty`(1) command to change this option. (We'll see in Chapter 18 how we can change this option from a program.) The following example shows how this works:

```
$ cat temp.foo &                execute in background
[1]     1719
$ hello, world                  the output from the background job appears after the prompt
                                we press RETURN
[1] +  Done          cat temp.foo &
$ stty tostop                   disable ability of background jobs to output to controlling terminal
$ cat temp.foo &                try it again in the background
[1]     1721
$                               we press RETURN and find the job is stopped
[1] + Stopped(SIGTTOU)        cat temp.foo &
$ fg %1                         resume stopped job in the foreground
cat temp.foo                    the shell tells us which job is now in the foreground
hello, world                    and here is its output
```

When we disallow background jobs from writing to the controlling terminal, `cat` will block when it tries to write to its standard output, because the terminal driver identifies the write as coming from a background process and sends the job the `SIGTTOU` signal. As with the previous example, when we use the shell's `fg` command to bring the job into the foreground, the job completes.

Figure 9.9 summarizes some of the features of job control that we've been describing. The solid lines through the terminal driver box mean that the terminal I/O and the terminal-generated signals are always connected from the foreground process

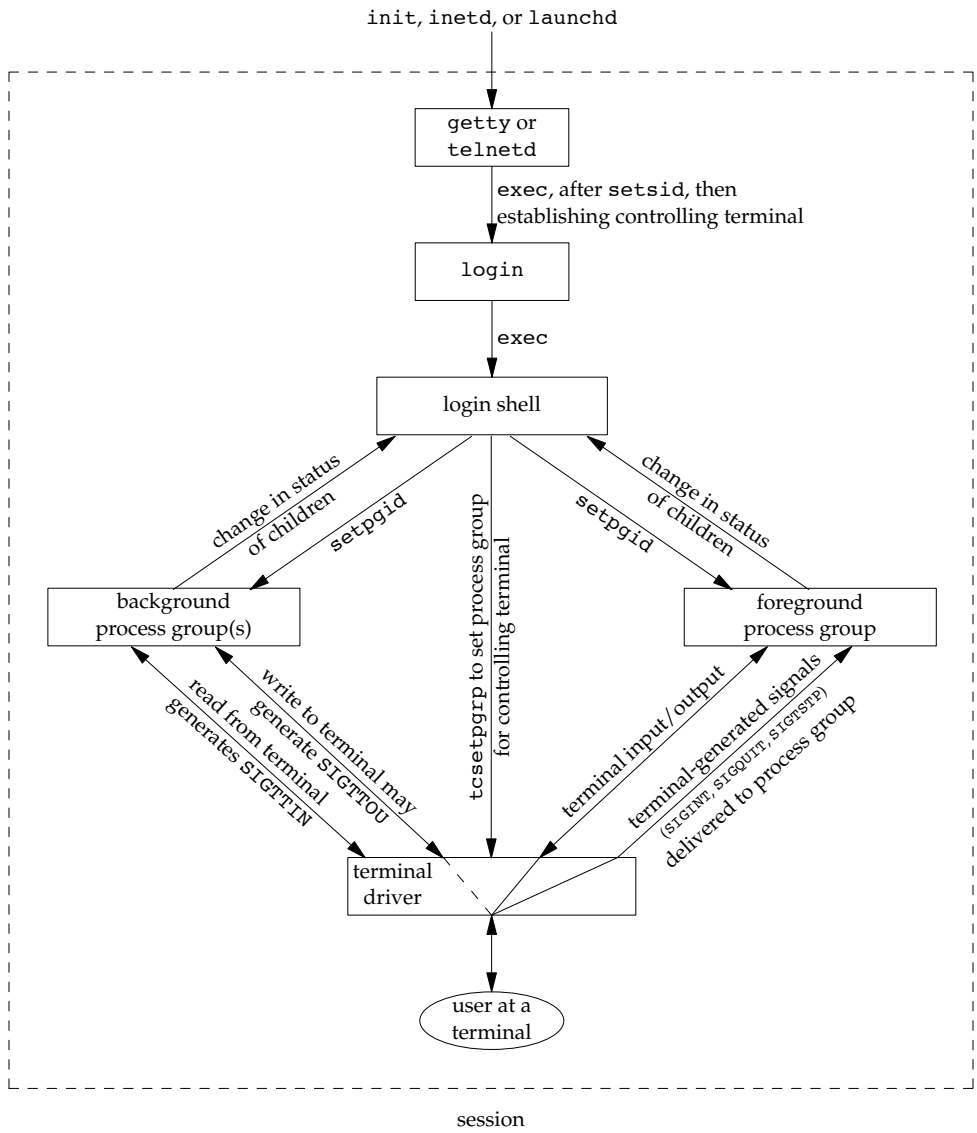


Figure 9.9 Summary of job control features with foreground and background jobs, and terminal driver

group to the actual terminal. The dashed line corresponding to the SIGTTOU signal means that whether the output from a process in the background process group appears on the terminal is an option.

Is job control necessary or desirable? Job control was originally designed and implemented before windowing terminals were widespread. Some people claim that a well-designed windowing system removes any need for job control. Some complain that the implementation of job control—requiring support from the kernel, the terminal

driver, the shell, and some applications—is a hack. Some use job control with a windowing system, claiming a need for both. Regardless of your opinion, job control is a required feature of POSIX.1.

9.9 Shell Execution of Programs

Let's examine how the shells execute programs and how this relates to the concepts of process groups, controlling terminals, and sessions. To do this, we'll use the `ps` command again.

First, we'll use a shell that doesn't support job control—the classic Bourne shell running on Solaris. If we execute

```
ps -o pid,ppid,pgid,sid,comm
```

the output is

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1774   949   949   949 ps
```

The parent of the `ps` command is the shell, which we would expect. Both the shell and the `ps` command are in the same session and foreground process group (949). We say that 949 is the foreground process group because that is what you get when you execute a command with a shell that doesn't support job control.

> Some platforms support an option to have the `ps`(1) command print the process group ID associated with the session's controlling terminal. This value would be shown under the TPGID column. Unfortunately, the output of the `ps` command often differs among versions of the UNIX System. For example, Solaris 10 doesn't support this option. Under FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8, the command
>
> ```
> ps -o pid,ppid,pgid,sid,tpgid,comm
> ```
>
> prints exactly the information we want.
>
> Note that it is misleading to associate a process with a terminal process group ID (the TPGID column). A process does not have a terminal process control group. A process belongs to a process group, and the process group belongs to a session. The session may or may not have a controlling terminal. If the session does have a controlling terminal, then the terminal device knows the process group ID of the foreground process. This value can be set in the terminal driver with the `tcsetpgrp` function, as we show in Figure 9.9. The foreground process group ID is an attribute of the terminal, not the process. This value from the terminal device driver is what `ps` prints as the TPGID. If it finds that the session doesn't have a controlling terminal, `ps` prints either 0 or –1, depending on the platform.

If we execute the command in the background,

```
ps -o pid,ppid,pgid,sid,comm &
```

the only value that changes is the process ID of the command:

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1812   949   949   949 ps
```

This shell doesn't know about job control, so the background job is not put into its own process group and the controlling terminal isn't taken away from the background job.

Now let's look at how the Bourne shell handles a pipeline. When we execute

```
ps -o pid,ppid,pgid,sid,comm | cat1
```

the output is

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1823   949   949   949 cat1
 1824  1823   949   949 ps
```

(The program `cat1` is just a copy of the standard `cat` program, with a different name. We have another copy of `cat` with the name `cat2`, which we'll use later in this section. When we have two copies of `cat` in a pipeline, the different names let us differentiate between the two programs.) Note that the last process in the pipeline is the child of the shell and that the first process in the pipeline is a child of the last process. It appears that the shell `fork`s a copy of itself and that this copy then `fork`s to make each of the previous processes in the pipeline.

If we execute the pipeline in the background,

```
ps -o pid,ppid,pgid,sid,comm | cat1 &
```

only the process IDs change. Since the shell doesn't handle job control, the process group ID of the background processes remains 949, as does the process group ID of the session.

What happens in this case if a background process tries to read from its controlling terminal? For example, suppose that we execute

```
cat > temp.foo &
```

With job control, this is handled by placing the background job into a background process group, which causes the signal `SIGTTIN` to be generated if the background job tries to read from the controlling terminal. The way this is handled without job control is that the shell automatically redirects the standard input of a background process to `/dev/null`, if the process doesn't redirect standard input itself. A read from `/dev/null` generates an end of file. This means that our background `cat` process immediately reads an end of file and terminates normally.

The previous paragraph adequately handles the case of a background process accessing the controlling terminal through its standard input, but what happens if a background process specifically opens `/dev/tty` and reads from the controlling terminal? The answer is "It depends," but the result is probably not what we want. For example,

```
crypt < salaries | lpr &
```

is such a pipeline. We run it in the background, but the `crypt` program opens `/dev/tty`, changes the terminal characteristics (to disable echoing), reads from the device, and resets the terminal characteristics. When we execute this background pipeline, the prompt `Password:` from `crypt` is printed on the terminal, but what we enter (the encryption password) is read by the shell, which tries to execute a command

of that name. The next line we enter to the shell is taken as the password, and the file is not encrypted correctly, sending junk to the printer. Here we have two processes trying to read from the same device at the same time, and the result depends on the system. Job control, as we described earlier, handles this multiplexing of a single terminal between multiple processes in a better fashion.

Returning to our Bourne shell example, if we execute three processes in the pipeline, we can examine the process control used by this shell:

```
ps -o pid,ppid,pgid,sid,comm | cat1 | cat2
```

This pipeline generates the following output:

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1988   949   949   949 cat2
 1989  1988   949   949 ps
 1990  1988   949   949 cat1
```

> Don't be alarmed if the output on your system doesn't show the proper command names. Sometimes you might get results such as
>
> ```
>  PID  PPID  PGID   SID COMMAND
>  949   947   949   949 sh
> 1831   949   949   949 sh
> 1832  1831   949   949 ps
> 1833  1831   949   949 sh
> ```
>
> What's happening here is that the `ps` process is racing with the shell, which is forking and executing the `cat` commands. In this case, the shell hasn't yet completed the call to `exec` when `ps` has obtained the list of processes to print.

Again, the last process in the pipeline is the child of the shell, and all previous processes in the pipeline are children of the last process. Figure 9.10 shows what is happening.

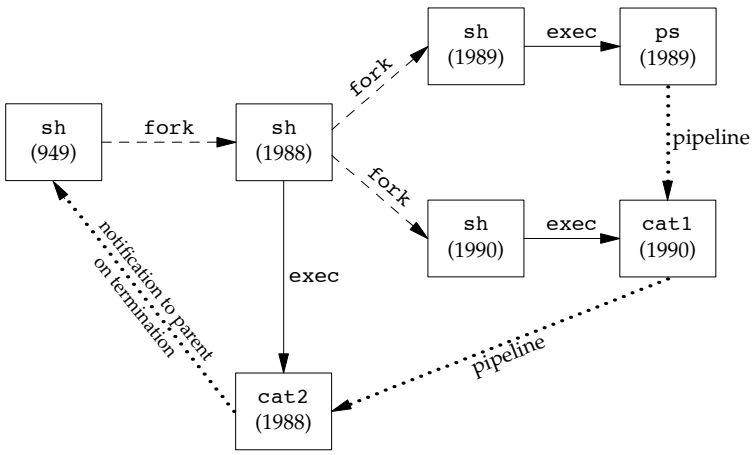


Figure 9.10 Processes in the pipeline `ps | cat1 | cat2` when invoked by Bourne shell

Since the last process in the pipeline is the child of the login shell, the shell is notified when that process (`cat2`) terminates.

Now let's examine the same examples using a job-control shell running on Linux. This shows the way these shells handle background jobs. We'll use the Bourne-again shell in this example; the results with other job-control shells are almost identical.

```
ps -o pid,ppid,pgid,sid,tpgid,comm
```

gives us

```
  PID  PPID  PGID   SID TPGID COMMAND
 2837  2818  2837  2837  5796 bash
 5796  2837  5796  2837  5796 ps
```

(Starting with this example, we show the foreground process group in a **bolder font**.) We immediately see a difference from our Bourne shell example. The Bourne-again shell places the foreground job (`ps`) into its own process group (5796). The `ps` command is the process group leader and the only process in this process group. Furthermore, this process group is the foreground process group, since it has the controlling terminal. Our login shell is a background process group while the `ps` command executes. Note, however, that both process groups, 2837 and 5796, are members of the same session. Indeed, we'll see that the session never changes through our examples in this section.

Executing this process in the background,

```
ps -o pid,ppid,pgid,sid,tpgid,comm &
```

gives us

```
  PID  PPID  PGID   SID TPGID COMMAND
 2837  2818  2837  2837  2837 bash
 5797  2837  5797  2837  2837 ps
```

Again, the `ps` command is placed into its own process group, but this time the process group (5797) is no longer the foreground process group—it is a background process group. The TPGID of 2837 indicates that the foreground process group is our login shell.

Executing two processes in a pipeline, as in

```
ps -o pid,ppid,pgid,sid,tpgid,comm | cat1
```

gives us

```
  PID  PPID  PGID   SID TPGID COMMAND
 2837  2818  2837  2837  5799 bash
 5799  2837  5799  2837  5799 ps
 5800  2837  5799  2837  5799 cat1
```

Both processes, `ps` and `cat1`, are placed into a new process group (5799), and this is the foreground process group. We can also see another difference between this example and the similar Bourne shell example. The Bourne shell created the last process in the pipeline first, and this final process was the parent of the first process. Here, the Bourne-again shell is the parent of both processes. If we execute this pipeline in the background,

```
ps -o pid,ppid,pgid,sid,tpgid,comm | cat1 &
```

the results are similar, but now `ps` and `cat1` are placed in the same background process group:

```
  PID  PPID  PGID   SID TPGID COMMAND
 2837  2818  2837  2837  2837 bash
 5801  2837  5801  2837  2837 ps
 5802  2837  5801  2837  2837 cat1
```

Note that the order in which a shell creates processes can differ depending on the particular shell in use.

9.10 Orphaned Process Groups

We've mentioned that a process whose parent terminates is called an orphan and is inherited by the `init` process. We now look at entire process groups that can be orphaned and see how POSIX.1 handles this situation.

Example

Consider a process that `fork`s a child and then terminates. Although this is nothing abnormal (it happens all the time), what happens if the child is stopped (using job control) when the parent terminates? How will the child ever be continued, and does the child know that it has been orphaned? Figure 9.11 shows this situation: the parent process has `fork`ed a child that stops, and the parent is about to exit.

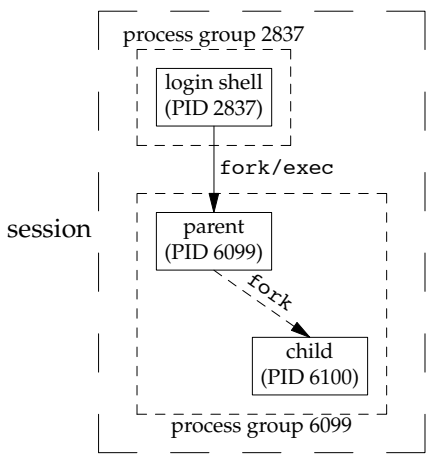


Figure 9.11 Example of a process group about to be orphaned

The program that creates this situation is shown in Figure 9.13. This program has some new features. Here, we are assuming a job-control shell. Recall from the previous section that the shell places the foreground process into its own process group (6099 in

```
#include "apue.h"
#include <errno.h>

static void
sig_hup(int signo)
{
    printf("SIGHUP received, pid = %ld\n", (long)getpid());
}

static void
pr_ids(char *name)
{
    printf("%s: pid = %ld, ppid = %ld, pgrp = %ld, tpgrp = %ld\n",
        name, (long)getpid(), (long)getppid(), (long)getpgrp(),
        (long)tcgetpgrp(STDIN_FILENO));
    fflush(stdout);
}

int
main(void)
{
    char    c;
    pid_t   pid;

    pr_ids("parent");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {   /* parent */
        sleep(5);       /* sleep to let child stop itself */
    } else {            /* child */
        pr_ids("child");
        signal(SIGHUP, sig_hup);    /* establish signal handler */
        kill(getpid(), SIGTSTP);    /* stop ourself */
        pr_ids("child");    /* prints only if we're continued */
        if (read(STDIN_FILENO, &c, 1) != 1)
            printf("read error %d on controlling TTY\n", errno);
    }
    exit(0);
}
```

Figure 9.12 Creating an orphaned process group

this example) and that the shell stays in its own process group (2837). The child inherits the process group of its parent (6099). After the `fork`,

- The parent sleeps for 5 seconds. This is our (imperfect) way of letting the child execute before the parent terminates.
- The child establishes a signal handler for the hang-up signal (`SIGHUP`) so we can see whether it is sent to the child. (We discuss signal handlers in Chapter 10.)
- The child sends itself the stop signal (`SIGTSTP`) with the `kill` function. This stops the child, similar to our stopping a foreground job with our terminal's suspend character (Control-Z).

- When the parent terminates, the child is orphaned, so the child's parent process ID becomes 1, which is the `init` process ID.
- At this point, the child is now a member of an *orphaned process group*. The POSIX.1 definition of an orphaned process group is one in which the parent of every member is either itself a member of the group or is not a member of the group's session. Another way of saying this is that the process group is not orphaned as long as a process in the group has a parent in a different process group but in the same session. If the process group is not orphaned, there is a chance that one of those parents in a different process group but in the same session will restart a stopped process in the process group that is not orphaned. Here, the parent of every process in the group (e.g., process 1 is the parent of process 6100) belongs to another session.
- Since the process group is orphaned when the parent terminates, and the process group contains a stopped process, POSIX.1 requires that every process in the newly orphaned process group be sent the hang-up signal (`SIGHUP`) followed by the continue signal (`SIGCONT`).
- This causes the child to be continued, after processing the hang-up signal. The default action for the hang-up signal is to terminate the process, so we have to provide a signal handler to catch the signal. We therefore expect the `printf` in the `sig_hup` function to appear before the `printf` in the `pr_ids` function.

Here is the output from the program shown in Figure 9.13:

```
$ ./a.out
parent: pid = 6099, ppid = 2837, pgrp = 6099, tpgrp = 6099
child: pid = 6100, ppid = 6099, pgrp = 6099, tpgrp = 6099
$ SIGHUP received, pid = 6100
child: pid = 6100, ppid = 1, pgrp = 6099, tpgrp = 2837
read error 5 on controlling TTY
```

Note that our shell prompt appears with the output from the child, since two processes—our login shell and the child—are writing to the terminal. As we expect, the parent process ID of the child has become 1.

After calling `pr_ids` in the child, the program tries to read from standard input. As we saw earlier in this chapter, when a process in a background process group tries to read from its controlling terminal, `SIGTTIN` is generated for the background process group. But here we have an orphaned process group; if the kernel were to stop it with this signal, the processes in the process group would probably never be continued. POSIX.1 specifies that the `read` is to return an error with `errno` set to `EIO` (whose value is 5 on this system) in this situation.

Finally, note that our child was placed in a background process group when the parent terminated, since the parent was executed as a foreground job by the shell. □

We'll see another example of orphaned process groups in Section 19.5 with the `pty` program.

9.11 FreeBSD Implementation

Having talked about the various attributes of a process, process group, session, and controlling terminal, it's worth looking at how all this can be implemented. We'll look briefly at the implementation used by FreeBSD. Some details of the SVR4 implementation of these features can be found in Williams [1989]. Figure 9.13 shows the various data structures used by FreeBSD.

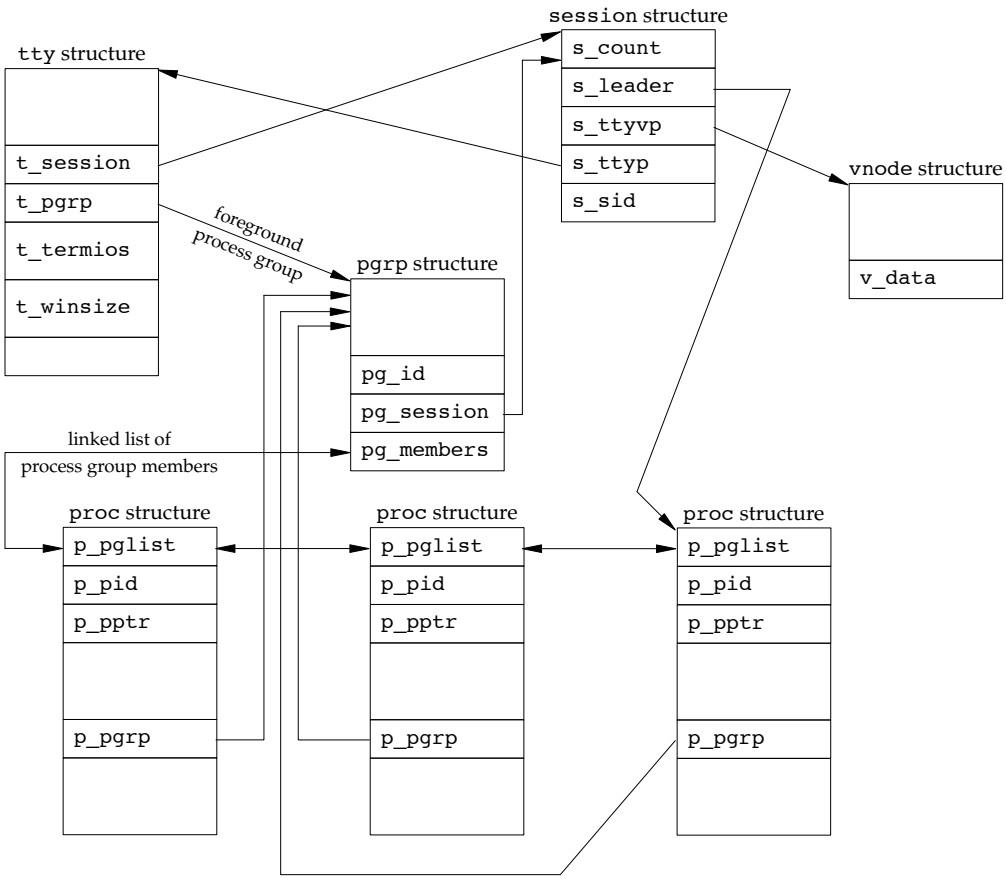


Figure 9.13 FreeBSD implementation of sessions and process groups

Let's look at all the fields that we've labeled, starting with the `session` structure. One of these structures is allocated for each session (e.g., each time `setsid` is called).

- `s_count` is the number of process groups in the session. When this counter is decremented to 0, the structure can be freed.

- `s_leader` is a pointer to the `proc` structure of the session leader.
- `s_ttyvp` is a pointer to the `vnode` structure of the controlling terminal.
- `s_ttyp` is a pointer to the `tty` structure of the controlling terminal.
- `s_sid` is the session ID. Recall that the concept of a session ID is not part of the Single UNIX Specification.

When `setsid` is called, a new `session` structure is allocated within the kernel. Now `s_count` is set to 1, `s_leader` is set to point to the `proc` structure of the calling process, `s_sid` is set to the process ID, and `s_ttyvp` and `s_ttyp` are set to null pointers, since the new session doesn't have a controlling terminal.

Let's move to the `tty` structure. The kernel contains one of these structures for each terminal device and each pseudo terminal device. (We talk more about pseudo terminals in Chapter 19.)

- `t_session` points to the `session` structure that has this terminal as its controlling terminal. (Note that the `tty` structure points to the `session` structure, and vice versa.) This pointer is used by the terminal to send a hang-up signal to the session leader if the terminal loses carrier (Figure 9.7).
- `t_pgrp` points to the `pgrp` structure of the foreground process group. This field is used by the terminal driver to send signals to the foreground process group. The three signals generated by entering special characters (interrupt, quit, and suspend) are sent to the foreground process group.
- `t_termios` is a structure containing all the special characters and related information for this terminal, such as baud rate, whether echo is enabled, and so on. We'll return to this structure in Chapter 18.
- `t_winsize` is a `winsize` structure that contains the current size of the terminal window. When the size of the terminal window changes, the `SIGWINCH` signal is sent to the foreground process group. We show how to set and fetch the terminal's current window size in Section 18.12.

To find the foreground process group of a particular session, the kernel has to start with the session structure, follow `s_ttyp` to get to the controlling terminal's `tty` structure, and then follow `t_pgrp` to get to the foreground process group's `pgrp` structure. The `pgrp` structure contains the information for a particular process group.

- `pg_id` is the process group ID.
- `pg_session` points to the `session` structure for the session to which this process group belongs.
- `pg_members` is a pointer to the list of `proc` structures that are members of this process group. The `p_pglist` structure in that `proc` structure is a doubly linked list entry that points to both the next process and the previous process in the group, and so on, until a null pointer is encountered in the `proc` structure of the last process in the group.

The proc structure contains all the information for a single process.

- p_pid contains the process ID.
- p_pptr is a pointer to the proc structure of the parent process.
- p_pgrp points to the pgrp structure of the process group to which this process belongs.
- p_pglist is a structure containing pointers to the next and previous processes in the process group, as we mentioned earlier.

Finally, we have the vnode structure. This structure is allocated when the controlling terminal device is opened. All references to /dev/tty in a process go through this vnode structure.

9.12 Summary

This chapter has described the relationships between groups of processes—sessions, which are made up of process groups. Job control is a feature supported by most UNIX systems today, and we've described how it's implemented by a shell that supports job control. The controlling terminal for a process, /dev/tty, is also involved in these process relationships.

We've made numerous references to the signals that are used in all these process relationships. The next chapter continues the discussion of signals, looking at all the UNIX System signals in detail.

Exercises

9.1 Refer back to our discussion of the utmp and wtmp files in Section 6.8. Why are the logout records written by the init process? Is this handled the same way for a network login?

9.2 Write a small program that calls fork and has the child create a new session. Verify that the child becomes a process group leader and that the child no longer has a controlling terminal.

10

Signals

10.1 Introduction

Signals are software interrupts. Most nontrivial application programs need to deal with signals. Signals provide a way of handling asynchronous events—for example, a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely.

Signals have been provided since the early versions of the UNIX System, but the signal model provided with systems such as Version 7 was not reliable. Signals could get lost, and it was difficult for a process to turn off selected signals when executing critical regions of code. Both 4.3BSD and SVR3 made changes to the signal model, adding what are called *reliable signals*. But the changes made by Berkeley and AT&T were incompatible. Fortunately, POSIX.1 standardized the reliable-signal routines, and that is what we describe here.

In this chapter, we start with an overview of signals and a description of what each signal is normally used for. Then we look at the problems with earlier implementations. It is often important to understand what is wrong with an implementation before seeing how to do things correctly. This chapter contains numerous examples that are not entirely correct and a discussion of the defects.

10.2 Signal Concepts

First, every signal has a name. These names all begin with the three characters `SIG`. For example, `SIGABRT` is the abort signal that is generated when a process calls the `abort` function. `SIGALRM` is the alarm signal that is generated when the timer set by the `alarm` function goes off. Version 7 had 15 different signals; SVR4 and 4.4BSD both had 31 different signals. FreeBSD 8.0 supports 32 different signals. Mac OS X 10.6.8 and

Linux 3.2.0 each support 31 different signals, whereas Solaris 10 supports 40 different signals. FreeBSD, Linux, and Solaris, however, support additional application-defined signals introduced to support real-time applications. Although the POSIX real-time extensions aren't covered in this book (refer to Gallmeister [1995] for more information), as of SUSv4 the real-time signal interfaces have moved to the base specification.

Signal names are all defined by positive integer constants (the signal number) in the header `<signal.h>`.

> Implementations actually define the individual signals in a different header file, but this header file is included by `<signal.h>`. It is considered bad form for the kernel to include header files meant for user-level applications, so if the applications and the kernel both need the same definitions, the information is placed in a kernel header file that is then included by the user-level header file. Thus both FreeBSD 8.0 and Mac OS X 10.6.8 define the signals in `<sys/signal.h>`. Linux 3.2.0 defines the signals in `<bits/signum.h>`, and Solaris 10 defines them in `<sys/iso/signal_iso.h>`.

No signal has a signal number of 0. We'll see in Section 10.9 that the `kill` function uses the signal number of 0 for a special case. POSIX.1 calls this value the *null signal*.

Numerous conditions can generate a signal:

- The terminal-generated signals occur when users press certain terminal keys. Pressing the DELETE key on the terminal (or Control-C on many systems) normally causes the interrupt signal (`SIGINT`) to be generated. This is how to stop a runaway program. (We'll see in Chapter 18 how this signal can be mapped to any character on the terminal.)
- Hardware exceptions generate signals: divide by 0, invalid memory reference, and the like. These conditions are usually detected by the hardware, and the kernel is notified. The kernel then generates the appropriate signal for the process that was running at the time the condition occurred. For example, `SIGSEGV` is generated for a process that executes an invalid memory reference.
- The `kill`(2) function allows a process to send any signal to another process or process group. Naturally, there are limitations: we have to be the owner of the process that we're sending the signal to, or we have to be the superuser.
- The `kill`(1) command allows us to send signals to other processes. This program is just an interface to the `kill` function. This command is often used to terminate a runaway background process.
- Software conditions can generate signals when a process should be notified of various events. These aren't hardware-generated conditions (as is the divide-by-0 condition), but software conditions. Examples are `SIGURG` (generated when out-of-band data arrives over a network connection), `SIGPIPE` (generated when a process writes to a pipe that has no reader), and `SIGALRM` (generated when an alarm clock set by the process expires).

Signals are classic examples of asynchronous events. They occur at what appear to be random times to the process. The process can't simply test a variable (such as `errno`) to see whether a signal has occurred; instead, the process has to tell the kernel "if and when this signal occurs, do the following."

We can tell the kernel to do one of three things when a signal occurs. We call this the *disposition* of the signal, or the *action* associated with a signal.

1. Ignore the signal. This works for most signals, but two signals can never be ignored: `SIGKILL` and `SIGSTOP`. The reason these two signals can't be ignored is to provide the kernel and the superuser with a surefire way of either killing or stopping any process. Also, if we ignore some of the signals that are generated by a hardware exception (such as illegal memory reference or divide by 0), the behavior of the process is undefined.
2. Catch the signal. To do this, we tell the kernel to call a function of ours whenever the signal occurs. In our function, we can do whatever we want to handle the condition. If we're writing a command interpreter, for example, when the user generates the interrupt signal at the keyboard, we probably want to return to the main loop of the program, terminating whatever command we were executing for the user. If the `SIGCHLD` signal is caught, it means that a child process has terminated, so the signal-catching function can call `waitpid` to fetch the child's process ID and termination status. As another example, if the process has created temporary files, we may want to write a signal-catching function for the `SIGTERM` signal (the termination signal that is the default signal sent by the `kill` command) to clean up the temporary files. Note that the two signals `SIGKILL` and `SIGSTOP` can't be caught.
3. Let the default action apply. Every signal has a default action, shown in Figure 10.1. Note that the default action for most signals is to terminate the process.

Figure 10.1 lists the names of all the signals, an indication of which systems support the signal, and the default action for the signal. The SUS column contains • if the signal is defined as part of the base POSIX.1 specification and **XSI** if it is defined as part of the XSI option.

When the default action is labeled "terminate+core," it means that a memory image of the process is left in the file named `core` of the current working directory of the process. (Because the file is named `core`, it shows how long this feature has been part of the UNIX System.) This file can be used with most UNIX System debuggers to examine the state of the process at the time it terminated.

> The generation of the `core` file is an implementation feature of most versions of the UNIX System. Although this feature is not part of POSIX.1, it is mentioned as a potential implementation-specific action in the Single UNIX Specification's XSI option.
>
> The name of the core file varies among implementations. On FreeBSD 8.0, for example, the core file is named *cmdname*.core, where *cmdname* is the name of the command corresponding to the process that received the signal. On Mac OS X 10.6.8, the core file is named core.*pid*, where *pid* is the ID of the process that received the signal. (These systems allow the core filename to be configured via a `sysctl` parameter. On Linux 3.2.0, the name is configured through `/proc/sys/kernel/core_pattern`.)
>
> Most implementations leave the core file in the current working directory of the corresponding process; Mac OS X places all core files in `/cores` instead.

Name	Description	ISO C	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Default action
SIGABRT	abnormal termination (abort)	•	•	•	•	•	•	terminate+core
SIGALRM	timer expired (alarm)		•	•	•	•	•	terminate
SIGBUS	hardware fault		•	•	•	•	•	terminate+core
SIGCANCEL	threads library internal use						•	ignore
SIGCHLD	change in status of child		•	•	•	•	•	ignore
SIGCONT	continue stopped process		•	•	•	•	•	continue/ignore
SIGEMT	hardware fault			•	•	•	•	terminate+core
SIGFPE	arithmetic exception	•	•	•	•	•	•	terminate+core
SIGFREEZE	checkpoint freeze						•	ignore
SIGHUP	hangup		•	•	•	•	•	terminate
SIGILL	illegal instruction	•	•	•	•	•	•	terminate+core
SIGINFO	status request from keyboard			•		•		ignore
SIGINT	terminal interrupt character	•	•	•	•	•	•	terminate
SIGIO	asynchronous I/O			•	•	•	•	terminate/ignore
SIGIOT	hardware fault			•	•	•	•	terminate+core
SIGJVM1	Java virtual machine internal use						•	ignore
SIGJVM2	Java virtual machine internal use						•	ignore
SIGKILL	termination		•	•	•	•	•	terminate
SIGLOST	resource lost						•	terminate
SIGLWP	threads library internal use			•			•	terminate/ignore
SIGPIPE	write to pipe with no readers		•	•	•	•	•	terminate
SIGPOLL	pollable event (poll)				•		•	terminate
SIGPROF	profiling time alarm (setitimer)			•	•	•	•	terminate
SIGPWR	power fail/restart				•		•	terminate/ignore
SIGQUIT	terminal quit character		•	•	•	•	•	terminate+core
SIGSEGV	invalid memory reference	•	•	•	•	•	•	terminate+core
SIGSTKFLT	coprocessor stack fault				•			terminate
SIGSTOP	stop		•	•	•	•	•	stop process
SIGSYS	invalid system call		XSI	•	•	•	•	terminate+core
SIGTERM	termination	•	•	•	•	•	•	terminate
SIGTHAW	checkpoint thaw						•	ignore
SIGTHR	threads library internal use			•				terminate
SIGTRAP	hardware fault		XSI	•	•	•	•	terminate+core
SIGTSTP	terminal stop character		•	•	•	•	•	stop process
SIGTTIN	background read from control tty		•	•	•	•	•	stop process
SIGTTOU	background write to control tty		•	•	•	•	•	stop process
SIGURG	urgent condition (sockets)		•	•	•	•	•	ignore
SIGUSR1	user-defined signal		•	•	•	•	•	terminate
SIGUSR2	user-defined signal		•	•	•	•	•	terminate
SIGVTALRM	virtual time alarm (setitimer)		XSI	•	•	•	•	terminate
SIGWAITING	threads library internal use						•	ignore
SIGWINCH	terminal window size change			•	•	•	•	ignore
SIGXCPU	CPU limit exceeded (setrlimit)		XSI	•	•	•	•	terminate or terminate+core
SIGXFSZ	file size limit exceeded (setrlimit)		XSI	•	•	•	•	terminate or terminate+core
SIGXRES	resource control exceeded						•	ignore

Figure 10.1 UNIX System signals

The core file will not be generated if (a) the process was set-user-ID and the current user is not the owner of the program file, (b) the process was set-group-ID and the current user is not the group owner of the file, (c) the user does not have permission to write in the current working directory, (d) the file already exists and the user does not have permission to write to it, or (e) the file is too big (recall the `RLIMIT_CORE` limit in Section 7.11). The permissions of the `core` file (assuming that the file doesn't already exist) are usually user-read and user-write, although Mac OS X sets only user-read.

In Figure 10.1, the signals with a description of "hardware fault" correspond to implementation-defined hardware faults. Many of these names are taken from the original PDP-11 implementation of the UNIX System. Check your system's manuals to determine exactly which type of error these signals correspond to.

We now describe each of these signals in more detail.

`SIGABRT` This signal is generated by calling the `abort` function (Section 10.17). The process terminates abnormally.

`SIGALRM` This signal is generated when a timer set with the `alarm` function expires (see Section 10.10 for more details). This signal is also generated when an interval timer set by the `setitimer`(2) function expires.

`SIGBUS` This signal indicates an implementation-defined hardware fault. Implementations usually generate this signal on certain types of memory faults, as we describe in Section 14.8.

`SIGCANCEL` This signal is used internally by the Solaris threads library. It is not meant for general use.

`SIGCHLD` Whenever a process terminates or stops, the `SIGCHLD` signal is sent to the parent. By default, this signal is ignored, so the parent must catch this signal if it wants to be notified whenever a child's status changes. The normal action in the signal-catching function is to call one of the `wait` functions to fetch the child's process ID and termination status.

Earlier releases of System V had a similar signal named `SIGCLD` (without the `H`). The semantics of this signal were different from those of other signals, and as far back as SVR2, the manual page strongly discouraged its use in new programs. (Strangely enough, this warning disappeared in the SVR3 and SVR4 versions of the manual page.) Applications should use the standard `SIGCHLD` signal, but be aware that many systems define `SIGCLD` to be the same as `SIGCHLD` for backward compatibility. If you maintain software that uses `SIGCLD`, you need to check your system's manual page to see which semantics it follows. We discuss these two signals in Section 10.7.

`SIGCONT` This job-control signal is sent to a stopped process when it is continued. The default action is to continue a stopped process, but to ignore the signal if the process wasn't stopped. A full-screen editor, for example, might catch this signal and use the signal handler to make a note to redraw the terminal screen. See Section 10.21 for additional details.

SIGEMT This indicates an implementation-defined hardware fault.

The name EMT comes from the PDP-11 "emulator trap" instruction. Not all platforms support this signal. On Linux, for example, SIGEMT is supported only for selected architectures, such as SPARC, MIPS, and PA-RISC.

SIGFPE This signals an arithmetic exception, such as divide by 0, floating-point overflow, and so on.

SIGFREEZE This signal is defined only by Solaris. It is used to notify processes that need to take special action before freezing the system state, such as might happen when a system goes into hibernation or suspended mode.

SIGHUP This signal is sent to the controlling process (session leader) associated with a controlling terminal if a disconnect is detected by the terminal interface. Referring to Figure 9.13, we see that the signal is sent to the process pointed to by the s_leader field in the session structure. This signal is generated for this condition only if the terminal's CLOCAL flag is not set. (The CLOCAL flag for a terminal is set if the attached terminal is local. The flag tells the terminal driver to ignore all modem status lines. We describe how to set this flag in Chapter 18.)

Note that the session leader that receives this signal may be in the background; see Figure 9.7 for an example. This differs from the normal terminal-generated signals (interrupt, quit, and suspend), which are always delivered to the foreground process group.

This signal is also generated if the session leader terminates. In this case, the signal is sent to each process in the foreground process group.

This signal is commonly used to notify daemon processes (Chapter 13) to reread their configuration files. The reason SIGHUP is chosen for this task is that a daemon should not have a controlling terminal and would normally never receive this signal.

SIGILL This signal indicates that the process has executed an illegal hardware instruction.

4.3BSD generated this signal from the abort function. SIGABRT is now used for this purpose.

SIGINFO This BSD signal is generated by the terminal driver when we type the status key (often Control-T). This signal is sent to all processes in the foreground process group (refer to Figure 9.9). This signal normally causes status information on processes in the foreground process group to be displayed on the terminal.

Linux doesn't provide support for SIGINFO, although the symbol is defined to be the same value as SIGPWR on the Alpha platform. This is most likely to provide some level of compatibility with software developed for OSF/1.

SIGINT This signal is generated by the terminal driver when we press the interrupt key (often DELETE or Control-C). This signal is sent to all processes in the foreground process group (refer to Figure 9.9). This signal is often used to terminate a runaway program, especially when it's generating a lot of unwanted output on the screen.

SIGIO This signal indicates an asynchronous I/O event. We discuss it in Section 14.5.2.

In Figure 10.1, we labeled the default action for SIGIO as either "terminate" or "ignore." Unfortunately, the default depends on the system. Under System V, SIGIO is identical to SIGPOLL, so its default action is to terminate the process. Under BSD, the default is to ignore the signal.

Linux 3.2.0 and Solaris 10 define SIGIO to be the same value as SIGPOLL, so the default behavior is to terminate the process. On FreeBSD 8.0 and Mac OS X 10.6.8, the default is to ignore the signal.

SIGIOT This indicates an implementation-defined hardware fault.

The name IOT comes from the PDP-11 mnemonic for the "input/output TRAP" instruction. Earlier versions of System V generated this signal from the abort function. SIGABRT is now used for this purpose.

On FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10, SIGIOT is defined to be the same value as SIGABRT.

SIGJVM1 A signal reserved for use by the Java virtual machine on Solaris.

SIGJVM2 Another signal reserved for use by the Java virtual machine on Solaris.

SIGKILL This signal is one of the two that can't be caught or ignored. It provides the system administrator with a sure way to kill any process.

SIGLOST This signal is used to notify a process running on a Solaris NFSv4 client system that a lock could not be reacquired during recovery.

SIGLWP This signal is used internally by the Solaris threads library; it is not available for general use. On FreeBSD, SIGLWP is defined to be an alias for SIGTHR.

SIGPIPE If we write to a pipeline but the reader has terminated, SIGPIPE is generated. We describe pipes in Section 15.2. This signal is also generated when a process writes to a socket of type SOCK_STREAM that is no longer connected. We describe sockets in Chapter 16.

SIGPOLL This signal is marked obsolescent in SUSv4, so it might be removed in a future version of the standard. It can be generated when a specific event occurs on a pollable device. We describe this signal with the poll function in Section 14.4.2. SIGPOLL originated with SVR3, and loosely corresponds to the BSD SIGIO and SIGURG signals.

On Linux and Solaris, SIGPOLL is defined to have the same value as SIGIO.

SIGPROF This signal is marked obsolescent in SUSv4, so it might be removed in a future version of the standard. This signal is generated when a profiling interval timer set by the `setitimer`(2) function expires.

SIGPWR This signal is system dependent. Its main use is on a system that has an uninterruptible power supply (UPS). If power fails, the UPS takes over and the software can usually be notified. Nothing needs to be done at this point, as the system continues running on battery power. But if the battery gets low (for example, if the power is off for an extended period), the software is usually notified again; at this point, it behooves the system to shut everything down. This is when `SIGPWR` should be sent. On most systems, the process that is notified of the low-battery condition sends the `SIGPWR` signal to the `init` process, and `init` handles the system shutdown.

> Solaris 10 and some Linux distributions have entries in the `inittab` file for this purpose: `powerfail` and `powerwait` (or `powerokwait`).
>
> In Figure 10.1, we labeled the default action for `SIGPWR` as either "terminate" or "ignore." Unfortunately, the default depends on the system. The default on Linux is to terminate the process. On Solaris, the signal is ignored by default.

SIGQUIT This signal is generated by the terminal driver when we press the terminal quit key (often Control-backslash). This signal is sent to all processes in the foreground process group (refer to Figure 9.9). This signal not only terminates the foreground process group (as does `SIGINT`), but also generates a `core` file.

SIGSEGV This signal indicates that the process has made an invalid memory reference (which is usually a sign that the program has a bug, such as dereferencing an uninitialized pointer).

> The name SEGV stands for "segmentation violation."

SIGSTKFLT This signal is defined only by Linux. It showed up in the earliest versions of Linux, where it was intended to be used for stack faults taken by the math coprocessor. This signal is not generated by the kernel, but remains for backward compatibility.

SIGSTOP This job-control signal stops a process. It is similar to the interactive stop signal (`SIGTSTP`), but `SIGSTOP` cannot be caught or ignored.

SIGSYS This signals an invalid system call. Somehow, the process executed a machine instruction that the kernel thought was a system call, but the parameter with the instruction that indicates the type of system call was invalid. This might happen if you build a program that uses a new system call and you then try to run the same binary on an older version of the operating system where the system call doesn't exist.

`SIGTERM` This is the termination signal sent by the `kill`(1) command by default. Because it can be caught by applications, using `SIGTERM` gives programs a chance to terminate gracefully by cleaning up before exiting (in contrast to `SIGKILL`, which can't be caught or ignored).

`SIGTHAW` This signal is defined only by Solaris and is used to notify processes that need to take special action when the system resumes operation after being suspended.

`SIGTHR` This is a signal reserved for use by the thread library on FreeBSD. It is defined to have the same value as `SIGLWP`.

`SIGTRAP` This signal indicates an implementation-defined hardware fault.

> The signal name comes from the PDP-11 TRAP instruction. Implementations often use this signal to transfer control to a debugger when a breakpoint instruction is executed.

`SIGTSTP` This interactive stop signal is generated by the terminal driver when we press the terminal suspend key (often Control-Z). This signal is sent to all processes in the foreground process group (refer to Figure 9.9).

> Unfortunately, the term *stop* has different meanings. When discussing job control and signals, we talk about stopping and continuing jobs. The terminal driver, however, has historically used the term *stop* to refer to stopping and starting the terminal output using the Control-S and Control-Q characters. Therefore, the terminal driver calls the character that generates the interactive stop signal the suspend character, not the stop character.

`SIGTTIN` This signal is generated by the terminal driver when a process in a background process group tries to read from its controlling terminal. (Refer to the discussion of this topic in Section 9.8.) As special cases, if either (a) the reading process is ignoring or blocking this signal or (b) the process group of the reading process is orphaned, then the signal is not generated; instead, the read operation fails with `errno` set to `EIO`.

`SIGTTOU` This signal is generated by the terminal driver when a process in a background process group tries to write to its controlling terminal. (This is discussed in Section 9.8.) Unlike the case with background reads, a process can choose to allow background writes to the controlling terminal. We describe how to modify this option in Chapter 18.

If background writes are not allowed, then like the `SIGTTIN` signal, there are two special cases: if either (a) the writing process is ignoring or blocking this signal or (b) the process group of the writing process is orphaned, then the signal is not generated; instead, the write operation returns an error with `errno` set to `EIO`.

Regardless of whether background writes are allowed, certain terminal operations (other than writing) can also generate the `SIGTTOU` signal.

These include `tcsetattr`, `tcsendbreak`, `tcdrain`, `tcflush`, `tcflow`, and `tcsetpgrp`. We describe these terminal operations in Chapter 18.

`SIGURG` This signal notifies the process that an urgent condition has occurred. It is optionally generated when out-of-band data is received on a network connection.

`SIGUSR1` This is a user-defined signal, for use in application programs.

`SIGUSR2` This is another user-defined signal, similar to `SIGUSR1`, for use in application programs.

`SIGVTALRM` This signal is generated when a virtual interval timer set by the `setitimer`(2) function expires.

`SIGWAITING` This signal is used internally by the Solaris threads library, and is not available for general use.

`SIGWINCH` The kernel maintains the size of the window associated with each terminal and pseudo terminal. A process can get and set the window size with the `ioctl` function, which we describe in Section 18.12. If a process changes the window size from its previous value using the `ioctl` set-window-size command, the kernel generates the `SIGWINCH` signal for the foreground process group.

`SIGXCPU` The Single UNIX Specification supports the concept of resource limits as part of the XSI option; refer to Section 7.11. If the process exceeds its soft CPU time limit, the `SIGXCPU` signal is generated.

> In Figure 10.1, we labeled the default action for `SIGXCPU` as either "terminate" or "terminate with a core file." The default depends on the operating system. Linux 3.2.0 and Solaris 10 support a default action of terminate with a core file, whereas FreeBSD 8.0 and Mac OS X 10.6.8 support a default action of terminate without generating a core file. The Single UNIX Specification requires that the default action be to terminate the process abnormally. Whether a core file is generated is left up to the implementation.

`SIGXFSZ` This signal is generated if the process exceeds its soft file size limit; refer to Section 7.11.

> Just as with `SIGXCPU`, the default action taken with `SIGXFSZ` depends on the operating system. On Linux 3.2.0 and Solaris 10, the default is to terminate the process and create a core file. On FreeBSD 8.0 and Mac OS X 10.6.8, the default is to terminate the process without generating a core file. The Single UNIX Specification requires that the default action be to terminate the process abnormally. Whether a core file is generated is left up to the implementation.

`SIGXRES` This signal is defined only by Solaris. It is optionally used to notify processes that have exceeded a preconfigured resource value. The Solaris resource control mechanism is a general facility for controlling the use of shared resources among independent application sets.

10.3 signal Function

The simplest interface to the signal features of the UNIX System is the `signal` function.

```
#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);
```

Returns: previous disposition of signal (see following) if OK, `SIG_ERR` on error

> The `signal` function is defined by ISO C, which doesn't involve multiple processes, process groups, terminal I/O, and the like. Therefore, its definition of signals is vague enough to be almost useless for UNIX systems.
>
> Implementations derived from UNIX System V support the `signal` function, but it provides the old unreliable-signal semantics. (We describe these older semantics in Section 10.4.) The `signal` function provides backward compatibility for applications that require the older semantics. New applications should not use these unreliable signals.
>
> 4.4BSD also provides the `signal` function, but it is defined in terms of the `sigaction` function (which we describe in Section 10.14), so using it under 4.4BSD provides the newer reliable-signal semantics. Most current systems follow this strategy, but Solaris 10 follows the System V semantics for the `signal` function.
>
> Because the semantics of `signal` differ among implementations, we must use the `sigaction` function instead. We provide an implementation of `signal` that uses `sigaction` in Section 10.14. All the examples in this text use the `signal` function from Figure 10.18 to give us consistent semantics regardless of which particular platform we use.

The *signo* argument is just the name of the signal from Figure 10.1. The value of *func* is (a) the constant `SIG_IGN`, (b) the constant `SIG_DFL`, or (c) the address of a function to be called when the signal occurs. If we specify `SIG_IGN`, we are telling the system to ignore the signal. (Remember that we cannot ignore the two signals `SIGKILL` and `SIGSTOP`.) When we specify `SIG_DFL`, we are setting the action associated with the signal to its default value (see the final column in Figure 10.1). When we specify the address of a function to be called when the signal occurs, we are arranging to "catch" the signal. We call the function either the *signal handler* or the *signal-catching function*.

The prototype for the `signal` function states that the function requires two arguments and returns a pointer to a function that returns nothing (`void`). The `signal` function's first argument, *signo*, is an integer. The second argument is a pointer to a function that takes a single integer argument and returns nothing. The function whose address is returned as the value of `signal` takes a single integer argument (the final `(int)`). In plain English, this declaration says that the signal handler is passed a single integer argument (the signal number) and that it returns nothing. When we call `signal` to establish the signal handler, the second argument is a pointer to the function. The return value from `signal` is the pointer to the previous signal handler.

> Many systems call the signal handler with additional, implementation-dependent arguments. We discuss this further in Section 10.14.

The perplexing `signal` function prototype shown at the beginning of this section can be made much simpler through the use of the following `typedef` [Plauger 1992]:

```
typedef void Sigfunc(int);
```

Then the prototype becomes

```
Sigfunc *signal(int, Sigfunc *);
```

We've included this `typedef` in `apue.h` (Appendix B) and use it with the functions in this chapter.

If we examine the system's header `<signal.h>`, we will probably find declarations of the form

```
#define SIG_ERR   (void (*)())-1
#define SIG_DFL   (void (*)())0
#define SIG_IGN   (void (*)())1
```

These constants can be used in place of the "pointer to a function that takes an integer argument and returns nothing," the second argument to `signal`, and the return value from `signal`. The three values used for these constants need not be –1, 0, and 1. They must be three values that can never be the address of any declarable function. Most UNIX systems use the values shown.

Example

Figure 10.2 shows a simple signal handler that catches either of the two user-defined signals and prints the signal number. In Section 10.10, we describe the `pause` function, which simply suspends the calling process until a signal is received.

```
#include "apue.h"

static void sig_usr(int);   /* one handler for both signals */

int
main(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR2");
    for ( ; ; )
        pause();
}

static void
sig_usr(int signo)      /* argument is signal number */
{
    if (signo == SIGUSR1)
        printf("received SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("received SIGUSR2\n");
    else
        err_dump("received signal %d\n", signo);
}
```

Figure 10.2 Simple program to catch `SIGUSR1` and `SIGUSR2`

We invoke the program in the background and use the kill(1) command to send it signals. Note that the term *kill* in the UNIX System is a misnomer. The kill(1) command and the kill(2) function just send a signal to a process or process group. Whether that signal terminates the process depends on which signal is sent and whether the process has arranged to catch the signal.

```
$ ./a.out &                          start process in background
[1]      7216                        job-control shell prints job number and process ID
$ kill -USR1 7216                    send it SIGUSR1
received SIGUSR1
$ kill -USR2 7216                    send it SIGUSR2
received SIGUSR2
$ kill 7216                          now send it SIGTERM
[1]+  Terminated          ./a.out
```

When we send the SIGTERM signal, the process is terminated, since it doesn't catch the signal, and the default action for the signal is termination. □

Program Start-Up

When a program is executed, the status of all signals is either default or ignore. Normally, all signals are set to their default action, unless the process that calls exec is ignoring the signal. Specifically, the exec functions change the disposition of any signals being caught to their default action and leave the status of all other signals alone. (Naturally, a signal that is being caught by a process that calls exec cannot be caught by the same function in the new program, since the address of the signal-catching function in the caller probably has no meaning in the new program file that is executed.)

One specific example of this signal status behavior is how an interactive shell treats the interrupt and quit signals for a background process. With a shell that doesn't support job control, when we execute a process in the background, as in

```
cc main.c &
```

the shell automatically sets the disposition of the interrupt and quit signals in the background process to be ignored. This is done so that if we type the interrupt character, it doesn't affect the background process. If this weren't done and we typed the interrupt character, it would terminate not only the foreground process, but also all the background processes.

Many interactive programs that catch these two signals have code that looks like

```
void sig_int(int), sig_quit(int);

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, sig_int);
if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)
    signal(SIGQUIT, sig_quit);
```

Following this approach, the process catches the signal only if the signal is not currently being ignored.

These two calls to `signal` also show a limitation of the `signal` function: we are not able to determine the current disposition of a signal without changing the disposition. We'll see later in this chapter how the `sigaction` function allows us to determine a signal's disposition without changing it.

Process Creation

When a process calls `fork`, the child inherits the parent's signal dispositions. Here, since the child starts off with a copy of the parent's memory image, the address of a signal-catching function has meaning in the child.

10.4 Unreliable Signals

In earlier versions of the UNIX System (such as Version 7), signals were unreliable. By this we mean that signals could get lost: a signal could occur and the process would never know about it. Also, a process had little control over a signal: a process could catch the signal or ignore it. Sometimes, we would like to tell the kernel to block a signal: don't ignore it, just remember if it occurs, and tell us later when we're ready.

> Changes were made with 4.2BSD to provide what are called *reliable signals*. A different set of changes was then made in SVR3 to provide reliable signals under System V. POSIX.1 chose the BSD model to standardize.

One problem with these early versions was that the action for a signal was reset to its default each time the signal occurred. (In the previous example, when we ran the program in Figure 10.2, we avoided this detail by catching each signal only once.) The classic example from programming books that described these earlier systems concerns how to handle the interrupt signal. The code that was described usually looked like

```
int    sig_int();        /* my signal handling function */
  .
  .
  .
signal(SIGINT, sig_int); /* establish handler */
  .
  .
  .

sig_int()
{
    signal(SIGINT, sig_int); /* reestablish handler for next time */
  .                          /* process the signal ... */
  .
  .
}
```

(The reason the signal handler is declared as returning an integer is that these early systems didn't support the ISO C `void` data type.)

The problem with this code fragment is that there is a window of time—after the signal has occurred, but before the call to `signal` in the signal handler—when the interrupt signal could occur another time. This second signal would cause the default

action to occur, which for this signal terminates the process. This is one of those conditions that works correctly most of the time, causing us to think that it is correct, when it isn't.

Another problem with these earlier systems was that the process was unable to turn a signal off when it didn't want the signal to occur. All the process could do was ignore the signal. There are times when we would like to tell the system "prevent the following signals from interrupting me, but remember if they do occur." The classic example that demonstrates this flaw is shown by a piece of code that catches a signal and sets a flag for the process that indicates that the signal occurred:

```
int   sig_int();              /* my signal handling function */
int   sig_int_flag;           /* set nonzero when signal occurs */

main()
{
    signal(SIGINT, sig_int); /* establish handler */
      .
      .
      .
    while (sig_int_flag == 0)
        pause();              /* go to sleep, waiting for signal */
      .
      .
      .
}

sig_int()
{
    signal(SIGINT, sig_int); /* reestablish handler for next time */
    sig_int_flag = 1;        /* set flag for main loop to examine */
}
```

Here, the process is calling the `pause` function to put it to sleep until a signal is caught. When the signal is caught, the signal handler just sets the flag `sig_int_flag` to a nonzero value. The process is automatically awakened by the kernel after the signal handler returns, notices that the flag is nonzero, and does whatever it needs to do. But there is a window of time when things can go wrong. If the signal occurs after the test of `sig_int_flag` but before the call to `pause`, the process could go to sleep forever (assuming that the signal is never generated again). This occurrence of the signal is lost. This is another example of some code that isn't right, yet it works most of the time. Debugging this type of problem can be difficult.

10.5 Interrupted System Calls

A characteristic of earlier UNIX systems was that if a process caught a signal while the process was blocked in a "slow" system call, the system call was interrupted. The system call returned an error and `errno` was set to `EINTR`. This was done under the assumption that since a signal occurred and the process caught it, there is a good chance that something has happened that should wake up the blocked system call.

> Here, we have to differentiate between a system call and a function. It is a system call within the kernel that is interrupted when a signal is caught.

To support this feature, the system calls are divided into two categories: the "slow" system calls and all the others. The slow system calls are those that can block forever. Included in this category are

- Reads that can block the caller forever if data isn't present with certain file types (pipes, terminal devices, and network devices)
- Writes that can block the caller forever if the data can't be accepted immediately by these same file types
- Opens on certain file types that block the caller until some condition occurs (such as a terminal device open waiting until an attached modem answers the phone)
- The `pause` function (which by definition puts the calling process to sleep until a signal is caught) and the `wait` function
- Certain `ioctl` operations
- Some of the interprocess communication functions (Chapter 15)

The notable exception to these slow system calls is anything related to disk I/O. Although a read or a write of a disk file can block the caller temporarily (while the disk driver queues the request and then the request is executed), unless a hardware error occurs, the I/O operation always returns and unblocks the caller quickly.

One condition that is handled by interrupted system calls, for example, is when a process initiates a read from a terminal device and the user at the terminal walks away from the terminal for an extended period. In this example, the process could be blocked for hours or days and would remain so unless the system was taken down.

> POSIX.1 semantics for interrupted `read`s and `write`s changed with the 2001 version of the standard. Earlier versions gave implementations a choice of how to deal with `read`s and `write`s that have processed partial amounts of data. If `read` has received and transferred data to an application's buffer, but has not yet received all that the application requested and is then interrupted, the operating system could either fail the system call, with `errno` set to `EINTR`, or allow the system call to succeed, returning the partial amount of data received. Similarly, if `write` is interrupted after transferring some of the data in an application's buffer, the operation system could either fail the system call, with `errno` set to `EINTR`, or allow the system call to succeed, returning the partial amount of data written. Historically, implementations derived from System V fail the system call, whereas BSD-derived implementations return partial success. With the 2001 version of the POSIX.1 standard, the BSD-style semantics are required.

The problem with interrupted system calls is that we now have to handle the error return explicitly. The typical code sequence (assuming a read operation and assuming that we want to restart the read even if it's interrupted) would be

```
again:
    if ((n = read(fd, buf, BUFFSIZE)) < 0) {
        if (errno == EINTR)
            goto again;     /* just an interrupted system call */
        /* handle other errors */
    }
```

To prevent applications from having to handle interrupted system calls, 4.2BSD introduced the automatic restarting of certain interrupted system calls. The system calls that were automatically restarted are `ioctl`, `read`, `readv`, `write`, `writev`, `wait`, and `waitpid`. As we've mentioned, the first five of these functions are interrupted by a signal only if they are operating on a slow device; `wait` and `waitpid` are always interrupted when a signal is caught. Since this caused a problem for some applications that didn't want the operation restarted if it was interrupted, 4.3BSD allowed the process to disable this feature on a per-signal basis.

> POSIX.1 requires an implementation to restart system calls only when the `SA_RESTART` flag is in effect for the interrupting signal. As we will see in Section 10.14, this flag is used with the `sigaction` function to allow applications to request that interrupted system calls be restarted.
>
> Historically, when using the `signal` function to establish a signal handler, implementations varied with respect to how interrupted system calls were handled. System V never restarted system calls by default. BSD, in contrast, restarted them if the calls were interrupted by signals. On FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8, when signal handlers are installed with the `signal` function, interrupted system calls will be restarted. The default on Solaris 10, however, is to return an error (`EINTR`) instead when system calls are interrupted by signal handlers installed with the `signal` function. By using our own implementation of the `signal` function (shown in Figure 10.18), we avoid having to deal with these differences.

One of the reasons 4.2BSD introduced the automatic restart feature is that sometimes we don't know that the input or output device is a slow device. If the program we write can be used interactively, then it might be reading or writing a slow device, since terminals fall into this category. If we catch signals in this program, and if the system doesn't provide the restart capability, then we have to test every read or write for the interrupted error return and reissue the read or write.

Figure 10.3 summarizes the signal functions and their semantics provided by the various implementations.

Functions	System	Signal handler remains installed	Ability to block signals	Automatic restart of interrupted system calls?
`signal`	ISO C, POSIX.1	unspecified	unspecified	unspecified
	V7, SVR2, SVR3			never
	SVR4, Solaris			never
	4.2BSD	•	•	always
	4.3BSD, 4.4BSD, FreeBSD, Linux, Mac OS X	•	•	default
`sigaction`	POSIX.1, 4.4BSD, SVR4, FreeBSD, Linux, Mac OS X, Solaris	•	•	optional

Figure 10.3 Features provided by various signal implementations

Be aware that UNIX systems from other vendors can have values different from those shown in this figure. For example, `sigaction` under SunOS 4.1.2 restarts an interrupted system call by default, unlike the platforms listed in Figure 10.3.

In Figure 10.18, we provide our own version of the `signal` function that automatically tries to restart interrupted system calls (other than for the `SIGALRM` signal). In Figure 10.19, we provide another function, `signal_intr`, that tries to never do the restart.

We talk more about interrupted system calls in Section 14.4 with regard to the `select` and `poll` functions.

10.6 Reentrant Functions

When a signal that is being caught is handled by a process, the normal sequence of instructions being executed by the process is temporarily interrupted by the signal handler. The process then continues executing, but the instructions in the signal handler are now executed. If the signal handler returns (instead of calling `exit` or `longjmp`, for example), then the normal sequence of instructions that the process was executing when the signal was caught continues executing. (This is similar to what happens when a hardware interrupt occurs.) But in the signal handler, we can't tell where the process was executing when the signal was caught. What if the process was in the middle of allocating additional memory on its heap using `malloc`, and we call `malloc` from the signal handler? Or, what if the process was in the middle of a call to a function, such as `getpwnam` (Section 6.2), that stores its result in a static location, and we call the same function from the signal handler? In the `malloc` example, havoc can result for the process, since `malloc` usually maintains a linked list of all its allocated areas, and it may have been in the middle of changing this list. In the case of `getpwnam`, the information returned to the normal caller can get overwritten with the information returned to the signal handler.

The Single UNIX Specification specifies the functions that are guaranteed to be safe to call from within a signal handler. These functions are reentrant and are called *async-signal safe* by the Single UNIX Specification. Besides being reentrant, they block any signals during operation if delivery of a signal might cause inconsistencies. Figure 10.4 lists these async-signal safe functions. Most of the functions that are not included in Figure 10.4 are missing because (a) they are known to use static data structures, (b) they call `malloc` or `free`, or (c) they are part of the standard I/O library. Most implementations of the standard I/O library use global data structures in a nonreentrant way. Note that even though we call `printf` from signal handlers in some of our examples, it is not guaranteed to produce the expected results, since the signal handler can interrupt a call to `printf` from our main program.

Be aware that even if we call a function listed in Figure 10.4 from a signal handler, there is only one `errno` variable per thread (recall the discussion of `errno` and threads in Section 1.7), and we might potentially modify its value. Consider a signal handler that is invoked right after `main` has set `errno`. If the signal handler calls `read`, for example, this call can change the value of `errno`, wiping out the value that was just

abort	faccessat	linkat	select	socketpair
accept	fchmod	listen	sem_post	stat
access	fchmodat	lseek	send	symlink
aio_error	fchown	lstat	sendmsg	symlinkat
aio_return	fchownat	mkdir	sendto	tcdrain
aio_suspend	fcntl	mkdirat	setgid	tcflow
alarm	fdatasync	mkfifo	setpgid	tcflush
bind	fexecve	mkfifoat	setsid	tcgetattr
cfgetispeed	fork	mknod	setsockopt	tcgetpgrp
cfgetospeed	fstat	mknodat	setuid	tcsendbreak
cfsetispeed	fstatat	open	shutdown	tcsetattr
cfsetospeed	fsync	openat	sigaction	tcsetpgrp
chdir	ftruncate	pause	sigaddset	time
chmod	futimens	pipe	sigdelset	timer_getoverrun
chown	getegid	poll	sigemptyset	timer_gettime
clock_gettime	geteuid	posix_trace_event	sigfillset	timer_settime
close	getgid	pselect	sigismember	times
connect	getgroups	raise	signal	umask
creat	getpeername	read	sigpause	uname
dup	getpgrp	readlink	sigpending	unlink
dup2	getpid	readlinkat	sigprocmask	unlinkat
execl	getppid	recv	sigqueue	utime
execle	getsockname	recvfrom	sigset	utimensat
execv	getsockopt	recvmsg	sigsuspend	utimes
execve	getuid	rename	sleep	wait
_Exit	kill	renameat	sockatmark	waitpid
_exit	link	rmdir	socket	write

Figure 10.4 Reentrant functions that may be called from a signal handler

stored in `main`. Therefore, as a general rule, when calling the functions listed in Figure 10.4 from a signal handler, we should save and restore `errno`. (Be aware that a commonly caught signal is `SIGCHLD`, and its signal handler usually calls one of the `wait` functions. All the `wait` functions can change `errno`.)

Note that `longjmp` (Section 7.10) and `siglongjmp` (Section 10.15) are missing from Figure 10.4, because the signal may have occurred while the main routine was updating a data structure in a nonreentrant way. This data structure could be left half updated if we call `siglongjmp` instead of returning from the signal handler. If it is going to do such things as update global data structures, as we describe here, while catching signals that cause `sigsetjmp` to be executed, an application needs to block the signals while updating the data structures.

Example

Figure 10.5 shows a program that calls the nonreentrant function `getpwnam` from a signal handler that is called every second. We describe the `alarm` function in Section 10.10. We use it here to generate a `SIGALRM` signal every second.

```
#include "apue.h"
#include <pwd.h>

static void
my_alarm(int signo)
{
    struct passwd   *rootptr;

    printf("in signal handler\n");
    if ((rootptr = getpwnam("root")) == NULL)
            err_sys("getpwnam(root) error");
    alarm(1);
}

int
main(void)
{
    struct passwd   *ptr;

    signal(SIGALRM, my_alarm);
    alarm(1);
    for ( ; ; ) {
        if ((ptr = getpwnam("sar")) == NULL)
            err_sys("getpwnam error");
        if (strcmp(ptr->pw_name, "sar") != 0)
            printf("return value corrupted!, pw_name = %s\n",
                    ptr->pw_name);
    }
}
```

Figure 10.5 Call a nonreentrant function from a signal handler

When this program was run, the results were random. Usually, the program would be terminated by a `SIGSEGV` signal when the signal handler returned after several iterations. An examination of the `core` file showed that the `main` function had called `getpwnam`, but that when `getpwnam` called `free`, the signal handler interrupted it and called `getpwnam`, which in turn called `free`. The data structures maintained by `malloc` and `free` had been corrupted when the signal handler (indirectly) called `free` while the `main` function was also calling `free`. Occasionally, the program would run for several seconds before crashing with a `SIGSEGV` error. When the `main` function did run correctly after the signal had been caught, the return value was sometimes corrupted and sometimes fine.

As shown by this example, if we call a nonreentrant function from a signal handler, the results are unpredictable. □

10.7 `SIGCLD` Semantics

Two signals that continually generate confusion are `SIGCLD` and `SIGCHLD`. The name `SIGCLD` (without the `H`) is from System V, and this signal has different semantics from the BSD signal, named `SIGCHLD`. The POSIX.1 signal is also named `SIGCHLD`.

The semantics of the BSD SIGCHLD signal are normal, in the sense that its semantics are similar to those of all other signals. When the signal occurs, the status of a child has changed, and we need to call one of the wait functions to determine what has happened.

System V, however, has traditionally handled the SIGCLD signal differently from other signals. SVR4-based systems continue this questionable tradition (i.e., compatibility constraint) if we set its disposition using either signal or sigset (the older, SVR3-compatible functions to set the disposition of a signal). This older handling of SIGCLD consists of the following behavior:

1. If the process specifically sets its disposition to SIG_IGN, children of the calling process will not generate zombie processes. Note that this is different from its default action (SIG_DFL), which from Figure 10.1 is to be ignored. Instead, on termination, the status of these child processes is discarded. If it subsequently calls one of the wait functions, the calling process will block until all its children have terminated, and then wait returns –1 with errno set to ECHILD. (The default disposition of this signal is to be ignored, but this default will not cause the preceding semantics to occur. Instead, we specifically have to set its disposition to SIG_IGN.)

   POSIX.1 does not specify what happens when SIGCHLD is ignored, so this behavior is allowed. The XSI option requires this behavior to be supported for SIGCHLD.

   4.4BSD always generates zombies if SIGCHLD is ignored. If we want to avoid zombies, we have to wait for our children. With SVR4, if either signal or sigset is called to set the disposition of SIGCHLD to be ignored, zombies are never generated. All four platforms described in this book follow SVR4 in this behavior.

   With sigaction, we can set the SA_NOCLDWAIT flag (Figure 10.16) to avoid zombies. This action is also supported on all four platforms.

2. If we set the disposition of SIGCLD to be caught, the kernel immediately checks whether any child processes are ready to be waited for and, if so, calls the SIGCLD handler.

Item 2 changes the way we have to write a signal handler for this signal, as illustrated in the following example.

Example

Recall from Section 10.4 that the first thing to do on entry to a signal handler is to call signal again, to reestablish the handler. (This action is intended to minimize the window of time when the signal is reset back to its default and could get lost.) We show this in Figure 10.6. This program doesn't work on traditional System V platforms. The output is a continual string of SIGCLD received lines. Eventually, the process runs out of stack space and terminates abnormally.

```
#include    "apue.h"
#include    <sys/wait.h>

static void sig_cld(int);

int
main()
{
    pid_t   pid;

    if (signal(SIGCLD, sig_cld) == SIG_ERR)
        perror("signal error");
    if ((pid = fork()) < 0) {
        perror("fork error");
    } else if (pid == 0) {      /* child */
        sleep(2);
        _exit(0);
    }

    pause();    /* parent */
    exit(0);
}

static void
sig_cld(int signo)  /* interrupts pause() */
{
    pid_t   pid;
    int     status;

    printf("SIGCLD received\n");

    if (signal(SIGCLD, sig_cld) == SIG_ERR) /* reestablish handler */
        perror("signal error");

    if ((pid = wait(&status)) < 0)      /* fetch child status */
        perror("wait error");

    printf("pid = %d\n", pid);
}
```

Figure 10.6 System V `SIGCLD` handler that doesn't work

FreeBSD 8.0 and Mac OS X 10.6.8 don't exhibit this problem, because BSD-based systems generally don't support historical System V semantics for `SIGCLD`. Linux 3.2.0 also doesn't exhibit this problem, because it doesn't call the `SIGCHLD` signal handler when a process arranges to catch `SIGCHLD` and child processes are ready to be `wait`ed for, even though `SIGCLD` and `SIGCHLD` are defined to be the same value. Solaris 10, on the other hand, does call the signal handler in this situation, but includes extra code in the kernel to avoid this problem.

Although the four platforms described in this book solve this problem, realize that platforms (such as AIX) still exist that haven't addressed it.

The problem with this program is that the call to `signal` at the beginning of the signal handler invokes item 2 from the preceding discussion—the kernel checks whether a child needs to be `wait`ed for (which is the case, since we're processing a `SIGCLD` signal), so it generates another call to the signal handler. The signal handler calls `signal`, and the whole process starts over again.

To fix this program, we have to move the call to `signal` after the call to `wait`. By doing this, we call `signal` after fetching the child's termination status; the signal is generated again by the kernel only if some other child has since terminated.

> POSIX.1 states that when we establish a signal handler for `SIGCHLD` and there exists a terminated child we have not yet `wait`ed for, it is unspecified whether the signal is generated. This allows the behavior described previously. But since POSIX.1 does not reset a signal's disposition to its default when the signal occurs (assuming that we're using the POSIX.1 `sigaction` function to set its disposition), there is no need for us to ever establish a signal handler for `SIGCHLD` within that handler.
>
> □

Be cognizant of the `SIGCHLD` semantics for your implementation. Be especially aware of some systems that `#define SIGCHLD` to be `SIGCLD`, or vice versa. Changing the name may allow you to compile a program that was written for another system, but if that program depends on the other semantics, it may not work.

> Of the four platforms described in this text, only Linux 3.2.0 and Solaris 10 define `SIGCLD`. On these platforms, `SIGCLD` is equivalent to `SIGCHLD`.

10.8 Reliable-Signal Terminology and Semantics

We need to define some of the terms used throughout our discussion of signals. First, a signal is *generated* for a process (or sent to a process) when the event that causes the signal occurs. The event could be a hardware exception (e.g., divide by 0), a software condition (e.g., an `alarm` timer expiring), a terminal-generated signal, or a call to the `kill` function. When the signal is generated, the kernel usually sets a flag of some form in the process table.

We say that a signal is *delivered* to a process when the action for a signal is taken. During the time between the generation of a signal and its delivery, the signal is said to be *pending*.

A process has the option of *blocking* the delivery of a signal. If a signal that is blocked is generated for a process, and if the action for that signal is either the default action or to catch the signal, then the signal remains pending for the process until the process either (a) unblocks the signal or (b) changes the action to ignore the signal. The system determines what to do with a blocked signal when the signal is delivered, not when it's generated. This allows the process to change the action for the signal before it's delivered. The `sigpending` function (Section 10.13) can be called by a process to determine which signals are blocked and pending.

What happens if a blocked signal is generated more than once before the process unblocks the signal? POSIX.1 allows the system to deliver the signal either once or more than once. If the system delivers the signal more than once, we say that the signals are queued. Most UNIX systems, however, do *not* queue signals unless they support the real-time extensions to POSIX.1. Instead, the UNIX kernel simply delivers the signal once.

> With SUSv4, the real-time signal functionality moved from the real-time extensions to the base specification. As time goes on, more systems will support queueing signals even if they don't support the real-time extensions. We discuss queueing signals further in Section 10.20.
>
> The manual pages for SVR2 claimed that the `SIGCLD` signal was queued while the process was executing its `SIGCLD` signal handler. Although this might have been true on a conceptual level, the actual implementation was different. Instead, the signal was regenerated by the kernel as we described in Section 10.7. In SVR3, the manual was changed to indicate that the `SIGCLD` signal was ignored while the process was executing its signal handler for `SIGCLD`. The SVR4 manual removed any mention of what happens to `SIGCLD` signals that are generated while a process is executing its `SIGCLD` signal handler.
>
> The SVR4 `sigaction`(2) manual page in AT&T [1990e] claims that the `SA_SIGINFO` flag (Figure 10.16) causes signals to be reliably queued. This is wrong. Apparently, this feature was partially implemented within the kernel, but it is not enabled in SVR4. Curiously, the SVID didn't make the same claims of reliable queuing.

What happens if more than one signal is ready to be delivered to a process? POSIX.1 does not specify the order in which the signals are delivered to the process. The Rationale for POSIX.1 does suggest, however, that signals related to the current state of the process be delivered before other signals. (`SIGSEGV` is one such signal.)

Each process has a *signal mask* that defines the set of signals currently blocked from delivery to that process. We can think of this mask as having one bit for each possible signal. If the bit is on for a given signal, that signal is currently blocked. A process can examine and change its current signal mask by calling `sigprocmask`, which we describe in Section 10.12.

Since it is possible for the number of signals to exceed the number of bits in an integer, POSIX.1 defines a data type, called `sigset_t`, that holds a *signal set*. The signal mask, for example, is stored in one of these signal sets. We describe five functions that operate on signal sets in Section 10.11.

10.9 `kill` and `raise` Functions

The `kill` function sends a signal to a process or a group of processes. The `raise` function allows a process to send a signal to itself.

> The `raise` function was originally defined by ISO C. POSIX.1 includes it to align itself with the ISO C standard, but POSIX.1 extends the specification of `raise` to deal with threads (we discuss how threads interact with signals in Section 12.8). Since ISO C does not deal with multiple processes, it could not define a function, such as `kill`, that requires a process ID argument.

```
#include <signal.h>

int kill(pid_t pid, int signo);

int raise(int signo);
```

Both return: 0 if OK, –1 on error

The call

```
raise(signo);
```

is equivalent to the call

```
kill(getpid(), signo);
```

There are four different conditions for the *pid* argument to `kill`.

pid > 0 The signal is sent to the process whose process ID is *pid*.

pid == 0 The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal. Note that the term *all processes* excludes an implementation-defined set of system processes. For most UNIX systems, this set of system processes includes the kernel processes and `init` (pid 1).

pid < 0 The signal is sent to all processes whose process group ID equals the absolute value of *pid* and for which the sender has permission to send the signal. Again, the set of all processes excludes certain system processes, as described earlier.

pid == –1 The signal is sent to all processes on the system for which the sender has permission to send the signal. As before, the set of processes excludes certain system processes.

As we've mentioned, a process needs permission to send a signal to another process. The superuser can send a signal to any process. For other users, the basic rule is that the real or effective user ID of the sender has to equal the real or effective user ID of the receiver. If the implementation supports `_POSIX_SAVED_IDS` (as POSIX.1 now requires), the saved set-user-ID of the receiver is checked instead of its effective user ID. One special case for the permission testing also exists: if the signal being sent is `SIGCONT`, a process can send it to any other process in the same session.

POSIX.1 defines signal number 0 as the null signal. If the *signo* argument is 0, then the normal error checking is performed by `kill`, but no signal is sent. This technique is often used to determine if a specific process still exists. If we send the process the null signal and it doesn't exist, `kill` returns –1 and `errno` is set to `ESRCH`. Be aware, however, that UNIX systems recycle process IDs after some amount of time, so the existence of a process with a given process ID does not necessarily mean that it's the process that you think it is.

Also understand that the test for process existence is not atomic. By the time that `kill` returns the answer to the caller, the process in question might have exited, so the answer is of limited value.

If the call to `kill` causes the signal to be generated for the calling process and if the signal is not blocked, either *signo* or some other pending, unblocked signal is delivered to the process before `kill` returns. (Additional conditions occur with threads; see Section 12.8 for more information.)

10.10 `alarm` and `pause` Functions

The `alarm` function allows us to set a timer that will expire at a specified time in the future. When the timer expires, the `SIGALRM` signal is generated. If we ignore or don't catch this signal, its default action is to terminate the process.

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

Returns: 0 or number of seconds until previously set alarm

The *seconds* value is the number of clock seconds in the future when the signal should be generated. When that time occurs, the signal is generated by the kernel, although additional time could elapse before the process gets control to handle the signal, because of processor scheduling delays.

> Earlier UNIX System implementations warned that the signal could also be sent up to 1 second early. POSIX.1 does not allow this behavior.

There is only one of these alarm clocks per process. If, when we call `alarm`, a previously registered alarm clock for the process has not yet expired, the number of seconds left for that alarm clock is returned as the value of this function. That previously registered alarm clock is replaced by the new value.

If a previously registered alarm clock for the process has not yet expired and if the *seconds* value is 0, the previous alarm clock is canceled. The number of seconds left for that previous alarm clock is still returned as the value of the function.

Although the default action for `SIGALRM` is to terminate the process, most processes that use an alarm clock catch this signal. If the process then wants to terminate, it can perform whatever cleanup is required before terminating. If we intend to catch `SIGALRM`, we need to be careful to install its signal handler before calling `alarm`. If we call `alarm` first and are sent `SIGALRM` before we can install the signal handler, our process will terminate.

The `pause` function suspends the calling process until a signal is caught.

```
#include <unistd.h>

int pause(void);
```

Returns: –1 with `errno` set to `EINTR`

The only time pause returns is if a signal handler is executed and that handler returns. In that case, pause returns –1 with errno set to EINTR.

Example

Using alarm and pause, we can put a process to sleep for a specified amount of time. The sleep1 function in Figure 10.7 appears to do this (but it has problems, as we shall see shortly).

```
#include    <signal.h>
#include    <unistd.h>

static void
sig_alrm(int signo)
{
    /* nothing to do, just return to wake up the pause */
}

unsigned int
sleep1(unsigned int seconds)
{
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(seconds);
    alarm(seconds);     /* start the timer */
    pause();            /* next caught signal wakes us up */
    return(alarm(0));   /* turn off timer, return unslept time */
}
```

Figure 10.7 Simple, incomplete implementation of sleep

This function looks like the sleep function, which we describe in Section 10.19, but this simple implementation has three problems.

1. If the caller already has an alarm set, that alarm is erased by the first call to alarm. We can correct this by looking at alarm's return value. If the number of seconds until some previously set alarm is less than the argument, then we should wait only until the existing alarm expires. If the previously set alarm will go off after ours, then before returning we should reset this alarm to occur at its designated time in the future.

2. We have modified the disposition for SIGALRM. If we're writing a function for others to call, we should save the disposition when our function is called and restore it when we're done. We can correct this by saving the return value from signal and resetting the disposition before our function returns.

3. There is a race condition between the first call to alarm and the call to pause. On a busy system, it's possible for the alarm to go off and the signal handler to be called before we call pause. If that happens, the caller is suspended forever in the call to pause (assuming that some other signal isn't caught).

Earlier implementations of `sleep` looked like our program, with problems 1 and 2 corrected as described. There are two ways to correct problem 3. The first uses `setjmp`, which we show in the next example. The other uses `sigprocmask` and `sigsuspend`, and we describe it in Section 10.19. □

Example

The SVR2 implementation of `sleep` used `setjmp` and `longjmp` (Section 7.10) to avoid the race condition described in problem 3 of the previous example. A simple version of this function, called `sleep2`, is shown in Figure 10.8. (To reduce the size of this example, we don't handle problems 1 and 2 described earlier.)

```
#include    <setjmp.h>
#include    <signal.h>
#include    <unistd.h>

static jmp_buf  env_alrm;

static void
sig_alrm(int signo)
{
    longjmp(env_alrm, 1);
}

unsigned int
sleep2(unsigned int seconds)
{
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(seconds);
    if (setjmp(env_alrm) == 0) {
        alarm(seconds);     /* start the timer */
        pause();            /* next caught signal wakes us up */
    }
    return(alarm(0));       /* turn off timer, return unslept time */
}
```

Figure 10.8 Another (imperfect) implementation of `sleep`

The `sleep2` function avoids the race condition from Figure 10.7. Even if the `pause` is never executed, the `sleep2` function returns when the `SIGALRM` occurs.

There is, however, another subtle problem with the `sleep2` function that involves its interaction with other signals. If the `SIGALRM` interrupts some other signal handler, then when we call `longjmp`, we abort the other signal handler. Figure 10.9 shows this scenario. The loop in the `SIGINT` handler was written so that it executes for longer than 5 seconds on one of the systems used by the author. We simply want it to execute longer than the argument to `sleep2`. The integer `k` is declared as `volatile` to prevent an optimizing compiler from discarding the loop.

```
#include "apue.h"

unsigned int    sleep2(unsigned int);
static void     sig_int(int);

int
main(void)
{
    unsigned int    unslept;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    unslept = sleep2(5);
    printf("sleep2 returned: %u\n", unslept);
    exit(0);
}

static void
sig_int(int signo)
{
    int             i, j;
    volatile int    k;

    /*
     * Tune these loops to run for more than 5 seconds
     * on whatever system this test program is run.
     */
    printf("\nsig_int starting\n");
    for (i = 0; i < 300000; i++)
        for (j = 0; j < 4000; j++)
            k += i * j;
    printf("sig_int finished\n");
}
```

Figure 10.9 Calling `sleep2` from a program that catches other signals

When we execute the program shown in Figure 10.9 and interrupt the sleep by typing the interrupt character, we get the following output:

```
$ ./a.out
^C                              we type the interrupt character
sig_int starting
sleep2 returned: 0
```

We can see that the `longjmp` from the `sleep2` function aborted the other signal handler, `sig_int`, even though it wasn't finished. This is what you'll encounter if you mix the SVR2 `sleep` function with other signal handling. See Exercise 10.3. □

The purpose of the `sleep1` and `sleep2` examples is to show the pitfalls in dealing naively with signals. The following sections will show ways around all these problems, so we can handle signals reliably, without interfering with other pieces of code.

Example

A common use for `alarm`, in addition to implementing the `sleep` function, is to put an upper time limit on operations that can block. For example, if we have a `read` operation on a device that can block (a "slow" device, as described in Section 10.5), we might want the `read` to time out after some amount of time. The program in Figure 10.10 does this, reading one line from standard input and writing it to standard output.

```
#include "apue.h"

static void sig_alrm(int);

int
main(void)
{
    int     n;
    char    line[MAXLINE];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");

    alarm(10);
    if ((n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("read error");
    alarm(0);

    write(STDOUT_FILENO, line, n);
    exit(0);
}

static void
sig_alrm(int signo)
{
    /* nothing to do, just return to interrupt the read */
}
```

Figure 10.10 Calling `read` with a timeout

This sequence of code is common in UNIX applications, but this program has two problems.

1. The program in Figure 10.10 has one of the same flaws that we described in Figure 10.7: a race condition between the first call to `alarm` and the call to `read`. If the kernel blocks the process between these two function calls for longer than the alarm period, the `read` could block forever. Most operations of this type use a long alarm period, such as a minute or more, making this unlikely; nevertheless, it is a race condition.
2. If system calls are automatically restarted, the `read` is not interrupted when the `SIGALRM` signal handler returns. In this case, the timeout does nothing.

Here, we specifically want a slow system call to be interrupted. We'll see a portable way to do this in Section 10.14. □

Example

Let's redo the preceding example using `longjmp`. This way, we don't need to worry about whether a slow system call is interrupted.

```
#include "apue.h"
#include <setjmp.h>

static void     sig_alrm(int);
static jmp_buf  env_alrm;

int
main(void)
{
    int     n;
    char    line[MAXLINE];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");
    if (setjmp(env_alrm) != 0)
        err_quit("read timeout");

    alarm(10);
    if ((n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("read error");
    alarm(0);

    write(STDOUT_FILENO, line, n);
    exit(0);
}

static void
sig_alrm(int signo)
{
    longjmp(env_alrm, 1);
}
```

Figure 10.11 Calling `read` with a timeout, using `longjmp`

This version works as expected, regardless of whether the system restarts interrupted system calls. Realize, however, that we still have the problem of interactions with other signal handlers, as in Figure 10.8. □

If we want to set a time limit on an I/O operation, we need to use `longjmp`, as shown previously, while recognizing its possible interaction with other signal handlers. Another option is to use the `select` or `poll` functions, described in Sections 14.4.1 and 14.4.2.

10.11 Signal Sets

We need a data type to represent multiple signals—a *signal set*. We'll use this data type with such functions as `sigprocmask` (in the next section) to tell the kernel not to allow any of the signals in the set to occur. As we mentioned earlier, the number of different signals can exceed the number of bits in an integer, so in general we can't use an integer to represent the set with one bit per signal. POSIX.1 defines the data type `sigset_t` to contain a signal set and the following five functions to manipulate signal sets.

```
#include <signal.h>

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signo);

int sigdelset(sigset_t *set, int signo);
```

All four return: 0 if OK, –1 on error

```
int sigismember(const sigset_t *set, int signo);
```

Returns: 1 if true, 0 if false, –1 on error

The function `sigemptyset` initializes the signal set pointed to by *set* so that all signals are excluded. The function `sigfillset` initializes the signal set so that all signals are included. All applications have to call either `sigemptyset` or `sigfillset` once for each signal set, before using the signal set, because we cannot assume that the C initialization for external and static variables (0) corresponds to the implementation of signal sets on a given system.

Once we have initialized a signal set, we can add and delete specific signals in the set. The function `sigaddset` adds a single signal to an existing set, and `sigdelset` removes a single signal from a set. In all the functions that take a signal set as an argument, we always pass the address of the signal set as the argument.

Implementation

If the implementation has fewer signals than bits in an integer, a signal set can be implemented using one bit per signal. For the remainder of this section, assume that an implementation has 31 signals and 32-bit integers. The `sigemptyset` function zeros the integer, and the `sigfillset` function turns on all the bits in the integer. These two functions can be implemented as macros in the `<signal.h>` header:

```
#define sigemptyset(ptr)  (*(ptr) = 0)
#define sigfillset(ptr)   (*(ptr) = ~(sigset_t)0, 0)
```

Note that `sigfillset` must return 0, in addition to setting all the bits on in the signal set, so we use C's comma operator, which returns the value after the comma as the value of the expression.

Using this implementation, `sigaddset` turns on a single bit and `sigdelset` turns off a single bit; `sigismember` tests a certain bit. Since no signal is ever numbered 0, we subtract 1 from the signal number to obtain the bit to manipulate. Figure 10.12 shows implementations of these functions.

```
#include    <signal.h>
#include    <errno.h>

/*
 * <signal.h> usually defines NSIG to include signal number 0.
 */
#define SIGBAD(signo)   ((signo) <= 0 || (signo) >= NSIG)

int
sigaddset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) {
        errno = EINVAL;
        return(-1);
    }
    *set |= 1 << (signo - 1);       /* turn bit on */
    return(0);
}

int
sigdelset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) {
        errno = EINVAL;
        return(-1);
    }
    *set &= ~(1 << (signo - 1));    /* turn bit off */
    return(0);
}

int
sigismember(const sigset_t *set, int signo)
{
    if (SIGBAD(signo)) {
        errno = EINVAL;
        return(-1);
    }
    return((*set & (1 << (signo - 1))) != 0);
}
```

Figure 10.12 An implementation of `sigaddset`, `sigdelset`, and `sigismember`

We might be tempted to implement these three functions as one-line macros in the `<signal.h>` header, but POSIX.1 requires us to check the signal number argument for validity and to set `errno` if it is invalid. This is more difficult to do in a macro than in a function.

10.12 sigprocmask Function

Recall from Section 10.8 that the signal mask of a process is the set of signals currently blocked from delivery to that process. A process can examine its signal mask, change its signal mask, or perform both operations in one step by calling the following function.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *restrict set,
                sigset_t *restrict oset);
```

Returns: 0 if OK, –1 on error

First, if *oset* is a non-null pointer, the current signal mask for the process is returned through *oset*.

Second, if *set* is a non-null pointer, the *how* argument indicates how the current signal mask is modified. Figure 10.13 describes the possible values for *how*. SIG_BLOCK is an inclusive-OR operation, whereas SIG_SETMASK is an assignment. Note that SIGKILL and SIGSTOP can't be blocked.

how	Description
SIG_BLOCK	The new signal mask for the process is the union of its current signal mask and the signal set pointed to by *set*. That is, *set* contains the additional signals that we want to block.
SIG_UNBLOCK	The new signal mask for the process is the intersection of its current signal mask and the complement of the signal set pointed to by *set*. That is, *set* contains the signals that we want to unblock.
SIG_SETMASK	The new signal mask for the process is replaced by the value of the signal set pointed to by *set*.

Figure 10.13 Ways to change the current signal mask using sigprocmask

If *set* is a null pointer, the signal mask of the process is not changed, and *how* is ignored.

After calling sigprocmask, if any unblocked signals are pending, at least one of these signals is delivered to the process before sigprocmask returns.

> The sigprocmask function is defined only for single-threaded processes. A separate function is provided to manipulate a thread's signal mask in a multithreaded process. We'll discuss this in Section 12.8.

Example

Figure 10.14 shows a function that prints the names of the signals in the signal mask of the calling process. We call this function from the programs shown in Figure 10.20 and Figure 10.22.

```
#include "apue.h"
#include <errno.h>

void
pr_mask(const char *str)
{
    sigset_t    sigset;
    int         errno_save;

    errno_save = errno;     /* we can be called by signal handlers */
    if (sigprocmask(0, NULL, &sigset) < 0) {
        err_ret("sigprocmask error");
    } else {
        printf("%s", str);
        if (sigismember(&sigset, SIGINT))
            printf(" SIGINT");
        if (sigismember(&sigset, SIGQUIT))
            printf(" SIGQUIT");
        if (sigismember(&sigset, SIGUSR1))
            printf(" SIGUSR1");
        if (sigismember(&sigset, SIGALRM))
            printf(" SIGALRM");

        /* remaining signals can go here  */

        printf("\n");
    }

    errno = errno_save;     /* restore errno */
}
```

Figure 10.14 Print the signal mask for the process

To save space, we don't test the signal mask for every signal that we listed in Figure 10.1. (See Exercise 10.9.) □

10.13 sigpending Function

The `sigpending` function returns the set of signals that are blocked from delivery and currently pending for the calling process. The set of signals is returned through the *set* argument.

```
#include <signal.h>

int sigpending(sigset_t *set);
```

Returns: 0 if OK, –1 on error

Example

Figure 10.15 shows many of the signal features that we've been describing.

```
#include "apue.h"

static void sig_quit(int);

int
main(void)
{
    sigset_t    newmask, oldmask, pendmask;

    if (signal(SIGQUIT, sig_quit) == SIG_ERR)
        err_sys("can't catch SIGQUIT");

    /*
     * Block SIGQUIT and save current signal mask.
     */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    sleep(5);   /* SIGQUIT here will remain pending */

    if (sigpending(&pendmask) < 0)
        err_sys("sigpending error");
    if (sigismember(&pendmask, SIGQUIT))
        printf("\nSIGQUIT pending\n");

    /*
     * Restore signal mask which unblocks SIGQUIT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    printf("SIGQUIT unblocked\n");

    sleep(5);   /* SIGQUIT here will terminate with core file */
    exit(0);
}

static void
sig_quit(int signo)
{
    printf("caught SIGQUIT\n");
    if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
        err_sys("can't reset SIGQUIT");
}
```

Figure 10.15 Example of signal sets and `sigprocmask`

The process blocks SIGQUIT, saving its current signal mask (to restore later), and then goes to sleep for 5 seconds. Any occurrence of the quit signal during this period is blocked and won't be delivered until the signal is unblocked. At the end of the 5-second sleep, we check whether the signal is pending and unblock the signal.

Note that we saved the old mask when we blocked the signal. To unblock the signal, we did a SIG_SETMASK of the old mask. Alternatively, we could SIG_UNBLOCK only the signal that we had blocked. Be aware, however, if we write a function that can be called by others and if we need to block a signal in our function, we can't use SIG_UNBLOCK to unblock the signal. In this case, we have to use SIG_SETMASK and restore the signal mask to its prior value, because it's possible that the caller had specifically blocked this signal before calling our function. We'll see an example of this in the system function in Section 10.18.

If we generate the quit signal during this sleep period, the signal is now pending and unblocked, so it is delivered before sigprocmask returns. We'll see this occur because the printf in the signal handler is output before the printf that follows the call to sigprocmask.

The process then goes to sleep for another 5 seconds. If we generate the quit signal during this sleep period, the signal should terminate the process, since we reset the handling of the signal to its default when we caught it. In the following output, the terminal prints ^\ when we input Control-backslash, the terminal quit character:

```
$ ./a.out
^\                          generate signal once (before 5 seconds are up)
SIGQUIT pending             after return from sleep
caught SIGQUIT              in signal handler
SIGQUIT unblocked           after return from sigprocmask
^\Quit(coredump)            generate signal again
$ ./a.out
^\^\^\^\^\^\^\^\^\^\        generate signal 10 times (before 5 seconds are up)
SIGQUIT pending
caught SIGQUIT              signal is generated only once
SIGQUIT unblocked
^\Quit(coredump)            generate signal again
```

The message Quit(coredump) is printed by the shell when it sees that its child terminated abnormally. Note that when we run the program the second time, we generate the quit signal ten times while the process is asleep, yet the signal is delivered only once to the process when it's unblocked. This demonstrates that signals are not queued on this system. □

10.14 sigaction Function

The sigaction function allows us to examine or modify (or both) the action associated with a particular signal. This function supersedes the signal function from earlier releases of the UNIX System. Indeed, at the end of this section, we show an implementation of signal using sigaction.

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *restrict act,
              struct sigaction *restrict oact);
```

Returns: 0 if OK, –1 on error

The argument *signo* is the signal number whose action we are examining or modifying. If the *act* pointer is non-null, we are modifying the action. If the *oact* pointer is non-null, the system returns the previous action for the signal through the *oact* pointer. This function uses the following structure:

```
struct sigaction {
  void     (*sa_handler)(int);  /* addr of signal handler, */
                                /* or SIG_IGN, or SIG_DFL */
  sigset_t sa_mask;             /* additional signals to block */
  int      sa_flags;            /* signal options, Figure 10.16 */

  /* alternate handler */
  void     (*sa_sigaction)(int, siginfo_t *, void *);
};
```

When changing the action for a signal, if the `sa_handler` field contains the address of a signal-catching function (as opposed to either of the constants `SIG_IGN` or `SIG_DFL`), then the `sa_mask` field specifies a set of signals that are added to the signal mask of the process before the signal-catching function is called. If and when the signal-catching function returns, the signal mask of the process is reset to its previous value. This way, we are able to block certain signals whenever a signal handler is invoked. The operating system includes the signal being delivered in the signal mask when the handler is invoked. Hence, we are guaranteed that whenever we are processing a given signal, another occurrence of that same signal is blocked until we're finished processing the first occurrence. Recall from Section 10.8 that additional occurrences of the same signal are usually not queued. If the signal occurs five times while it is blocked, when we unblock the signal, the signal-handling function for that signal will usually be invoked only one time. (This characteristic was illustrated in the previous example.)

Once we install an action for a given signal, that action remains installed until we explicitly change it by calling `sigaction`. Unlike earlier systems with their unreliable signals, POSIX.1 requires that a signal handler remain installed until explicitly changed.

The `sa_flags` field of the *act* structure specifies various options for the handling of this signal. Figure 10.16 details the meaning of these options when set. The SUS column contains • if the flag is defined as part of the base POSIX.1 specification, and **XSI** if it is defined as part of the XSI option.

The `sa_sigaction` field is an alternative signal handler used when the `SA_SIGINFO` flag is used with `sigaction`. Implementations might use the same storage for both the `sa_sigaction` field and the `sa_handler` field, so applications can use only one of these fields at a time.

Option	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	Description
SA_INTERRUPT			•			System calls interrupted by this signal are not automatically restarted (the XSI default for sigaction). See Section 10.5 for more information.
SA_NOCLDSTOP	•	•	•	•	•	If *signo* is SIGCHLD, do not generate this signal when a child process stops (job control). This signal is still generated, of course, when a child terminates (but see the SA_NOCLDWAIT option below). When the XSI option is supported, SIGCHLD won't be sent when a stopped child continues if this flag is set.
SA_NOCLDWAIT	•	•	•	•	•	If *signo* is SIGCHLD, this option prevents the system from creating zombie processes when children of the calling process terminate. If it subsequently calls wait, the calling process blocks until all its child processes have terminated and then returns –1 with errno set to ECHILD. (Recall Section 10.7.)
SA_NODEFER	•	•	•	•	•	When this signal is caught, the signal is not automatically blocked by the system while the signal-catching function executes (unless the signal is also included in sa_mask). Note that this type of operation corresponds to the earlier unreliable signals.
SA_ONSTACK	XSI	•	•	•	•	If an alternative stack has been declared with sigaltstack(2), this signal is delivered to the process on the alternative stack.
SA_RESETHAND	•	•	•	•	•	The disposition for this signal is reset to SIG_DFL, and the SA_SIGINFO flag is cleared on entry to the signal-catching function. Note that this type of operation corresponds to the earlier unreliable signals. The disposition for the two signals SIGILL and SIGTRAP can't be reset automatically, however. Setting this flag can optionally cause sigaction to behave as if SA_NODEFER is also set.
SA_RESTART	•	•	•	•	•	System calls interrupted by this signal are automatically restarted. (Refer to Section 10.5.)
SA_SIGINFO	•	•	•	•	•	This option provides additional information to a signal handler: a pointer to a siginfo structure and a pointer to an identifier for the process context.

Figure 10.16 Option flags (sa_flags) for the handling of each signal

Normally, the signal handler is called as

```
void handler(int signo);
```

but if the SA_SIGINFO flag is set, the signal handler is called as

```
void handler(int signo, siginfo_t *info, void *context);
```

The siginfo structure contains information about why the signal was generated. An example of what it might look like is shown below. All POSIX.1-compliant implementations must include at least the si_signo and si_code members. Additionally, implementations that are XSI compliant contain at least the following fields:

```
struct siginfo {
  int          si_signo;  /* signal number */
  int          si_errno;  /* if nonzero, errno value from errno.h */
  int          si_code;   /* additional info (depends on signal) */
  pid_t        si_pid;    /* sending process ID */
  uid_t        si_uid;    /* sending process real user ID */
  void        *si_addr;   /* address that caused the fault */
  int          si_status; /* exit value or signal number */
  union sigval si_value;  /* application-specific value */
  /* possibly other fields also */
};
```

The sigval union contains the following fields:

```
  int   sival_int;
  void *sival_ptr;
```

Applications pass an integer value in si_value.sival_int or pass a pointer value in si_value.sival_ptr when delivering signals.

Figure 10.17 shows values of si_code for various signals, as defined by the Single UNIX Specification. Note that implementations may define additional code values.

If the signal is SIGCHLD, then the si_pid, si_status, and si_uid fields will be set. If the signal is SIGBUS, SIGILL, SIGFPE, or SIGSEGV, then the si_addr contains the address responsible for the fault, although the address might not be accurate. The si_errno field contains the error number corresponding to the condition that caused the signal to be generated, although its use is implementation defined.

The *context* argument to the signal handler is a typeless pointer that can be cast to a ucontext_t structure identifying the process context at the time of signal delivery. This structure contains at least the following fields:

```
ucontext_t *uc_link;     /* pointer to context resumed when */
                         /* this context returns */
sigset_t    uc_sigmask;  /* signals blocked when this context */
                         /* is active */
stack_t     uc_stack;    /* stack used by this context */
mcontext_t  uc_mcontext; /* machine-specific representation of */
                         /* saved context */
```

The uc_stack field describes the stack used by the current context. It contains at least the following members:

```
void  *ss_sp;    /* stack base or pointer */
size_t ss_size;  /* stack size */
int    ss_flags; /* flags */
```

When an implementation supports the real-time signal extensions, signal handlers established with the SA_SIGINFO flag will result in signals being queued reliably. A separate range of reserved signal numbers is available for real-time application use. Applications can pass information along with the signal by using the sigqueue function (Section 10.20).

Signal	Code	Reason
SIGILL	ILL_ILLOPC	illegal opcode
	ILL_ILLOPN	illegal operand
	ILL_ILLADR	illegal addressing mode
	ILL_ILLTRP	illegal trap
	ILL_PRVOPC	privileged opcode
	ILL_PRVREG	privileged register
	ILL_COPROC	coprocessor error
	ILL_BADSTK	internal stack error
SIGFPE	FPE_INTDIV	integer divide by zero
	FPE_INTOVF	integer overflow
	FPE_FLTDIV	floating-point divide by zero
	FPE_FLTOVF	floating-point overflow
	FPE_FLTUND	floating-point underflow
	FPE_FLTRES	floating-point inexact result
	FPE_FLTINV	invalid floating-point operation
	FPE_FLTSUB	subscript out of range
SIGSEGV	SEGV_MAPERR	address not mapped to object
	SEGV_ACCERR	invalid permissions for mapped object
SIGBUS	BUS_ADRALN	invalid address alignment
	BUS_ADRERR	nonexistent physical address
	BUS_OBJERR	object-specific hardware error
SIGTRAP	TRAP_BRKPT	process breakpoint trap
	TRAP_TRACE	process trace trap
SIGCHLD	CLD_EXITED	child has exited
	CLD_KILLED	child has terminated abnormally (no core)
	CLD_DUMPED	child has terminated abnormally with core
	CLD_TRAPPED	traced child has trapped
	CLD_STOPPED	child has stopped
	CLD_CONTINUED	stopped child has continued
Any	SI_USER	signal sent by kill
	SI_QUEUE	signal sent by sigqueue
	SI_TIMER	expiration of a timer set by timer_settime
	SI_ASYNCIO	completion of asynchronous I/O request
	SI_MESGQ	arrival of a message on a message queue (real-time extension)

Figure 10.17 siginfo_t code values

Example — `signal` Function

Let's now implement the `signal` function using `sigaction`. This is what many platforms do (and what a note in the POSIX.1 Rationale states was the intent of POSIX). Systems with binary compatibility constraints, on the other hand, might provide a `signal` function that supports the older, unreliable-signal semantics. Unless you specifically require these older, unreliable semantics (for backward compatibility), you should use the following implementation of `signal` or call `sigaction` directly. (As you might guess, an implementation of `signal` with the old semantics could call `sigaction` specifying `SA_RESETHAND` and `SA_NODEFER`.) All the examples in this text that call `signal` call the function shown in Figure 10.18.

```
#include "apue.h"

/* Reliable version of signal(), using POSIX sigaction().  */
Sigfunc *
signal(int signo, Sigfunc *func)
{
    struct sigaction    act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (signo == SIGALRM) {
#ifdef  SA_INTERRUPT
        act.sa_flags |= SA_INTERRUPT;
#endif
    } else {
        act.sa_flags |= SA_RESTART;
    }
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

Figure 10.18 An implementation of `signal` using `sigaction`

Note that we must use `sigemptyset` to initialize the `sa_mask` member of the structure. We're not guaranteed that `act.sa_mask = 0` does the same thing.

We intentionally set the `SA_RESTART` flag for all signals other than `SIGALRM`, so that any system call interrupted by these other signals will be automatically restarted. The reason we don't want `SIGALRM` restarted is to allow us to set a timeout for I/O operations. (Recall the discussion of Figure 10.10.)

Some older systems, such as SunOS, define the `SA_INTERRUPT` flag. These systems restart interrupted system calls by default, so specifying this flag causes system calls to be interrupted. Linux defines the `SA_INTERRUPT` flag for compatibility with applications that use it, but by default does not restart system calls when the signal handler is installed with `sigaction`. The Single UNIX Specification specifies that the `sigaction` function not restart interrupted system calls unless the `SA_RESTART` flag is specified. □

Example — signal_intr Function

Figure 10.19 shows a version of the signal function that tries to prevent any interrupted system calls from being restarted.

```
#include "apue.h"

Sigfunc *
signal_intr(int signo, Sigfunc *func)
{
    struct sigaction    act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
#ifdef  SA_INTERRUPT
    act.sa_flags |= SA_INTERRUPT;
#endif
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

Figure 10.19 The signal_intr function

For improved portability, we specify the SA_INTERRUPT flag, if defined by the system, to prevent interrupted system calls from being restarted. □

10.15 sigsetjmp and siglongjmp Functions

In Section 7.10, we described the setjmp and longjmp functions, which can be used for nonlocal branching. The longjmp function is often called from a signal handler to return to the main loop of a program, instead of returning from the handler. We saw this approach in Figures 10.8 and 10.11.

There is a problem in calling longjmp, however. When a signal is caught, the signal-catching function is entered, with the current signal automatically being added to the signal mask of the process. This prevents subsequent occurrences of that signal from interrupting the signal handler. If we longjmp out of the signal handler, what happens to the signal mask for the process?

> Under FreeBSD 8.0 and Mac OS X 10.6.8, setjmp and longjmp save and restore the signal mask. Linux 3.2.0 and Solaris 10, however, do not do this, although Linux supports an option to provide BSD behavior. FreeBSD and Mac OS X provide the functions _setjmp and _longjmp, which do not save and restore the signal mask.

To allow either form of behavior, POSIX.1 does not specify the effect of setjmp and longjmp on signal masks. Instead, two new functions, sigsetjmp and siglongjmp, are defined by POSIX.1. These two functions should always be used when branching from a signal handler.

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);

            Returns: 0 if called directly, nonzero if returning from a call to siglongjmp

void siglongjmp(sigjmp_buf env, int val);
```

The only difference between these functions and the `setjmp` and `longjmp` functions is that `sigsetjmp` has an additional argument. If *savemask* is nonzero, then `sigsetjmp` also saves the current signal mask of the process in *env*. When `siglongjmp` is called, if the *env* argument was saved by a call to `sigsetjmp` with a nonzero *savemask*, then `siglongjmp` restores the saved signal mask.

Example

The program in Figure 10.20 demonstrates how the signal mask that is installed by the system when a signal handler is invoked automatically includes the signal being caught. This program also illustrates the use of the `sigsetjmp` and `siglongjmp` functions.

```
#include "apue.h"
#include <setjmp.h>
#include <time.h>

static void                     sig_usr1(int);
static void                     sig_alrm(int);
static sigjmp_buf               jmpbuf;
static volatile sig_atomic_t    canjump;

int
main(void)
{
    if (signal(SIGUSR1, sig_usr1) == SIG_ERR)
        err_sys("signal(SIGUSR1) error");
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");

    pr_mask("starting main: ");     /* Figure 10.14 */

    if (sigsetjmp(jmpbuf, 1)) {

        pr_mask("ending main: ");

        exit(0);
    }
    canjump = 1;    /* now sigsetjmp() is OK */

    for ( ; ; )
        pause();
}

static void
sig_usr1(int signo)
```

```
{
    time_t  starttime;

    if (canjump == 0)
        return;     /* unexpected signal, ignore */

    pr_mask("starting sig_usr1: ");

    alarm(3);               /* SIGALRM in 3 seconds */
    starttime = time(NULL);
    for ( ; ; )             /* busy wait for 5 seconds */
        if (time(NULL) > starttime + 5)
            break;

    pr_mask("finishing sig_usr1: ");

    canjump = 0;
    siglongjmp(jmpbuf, 1);  /* jump back to main, don't return */
}

static void
sig_alrm(int signo)
{
    pr_mask("in sig_alrm: ");
}
```

Figure 10.20 Example of signal masks, `sigsetjmp`, and `siglongjmp`

This program demonstrates another technique that should be used whenever `siglongjmp` is called from a signal handler. We set the variable `canjump` to a nonzero value only after we've called `sigsetjmp`. This variable is examined in the signal handler, and `siglongjmp` is called only if the flag `canjump` is nonzero. This technique provides protection against the signal handler being called at some earlier or later time, when the jump buffer hasn't been initialized by `sigsetjmp`. (In this trivial program, we terminate quickly after the `siglongjmp`, but in larger programs, the signal handler may remain installed long after the `siglongjmp`.) Providing this type of protection usually isn't required with `longjmp` in normal C code (as opposed to a signal handler). Since a signal can occur at *any* time, however, we need the added protection in a signal handler.

Here, we use the data type `sig_atomic_t`, which is defined by the ISO C standard to be the type of variable that can be written without being interrupted. By this we mean that a variable of this type should not extend across page boundaries on a system with virtual memory and can be accessed with a single machine instruction, for example. We always include the ISO type qualifier `volatile` for these data types as well, since the variable is being accessed by two different threads of control: the `main` function and the asynchronously executing signal handler. Figure 10.21 shows a timeline for this program. We can divide Figure 10.21 into three parts: the left part (corresponding to `main`), the center part (`sig_usr1`), and the right part (`sig_alrm`). While the process is executing in the left part, its signal mask is 0 (no signals are blocked). While executing in the center part, its signal mask is `SIGUSR1`. While executing in the right part, its signal mask is `SIGUSR1 | SIGALRM`.

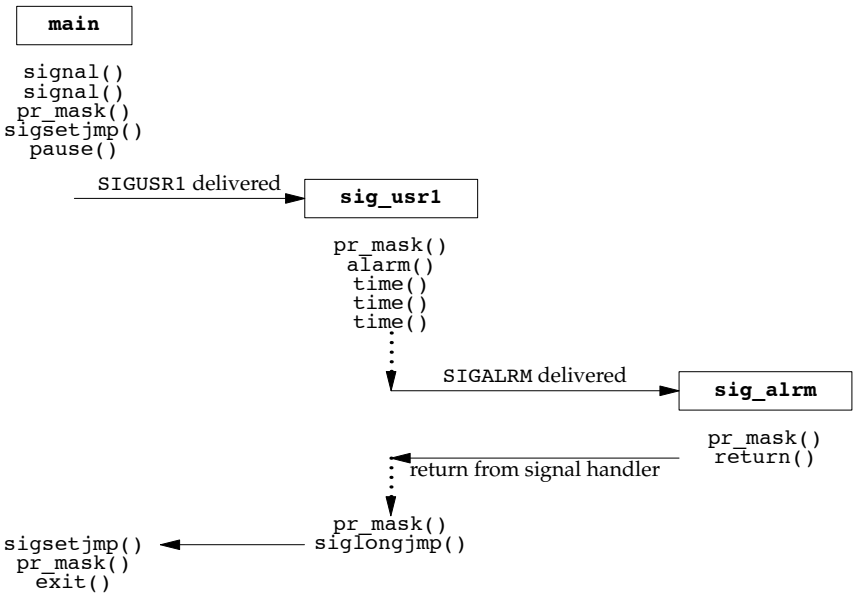


Figure 10.21 Timeline for example program handling two signals

Let's examine the output when the program in Figure 10.20 is executed:

```
$ ./a.out &                          start process in background
starting main:
[1]   531                            the job-control shell prints its process ID
$ kill -USR1 531                     send the process SIGUSR1
starting sig_usr1: SIGUSR1
$ in sig_alrm: SIGUSR1 SIGALRM
finishing sig_usr1: SIGUSR1
ending main:
                                     just press RETURN
[1] +  Done          ./a.out &
```

The output is what we expect: when a signal handler is invoked, the signal being caught is added to the current signal mask of the process. The original mask is restored when the signal handler returns. Also, `siglongjmp` restores the signal mask that was saved by `sigsetjmp`.

If we change the program in Figure 10.20 so that the calls to `sigsetjmp` and `siglongjmp` are replaced with calls to `setjmp` and `longjmp` on Linux (or `_setjmp` and `_longjmp` on FreeBSD), the final line of output becomes

```
ending main: SIGUSR1
```

This means that the `main` function is executing with the `SIGUSR1` signal blocked, after the call to `setjmp`. This probably isn't what we want. □

10.16 sigsuspend Function

We have seen how we can change the signal mask for a process to block and unblock selected signals. We can use this technique to protect critical regions of code that we don't want interrupted by a signal. But what if we want to unblock a signal and then pause, waiting for the previously blocked signal to occur? Assuming that the signal is SIGINT, the incorrect way to do this is

```
sigset_t    newmask, oldmask;

sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

/* block SIGINT and save current signal mask */
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
    err_sys("SIG_BLOCK error");

/* critical region of code */

/* restore signal mask, which unblocks SIGINT */
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
    err_sys("SIG_SETMASK error");

/* window is open */
pause();  /* wait for signal to occur */

/* continue processing */
```

If the signal is sent to the process while it is blocked, the signal delivery will be deferred until the signal is unblocked. To the application, this can look as if the signal occurs between the unblocking and the pause (depending on how the kernel implements signals). If this happens, or if the signal does occur between the unblocking and the pause, we have a problem. Any occurrence of the signal in this window of time is lost, in the sense that we might not see the signal again, in which case the pause will block indefinitely. This is another problem with the earlier unreliable signals.

To correct this problem, we need a way to both restore the signal mask and put the process to sleep in a single atomic operation. This feature is provided by the sigsuspend function.

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

Returns: –1 with errno set to EINTR

The signal mask of the process is set to the value pointed to by *sigmask*. Then the process is suspended until a signal is caught or until a signal occurs that terminates the process. If a signal is caught and if the signal handler returns, then sigsuspend returns, and the signal mask of the process is set to its value before the call to sigsuspend.

Note that there is no successful return from this function. If it returns to the caller, it always returns –1 with errno set to EINTR (indicating an interrupted system call).

Example

Figure 10.22 shows the correct way to protect a critical region of code from a specific signal.

```
#include "apue.h"

static void sig_int(int);

int
main(void)
{
    sigset_t    newmask, oldmask, waitmask;

    pr_mask("program start: ");

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    sigemptyset(&waitmask);
    sigaddset(&waitmask, SIGUSR1);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGINT);

    /*
     * Block SIGINT and save current signal mask.
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    /*
     * Critical region of code.
     */
    pr_mask("in critical region: ");

    /*
     * Pause, allowing all signals except SIGUSR1.
     */
    if (sigsuspend(&waitmask) != -1)
        err_sys("sigsuspend error");

    pr_mask("after return from sigsuspend: ");

    /*
     * Reset signal mask which unblocks SIGINT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");

    /*
     * And continue processing ...
     */
    pr_mask("program exit: ");

    exit(0);
}
```

```
static void
sig_int(int signo)
{
    pr_mask("\nin sig_int: ");
}
```

Figure 10.22 Protecting a critical region from a signal

When sigsuspend returns, it sets the signal mask to its value before the call. In this example, the SIGINT signal will be blocked, so we restore the signal mask to the value that we saved earlier (oldmask).

Running the program from Figure 10.22 produces the following output:

```
$ ./a.out
program start:
in critical region: SIGINT
^C                                    type the interrupt character
in sig_int: SIGINT SIGUSR1
after return from sigsuspend: SIGINT
program exit:
```

We added SIGUSR1 to the mask installed when we called sigsuspend so that when the signal handler ran, we could tell that the mask had actually changed. We can see that when sigsuspend returns, it restores the signal mask to its value before the call. □

Example

Another use of sigsuspend is to wait for a signal handler to set a global variable. In the program shown in Figure 10.23, we catch both the interrupt signal and the quit signal, but want to wake up the main routine only when the quit signal is caught.

```
#include "apue.h"

volatile sig_atomic_t   quitflag;   /* set nonzero by signal handler */

static void
sig_int(int signo)  /* one signal handler for SIGINT and SIGQUIT */
{
    if (signo == SIGINT)
        printf("\ninterrupt\n");
    else if (signo == SIGQUIT)
        quitflag = 1;   /* set flag for main loop */
}

int
main(void)
{
    sigset_t    newmask, oldmask, zeromask;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    if (signal(SIGQUIT, sig_int) == SIG_ERR)
```

```
        err_sys("signal(SIGQUIT) error");

    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);

    /*
     * Block SIGQUIT and save current signal mask.
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    while (quitflag == 0)
        sigsuspend(&zeromask);

    /*
     * SIGQUIT has been caught and is now blocked; do whatever.
     */
    quitflag = 0;

    /*
     * Reset signal mask which unblocks SIGQUIT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");

    exit(0);
}
```

Figure 10.23 Using `sigsuspend` to wait for a global variable to be set

Sample output from this program is

```
$ ./a.out
^C                  type the interrupt character
interrupt
^C                  type the interrupt character again
interrupt
^C                  and again
interrupt
^\ $                now terminate with the quit character
```

□

For portability between non-POSIX systems that support ISO C and POSIX.1 systems, the only thing we should do within a signal handler is assign a value to a variable of type `sig_atomic_t`—nothing else. POSIX.1 goes further and specifies a list of functions that are safe to call from within a signal handler (Figure 10.4), but if we do this, our code may not run correctly on non-POSIX systems.

Example

As another example of signals, we show how signals can be used to synchronize a parent and child. Figure 10.24 shows implementations of the five routines `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT`, and `WAIT_CHILD` from Section 8.9.

```
#include "apue.h"

static volatile sig_atomic_t sigflag; /* set nonzero by sig handler */
static sigset_t newmask, oldmask, zeromask;

static void
sig_usr(int signo)  /* one signal handler for SIGUSR1 and SIGUSR2 */
{
    sigflag = 1;
}

void
TELL_WAIT(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR1) error");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR2) error");
    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGUSR1);
    sigaddset(&newmask, SIGUSR2);

    /* Block SIGUSR1 and SIGUSR2, and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");
}

void
TELL_PARENT(pid_t pid)
{
    kill(pid, SIGUSR2);     /* tell parent we're done */
}

void
WAIT_PARENT(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask);  /* and wait for parent */
    sigflag = 0;

    /* Reset signal mask to original value */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}

void
TELL_CHILD(pid_t pid)
{
    kill(pid, SIGUSR1);         /* tell child we're done */
}

void
WAIT_CHILD(void)
```

```
{
    while (sigflag == 0)
        sigsuspend(&zeromask);  /* and wait for child */
    sigflag = 0;

    /* Reset signal mask to original value */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}
```

Figure 10.24 Routines to allow a parent and child to synchronize

We use the two user-defined signals: `SIGUSR1` is sent by the parent to the child, and `SIGUSR2` is sent by the child to the parent. In Figure 15.7, we show another implementation of these five functions using pipes. □

The `sigsuspend` function is fine if we want to go to sleep while we're waiting for a signal to occur (as we've shown in the previous two examples), but what if we want to call other system functions while we're waiting? Unfortunately, this problem has no bulletproof solution unless we use multiple threads and dedicate a separate thread to handling signals, as we discuss in Section 12.8.

Without using threads, the best we can do is to set a global variable in the signal handler when the signal occurs. For example, if we catch both `SIGINT` and `SIGALRM` and install the signal handlers using the `signal_intr` function, the signals will interrupt any slow system call that is blocked. The signals are most likely to occur when we're blocked in a call to the `read` function waiting for input from a slow device. (This is especially true for `SIGALRM`, since we set the alarm clock to prevent us from waiting forever for input.) The code to handle this looks similar to the following:

```
if (intr_flag)      /* flag set by our SIGINT handler */
    handle_intr();
if (alrm_flag)      /* flag set by our SIGALRM handler */
    handle_alrm();

/* signals occurring in here are lost */

while (read( ... ) < 0) {
    if (errno == EINTR) {
        if (alrm_flag)
            handle_alrm();
        else if (intr_flag)
            handle_intr();
    } else {
        /* some other error */
    }
} else if (n == 0) {
    /* end of file */
} else {
    /* process input */
}
```

We test each of the global flags before calling read and again if read returns an interrupted system call error. The problem occurs if either signal is caught between the first two if statements and the subsequent call to read. Signals occurring in here are lost, as indicated by the code comment. The signal handlers are called, and they set the appropriate global variable, but the read never returns (unless some data is ready to be read).

What we would like to be able to do is the following sequence of steps, in order.

1. Block SIGINT and SIGALRM.
2. Test the two global variables to see whether either signal has occurred and, if so, handle the condition.
3. Call read (or any other system function) and unblock the two signals, as an atomic operation.

The sigsuspend function helps us only if step 3 is a pause operation.

10.17 abort Function

We mentioned earlier that the abort function causes abnormal program termination.

```
#include <stdlib.h>

void abort(void);
```

This function never returns

This function sends the SIGABRT signal to the caller. (Processes should not ignore this signal.) ISO C states that calling abort will deliver an unsuccessful termination notification to the host environment by calling raise(SIGABRT).

ISO C requires that if the signal is caught and the signal handler returns, abort still doesn't return to its caller. If this signal is caught, the only way the signal handler can't return is if it calls exit, _exit, _Exit, longjmp, or siglongjmp. (Section 10.15 discusses the differences between longjmp and siglongjmp.) POSIX.1 also specifies that abort overrides the blocking or ignoring of the signal by the process.

The intent of letting the process catch the SIGABRT is to allow it to perform any cleanup that it wants to do before the process terminates. If the process doesn't terminate itself from this signal handler, POSIX.1 states that, when the signal handler returns, abort terminates the process.

The ISO C specification of this function leaves it up to the implementation as to whether output streams are flushed and whether temporary files (Section 5.13) are deleted. POSIX.1 goes further and allows an implementation to call fclose on open standard I/O streams before terminating if the call to abort terminates the process.

> Earlier versions of System V generated the SIGIOT signal from the abort function. Furthermore, it was possible for a process to ignore this signal or to catch it and return from the signal handler, in which case abort returned to its caller.

4.3BSD generated the SIGILL signal. Before doing this, the 4.3BSD function unblocked the signal and reset its disposition to SIG_DFL (terminate with core file). This prevented a process from either ignoring the signal or catching it.

Historically, implementations of abort have differed in how they deal with standard I/O streams. For defensive programming and improved portability, if we want standard I/O streams to be flushed, we specifically do it before calling abort. We do this in the err_dump function (Appendix B).

Since most UNIX System implementations of tmpfile call unlink immediately after creating the file, the ISO C warning about temporary files does not usually concern us.

Example

Figure 10.25 shows an implementation of the abort function as specified by POSIX.1.

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void
abort(void)         /* POSIX-style abort() function */
{
    sigset_t            mask;
    struct sigaction    action;

    /* Caller can't ignore SIGABRT, if so reset to default */
    sigaction(SIGABRT, NULL, &action);
    if (action.sa_handler == SIG_IGN) {
        action.sa_handler = SIG_DFL;
        sigaction(SIGABRT, &action, NULL);
    }
    if (action.sa_handler == SIG_DFL)
        fflush(NULL);           /* flush all open stdio streams */

    /* Caller can't block SIGABRT; make sure it's unblocked */
    sigfillset(&mask);
    sigdelset(&mask, SIGABRT);  /* mask has only SIGABRT turned off */
    sigprocmask(SIG_SETMASK, &mask, NULL);
    kill(getpid(), SIGABRT);    /* send the signal */

    /* If we're here, process caught SIGABRT and returned */
    fflush(NULL);               /* flush all open stdio streams */
    action.sa_handler = SIG_DFL;
    sigaction(SIGABRT, &action, NULL);  /* reset to default */
    sigprocmask(SIG_SETMASK, &mask, NULL);  /* just in case ... */
    kill(getpid(), SIGABRT);                /* and one more time */
    exit(1);    /* this should never be executed ... */
}
```

Figure 10.25 Implementation of POSIX.1 abort

We first see whether the default action will occur; if so, we flush all the standard I/O streams. This is not equivalent to calling `fclose` on all the open streams (since it just flushes them and doesn't close them), but when the process terminates, the system closes all open files. If the process catches the signal and returns, we flush all the streams again, since the process could have generated more output. The only condition we don't handle is the case where the process catches the signal and calls `_exit` or `_Exit`. In this case, any unflushed standard I/O buffers in memory are discarded. We assume that a caller that does this doesn't want the buffers flushed.

Recall from Section 10.9 that if calling `kill` causes the signal to be generated for the caller, and if the signal is not blocked (which we guarantee in Figure 10.25), then the signal (or some other pending, unlocked signal) is delivered to the process before `kill` returns. We block all signals except `SIGABRT`, so we know that if the call to `kill` returns, the process caught the signal and the signal handler returned. □

10.18 `system` Function

In Section 8.13, we showed an implementation of the `system` function. That version, however, did not do any signal handling. POSIX.1 requires that `system` ignore `SIGINT` and `SIGQUIT` and block `SIGCHLD`. Before showing a version that handles these signals correctly, let's see why we need to worry about signal handling.

Example

The program shown in Figure 10.26 uses the version of `system` from Section 8.13 to invoke the `ed`(1) editor. (This editor has been part of UNIX systems for a long time. We use it here because it is an interactive program that catches the interrupt and quit signals. If we invoke `ed` from a shell and type the interrupt character, it catches the interrupt signal and prints a question mark. The `ed` program also sets the disposition of the quit signal so that it is ignored.) The program in Figure 10.26 catches both `SIGINT` and `SIGCHLD`. If we invoke the program, we get

```
$ ./a.out
a                          append text to the editor's buffer
Here is one line of text
.                          period on a line by itself stops append mode
1,$p                       print first through last lines of buffer to see what's there
Here is one line of text
w temp.foo                 write the buffer to a file
25                         editor says it wrote 25 bytes
q                          and leave the editor
caught SIGCHLD
```

When the editor terminates, the system sends the `SIGCHLD` signal to the parent (the `a.out` process). We catch it and return from the signal handler. But if it is catching the `SIGCHLD` signal, the parent should be doing so because it has created its own children, so that it knows when its children have terminated. The delivery of this signal in the

```
#include "apue.h"

static void
sig_int(int signo)
{
    printf("caught SIGINT\n");
}

static void
sig_chld(int signo)
{
    printf("caught SIGCHLD\n");
}

int
main(void)
{
    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    if (signal(SIGCHLD, sig_chld) == SIG_ERR)
        err_sys("signal(SIGCHLD) error");
    if (system("/bin/ed") < 0)
        err_sys("system() error");
    exit(0);
}
```

Figure 10.26 Using `system` to invoke the `ed` editor

parent should be blocked while the `system` function is executing. Indeed, this is what POSIX.1 specifies. Otherwise, when the child created by `system` terminates, it would fool the caller of `system` into thinking that one of its own children terminated. The caller would then use one of the `wait` functions to get the termination status of the child, thereby preventing the `system` function from being able to obtain the child's termination status for its return value.

If we run the program again, this time sending the editor an interrupt signal, we get

```
$ ./a.out
a                   append text to the editor's buffer
hello, world
.                   period on a line by itself stops append mode
1,$p                print first through last lines to see what's there
hello, world
w temp.foo          write the buffer to a file
13                  editor says it wrote 13 bytes
^C                  type the interrupt character
?                   editor catches signal, prints question mark
caught SIGINT       and so does the parent process
q                   leave editor
caught SIGCHLD
```

Recall from Section 9.6 that typing the interrupt character causes the interrupt signal to be sent to all the processes in the foreground process group. Figure 10.27 shows the arrangement of the processes when the editor is running.

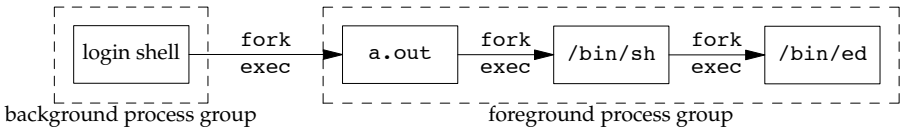


Figure 10.27 Foreground and background process groups for Figure 10.26

In this example, SIGINT is sent to all three foreground processes. (The shell ignores it.) As we can see from the output, both the a.out process and the editor catch the signal. But when we're running another program with the system function, we shouldn't have both the parent and the child catching the two terminal-generated signals: interrupt and quit. Instead, these two signals should be sent to the program that is running: the child. Since the command that is executed by system can be an interactive command (as is the ed program in this example) and since the caller of system gives up control while the program executes, waiting for it to finish, the caller of system should not be receiving these two terminal-generated signals. For this reason, POSIX.1 specifies that the system function should ignore these two signals while waiting for the command to complete. □

Example

Figure 10.28 shows an implementation of the system function with the required signal handling.

```
#include    <sys/wait.h>
#include    <errno.h>
#include    <signal.h>
#include    <unistd.h>

int
system(const char *cmdstring)   /* with appropriate signal handling */
{
    pid_t               pid;
    int                 status;
    struct sigaction    ignore, saveintr, savequit;
    sigset_t            chldmask, savemask;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with UNIX */

    ignore.sa_handler = SIG_IGN;    /* ignore SIGINT and SIGQUIT */
    sigemptyset(&ignore.sa_mask);
    ignore.sa_flags = 0;
```

```
    if (sigaction(SIGINT, &ignore, &saveintr) < 0)
        return(-1);
    if (sigaction(SIGQUIT, &ignore, &savequit) < 0)
        return(-1);
    sigemptyset(&chldmask);          /* now block SIGCHLD */
    sigaddset(&chldmask, SIGCHLD);
    if (sigprocmask(SIG_BLOCK, &chldmask, &savemask) < 0)
        return(-1);

    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {           /* child */
        /* restore previous signal actions & reset signal mask */
        sigaction(SIGINT, &saveintr, NULL);
        sigaction(SIGQUIT, &savequit, NULL);
        sigprocmask(SIG_SETMASK, &savemask, NULL);

        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* exec error */
    } else {                         /* parent */
        while (waitpid(pid, &status, 0) < 0)
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
    }

    /* restore previous signal actions & reset signal mask */
    if (sigaction(SIGINT, &saveintr, NULL) < 0)
        return(-1);
    if (sigaction(SIGQUIT, &savequit, NULL) < 0)
        return(-1);
    if (sigprocmask(SIG_SETMASK, &savemask, NULL) < 0)
        return(-1);

    return(status);
}
```

Figure 10.28 Correct POSIX.1 implementation of `system` function

If we link the program in Figure 10.26 with this implementation of the `system` function, the resulting binary differs from the last (flawed) one in the following ways.

1. No signal is sent to the calling process when we type the interrupt or quit character.
2. When the `ed` command exits, `SIGCHLD` is not sent to the calling process. Instead, it is blocked until we unblock it in the last call to `sigprocmask`, after the `system` function retrieves the child's termination status by calling `waitpid`.

POSIX.1 states that if wait or waitpid returns the status of a child process while SIGCHLD is pending, then SIGCHLD should not be delivered to the process unless the status of another child process is also available. FreeBSD 8.0, Mac OS X 10.6.8, and Solaris 10 all implement this semantic. Linux 3.2.0, however, doesn't—SIGCHLD remains pending after the system function calls waitpid; when the signal is unblocked, it is delivered to the caller. If we called wait in the sig_chld function in Figure 10.26, a Linux system would return –1 with errno set to ECHILD, since the system function already retrieved the termination status of the child.

Many older texts show the ignoring of the interrupt and quit signals as follows:

```
if ((pid = fork()) < 0) {
    err_sys("fork error");
} else if (pid == 0) {
    /* child */
    execl(...);
    _exit(127);
}

/* parent */
old_intr = signal(SIGINT, SIG_IGN);
old_quit = signal(SIGQUIT, SIG_IGN);
waitpid(pid, &status, 0)
signal(SIGINT, old_intr);
signal(SIGQUIT, old_quit);
```

The problem with this sequence of code is that we have no guarantee after the fork regarding whether the parent or child runs first. If the child runs first and the parent doesn't run for some time after, an interrupt signal might be generated before the parent is able to change its disposition to be ignored. For this reason, in Figure 10.28, we change the disposition of the signals before the fork.

Note that we have to reset the dispositions of these two signals in the child before the call to execl. This allows execl to change their dispositions to the default, based on the caller's dispositions, as we described in Section 8.10. □

Return Value from system

The return value from system is the termination status of the shell, which isn't always the termination status of the command string. We saw some examples in Figure 8.23, and the results were as we expected: if we execute a simple command, such as date, the termination status is 0. Executing the shell command exit 44 gave us a termination status of 44. What happens with signals?

Let's run the program in Figure 8.24 and send some signals to the command that's executing:

```
$ tsys "sleep 30"
^Cnormal termination, exit status = 130   we press the interrupt key
$ tsys "sleep 30"
^\sh: 946 Quit                            we press the quit key
normal termination, exit status = 131
```

When we terminate the `sleep` call with the interrupt signal, the `pr_exit` function (Figure 8.5) thinks that it terminated normally. The same thing happens when we kill the `sleep` call with the quit key. As this example demonstrates, the Bourne shell has a poorly documented feature in which its termination status is 128 plus the signal number, when the command it was executing is terminated by a signal. We can see this with the shell interactively.

```
$ sh                                    make sure we're running the Bourne shell
$ sh -c "sleep 30"
^C                                      press the interrupt key
$ echo $?                               print termination status of last command
130
$ sh -c "sleep 30"
^\sh: 962 Quit - core dumped            press the quit key
$ echo $?                               print termination status of last command
131
$ exit                                  leave Bourne shell
```

On the system being used, `SIGINT` has a value of 2 and `SIGQUIT` has a value of 3, giving us the shell's termination statuses of 130 and 131.

Let's try a similar example, but this time we'll send a signal directly to the shell and see what is returned by `system`:

```
$ tsys "sleep 30" &                     start it in background this time
9257
$ ps -f                                 look at the process IDs
     UID   PID  PPID  TTY      TIME CMD
     sar  9260   949  pts/5    0:00 ps -f
     sar  9258  9257  pts/5    0:00 sh -c sleep 30
     sar   949   947  pts/5    0:01 /bin/sh
     sar  9257   949  pts/5    0:00 tsys sleep 30
     sar  9259  9258  pts/5    0:00 sleep 30
$ kill -KILL 9258                       kill the shell itself
abnormal termination, signal number = 9
```

Here, we can see that the return value from `system` reports an abnormal termination only when the shell itself terminates abnormally.

> Other shells behave differently when handling terminal-generated signals, such as `SIGINT` and `SIGQUIT`. With `bash` and `dash`, for example, pressing the interrupt or quit key will result in an exit status indicating abnormal termination with the corresponding signal number. However, if we find our process executing `sleep` and send it a signal directly, so that the signal goes only to the individual process instead of the entire foreground process group, we will find that these shells behave like the Bourne shell and exit with a normal termination status of 128 plus the signal number.

When writing programs that use the `system` function, be sure to interpret the return value correctly. If you call `fork`, `exec`, and `wait` yourself, the termination status is not the same as if you call `system`.

10.19 sleep, nanosleep, and clock_nanosleep Functions

We've used the sleep function in numerous examples throughout the text, and we showed two flawed implementations of it in Figures 10.7 and 10.8.

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Returns: 0 or number of unslept seconds

This function causes the calling process to be suspended until either

1. The amount of wall clock time specified by *seconds* has elapsed.
2. A signal is caught by the process and the signal handler returns.

As with an alarm signal, the actual return may occur at a time later than requested because of other system activity.

In case 1, the return value is 0. When sleep returns early because of some signal being caught (case 2), the return value is the number of unslept seconds (the requested time minus the actual time slept).

Although sleep can be implemented with the alarm function (Section 10.10), this isn't required. If alarm is used, however, there can be interactions between the two functions. The POSIX.1 standard leaves all these interactions unspecified. For example, if we do an alarm(10) and 3 wall clock seconds later do a sleep(5), what happens? The sleep will return in 5 seconds (assuming that some other signal isn't caught in the interim), but will another SIGALRM be generated 2 seconds later? These details depend on the implementation.

> FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10 implement sleep using the nanosleep function, which allows the implementation to be independent of signals and the alarm timer. For portability, you shouldn't make any assumptions about the implementation of sleep, but if you have any intentions of mixing calls to sleep with any other timing functions, you need to be aware of possible interactions.

Example

Figure 10.29 shows an implementation of the POSIX.1 sleep function. This function is a modification of Figure 10.7, which handles signals reliably, avoiding the race condition in the earlier implementation. We still do not handle any interactions with previously set alarms. (As we mentioned, these interactions are explicitly undefined by POSIX.1.)

```
#include "apue.h"

static void
sig_alrm(int signo)
{
    /* nothing to do, just returning wakes up sigsuspend() */
}

unsigned int
```

```
sleep(unsigned int seconds)
{
    struct sigaction    newact, oldact;
    sigset_t            newmask, oldmask, suspmask;
    unsigned int        unslept;

    /* set our handler, save previous information */
    newact.sa_handler = sig_alrm;
    sigemptyset(&newact.sa_mask);
    newact.sa_flags = 0;
    sigaction(SIGALRM, &newact, &oldact);

    /* block SIGALRM and save current signal mask */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGALRM);
    sigprocmask(SIG_BLOCK, &newmask, &oldmask);

    alarm(seconds);
    suspmask = oldmask;

    /* make sure SIGALRM isn't blocked */
    sigdelset(&suspmask, SIGALRM);

    /* wait for any signal to be caught */
    sigsuspend(&suspmask);

    /* some signal has been caught, SIGALRM is now blocked */

    unslept = alarm(0);

    /* reset previous action */
    sigaction(SIGALRM, &oldact, NULL);

    /* reset signal mask, which unblocks SIGALRM */
    sigprocmask(SIG_SETMASK, &oldmask, NULL);
    return(unslept);
}
```

Figure 10.29 Reliable implementation of `sleep`

It takes more code to write this reliable implementation than what is shown in Figure 10.7. We don't use any form of nonlocal branching (as we did in Figure 10.8 to avoid the race condition between `alarm` and `pause`), so there is no effect on other signal handlers that may be executing when the `SIGALRM` is handled. □

The `nanosleep` function is similar to the `sleep` function, but provides nanosecond-level granularity.

```
#include <time.h>

int nanosleep(const struct timespec *reqtp, struct timespec *remtp);
```

Returns: 0 if slept for requested time or –1 on error

This function suspends the calling process until either the requested time has elapsed or the function is interrupted by a signal. The *reqtp* parameter specifies the amount of time

to sleep in seconds and nanoseconds. If the sleep interval is interrupted by a signal and the process doesn't terminate, the `timespec` structure pointed to by the *remtp* parameter will be set to the amount of time left in the sleep interval. We can set this parameter to `NULL` if we are uninterested in the time unslept.

If the system doesn't support nanosecond granularity, the requested time is rounded up. Because the `nanosleep` function doesn't involve the generation of any signals, we can use it without worrying about interactions with other functions.

> The `nanosleep` function used to belong to the Timers option in the Single UNIX Specification, but was moved to the base in SUSv4.

With the introduction of multiple system clocks (recall Section 6.10), we need a way to suspend the calling thread using a delay time relative to a particular clock. The `clock_nanosleep` function provides us with this capability.

```
#include <time.h>

int clock_nanosleep(clockid_t clock_id, int flags,
                    const struct timespec *reqtp, struct timespec *remtp);
```

Returns: 0 if slept for requested time or error number on failure

The *clock_id* argument specifies the clock against which the time delay is evaluated. Identifiers for clocks are listed in Figure 6.8. The *flags* argument is used to control whether the delay is absolute or relative. When *flags* is set to 0, the sleep time is relative (i.e., how long we want to sleep). When it is set to `TIMER_ABSTIME`, the sleep time is absolute (i.e., we want to sleep until the clock reaches the specified time).

The other arguments, *reqtp* and *remtp*, are the same as in the `nanosleep` function. However, when we use an absolute time, the *remtp* argument is unused, because it isn't needed; we can reuse the same value for the *reqtp* argument for additional calls to `clock_nanosleep` until the clock reaches the specified absolute time value.

Note that except for error returns, the call

```
clock_nanosleep(CLOCK_REALTIME, 0, reqtp, remtp);
```

has the same effect as the call

```
nanosleep(reqtp, remtp);
```

The problem with using a relative sleep is that some applications require precision with how long they sleep, and a relative sleep time can lead to sleeping longer than desired. For example, if an application wants to perform a task at regular intervals, it would have to get the current time, calculate the amount of time until the next time to execute the task, and then call `nanosleep`. Between the time that the current time is obtained and the call to `nanosleep` is made, processor scheduling and preemption can result in the relative sleep time extending past the desired interval. Using an absolute time improves the precision, even though a time-sharing process scheduler makes no guarantee that our task will execute immediately after our sleep time has ended.

> In older versions of the Single UNIX Specification, the `clock_nanosleep` function belonged to the Clock Selection option. In SUSv4, it was moved to the base.

10.20 sigqueue Function

In Section 10.8 we said that most UNIX systems don't queue signals. With the real-time extensions to POSIX.1, some systems began adding support for queueing signals. With SUSv4, the queued signal functionality has moved from the real-time extensions to the base specification.

Generally a signal carries one bit of information: the signal itself. In addition to queueing signals, these extensions allow applications to pass more information along with the delivery (recall Section 10.14). This information is embedded in a `siginfo` structure. Along with system-provided information, applications can pass an integer or a pointer to a buffer containing more information to the signal handler.

To use queued signals we have to do the following:

1. Specify the `SA_SIGINFO` flag when we install a signal handler using the `sigaction` function. If we don't specify this flag, the signal will be posted, but it is left up to the implementation whether the signal is queued.
2. Provide a signal handler in the `sa_sigaction` member of the `sigaction` structure instead of using the usual `sa_handler` field. Implementations might allow us to use the `sa_handler` field, but we won't be able to obtain the extra information sent with the `sigqueue` function.
3. Use the `sigqueue` function to send signals.

```
#include <signal.h>

int sigqueue(pid_t pid, int signo, const union sigval value)
```

Returns: 0 if OK, –1 on error

The `sigqueue` function is similar to the `kill` function, except that we can only direct signals to a single process with `sigqueue`, and we can use the *value* argument to transmit either an integer or a pointer value to the signal handler.

Signals can't be queued infinitely. Recall the `SIGQUEUE_MAX` limit from Figure 2.9 and Figure 2.11. When this limit is reached, `sigqueue` can fail with `errno` set to `EAGAIN`.

With the real-time signal enhancements, a separate set of signals was introduced for application use. These are the signal numbers between `SIGRTMIN` and `SIGRTMAX`, inclusive. Be aware that the default action for these signals is to terminate the process.

Figure 10.30 summarizes the way queued signals differ in behavior among the implementations covered in this text.

Behavior	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
supports `sigqueue`	•	•	•		•
queues other signals besides `SIGRTMIN` to `SIGRTMAX`	optional	•			•
queues signals even if the caller doesn't use the `SA_SIGINFO` flag	optional	•	•		

Figure 10.30 Behavior of queued signals on various platforms

Mac OS X 10.6.8 doesn't support `sigqueue` or real-time signals. On Solaris 10, `sigqueue` is in the real-time library, `librt`.

10.21 Job-Control Signals

Of the signals shown in Figure 10.1, POSIX.1 considers six to be job-control signals:

`SIGCHLD`	Child process has stopped or terminated.
`SIGCONT`	Continue process, if stopped.
`SIGSTOP`	Stop signal (can't be caught or ignored).
`SIGTSTP`	Interactive stop signal.
`SIGTTIN`	Read from controlling terminal by background process group member.
`SIGTTOU`	Write to controlling terminal by a background process group member.

Except for `SIGCHLD`, most application programs don't handle these signals: interactive shells usually do all the work required to handle them. When we type the suspend character (usually Control-Z), `SIGTSTP` is sent to all processes in the foreground process group. When we tell the shell to resume a job in the foreground or background, the shell sends all the processes in the job the `SIGCONT` signal. Similarly, if `SIGTTIN` or `SIGTTOU` is delivered to a process, the process is stopped by default, and the job-control shell recognizes this and notifies us.

An exception is a process that is managing the terminal—the `vi`(1) editor, for example. It needs to know when the user wants to suspend it so that it can restore the terminal's state to the way it was when `vi` was started. Also, when it resumes in the foreground, the `vi` editor needs to set the terminal state back to the way it wants it, and it needs to redraw the terminal screen. We see how a program such as `vi` handles this in the example that follows.

There are some interactions between the job-control signals. When any of the four stop signals (`SIGTSTP`, `SIGSTOP`, `SIGTTIN`, or `SIGTTOU`) is generated for a process, any pending `SIGCONT` signal for that process is discarded. Similarly, when the `SIGCONT` signal is generated for a process, any pending stop signals for that same process are discarded.

Note that the default action for `SIGCONT` is to continue the process, if it is stopped; otherwise, the signal is ignored. Normally, we don't have to do anything with this signal. When `SIGCONT` is generated for a process that is stopped, the process is continued, even if the signal is blocked or ignored.

Example

The program in Figure 10.31 demonstrates the normal sequence of code used when a program handles job control. This program simply copies its standard input to its standard output, but comments are given in the signal handler for typical actions performed by a program that manages a screen.

```
#include "apue.h"

#define BUFFSIZE    1024

static void
sig_tstp(int signo) /* signal handler for SIGTSTP */
{
    sigset_t    mask;

    /* ... move cursor to lower left corner, reset tty mode ... */

    /*
     * Unblock SIGTSTP, since it's blocked while we're handling it.
     */
    sigemptyset(&mask);
    sigaddset(&mask, SIGTSTP);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);

    signal(SIGTSTP, SIG_DFL);   /* reset disposition to default */

    kill(getpid(), SIGTSTP);    /* and send the signal to ourself */

    /* we won't return from the kill until we're continued */

    signal(SIGTSTP, sig_tstp);  /* reestablish signal handler */

    /* ... reset tty mode, redraw screen ... */
}

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    /*
     * Only catch SIGTSTP if we're running with a job-control shell.
     */
    if (signal(SIGTSTP, SIG_IGN) == SIG_DFL)
        signal(SIGTSTP, sig_tstp);

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}
```

Figure 10.31 How to handle SIGTSTP

When the program in Figure 10.31 starts, it arranges to catch the SIGTSTP signal only if the signal's disposition is SIG_DFL. The reason is that when the program is started by a shell that doesn't support job control (/bin/sh, for example), the signal's disposition

should be set to SIG_IGN. In fact, the shell doesn't explicitly ignore this signal; init sets the disposition of the three job-control signals (SIGTSTP, SIGTTIN, and SIGTTOU) to SIG_IGN. This disposition is then inherited by all login shells. Only a job-control shell should reset the disposition of these three signals to SIG_DFL.

When we type the suspend character, the process receives the SIGTSTP signal and the signal handler is invoked. At this point, we would do any terminal-related processing: move the cursor to the lower-left corner, restore the terminal mode, and so on. We then send ourself the same signal, SIGTSTP, after resetting its disposition to its default (stop the process) and unblocking the signal. We have to unblock it since we're currently handling that same signal, and the system blocks it automatically while it's being caught. At this point, the system stops the process. It is continued only when it receives (usually from the job-control shell, in response to an interactive fg command) a SIGCONT signal. We don't catch SIGCONT. Its default disposition is to continue the stopped process; when this happens, the program continues as though it returned from the kill function. When the program is continued, we reset the disposition for the SIGTSTP signal and do whatever terminal processing we want (we could redraw the screen, for example). □

10.22 Signal Names and Numbers

In this section, we describe how to map between signal numbers and names. Some systems provide the array

```
extern char *sys_siglist[];
```

The array index is the signal number, giving a pointer to the character string name of the signal.

> FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8 all provide this array of signal names. Solaris 10 does, too, but it uses the name _sys_siglist instead.

To print the character string corresponding to a signal number in a portable manner, we can use the psignal function.

```
#include <signal.h>

void psignal(int signo, const char *msg);
```

The string *msg* (which normally includes the name of the program) is output to the standard error, followed by a colon and a space, followed by a description of the signal, followed by a newline. If *msg* is NULL, then only the description is written to the standard error. This function is similar to perror (Section 1.7).

If you have a siginfo structure from an alternative sigaction signal handler, you can print the signal information with the psiginfo function.

```
#include <signal.h>

void psiginfo(const siginfo_t *info, const char *msg);
```

It operates in a similar manner to the `psignal` function. Although this function has access to more information than just the signal number, platforms vary in exactly what additional information is printed.

If you only need the string description of the signal and don't necessarily want to write it to standard error (you might want to write it to a log file, for example), you can use the `strsignal` function. This function is similar to `strerror` (also described in Section 1.7).

```
#include <string.h>

char *strsignal(int signo);
```

Returns: a pointer to a string describing the signal

Given a signal number, `strsignal` will return a string that describes the signal. This string can be used by applications to print error messages about signals received.

> All the platforms discussed in this book provide the `psignal` and `strsignal` functions, but differences do occur. On Solaris 10, `strsignal` will return a null pointer if the signal number is invalid, whereas FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8 return a string indicating that the signal number is unrecognized.
>
> Only Linux 3.2.0 and Solaris 10 support the `psiginfo` function.

Solaris provides a couple of functions to map a signal number to a signal name, and vice versa.

```
#include <signal.h>

int sig2str(int signo, char *str);

int str2sig(const char *str, int *signop);
```

Both return: 0 if OK, –1 on error

These functions are useful when writing interactive programs that need to accept and print signal names and numbers.

The `sig2str` function translates the given signal number into a string and stores the result in the memory pointed to by *str*. The caller must ensure that the memory is large enough to hold the longest string, including the terminating null byte. Solaris provides the constant `SIG2STR_MAX` in `<signal.h>` to define the maximum string length. The string consists of the signal name without the "SIG" prefix. For example, translating `SIGKILL` would result in the string "KILL" being stored in the *str* memory buffer.

The `str2sig` function translates the given name into a signal number. The signal number is stored in the integer pointed to by *signop*. The name can be either the signal name without the "SIG" prefix or a string representation of the decimal signal number (i.e., "9").

Note that `sig2str` and `str2sig` depart from common practice and don't set `errno` when they fail.

10.23 Summary

Signals are used in most nontrivial applications. An understanding of the hows and whys of signal handling is essential to advanced UNIX System programming. This chapter has taken a long and thorough look at UNIX System signals. We started by looking at the warts in previous implementations of signals and how they manifest themselves. We then proceeded to the POSIX.1 reliable-signal concept and all the related functions. Once we covered all these details, we were able to provide implementations of the POSIX.1 `abort`, `system`, and `sleep` functions. We finished with a look at the job-control signals and the ways that we can convert between signal names and signal numbers.

Exercises

10.1 In Figure 10.2, remove the `for (;;)` statement. What happens and why?

10.2 Implement the `sig2str` function described in Section 10.22.

10.3 Draw pictures of the stack frames when we run the program from Figure 10.9.

10.4 In Figure 10.11, we showed a technique that's often used to set a timeout on an I/O operation using `setjmp` and `longjmp`. The following code has also been seen:

```
signal(SIGALRM, sig_alrm);
alarm(60);
if (setjmp(env_alrm) != 0) {
    /* handle timeout */
    ...
}
...
```

What else is wrong with this sequence of code?

10.5 Using only a single timer (either `alarm` or the higher-precision `setitimer`), provide a set of functions that allows a process to set any number of timers.

10.6 Write the following program to test the parent–child synchronization functions in Figure 10.24. The process creates a file and writes the integer 0 to the file. The process then calls `fork`, and the parent and child alternate incrementing the counter in the file. Each time the counter is incremented, print which process (parent or child) is doing the increment.

10.7 In the function shown in Figure 10.25, if the caller catches `SIGABRT` and returns from the signal handler, why do we go to the trouble of resetting the disposition to its default and call `kill` the second time, instead of simply calling `_exit`?

10.8 Why do you think the `siginfo` structure (Section 10.14) includes the real user ID, instead of the effective user ID, in the `si_uid` field?

10.9 Rewrite the function in Figure 10.14 to handle all the signals from Figure 10.1. The function should consist of a single loop that iterates once for every signal in the current signal mask (not once for every possible signal).

10.10 Write a program that calls `sleep(60)` in an infinite loop. Every five times through the loop (every 5 minutes), fetch the current time of day and print the `tm_sec` field. Run the program overnight and explain the results. How would a program such as the `cron` daemon, which runs every minute on the minute, handle this situation?

10.11 Modify Figure 3.5 as follows: (a) change `BUFFSIZE` to 100; (b) catch the `SIGXFSZ` signal using the `signal_intr` function, printing a message when it's caught, and returning from the signal handler; and (c) print the return value from `write` if the requested number of bytes wasn't written. Modify the soft `RLIMIT_FSIZE` resource limit (Section 7.11) to 1,024 bytes and run your new program, copying a file that is larger than 1,024 bytes. (Try to set the soft resource limit from your shell. If you can't do this from your shell, call `setrlimit` directly from the program.) Run this program on the different systems that you have access to. What happens and why?

10.12 Write a program that calls `fwrite` with a large buffer (about one gigabyte). Before calling `fwrite`, call `alarm` to schedule a signal in 1 second. In your signal handler, print that the signal was caught and return. Does the call to `fwrite` complete? What's happening?

11

Threads

11.1 Introduction

We discussed processes in earlier chapters. We learned about the environment of a UNIX process, the relationships between processes, and ways to control processes. We saw that a limited amount of sharing can occur between related processes.

In this chapter, we'll look inside a process further to see how we can use multiple *threads of control* (or simply *threads*) to perform multiple tasks within the environment of a single process. All threads within a single process have access to the same process components, such as file descriptors and memory.

Anytime you try to share a single resource among multiple users, you have to deal with consistency. We'll conclude this chapter with a look at the synchronization mechanisms available to prevent multiple threads from viewing inconsistencies in their shared resources.

11.2 Thread Concepts

A typical UNIX process can be thought of as having a single thread of control: each process is doing only one thing at a time. With multiple threads of control, we can design our programs to do more than one thing at a time within a single process, with each thread handling a separate task. This approach can have several benefits.

- We can simplify code that deals with asynchronous events by assigning a separate thread to handle each event type. Each thread can then handle its event using a synchronous programming model. A synchronous programming model is much simpler than an asynchronous one.
- Multiple processes have to use complex mechanisms provided by the operating system to share memory and file descriptors, as we will see in Chapters 15

and 17. Threads, in contrast, automatically have access to the same memory address space and file descriptors.

- Some problems can be partitioned so that overall program throughput can be improved. A single-threaded process with multiple tasks to perform implicitly serializes those tasks, because there is only one thread of control. With multiple threads of control, the processing of independent tasks can be interleaved by assigning a separate thread per task. Two tasks can be interleaved only if they don't depend on the processing performed by each other.
- Similarly, interactive programs can realize improved response time by using multiple threads to separate the portions of the program that deal with user input and output from the other parts of the program.

Some people associate multithreaded programming with multiprocessor or multicore systems. The benefits of a multithreaded programming model can be realized even if your program is running on a uniprocessor. A program can be simplified using threads regardless of the number of processors, because the number of processors doesn't affect the program structure. Furthermore, as long as your program has to block when serializing tasks, you can still see improvements in response time and throughput when running on a uniprocessor, because some threads might be able to run while others are blocked.

A thread consists of the information necessary to represent an execution context within a process. This includes a *thread ID* that identifies the thread within a process, a set of register values, a stack, a scheduling priority and policy, a signal mask, an `errno` variable (recall Section 1.7), and thread-specific data (Section 12.6). Everything within a process is sharable among the threads in a process, including the text of the executable program, the program's global and heap memory, the stacks, and the file descriptors.

The threads interfaces we're about to see are from POSIX.1-2001. The threads interfaces, also known as "pthreads" for "POSIX threads," originally were optional in POSIX.1-2001, but SUSv4 moved them to the base. The feature test macro for POSIX threads is `_POSIX_THREADS`. Applications can either use this in an `#ifdef` test to determine at compile time whether threads are supported or call `sysconf` with the `_SC_THREADS` constant to determine this at runtime. Systems conforming to SUSv4 define the symbol `_POSIX_THREADS` to have the value 200809L.

11.3 Thread Identification

Just as every process has a process ID, every thread has a thread ID. Unlike the process ID, which is unique in the system, the thread ID has significance only within the context of the process to which it belongs.

Recall that a process ID, represented by the `pid_t` data type, is a non-negative integer. A thread ID is represented by the `pthread_t` data type. Implementations are allowed to use a structure to represent the `pthread_t` data type, so portable implementations can't treat them as integers. Therefore, a function must be used to compare two thread IDs.

```
#include <pthread.h>

int pthread_equal(pthread_t tid1, pthread_t tid2);
```

Returns: nonzero if equal, 0 otherwise

Linux 3.2.0 uses an unsigned long integer for the `pthread_t` data type. Solaris 10 represents the `pthread_t` data type as an unsigned integer. FreeBSD 8.0 and Mac OS X 10.6.8 use a pointer to the `pthread` structure for the `pthread_t` data type.

A consequence of allowing the `pthread_t` data type to be a structure is that there is no portable way to print its value. Sometimes, it is useful to print thread IDs during program debugging, but there is usually no need to do so otherwise. At worst, this results in nonportable debug code, so it is not much of a limitation.

A thread can obtain its own thread ID by calling the `pthread_self` function.

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Returns: the thread ID of the calling thread

This function can be used with `pthread_equal` when a thread needs to identify data structures that are tagged with its thread ID. For example, a master thread might place work assignments on a queue and use the thread ID to control which jobs go to each worker thread. This situation is illustrated in Figure 11.1. A single master thread places new jobs on a work queue. A pool of three worker threads removes jobs from the queue. Instead of allowing each thread to process whichever job is at the head of the queue, the master thread controls job assignment by placing the ID of the thread that should process the job in each job structure. Each worker thread then removes only jobs that are tagged with its own thread ID.

11.4 Thread Creation

The traditional UNIX process model supports only one thread of control per process. Conceptually, this is the same as a threads-based model whereby each process is made up of only one thread. With pthreads, when a program runs, it also starts out as a single process with a single thread of control. As the program runs, its behavior should be indistinguishable from the traditional process, until it creates more threads of control. Additional threads can be created by calling the `pthread_create` function.

```
#include <pthread.h>

int pthread_create(pthread_t *restrict tidp,
                   const pthread_attr_t *restrict attr,
                   void *(*start_rtn)(void *), void *restrict arg);
```

Returns: 0 if OK, error number on failure

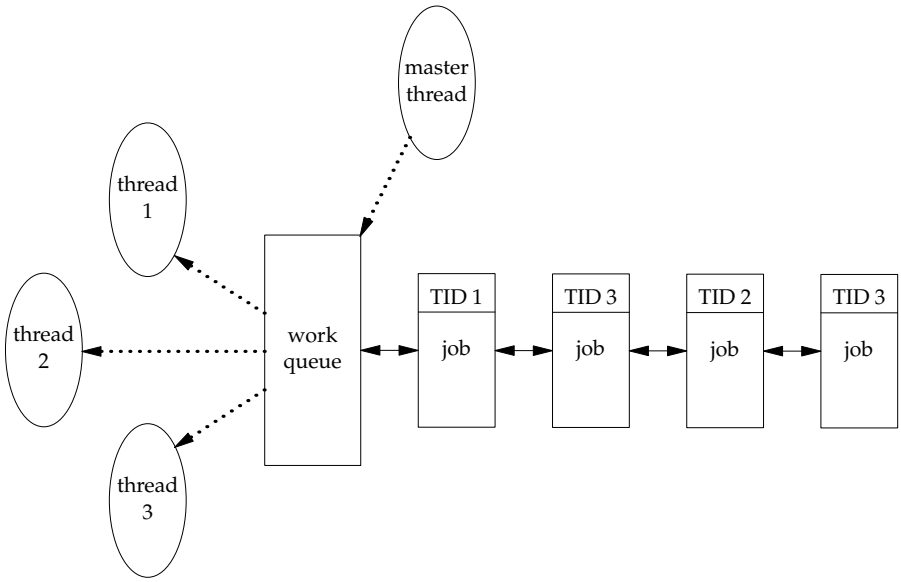


Figure 11.1 Work queue example

The memory location pointed to by *tidp* is set to the thread ID of the newly created thread when `pthread_create` returns successfully. The *attr* argument is used to customize various thread attributes. We'll cover thread attributes in Section 12.3, but for now, we'll set this to `NULL` to create a thread with the default attributes.

The newly created thread starts running at the address of the *start_rtn* function. This function takes a single argument, *arg*, which is a typeless pointer. If you need to pass more than one argument to the *start_rtn* function, then you need to store them in a structure and pass the address of the structure in *arg*.

When a thread is created, there is no guarantee which will run first: the newly created thread or the calling thread. The newly created thread has access to the process address space and inherits the calling thread's floating-point environment and signal mask; however, the set of pending signals for the thread is cleared.

Note that the pthread functions usually return an error code when they fail. They don't set `errno` like the other POSIX functions. The per-thread copy of `errno` is provided only for compatibility with existing functions that use it. With threads, it is cleaner to return the error code from the function, thereby restricting the scope of the error to the function that caused it, instead of relying on some global state that is changed as a side effect of the function.

Example

Although there is no portable way to print the thread ID, we can write a small test program that does, to gain some insight into how threads work. The program in

Figure 11.2 creates one thread and prints the process and thread IDs of the new thread and the initial thread.

```
#include "apue.h"
#include <pthread.h>

pthread_t ntid;

void
printids(const char *s)
{
    pid_t       pid;
    pthread_t   tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %lu tid %lu (0x%lx)\n", s, (unsigned long)pid,
      (unsigned long)tid, (unsigned long)tid);
}

void *
thr_fn(void *arg)
{
    printids("new thread: ");
    return((void *)0);
}

int
main(void)
{
    int     err;

    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0)
        err_exit(err, "can't create thread");
    printids("main thread:");
    sleep(1);
    exit(0);
}
```

Figure 11.2 Printing thread IDs

This example has two oddities, which are necessary to handle races between the main thread and the new thread. (We'll learn better ways to deal with these conditions later in this chapter.) The first is the need to sleep in the main thread. If it doesn't sleep, the main thread might exit, thereby terminating the entire process before the new thread gets a chance to run. This behavior is dependent on the operating system's threads implementation and scheduling algorithms.

The second oddity is that the new thread obtains its thread ID by calling `pthread_self` instead of reading it out of shared memory or receiving it as an argument to its thread-start routine. Recall that `pthread_create` will return the

thread ID of the newly created thread through the first parameter (*tidp*). In our example, the main thread stores this ID in `ntid`, but the new thread can't safely use it. If the new thread runs before the main thread returns from calling `pthread_create`, then the new thread will see the uninitialized contents of `ntid` instead of the thread ID.

Running the program in Figure 11.2 on Solaris gives us

```
$ ./a.out
main thread: pid 20075 tid 1 (0x1)
new thread:  pid 20075 tid 2 (0x2)
```

As we expect, both threads have the same process ID, but different thread IDs. Running the program in Figure 11.2 on FreeBSD gives us

```
$ ./a.out
main thread: pid 37396 tid 673190208 (0x28201140)
new thread:  pid 37396 tid 673280320 (0x28217140)
```

As we expect, both threads have the same process ID. If we look at the thread IDs as decimal integers, the values look strange, but if we look at them in hexadecimal format, they make more sense. As we noted earlier, FreeBSD uses a pointer to the thread data structure for its thread ID.

We would expect Mac OS X to be similar to FreeBSD; however, the thread ID for the main thread is from a different address range than the thread IDs for threads created with `pthread_create`:

```
$ ./a.out
main thread: pid 31807 tid 140735073889440 (0x7fff70162ca0)
new thread:  pid 31807 tid 4295716864 (0x1000b7000)
```

Running the same program on Linux gives us

```
$ ./a.out
main thread: pid 17874 tid 140693894424320 (0x7ff5d9996700)
new thread:  pid 17874 tid 140693886129920 (0x7ff5d91ad700)
```

The Linux thread IDs look like pointers, even though they are represented as unsigned long integers.

> The threads implementation changed between Linux 2.4 and Linux 2.6. In Linux 2.4, LinuxThreads implemented each thread with a separate process. This made it difficult to match the behavior of POSIX threads. In Linux 2.6, the Linux kernel and threads library were overhauled to use a new threads implementation called the Native POSIX Thread Library (NPTL). This supported a model of multiple threads within a single process and made it easier to support POSIX threads semantics.
>
> □

11.5 Thread Termination

If any thread within a process calls `exit`, `_Exit`, or `_exit`, then the entire process terminates. Similarly, when the default action is to terminate the process, a signal sent to a thread will terminate the entire process (we'll talk more about the interactions between signals and threads in Section 12.8).

A single thread can exit in three ways, thereby stopping its flow of control, without terminating the entire process.

1. The thread can simply return from the start routine. The return value is the thread's exit code.
2. The thread can be canceled by another thread in the same process.
3. The thread can call `pthread_exit`.

```
#include <pthread.h>

void pthread_exit(void *rval_ptr);
```

The *rval_ptr* argument is a typeless pointer, similar to the single argument passed to the start routine. This pointer is available to other threads in the process by calling the `pthread_join` function.

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **rval_ptr);
                                        Returns: 0 if OK, error number on failure
```

The calling thread will block until the specified thread calls `pthread_exit`, returns from its start routine, or is canceled. If the thread simply returned from its start routine, *rval_ptr* will contain the return code. If the thread was canceled, the memory location specified by *rval_ptr* is set to `PTHREAD_CANCELED`.

By calling `pthread_join`, we automatically place the thread with which we're joining in the detached state (discussed shortly) so that its resources can be recovered. If the thread was already in the detached state, `pthread_join` can fail, returning `EINVAL`, although this behavior is implementation-specific.

If we're not interested in a thread's return value, we can set *rval_ptr* to `NULL`. In this case, calling `pthread_join` allows us to wait for the specified thread, but does not retrieve the thread's termination status.

Example

Figure 11.3 shows how to fetch the exit code from a thread that has terminated.

```
#include "apue.h"
#include <pthread.h>

void *
thr_fn1(void *arg)
{
    printf("thread 1 returning\n");
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
```

```
    printf("thread 2 exiting\n");
    pthread_exit((void *)2);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_exit(err, "can't create thread 1");
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_exit(err, "can't create thread 2");
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 1");
    printf("thread 1 exit code %ld\n", (long)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 2");
    printf("thread 2 exit code %ld\n", (long)tret);
    exit(0);
}
```

Figure 11.3 Fetching the thread exit status

Running the program in Figure 11.3 gives us

```
$ ./a.out
thread 1 returning
thread 2 exiting
thread 1 exit code 1
thread 2 exit code 2
```

As we can see, when a thread exits by calling `pthread_exit` or by simply returning from the start routine, the exit status can be obtained by another thread by calling `pthread_join`. □

The typeless pointer passed to `pthread_create` and `pthread_exit` can be used to pass more than a single value. The pointer can be used to pass the address of a structure containing more complex information. Be careful that the memory used for the structure is still valid when the caller has completed. If the structure was allocated on the caller's stack, for example, the memory contents might have changed by the time the structure is used. If a thread allocates a structure on its stack and passes a pointer to this structure to `pthread_exit`, then the stack might be destroyed and its memory reused for something else by the time the caller of `pthread_join` tries to use it.

Example

The program in Figure 11.4 shows the problem with using an automatic variable (allocated on the stack) as the argument to `pthread_exit`.

```
#include "apue.h"
#include <pthread.h>

struct foo {
    int a, b, c, d;
};

void
printfoo(const char *s, const struct foo *fp)
{
    printf("%s", s);
    printf("  structure at 0x%lx\n", (unsigned long)fp);
    printf("  foo.a = %d\n", fp->a);
    printf("  foo.b = %d\n", fp->b);
    printf("  foo.c = %d\n", fp->c);
    printf("  foo.d = %d\n", fp->d);
}

void *
thr_fn1(void *arg)
{
    struct foo  foo = {1, 2, 3, 4};

    printfoo("thread 1:\n", &foo);
    pthread_exit((void *)&foo);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2: ID is %lu\n", (unsigned long)pthread_self());
    pthread_exit((void *)0);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    struct foo  *fp;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_exit(err, "can't create thread 1");
    err = pthread_join(tid1, (void *)&fp);
    if (err != 0)
        err_exit(err, "can't join with thread 1");
    sleep(1);
    printf("parent starting second thread\n");
```

```
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_exit(err, "can't create thread 2");
    sleep(1);
    printfoo("parent:\n", fp);
    exit(0);
}
```

Figure 11.4 Incorrect use of `pthread_exit` argument

When we run this program on Linux, we get

```
$ ./a.out
thread 1:
  structure at 0x7f2c83682ed0
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
parent starting second thread
thread 2: ID is 139829159933696
parent:
  structure at 0x7f2c83682ed0
  foo.a = -2090321472
  foo.b = 32556
  foo.c = 1
  foo.d = 0
```

Of course, the results vary, depending on the memory architecture, the compiler, and the implementation of the threads library. The results on Solaris are similar:

```
$ ./a.out
thread 1:
  structure at 0xffffffff7f0fbf30
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
parent starting second thread
thread 2: ID is 3
parent:
  structure at 0xffffffff7f0fbf30
  foo.a = -1
  foo.b = 2136969048
  foo.c = -1
  foo.d = 2138049024
```

As we can see, the contents of the structure (allocated on the stack of thread *tid1*) have changed by the time the main thread can access the structure. Note how the stack of the second thread (*tid2*) has overwritten the first thread's stack. To solve this problem, we can either use a global structure or allocate the structure using `malloc`.

On Mac OS X, we get different results:

```
$ ./a.out
thread 1:
  structure at 0x1000b6f00
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
parent starting second thread
thread 2: ID is 4295716864
parent:
  structure at 0x1000b6f00
Segmentation fault (core dumped)
```

In this case, the memory is no longer valid when the parent tries to access the structure passed to it by the first thread that exited, and the parent is sent the `SIGSEGV` signal.

On FreeBSD, the memory hasn't been overwritten by the time the parent accesses it, and we get

```
thread 1:
  structure at 0xbf9fef88
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
parent starting second thread
thread 2: ID is 673279680
parent:
  structure at 0xbf9fef88
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
```

Even though the memory is still intact after the thread exits, we can't depend on this always being the case. It certainly isn't what we observe on the other platforms. □

One thread can request that another in the same process be canceled by calling the `pthread_cancel` function.

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);
```

Returns: 0 if OK, error number on failure

In the default circumstances, `pthread_cancel` will cause the thread specified by *tid* to behave as if it had called `pthread_exit` with an argument of `PTHREAD_CANCELED`. However, a thread can elect to ignore or otherwise control how it is canceled. We will discuss this in detail in Section 12.7. Note that `pthread_cancel` doesn't wait for the thread to terminate; it merely makes the request.

A thread can arrange for functions to be called when it exits, similar to the way that the `atexit` function (Section 7.3) can be used by a process to arrange that functions are to be called when the process exits. The functions are known as *thread cleanup handlers*. More than one cleanup handler can be established for a thread. The handlers are recorded in a stack, which means that they are executed in the reverse order from that with which they were registered.

```
#include <pthread.h>

void pthread_cleanup_push(void (*rtn)(void *), void *arg);

void pthread_cleanup_pop(int execute);
```

The `pthread_cleanup_push` function schedules the cleanup function, *rtn*, to be called with the single argument, *arg*, when the thread performs one of the following actions:

- Makes a call to `pthread_exit`
- Responds to a cancellation request
- Makes a call to `pthread_cleanup_pop` with a nonzero *execute* argument

If the *execute* argument is set to zero, the cleanup function is not called. In either case, `pthread_cleanup_pop` removes the cleanup handler established by the last call to `pthread_cleanup_push`.

A restriction with these functions is that, because they can be implemented as macros, they must be used in matched pairs within the same scope in a thread. The macro definition of `pthread_cleanup_push` can include a `{` character, in which case the matching `}` character is in the `pthread_cleanup_pop` definition.

Example

Figure 11.5 shows how to use thread cleanup handlers. Although the example is somewhat contrived, it illustrates the mechanics involved. Note that although we never intend to pass zero as an argument to the thread start-up routines, we still need to match calls to `pthread_cleanup_pop` with the calls to `pthread_cleanup_push`; otherwise, the program might not compile.

```
#include "apue.h"
#include <pthread.h>

void
cleanup(void *arg)
{
    printf("cleanup: %s\n", (char *)arg);
}

void *
thr_fn1(void *arg)
```

```
{
    printf("thread 1 start\n");
    pthread_cleanup_push(cleanup, "thread 1 first handler");
    pthread_cleanup_push(cleanup, "thread 1 second handler");
    printf("thread 1 push complete\n");
    if (arg)
        return((void *)1);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2 start\n");
    pthread_cleanup_push(cleanup, "thread 2 first handler");
    pthread_cleanup_push(cleanup, "thread 2 second handler");
    printf("thread 2 push complete\n");
    if (arg)
        pthread_exit((void *)2);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    pthread_exit((void *)2);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, (void *)1);
    if (err != 0)
        err_exit(err, "can't create thread 1");
    err = pthread_create(&tid2, NULL, thr_fn2, (void *)1);
    if (err != 0)
        err_exit(err, "can't create thread 2");
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 1");
    printf("thread 1 exit code %ld\n", (long)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 2");
    printf("thread 2 exit code %ld\n", (long)tret);
    exit(0);
}
```

Figure 11.5 Thread cleanup handler

Running the program in Figure 11.5 on Linux or Solaris gives us

```
$ ./a.out
thread 1 start
thread 1 push complete
thread 2 start
thread 2 push complete
cleanup: thread 2 second handler
cleanup: thread 2 first handler
thread 1 exit code 1
thread 2 exit code 2
```

From the output, we can see that both threads start properly and exit, but that only the second thread's cleanup handlers are called. Thus, if the thread terminates by returning from its start routine, its cleanup handlers are not called, although this behavior varies among implementations. Also note that the cleanup handlers are called in the reverse order from which they were installed.

If we run the same program on FreeBSD or Mac OS X, we see that the program incurs a segmentation violation and drops core. This happens because on these systems, `pthread_cleanup_push` is implemented as a macro that stores some context on the stack. When thread 1 returns in between the call to `pthread_cleanup_push` and the call to `pthread_cleanup_pop`, the stack is overwritten and these platforms try to use this (now corrupted) context when they invoke the cleanup handlers. In the Single UNIX Specification, returning while in between a matched pair of calls to `pthread_cleanup_push` and `pthread_cleanup_pop` results in undefined behavior. The only portable way to return in between these two functions is to call `pthread_exit`. □

By now, you should begin to see similarities between the thread functions and the process functions. Figure 11.6 summarizes the similar functions.

Process primitive	Thread primitive	Description
`fork`	`pthread_create`	create a new flow of control
`exit`	`pthread_exit`	exit from an existing flow of control
`waitpid`	`pthread_join`	get exit status from flow of control
`atexit`	`pthread_cleanup_push`	register function to be called at exit from flow of control
`getpid`	`pthread_self`	get ID for flow of control
`abort`	`pthread_cancel`	request abnormal termination of flow of control

Figure 11.6 Comparison of process and thread primitives

By default, a thread's termination status is retained until we call `pthread_join` for that thread. A thread's underlying storage can be reclaimed immediately on termination if the thread has been *detached*. After a thread is detached, we can't use the `pthread_join` function to wait for its termination status, because calling `pthread_join` for a detached thread results in undefined behavior. We can detach a thread by calling `pthread_detach`.

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

Returns: 0 if OK, error number on failure

As we will see in the next chapter, we can create a thread that is already in the detached state by modifying the thread attributes we pass to `pthread_create`.

11.6 Thread Synchronization

When multiple threads of control share the same memory, we need to make sure that each thread sees a consistent view of its data. If each thread uses variables that other threads don't read or modify, no consistency problems will exist. Similarly, if a variable is read-only, there is no consistency problem with more than one thread reading its value at the same time. However, when one thread can modify a variable that other threads can read or modify, we need to synchronize the threads to ensure that they don't use an invalid value when accessing the variable's memory contents.

When one thread modifies a variable, other threads can potentially see inconsistencies when reading the value of that variable. On processor architectures in which the modification takes more than one memory cycle, this can happen when the memory read is interleaved between the memory write cycles. Of course, this behavior is architecture dependent, but portable programs can't make any assumptions about what type of processor architecture is being used.

Figure 11.7 shows a hypothetical example of two threads reading and writing the same variable. In this example, thread A reads the variable and then writes a new value to it, but the write operation takes two memory cycles. If thread B reads the same variable between the two write cycles, it will see an inconsistent value.

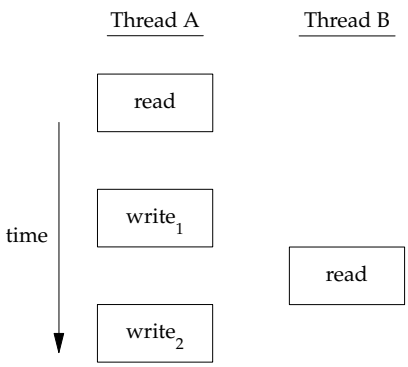


Figure 11.7 Interleaved memory cycles with two threads

To solve this problem, the threads have to use a lock that will allow only one thread to access the variable at a time. Figure 11.8 shows this synchronization. If it wants to

read the variable, thread B acquires a lock. Similarly, when thread A updates the variable, it acquires the same lock. Thus thread B will be unable to read the variable until thread A releases the lock.

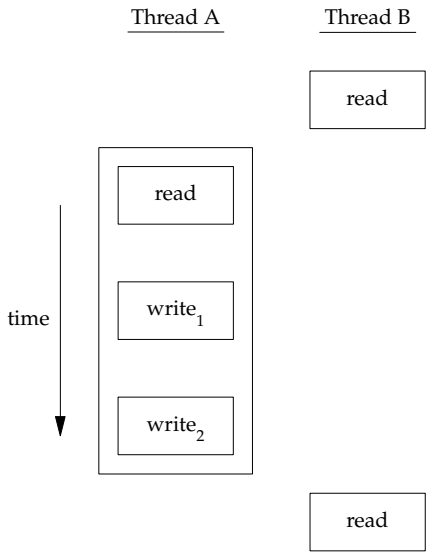


Figure 11.8 Two threads synchronizing memory access

We also need to synchronize two or more threads that might try to modify the same variable at the same time. Consider the case in which we increment a variable (Figure 11.9). The increment operation is usually broken down into three steps.

1. Read the memory location into a register.
2. Increment the value in the register.
3. Write the new value back to the memory location.

If two threads try to increment the same variable at almost the same time without synchronizing with each other, the results can be inconsistent. You end up with a value that is either one or two greater than before, depending on the value observed when the second thread starts its operation. If the second thread performs step 1 before the first thread performs step 3, the second thread will read the same initial value as the first thread, increment it, and write it back, with no net effect.

If the modification is atomic, then there isn't a race. In the previous example, if the increment takes only one memory cycle, then no race exists. If our data always appears to be *sequentially consistent*, then we need no additional synchronization. Our operations are sequentially consistent when multiple threads can't observe inconsistencies in our data. In modern computer systems, memory accesses take multiple bus cycles, and multiprocessors generally interleave bus cycles among multiple processors, so we aren't guaranteed that our data is sequentially consistent.

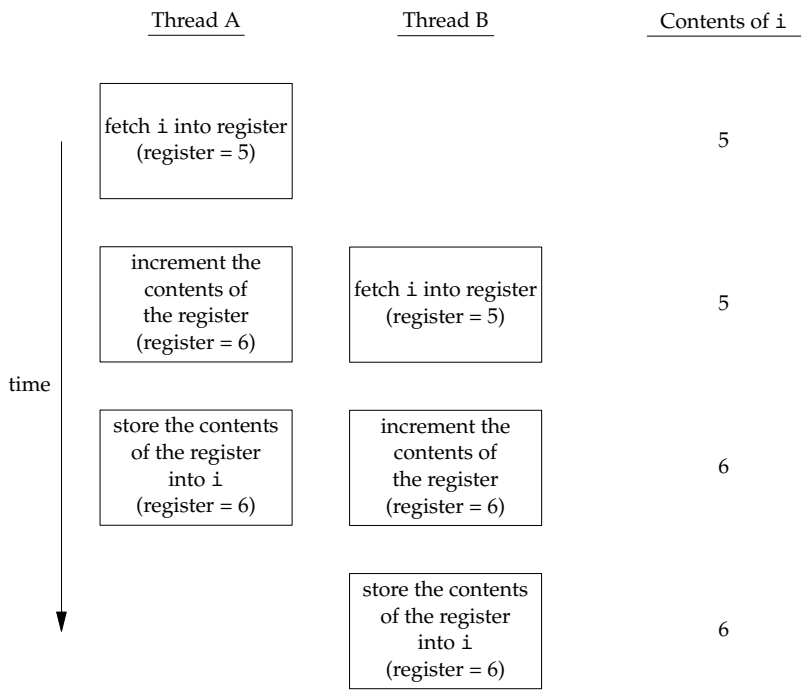


Figure 11.9 Two unsynchronized threads incrementing the same variable

In a sequentially consistent environment, we can explain modifications to our data as a sequential step of operations taken by the running threads. We can say such things as "Thread A incremented the variable, then thread B incremented the variable, so its value is two greater than before" or "Thread B incremented the variable, then thread A incremented the variable, so its value is two greater than before." No possible ordering of the two threads can result in any other value of the variable.

Besides the computer architecture, races can arise from the ways in which our programs use variables, creating places where it is possible to view inconsistencies. For example, we might increment a variable and then make a decision based on its value. The combination of the increment step and the decision-making step isn't atomic, which opens a window where inconsistencies can arise.

11.6.1 Mutexes

We can protect our data and ensure access by only one thread at a time by using the pthreads mutual-exclusion interfaces. A *mutex* is basically a lock that we set (lock) before accessing a shared resource and release (unlock) when we're done. While it is set, any other thread that tries to set it will block until we release it. If more than one thread is blocked when we unlock the mutex, then all threads blocked on the lock will be made runnable, and the first one to run will be able to set the lock. The others will

see that the mutex is still locked and go back to waiting for it to become available again. In this way, only one thread will proceed at a time.

This mutual-exclusion mechanism works only if we design our threads to follow the same data-access rules. The operating system doesn't serialize access to data for us. If we allow one thread to access a shared resource without first acquiring a lock, then inconsistencies can occur even though the rest of our threads do acquire the lock before attempting to access the shared resource.

A mutex variable is represented by the `pthread_mutex_t` data type. Before we can use a mutex variable, we must first initialize it by either setting it to the constant `PTHREAD_MUTEX_INITIALIZER` (for statically allocated mutexes only) or calling `pthread_mutex_init`. If we allocate the mutex dynamically (by calling `malloc`, for example), then we need to call `pthread_mutex_destroy` before freeing the memory.

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Both return: 0 if OK, error number on failure

To initialize a mutex with the default attributes, we set *attr* to `NULL`. We will discuss mutex attributes in Section 12.4.

To lock a mutex, we call `pthread_mutex_lock`. If the mutex is already locked, the calling thread will block until the mutex is unlocked. To unlock a mutex, we call `pthread_mutex_unlock`.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

All return: 0 if OK, error number on failure

If a thread can't afford to block, it can use `pthread_mutex_trylock` to lock the mutex conditionally. If the mutex is unlocked at the time `pthread_mutex_trylock` is called, then `pthread_mutex_trylock` will lock the mutex without blocking and return 0. Otherwise, `pthread_mutex_trylock` will fail, returning `EBUSY` without locking the mutex.

Example

Figure 11.10 illustrates a mutex used to protect a data structure. When more than one thread needs to access a dynamically allocated object, we can embed a reference count in the object to ensure that we don't free its memory before all threads are done using it.

```
#include <stdlib.h>
#include <pthread.h>

struct foo {
    int             f_count;
    pthread_mutex_t f_lock;
    int             f_id;
    /* ... more stuff here ... */
};

struct foo *
foo_alloc(int id) /* allocate the object */
{
    struct foo *fp;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        fp->f_id = id;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        /* ... continue initialization ... */
    }
    return(fp);
}

void
foo_hold(struct foo *fp) /* add a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
    fp->f_count++;
    pthread_mutex_unlock(&fp->f_lock);
}

void
foo_rele(struct foo *fp) /* release a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
    if (--fp->f_count == 0) { /* last reference */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

Figure 11.10 Using a mutex to protect a data structure

We lock the mutex before incrementing the reference count, decrementing the reference count, and checking whether the reference count reaches zero. No locking is

necessary when we initialize the reference count to 1 in the `foo_alloc` function, because the allocating thread is the only reference to it so far. If we were to place the structure on a list at this point, it could be found by other threads, so we would need to lock it first.

Before using the object, threads are expected to add a reference to it by calling `foo_hold`. When they are done, they must call `foo_rele` to release the reference. When the last reference is released, the object's memory is freed.

In this example, we have ignored how threads find an object before calling `foo_hold`. Even though the reference count is zero, it would be a mistake for `foo_rele` to free the object's memory if another thread is blocked on the mutex in a call to `foo_hold`. We can avoid this problem by ensuring that the object can't be found before freeing its memory. We'll see how to do this in the examples that follow. □

11.6.2 Deadlock Avoidance

A thread will deadlock itself if it tries to lock the same mutex twice, but there are less obvious ways to create deadlocks with mutexes. For example, when we use more than one mutex in our programs, a deadlock can occur if we allow one thread to hold a mutex and block while trying to lock a second mutex at the same time that another thread holding the second mutex tries to lock the first mutex. Neither thread can proceed, because each needs a resource that is held by the other, so we have a deadlock.

Deadlocks can be avoided by carefully controlling the order in which mutexes are locked. For example, assume that you have two mutexes, A and B, that you need to lock at the same time. If all threads always lock mutex A before mutex B, no deadlock can occur from the use of the two mutexes (but you can still deadlock on other resources). Similarly, if all threads always lock mutex B before mutex A, no deadlock will occur. You'll have the potential for a deadlock only when one thread attempts to lock the mutexes in the opposite order from another thread.

Sometimes, an application's architecture makes it difficult to apply a lock ordering. If enough locks and data structures are involved that the functions you have available can't be molded to fit a simple hierarchy, then you'll have to try some other approach. In this case, you might be able to release your locks and try again at a later time. You can use the `pthread_mutex_trylock` interface to avoid deadlocking in this case. If you are already holding locks and `pthread_mutex_trylock` is successful, then you can proceed. If it can't acquire the lock, however, you can release the locks you already hold, clean up, and try again later.

Example

In this example, we update Figure 11.10 to show the use of two mutexes. We avoid deadlocks by ensuring that when we need to acquire two mutexes at the same time, we always lock them in the same order. The second mutex protects a hash list that we use to keep track of the `foo` data structures. Thus the `hashlock` mutex protects both the `fh` hash table and the `f_next` hash link field in the `foo` structure. The `f_lock` mutex in the `foo` structure protects access to the remainder of the `foo` structure's fields.

```
#include <stdlib.h>
#include <pthread.h>

#define NHASH 29
#define HASH(id) (((unsigned long)id)%NHASH)

struct foo *fh[NHASH];

pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;

struct foo {
    int             f_count;
    pthread_mutex_t f_lock;
    int             f_id;
    struct foo     *f_next; /* protected by hashlock */
    /* ... more stuff here ... */
};

struct foo *
foo_alloc(int id) /* allocate the object */
{
    struct foo  *fp;
    int         idx;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        fp->f_id = id;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        idx = HASH(id);
        pthread_mutex_lock(&hashlock);
        fp->f_next = fh[idx];
        fh[idx] = fp;
        pthread_mutex_lock(&fp->f_lock);
        pthread_mutex_unlock(&hashlock);
        /* ... continue initialization ... */
        pthread_mutex_unlock(&fp->f_lock);
    }
    return(fp);
}

void
foo_hold(struct foo *fp) /* add a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
    fp->f_count++;
    pthread_mutex_unlock(&fp->f_lock);
}

struct foo *
foo_find(int id) /* find an existing object */
{
```

```
    struct foo  *fp;

    pthread_mutex_lock(&hashlock);
    for (fp = fh[HASH(id)]; fp != NULL; fp = fp->f_next) {
        if (fp->f_id == id) {
            foo_hold(fp);
            break;
        }
    }
    pthread_mutex_unlock(&hashlock);
    return(fp);
}

void
foo_rele(struct foo *fp) /* release a reference to the object */
{
    struct foo  *tfp;
    int         idx;

    pthread_mutex_lock(&fp->f_lock);
    if (fp->f_count == 1) { /* last reference */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_lock(&hashlock);
        pthread_mutex_lock(&fp->f_lock);
        /* need to recheck the condition */
        if (fp->f_count != 1) {
            fp->f_count--;
            pthread_mutex_unlock(&fp->f_lock);
            pthread_mutex_unlock(&hashlock);
            return;
        }
        /* remove from list */
        idx = HASH(fp->f_id);
        tfp = fh[idx];
        if (tfp == fp) {
            fh[idx] = fp->f_next;
        } else {
            while (tfp->f_next != fp)
                tfp = tfp->f_next;
            tfp->f_next = fp->f_next;
        }
        pthread_mutex_unlock(&hashlock);
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        fp->f_count--;
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

Figure 11.11 Using two mutexes

Comparing Figure 11.11 with Figure 11.10, we see that our allocation function now locks the hash list lock, adds the new structure to a hash bucket, and before unlocking the hash list lock, locks the mutex in the new structure. Since the new structure is placed on a global list, other threads can find it, so we need to block them if they try to access the new structure, until we are done initializing it.

The `foo_find` function locks the hash list lock and searches for the requested structure. If it is found, we increase the reference count and return a pointer to the structure. Note that we honor the lock ordering by locking the hash list lock in `foo_find` before `foo_hold` locks the `foo` structure's `f_lock` mutex.

Now with two locks, the `foo_rele` function is more complicated. If this is the last reference, we need to unlock the structure mutex so that we can acquire the hash list lock, since we'll need to remove the structure from the hash list. Then we reacquire the structure mutex. Because we could have blocked since the last time we held the structure mutex, we need to recheck the condition to see whether we still need to free the structure. If another thread found the structure and added a reference to it while we blocked to honor the lock ordering, we simply need to decrement the reference count, unlock everything, and return.

This locking approach is complex, so we need to revisit our design. We can simplify things considerably by using the hash list lock to protect the structure reference count, too. The structure mutex can be used to protect everything else in the `foo` structure. Figure 11.12 reflects this change.

```
#include <stdlib.h>
#include <pthread.h>

#define NHASH 29
#define HASH(id) (((unsigned long)id)%NHASH)

struct foo *fh[NHASH];
pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;

struct foo {
    int             f_count; /* protected by hashlock */
    pthread_mutex_t f_lock;
    int             f_id;
    struct foo     *f_next; /* protected by hashlock */
    /* ... more stuff here ... */
};

struct foo *
foo_alloc(int id) /* allocate the object */
{
    struct foo  *fp;
    int         idx;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        fp->f_id = id;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
```

```
            return(NULL);
        }
        idx = HASH(id);
        pthread_mutex_lock(&hashlock);
        fp->f_next = fh[idx];
        fh[idx] = fp;
        pthread_mutex_lock(&fp->f_lock);
        pthread_mutex_unlock(&hashlock);
        /* ... continue initialization ... */
        pthread_mutex_unlock(&fp->f_lock);
    }
    return(fp);
}

void
foo_hold(struct foo *fp) /* add a reference to the object */
{
    pthread_mutex_lock(&hashlock);
    fp->f_count++;
    pthread_mutex_unlock(&hashlock);
}

struct foo *
foo_find(int id) /* find an existing object */
{
    struct foo  *fp;

    pthread_mutex_lock(&hashlock);
    for (fp = fh[HASH(id)]; fp != NULL; fp = fp->f_next) {
        if (fp->f_id == id) {
            fp->f_count++;
            break;
        }
    }
    pthread_mutex_unlock(&hashlock);
    return(fp);
}

void
foo_rele(struct foo *fp) /* release a reference to the object */
{
    struct foo  *tfp;
    int         idx;

    pthread_mutex_lock(&hashlock);
    if (--fp->f_count == 0) { /* last reference, remove from list */
        idx = HASH(fp->f_id);
        tfp = fh[idx];
        if (tfp == fp) {
            fh[idx] = fp->f_next;
        } else {
            while (tfp->f_next != fp)
```

```
                tfp = tfp->f_next;
            tfp->f_next = fp->f_next;
        }
        pthread_mutex_unlock(&hashlock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        pthread_mutex_unlock(&hashlock);
    }
}
```

Figure 11.12 Simplified locking

Note how much simpler the program in Figure 11.12 is compared to the program in Figure 11.11. The lock-ordering issues surrounding the hash list and the reference count go away when we use the same lock for both purposes. Multithreaded software design involves these types of trade-offs. If your locking granularity is too coarse, you end up with too many threads blocking behind the same locks, with little improvement possible from concurrency. If your locking granularity is too fine, then you suffer bad performance from excess locking overhead, and you end up with complex code. As a programmer, you need to find the correct balance between code complexity and performance, while still satisfying your locking requirements. □

11.6.3 `pthread_mutex_timedlock` Function

One additional mutex primitive allows us to bound the time that a thread blocks when a mutex it is trying to acquire is already locked. The `pthread_mutex_timedlock` function is equivalent to `pthread_mutex_lock`, but if the timeout value is reached, `pthread_mutex_timedlock` will return the error code `ETIMEDOUT` without locking the mutex.

```
#include <pthread.h>
#include <time.h>

int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
                            const struct timespec *restrict tsptr);
```

Returns: 0 if OK, error number on failure

The timeout specifies how long we are willing to wait in terms of absolute time (as opposed to relative time; we specify that we are willing to block until time X instead of saying that we are willing to block for Y seconds). The timeout is represented by the `timespec` structure, which describes time in terms of seconds and nanoseconds.

Example

In Figure 11.13, we see how to use `pthread_mutex_timedlock` to avoid blocking indefinitely.

```
#include "apue.h"
#include <pthread.h>

int
main(void)
{
    int err;
    struct timespec tout;
    struct tm *tmp;
    char buf[64];
    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

    pthread_mutex_lock(&lock);
    printf("mutex is locked\n");
    clock_gettime(CLOCK_REALTIME, &tout);
    tmp = localtime(&tout.tv_sec);
    strftime(buf, sizeof(buf), "%r", tmp);
    printf("current time is %s\n", buf);
    tout.tv_sec += 10;  /* 10 seconds from now */
    /* caution: this could lead to deadlock */
    err = pthread_mutex_timedlock(&lock, &tout);
    clock_gettime(CLOCK_REALTIME, &tout);
    tmp = localtime(&tout.tv_sec);
    strftime(buf, sizeof(buf), "%r", tmp);
    printf("the time is now %s\n", buf);
    if (err == 0)
        printf("mutex locked again!\n");
    else
        printf("can't lock mutex again: %s\n", strerror(err));
    exit(0);
}
```

Figure 11.13 Using `pthread_mutex_timedlock`

Here is the output from the program in Figure 11.13.

```
$ ./a.out
mutex is locked
current time is 11:41:58 AM
the time is now 11:42:08 AM
can't lock mutex again: Connection timed out
```

This program deliberately locks a mutex it already owns to demonstrate how `pthread_mutex_timedlock` works. This strategy is not recommended in practice, because it can lead to deadlock.

Note that the time blocked can vary for several reasons: the start time could have been in the middle of a second, the resolution of the system's clock might not be fine enough to support the resolution of our timeout, or scheduling delays could prolong the amount of time until the program continues execution. □

Mac OS X 10.6.8 doesn't support `pthread_mutex_timedlock` yet, but FreeBSD 8.0, Linux 3.2.0, and Solaris 10 do support it, although Solaris still bundles it in the real-time library, `librt`. Solaris 10 also provides an alternative function that uses a relative timeout.

11.6.4 Reader–Writer Locks

Reader–writer locks are similar to mutexes, except that they allow for higher degrees of parallelism. With a mutex, the state is either locked or unlocked, and only one thread can lock it at a time. Three states are possible with a reader–writer lock: locked in read mode, locked in write mode, and unlocked. Only one thread at a time can hold a reader–writer lock in write mode, but multiple threads can hold a reader–writer lock in read mode at the same time.

When a reader–writer lock is write locked, all threads attempting to lock it block until it is unlocked. When a reader–writer lock is read locked, all threads attempting to lock it in read mode are given access, but any threads attempting to lock it in write mode block until all the threads have released their read locks. Although implementations vary, reader–writer locks usually block additional readers if a lock is already held in read mode and a thread is blocked trying to acquire the lock in write mode. This prevents a constant stream of readers from starving waiting writers.

Reader–writer locks are well suited for situations in which data structures are read more often than they are modified. When a reader–writer lock is held in write mode, the data structure it protects can be modified safely, since only one thread at a time can hold the lock in write mode. When the reader–writer lock is held in read mode, the data structure it protects can be read by multiple threads, as long as the threads first acquire the lock in read mode.

Reader–writer locks are also called shared–exclusive locks. When a reader–writer lock is read locked, it is said to be locked in shared mode. When it is write locked, it is said to be locked in exclusive mode.

As with mutexes, reader–writer locks must be initialized before use and destroyed before freeing their underlying memory.

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *restrict attr);

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

Both return: 0 if OK, error number on failure

A reader–writer lock is initialized by calling `pthread_rwlock_init`. We can pass a null pointer for *attr* if we want the reader–writer lock to have the default attributes. We discuss reader–writer lock attributes in Section 12.4.2.

The Single UNIX Specification defines the `PTHREAD_RWLOCK_INITIALIZER` constant in the XSI option. It can be used to initialize a statically allocated reader–writer lock when the default attributes are sufficient.

Before freeing the memory backing a reader–writer lock, we need to call `pthread_rwlock_destroy` to clean it up. If `pthread_rwlock_init` allocated any

resources for the reader–writer lock, `pthread_rwlock_destroy` frees those resources. If we free the memory backing a reader–writer lock without first calling `pthread_rwlock_destroy`, any resources assigned to the lock will be lost.

To lock a reader–writer lock in read mode, we call `pthread_rwlock_rdlock`. To write lock a reader–writer lock, we call `pthread_rwlock_wrlock`. Regardless of how we lock a reader–writer lock, we can unlock it by calling `pthread_rwlock_unlock`.

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

All return: 0 if OK, error number on failure

Implementations might place a limit on the number of times a reader–writer lock can be locked in shared mode, so we need to check the return value of `pthread_rwlock_rdlock`. Even though `pthread_rwlock_wrlock` and `pthread_rwlock_unlock` have error returns, and technically we should always check for errors when we call functions that can potentially fail, we don't need to check them if we design our locking properly. The only error returns defined are when we use them improperly, such as with an uninitialized lock, or when we might deadlock by attempting to acquire a lock we already own. However, be aware that specific implementations might define additional error returns.

The Single UNIX Specification also defines conditional versions of the reader–writer locking primitives.

```
#include <pthread.h>

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

Both return: 0 if OK, error number on failure

When the lock can be acquired, these functions return 0. Otherwise, they return the error `EBUSY`. These functions can be used to avoid deadlocks in situations where conforming to a lock hierarchy is difficult, as we discussed previously.

Example

The program in Figure 11.14 illustrates the use of reader–writer locks. A queue of job requests is protected by a single reader–writer lock. This example shows a possible implementation of Figure 11.1, whereby multiple worker threads obtain jobs assigned to them by a single master thread.

```
#include <stdlib.h>
#include <pthread.h>

struct job {
    struct job *j_next;
    struct job *j_prev;
```

```
    pthread_t   j_id;   /* tells which thread handles this job */
    /* ... more stuff here ... */
};

struct queue {
    struct job      *q_head;
    struct job      *q_tail;
    pthread_rwlock_t q_lock;
};

/*
 * Initialize a queue.
 */
int
queue_init(struct queue *qp)
{
    int err;

    qp->q_head = NULL;
    qp->q_tail = NULL;
    err = pthread_rwlock_init(&qp->q_lock, NULL);
    if (err != 0)
        return(err);
    /* ... continue initialization ... */
    return(0);
}

/*
 * Insert a job at the head of the queue.
 */
void
job_insert(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = qp->q_head;
    jp->j_prev = NULL;
    if (qp->q_head != NULL)
        qp->q_head->j_prev = jp;
    else
        qp->q_tail = jp;    /* list was empty */
    qp->q_head = jp;
    pthread_rwlock_unlock(&qp->q_lock);
}

/*
 * Append a job on the tail of the queue.
 */
void
job_append(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    jp->j_next = NULL;
```

```
    jp->j_prev = qp->q_tail;
    if (qp->q_tail != NULL)
        qp->q_tail->j_next = jp;
    else
        qp->q_head = jp;    /* list was empty */
    qp->q_tail = jp;
    pthread_rwlock_unlock(&qp->q_lock);
}

/*
 * Remove the given job from a queue.
 */
void
job_remove(struct queue *qp, struct job *jp)
{
    pthread_rwlock_wrlock(&qp->q_lock);
    if (jp == qp->q_head) {
        qp->q_head = jp->j_next;
        if (qp->q_tail == jp)
            qp->q_tail = NULL;
        else
            jp->j_next->j_prev = jp->j_prev;
    } else if (jp == qp->q_tail) {
        qp->q_tail = jp->j_prev;
        jp->j_prev->j_next = jp->j_next;
    } else {
        jp->j_prev->j_next = jp->j_next;
        jp->j_next->j_prev = jp->j_prev;
    }
    pthread_rwlock_unlock(&qp->q_lock);
}

/*
 * Find a job for the given thread ID.
 */
struct job *
job_find(struct queue *qp, pthread_t id)
{
    struct job *jp;

    if (pthread_rwlock_rdlock(&qp->q_lock) != 0)
        return(NULL);

    for (jp = qp->q_head; jp != NULL; jp = jp->j_next)
        if (pthread_equal(jp->j_id, id))
            break;

    pthread_rwlock_unlock(&qp->q_lock);
    return(jp);
}
```

Figure 11.14 Using reader–writer locks

In this example, we lock the queue's reader–writer lock in write mode whenever we need to add a job to the queue or remove a job from the queue. Whenever we search the queue, we grab the lock in read mode, allowing all the worker threads to search the queue concurrently. Using a reader–writer lock will improve performance in this case only if threads search the queue much more frequently than they add or remove jobs.

The worker threads take only those jobs that match their thread ID off the queue. Since the job structures are used only by one thread at a time, they don't need any extra locking. □

11.6.5 Reader–Writer Locking with Timeouts

Just as with mutexes, the Single UNIX Specification provides functions to lock reader–writer locks with a timeout to give applications a way to avoid blocking indefinitely while trying to acquire a reader–writer lock. These functions are `pthread_rwlock_timedrdlock` and `pthread_rwlock_timedwrlock`.

```
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
                               const struct timespec *restrict tsptr);

int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
                               const struct timespec *restrict tsptr);
```

Both return: 0 if OK, error number on failure

These functions behave like their "untimed" counterparts. The *tsptr* argument points to a `timespec` structure specifying the time at which the thread should stop blocking. If they can't acquire the lock, these functions return the `ETIMEDOUT` error when the timeout expires. Like the `pthread_mutex_timedlock` function, the timeout specifies an absolute time, not a relative one.

11.6.6 Condition Variables

Condition variables are another synchronization mechanism available to threads. These synchronization objects provide a place for threads to rendezvous. When used with mutexes, condition variables allow threads to wait in a race-free way for arbitrary conditions to occur.

The condition itself is protected by a mutex. A thread must first lock the mutex to change the condition state. Other threads will not notice the change until they acquire the mutex, because the mutex must be locked to be able to evaluate the condition.

Before a condition variable is used, it must first be initialized. A condition variable, represented by the `pthread_cond_t` data type, can be initialized in two ways. We can assign the constant `PTHREAD_COND_INITIALIZER` to a statically allocated condition

variable, but if the condition variable is allocated dynamically, we can use the pthread_cond_init function to initialize it.

We can use the pthread_cond_destroy function to deinitialize a condition variable before freeing its underlying memory.

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);

int pthread_cond_destroy(pthread_cond_t *cond);
```

Both return: 0 if OK, error number on failure

Unless you need to create a conditional variable with nondefault attributes, the *attr* argument to pthread_cond_init can be set to NULL. We will discuss condition variable attributes in Section 12.4.3.

We use pthread_cond_wait to wait for a condition to be true. A variant is provided to return an error code if the condition hasn't been satisfied in the specified amount of time.

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict tsptr);
```

Both return: 0 if OK, error number on failure

The mutex passed to pthread_cond_wait protects the condition. The caller passes it locked to the function, which then atomically places the calling thread on the list of threads waiting for the condition and unlocks the mutex. This closes the window between the time that the condition is checked and the time that the thread goes to sleep waiting for the condition to change, so that the thread doesn't miss a change in the condition. When pthread_cond_wait returns, the mutex is again locked.

The pthread_cond_timedwait function provides the same functionality as the pthread_cond_wait function with the addition of the timeout (*tsptr*). The timeout value specifies how long we are willing to wait expressed as a timespec structure.

Just as we saw in Figure 11.13, we need to specify how long we are willing to wait as an absolute time instead of a relative time. For example, suppose we are willing to wait 3 minutes. Instead of translating 3 minutes into a timespec structure, we need to translate now + 3 minutes into a timespec structure.

We can use the clock_gettime function (Section 6.10) to get the current time expressed as a timespec structure. However, this function is not yet supported on all platforms. Alternatively, we can use the gettimeofday function to get the current time expressed as a timeval structure and translate it into a timespec structure. To

obtain the absolute time for the timeout value, we can use the following function (assuming the maximum time blocked is expressed in minutes):

```
#include <sys/time.h>
#include <stdlib.h>

void
maketimeout(struct timespec *tsp, long minutes)
{
    struct timeval now;

    /* get the current time */
    gettimeofday(&now, NULL);
    tsp->tv_sec = now.tv_sec;
    tsp->tv_nsec = now.tv_usec * 1000; /* usec to nsec */
    /* add the offset to get timeout value */
    tsp->tv_sec += minutes * 60;
}
```

If the timeout expires without the condition occurring, `pthread_cond_timedwait` will reacquire the mutex and return the error `ETIMEDOUT`. When it returns from a successful call to `pthread_cond_wait` or `pthread_cond_timedwait`, a thread needs to reevaluate the condition, since another thread might have run and already changed the condition.

There are two functions to notify threads that a condition has been satisfied. The `pthread_cond_signal` function will wake up at least one thread waiting on a condition, whereas the `pthread_cond_broadcast` function will wake up all threads waiting on a condition.

> The POSIX specification allows for implementations of `pthread_cond_signal` to wake up more than one thread, to make the implementation simpler.

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

Both return: 0 if OK, error number on failure

When we call `pthread_cond_signal` or `pthread_cond_broadcast`, we are said to be *signaling* the thread or condition. We have to be careful to signal the threads only after changing the state of the condition.

Example

Figure 11.15 shows an example of how to use a condition variable and a mutex together to synchronize threads.

```
#include <pthread.h>

struct msg {
    struct msg *m_next;
    /* ... more stuff here ... */
};

struct msg *workq;

pthread_cond_t qready = PTHREAD_COND_INITIALIZER;

pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;

void
process_msg(void)
{
    struct msg *mp;

    for (;;) {
        pthread_mutex_lock(&qlock);
        while (workq == NULL)
            pthread_cond_wait(&qready, &qlock);
        mp = workq;
        workq = mp->m_next;
        pthread_mutex_unlock(&qlock);
        /* now process the message mp */
    }
}

void
enqueue_msg(struct msg *mp)
{
    pthread_mutex_lock(&qlock);
    mp->m_next = workq;
    workq = mp;
    pthread_mutex_unlock(&qlock);
    pthread_cond_signal(&qready);
}
```

Figure 11.15 Using a condition variable

The condition is the state of the work queue. We protect the condition with a mutex and evaluate the condition in a `while` loop. When we put a message on the work queue, we need to hold the mutex, but we don't need to hold the mutex when we signal the waiting threads. As long as it is okay for a thread to pull the message off the queue before we call `cond_signal`, we can do this after releasing the mutex. Since we check the condition in a `while` loop, this doesn't present a problem; a thread will wake up, find that the queue is still empty, and go back to waiting again. If the code couldn't tolerate this race, we would need to hold the mutex when we signal the threads. □

11.6.7 Spin Locks

A spin lock is like a mutex, except that instead of blocking a process by sleeping, the process is blocked by busy-waiting (spinning) until the lock can be acquired. A spin lock could be used in situations where locks are held for short periods of times and threads don't want to incur the cost of being descheduled.

Spin locks are often used as low-level primitives to implement other types of locks. Depending on the system architecture, they can be implemented efficiently using test-and-set instructions. Although efficient, they can lead to wasting CPU resources: while a thread is spinning and waiting for a lock to become available, the CPU can't do anything else. This is why spin locks should be held only for short periods of time.

Spin locks are useful when used in a nonpreemptive kernel: besides providing a mutual exclusion mechanism, they block interrupts so an interrupt handler can't deadlock the system by trying to acquire a spin lock that is already locked (think of interrupts as another type of preemption). In these types of kernels, interrupt handlers can't sleep, so the only synchronization primitives they can use are spin locks.

However, at user level, spin locks are not as useful unless you are running in a real-time scheduling class that doesn't allow preemption. User-level threads running in a time-sharing scheduling class can be descheduled when their time quantum expires or when a thread with a higher scheduling priority becomes runnable. In these cases, if a thread is holding a spin lock, it will be put to sleep and other threads blocked on the lock will continue spinning longer than intended.

Many mutex implementations are so efficient that the performance of applications using mutex locks is equivalent to their performance if they had used spin locks. In fact, some mutex implementations will spin for a limited amount of time trying to acquire the mutex, and only sleep when the spin count threshold is reached. These factors, combined with advances in modern processors that allow them to context switch at faster and faster rates, make spin locks useful only in limited circumstances.

The interfaces for spin locks are similar to those for mutexes, making it relatively easy to replace one with the other. We can initialize a spin lock with the `pthread_spin_init` function. To deinitialize a spin lock, we can call the `pthread_spin_destroy` function.

```
#include <pthread.h>

int pthread_spin_init(pthread_spinlock_t *lock, int pshared);

int pthread_spin_destroy(pthread_spinlock_t *lock);
```

Both return: 0 if OK, error number on failure

Only one attribute is specified for spin locks, which matters only if the platform supports the Thread Process-Shared Synchronization option (now mandatory in the Single UNIX Specification; recall Figure 2.5). The *pshared* argument represents the *process-shared* attribute, which indicates how the spin lock will be acquired. If it is set to `PTHREAD_PROCESS_SHARED`, then the spin lock can be acquired by threads that have access to the lock's underlying memory, even if those threads are from different processes. Otherwise, the *pshared* argument is set to `PTHREAD_PROCESS_PRIVATE` and the spin lock can be accessed only from threads within the process that initialized it.

To lock the spin lock, we can call either `pthread_spin_lock`, which will spin until the lock is acquired, or `pthread_spin_trylock`, which will return the `EBUSY` error if the lock can't be acquired immediately. Note that `pthread_spin_trylock` doesn't spin. Regardless of how it was locked, a spin lock can be unlocked by calling `pthread_spin_unlock`.

```
#include <pthread.h>

int pthread_spin_lock(pthread_spinlock_t *lock);

int pthread_spin_trylock(pthread_spinlock_t *lock);

int pthread_spin_unlock(pthread_spinlock_t *lock);
```

All return: 0 if OK, error number on failure

Note that if a spin lock is currently unlocked, then the `pthread_spin_lock` function can lock it without spinning. If the thread already has it locked, the results are undefined. The call to `pthread_spin_lock` could fail with the `EDEADLK` error (or some other error), or the call could spin indefinitely. The behavior depends on the implementation. If we try to unlock a spin lock that is not locked, the results are also undefined.

If either `pthread_spin_lock` or `pthread_spin_trylock` returns 0, then the spin lock is locked. We need to be careful not to call any functions that might sleep while holding the spin lock. If we do, then we'll waste CPU resources by extending the time other threads will spin if they try to acquire it.

11.6.8 Barriers

Barriers are a synchronization mechanism that can be used to coordinate multiple threads working in parallel. A barrier allows each thread to wait until all cooperating threads have reached the same point, and then continue executing from there. We've already seen one form of barrier—the `pthread_join` function acts as a barrier to allow one thread to wait until another thread exits.

Barrier objects are more general than this, however. They allow an arbitrary number of threads to wait until all of the threads have completed processing, but the threads don't have to exit. They can continue working after all threads have reached the barrier.

We can use the `pthread_barrier_init` function to initialize a barrier, and we can use the `pthread_barrier_destroy` function to deinitialize a barrier.

```
#include <pthread.h>

int pthread_barrier_init(pthread_barrier_t *restrict barrier,
                         const pthread_barrierattr_t *restrict attr,
                         unsigned int count);

int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Both return: 0 if OK, error number on failure

When we initialize a barrier, we use the *count* argument to specify the number of threads that must reach the barrier before all of the threads will be allowed to continue. We use the *attr* argument to specify the attributes of the barrier object, which we'll look at more closely in the next chapter. For now, we can set *attr* to `NULL` to initialize a barrier with the default attributes. If the `pthread_barrier_init` function allocated any resources for the barrier, the resources will be freed when we deinitialize the barrier by calling the `pthread_barrier_destroy` function.

We use the `pthread_barrier_wait` function to indicate that a thread is done with its work and is ready to wait for all the other threads to catch up.

```
#include <pthread.h>

int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Returns: 0 or `PTHREAD_BARRIER_SERIAL_THREAD` if OK, error number on failure

The thread calling `pthread_barrier_wait` is put to sleep if the barrier count (set in the call to `pthread_barrier_init`) is not yet satisfied. If the thread is the last one to call `pthread_barrier_wait`, thereby satisfying the barrier count, all of the threads are awakened.

To one arbitrary thread, it will appear as if the `pthread_barrier_wait` function returned a value of `PTHREAD_BARRIER_SERIAL_THREAD`. The remaining threads see a return value of 0. This allows one thread to continue as the master to act on the results of the work done by all of the other threads.

Once the barrier count is reached and the threads are unblocked, the barrier can be used again. However, the barrier count can't be changed unless we call the `pthread_barrier_destroy` function followed by the `pthread_barrier_init` function with a different count.

Example

Figure 11.16 shows how a barrier can be used to synchronize threads cooperating on a single task.

```
#include "apue.h"
#include <pthread.h>
#include <limits.h>
#include <sys/time.h>

#define NTHR   8                /* number of threads */
#define NUMNUM 8000000L         /* number of numbers to sort */
#define TNUM   (NUMNUM/NTHR)    /* number to sort per thread */

long nums[NUMNUM];
long snums[NUMNUM];

pthread_barrier_t b;

#ifdef SOLARIS
#define heapsort qsort
#else
extern int heapsort(void *, size_t, size_t,
```

```
                    int (*)(const void *, const void *));
#endif

/*
 * Compare two long integers (helper function for heapsort)
 */
int
complong(const void *arg1, const void *arg2)
{
    long l1 = *(long *)arg1;
    long l2 = *(long *)arg2;

    if (l1 == l2)
        return 0;
    else if (l1 < l2)
        return -1;
    else
        return 1;
}

/*
 * Worker thread to sort a portion of the set of numbers.
 */
void *
thr_fn(void *arg)
{
    long    idx = (long)arg;

    heapsort(&nums[idx], TNUM, sizeof(long), complong);
    pthread_barrier_wait(&b);

    /*
     * Go off and perform more work ...
     */
    return((void *)0);
}

/*
 * Merge the results of the individual sorted ranges.
 */
void
merge()
{
    long    idx[NTHR];
    long    i, minidx, sidx, num;

    for (i = 0; i < NTHR; i++)
        idx[i] = i * TNUM;
    for (sidx = 0; sidx < NUMNUM; sidx++) {
        num = LONG_MAX;
        for (i = 0; i < NTHR; i++) {
            if ((idx[i] < (i+1)*TNUM) && (nums[idx[i]] < num)) {
                num = nums[idx[i]];
```

```
                minidx = i;
            }
        }
        snums[sidx] = nums[idx[minidx]];
        idx[minidx]++;
    }
}

int
main()
{
    unsigned long   i;
    struct timeval  start, end;
    long long       startusec, endusec;
    double          elapsed;
    int             err;
    pthread_t       tid;

    /*
     * Create the initial set of numbers to sort.
     */
    srandom(1);
    for (i = 0; i < NUMNUM; i++)
        nums[i] = random();

    /*
     * Create 8 threads to sort the numbers.
     */
    gettimeofday(&start, NULL);
    pthread_barrier_init(&b, NULL, NTHR+1);
    for (i = 0; i < NTHR; i++) {
        err = pthread_create(&tid, NULL, thr_fn, (void *)(i * TNUM));
        if (err != 0)
            err_exit(err, "can't create thread");
    }
    pthread_barrier_wait(&b);
    merge();
    gettimeofday(&end, NULL);

    /*
     * Print the sorted list.
     */
    startusec = start.tv_sec * 1000000 + start.tv_usec;
    endusec = end.tv_sec * 1000000 + end.tv_usec;
    elapsed = (double)(endusec - startusec) / 1000000.0;
    printf("sort took %.4f seconds\n", elapsed);
    for (i = 0; i < NUMNUM; i++)
        printf("%ld\n", snums[i]);
    exit(0);
}
```

Figure 11.16 Using a barrier

This example shows the use of a barrier in a simplified situation where the threads perform only one task. In more realistic situations, the worker threads will continue with other activities after the call to `pthread_barrier_wait` returns.

In the example, we use eight threads to divide the job of sorting 8 million numbers. Each thread sorts 1 million numbers using the heapsort algorithm (see Knuth [1998] for details). Then the main thread calls a function to merge the results.

We don't need to use the `PTHREAD_BARRIER_SERIAL_THREAD` return value from `pthread_barrier_wait` to decide which thread merges the results, because we use the main thread for this task. That is why we specify the barrier count as one more than the number of worker threads; the main thread counts as one waiter.

If we write a program to sort 8 million numbers with heapsort using 1 thread only, we will see a performance improvement when comparing it to the program in Figure 11.16. On a system with 8 cores, the single-threaded program sorted 8 million numbers in 12.14 seconds. On the same system, using 8 threads in parallel and 1 thread to merge the results, the same set of 8 million numbers was sorted in 1.91 seconds, 6 times faster. □

11.7 Summary

In this chapter, we introduced the concept of threads and discussed the POSIX.1 primitives available to create and destroy them. We also introduced the problem of thread synchronization. We discussed five fundamental synchronization mechanisms—mutexes, reader–writer locks, condition variables, spin locks, and barriers—and we saw how to use them to protect shared resources.

Exercises

11.1 Modify the example code shown in Figure 11.4 to pass the structure between the threads properly.

11.2 In the example code shown in Figure 11.14, what additional synchronization (if any) is necessary to allow the master thread to change the thread ID associated with a pending job? How would this affect the `job_remove` function?

11.3 Apply the techniques shown in Figure 11.15 to the worker thread example (Figures 11.1 and 11.14) to implement the worker thread function. Don't forget to update the `queue_init` function to initialize the condition variable and change the `job_insert` and `job_append` functions to signal the worker threads. What difficulties arise?

11.4 Which sequence of steps is correct?

1. Lock a mutex (`pthread_mutex_lock`).
2. Change the condition protected by the mutex.
3. Signal threads waiting on the condition (`pthread_cond_broadcast`).
4. Unlock the mutex (`pthread_mutex_unlock`).

or

1. Lock a mutex (`pthread_mutex_lock`).
2. Change the condition protected by the mutex.
3. Unlock the mutex (`pthread_mutex_unlock`).
4. Signal threads waiting on the condition (`pthread_cond_broadcast`).

11.5 What synchronization primitives would you need to implement a barrier? Provide an implementation of the `pthread_barrier_wait` function.

This page intentionally left blank

12

Thread Control

12.1 Introduction

In Chapter 11, we learned the basics about threads and thread synchronization. In this chapter, we will learn the details of controlling thread behavior. We will look at thread attributes and synchronization primitive attributes, which we ignored in the previous chapter in favor of the default behavior.

We will follow this with a look at how threads can keep data private from other threads in the same process. Then we will wrap up the chapter with a look at how some process-based system calls interact with threads.

12.2 Thread Limits

We discussed the `sysconf` function in Section 2.5.4. The Single UNIX Specification defines several limits associated with the operation of threads, which we didn't show in Figure 2.11. As with other system limits, the thread limits can be queried using `sysconf`. Figure 12.1 summarizes these limits.

As with the other limits reported by `sysconf`, use of these limits is intended to promote application portability among different operating system implementations. For example, if your application requires that you create four threads for every file you manage, you might have to limit the number of files you can manage concurrently if the system won't let you create enough threads.

Name of limit	Description	*name* argument
PTHREAD_DESTRUCTOR_ITERATIONS	maximum number of times an implementation will try to destroy the thread-specific data when a thread exits (Section 12.6)	_SC_THREAD_DESTRUCTOR_ITERATIONS
PTHREAD_KEYS_MAX	maximum number of keys that can be created by a process (Section 12.6)	_SC_THREAD_KEYS_MAX
PTHREAD_STACK_MIN	minimum number of bytes that can be used for a thread's stack (Section 12.3)	_SC_THREAD_STACK_MIN
PTHREAD_THREADS_MAX	maximum number of threads that can be created in a process (Section 12.3)	_SC_THREAD_THREADS_MAX

Figure 12.1 Thread limits and *name* arguments to `sysconf`

Figure 12.2 shows the values of the thread limits for the four implementations described in this book. If the implementation's limit is indeterminate, "no limit" is listed. This doesn't mean that the value is unlimited, however.

> Note that although an implementation may not provide access to these limits, that doesn't mean that the limits don't exist. It just means that the implementation doesn't provide us with a way to get at them using `sysconf`.

Limit	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
PTHREAD_DESTRUCTOR_ITERATIONS	4	4	4	no limit
PTHREAD_KEYS_MAX	256	1,024	512	no limit
PTHREAD_STACK_MIN	2,048	16,384	8,192	8,192
PTHREAD_THREADS_MAX	no limit	no limit	no limit	no limit

Figure 12.2 Examples of thread configuration limits

12.3 Thread Attributes

The pthread interface allows us to fine-tune the behavior of threads and synchronization objects by setting various attributes associated with each object. Generally, the functions for managing these attributes follow the same pattern:

1. Each object is associated with its own type of attribute object (threads with thread attributes, mutexes with mutex attributes, and so on). An attribute object can represent multiple attributes. The attribute object is opaque to applications. This means that applications aren't supposed to know anything about its internal structure, which promotes application portability. Instead, functions are provided to manage the attributes objects.

2. An initialization function exists to set the attributes to their default values.
3. Another function exists to destroy the attributes object. If the initialization function allocated any resources associated with the attributes object, the destroy function frees those resources.
4. Each attribute has a function to get the value of the attribute from the attribute object. Because the function returns 0 on success or an error number on failure, the value is returned to the caller by storing it in the memory location specified by one of the arguments.
5. Each attribute has a function to set the value of the attribute. In this case, the value is passed as an argument, *by value*.

In all the examples in which we called `pthread_create` in Chapter 11, we passed in a null pointer instead of passing in a pointer to a `pthread_attr_t` structure. We can use the `pthread_attr_t` structure to modify the default attributes, and associate these attributes with threads that we create. We use the `pthread_attr_init` function to initialize the `pthread_attr_t` structure. After calling `pthread_attr_init`, the `pthread_attr_t` structure contains the default values for all the thread attributes supported by the implementation.

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_destroy(pthread_attr_t *attr);
```

Both return: 0 if OK, error number on failure

To deinitialize a `pthread_attr_t` structure, we call `pthread_attr_destroy`. If an implementation of `pthread_attr_init` allocated any dynamic memory for the attribute object, `pthread_attr_destroy` will free that memory. In addition, `pthread_attr_destroy` will initialize the attribute object with invalid values, so if it is used by mistake, `pthread_create` will return an error code.

The thread attributes defined by POSIX.1 are summarized in Figure 12.3. POSIX.1 defines additional attributes in the Thread Execution Scheduling option, intended to support real-time applications, but we don't discuss them here. In Figure 12.3, we also show which platforms support each thread attribute.

Name	Description	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
detachstate	detached thread attribute	•	•	•	•
guardsize	guard buffer size in bytes at end of thread stack	•	•	•	•
stackaddr	lowest address of thread stack	•	•	•	•
stacksize	minimum size in bytes of thread stack	•	•	•	•

Figure 12.3 POSIX.1 thread attributes

In Section 11.5, we introduced the concept of detached threads. If we are no longer interested in an existing thread's termination status, we can use `pthread_detach` to allow the operating system to reclaim the thread's resources when the thread exits.

If we know that we don't need the thread's termination status at the time we create the thread, we can arrange for the thread to start out in the detached state by modifying the *detachstate* thread attribute in the `pthread_attr_t` structure. We can use the `pthread_attr_setdetachstate` function to set the *detachstate* thread attribute to one of two legal values: `PTHREAD_CREATE_DETACHED` to start the thread in the detached state or `PTHREAD_CREATE_JOINABLE` to start the thread normally, so its termination status can be retrieved by the application.

```
#include <pthread.h>

int pthread_attr_getdetachstate(const pthread_attr_t *restrict attr,
                                int *detachstate);

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

Both return: 0 if OK, error number on failure

We can call `pthread_attr_getdetachstate` to obtain the current *detachstate* attribute. The integer pointed to by the second argument is set to either `PTHREAD_CREATE_DETACHED` or `PTHREAD_CREATE_JOINABLE`, depending on the value of the attribute in the given `pthread_attr_t` structure.

Example

Figure 12.4 shows a function that can be used to create a thread in the detached state.

```
#include "apue.h"
#include <pthread.h>

int
makethread(void *(*fn)(void *), void *arg)
{
    int             err;
    pthread_t       tid;
    pthread_attr_t  attr;

    err = pthread_attr_init(&attr);
    if (err != 0)
        return(err);
    err = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    if (err == 0)
        err = pthread_create(&tid, &attr, fn, arg);
    pthread_attr_destroy(&attr);
    return(err);
}
```

Figure 12.4 Creating a thread in the detached state

Note that we ignore the return value from the call to `pthread_attr_destroy`. In this case, we initialized the thread attributes properly, so `pthread_attr_destroy` shouldn't fail. Nonetheless, if it does fail, cleaning up would be difficult: we would have to destroy the thread we just created, which might already be running, asynchronous to the execution of this function. When we choose to ignore the error

return from pthread_attr_destroy, the worst that can happen is that we leak a small amount of memory if pthread_attr_init had allocated any. But if pthread_attr_init succeeded in initializing the thread attributes and then pthread_attr_destroy failed to clean up, we have no recovery strategy anyway, because the attributes structure is opaque to the application. The only interface defined to clean up the structure is pthread_attr_destroy, and it just failed. □

Support for thread stack attributes is optional for a POSIX-conforming operating system, but is required if the system supports the XSI option in the Single UNIX Specification. At compile time, you can check whether your system supports each thread stack attribute by using the _POSIX_THREAD_ATTR_STACKADDR and _POSIX_THREAD_ATTR_STACKSIZE symbols. If one of these symbols is defined, then the system supports the corresponding thread stack attribute. Alternatively, you can check for support at runtime, by using the _SC_THREAD_ATTR_STACKADDR and _SC_THREAD_ATTR_STACKSIZE parameters to the sysconf function.

We can manage the stack attributes using the pthread_attr_getstack and pthread_attr_setstack functions.

```
#include <pthread.h>

int pthread_attr_getstack(const pthread_attr_t *restrict attr,
                          void **restrict stackaddr,
                          size_t *restrict stacksize);

int pthread_attr_setstack(pthread_attr_t *attr,
                          void *stackaddr, size_t stacksize);
```

Both return: 0 if OK, error number on failure

With a process, the amount of virtual address space is fixed. Since there is only one stack, its size usually isn't a problem. With threads, however, the same amount of virtual address space must be shared by all the thread stacks. You might have to reduce your default thread stack size if your application uses so many threads that the cumulative size of their stacks exceeds the available virtual address space. On the other hand, if your threads call functions that allocate large automatic variables or call functions many stack frames deep, you might need more than the default stack size.

If you run out of virtual address space for thread stacks, you can use malloc or mmap (see Section 14.8) to allocate space for an alternative stack and use pthread_attr_setstack to change the stack location of threads you create. The address specified by the *stackaddr* parameter is the lowest addressable address in the range of memory to be used as the thread's stack, aligned at the proper boundary for the processor architecture. Of course, this assumes that the virtual address range used by malloc or mmap is different from the range currently in use for a thread's stack.

The *stackaddr* thread attribute is defined as the lowest memory address for the stack. This is not necessarily the start of the stack, however. If stacks grow from higher addresses to lower addresses for a given processor architecture, the *stackaddr* thread attribute will be the end of the stack instead of the beginning.

An application can also get and set the *stacksize* thread attribute using the pthread_attr_getstacksize and pthread_attr_setstacksize functions.

```
#include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                              size_t *restrict stacksize);

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

Both return: 0 if OK, error number on failure

The `pthread_attr_setstacksize` function is useful when you want to change the default stack size but don't want to deal with allocating the thread stacks on your own. When setting the *stacksize* attribute, the size we choose can't be smaller than `PTHREAD_STACK_MIN`.

The *guardsize* thread attribute controls the size of the memory extent after the end of the thread's stack to protect against stack overflow. Its default value is implementation defined, but a commonly used value is the system page size. We can set the *guardsize* thread attribute to 0 to disable this feature: no guard buffer will be provided in this case. Also, if we change the *stackaddr* thread attribute, the system assumes that we will be managing our own stacks and disables stack guard buffers, just as if we had set the *guardsize* thread attribute to 0.

```
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
                              size_t *restrict guardsize);

int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
```

Both return: 0 if OK, error number on failure

If the *guardsize* thread attribute is modified, the operating system might round it up to an integral multiple of the page size. If the thread's stack pointer overflows into the guard area, the application will receive an error, possibly with a signal.

The Single UNIX Specification defines several other optional thread attributes intended for use by real-time applications. We will not discuss them here.

Threads have other attributes not represented by the `pthread_attr_t` structure: the cancelability state and the cancelability type. We discuss them in Section 12.7.

12.4 Synchronization Attributes

Just as threads have attributes, so too do their synchronization objects. In Section 11.6.7, we saw how spin locks have one attribute called the *process-shared* attribute. In this section, we discuss the attributes of mutexes, reader–writer locks, condition variables, and barriers.

12.4.1 Mutex Attributes

Mutex attributes are represented by a `pthread_mutexattr_t` structure. Whenever we initialized a mutex in Chapter 11, we accepted the default attributes by using the

PTHREAD_MUTEX_INITIALIZER constant or by calling the pthread_mutex_init function with a null pointer for the argument that points to the mutex attribute structure.

When dealing with nondefault attributes, we use pthread_mutexattr_init to initialize a pthread_mutexattr_t structure and pthread_mutexattr_destroy to deinitialize one.

```
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Both return: 0 if OK, error number on failure

The pthread_mutexattr_init function will initialize the pthread_mutexattr_t structure with the default mutex attributes. There are three attributes of interest: the *process-shared* attribute, the *robust* attribute, and the *type* attribute. Within POSIX.1, the *process-shared* attribute is optional; you can test whether a platform supports it by checking whether the _POSIX_THREAD_PROCESS_SHARED symbol is defined. You can also check at runtime by passing the _SC_THREAD_PROCESS_SHARED parameter to the sysconf function. Although this option is not required to be provided by POSIX-conforming operating systems, the Single UNIX Specification requires that XSI-conforming operating systems do support it.

Within a process, multiple threads can access the same synchronization object. This is the default behavior, as we saw in Chapter 11. In this case, the *process-shared* mutex attribute is set to PTHREAD_PROCESS_PRIVATE.

As we shall see in Chapters 14 and 15, mechanisms exist that allow independent processes to map the same extent of memory into their independent address spaces. Access to shared data by multiple processes usually requires synchronization, just as does access to shared data by multiple threads. If the *process-shared* mutex attribute is set to PTHREAD_PROCESS_SHARED, a mutex allocated from a memory extent shared between multiple processes may be used for synchronization by those processes.

We can use the pthread_mutexattr_getpshared function to query a pthread_mutexattr_t structure for its *process-shared* attribute. We can change the *process-shared* attribute with the pthread_mutexattr_setpshared function.

```
#include <pthread.h>

int pthread_mutexattr_getpshared(const pthread_mutexattr_t *
                                 restrict attr,
                                 int *restrict pshared);

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
                                 int pshared);
```

Both return: 0 if OK, error number on failure

The *process-shared* mutex attribute allows the pthread library to provide more efficient mutex implementations when the attribute is set to PTHREAD_PROCESS_PRIVATE, which is the default case with multithreaded applications. The pthread library can then

restrict the more expensive implementation to the case in which mutexes are shared among processes.

The *robust* mutex attribute is related to mutexes that are shared among multiple processes. It is meant to address the problem of mutex state recovery when a process terminates while holding a mutex. When this happens, the mutex is left in a locked state and recovery is difficult. Threads blocked on the lock in other processes will block indefinitely.

We can use the `pthread_mutexattr_getrobust` function to get the value of the *robust* mutex attribute. To set the value of the *robust* mutex attribute, we can call the `pthread_mutexattr_setrobust` function.

```
#include <pthread.h>

int pthread_mutexattr_getrobust(const pthread_mutexattr_t *
                                restrict attr,
                                int *restrict robust);

int pthread_mutexattr_setrobust(pthread_mutexattr_t *attr,
                                int robust);
```

Both return: 0 if OK, error number on failure

There are two possible values for the *robust* attribute. The default is `PTHREAD_MUTEX_STALLED`, which means that no special action is taken when a process terminates while holding a mutex. In this case, use of the mutex can result in undefined behavior, and applications waiting for it to be unlocked are effectively "stalled." The other value is `PTHREAD_MUTEX_ROBUST`. This value will cause a thread blocked in a call to `pthread_mutex_lock` to acquire the lock when another process holding the lock terminates without first unlocking it, but the return value from `pthread_mutex_lock` is `EOWNERDEAD` instead of 0. Applications can use this special return value as an indication that they need to recover whatever state the mutex was protecting, if possible (the details of what state is being protected and how it can be recovered will vary among applications). Note that the `EOWNERDEAD` error return isn't really an error in this case, because the caller will own the lock.

Using robust mutexes changes the way we use `pthread_mutex_lock`, because we now have to check for three return values instead of two: success with no recovery needed, success but recovery needed, and failure. However, if we don't use robust mutexes, then we can continue to check only for success and failure.

> Of the four platforms covered in this text, only Linux 3.2.0 currently supports robust pthread mutexes. Solaris 10 supports robust mutexes only in its Solaris threads library (see the `mutex_init`(3C) Solaris manual page for more information). However, in Solaris 11, robust pthread mutexes are supported.

If the application state can't be recovered, the mutex will be in a permanently unusable state after the thread unlocks the mutex. To prevent this problem, the thread can call the `pthread_mutex_consistent` function to indicate that the state associated with the mutex is consistent before unlocking the mutex.

```
#include <pthread.h>

int pthread_mutex_consistent(pthread_mutex_t * mutex);
```

Returns: 0 if OK, error number on failure

If a thread unlocks a mutex without first calling `pthread_mutex_consistent`, then other threads that are blocked while trying to acquire the mutex will see error returns of `ENOTRECOVERABLE`. If this happens, the mutex is no longer usable. By calling `pthread_mutex_consistent` beforehand, a thread allows the mutex to behave normally, so it can continue to be used.

The *type* mutex attribute controls the locking characteristics of the mutex. POSIX.1 defines four types:

`PTHREAD_MUTEX_NORMAL` A standard mutex type that doesn't do any special error checking or deadlock detection.

`PTHREAD_MUTEX_ERRORCHECK` A mutex type that provides error checking.

`PTHREAD_MUTEX_RECURSIVE` A mutex type that allows the same thread to lock it multiple times without first unlocking it. A recursive mutex maintains a lock count and isn't released until it is unlocked the same number of times it is locked. Thus, if you lock a recursive mutex twice and then unlock it, the mutex remains locked until it is unlocked a second time.

`PTHREAD_MUTEX_DEFAULT` A mutex type providing default characteristics and behavior. Implementations are free to map it to one of the other mutex types. For example, Linux 3.2.0 maps this type to the normal mutex type, whereas FreeBSD 8.0 maps it to the error-checking type.

The behavior of the four types is summarized in Figure 12.5. The "Unlock when not owned" column refers to one thread unlocking a mutex that was locked by a different thread. The "Unlock when unlocked" column refers to what happens when a thread unlocks a mutex that is already unlocked, which usually is a coding mistake.

Mutex type	Relock without unlock?	Unlock when not owned?	Unlock when unlocked?
`PTHREAD_MUTEX_NORMAL`	deadlock	undefined	undefined
`PTHREAD_MUTEX_ERRORCHECK`	returns error	returns error	returns error
`PTHREAD_MUTEX_RECURSIVE`	allowed	returns error	returns error
`PTHREAD_MUTEX_DEFAULT`	undefined	undefined	undefined

Figure 12.5 Mutex type behavior

We can use the pthread_mutexattr_gettype function to get the mutex *type* attribute. To change the attribute, we can use the pthread_mutexattr_settype function.

```
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *
                              restrict attr, int *restrict type);

int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

Both return: 0 if OK, error number on failure

Recall from Section 11.6.6 that a mutex is used to protect the condition that is associated with a condition variable. Before blocking the thread, the pthread_cond_wait and the pthread_cond_timedwait functions release the mutex associated with the condition. This allows other threads to acquire the mutex, change the condition, release the mutex, and signal the condition variable. Since the mutex must be held to change the condition, it is not a good idea to use a recursive mutex. If a recursive mutex is locked multiple times and used in a call to pthread_cond_wait, the condition can never be satisfied, because the unlock done by pthread_cond_wait doesn't release the mutex.

Recursive mutexes are useful when you need to adapt existing single-threaded interfaces to a multithreaded environment, but can't change the interfaces to your functions because of compatibility constraints. However, using recursive locks can be tricky, and they should be used only when no other solution is possible.

Example

Figure 12.6 illustrates a situation in which a recursive mutex might seem to solve a concurrency problem. Assume that func1 and func2 are existing functions in a library whose interfaces can't be changed, because applications exist that call them and those applications can't be changed.

To keep the interfaces the same, we embed a mutex in the data structure whose address (x) is passed in as an argument. This is possible only if we have provided an allocator function for the structure, so the application doesn't know about its size (assuming we must increase its size when we add a mutex to it).

> This is also possible if we originally defined the structure with enough padding to allow us now to replace some pad fields with a mutex. Unfortunately, most programmers are unskilled at predicting the future, so this is not a common practice.

If both func1 and func2 must manipulate the structure and it is possible to access it from more than one thread at a time, then func1 and func2 must lock the mutex before manipulating the structure. If func1 must call func2, we will deadlock if the mutex type is not recursive. We could avoid using a recursive mutex if we could release

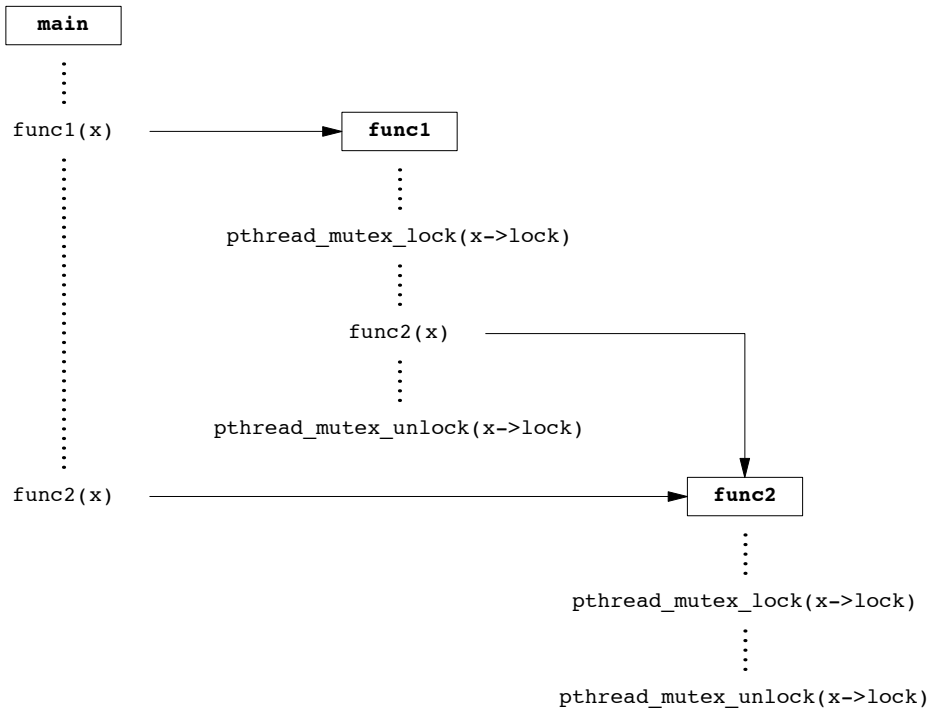


Figure 12.6 Recursive locking opportunity

the mutex before calling `func2` and reacquire it after `func2` returns, but this approach opens a window where another thread can possibly grab control of the mutex and change the data structure in the middle of `func1`. This may not be acceptable, depending on what protection the mutex is intended to provide.

Figure 12.7 shows an alternative to using a recursive mutex in this case. We can leave the interfaces to `func1` and `func2` unchanged and avoid a recursive mutex by providing a private version of `func2`, called `func2_locked`. To call `func2_locked`, we must hold the mutex embedded in the data structure whose address we pass as the argument. The body of `func2_locked` contains a copy of `func2`, and `func2` now simply acquires the mutex, calls `func2_locked`, and then releases the mutex.

If we didn't have to leave the interfaces to the library functions unchanged, we could have added a second parameter to each function to indicate whether the structure is locked by the caller. It is usually better to leave the interfaces unchanged if we can, however, instead of polluting them with implementation artifacts.

The strategy of providing locked and unlocked versions of functions is usually applicable in simple situations. In more complex situations, such as when the library needs to call a function outside the library, which then might call back into the library, we need to rely on recursive locks. □

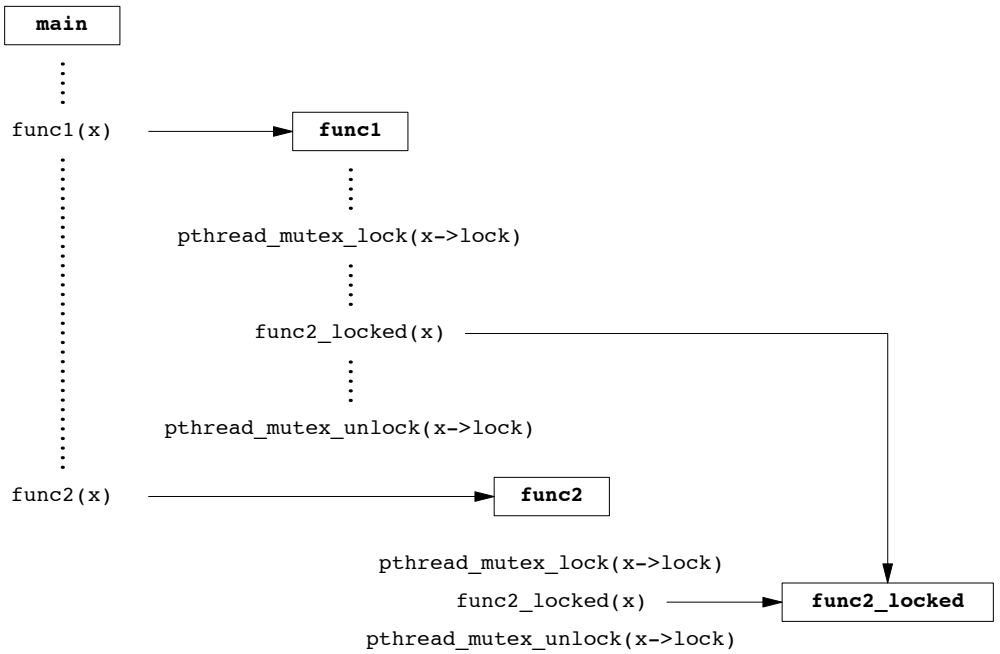


Figure 12.7 Avoiding a recursive locking opportunity

Example

The program in Figure 12.8 illustrates another situation in which a recursive mutex is necessary. Here, we have a "timeout" function that allows us to schedule another function to be run at some time in the future. Assuming that threads are an inexpensive resource, we can create a thread for each pending timeout. The thread waits until the time has been reached, and then it calls the function we've requested.

The problem arises when we can't create a thread or when the scheduled time to run the function has already passed. In these cases, we simply call the requested function now, from the current context. Since the function acquires the same lock that we currently hold, a deadlock will occur unless the lock is recursive.

```
#include "apue.h"
#include <pthread.h>
#include <time.h>
#include <sys/time.h>

extern int makethread(void *(*)(void *), void *);

struct to_info {
    void          (*to_fn)(void *); /* function */
```

```
    void           *to_arg;         /* argument */
    struct timespec to_wait;        /* time to wait */
};

#define SECTONSEC  1000000000   /* seconds to nanoseconds */

#if !defined(CLOCK_REALTIME) || defined(BSD)
#define clock_nanosleep(ID, FL, REQ, REM)   nanosleep((REQ), (REM))
#endif

#ifndef CLOCK_REALTIME
#define CLOCK_REALTIME 0
#define USECTONSEC 1000     /* microseconds to nanoseconds */

void
clock_gettime(int id, struct timespec *tsp)
{
    struct timeval tv;

    gettimeofday(&tv, NULL);
    tsp->tv_sec = tv.tv_sec;
    tsp->tv_nsec = tv.tv_usec * USECTONSEC;
}
#endif

void *
timeout_helper(void *arg)
{
    struct to_info  *tip;

    tip = (struct to_info *)arg;
    clock_nanosleep(CLOCK_REALTIME, 0, &tip->to_wait, NULL);
    (*tip->to_fn)(tip->to_arg);
    free(arg);
    return(0);
}

void
timeout(const struct timespec *when, void (*func)(void *), void *arg)
{
    struct timespec now;
    struct to_info  *tip;
    int             err;

    clock_gettime(CLOCK_REALTIME, &now);
    if ((when->tv_sec > now.tv_sec) ||
      (when->tv_sec == now.tv_sec && when->tv_nsec > now.tv_nsec)) {
        tip = malloc(sizeof(struct to_info));
        if (tip != NULL) {
            tip->to_fn = func;
            tip->to_arg = arg;
            tip->to_wait.tv_sec = when->tv_sec - now.tv_sec;
            if (when->tv_nsec >= now.tv_nsec) {
                tip->to_wait.tv_nsec = when->tv_nsec - now.tv_nsec;
            } else {
```

```
                tip->to_wait.tv_sec--;
                tip->to_wait.tv_nsec = SECTONSEC - now.tv_nsec +
                  when->tv_nsec;
            }
            err = makethread(timeout_helper, (void *)tip);
            if (err == 0)
                return;
            else
                free(tip);
        }
    }

    /*
     * We get here if (a) when <= now, or (b) malloc fails, or
     * (c) we can't make a thread, so we just call the function now.
     */
    (*func)(arg);
}

pthread_mutexattr_t attr;
pthread_mutex_t mutex;

void
retry(void *arg)
{
    pthread_mutex_lock(&mutex);

    /* perform retry steps ... */

    pthread_mutex_unlock(&mutex);
}

int
main(void)
{
    int             err, condition, arg;
    struct timespec when;

    if ((err = pthread_mutexattr_init(&attr)) != 0)
        err_exit(err, "pthread_mutexattr_init failed");
    if ((err = pthread_mutexattr_settype(&attr,
      PTHREAD_MUTEX_RECURSIVE)) != 0)
        err_exit(err, "can't set recursive type");
    if ((err = pthread_mutex_init(&mutex, &attr)) != 0)
        err_exit(err, "can't create recursive mutex");

    /* continue processing ... */

    pthread_mutex_lock(&mutex);

    /*
     * Check the condition under the protection of a lock to
     * make the check and the call to timeout atomic.
     */
    if (condition) {
        /*
```

```
         * Calculate the absolute time when we want to retry.
         */
        clock_gettime(CLOCK_REALTIME, &when);
        when.tv_sec += 10;  /* 10 seconds from now */
        timeout(&when, retry, (void *)((unsigned long)arg));
    }
    pthread_mutex_unlock(&mutex);

    /* continue processing ... */

    exit(0);
}
```

Figure 12.8 Using a recursive mutex

We use the `makethread` function from Figure 12.4 to create a thread in the detached state. Because the `func` function argument passed to the `timeout` function will run in the future, we don't want to wait around for the thread to complete.

We could call `sleep` to wait for the timeout to expire, but that gives us only second granularity. If we want to wait for some time other than an integral number of seconds, we need to use `nanosleep` or `clock_nanosleep`, both of which allow us to sleep at higher resolution.

> On systems that don't define `CLOCK_REALTIME`, we define `clock_nanosleep` in terms of `nanosleep`. However, FreeBSD 8.0 defines this symbol to support `clock_gettime` and `clock_settime`, but doesn't support `clock_nanosleep` (only Linux 3.2.0 and Solaris 10 currently support `clock_nanosleep`.)
>
> Additionally, on systems that don't define `CLOCK_REALTIME`, we provide our own implementation of `clock_gettime` that calls `gettimeofday` and translates microseconds to nanoseconds.

The caller of `timeout` needs to hold a mutex to check the condition and to schedule the `retry` function as an atomic operation. The `retry` function will try to lock the same mutex. Unless the mutex is recursive, a deadlock will occur if the `timeout` function calls `retry` directly. □

12.4.2 Reader–Writer Lock Attributes

Reader–writer locks also have attributes, similar to mutexes. We use `pthread_rwlockattr_init` to initialize a `pthread_rwlockattr_t` structure and `pthread_rwlockattr_destroy` to deinitialize the structure.

```
#include <pthread.h>

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

Both return: 0 if OK, error number on failure

The only attribute supported for reader–writer locks is the *process-shared* attribute. It is identical to the mutex *process-shared* attribute. Just as with the mutex *process-shared* attributes, a pair of functions is provided to get and set the *process-shared* attributes of reader–writer locks.

```
#include <pthread.h>

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *
                                  restrict attr,
                                  int *restrict pshared);

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
                                  int pshared);
```

Both return: 0 if OK, error number on failure

Although POSIX defines only one reader–writer lock attribute, implementations are free to define additional, nonstandard ones.

12.4.3 Condition Variable Attributes

The Single UNIX Specification currently defines two attributes for condition variables: the *process-shared* attribute and the *clock* attribute. As with the other attribute objects, a pair of functions initialize and deinitialize condition variable attribute objects.

```
#include <pthread.h>

int pthread_condattr_init(pthread_condattr_t *attr);

int pthread_condattr_destroy(pthread_condattr_t *attr);
```

Both return: 0 if OK, error number on failure

The *process-shared* attribute is the same as with the other synchronization attributes. It controls whether condition variables can be used by threads within a single process only or from within multiple processes. To find the current value of the *process-shared* attribute, we use the `pthread_condattr_getpshared` function. To set its value, we use the `pthread_condattr_setpshared` function.

```
#include <pthread.h>

int pthread_condattr_getpshared(const pthread_condattr_t *
                                restrict attr,
                                int *restrict pshared);

int pthread_condattr_setpshared(pthread_condattr_t *attr,
                                int pshared);
```

Both return: 0 if OK, error number on failure

The *clock* attribute controls which clock is used when evaluating the timeout argument (*tsptr*) of the `pthread_cond_timedwait` function. The legal values are the

clock IDs listed in Figure 6.8. We can use the pthread_condattr_getclock function to retrieve the clock ID that will be used by the pthread_cond_timedwait function for the condition variable that was initialized with the pthread_condattr_t object. We can change the clock ID with the pthread_condattr_setclock function.

```
#include <pthread.h>

int pthread_condattr_getclock(const pthread_condattr_t *
                              restrict attr,
                              clockid_t *restrict clock_id);

int pthread_condattr_setclock(pthread_condattr_t *attr,
                              clockid_t clock_id);
```

Both return: 0 if OK, error number on failure

Curiously, the Single UNIX Specification doesn't define the *clock* attribute for any of the other attribute objects that have a wait function with a timeout.

12.4.4 Barrier Attributes

Barriers have attributes, too. We can use the pthread_barrierattr_init function to initialize a barrier attributes object and the pthread_barrierattr_destroy function to deinitialize a barrier attributes object.

```
#include <pthread.h>

int pthread_barrierattr_init(pthread_barrierattr_t *attr);

int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
```

Both return: 0 if OK, error number on failure

The only barrier attribute currently defined is the *process-shared* attribute, which controls whether a barrier can be used by threads from multiple processes or only from within the process that initialized the barrier. As with the other attribute objects, we have one function to get the attribute (pthread_barrierattr_getpshared) value and one function to set the value (pthread_barrierattr_setpshared).

```
#include <pthread.h>

int pthread_barrierattr_getpshared(const pthread_barrierattr_t *
                              restrict attr,
                              int *restrict pshared);

int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
                              int pshared);
```

Both return: 0 if OK, error number on failure

The value of the *process-shared* attribute can be either `PTHREAD_PROCESS_SHARED` (accessible to threads from multiple processes) or `PTHREAD_PROCESS_PRIVATE` (accessible to only threads in the process that initialized the barrier).

12.5 Reentrancy

We discussed reentrant functions and signal handlers in Section 10.6. Threads are similar to signal handlers when it comes to reentrancy. In both cases, multiple threads of control can potentially call the same function at the same time.

basename	getchar_unlocked	getservent	putc_unlocked
catgets	getdate	getutxent	putchar_unlocked
crypt	getenv	getutxid	putenv
dbm_clearerr	getgrent	getutxline	pututxline
dbm_close	getgrgid	gmtime	rand
dbm_delete	getgrnam	hcreate	readdir
dbm_error	gethostent	hdestroy	setenv
dbm_fetch	getlogin	hsearch	setgrent
dbm_firstkey	getnetbyaddr	inet_ntoa	setkey
dbm_nextkey	getnetbyname	l64a	setpwent
dbm_open	getnetent	lgamma	setutxent
dbm_store	getopt	lgammaf	strerror
dirname	getprotobyname	lgammal	strsignal
dlerror	getprotobynumber	localeconv	strtok
drand48	getprotoent	localtime	system
encrypt	getpwent	lrand48	ttyname
endgrent	getpwnam	mrand48	unsetenv
endpwent	getpwuid	nftw	wcstombs
endutxent	getservbyname	nl_langinfo	wctomb
getc_unlocked	getservbyport	ptsname	

Figure 12.9 Functions *not* guaranteed to be thread-safe by POSIX.1

If a function can be safely called by multiple threads at the same time, we say that the function is *thread-safe*. All functions defined in the Single UNIX Specification are guaranteed to be thread-safe, except those listed in Figure 12.9. In addition, the `ctermid` and `tmpnam` functions are not guaranteed to be thread-safe if they are passed a null pointer. Similarly, there is no guarantee that `wcrtomb` and `wcsrtombs` are thread-safe when they are passed a null pointer for their `mbstate_t` argument.

Implementations that support thread-safe functions will define the `_POSIX_THREAD_SAFE_FUNCTIONS` symbol in `<unistd.h>`. Applications can also use the `_SC_THREAD_SAFE_FUNCTIONS` argument with `sysconf` to check for support of thread-safe functions at runtime. Prior to Version 4 of the Single UNIX Specification, all XSI-conforming implementations were required to support thread-safe functions. With SUSv4, however, thread-safe function support is now required for an implementation to be considered POSIX conforming.

With thread-safe functions, implementations provide alternative, thread-safe versions of some of the POSIX.1 functions that aren't thread-safe. Figure 12.10 lists the

thread-safe versions of these functions. The functions have the same names as their non-thread-safe relatives, but with an `_r` appended at the end of the name, signifying that these versions are reentrant. Many functions are not thread-safe, because they return data stored in a static memory buffer. They are made thread-safe by changing their interfaces to require that the caller provide its own buffer.

`getgrgid_r`	`localtime_r`
`getgrnam_r`	`readdir_r`
`getlogin_r`	`strerror_r`
`getpwnam_r`	`strtok_r`
`getpwuid_r`	`ttyname_r`
`gmtime_r`	

Figure 12.10 Alternative thread-safe functions

If a function is reentrant with respect to multiple threads, we say that it is thread-safe. This doesn't tell us, however, whether the function is reentrant with respect to signal handlers. We say that a function that is safe to be reentered from an asynchronous signal handler is *async-signal safe*. We saw the async-signal safe functions in Figure 10.4 when we discussed reentrant functions in Section 10.6.

In addition to the functions listed in Figure 12.10, POSIX.1 provides a way to manage `FILE` objects in a thread-safe way. You can use `flockfile` and `ftrylockfile` to obtain a lock associated with a given `FILE` object. This lock is recursive: you can acquire it again, while you already hold it, without deadlocking. Although the exact implementation of the lock is unspecified, all standard I/O routines that manipulate `FILE` objects are required to behave as if they call `flockfile` and `funlockfile` internally.

```
#include <stdio.h>

int ftrylockfile(FILE *fp);
```

Returns: 0 if OK, nonzero if lock can't be acquired

```
void flockfile(FILE *fp);

void funlockfile(FILE *fp);
```

Although the standard I/O routines might be implemented to be thread-safe from the perspective of their own internal data structures, it is still useful to expose the locking to applications. This allows applications to compose multiple calls to standard I/O functions into atomic sequences. Of course, when dealing with multiple `FILE` objects, you need to beware of potential deadlocks and to order your locks carefully.

If the standard I/O routines acquire their own locks, then we can run into serious performance degradation when doing character-at-a-time I/O. In this situation, we end up acquiring and releasing a lock for every character read or written. To avoid this overhead, unlocked versions of the character-based standard I/O routines are available.

```
#include <stdio.h>

int getchar_unlocked(void);

int getc_unlocked(FILE *fp);
```

Both return: the next character if OK, EOF on end of file or error

```
int putchar_unlocked(int c);

int putc_unlocked(int c, FILE *fp);
```

Both return: *c* if OK, EOF on error

These four functions should not be called unless they are surrounded by calls to `flockfile` (or `ftrylockfile`) and `funlockfile`. Otherwise, unpredictable results can occur (i.e., the types of problems that result from unsynchronized access to data by multiple threads of control).

Once you lock the `FILE` object, you can make multiple calls to these functions before releasing the lock. This amortizes the locking overhead across the amount of data read or written.

Example

Figure 12.11 shows a possible implementation of `getenv` (Section 7.9). This version is not reentrant. If two threads call it at the same time, they will see inconsistent results, because the string returned is stored in a single static buffer that is shared by all threads calling `getenv`.

```
#include <limits.h>
#include <string.h>

#define MAXSTRINGSZ 4096

static char envbuf[MAXSTRINGSZ];

extern char **environ;

char *
getenv(const char *name)
{
    int i, len;

    len = strlen(name);
    for (i = 0; environ[i] != NULL; i++) {
        if ((strncmp(name, environ[i], len) == 0) &&
          (environ[i][len] == '=')) {
            strncpy(envbuf, &environ[i][len+1], MAXSTRINGSZ-1);
            return(envbuf);
        }
    }
    return(NULL);
}
```

Figure 12.11 A nonreentrant version of `getenv`

We show a reentrant version of `getenv` in Figure 12.12. This version is called `getenv_r`. It uses the `pthread_once` function to ensure that the `thread_init` function is called only once per process, regardless of how many threads might race to call `getenv_r` at the same time. We'll have more to say about the `pthread_once` function in Section 12.6.

```
#include <string.h>
#include <errno.h>
#include <pthread.h>
#include <stdlib.h>

extern char **environ;

pthread_mutex_t env_mutex;

static pthread_once_t init_done = PTHREAD_ONCE_INIT;

static void
thread_init(void)
{
    pthread_mutexattr_t attr;

    pthread_mutexattr_init(&attr);
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&env_mutex, &attr);
    pthread_mutexattr_destroy(&attr);
}

int
getenv_r(const char *name, char *buf, int buflen)
{
    int i, len, olen;

    pthread_once(&init_done, thread_init);
    len = strlen(name);
    pthread_mutex_lock(&env_mutex);
    for (i = 0; environ[i] != NULL; i++) {
        if ((strncmp(name, environ[i], len) == 0) &&
          (environ[i][len] == '=')) {
            olen = strlen(&environ[i][len+1]);
            if (olen >= buflen) {
                pthread_mutex_unlock(&env_mutex);
                return(ENOSPC);
            }
            strcpy(buf, &environ[i][len+1]);
            pthread_mutex_unlock(&env_mutex);
            return(0);
        }
    }
    pthread_mutex_unlock(&env_mutex);
    return(ENOENT);
}
```

Figure 12.12 A reentrant (thread-safe) version of `getenv`

To make `getenv_r` reentrant, we changed the interface so that the caller must provide its own buffer. Thus each thread can use a different buffer to avoid interfering with the others. Note, however, that this is not enough to make `getenv_r` thread-safe. To make `getenv_r` thread-safe, we need to protect against changes to the environment while we are searching for the requested string. We can use a mutex to serialize access to the environment list by `getenv_r` and `putenv`.

We could have used a reader–writer lock to allow multiple concurrent calls to `getenv_r`, but the added concurrency probably wouldn't improve the performance of our program by very much, for two reasons. First, the environment list usually isn't very long, so we won't hold the mutex for too long while we scan the list. Second, calls to `getenv` and `putenv` are infrequent, so if we improve their performance, we won't affect the overall performance of the program very much.

Even though we can make `getenv_r` thread-safe, that doesn't mean that it is reentrant with respect to signal handlers. If we were to use a nonrecursive mutex, we would run the risk that a thread would deadlock itself if it called `getenv_r` from a signal handler. If the signal handler interrupted the thread while it was executing `getenv_r`, we would already be holding `env_mutex` locked, so another attempt to lock it would block, causing the thread to deadlock. Thus we must use a recursive mutex to prevent other threads from changing the data structures while we look at them and to prevent deadlocks from signal handlers. The problem is that the pthread functions are not guaranteed to be async-signal safe, so we can't use them to make another function async-signal safe. □

12.6 Thread-Specific Data

Thread-specific data, also known as thread-private data, is a mechanism for storing and finding data associated with a particular thread. The reason we call the data thread-specific, or thread-private, is that we'd like each thread to access its own separate copy of the data, without worrying about synchronizing access with other threads.

Many people went to a lot of trouble designing a threads model that promotes sharing process data and attributes. So why would anyone want to promote interfaces that prevent sharing in this model? There are two reasons.

First, sometimes we need to maintain data on a per-thread basis. Since there is no guarantee that thread IDs are small, sequential integers, we can't simply allocate an array of per-thread data and use the thread ID as the index. Even if we could depend on small, sequential thread IDs, we'd like a little extra protection so that one thread can't mess with another's data.

The second reason for thread-private data is to provide a mechanism for adapting process-based interfaces to a multithreaded environment. An obvious example of this is `errno`. Recall the discussion of `errno` in Section 1.7. Older interfaces (before the advent of threads) defined `errno` as an integer that is accessible globally within the context of a process. System calls and library routines set `errno` as a side effect of failing. To make it possible for threads to use these same system calls and library

routines, errno is redefined as thread-private data. Thus one thread making a call that sets errno doesn't affect the value of errno for the other threads in the process.

Recall that all threads in a process have access to the entire address space of the process. Other than using registers, there is no way for one thread to prevent another from accessing its data. This is true even for thread-specific data. Even though the underlying implementation doesn't prevent access, the functions provided to manage thread-specific data promote data separation among threads by making it more difficult for threads to gain access to thread-specific data from other threads.

Before allocating thread-specific data, we need to create a *key* to associate with the data. The key will be used to gain access to the thread-specific data. We use pthread_key_create to create such a key.

```
#include <pthread.h>

int pthread_key_create(pthread_key_t *keyp, void (*destructor)(void *));
```

Returns: 0 if OK, error number on failure

The key created is stored in the memory location pointed to by *keyp*. The same key can be used by all threads in the process, but each thread will associate a different thread-specific data address with the key. When the key is created, the data address for each thread is set to a null value.

In addition to creating a key, pthread_key_create associates an optional destructor function with the key. When the thread exits, if the data address has been set to a non-null value, the destructor function is called with the data address as the only argument. If *destructor* is null, then no destructor function is associated with the key. When the thread exits normally, either by calling pthread_exit or by returning, the destructor is called. Also, if the thread is canceled, the destructor is called, but only after the last cleanup handler returns. But if the thread calls exit, _exit, _Exit, or abort, or otherwise exits abnormally, the destructor is not called.

Threads typically use malloc to allocate memory for their thread-specific data. The destructor function usually frees the memory that was allocated. If the thread exited without freeing the memory, then the memory would be lost—leaked by the process.

A thread can allocate multiple keys for thread-specific data. Each key can have a destructor associated with it. There can be a different destructor function for each key, or all of the keys can use the same function. Each operating system implementation can place a limit on the number of keys a process can allocate (recall PTHREAD_KEYS_MAX from Figure 12.1).

When a thread exits, the destructors for its thread-specific data are called in an implementation-defined order. It is possible for the destructor to call another function that creates new thread-specific data and associate it with the key. After all destructors are called, the system will check whether any non-null thread-specific values were associated with the keys and, if so, call the destructors again. This process repeats until either all keys for the thread have null thread-specific data values or a maximum of PTHREAD_DESTRUCTOR_ITERATIONS (Figure 12.1) attempts have been made.

We can break the association of a key with the thread-specific data values for all threads by calling pthread_key_delete.

```
#include <pthread.h>

int pthread_key_delete(pthread_key_t key);
```

Returns: 0 if OK, error number on failure

Note that calling `pthread_key_delete` will not invoke the destructor function associated with the key. To free any memory associated with the key's thread-specific data values, we need to take additional steps in the application.

We need to ensure that a key we allocate doesn't change because of a race during initialization. Code like the following can result in two threads both calling `pthread_key_create`:

```
void destructor(void *);

pthread_key_t key;
int init_done = 0;

int
threadfunc(void *arg)
{
     if (!init_done) {
          init_done = 1;
          err = pthread_key_create(&key, destructor);
     }
     .
     .
     .
}
```

Depending on how the system schedules threads, some threads might see one key value, whereas other threads might see a different value. The way to solve this race is to use `pthread_once`.

```
#include <pthread.h>

pthread_once_t initflag = PTHREAD_ONCE_INIT;

int pthread_once(pthread_once_t *initflag, void (*initfn)(void));
```

Returns: 0 if OK, error number on failure

The *initflag* must be a nonlocal variable (i.e., global or static) and initialized to `PTHREAD_ONCE_INIT`.

If each thread calls `pthread_once`, the system guarantees that the initialization routine, *initfn*, will be called only once, on the first call to `pthread_once`. The proper way to create a key without a race is as follows:

```
void destructor(void *);

pthread_key_t key;
pthread_once_t init_done = PTHREAD_ONCE_INIT;
```

```
void
thread_init(void)
{
     err = pthread_key_create(&key, destructor);
}

int
threadfunc(void *arg)
{
     pthread_once(&init_done, thread_init);
     ⋮
}
```

Once a key is created, we can associate thread-specific data with the key by calling `pthread_setspecific`. We can obtain the address of the thread-specific data with `pthread_getspecific`.

```
#include <pthread.h>

void *pthread_getspecific(pthread_key_t key);
```

Returns: thread-specific data value or `NULL` if no value has been associated with the key

```
int pthread_setspecific(pthread_key_t key, const void *value);
```

Returns: 0 if OK, error number on failure

If no thread-specific data has been associated with a key, `pthread_getspecific` will return a null pointer. We can use this return value to determine whether we need to call `pthread_setspecific`.

Example

In Figure 12.11, we showed a hypothetical implementation of `getenv`. We came up with a new interface to provide the same functionality, but in a thread-safe way (Figure 12.12). But what would happen if we couldn't modify our application programs to use the new interface? In that case, we could use thread-specific data to maintain a per-thread copy of the data buffer used to hold the return string. This is shown in Figure 12.13.

```
#include <limits.h>
#include <string.h>
#include <pthread.h>
#include <stdlib.h>

#define MAXSTRINGSZ 4096

static pthread_key_t key;
static pthread_once_t init_done = PTHREAD_ONCE_INIT;
pthread_mutex_t env_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
extern char **environ;

static void
thread_init(void)
{
    pthread_key_create(&key, free);
}

char *
getenv(const char *name)
{
    int     i, len;
    char    *envbuf;

    pthread_once(&init_done, thread_init);
    pthread_mutex_lock(&env_mutex);
    envbuf = (char *)pthread_getspecific(key);
    if (envbuf == NULL) {
        envbuf = malloc(MAXSTRINGSZ);
        if (envbuf == NULL) {
            pthread_mutex_unlock(&env_mutex);
            return(NULL);
        }
        pthread_setspecific(key, envbuf);
    }
    len = strlen(name);
    for (i = 0; environ[i] != NULL; i++) {
        if ((strncmp(name, environ[i], len) == 0) &&
          (environ[i][len] == '=')) {
            strncpy(envbuf, &environ[i][len+1], MAXSTRINGSZ-1);
            pthread_mutex_unlock(&env_mutex);
            return(envbuf);
        }
    }
    pthread_mutex_unlock(&env_mutex);
    return(NULL);
}
```

Figure 12.13 A thread-safe, compatible version of `getenv`

We use `pthread_once` to ensure that only one key is created for the thread-specific data we will use. If `pthread_getspecific` returns a null pointer, we need to allocate the memory buffer and associate it with the key. Otherwise, we use the memory buffer returned by `pthread_getspecific`. For the destructor function, we use `free` to free the memory previously allocated by `malloc`. The destructor function will be called with the value of the thread-specific data only if the value is non-null.

Note that although this version of `getenv` is thread-safe, it is not async-signal safe. Even if we made the mutex recursive, we could not make it reentrant with respect to signal handlers because it calls `malloc`, which itself is not async-signal safe. □

12.7 Cancel Options

Two thread attributes that are not included in the `pthread_attr_t` structure are the *cancelability state* and the *cancelability type*. These attributes affect the behavior of a thread in response to a call to `pthread_cancel` (Section 11.5).

The *cancelability state* attribute can be either `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`. A thread can change its *cancelability state* by calling `pthread_setcancelstate`.

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
```

Returns: 0 if OK, error number on failure

In one atomic operation, `pthread_setcancelstate` sets the current *cancelability state* to *state* and stores the previous *cancelability state* in the memory location pointed to by *oldstate*.

Recall from Section 11.5 that a call to `pthread_cancel` doesn't wait for a thread to terminate. In the default case, a thread will continue to execute after a cancellation request is made until the thread reaches a *cancellation point*. A cancellation point is a place where the thread checks whether it has been canceled, and if so, acts on the request. POSIX.1 guarantees that cancellation points will occur when a thread calls any of the functions listed in Figure 12.14.

accept	mq_timedsend	pthread_join	sendto
aio_suspend	msgrcv	pthread_testcancel	sigsuspend
clock_nanosleep	msgsnd	pwrite	sigtimedwait
close	msync	read	sigwait
connect	nanosleep	readv	sigwaitinfo
creat	open	recv	sleep
fcntl	openat	recvfrom	system
fdatasync	pause	recvmsg	tcdrain
fsync	poll	select	wait
lockf	pread	sem_timedwait	waitid
mq_receive	pselect	sem_wait	waitpid
mq_send	pthread_cond_timedwait	send	write
mq_timedreceive	pthread_cond_wait	sendmsg	writev

Figure 12.14 Cancellation points defined by POSIX.1

A thread starts with a default *cancelability state* of `PTHREAD_CANCEL_ENABLE`. When the state is set to `PTHREAD_CANCEL_DISABLE`, a call to `pthread_cancel` will not kill the thread. Instead, the cancellation request remains pending for the thread. When the state is enabled again, the thread will act on any pending cancellation requests at the next cancellation point.

In addition to the functions listed in Figure 12.14, POSIX.1 specifies the functions listed in Figure 12.15 as optional cancellation points.

access	fseeko	getwchar	putwc
catclose	fsetpos	glob	putwchar
catgets	fstat	iconv_close	readdir
catopen	fstatat	iconv_open	readdir_r
chmod	ftell	ioctl	readlink
chown	ftello	link	readlinkat
closedir	futimens	linkat	remove
closelog	fwprintf	lio_listio	rename
ctermid	fwrite	localtime	renameat
dbm_close	fwscanf	localtime_r	rewind
dbm_delete	getaddrinfo	lockf	rewinddir
dbm_fetch	getc	lseek	scandir
dbm_nextkey	getc_unlocked	lstat	scanf
dbm_open	getchar	mkdir	seekdir
dbm_store	getchar_unlocked	mkdirat	semop
dlclose	getcwd	mkdtemp	setgrent
dlopen	getdate	mkfifo	sethostent
dprintf	getdelim	mkfifoat	setnetent
endgrent	getgrent	mknod	setprotoent
endhostent	getgrgid	mknodat	setpwent
endnetent	getgrgid_r	mkstemp	setservent
endprotoent	getgrnam	mktime	setutxent
endpwent	getgrnam_r	nftw	stat
endservent	gethostent	opendir	strerror
endutxent	gethostid	openlog	strerror_r
faccessat	gethostname	pathconf	strftime
fchmod	getline	pclose	symlink
fchmodat	getlogin	perror	symlinkat
fchown	getlogin_r	popen	sync
fchownat	getnameinfo	posix_fadvise	syslog
fclose	getnetbyaddr	posix_fallocate	tmpfile
fcntl	getnetbyname	posix_madvise	ttyname
fflush	getnetent	posix_openpt	ttyname_r
fgetc	getopt	posix_spawn	tzset
fgetpos	getprotobyname	posix_spawnp	ungetc
fgets	getprotobynumber	posix_typed_mem_open	ungetwc
fgetwc	getprotoent	printf	unlink
fgetws	getpwent	psiginfo	unlinkat
fmtmsg	getpwnam	psignal	utimensat
fopen	getpwnam_r	pthread_rwlock_rdlock	utimes
fpathconf	getpwuid	pthread_rwlock_timedrdlock	vdprintf
fprintf	getpwuid_r	pthread_rwlock_timedwrlock	vfprintf
fputc	getservbyname	pthread_rwlock_wrlock	vfwprintf
fputs	getservbyport	putc	vprintf
fputwc	getservent	putc_unlocked	vwprintf
fputws	getutxent	putchar	wcsftime
fread	getutxid	putchar_unlocked	wordexp
freopen	getutxline	puts	wprintf
fscanf	getwc	pututxline	wscanf
fseek			

Figure 12.15 Optional cancellation points defined by POSIX.1

Several of the functions listed in Figure 12.15, such as the ones dealing with message catalogs and wide character sets, are not discussed further in this text.

If your application doesn't call one of the functions in Figure 12.14 or Figure 12.15 for a long period of time (if it is compute bound, for example), then you can call `pthread_testcancel` to add your own cancellation points to the program.

```
#include <pthread.h>

void pthread_testcancel(void);
```

When you call `pthread_testcancel`, if a cancellation request is pending and if cancellation has not been disabled, the thread will be canceled. When cancellation is disabled, however, calls to `pthread_testcancel` have no effect.

The default cancellation type we have been describing is known as *deferred cancellation*. After a call to `pthread_cancel`, the actual cancellation doesn't occur until the thread hits a cancellation point. We can change the cancellation type by calling `pthread_setcanceltype`.

```
#include <pthread.h>

int pthread_setcanceltype(int type, int *oldtype);
```

Returns: 0 if OK, error number on failure

The `pthread_setcanceltype` function sets the cancellation type to *type* (either `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`) and returns the previous type in the integer pointed to by *oldtype*.

Asynchronous cancellation differs from deferred cancellation in that the thread can be canceled at any time. The thread doesn't necessarily need to hit a cancellation point for it to be canceled.

12.8 Threads and Signals

Dealing with signals can be complicated even with a process-based paradigm. Introducing threads into the picture makes things even more complicated.

Each thread has its own signal mask, but the signal disposition is shared by all threads in the process. As a consequence, individual threads can block signals, but when a thread modifies the action associated with a given signal, all threads share the action. Thus, if one thread chooses to ignore a given signal, another thread can undo that choice by restoring the default disposition or installing a signal handler for that signal.

Signals are delivered to a single thread in the process. If the signal is related to a hardware fault, the signal is usually sent to the thread whose action caused the event. Other signals, on the other hand, are delivered to an arbitrary thread.

In Section 10.12, we discussed how processes can use the `sigprocmask` function to block signals from delivery. However, the behavior of `sigprocmask` is undefined in a multithreaded process. Threads have to use the `pthread_sigmask` function instead.

```
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *restrict set,
                    sigset_t *restrict oset);
```

Returns: 0 if OK, error number on failure

The `pthread_sigmask` function is identical to `sigprocmask`, except that `pthread_sigmask` works with threads and returns an error code on failure instead of setting `errno` and returning –1. Recall that the *set* argument contains the set of signals that the thread will use to modify its signal mask. The *how* argument can take on one of three values: `SIG_BLOCK` to add the set of signals to the thread's signal mask, `SIG_SETMASK` to replace the thread's signal mask with the set of signals, or `SIG_UNBLOCK` to remove the set of signals from the thread's signal mask. If the *oset* argument is not null, the thread's previous signal mask is stored in the `sigset_t` structure to which it points. A thread can get its current signal mask by setting the *set* argument to `NULL` and setting the *oset* argument to the address of a `sigset_t` structure. In this case, the *how* argument is ignored.

A thread can wait for one or more signals to occur by calling `sigwait`.

```
#include <signal.h>

int sigwait(const sigset_t *restrict set, int *restrict signop);
```

Returns: 0 if OK, error number on failure

The *set* argument specifies the set of signals for which the thread is waiting. On return, the integer to which *signop* points will contain the number of the signal that was delivered.

If one of the signals specified in the set is pending at the time `sigwait` is called, then `sigwait` will return without blocking. Before returning, `sigwait` removes the signal from the set of signals pending for the process. If the implementation supports queued signals, and multiple instances of a signal are pending, `sigwait` will remove only one instance of the signal; the other instances will remain queued.

To avoid erroneous behavior, a thread must block the signals it is waiting for before calling `sigwait`. The `sigwait` function will atomically unblock the signals and wait until one is delivered. Before returning, `sigwait` will restore the thread's signal mask. If the signals are not blocked at the time that `sigwait` is called, then a timing window is opened up where one of the signals can be delivered to the thread before it completes its call to `sigwait`.

The advantage to using `sigwait` is that it can simplify signal handling by allowing us to treat asynchronously generated signals in a synchronous manner. We can prevent the signals from interrupting the threads by adding them to each thread's signal mask. Then we can dedicate specific threads to handling the signals. These dedicated threads can make function calls without having to worry about which functions are safe to call from a signal handler, because they are being called from normal thread context, not from a traditional signal handler interrupting a normal thread's execution.

If multiple threads are blocked in calls to `sigwait` for the same signal, only one of the threads will return from `sigwait` when the signal is delivered. If a signal is being

caught (the process has established a signal handler by using `sigaction`, for example) and a thread is waiting for the same signal in a call to `sigwait`, it is left up to the implementation to decide which way to deliver the signal. The implementation could either allow `sigwait` to return or invoke the signal handler, but not both.

To send a signal to a process, we call `kill` (Section 10.9). To send a signal to a thread, we call `pthread_kill`.

```
#include <signal.h>

int pthread_kill(pthread_t thread, int signo);
```

Returns: 0 if OK, error number on failure

We can pass a *signo* value of 0 to check for existence of the thread. If the default action for a signal is to terminate the process, then sending the signal to a thread will still kill the entire process.

Note that alarm timers are a process resource, and all threads share the same set of alarms. Thus, it is not possible for multiple threads in a process to use alarm timers without interfering (or cooperating) with one another (this is the subject of Exercise 12.6).

Example

Recall that in Figure 10.23, we waited for the signal handler to set a flag indicating that the main program should exit. The only threads of control that could run were the main thread and the signal handler, so blocking the signals was sufficient to avoid missing a change to the flag. With threads, we need to use a mutex to protect the flag, as we show in Figure 12.16.

```
#include "apue.h"
#include <pthread.h>

int         quitflag;   /* set nonzero by thread */
sigset_t    mask;

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t waitloc = PTHREAD_COND_INITIALIZER;

void *
thr_fn(void *arg)
{
    int err, signo;

    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0)
            err_exit(err, "sigwait failed");
        switch (signo) {
        case SIGINT:
            printf("\ninterrupt\n");
            break;
```

```
        case SIGQUIT:
            pthread_mutex_lock(&lock);
            quitflag = 1;
            pthread_mutex_unlock(&lock);
            pthread_cond_signal(&waitloc);
            return(0);

        default:
            printf("unexpected signal %d\n", signo);
            exit(1);
        }
    }
}

int
main(void)
{
    int         err;
    sigset_t    oldmask;
    pthread_t   tid;

    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGQUIT);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, &oldmask)) != 0)
        err_exit(err, "SIG_BLOCK error");

    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "can't create thread");

    pthread_mutex_lock(&lock);
    while (quitflag == 0)
        pthread_cond_wait(&waitloc, &lock);
    pthread_mutex_unlock(&lock);

    /* SIGQUIT has been caught and is now blocked; do whatever */
    quitflag = 0;

    /* reset signal mask which unblocks SIGQUIT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    exit(0);
}
```

Figure 12.16 Synchronous signal handling

Instead of relying on a signal handler that interrupts the main thread of control, we dedicate a separate thread of control to handle the signals. We change the value of `quitflag` under the protection of a mutex so that the main thread of control can't miss the wake-up call made when we call `pthread_cond_signal`. We use the same mutex

in the main thread of control to check the value of the flag and atomically release the mutex when we wait for the condition.

Note that we block SIGINT and SIGQUIT in the beginning of the main thread. When we create the thread to handle signals, the thread inherits the current signal mask. Since sigwait will unblock the signals, only one thread is available to receive signals. This enables us to code the main thread without having to worry about interrupts from these signals.

If we run this program, we get output similar to that from Figure 10.23:

```
$ ./a.out
^?                  type the interrupt character
interrupt
^?                  type the interrupt character again
interrupt
^?                  and again
interrupt
^\ $                now terminate with quit character
```

□

12.9 Threads and fork

When a thread calls fork, a copy of the entire process address space is made for the child. Recall the discussion of copy-on-write in Section 8.3. The child is an entirely different process from the parent, and as long as neither one makes changes to its memory contents, copies of the memory pages can be shared between parent and child.

By inheriting a copy of the address space, the child also inherits the state of every mutex, reader–writer lock, and condition variable from the parent process. If the parent consists of more than one thread, the child will need to clean up the lock state if it isn't going to call exec immediately after fork returns.

Inside the child process, only one thread exists. It is made from a copy of the thread that called fork in the parent. If the threads in the parent process hold any locks, the same locks will also be held in the child process. The problem is that the child process doesn't contain copies of the threads holding the locks, so there is no way for the child to know which locks are held and need to be unlocked.

This problem can be avoided if the child calls one of the exec functions directly after returning from fork. In this case, the old address space is discarded, so the lock state doesn't matter. This is not always possible, however, so if the child needs to continue processing, we need to use a different strategy.

To avoid problems with inconsistent state in a multithreaded process, POSIX.1 states that only async-signal safe functions should be called by a child process between the time that fork returns and the time that the child calls one of the exec functions. This limits what the child can do before calling exec, but doesn't address the problem of lock state in the child process.

To clean up the lock state, we can establish *fork handlers* by calling the function pthread_atfork.

```
#include <pthread.h>

int pthread_atfork(void (*prepare)(void), void (*parent)(void),
                   void (*child)(void));
```

Returns: 0 if OK, error number on failure

With `pthread_atfork`, we can install up to three functions to help clean up the locks. The *prepare* fork handler is called in the parent before `fork` creates the child process. This fork handler's job is to acquire all locks defined by the parent. The *parent* fork handler is called in the context of the parent after `fork` has created the child process, but before `fork` has returned. This fork handler's job is to unlock all the locks acquired by the *prepare* fork handler. The *child* fork handler is called in the context of the child process before returning from `fork`. Like the *parent* fork handler, the *child* fork handler must release all the locks acquired by the *prepare* fork handler.

Note that the locks are not locked once and unlocked twice, as it might appear. When the child address space is created, it gets a copy of all locks that the parent defined. Because the *prepare* fork handler acquired all the locks, the memory in the parent and the memory in the child start out with identical contents. When the parent and the child unlock their "copy" of the locks, new memory is allocated for the child, and the memory contents from the parent are copied to the child's memory (copy-on-write), so we are left with a situation that looks as if the parent locked all its copies of the locks and the child locked all its copies of the locks. The parent and the child end up unlocking duplicate locks stored in different memory locations, as if the following sequence of events occurred:

1. The parent acquired all its locks.
2. The child acquired all its locks.
3. The parent released its locks.
4. The child released its locks.

We can call `pthread_atfork` multiple times to install more than one set of fork handlers. If we don't have a need to use one of the handlers, we can pass a null pointer for the particular handler argument, and it will have no effect. When multiple fork handlers are used, the order in which the handlers are called differs. The *parent* and *child* fork handlers are called in the order in which they were registered, whereas the *prepare* fork handlers are called in the opposite order from which they were registered. This ordering allows multiple modules to register their own fork handlers and still honor the locking hierarchy.

For example, assume that module A calls functions from module B and that each module has its own set of locks. If the locking hierarchy is A before B, module B must install its fork handlers before module A. When the parent calls `fork`, the following steps are taken, assuming that the child process runs before the parent:

1. The *prepare* fork handler from module A is called to acquire all of module A's locks.
2. The *prepare* fork handler from module B is called to acquire all of module B's locks.

3. A child process is created.
4. The *child* fork handler from module B is called to release all of module B's locks in the child process.
5. The *child* fork handler from module A is called to release all of module A's locks in the child process.
6. The `fork` function returns to the child.
7. The *parent* fork handler from module B is called to release all of module B's locks in the parent process.
8. The *parent* fork handler from module A is called to release all of module A's locks in the parent process.
9. The `fork` function returns to the parent.

If the fork handlers serve to clean up the lock state, what cleans up the state of condition variables? On some implementations, condition variables might not need any cleaning up. However, an implementation that uses a lock as part of the implementation of condition variables will require cleaning up. The problem is that no interface exists to allow us to do this. If the lock is embedded in the condition variable data structure, then we can't use condition variables after calling `fork`, because there is no portable way to clean up its state. On the other hand, if an implementation uses a global lock to protect all condition variable data structures in a process, then the implementation itself can clean up the lock in the `fork` library routine. Application programs shouldn't rely on implementation details like this, however.

Example

The program in Figure 12.17 illustrates the use of `pthread_atfork` and fork handlers.

```
#include "apue.h"
#include <pthread.h>

pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;

void
prepare(void)
{
    int err;

    printf("preparing locks...\n");
    if ((err = pthread_mutex_lock(&lock1)) != 0)
        err_cont(err, "can't lock lock1 in prepare handler");
    if ((err = pthread_mutex_lock(&lock2)) != 0)
        err_cont(err, "can't lock lock2 in prepare handler");
}

void
parent(void)
{
```

```
    int err;

    printf("parent unlocking locks...\n");
    if ((err = pthread_mutex_unlock(&lock1)) != 0)
        err_cont(err, "can't unlock lock1 in parent handler");
    if ((err = pthread_mutex_unlock(&lock2)) != 0)
        err_cont(err, "can't unlock lock2 in parent handler");
}

void
child(void)
{
    int err;

    printf("child unlocking locks...\n");
    if ((err = pthread_mutex_unlock(&lock1)) != 0)
        err_cont(err, "can't unlock lock1 in child handler");
    if ((err = pthread_mutex_unlock(&lock2)) != 0)
        err_cont(err, "can't unlock lock2 in child handler");
}

void *
thr_fn(void *arg)
{
    printf("thread started...\n");
    pause();
    return(0);
}

int
main(void)
{
    int         err;
    pid_t       pid;
    pthread_t   tid;

    if ((err = pthread_atfork(prepare, parent, child)) != 0)
        err_exit(err, "can't install fork handlers");
    if ((err = pthread_create(&tid, NULL, thr_fn, 0)) != 0)
        err_exit(err, "can't create thread");

    sleep(2);
    printf("parent about to fork...\n");

    if ((pid = fork()) < 0)
        err_quit("fork failed");
    else if (pid == 0)  /* child */
        printf("child returned from fork\n");
    else        /* parent */
        printf("parent returned from fork\n");
    exit(0);
}
```

Figure 12.17 `pthread_atfork` example

In Figure 12.17, we define two mutexes, `lock1` and `lock2`. The *prepare* fork handler acquires them both, the *child* fork handler releases them in the context of the child process, and the *parent* fork handler releases them in the context of the parent process.

When we run this program, we get the following output:

```
$ ./a.out
thread started...
parent about to fork...
preparing locks...
child unlocking locks...
child returned from fork
parent unlocking locks...
parent returned from fork
```

As we can see, the *prepare* fork handler runs after `fork` is called, the *child* fork handler runs before `fork` returns in the child, and the *parent* fork handler runs before `fork` returns in the parent. □

Although the `pthread_atfork` mechanism is intended to make locking state consistent after a `fork`, it has several drawbacks that make it usable in only limited circumstances:

- There is no good way to reinitialize the state for more complex synchronization objects such as condition variables and barriers.
- Some implementations of error-checking mutexes will generate errors when the child fork handler tries to unlock a mutex that was locked by the parent.
- Recursive mutexes can't be cleaned up in the child fork handler, because there is no way to determine the number of times one has been locked.
- If child processes are allowed to call only async-signal safe functions, then the child fork handler shouldn't even be able to clean up synchronization objects, because none of the functions that are used to manipulate them are async-signal safe. The practical problem is that a synchronization object might be in an intermediate state when one thread calls `fork`, but the synchronization object can't be cleaned up unless it is in a consistent state.
- If an application calls `fork` in a signal handler (which is legal, because `fork` is async-signal safe), then the fork handlers registered by `pthread_atfork` can call only async-signal safe functions, or else the results are undefined.

12.10 Threads and I/O

We introduced the `pread` and `pwrite` functions in Section 3.11. These functions are helpful in a multithreaded environment, because all threads in a process share the same file descriptors.

Consider two threads reading from or writing to the same file descriptor at the same time.

Thread A	Thread B
`lseek(fd, 300, SEEK_SET);`	`lseek(fd, 700, SEEK_SET);`
`read(fd, buf1, 100);`	`read(fd, buf2, 100);`

If thread A executes the call to `lseek` and then thread B calls `lseek` before thread A calls `read`, then both threads will end up reading the same record. Clearly, this isn't what was intended.

To solve this problem, we can use `pread` to make the setting of the offset and the reading of the data one atomic operation.

Thread A	Thread B
`pread(fd, buf1, 100, 300);`	`pread(fd, buf2, 100, 700);`

Using `pread`, we can ensure that thread A reads the record at offset 300, whereas thread B reads the record at offset 700. We can use `pwrite` to solve the problem of concurrent threads writing to the same file.

12.11 Summary

Threads provide an alternative model for partitioning concurrent tasks in UNIX systems. They promote sharing among separate threads of control, but present unique synchronization problems. In this chapter, we looked at how we can fine-tune our threads and their synchronization primitives. We discussed reentrancy with threads. We also looked at how threads interact with some of the process-oriented system calls.

Exercises

12.1 Run the program in Figure 12.17 on a Linux system, but redirect the output into a file. Explain the results.

12.2 Implement `putenv_r`, a reentrant version of `putenv`. Make sure that your implementation is async-signal safe as well as thread-safe.

12.3 Can you make the `getenv` function shown in Figure 12.13 async-signal safe by blocking signals at the beginning of the function and restoring the previous signal mask before returning? Explain.

12.4 Write a program to exercise the version of `getenv` from Figure 12.13. Compile and run the program on FreeBSD. What happens? Explain.

12.5 Given that you can create multiple threads to perform different tasks within a program, explain why you might still need to use `fork`.

12.6 Reimplement the program in Figure 10.29 to make it thread-safe without using `nanosleep` or `clock_nanosleep`.

12.7 After calling `fork`, could we safely reinitialize a condition variable in the child process by first destroying the condition variable with `pthread_cond_destroy` and then initializing it with `pthread_cond_init`?

12.8 The `timeout` function in Figure 12.8 can be simplified substantially. Explain how.

13

Daemon Processes

13.1 Introduction

Daemons are processes that live for a long time. They are often started when the system is bootstrapped and terminate only when the system is shut down. Because they don't have a controlling terminal, we say that they run in the background. UNIX systems have numerous daemons that perform day-to-day activities.

In this chapter, we look at the process structure of daemons and explore how to write a daemon. Since a daemon does not have a controlling terminal, we need to see how a daemon can report error conditions when something goes wrong.

> For a discussion of the historical background of the term *daemon* as it applies to computer systems, see Raymond [1996].

13.2 Daemon Characteristics

Let's look at some common system daemons and how they relate to the concepts of process groups, controlling terminals, and sessions that we described in Chapter 9. The `ps`(1) command prints the status of various processes in the system. There are a multitude of options—consult your system's manual for all the details. We'll execute

```
ps -axj
```

under BSD-based systems to see the information we need for this discussion. The `-a` option shows the status of processes owned by others, and `-x` shows processes that don't have a controlling terminal. The `-j` option displays the job-related information: the session ID, process group ID, controlling terminal, and terminal process group ID.

Under System V–based systems, a similar command is `ps -efj`. (In an attempt to improve security, some UNIX systems don't allow us to use `ps` to look at any processes other than our own.) The output from `ps` looks like

```
UID        PID  PPID  PGID   SID  TTY  CMD
root         1     0     1     1  ?    /sbin/init
root         2     0     0     0  ?    [kthreadd]
root         3     2     0     0  ?    [ksoftirqd/0]
root         6     2     0     0  ?    [migration/0]
root         7     2     0     0  ?    [watchdog/0]
root        21     2     0     0  ?    [cpuset]
root        22     2     0     0  ?    [khelper]
root        26     2     0     0  ?    [sync_supers]
root        27     2     0     0  ?    [bdi-default]
root        29     2     0     0  ?    [kblockd]
root        35     2     0     0  ?    [kswapd0]
root        49     2     0     0  ?    [scsi_eh_0]
root       256     2     0     0  ?    [jbd2/sda5-8]
root       257     2     0     0  ?    [ext4-dio-unwrit]
syslog     847     1   843   843  ?    rsyslogd -c5
root       906     1   906   906  ?    /usr/sbin/cupsd -F
root      1037     1  1037  1037  ?    /usr/sbin/inetd
root      1067     1  1067  1067  ?    cron
daemon    1068     1  1068  1068  ?    atd
root      8196     1  8196  8196  ?    /usr/sbin/sshd -D
root     13047     2     0     0  ?    [kworker/1:0]
root     14596     2     0     0  ?    [flush-8:0]
root     26464     1 26464 26464  ?    rpcbind -w
statd    28490     1 28490 28490  ?    rpc.statd -L
root     28553     2     0     0  ?    [rpciod]
root     28554     2     0     0  ?    [nfsiod]
root     28561     1 28561 28561  ?    rpc.idmapd
root     28761     2     0     0  ?    [lockd]
root     28764     2     0     0  ?    [nfsd]
root     28775     1 28775 28775  ?    /usr/sbin/rpc.mountd --manage-gids
```

We have removed a few columns that don't interest us, such as the accumulated CPU time. The column headings, in order, are the user ID, process ID, parent process ID, process group ID, session ID, terminal name, and command string.

> The system that this `ps` command was run on (Linux 3.2.0) supports the notion of a session ID, which we mentioned with the `setsid` function in Section 9.5. The session ID is simply the process ID of the session leader. Some BSD-based systems, such as Mac OS X 10.6.8, will print the address of the `session` structure corresponding to the process group that the process belongs to (Section 9.11) instead of the session ID.

The system processes you see will depend on the operating system implementation. Anything with a parent process ID of 0 is usually a kernel process started as part of the system bootstrap procedure. (An exception is `init`, which is a user-level command started by the kernel at boot time.) Kernel processes are special and generally exist for the entire lifetime of the system. They run with superuser privileges and have no controlling terminal and no command line.

In the sample `ps` output, kernel daemons appear with their names in square brackets. This version of Linux uses a special kernel process, `kthreadd`, to create other kernel processes, so `kthreadd` appears as the parent of the other kernel daemons. Each kernel component that needs to perform work in a process context, but that isn't invoked from the context of a user-level process, will usually have its own kernel daemon. For example, on Linux

- The `kswapd` daemon is also known as the pageout daemon. It supports the virtual memory subsystem by writing dirty pages to disk slowly over time, so the pages can be reclaimed.
- The `flush` daemon flushes dirty pages to disk when available memory reaches a configured minimum threshold. It also flushes dirty pages back to disk at regular intervals to decrease data loss in the event of a system failure. Several flush daemons can exist—one for each backing device. The sample output shows one flush daemon named `flush-8:0`. In the name, the backing device is identified by its major device number (8) and its minor device number (0).
- The `sync_supers` daemon periodically flushes file system metadata to disk.
- The `jbd` daemon helps implement the journal in the `ext4` file system.

Process 1 is usually `init` (`launchd` on Mac OS X), as we described in Section 8.2. It is a system daemon responsible for, among other things, starting system services specific to various run levels. These services are usually implemented with the help of their own daemons.

The `rpcbind` daemon provides the service of mapping RPC (Remote Procedure Call) program numbers to network port numbers. The `rsyslogd` daemon is available to any program to log system messages for an administrator. The messages may be printed on a console device and also written to a file. (We describe the `syslog` facility in Section 13.4.)

We talked about the `inetd` daemon in Section 9.3. It listens on the system's network interfaces for incoming requests for various network servers. The `nfsd`, `nfsiod`, `lockd`, `rpciod`, `rpc.idmapd`, `rpc.statd`, and `rpc.mountd` daemons provide support for the Network File System (NFS). Note that the first four are kernel daemons, while the last three are user-level daemons.

The `cron` daemon executes commands at regularly scheduled dates and times. Numerous system administration tasks are handled by `cron` running programs at regularly intervals. The `atd` daemon is similar to `cron`; it allows users to execute jobs at specified times, but it executes each job once only, instead of repeatedly at regularly scheduled times. The `cupsd` daemon is a print spooler; it handles print requests on the system. The `sshd` daemon provides secure remote login and execution facilities.

Note that most of the daemons run with superuser (root) privileges. None of the daemons has a controlling terminal: the terminal name is set to a question mark. The kernel daemons are started without a controlling terminal. The lack of a controlling terminal in the user-level daemons is probably the result of the daemons having called `setsid`. Most of the user-level daemons are process group leaders and session leaders, and are the only processes in their process group and session. (The one exception is `rsyslogd`.) Finally, note that the parent of the user-level daemons is the `init` process.

13.3 Coding Rules

Some basic rules to coding a daemon prevent unwanted interactions from happening. We state these rules here and then show a function, `daemonize`, that implements them.

1. Call `umask` to set the file mode creation mask to a known value, usually 0. The inherited file mode creation mask could be set to deny certain permissions. If the daemon process creates files, it may want to set specific permissions. For example, if it creates files with group-read and group-write enabled, a file mode creation mask that turns off either of these permissions would undo its efforts. On the other hand, if the daemon calls library functions that result in files being created, then it might make sense to set the file mode create mask to a more restrictive value (such as 007), since the library functions might not allow the caller to specify the permissions through an explicit argument.
2. Call `fork` and have the parent `exit`. This does several things. First, if the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done. Second, the child inherits the process group ID of the parent but gets a new process ID, so we're guaranteed that the child is not a process group leader. This is a prerequisite for the call to `setsid` that is done next.
3. Call `setsid` to create a new session. The three steps listed in Section 9.5 occur. The process (a) becomes the leader of a new session, (b) becomes the leader of a new process group, and (c) is disassociated from its controlling terminal.

   > Under System V–based systems, some people recommend calling `fork` again at this point, terminating the parent, and continuing the daemon in the child. This guarantees that the daemon is not a session leader, which prevents it from acquiring a controlling terminal under the System V rules (Section 9.6). Alternatively, to avoid acquiring a controlling terminal, be sure to specify `O_NOCTTY` whenever opening a terminal device.

4. Change the current working directory to the root directory. The current working directory inherited from the parent could be on a mounted file system. Since daemons normally exist until the system is rebooted, if the daemon stays on a mounted file system, that file system cannot be unmounted.

   Alternatively, some daemons might change the current working directory to a specific location where they will do all their work. For example, a line printer spooling daemon might change its working directory to its spool directory.
5. Unneeded file descriptors should be closed. This prevents the daemon from holding open any descriptors that it may have inherited from its parent (which could be a shell or some other process). We can use our `open_max` function (Figure 2.17) or the `getrlimit` function (Section 7.11) to determine the highest descriptor and close all descriptors up to that value.
6. Some daemons open file descriptors 0, 1, and 2 to `/dev/null` so that any library routines that try to read from standard input or write to standard output or standard error will have no effect. Since the daemon is not associated with a

terminal device, there is nowhere for output to be displayed, nor is there anywhere to receive input from an interactive user. Even if the daemon was started from an interactive session, the daemon runs in the background, and the login session can terminate without affecting the daemon. If other users log in on the same terminal device, we wouldn't want output from the daemon showing up on the terminal, and the users wouldn't expect their input to be read by the daemon.

Example

Figure 13.1 shows a function that can be called from a program that wants to initialize itself as a daemon.

```
#include "apue.h"
#include <syslog.h>
#include <fcntl.h>
#include <sys/resource.h>

void
daemonize(const char *cmd)
{
    int                 i, fd0, fd1, fd2;
    pid_t               pid;
    struct rlimit       rl;
    struct sigaction    sa;

    /*
     * Clear file creation mask.
     */
    umask(0);

    /*
     * Get maximum number of file descriptors.
     */
    if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
        err_quit("%s: can't get file limit", cmd);

    /*
     * Become a session leader to lose controlling TTY.
     */
    if ((pid = fork()) < 0)
        err_quit("%s: can't fork", cmd);
    else if (pid != 0) /* parent */
        exit(0);
    setsid();

    /*
     * Ensure future opens won't allocate controlling TTYs.
     */
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
```

```
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0)
        err_quit("%s: can't ignore SIGHUP", cmd);
    if ((pid = fork()) < 0)
        err_quit("%s: can't fork", cmd);
    else if (pid != 0) /* parent */
        exit(0);

    /*
     * Change the current working directory to the root so
     * we won't prevent file systems from being unmounted.
     */
    if (chdir("/") < 0)
        err_quit("%s: can't change directory to /", cmd);

    /*
     * Close all open file descriptors.
     */
    if (rl.rlim_max == RLIM_INFINITY)
        rl.rlim_max = 1024;
    for (i = 0; i < rl.rlim_max; i++)
        close(i);

    /*
     * Attach file descriptors 0, 1, and 2 to /dev/null.
     */
    fd0 = open("/dev/null", O_RDWR);
    fd1 = dup(0);
    fd2 = dup(0);

    /*
     * Initialize the log file.
     */
    openlog(cmd, LOG_CONS, LOG_DAEMON);
    if (fd0 != 0 || fd1 != 1 || fd2 != 2) {
        syslog(LOG_ERR, "unexpected file descriptors %d %d %d",
          fd0, fd1, fd2);
        exit(1);
    }
}
```

Figure 13.1 Initialize a daemon process

If the `daemonize` function is called from a `main` program that then goes to sleep, we can check the status of the daemon with the `ps` command:

```
$ ./a.out
$ ps -efj
UID      PID  PPID  PGID   SID  TTY  CMD
sar    13800     1 13799 13799  ?    ./a.out
$ ps -efj | grep 13799
sar    13800     1 13799 13799  ?    ./a.out
```

We can also use `ps` to verify that no active process exists with ID 13799. This means that our daemon is in an orphaned process group (Section 9.10) and is not a session leader and, therefore, has no chance of allocating a controlling terminal. This is a result of performing the second `fork` in the `daemonize` function. We can see that our daemon has been initialized correctly. □

13.4 Error Logging

One problem a daemon has is how to handle error messages. It can't simply write to standard error, since it shouldn't have a controlling terminal. We don't want all the daemons writing to the console device, because on many workstations the console device runs a windowing system. We also don't want each daemon writing its own error messages into a separate file. It would be a headache for anyone administering the system to keep up with which daemon writes to which log file and to check these files on a regular basis. A central daemon error-logging facility is required.

> The BSD `syslog` facility was developed at Berkeley and used widely in 4.2BSD. Most systems derived from BSD support `syslog`. Until SVR4, System V never had a central daemon logging facility. The `syslog` function is included in the XSI option in the Single UNIX Specification.

The BSD `syslog` facility has been widely used since 4.2BSD. Most daemons use this facility. Figure 13.2 illustrates its structure.

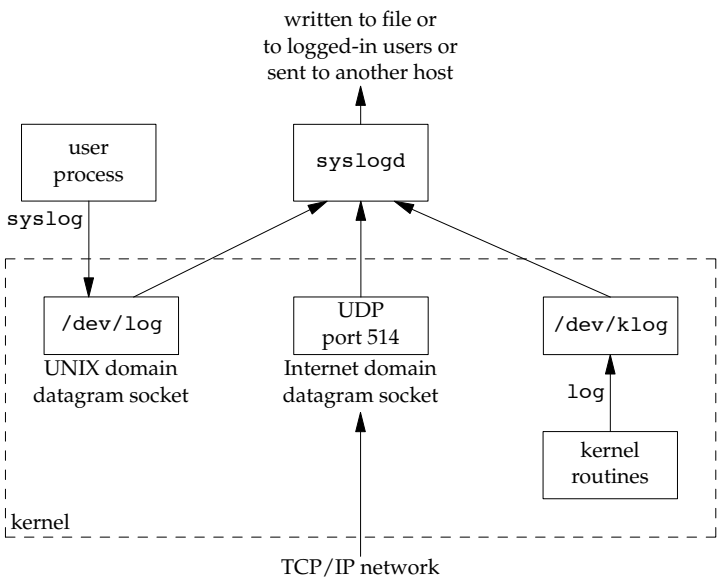


Figure 13.2 The BSD `syslog` facility

There are three ways to generate log messages:

1. Kernel routines can call the `log` function. These messages can be read by any user process that `open`s and `read`s the `/dev/klog` device. We won't describe this function any further, since we're not interested in writing kernel routines.
2. Most user processes (daemons) call the `syslog`(3) function to generate log messages. We describe its calling sequence later. This causes the message to be sent to the UNIX domain datagram socket `/dev/log`.
3. A user process on this host, or on some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514. Note that the `syslog` function never generates these UDP datagrams: they require explicit network programming by the process generating the log message.

Refer to Stevens, Fenner, and Rudoff [2004] for details on UNIX domain sockets and UDP sockets.

Normally, the `syslogd` daemon reads all three forms of log messages. On start-up, this daemon reads a configuration file, usually `/etc/syslog.conf`, which determines where different classes of messages are to be sent. For example, urgent messages can be sent to the system administrator (if logged in) and printed on the console, whereas warnings may be logged to a file.

Our interface to this facility is through the `syslog` function.

```
#include <syslog.h>

void openlog(const char *ident, int option, int facility);

void syslog(int priority, const char *format, ...);

void closelog(void);

int setlogmask(int maskpri);
```

Returns: previous log priority mask value

Calling `openlog` is optional. If it's not called, the first time `syslog` is called, `openlog` is called automatically. Calling `closelog` is also optional—it just closes the descriptor that was being used to communicate with the `syslogd` daemon.

Calling `openlog` lets us specify an *ident* that is added to each log message. This is normally the name of the program (e.g. `cron`, `inetd`). The *option* argument is a bitmask specifying various options. Figure 13.3 describes the available options, including a bullet in the XSI column if the option is included in the `openlog` definition in the Single UNIX Specification.

The *facility* argument for `openlog` is taken from Figure 13.4. Note that the Single UNIX Specification defines only a subset of the facility codes typically available on a given platform. The reason for the *facility* argument is to let the configuration file specify that messages from different facilities are to be handled differently. If we don't call `openlog`, or if we call it with a *facility* of 0, we can still specify the facility as part of the *priority* argument to `syslog`.

We call `syslog` to generate a log message. The *priority* argument is a combination of the *facility*, shown in Figure 13.4, and a *level*, shown in Figure 13.5. These *level*s are ordered by priority, from highest to lowest.

option	XSI	Description
LOG_CONS	•	If the log message can't be sent to syslogd via the UNIX domain datagram, the message is written to the console instead.
LOG_NDELAY	•	Open the UNIX domain datagram socket to the syslogd daemon immediately; don't wait until the first message is logged. Normally, the socket is not opened until the first message is logged.
LOG_NOWAIT	•	Do not wait for child processes that might have been created in the process of logging the message. This prevents conflicts with applications that catch SIGCHLD, since the application might have retrieved the child's status by the time that syslog calls wait.
LOG_ODELAY	•	Delay the opening of the connection to the syslogd daemon until the first message is logged.
LOG_PERROR		Write the log message to standard error in addition to sending it to syslogd. (Unavailable on Solaris.)
LOG_PID	•	Log the process ID with each message. This is intended for daemons that fork a child process to handle different requests (as compared to daemons, such as syslogd, that never call fork).

Figure 13.3 The *option* argument for openlog

The *format* argument and any remaining arguments are passed to the vsprintf function for formatting. Any occurrences of the characters %m in *format* are first replaced with the error message string (strerror) corresponding to the value of errno.

The setlogmask function can be used to set the log priority mask for the process. This function returns the previous mask. When the log priority mask is set, messages are not logged unless their priority is set in the log priority mask. Note that attempts to set the log priority mask to 0 will have no effect.

The logger(1) program is also provided by many systems as a way to send log messages to the syslog facility. Some implementations allow optional arguments to this program, specifying the *facility*, *level*, and *ident*, although the Single UNIX Specification doesn't define any options. The logger command is intended for a shell script running noninteractively that needs to generate log messages.

Example

In a (hypothetical) line printer spooler daemon, you might encounter the sequence

```
openlog("lpd", LOG_PID, LOG_LPR);
syslog(LOG_ERR, "open error for %s: %m", filename);
```

The first call sets the *ident* string to the program name, specifies that the process ID should always be printed, and sets the default *facility* to the line printer system. The call to syslog specifies an error condition and a message string. If we had not called openlog, the second call could have been

```
syslog(LOG_ERR | LOG_LPR, "open error for %s: %m", filename);
```

Here, we specify the *priority* argument as a combination of a *level* and a *facility*. □

facility	XSI	Description
LOG_AUDIT		the audit facility
LOG_AUTH		authorization programs: login, su, getty, ...
LOG_AUTHPRIV		same as LOG_AUTH, but logged to file with restricted permissions
LOG_CONSOLE		messages written to /dev/console
LOG_CRON		cron and at
LOG_DAEMON		system daemons: inetd, routed, ...
LOG_FTP		the FTP daemon (ftpd)
LOG_KERN		messages generated by the kernel
LOG_LOCAL0	•	reserved for local use
LOG_LOCAL1	•	reserved for local use
LOG_LOCAL2	•	reserved for local use
LOG_LOCAL3	•	reserved for local use
LOG_LOCAL4	•	reserved for local use
LOG_LOCAL5	•	reserved for local use
LOG_LOCAL6	•	reserved for local use
LOG_LOCAL7	•	reserved for local use
LOG_LPR		line printer system: lpd, lpc, ...
LOG_MAIL		the mail system
LOG_NEWS		the Usenet network news system
LOG_NTP		the network time protocol system
LOG_SECURITY		the security subsystem
LOG_SYSLOG		the syslogd daemon itself
LOG_USER	•	messages from other user processes (default)
LOG_UUCP		the UUCP system

Figure 13.4 The *facility* argument for openlog

level	Description
LOG_EMERG	emergency (system is unusable) (highest priority)
LOG_ALERT	condition that must be fixed immediately
LOG_CRIT	critical condition (e.g., hard device error)
LOG_ERR	error condition
LOG_WARNING	warning condition
LOG_NOTICE	normal, but significant condition
LOG_INFO	informational message
LOG_DEBUG	debug message (lowest priority)

Figure 13.5 The syslog *level*s (ordered)

In addition to syslog, many platforms provide a variant that handles variable argument lists.

```
#include <syslog.h>
#include <stdarg.h>

void vsyslog(int priority, const char *format, va_list arg);
```

All four platforms described in this book provide vsyslog, but this function is not included in the Single UNIX Specification. Note that to make its declaration visible to your application,

you might need to define an additional symbol, such as `__BSD_VISIBLE` on FreeBSD or `__USE_BSD` on Linux.

Most `syslogd` implementations will queue messages for a short time. If a duplicate message arrives during this period, the `syslog` daemon will not write it to the log. Instead, the daemon prints a message similar to "last message repeated *N* times."

13.5 Single-Instance Daemons

Some daemons are implemented so that only a single copy of the daemon should be running at a time for proper operation. Such a daemon might need exclusive access to a device, for example. In the case of the `cron` daemon, if multiple instances were running, each copy might try to start a single scheduled operation, resulting in duplicate operations and probably an error.

If the daemon needs to access a device, the device driver will sometimes prevent multiple attempts to open the corresponding device node in `/dev`. This restricts us to one copy of the daemon running at a time. If no such device is available, however, we need to do the work ourselves.

The file- and record-locking mechanism provides the basis for one way to ensure that only one copy of a daemon is running. (We discuss file and record locking in Section 14.3.) If each daemon creates a file with a fixed name and places a write lock on the entire file, only one such write lock will be allowed to be created. Successive attempts to create write locks will fail, serving as an indication to successive copies of the daemon that another instance is already running.

File and record locking provides a convenient mutual-exclusion mechanism. If the daemon obtains a write-lock on an entire file, the lock will be removed automatically if the daemon exits. This simplifies recovery, eliminating the need for us to clean up from the previous instance of the daemon.

Example

The function shown in Figure 13.6 illustrates the use of file and record locking to ensure that only one copy of a daemon is running.

```
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <syslog.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <sys/stat.h>

#define LOCKFILE "/var/run/daemon.pid"
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)

extern int lockfile(int);
```

```
int
already_running(void)
{
    int     fd;
    char    buf[16];

    fd = open(LOCKFILE, O_RDWR|O_CREAT, LOCKMODE);
    if (fd < 0) {
        syslog(LOG_ERR, "can't open %s: %s", LOCKFILE, strerror(errno));
        exit(1);
    }
    if (lockfile(fd) < 0) {
        if (errno == EACCES || errno == EAGAIN) {
            close(fd);
            return(1);
        }
        syslog(LOG_ERR, "can't lock %s: %s", LOCKFILE, strerror(errno));
        exit(1);
    }
    ftruncate(fd, 0);
    sprintf(buf, "%ld", (long)getpid());
    write(fd, buf, strlen(buf)+1);
    return(0);
}
```

Figure 13.6 Ensure that only one copy of a daemon is running

Each copy of the daemon will try to create a file and write its process ID in the file. This will allow administrators to identify the process easily. If the file is already locked, the `lockfile` function will fail with `errno` set to `EACCES` or `EAGAIN`, so we return 1, indicating that the daemon is already running. Otherwise, we truncate the file, write our process ID to it, and return 0.

We need to truncate the file, because the previous instance of the daemon might have had a process ID larger than ours, with a larger string length. For example, if the previous instance of the daemon was process ID 12345, and the new instance is process ID 9999, when we write the process ID to the file, we will be left with 99995 in the file. Truncating the file prevents data from the previous daemon appearing as if it applies to the current daemon. □

13.6 Daemon Conventions

Several common conventions are followed by daemons in the UNIX System.

- If the daemon uses a lock file, the file is usually stored in `/var/run`. Note, however, that the daemon might need superuser permissions to create a file here. The name of the file is usually *name*`.pid`, where *name* is the name of the daemon or the service. For example, on Linux, the name of the `cron` daemon's lock file is `/var/run/crond.pid`.

- If the daemon supports configuration options, they are usually stored in `/etc`. The configuration file is named *name*`.conf`, where *name* is the name of the daemon or the name of the service. For example, the configuration for the `syslogd` daemon is usually `/etc/syslog.conf`.
- Daemons can be started from the command line, but they are usually started from one of the system initialization scripts (`/etc/rc*` or `/etc/init.d/*`). If the daemon should be restarted automatically when it exits, we can arrange for `init` to restart it if we include a `respawn` entry for it in `/etc/inittab` (assuming the system uses a System V style `init` command).
- If a daemon has a configuration file, the daemon reads the file when it starts, but usually won't look at it again. If an administrator changes the configuration, the daemon would need to be stopped and restarted to account for the configuration changes. To avoid this, some daemons will catch `SIGHUP` and reread their configuration files when they receive the signal. Since they aren't associated with terminals and are either session leaders without controlling terminals or members of orphaned process groups, daemons have no reason to expect to receive `SIGHUP`. Thus they can safely reuse it.

Example

The program shown in Figure 13.7 shows one way a daemon can reread its configuration file. The program uses `sigwait` and multiple threads, as discussed in Section 12.8.

```
#include "apue.h"
#include <pthread.h>
#include <syslog.h>

sigset_t    mask;

extern int already_running(void);

void
reread(void)
{
    /* ... */
}

void *
thr_fn(void *arg)
{
    int err, signo;

    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0) {
            syslog(LOG_ERR, "sigwait failed");
            exit(1);
```

```
        }

        switch (signo) {
        case SIGHUP:
            syslog(LOG_INFO, "Re-reading configuration file");
            reread();
            break;

        case SIGTERM:
            syslog(LOG_INFO, "got SIGTERM; exiting");
            exit(0);

        default:
            syslog(LOG_INFO, "unexpected signal %d\n", signo);
        }
    }
    return(0);
}

int
main(int argc, char *argv[])
{
    int                 err;
    pthread_t           tid;
    char                *cmd;
    struct sigaction    sa;

    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
    else
        cmd++;

    /*
     * Become a daemon.
     */
    daemonize(cmd);

    /*
     * Make sure only one copy of the daemon is running.
     */
    if (already_running()) {
        syslog(LOG_ERR, "daemon already running");
        exit(1);
    }

    /*
     * Restore SIGHUP default and block all signals.
     */
    sa.sa_handler = SIG_DFL;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0)
        err_quit("%s: can't restore SIGHUP default");
```

```
    sigfillset(&mask);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
        err_exit(err, "SIG_BLOCK error");

    /*
     * Create a thread to handle SIGHUP and SIGTERM.
     */
    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "can't create thread");

    /*
     * Proceed with the rest of the daemon.
     */
    /* ... */
    exit(0);
}
```

Figure 13.7 Daemon rereading configuration files

We call `daemonize` from Figure 13.1 to initialize the daemon. When it returns, we call `already_running` from Figure 13.6 to ensure that only one copy of the daemon is running. At this point, `SIGHUP` is still ignored, so we need to reset the disposition to the default behavior; otherwise, the thread calling `sigwait` may never see the signal.

We block all signals, as is recommended for multithreaded programs, and create a thread to handle signals. The thread's only job is to wait for `SIGHUP` and `SIGTERM`. When it receives `SIGHUP`, the thread calls `reread` to reread its configuration file. When it receives `SIGTERM`, the thread logs a message and exits.

Recall from Figure 10.1 that the default action for `SIGHUP` and `SIGTERM` is to terminate the process. Because we block these signals, the daemon will not die when one of them is sent to the process. Instead, the thread calling `sigwait` will return with an indication that the signal has been received. □

Example

Not all daemons are multithreaded. The program in Figure 13.8 shows how a single-threaded daemon can catch `SIGHUP` and reread its configuration file.

```
#include "apue.h"
#include <syslog.h>
#include <errno.h>

extern int lockfile(int);
extern int already_running(void);

void
reread(void)
{
    /* ... */
```

```
}

void
sigterm(int signo)
{
    syslog(LOG_INFO, "got SIGTERM; exiting");
    exit(0);
}

void
sighup(int signo)
{
    syslog(LOG_INFO, "Re-reading configuration file");
    reread();
}

int
main(int argc, char *argv[])
{
    char                *cmd;
    struct sigaction    sa;

    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
    else
        cmd++;

    /*
     * Become a daemon.
     */
    daemonize(cmd);

    /*
     * Make sure only one copy of the daemon is running.
     */
    if (already_running()) {
        syslog(LOG_ERR, "daemon already running");
        exit(1);
    }

    /*
     * Handle signals of interest.
     */
    sa.sa_handler = sigterm;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGHUP);
    sa.sa_flags = 0;
    if (sigaction(SIGTERM, &sa, NULL) < 0) {
        syslog(LOG_ERR, "can't catch SIGTERM: %s", strerror(errno));
        exit(1);
    }
    sa.sa_handler = sighup;
    sigemptyset(&sa.sa_mask);
```

```
    sigaddset(&sa.sa_mask, SIGTERM);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0) {
        syslog(LOG_ERR, "can't catch SIGHUP: %s", strerror(errno));
        exit(1);
    }

    /*
     * Proceed with the rest of the daemon.
     */
    /* ... */
    exit(0);
}
```

Figure 13.8 Alternative implementation of daemon rereading configuration files

After initializing the daemon, we install signal handlers for `SIGHUP` and `SIGTERM`. We can either place the reread logic in the signal handler or just set a flag in the handler and have the main thread of the daemon do all the work instead. □

13.7 Client–Server Model

A common use for a daemon process is as a server process. Indeed, in Figure 13.2, we can call the `syslogd` process a server that has messages sent to it by user processes (clients) using a UNIX domain datagram socket.

In general, a *server* is a process that waits for a *client* to contact it, requesting some type of service. In Figure 13.2, the service being provided by the `syslogd` server is the logging of an error message.

In Figure 13.2, the communication between the client and the server is one way. The client sends its service request to the server; the server sends nothing back to the client. In the upcoming chapters, we'll see numerous examples of two-way communication between a client and a server—the client sends a request to the server, and the server sends a reply back to the client.

It is common to find servers that `fork` and `exec` another program to provide service to a client. These servers often manage multiple file descriptors: communication endpoints, configuration files, log files, and the like. At best, it would be careless to leave these file descriptors open in the child process, because they probably won't be used in the program executed by the child, especially if the program is unrelated to the server. At worst, leaving them open could pose a security problem—the program executed could do something malicious, such as change the server's configuration file or trick the client into thinking it is communicating with the server, thereby gaining access to unauthorized information.

An easy solution to this problem is to set the close-on-exec flag for all file descriptors that the executed program won't need. Figure 13.9 shows a function that we can use in a server process to do just this.

```
#include "apue.h"
#include <fcntl.h>

int
set_cloexec(int fd)
{
    int     val;

    if ((val = fcntl(fd, F_GETFD, 0)) < 0)
        return(-1);

    val |= FD_CLOEXEC;      /* enable close-on-exec */

    return(fcntl(fd, F_SETFD, val));
}
```

Figure 13.9 Set close-on-exec flag

13.8 Summary

Daemon processes are running all the time on most UNIX systems. Initializing our own process to run as a daemon takes some care and an understanding of the process relationships described in Chapter 9. In this chapter, we developed a function that can be called by a daemon process to initialize itself correctly.

We also discussed the ways a daemon can log error messages, since a daemon normally doesn't have a controlling terminal. We discussed several conventions that daemons follow on most UNIX systems and showed examples of how to implement some of these conventions.

Exercises

13.1 As we might guess from Figure 13.2, when the `syslog` facility is initialized, either by calling `openlog` directly or on the first call to `syslog`, the special device file for the UNIX domain datagram socket, `/dev/log`, has to be opened. What happens if the user process (the daemon) calls `chroot` before calling `openlog`?

13.2 Recall the sample `ps` output from Section 13.2. The only user-level daemon that isn't a session leader is the `rsyslogd` process. Explain why the `syslogd` daemon isn't a session leader.

13.3 List all the daemons active on your system, and identify the function of each one.

13.4 Write a program that calls the `daemonize` function in Figure 13.1. After calling this function, call `getlogin` (Section 8.15) to see whether the process has a login name now that it has become a daemon. Print the results to a file.

14

Advanced I/O

14.1 Introduction

This chapter covers numerous topics and functions that we lump under the term *advanced I/O*: nonblocking I/O, record locking, I/O multiplexing (the `select` and `poll` functions), asynchronous I/O, the `readv` and `writev` functions, and memory-mapped I/O (`mmap`). We need to cover these topics before describing interprocess communication in Chapter 15, Chapter 17, and many of the examples in later chapters.

14.2 Nonblocking I/O

In Section 10.5, we said that system calls are divided into two categories: the "slow" ones and all the others. The slow system calls are those that can block forever. They include

- Reads that can block the caller forever if data isn't present with certain file types (pipes, terminal devices, and network devices)
- Writes that can block the caller forever if the data can't be accepted immediately by these same file types (e.g., no room in the pipe, network flow control)
- Opens that block until some condition occurs on certain file types (such as an open of a terminal device that waits until an attached modem answers the phone, or an open of a FIFO for writing only, when no other process has the FIFO open for reading)
- Reads and writes of files that have mandatory record locking enabled

- Certain `ioctl` operations
- Some of the interprocess communication functions (Chapter 15)

We also said that system calls related to disk I/O are not considered slow, even though the read or write of a disk file can block the caller temporarily.

Nonblocking I/O lets us issue an I/O operation, such as an `open`, `read`, or `write`, and not have it block forever. If the operation cannot be completed, the call returns immediately with an error noting that the operation would have blocked.

There are two ways to specify nonblocking I/O for a given descriptor.

1. If we call `open` to get the descriptor, we can specify the `O_NONBLOCK` flag (Section 3.3).
2. For a descriptor that is already open, we call `fcntl` to turn on the `O_NONBLOCK` file status flag (Section 3.14). Figure 3.12 shows a function that we can call to turn on any of the file status flags for a descriptor.

> Earlier versions of System V used the flag `O_NDELAY` to specify nonblocking mode. These versions of System V returned a value of 0 from the `read` function if there wasn't any data to be read. Since this use of a return value of 0 overlapped with the normal UNIX System convention of 0 meaning the end of file, POSIX.1 chose to provide a nonblocking flag with a different name and different semantics. Indeed, with these older versions of System V, when we get a return of 0 from `read`, we don't know whether the call would have blocked or whether the end of file was encountered. We'll see that POSIX.1 requires that `read` return –1 with `errno` set to `EAGAIN` if there is no data to read from a nonblocking descriptor. Some platforms derived from System V support both the older `O_NDELAY` and the POSIX.1 `O_NONBLOCK`, but in this text we'll use only the POSIX.1 feature. The older `O_NDELAY` is intended for backward compatibility and should not be used in new applications.
>
> 4.3BSD provided the `FNDELAY` flag for `fcntl`, and its semantics were slightly different. Instead of affecting only the file status flags for the descriptor, the flags for either the terminal device or the socket were also changed to be nonblocking, thereby affecting all users of the terminal or socket, not just the users sharing the same file table entry (4.3BSD nonblocking I/O worked only on terminals and sockets). Also, 4.3BSD returned `EWOULDBLOCK` if an operation on a nonblocking descriptor could not complete without blocking. Today, BSD-based systems provide the POSIX.1 `O_NONBLOCK` flag and define `EWOULDBLOCK` to be the same as `EAGAIN`. These systems provide nonblocking semantics consistent with other POSIX-compatible systems: changes in file status flags affect all users of the same file table entry, but are independent of accesses to the same device through other file table entries. (Refer to Figures 3.7 and 3.9.)

Example

Let's look at an example of nonblocking I/O. The program in Figure 14.1 reads up to 500,000 bytes from the standard input and attempts to write it to the standard output. The standard output is first set to be nonblocking. The output is in a loop, with the results of each `write` being printed on the standard error. The function `clr_fl` is similar to the function `set_fl` that we showed in Figure 3.12. This new function simply clears one or more of the flag bits.

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>

char    buf[500000];

int
main(void)
{
    int     ntowrite, nwrite;
    char    *ptr;

    ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
    fprintf(stderr, "read %d bytes\n", ntowrite);

    set_fl(STDOUT_FILENO, O_NONBLOCK);  /* set nonblocking */

    ptr = buf;
    while (ntowrite > 0) {
        errno = 0;
        nwrite = write(STDOUT_FILENO, ptr, ntowrite);
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);

        if (nwrite > 0) {
            ptr += nwrite;
            ntowrite -= nwrite;
        }
    }

    clr_fl(STDOUT_FILENO, O_NONBLOCK);  /* clear nonblocking */

    exit(0);
}
```

Figure 14.1 Large nonblocking `write`

If the standard output is a regular file, we expect the `write` to be executed once:

```
$ ls -l /etc/services                              print file size
-rw-r--r--  1 root     677959 Jun 23  2009 /etc/services
$ ./a.out < /etc/services > temp.file              try a regular file first
read 500000 bytes
nwrite = 500000, errno = 0                         a single write
$ ls -l temp.file                                  verify size of output file
-rw-rw-r--  1 sar      500000 Apr  1 13:03 temp.file
```

But if the standard output is a terminal, we expect the `write` to return a partial count sometimes and an error at other times. This is what we see:

```
$ ./a.out < /etc/services 2>stderr.out          output to terminal
                                                lots of output to terminal ...
$ cat stderr.out
read 500000 bytes
nwrite = 999, errno = 0
nwrite = -1, errno = 35
nwrite = -1, errno = 35
nwrite = -1, errno = 35
nwrite = -1, errno = 35
nwrite = 1001, errno = 0
nwrite = -1, errno = 35
nwrite = 1002, errno = 0
nwrite = 1004, errno = 0
nwrite = 1003, errno = 0
nwrite = 1003, errno = 0
nwrite = 1005, errno = 0
nwrite = -1, errno = 35                         61 of these errors
   .
   .
   .
nwrite = 1006, errno = 0
nwrite = 1004, errno = 0
nwrite = 1005, errno = 0
nwrite = 1006, errno = 0
nwrite = -1, errno = 35                         108 of these errors
   .
   .
   .
nwrite = 1006, errno = 0
nwrite = 1005, errno = 0
nwrite = 1005, errno = 0
nwrite = -1, errno = 35                         681 of these errors
   .
   .
   .
                                                and so on ...
nwrite = 347, errno = 0
```

On this system, the `errno` of 35 is `EAGAIN`. The amount of data accepted by the terminal driver varies from system to system. The results will also vary depending on how you are logged in to the system: on the system console, on a hard-wired terminal, on a network connection using a pseudo terminal. If you are running a windowing system on your terminal, you are also going through a pseudo terminal device. □

In this example, the program issues more than 9,000 `write` calls, even though only 500 are needed to output the data. The rest just return an error. This type of loop, called *polling*, is a waste of CPU time on a multiuser system. In Section 14.4, we'll see that I/O multiplexing with a nonblocking descriptor is a more efficient way to do this.

Sometimes, we can avoid using nonblocking I/O by designing our applications to use multiple threads (see Chapter 11). We can allow individual threads to block in I/O calls if we can continue to make progress in other threads. This can sometimes simplify the design, as we shall see in Chapter 21; at other times, however, the overhead of synchronization can add more complexity than is saved from using threads.

14.3 Record Locking

What happens when two people edit the same file at the same time? In most UNIX systems, the final state of the file corresponds to the last process that wrote the file. In some applications, however, such as a database system, a process needs to be certain that it alone is writing to a file. To provide this capability for processes that need it, commercial UNIX systems provide record locking. (In Chapter 20, we develop a database library that uses record locking.)

Record locking is the term normally used to describe the ability of a process to prevent other processes from modifying a region of a file while the first process is reading or modifying that portion of the file. Under the UNIX System, "record" is a misnomer; the UNIX kernel does not have a notion of records in a file. A better term is *byte-range locking*, given that it is a range of a file (possibly the entire file) that is locked.

History

One of the criticisms of early UNIX systems was that they couldn't be used to run database systems, because they did not support locking portions of files. As UNIX systems found their way into business computing environments, various groups added support for record locking (differently, of course).

Early Berkeley releases supported only the `flock` function. This function locks only entire files, not regions of a file.

Record locking was added to System V Release 3 through the `fcntl` function. The `lockf` function was built on top of this, providing a simplified interface. These functions allowed callers to lock arbitrary byte ranges in a file, ranging from the entire file down to a single byte within the file.

POSIX.1 chose to standardize on the `fcntl` approach. Figure 14.2 shows the forms of record locking provided by various systems. Note that the Single UNIX Specification includes `lockf` in the XSI option.

System	Advisory	Mandatory	`fcntl`	`lockf`	`flock`
SUS	•		•	XSI	
FreeBSD 8.0	•		•	•	•
Linux 3.2.0	•	•	•	•	•
Mac OS X 10.6.8	•		•	•	•
Solaris 10	•	•	•	•	•

Figure 14.2 Forms of record locking supported by various UNIX systems

We describe the difference between advisory locking and mandatory locking later in this section. In this text, we describe only the POSIX.1 `fcntl` locking.

> Record locking was originally added to Version 7 in 1980 by John Bass. The system call entry into the kernel was a function named `locking`. This function provided mandatory record locking and propagated through many versions of System III. Xenix systems picked up this function, and some Intel-based System V derivatives, such as OpenServer 5, continued to support it in a Xenix-compatibility library.

`fcntl` Record Locking

Let's repeat the prototype for the `fcntl` function from Section 3.14.

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* struct flock *flockptr */ );
```

Returns: depends on *cmd* if OK (see following), –1 on error

For record locking, *cmd* is `F_GETLK`, `F_SETLK`, or `F_SETLKW`. The third argument (which we'll call *flockptr*) is a pointer to an `flock` structure.

```
struct flock {
  short  l_type;   /* F_RDLCK, F_WRLCK, or F_UNLCK */
  short  l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END */
  off_t  l_start;  /* offset in bytes, relative to l_whence */
  off_t  l_len;    /* length, in bytes; 0 means lock to EOF */
  pid_t  l_pid;    /* returned with F_GETLK */
};
```

This structure describes

- The type of lock desired: `F_RDLCK` (a shared read lock), `F_WRLCK` (an exclusive write lock), or `F_UNLCK` (unlocking a region)
- The starting byte offset of the region being locked or unlocked (`l_start` and `l_whence`)
- The size of the region in bytes (`l_len`)
- The ID (`l_pid`) of the process holding the lock that can block the current process (returned by `F_GETLK` only)

Numerous rules apply to the specification of the region to be locked or unlocked.

- The two elements that specify the starting offset of the region are similar to the last two arguments of the `lseek` function (Section 3.6). Indeed, the `l_whence` member is specified as `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`.
- Locks can start and extend beyond the current end of file, but cannot start or extend before the beginning of the file.
- If `l_len` is 0, it means that the lock extends to the largest possible offset of the file. This allows us to lock a region starting anywhere in the file, up through and including any data that is appended to the file. (We don't have to try to guess how many bytes might be appended to the file.)
- To lock the entire file, we set `l_start` and `l_whence` to point to the beginning of the file and specify a length (`l_len`) of 0. (There are several ways to specify the beginning of the file, but most applications specify `l_start` as 0 and `l_whence` as `SEEK_SET`.)

We previously mentioned two types of locks: a shared read lock (`l_type` of `F_RDLCK`) and an exclusive write lock (`F_WRLCK`). The basic rule is that any number of processes can have a shared read lock on a given byte, but only one process can have an exclusive write lock on a given byte. Furthermore, if there are one or more read locks on a byte, there can't be any write locks on that byte; if there is an exclusive write lock on a byte, there can't be any read locks on that byte. We show this compatibility rule in Figure 14.3.

		Request for	
		read lock	write lock
Region currently has	no locks	OK	OK
	one or more read locks	OK	denied
	one write lock	denied	denied

Figure 14.3 Compatibility between different lock types

The compatibility rule applies to lock requests made from different processes, not to multiple lock requests made by a single process. If a process has an existing lock on a range of a file, a subsequent attempt to place a lock on the same range by the same process will replace the existing lock with the new one. Thus, if a process has a write lock on bytes 16–32 of a file and then tries to place a read lock on bytes 16–32, the request will succeed, and the write lock will be replaced by a read lock.

To obtain a read lock, the descriptor must be open for reading; to obtain a write lock, the descriptor must be open for writing.

We can now describe the three commands for the `fcntl` function.

`F_GETLK` Determine whether the lock described by *flockptr* is blocked by some other lock. If a lock exists that would prevent ours from being created, the information on that existing lock overwrites the information pointed to by *flockptr*. If no lock exists that would prevent ours from being created, the structure pointed to by *flockptr* is left unchanged except for the `l_type` member, which is set to `F_UNLCK`.

`F_SETLK` Set the lock described by *flockptr*. If we are trying to obtain a read lock (`l_type` of `F_RDLCK`) or a write lock (`l_type` of `F_WRLCK`) and the compatibility rule prevents the system from giving us the lock (Figure 14.3), `fcntl` returns immediately with `errno` set to either `EACCES` or `EAGAIN`.

> Although POSIX allows an implementation to return either error code, all four implementations described in this text return `EAGAIN` if the locking request cannot be satisfied.

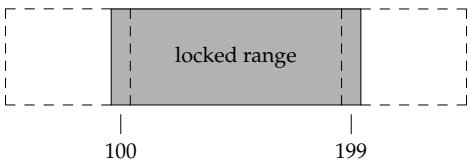
This command is also used to clear the lock described by *flockptr* (`l_type` of `F_UNLCK`).

`F_SETLKW` This command is a blocking version of `F_SETLK`. (The `W` in the command name means *wait*.) If the requested read lock or write lock cannot be granted because another process currently has some part of the requested region locked, the calling process is put to sleep. The process wakes up either when the lock becomes available or when interrupted by a signal.

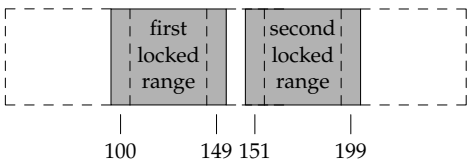
Be aware that testing for a lock with `F_GETLK` and then trying to obtain that lock with `F_SETLK` or `F_SETLKW` is not an atomic operation. We have no guarantee that, between the two `fcntl` calls, some other process won't come in and obtain the same lock. If we don't want to block while waiting for a lock to become available to us, we must handle the possible error returns from `F_SETLK`.

> Note that POSIX.1 doesn't specify what happens when one process read locks a range of a file, a second process blocks while trying to get a write lock on the same range, and a third processes then attempts to get another read lock on the range. If the third process is allowed to place a read lock on the range just because the range is already read locked, then the implementation might starve processes with pending write locks. Thus, as additional requests to read lock the same range arrive, the time that the process with the pending write-lock request has to wait is extended. If the read-lock requests arrive quickly enough without a lull in the arrival rate, then the writer could wait for a long time.

When setting or releasing a lock on a file, the system combines or splits adjacent areas as required. For example, if we lock bytes 100 through 199 and then unlock byte 150, the kernel still maintains the locks on bytes 100 through 149 and bytes 151 through 199. Figure 14.4 illustrates the byte-range locks in this situation.



File after locking bytes 100 through 199



File after unlocking byte 150

Figure 14.4 File byte-range lock diagram

If we were to lock byte 150, the system would coalesce the adjacent locked regions into a single region from byte 100 through 199. The resulting picture would be the first diagram in Figure 14.4, the same as when we started.

Example — Requesting and Releasing a Lock

To save ourselves from having to allocate an `flock` structure and fill in all the elements each time, the function `lock_reg` in Figure 14.5 handles all these details.

```
#include "apue.h"
#include <fcntl.h>

int
lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
{
    struct flock    lock;

    lock.l_type = type;     /* F_RDLCK, F_WRLCK, F_UNLCK */
    lock.l_start = offset;  /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;       /* #bytes (0 means to EOF) */

    return(fcntl(fd, cmd, &lock));
}
```

Figure 14.5 Function to lock or unlock a region of a file

Since most locking calls are to lock or unlock a region (the command `F_GETLK` is rarely used), we normally use one of the following five macros, which are defined in `apue.h` (Appendix B).

```
#define read_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))
```

We have purposely defined the first three arguments to these macros in the same order as the `lseek` function. □

Example — Testing for a Lock

Figure 14.6 defines the function `lock_test` that we'll use to test for a lock.

```
#include "apue.h"
#include <fcntl.h>

pid_t
lock_test(int fd, int type, off_t offset, int whence, off_t len)
{
```

```
    struct flock    lock;

    lock.l_type = type;     /* F_RDLCK or F_WRLCK */
    lock.l_start = offset;  /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;       /* #bytes (0 means to EOF) */

    if (fcntl(fd, F_GETLK, &lock) < 0)
        err_sys("fcntl error");

    if (lock.l_type == F_UNLCK)
        return(0);      /* false, region isn't locked by another proc */
    return(lock.l_pid); /* true, return pid of lock owner */
}
```

Figure 14.6 Function to test for a locking condition

If a lock exists that would block the request specified by the arguments, this function returns the process ID of the process holding the lock. Otherwise, the function returns 0 (false). We normally call this function from the following two macros (defined in `apue.h`):

```
#define is_read_lockable(fd, offset, whence, len) \
          (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
          (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)
```

Note that the `lock_test` function can't be used by a process to see whether it is currently holding a portion of a file locked. The definition of the `F_GETLK` command states that the information returned applies to an existing lock that would prevent us from creating our own lock. Since the `F_SETLK` and `F_SETLKW` commands always replace a process's existing lock if it exists, we can never block on our own lock; thus, the `F_GETLK` command will never report our own lock. □

Example — Deadlock

Deadlock occurs when two processes are each waiting for a resource that the other has locked. The potential for deadlock exists if a process that controls a locked region is put to sleep when it tries to lock another region that is controlled by a different process.

Figure 14.7 shows an example of deadlock. The child locks byte 0 and the parent locks byte 1. Each then tries to lock the other's already locked byte. We use the parent–child synchronization routines from Section 8.9 (`TELL_xxx` and `WAIT_xxx`) so that each process can wait for the other to obtain its lock.

```
#include "apue.h"
#include <fcntl.h>

static void
lockabyte(const char *name, int fd, off_t offset)
```

```
{
    if (writew_lock(fd, offset, SEEK_SET, 1) < 0)
        err_sys("%s: writew_lock error", name);
    printf("%s: got the lock, byte %lld\n", name, (long long)offset);
}

int
main(void)
{
    int     fd;
    pid_t   pid;

    /*
     * Create a file and write two bytes to it.
     */
    if ((fd = creat("templock", FILE_MODE)) < 0)
        err_sys("creat error");
    if (write(fd, "ab", 2) != 2)
        err_sys("write error");

    TELL_WAIT();
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {          /* child */
        lockabyte("child", fd, 0);
        TELL_PARENT(getppid());
        WAIT_PARENT();
        lockabyte("child", fd, 1);
    } else {                        /* parent */
        lockabyte("parent", fd, 1);
        TELL_CHILD(pid);
        WAIT_CHILD();
        lockabyte("parent", fd, 0);
    }
    exit(0);
}
```

Figure 14.7 Example of deadlock detection

Running the program in Figure 14.7 gives us

```
$ ./a.out
parent: got the lock, byte 1
child: got the lock, byte 0
parent: writew_lock error: Resource deadlock avoided
child: got the lock, byte 1
```

When a deadlock is detected, the kernel has to choose one process to receive the error return. In this example, the parent was chosen, but this is an implementation detail. On some systems, the child always receives the error. On other systems, the parent always gets the error. On some systems, you might even see the errors split between the child and the parent as multiple lock attempts are made. □

Implied Inheritance and Release of Locks

Three rules govern the automatic inheritance and release of record locks.

1. Locks are associated with a process and a file. This has two implications. The first is obvious: when a process terminates, all its locks are released. The second is far from obvious: whenever a descriptor is closed, any locks on the file referenced by that descriptor for that process are released. This means that if we make the calls

   ```
   fd1 = open(pathname, ...);
   read_lock(fd1, ...);
   fd2 = dup(fd1);
   close(fd2);
   ```

   after the `close(fd2)`, the lock that was obtained on `fd1` is released. The same thing would happen if we replaced the `dup` with `open`, as in

   ```
   fd1 = open(pathname, ...);
   read_lock(fd1, ...);
   fd2 = open(pathname, ...)
   close(fd2);
   ```

   to open the same file on another descriptor.
2. Locks are never inherited by the child across a `fork`. This means that if a process obtains a lock and then calls `fork`, the child is considered another process with regard to the lock that was obtained by the parent. The child has to call `fcntl` to obtain its own locks on any descriptors that were inherited across the `fork`. This constraint makes sense because locks are meant to prevent multiple processes from writing to the same file at the same time. If the child inherited locks across a `fork`, both the parent and the child could write to the same file at the same time.
3. Locks are inherited by a new program across an `exec`. Note, however, that if the close-on-exec flag is set for a file descriptor, all locks for the underlying file are released when the descriptor is closed as part of an `exec`.

FreeBSD Implementation

Let's take a brief look at the data structures used in the FreeBSD implementation. This should help clarify rule 1, which states that locks are associated with a process and a file.

Consider a process that executes the following statements (ignoring error returns):

```
fd1 = open(pathname, ...);
write_lock(fd1, 0, SEEK_SET, 1);    /* parent write locks byte 0 */
if ((pid = fork()) > 0) {           /* parent */
    fd2 = dup(fd1);
    fd3 = open(pathname, ...);
} else if (pid == 0) {
    read_lock(fd1, 1, SEEK_SET, 1); /* child read locks byte 1 */
}
pause();
```

Figure 14.8 shows the resulting data structures after both the parent and the child have paused.

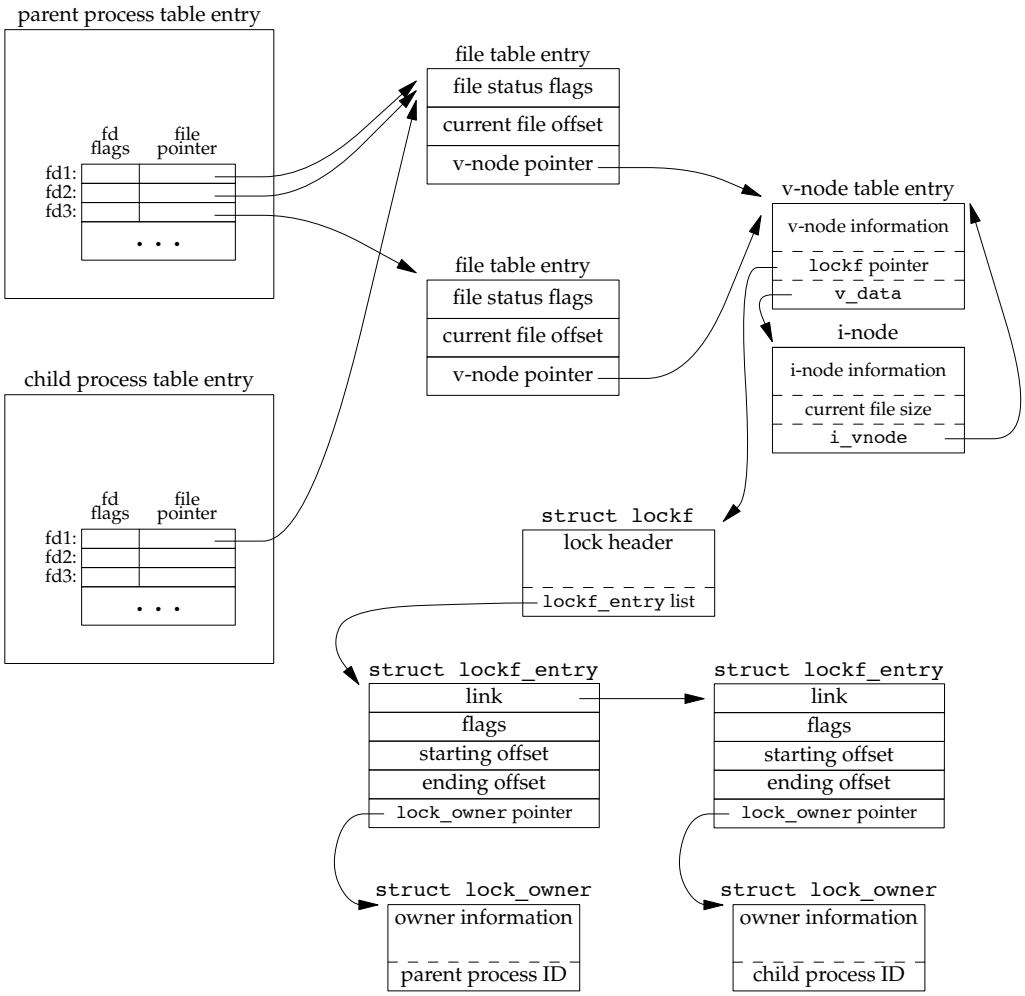


Figure 14.8 The FreeBSD data structures for record locking

We've shown the data structures that result from the `open`, `fork`, and `dup` calls earlier (Figures 3.9 and 8.2). What is new here are the `lockf` structures that are linked together from the i-node structure. Each `lockf` structure describes one locked region (defined by an offset and length) for a given process. We show two of these structures: one for the parent's call to `write_lock` and one for the child's call to `read_lock`. Each structure contains the corresponding process ID.

In the parent, closing any one of `fd1`, `fd2`, or `fd3` causes the parent's lock to be released. When any one of these three file descriptors is closed, the kernel goes through

the linked list of locks for the corresponding i-node and releases the locks held by the calling process. The kernel can't tell (and doesn't care) which descriptor of the three was used by the parent to obtain the lock.

Example

In the program in Figure 13.6, we saw how a daemon can use a lock on a file to ensure that only one copy of the daemon is running. Figure 14.9 shows the implementation of the `lockfile` function used by the daemon to place a write lock on a file.

```
#include <unistd.h>
#include <fcntl.h>

int
lockfile(int fd)
{
    struct flock fl;

    fl.l_type = F_WRLCK;
    fl.l_start = 0;
    fl.l_whence = SEEK_SET;
    fl.l_len = 0;
    return(fcntl(fd, F_SETLK, &fl));
}
```

Figure 14.9 Place a write lock on an entire file

Alternatively, we could define the `lockfile` function in terms of the `write_lock` function:

```
#define lockfile(fd) write_lock((fd), 0, SEEK_SET, 0)
```

□

Locks at End of File

We need to use caution when locking or unlocking byte ranges relative to the end of file. Most implementations convert an `l_whence` value of `SEEK_CUR` or `SEEK_END` into an absolute file offset, using `l_start` and the file's current position or current length. Often, however, we need to specify a lock relative to the file's current length, but we can't call `fstat` to obtain the current file size, since we don't have a lock on the file. (There's a chance that another process could change the file's length between the call to `fstat` and the lock call.)

Consider the following sequence of steps:

```
writew_lock(fd, 0, SEEK_END, 0);
write(fd, buf, 1);
un_lock(fd, 0, SEEK_END);
write(fd, buf, 1);
```

This sequence of code might not do what you expect. It obtains a write lock from the current end of the file onward, covering any future data we might append to the file.

Assuming that we are at end of file when we perform the first `write`, this operation will extend the file by one byte, and that byte will be locked. The unlock operation that follows has the effect of removing the locks for future writes that append data to the file, but it leaves a lock on the last byte in the file. When the second write occurs, the end of file is extended by one byte, but this byte is not locked. The state of the file locks for this sequence of steps is shown in Figure 14.10.

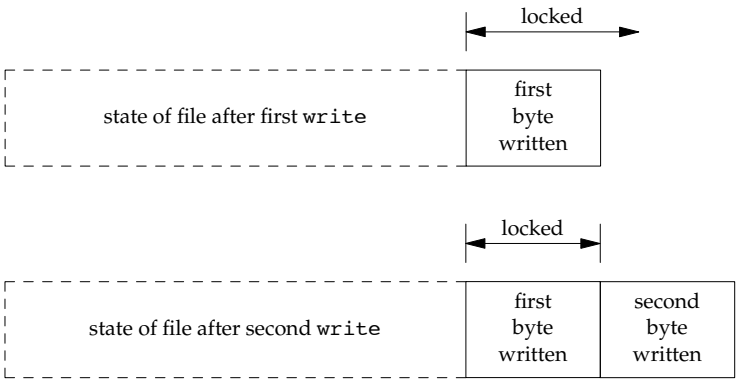


Figure 14.10 File range lock diagram

When a portion of a file is locked, the kernel converts the offset specified into an absolute file offset. In addition to specifying an absolute file offset (`SEEK_SET`), `fcntl` allows us to specify this offset relative to a point in the file: current (`SEEK_CUR`) or end of file (`SEEK_END`). The kernel needs to remember the locks independent of the current file offset or end of file, because the current offset and end of file can change, and changes to these attributes shouldn't affect the state of existing locks.

If we intended to remove the lock covering the byte we wrote in the first write, we could have specified the length as –1. Negative length values represent the bytes before the specified offset.

Advisory versus Mandatory Locking

Consider a library of database access routines. If all the functions in the library handle record locking in a consistent way, then we say that any set of processes using these functions to access a database are *cooperating processes*. It is feasible for these database access functions to use advisory locking if they are the only ones being used to access the database. But advisory locking doesn't prevent some other process that has write permission for the database file from writing whatever it wants to the database file. This rogue process would be an uncooperating process, since it's not using the accepted method (the library of database functions) to access the database.

Mandatory locking causes the kernel to check every `open`, `read`, and `write` to verify that the calling process isn't violating a lock on the file being accessed. Mandatory locking is sometimes called *enforcement-mode locking*.

We saw in Figure 14.2 that Linux 3.2.0 and Solaris 10 provide mandatory record locking, but FreeBSD 8.0 and Mac OS X 10.6.8 do not. Mandatory record locking is not part of the Single UNIX Specification. On Linux, if you want mandatory locking, you need to enable it on a per file system basis by using the `-o mand` option to the `mount` command.

Mandatory locking is enabled for a particular file by turning on the set-group-ID bit and turning off the group-execute bit. (Recall Figure 4.12.) Since the set-group-ID bit makes no sense when the group-execute bit is off, the designers of SVR3 chose this way to specify that the locking for a file is to be mandatory locking and not advisory locking.

What happens to a process that tries to `read` or `write` a file that has mandatory locking enabled and that part of the file is currently locked by another process? The answer depends on the type of operation (`read` or `write`), the type of lock held by the other process (read lock or write lock), and whether the descriptor for the `read` or `write` is nonblocking. Figure 14.11 shows the eight possibilities.

Type of existing lock on region held by other process	Blocking descriptor, tries to		Nonblocking descriptor, tries to	
	`read`	`write`	`read`	`write`
read lock	OK	blocks	OK	`EAGAIN`
write lock	blocks	blocks	`EAGAIN`	`EAGAIN`

Figure 14.11 Effect of mandatory locking on `read`s and `write`s by other processes

In addition to the `read` and `write` functions in Figure 14.11, the `open` function can be affected by mandatory record locks held by another process. Normally, `open` succeeds, even if the file being opened has outstanding mandatory record locks. The next `read` or `write` follows the rules listed in Figure 14.11. But if the file being opened has outstanding mandatory record locks (either read locks or write locks), and if the flags in the call to `open` specify either `O_TRUNC` or `O_CREAT`, then `open` returns an error of `EAGAIN` immediately, regardless of whether `O_NONBLOCK` is specified.

Only Solaris treats the `O_CREAT` flag as an error case. Linux allows the `O_CREAT` flag to be specified when opening a file with an outstanding mandatory lock. Generating the `open` error for `O_TRUNC` makes sense, because the file cannot be truncated if it is read locked or write locked by another process. Generating the error for `O_CREAT`, however, makes little sense; this flag says to create the file only if it doesn't already exist, but it has to exist to be record locked by another process.

This handling of locking conflicts with `open` can lead to surprising results. While developing the exercises in this section, a test program was run that opened a file (whose mode specified mandatory locking), established a read lock on an entire file, and then went to sleep for a while. (Recall from Figure 14.11 that a read lock should prevent writing to the file by other processes.) During this sleep period, the following behavior was seen in other typical UNIX System programs.

- The same file could be edited with the `ed` editor, and the results written back to disk! The mandatory record locking had no effect at all. Using the system call trace feature provided by some versions of the UNIX System, it was seen that `ed`

wrote the new contents to a temporary file, removed the original file, and then renamed the temporary file to be the original file. The mandatory record locking has no effect on the `unlink` function, which allowed this to happen.

> Under FreeBSD 8.0 and Solaris 10, we can obtain the system call trace of a process with the `truss`(1) command. Linux 3.2.0 provides the `strace`(1) command for the same purpose. Mac OS X 10.6.8 provides the `dtruss`(1m) command to trace system calls, but its use requires superuser privileges.

- The `vi` editor was never able to edit the file. It could read the file's contents, but whenever we tried to write new data to the file, `EAGAIN` was returned. If we tried to append new data to the file, the `write` blocked. This behavior from `vi` is what we expect.
- Using the Korn shell's `>` and `>>` operators to overwrite or append to the file resulted in the error "cannot create."
- Using the same two operators with the Bourne shell resulted in an error for `>`, but the `>>` operator just blocked until the mandatory lock was removed, and then proceeded. (The difference in the handling of the append operator occurs because the Korn shell `open`s the file with `O_CREAT` and `O_APPEND`, and we mentioned earlier that specifying `O_CREAT` generates an error. The Bourne shell, however, doesn't specify `O_CREAT` if the file already exists, so the `open` succeeds but the next `write` blocks.)

Results will vary, depending on the version of the operating system you are using. The bottom line, as demonstrated by this exercise, is to be wary of mandatory record locking. As seen with the `ed` example, it can be circumvented.

Mandatory record locking can also be used by a malicious user to hold a read lock on a file that is publicly readable. This can prevent anyone from writing to the file. (Of course, the file has to have mandatory record locking enabled for this to occur, which may require the user to be able to change the permission bits of the file.) Consider a database file that is world readable and has mandatory record locking enabled. If a malicious user were to hold a read lock on the entire file, the file could not be written to by other processes.

Example

We can run the program in Figure 14.12 to determine whether our system supports mandatory locking.

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    int             fd;
```

```
    pid_t           pid;
    char            buf[5];
    struct stat     statbuf;

    if (argc != 2) {
        fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(1);
    }
    if ((fd = open(argv[1], O_RDWR | O_CREAT | O_TRUNC, FILE_MODE)) < 0)
        err_sys("open error");
    if (write(fd, "abcdef", 6) != 6)
        err_sys("write error");

    /* turn on set-group-ID and turn off group-execute */
    if (fstat(fd, &statbuf) < 0)
        err_sys("fstat error");
    if (fchmod(fd, (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("fchmod error");

    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {   /* parent */
        /* write lock entire file */
        if (write_lock(fd, 0, SEEK_SET, 0) < 0)
            err_sys("write_lock error");

        TELL_CHILD(pid);

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");
    } else {                /* child */
        WAIT_PARENT();      /* wait for parent to set lock */

        set_fl(fd, O_NONBLOCK);

        /* first let's see what error we get if region is locked */
        if (read_lock(fd, 0, SEEK_SET, 0) != -1)    /* no wait */
            err_sys("child: read_lock succeeded");
        printf("read_lock of already-locked region returns %d\n",
          errno);

        /* now try to read the mandatory locked file */
        if (lseek(fd, 0, SEEK_SET) == -1)
            err_sys("lseek error");
        if (read(fd, buf, 2) < 0)
            err_ret("read failed (mandatory locking works)");
        else
            printf("read OK (no mandatory locking), buf = %2.2s\n",
              buf);
    }
    exit(0);
}
```

Figure 14.12 Determine whether mandatory locking is supported

This program creates a file and enables mandatory locking for the file. The program then splits into parent and child, with the parent obtaining a write lock on the entire file. The child first sets its descriptor to be nonblocking and then attempts to obtain a read lock on the file, expecting to get an error. This lets us see whether the system returns `EACCES` or `EAGAIN`. Next, the child rewinds the file and tries to `read` from the file. If mandatory locking is provided, the `read` should return `EACCES` or `EAGAIN` (since the descriptor is nonblocking). Otherwise, the `read` returns the data that it read. Running this program under Solaris 10 (which supports mandatory locking) gives us

```
$ ./a.out temp.lock
read_lock of already-locked region returns 11
read failed (mandatory locking works): Resource temporarily unavailable
```

If we look at either the system's headers or the `intro`(2) manual page, we see that an `errno` of 11 corresponds to `EAGAIN`. Under FreeBSD 8.0, we get

```
$ ./a.out temp.lock
read_lock of already-locked region returns 35
read OK (no mandatory locking), buf = ab
```

Here, an `errno` of 35 corresponds to `EAGAIN`. Mandatory locking is not supported. □

Example

Let's return to the first question posed in this section: what happens when two people edit the same file at the same time? The normal UNIX System text editors do not use record locking, so the answer is still that the final result of the file corresponds to the last process that wrote the file.

Some versions of the `vi` editor use advisory record locking. Even if we were using one of these versions of `vi`, it still doesn't prevent users from running another editor that doesn't use advisory record locking.

If the system provides mandatory record locking, we could modify our favorite editor to use it (if we have the editor's source code). Not having the source code for the editor, we might try the following. We write our own program that is a front end to `vi`. This program immediately calls `fork`, and the parent just waits for the child to complete. The child opens the file specified on the command line, enables mandatory locking, obtains a write lock on the entire file, and then executes `vi`. While `vi` is running, the file is write locked, so other users can't modify it. When `vi` terminates, the parent's `wait` returns and our front end terminates.

A small front-end program of this type can be written, but it doesn't work. The problem is that it is common practice for editors to read their input file and then close it. A lock is released on a file whenever a descriptor that references that file is closed. As a result, when the editor closes the file after reading its contents, the lock is gone. There is no way to prevent this from happening in the front-end program. □

We'll use record locking in Chapter 20 in our database library to provide concurrent access to multiple processes. We'll also provide some timing measurements to see how record locking affects a process.

14.4 I/O Multiplexing

When we read from one descriptor and write to another, we can use blocking I/O in a loop, such as

```
while ((n = read(STDIN_FILENO, buf, BUFSIZ)) > 0)
    if (write(STDOUT_FILENO, buf, n) != n)
        err_sys("write error");
```

We see this form of blocking I/O over and over again. What if we have to read from two descriptors? In this case, we can't do a blocking `read` on either descriptor, as data may appear on one descriptor while we're blocked in a `read` on the other. A different technique is required to handle this case.

Let's look at the structure of the `telnet`(1) command. In this program, we read from the terminal (standard input) and write to a network connection, and we read from the network connection and write to the terminal (standard output). At the other end of the network connection, the `telnetd` daemon reads what we typed and presents it to a shell as if we were logged in to the remote machine. The `telnetd` daemon sends any output generated by the commands we type back to us through the `telnet` command, to be displayed on our terminal. Figure 14.13 shows a picture of this arrangement.



Figure 14.13 Overview of `telnet` program

The `telnet` process has two inputs and two outputs. We can't do a blocking `read` on either of the inputs, as we never know which input will have data for us.

One way to handle this particular problem is to divide the process in two pieces (using `fork`), with each half handling one direction of data. We show this in Figure 14.14. (The `cu`(1) command provided with System V's `uucp` communication package was structured like this.)

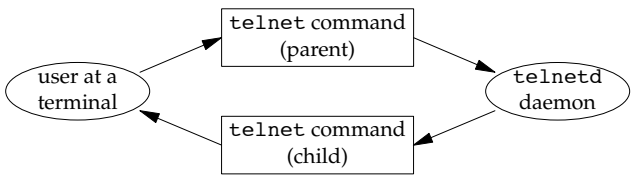


Figure 14.14 The `telnet` program using two processes

If we use two processes, we can let each process do a blocking `read`. But this leads to a problem when the operation terminates. If an end of file is received by the child (the

network connection is disconnected by the `telnetd` daemon), then the child terminates and the parent is notified by the `SIGCHLD` signal. But if the parent terminates (the user enters an end-of-file character at the terminal), then the parent has to tell the child to stop. We can use a signal for this (`SIGUSR1`, for example), but it does complicate the program somewhat.

Instead of two processes, we could use two threads in a single process. This avoids the termination complexity, but requires that we deal with synchronization between the threads, which could add more complexity than it saves.

We could use nonblocking I/O in a single process by setting both descriptors to be nonblocking and issuing a `read` on the first descriptor. If data is present, we read it and process it. If there is no data to read, the call returns immediately. We then do the same thing with the second descriptor. After this, we wait for some amount of time (a few seconds, perhaps) and then try to read from the first descriptor again. This type of loop is called *polling*. The problem is that it wastes CPU time. Most of the time, there won't be data to read, so we waste time performing the `read` system calls. We also have to guess how long to wait each time around the loop. Although it works on any system that supports nonblocking I/O, polling should be avoided on a multitasking system.

Another technique is called *asynchronous I/O*. With this technique, we tell the kernel to notify us with a signal when a descriptor is ready for I/O. There are two problems with this approach. First, although systems provide their own limited forms of asynchronous I/O, POSIX chose to standardize a different set of interfaces, so portability can be an issue. (In the past, POSIX asynchronous I/O was an optional facility in the Single UNIX Specification, but these interfaces are required as of SUSv4.) System V provides the `SIGPOLL` signal to support a limited form of asynchronous I/O, but this signal works only if the descriptor refers to a STREAMS device. BSD has a similar signal, `SIGIO`, but it has similar limitations: it works only on descriptors that refer to terminal devices or networks.

The second problem with this technique is that the limited forms use only one signal per process (`SIGPOLL` or `SIGIO`). If we enable this signal for two descriptors (in the example we've been talking about, reading from two descriptors), the occurrence of the signal doesn't tell us which descriptor is ready. Although the POSIX.1 asynchronous I/O interfaces allow us to select which signal to use for notification, the number of signals we can use is still far less than the number of possible open file descriptors. To determine which descriptor is ready, we would need to set each file descriptor to nonblocking mode and try the descriptors in sequence. We discuss asynchronous I/O in Section 14.5.

A better technique is to use *I/O multiplexing*. To do this, we build a list of the descriptors that we are interested in (usually more than one descriptor) and call a function that doesn't return until one of the descriptors is ready for I/O. Three functions—`poll`, `pselect`, and `select`—allow us to perform I/O multiplexing. On return from these functions, we are told which descriptors are ready for I/O.

> POSIX specifies that `<sys/select.h>` be included to pull the information for `select` into your program. Older systems require that you include `<sys/types.h>`, `<sys/time.h>`, and `<unistd.h>`. Check the `select` manual page to see what your system supports.

I/O multiplexing was provided with the `select` function in 4.2BSD. This function has always worked with any descriptor, although its main use has been for terminal I/O and network I/O. SVR3 added the `poll` function when the STREAMS mechanism was added. Initially, `poll` worked only with STREAMS devices. In SVR4, support was added to allow `poll` to work on any descriptor.

14.4.1 `select` and `pselect` Functions

The `select` function lets us do I/O multiplexing under all POSIX-compatible platforms. The arguments we pass to `select` tell the kernel

- Which descriptors we're interested in.
- Which conditions we're interested in for each descriptor. (Do we want to read from a given descriptor? Do we want to write to a given descriptor? Are we interested in an exception condition for a given descriptor?)
- How long we want to wait. (We can wait forever, wait a fixed amount of time, or not wait at all.)

On the return from `select`, the kernel tells us

- The total count of the number of descriptors that are ready
- Which descriptors are ready for each of the three conditions (read, write, or exception condition)

With this return information, we can call the appropriate I/O function (usually `read` or `write`) and know that the function won't block.

```
#include <sys/select.h>

int select(int maxfdp1, fd_set *restrict readfds,
           fd_set *restrict writefds, fd_set *restrict exceptfds,
           struct timeval *restrict tvptr);
```

Returns: count of ready descriptors, 0 on timeout, –1 on error

Let's look at the last argument first. It specifies how long we want to wait in terms of seconds and microseconds (recall Section 4.20). There are three conditions.

tvptr `== NULL`

Wait forever. This infinite wait can be interrupted if we catch a signal. Return is made when one of the specified descriptors is ready or when a signal is caught. If a signal is caught, `select` returns –1 with `errno` set to `EINTR`.

tvptr->tv_sec `==` 0 `&&` *tvptr->tv_usec* `==` 0

Don't wait at all. All the specified descriptors are tested, and return is made immediately. This is a way to poll the system to find out the status of multiple descriptors without blocking in the `select` function.

tvptr->tv_sec `!=` 0 || *tvptr->tv_usec* `!=` 0

Wait the specified number of seconds and microseconds. Return is made when one of the specified descriptors is ready or when the timeout value expires. If the timeout expires before any of the descriptors is ready, the return value is 0. (If the system doesn't provide microsecond resolution, the *tvptr->tv_usec* value is rounded up to the nearest supported value.) As with the first condition, this wait can also be interrupted by a caught signal.

POSIX.1 allows an implementation to modify the `timeval` structure, so after `select` returns, you can't rely on the structure containing the same values it did before calling `select`. FreeBSD 8.0, Mac OS X 10.6.8, and Solaris 10 all leave the structure unchanged, but Linux 3.2.0 will update it with the time remaining if `select` returns before the timeout value expires.

The middle three arguments—*readfds*, *writefds*, and *exceptfds*—are pointers to *descriptor sets*. These three sets specify which descriptors we're interested in and for which conditions (readable, writable, or an exception condition). A descriptor set is stored in an `fd_set` data type. This data type is chosen by the implementation so that it can hold one bit for each possible descriptor. We can consider it to be just a big array of bits, as shown in Figure 14.15.

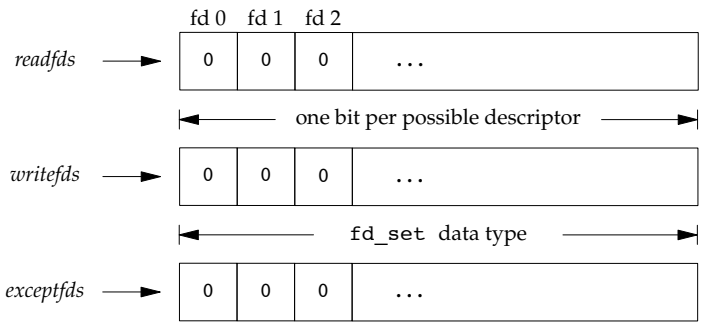


Figure 14.15 Specifying the read, write, and exception descriptors for `select`

The only thing we can do with the `fd_set` data type is allocate a variable of this type, assign a variable of this type to another variable of the same type, or use one of the following four functions on a variable of this type.

```
#include <sys/select.h>

int FD_ISSET(int fd, fd_set *fdset);

                                        Returns: nonzero if fd is in set, 0 otherwise

void FD_CLR(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

These interfaces can be implemented as either macros or functions. An `fd_set` is set to all zero bits by calling `FD_ZERO`. To turn on a single bit in a set, we use `FD_SET`. We can clear a single bit by calling `FD_CLR`. Finally, we can test whether a given bit is turned on in the set with `FD_ISSET`.

After declaring a descriptor set, we must zero the set using `FD_ZERO`. We then set bits in the set for each descriptor that we're interested in, as in

```
fd_set   rset;
int      fd;

FD_ZERO(&rset);
FD_SET(fd, &rset);
FD_SET(STDIN_FILENO, &rset);
```

On return from `select`, we can test whether a given bit in the set is still on using `FD_ISSET`:

```
if (FD_ISSET(fd, &rset)) {
      .
      .
      .
}
```

Any (or all) of the middle three arguments to `select` (the pointers to the descriptor sets) can be null pointers if we're not interested in that condition. If all three pointers are `NULL`, then we have a higher-precision timer than is provided by `sleep`. (Recall from Section 10.19 that `sleep` waits for an integral number of seconds. With `select`, we can wait for intervals less than one second; the actual resolution depends on the system's clock.) Exercise 14.5 shows such a function.

The first argument to `select`, *maxfdp1*, stands for "maximum file descriptor plus 1." We calculate the highest descriptor that we're interested in, considering all three of the descriptor sets, add 1, and that's the first argument. We could just set the first argument to `FD_SETSIZE`, a constant in `<sys/select.h>` that specifies the maximum number of descriptors (often 1,024), but this value is too large for most applications. Indeed, most applications probably use between 3 and 10 descriptors. (Some applications need many more descriptors, but these UNIX programs are atypical.) By specifying the highest descriptor that we're interested in, we can prevent the kernel from going through hundreds of unused bits in the three descriptor sets, looking for bits that are turned on.

As an example, Figure 14.16 shows what two descriptor sets look like if we write

```
fd_set  readset, writeset;

FD_ZERO(&readset);
FD_ZERO(&writeset);
FD_SET(0, &readset);
FD_SET(3, &readset);
FD_SET(1, &writeset);
FD_SET(2, &writeset);
select(4, &readset, &writeset, NULL, NULL);
```

The reason we have to add 1 to the maximum descriptor number is that descriptors start at 0, and the first argument is really a count of the number of descriptors to check (starting with descriptor 0).

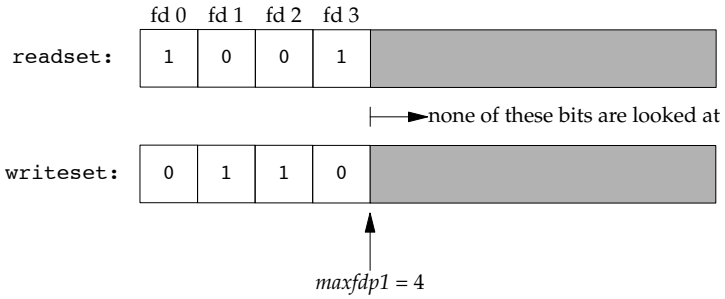


Figure 14.16 Example descriptor sets for `select`

There are three possible return values from `select`.

1. A return value of –1 means that an error occurred. This can happen, for example, if a signal is caught before any of the specified descriptors are ready. In this case, none of the descriptor sets will be modified.
2. A return value of 0 means that no descriptors are ready. This happens if the time limit expires before any of the descriptors are ready. When this happens, all the descriptor sets will be zeroed out.
3. A positive return value specifies the number of descriptors that are ready. This value is the sum of the descriptors ready in all three sets, so if the same descriptor is ready to be read *and* written, it will be counted twice in the return value. The only bits left on in the three descriptor sets are the bits corresponding to the descriptors that are ready.

We now need to be more specific about what ''ready'' means.

- A descriptor in the read set (*readfds*) is considered ready if a `read` from that descriptor won't block.
- A descriptor in the write set (*writefds*) is considered ready if a `write` to that descriptor won't block.
- A descriptor in the exception set (*exceptfds*) is considered ready if an exception condition is pending on that descriptor. Currently, an exception condition corresponds to either the arrival of out-of-band data on a network connection or certain conditions occurring on a pseudo terminal that has been placed into packet mode. (Section 15.10 of Stevens [1990] describes this latter condition.)
- File descriptors for regular files always return ready for reading, writing, and exception conditions.

It is important to realize that whether a descriptor is blocking or not doesn't affect whether `select` blocks. That is, if we have a nonblocking descriptor that we want to read from and we call `select` with a timeout value of 5 seconds, `select` will block for up to 5 seconds. Similarly, if we specify an infinite timeout, `select` blocks until data is ready for the descriptor or until a signal is caught.

If we encounter the end of file on a descriptor, that descriptor is considered readable by `select`. We then call `read` and it returns 0—the way to signify end of file on UNIX systems. (Many people incorrectly assume that `select` indicates an exception condition on a descriptor when the end of file is reached.)

POSIX.1 also defines a variant of `select` called `pselect`.

```
#include <sys/select.h>

int pselect(int maxfdp1, fd_set *restrict readfds,
            fd_set *restrict writefds, fd_set *restrict exceptfds,
            const struct timespec *restrict tsptr,
            const sigset_t *restrict sigmask);
```

Returns: count of ready descriptors, 0 on timeout, –1 on error

The `pselect` function is identical to `select`, with the following exceptions.

- The timeout value for `select` is specified by a `timeval` structure, but for `pselect`, a `timespec` structure is used. (Recall the definition of the `timespec` structure in Section 4.2.) Instead of seconds and microseconds, the `timespec` structure represents the timeout value in seconds and nanoseconds. This provides a higher-resolution timeout if the platform supports that fine a level of granularity.
- The timeout value for `pselect` is declared `const`, and we are guaranteed that its value will not change as a result of calling `pselect`.
- An optional signal mask argument is available with `pselect`. If *sigmask* is `NULL`, `pselect` behaves as `select` does with respect to signals. Otherwise, *sigmask* points to a signal mask that is atomically installed when `pselect` is called. On return, the previous signal mask is restored.

14.4.2 `poll` Function

The `poll` function is similar to `select`, but the programmer interface is different. This function was originally introduced in System V to support the STREAMS subsystem, but we are able to use it with any type of file descriptor.

```
#include <poll.h>

int poll(struct pollfd fdarray[], nfds_t nfds, int timeout);
```

Returns: count of ready descriptors, 0 on timeout, –1 on error

With poll, instead of building a set of descriptors for each condition (readability, writability, and exception condition) as we did with select, we build an array of pollfd structures, with each array element specifying a descriptor number and the conditions that we're interested in for that descriptor:

```
struct pollfd {
  int    fd;       /* file descriptor to check, or <0 to ignore */
  short  events;   /* events of interest on fd */
  short  revents;  /* events that occurred on fd */
};
```

The number of elements in the *fdarray* array is specified by *nfds*.

> Historically, there have been differences in how the *nfds* parameter was declared. SVR3 specified the number of elements in the array as an unsigned long, which seems excessive. In the SVR4 manual [AT&T 1990d], the prototype for poll showed the data type of the second argument as size_t. (Recall the primitive system data types from Figure 2.21.) But the actual prototype in the <poll.h> header still showed the second argument as an unsigned long. The Single UNIX Specification defines the new type nfds_t to allow the implementation to select the appropriate type and hide the details from applications. Note that this type has to be large enough to hold an integer, since the return value represents the number of entries in the array with satisfied events.
>
> The SVID corresponding to SVR4 [AT&T 1989] showed the first argument to poll as struct pollfd *fdarray*[], whereas the SVR4 manual page [AT&T 1990d] showed this argument as struct pollfd **fdarray*. In the C language, both declarations are equivalent. We use the first declaration to reiterate that fdarray points to an array of structures and not a pointer to a single structure.

To tell the kernel which events we're interested in for each descriptor, we have to set the events member of each array element to one or more of the values in Figure 14.17. On return, the revents member is set by the kernel, thereby specifying which events have occurred for each descriptor. (Note that poll doesn't change the events member. This behavior differs from that of select, which modifies its arguments to indicate what is ready.)

Name	Input to events?	Result from revents?	Description
POLLIN	•	•	Data other than high priority data can be read without blocking (equivalent to POLLRDNORM | POLLRDBAND).
POLLRDNORM	•	•	Normal data can be read without blocking.
POLLRDBAND	•	•	Priority data can be read without blocking.
POLLPRI	•	•	High-priority data can be read without blocking.
POLLOUT	•	•	Normal data can be written without blocking.
POLLWRNORM	•	•	Same as POLLOUT.
POLLWRBAND	•	•	Priority data can be written without blocking.
POLLERR		•	An error has occurred.
POLLHUP		•	A hangup has occurred.
POLLNVAL		•	The descriptor does not reference an open file.

Figure 14.17 The events and revents flags for poll

The first four rows of Figure 14.17 test for readability, the next three test for writability, and the final three are for exception conditions. The last three rows in Figure 14.17 are set by the kernel on return. These three values are returned in `revents` when the condition occurs, even if they weren't specified in the `events` field.

> The poll event names containing the term *BAND* refer to priority bands in STREAMS. Refer to Rago [1993] for more information about STREAMS and priority bands.

When a descriptor is hung up (`POLLHUP`), we can no longer write to the descriptor. There may, however, still be data to be read from the descriptor.

The final argument to `poll` specifies how long we want to wait. As with `select`, there are three cases.

timeout `==` –1
: Wait forever. (Some systems define the constant `INFTIM` in `<stropts.h>` as –1.) We return when one of the specified descriptors is ready or when a signal is caught. If a signal is caught, `poll` returns –1 with `errno` set to `EINTR`.

timeout `==` 0
: Don't wait. All the specified descriptors are tested, and we return immediately. This is a way to poll the system to find out the status of multiple descriptors, without blocking in the call to `poll`.

timeout `>` 0
: Wait *timeout* milliseconds. We return when one of the specified descriptors is ready or when the *timeout* expires. If the *timeout* expires before any of the descriptors is ready, the return value is 0. (If your system doesn't provide millisecond resolution, *timeout* is rounded up to the nearest supported value.)

It is important to realize the difference between an end of file and a hangup. If we're entering data from the terminal and type the end-of-file character, `POLLIN` is turned on so we can read the end-of-file indication (`read` returns 0). `POLLHUP` is not turned on in `revents`. If we're reading from a modem and the telephone line is hung up, we'll receive the `POLLHUP` notification.

As with `select`, whether a descriptor is blocking doesn't affect whether `poll` blocks.

Interruptibility of `select` and `poll`

When the automatic restarting of interrupted system calls was introduced with 4.2BSD (Section 10.5), the `select` function was never restarted. This characteristic continues with most systems even if the `SA_RESTART` option is specified. But under SVR4, if `SA_RESTART` was specified, even `select` and `poll` were automatically restarted. To prevent this from catching us when we port software to systems derived from SVR4, we'll always use the `signal_intr` function (Figure 10.19) if the signal could interrupt a call to `select` or `poll`.

None of the implementations described in this book restart `poll` or `select` when a signal is received, even if the `SA_RESTART` flag is used.

14.5 Asynchronous I/O

Using `select` and `poll`, as described in the previous section, is a synchronous form of notification. The system doesn't tell us anything until we ask (by calling either `select` or `poll`). As we saw in Chapter 10, signals provide an asynchronous form of notification that something has happened. All systems derived from BSD and System V provide some form of asynchronous I/O, using a signal (`SIGPOLL` in System V; `SIGIO` in BSD) to notify the process that something of interest has happened on a descriptor. As mentioned in the previous section, these forms of asynchronous I/O are limited: they don't work with all file types and they allow the use of only one signal. If we enable more than one descriptor for asynchronous I/O, we cannot tell which descriptor the signal corresponds to when the signal is delivered.

Version 4 of the Single UNIX Specification moved the general asynchronous I/O mechanism from the real-time extensions to the base specification. This mechanism addresses the limitations that exist with these older asynchronous I/O facilities.

Before we look at the different ways to use asynchronous I/O, we need to discuss the costs. When we decide to use asynchronous I/O, we complicate the design of our application by choosing to juggle multiple concurrent operations. A simpler approach may be to use multiple threads, which would allow us to write the program using a synchronous model, and let the threads run asynchronous to each other.

We incur additional complexity when we use the POSIX asynchronous I/O interfaces:

- We have to worry about three sources of errors for every asynchronous operation: one associated with the submission of the operation, one associated with the result of the operation itself, and one associated with the functions used to determine the status of the asynchronous operations.
- The interfaces themselves involve a lot of extra setup and processing rules compared to their conventional counterparts, as we shall see.

  We can't really call the non-asynchronous I/O function calls "synchronous," because although they are synchronous with respect to the program flow, they aren't synchronous with respect to the I/O. Recall the discussion of synchronous writes in Chapter 3. We call a write "synchronous" if the data we write is persistent when we return from the call to the `write` function. We also can't differentiate the conventional I/O function calls from the asynchronous ones by referring to the conventional calls as the "standard" I/O calls, because this confuses them with the function calls in the standard I/O library. To avoid confusion, we'll refer to the `read` and `write` functions as the "conventional" I/O function calls in this section.

- Recovering from errors can be difficult. For example, if we submit multiple asynchronous writes and one fails, how should we proceed? If the writes are related, we might have to undo the ones that succeeded.

14.5.1 System V Asynchronous I/O

System V provides a limited form of asynchronous I/O that works only with STREAMS devices and STREAMS pipes. The System V asynchronous I/O signal is `SIGPOLL`.

To enable asynchronous I/O for a STREAMS device, we have to call `ioctl` with a second argument (*request*) of `I_SETSIG`. The third argument is an integer value formed from one or more of the constants in Figure 14.18. These constants are defined in `<stropts.h>`.

> Interfaces related to the STREAMS mechanism were marked obsolescent in SUSv4, so we don't cover them in any detail. See Rago [1993] for more information about STREAMS.

Constant	Description
`S_INPUT`	We can read data (other than high-priority data) without blocking.
`S_RDNORM`	We can read normal data without blocking.
`S_RDBAND`	We can read priority data without blocking.
`S_BANDURG`	If this constant is specified with `S_RDBAND`, the `SIGURG` signal is generated instead of `SIGPOLL` when we can read priority data without blocking.
`S_HIPRI`	We can read high-priority data without blocking.
`S_OUTPUT`	We can write normal data without blocking.
`S_WRNORM`	Same as `S_OUTPUT`.
`S_WRBAND`	We can write priority data without blocking.
`S_MSG`	The `SIGPOLL` signal message has reached the stream head.
`S_ERROR`	The stream has an error.
`S_HANGUP`	The stream has hung up.

Figure 14.18 Conditions for generating `SIGPOLL` signal

In addition to calling `ioctl` to specify the conditions that should generate the `SIGPOLL` signal, we have to establish a signal handler for this signal. Recall from Figure 10.1 that the default action for `SIGPOLL` is to terminate the process, so we should establish the signal handler before calling `ioctl`.

14.5.2 BSD Asynchronous I/O

Asynchronous I/O in BSD-derived systems is a combination of two signals: `SIGIO` and `SIGURG`. The former is the general asynchronous I/O signal, and the latter is used only to notify the process that out-of-band data has arrived on a network connection.

To receive the `SIGIO` signal, we need to perform three steps.

1. Establish a signal handler for `SIGIO`, by calling either `signal` or `sigaction`.
2. Set the process ID or process group ID to receive the signal for the descriptor, by calling `fcntl` with a command of `F_SETOWN` (Section 3.14).

3. Enable asynchronous I/O on the descriptor by calling `fcntl` with a command of `F_SETFL` to set the `O_ASYNC` file status flag (Figure 3.10).

Step 3 can be performed only on descriptors that refer to terminals or networks, which is a fundamental limitation of the BSD asynchronous I/O facility.

For the `SIGURG` signal, we need perform only steps 1 and 2. `SIGURG` is generated only for descriptors that refer to network connections that support out-of-band data, such as TCP connections.

14.5.3 POSIX Asynchronous I/O

The POSIX asynchronous I/O interfaces give us a consistent way to perform asynchronous I/O, regardless of the type of file. These interfaces were adopted from the real-time draft standard, which themselves were an option in the Single UNIX Specification. In Version 4, the Single UNIX Specification moved these interfaces to the base, so they are now required to be supported by all platforms.

The asynchronous I/O interfaces use AIO control blocks to describe I/O operations. The `aiocb` structure defines an AIO control block. It contains at least the fields shown in the following structure (implementations might include additional fields):

```
struct aiocb {
  int             aio_fildes;      /* file descriptor */
  off_t           aio_offset;      /* file offset for I/O */
  volatile void  *aio_buf;         /* buffer for I/O */
  size_t          aio_nbytes;      /* number of bytes to transfer */
  int             aio_reqprio;     /* priority */
  struct sigevent aio_sigevent;    /* signal information */
  int             aio_lio_opcode;  /* operation for list I/O */
};
```

The `aio_fildes` field is the file descriptor open for the file to be read or written. The read or write starts at the offset specified by `aio_offset`. For a read, data is copied to the buffer that begins at the address specified by `aio_buf`. For a write, data is copied from this buffer. The `aio_nbytes` field contains the number of bytes to read or write.

Note that we have to provide an explicit offset when we perform asynchronous I/O. The asynchronous I/O interfaces don't affect the file offset maintained by the operating system. This won't be a problem as long as we never mix asynchronous I/O functions with conventional I/O functions on the same file in a process. Also note that if we write to a file opened in append mode (with `O_APPEND`) using an asynchronous interface, the `aio_offset` field in the AIO control block is ignored by the system.

The other fields don't correspond to the conventional I/O functions. The `aio_reqprio` field is a hint that gives applications a way to suggest an ordering for the asynchronous I/O requests. The system has only limited control over the exact ordering, however, so there is no guarantee that the hint will be honored. The `aio_lio_opcode` field is used only with list-based asynchronous I/O, which we'll

discuss shortly. The aio_sigevent field controls how the application is notified about the completion of the I/O event. It is described by a sigevent structure.

```
struct sigevent {
  int             sigev_notify;                 /* notify type */
  int             sigev_signo;                  /* signal number */
  union sigval    sigev_value;                  /* notify argument */
  void (*sigev_notify_function)(union sigval); /* notify function */
  pthread_attr_t *sigev_notify_attributes;      /* notify attrs */
};
```

The sigev_notify field controls the type of notification. It can take on one of three values.

SIGEV_NONE The process is not notified when the asynchronous I/O request completes.

SIGEV_SIGNAL The signal specified by the sigev_signo field is generated when the asynchronous I/O request completes. If the application has elected to catch the signal and has specified the SA_SIGINFO flag when establishing the signal handler, the signal is queued (if the implementation supports queued signals). The signal handler is passed a siginfo structure whose si_value field is set to sigev_value (again, if SA_SIGINFO is used).

SIGEV_THREAD The function specified by the sigev_notify_function field is called when the asynchronous I/O request completes. It is passed the sigev_value field as its only argument. The function is executed in a separate thread in a detached state, unless the sigev_notify_attributes field is set to the address of a pthread attribute structure specifying alternative attributes for the thread.

To perform asynchronous I/O, we need to initialize an AIO control block and call either the aio_read function to make an asynchronous read or the aio_write function to make an asynchronous write.

```
#include <aio.h>

int aio_read(struct aiocb *aiocb);

int aio_write(struct aiocb *aiocb);
```

Both return: 0 if OK, –1 on error

When these functions return success, the asynchronous I/O request has been queued for processing by the operating system. The return value bears no relation to the result of the actual I/O operation. While the I/O operation is pending, we have to be careful to ensure that the AIO control block and data buffer remain stable; their underlying memory must remain valid and we can't reuse them until the I/O operation completes.

To force all pending asynchronous writes to persistent storage without waiting, we can set up an AIO control block and call the aio_fsync function.

```
#include <aio.h>

int aio_fsync(int op, struct aiocb *aiocb);
```

Returns: 0 if OK, –1 on error

The `aio_fildes` field in the AIO control block indicates the file whose asynchronous writes are synched. If the *op* argument is set to `O_DSYNC`, then the operation behaves like a call to `fdatasync`. Otherwise, if *op* is set to `O_SYNC`, the operation behaves like a call to `fsync`.

Like the `aio_read` and `aio_write` functions, the `aio_fsync` operation returns when the synch is scheduled. The data won't be persistent until the asynchronous synch completes. The AIO control block controls how we are notified, just as with the `aio_read` and `aio_write` functions.

To determine the completion status of an asynchronous read, write, or synch operation, we need to call the `aio_error` function.

```
#include <aio.h>

int aio_error(const struct aiocb *aiocb);
```

Returns: (see following)

The return value tells us one of four things.

`0`	The asynchronous operation completed successfully. We need to call the `aio_return` function to obtain the return value from the operation.
–1	The call to `aio_error` failed. In this case, `errno` tells us why.
`EINPROGRESS`	The asynchronous read, write, or synch is still pending.
anything else	Any other return value gives us the error code corresponding to the failed asynchronous operation.

If the asynchronous operation succeeded, we can call the `aio_return` function to get the asynchronous operation's return value.

```
#include <aio.h>

ssize_t aio_return(const struct aiocb *aiocb);
```

Returns: (see following)

Until the asynchronous operation completes, we need to be careful to avoid calling the `aio_return` function. The results are undefined until the operation completes. We also need to be careful to call `aio_return` only one time per asynchronous I/O operation. Once we call this function, the operating system is free to deallocate the record containing the I/O operation's return value.

The `aio_return` function will return –1 and set `errno` if `aio_return` itself fails. Otherwise, it will return the results of the asynchronous operation. In this case, it will return whatever `read`, `write`, or `fsync` would have returned on success if one of those functions had been called.

We use asynchronous I/O when we have other processing to do and we don't want to block while performing the I/O operation. However, when we have completed the processing and find that we still have asynchronous operations outstanding, we can call the `aio_suspend` function to block until an operation completes.

```
#include <aio.h>

int aio_suspend(const struct aiocb *const list[], int nent,
                const struct timespec *timeout);
```

Returns: 0 if OK, –1 on error

One of three things can cause `aio_suspend` to return. If we are interrupted by a signal, it returns –1 with `errno` set to `EINTR`. If the time limit specified by the optional *timeout* argument expires without any of the I/O operations completing, then `aio_suspend` returns –1 with `errno` set to `EAGAIN` (we can pass a null pointer for the *timeout* argument if we want to block without a time limit). If any of the I/O operations complete, `aio_suspend` returns 0. If all asynchronous I/O operations are complete when we call `aio_suspend`, then `aio_suspend` will return without blocking.

The *list* argument is a pointer to an array of AIO control blocks and the *nent* argument indicates the number of entries in the array. Null pointers in the array are skipped; the other entries must point to AIO control blocks that have been used to initiate asynchronous I/O operations.

When we have pending asynchronous I/O operations that we no longer want to complete, we can attempt to cancel them with the `aio_cancel` function.

```
#include <aio.h>

int aio_cancel(int fd, struct aiocb *aiocb);
```

Returns: (see following)

The *fd* argument specifies the file descriptor with the outstanding asynchronous I/O operations. If the *aiocb* argument is `NULL`, then the system attempts to cancel all outstanding asynchronous I/O operations on the file. Otherwise, the system attempts to cancel the single asynchronous I/O operation described by the AIO control block. We say that the system "attempts" to cancel the operations, because there is no guarantee that the system will be able to cancel any operations that are in progress.

The `aio_cancel` function can return one of four values:

`AIO_ALLDONE`	All of the operations completed before the attempt to cancel them.
`AIO_CANCELED`	All of the requested operations have been canceled.
`AIO_NOTCANCELED`	At least one of the requested operations could not be canceled.
–1	The call to `aio_cancel` failed. The error code will be stored in `errno`.

If an asynchronous I/O operation is successfully canceled, calling the `aio_error` function on the corresponding AIO control block will return the error `ECANCELED`. If the operation can't be canceled, then the corresponding AIO control block is unchanged by the call to `aio_cancel`.

One additional function is included with the asynchronous I/O interfaces, although it can be used in either a synchronous or an asynchronous manner. The `lio_listio` function submits a set of I/O requests described by a list of AIO control blocks.

```
#include <aio.h>

int lio_listio(int mode, struct aiocb *restrict const list[restrict],
               int nent, struct sigevent *restrict sigev);
```

Returns: 0 if OK, –1 on error

The *mode* argument determines whether the I/O is truly asynchronous. When it is set to `LIO_WAIT`, the `lio_listio` function won't return until all of the I/O operations specified by the list are complete. In this case, the *sigev* argument is ignored. When the *mode* argument is set to `LIO_NOWAIT`, then the `lio_listio` function returns as soon as the I/O requests are queued. The process is notified asynchronously when all of the I/O operations complete, as specified by the *sigev* argument. If we don't want to be notified, we can set *sigev* to `NULL`. Note that the individual AIO control blocks themselves may also enable asynchronous notification when an individual operation completes. The asynchronous notification specified by the *sigev* argument is in addition to these, and is sent only when all of the I/O operations complete.

The *list* argument points to a list of AIO control blocks specifying the I/O operations to perform. The *nent* argument specifies the number of elements in the array. The list of AIO control blocks can contain `NULL` pointers; these entries are ignored.

In each AIO control block, the `aio_lio_opcode` field specifies whether the operation is a read (`LIO_READ`), a write (`LIO_WRITE`), or a no-op (`LIO_NOP`), which is ignored. A read is treated as if the corresponding AIO control block had been passed to the `aio_read` function. Similarly, a write is treated as if the AIO control block had been passed to `aio_write`.

Implementations can limit the number of asynchronous I/O operations we are allowed to have outstanding. The limits are runtime invariants, and are summarized in Figure 14.19.

Name	Description	Minimum acceptable value
`AIO_LISTIO_MAX`	maximum number of I/O operations in a single list I/O call	`_POSIX_AIO_LISTIO_MAX (2)`
`AIO_MAX`	maximum number of outstanding asynchronous I/O operations	`_POSIX_AIO_MAX (1)`
`AIO_PRIO_DELTA_MAX`	maximum amount by which a process can decrease its asynchronous I/O priority level	0

Figure 14.19 POSIX.1 runtime invariant values for asynchronous I/O

We can determine the value of AIO_LISTIO_MAX by calling the sysconf function with the *name* argument set to _SC_IO_LISTIO_MAX. Similarly, we can determine the value of AIO_MAX by calling sysconf with the *name* argument set to _SC_AIO_MAX, and we can get the value of AIO_PRIO_DELTA_MAX by calling sysconf with its argument set to _SC_AIO_PRIO_DELTA_MAX.

The POSIX asynchronous I/O interfaces were originally introduced to provide real-time applications with a way to avoid being blocked while performing I/O operations. Now we'll look at an example of how to use the interfaces.

Example

We don't discuss real-time programming in this text, but because the POSIX asynchronous I/O interfaces are now part of the base specification in the Single UNIX Specification, we'll look at how to use them. To compare the asynchronous I/O interfaces with their conventional counterparts, we'll look at the task of translating a file from one format to another.

The program shown in Figure 14.20 translates a file using the ROT-13 algorithm that the USENET news system, popular in the 1980s, used to obscure text that might be offensive or contain spoilers or joke punchlines. The algorithm rotates the characters 'a' to 'z' and 'A' to 'Z' by 13 positions, but leaves all other characters unchanged.

```
#include "apue.h"
#include <ctype.h>
#include <fcntl.h>

#define BSZ 4096

unsigned char buf[BSZ];

unsigned char
translate(unsigned char c)
{
    if (isalpha(c)) {
        if (c >= 'n')
            c -= 13;
        else if (c >= 'a')
            c += 13;
        else if (c >= 'N')
            c -= 13;
        else
            c += 13;
    }
    return(c);
}

int
main(int argc, char* argv[])
{
    int ifd, ofd, i, n, nw;
```

```
    if (argc != 3)
        err_quit("usage: rot13 infile outfile");
    if ((ifd = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s", argv[1]);
    if ((ofd = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, FILE_MODE)) < 0)
        err_sys("can't create %s", argv[2]);

    while ((n = read(ifd, buf, BSZ)) > 0) {
        for (i = 0; i < n; i++)
            buf[i] = translate(buf[i]);
        if ((nw = write(ofd, buf, n)) != n) {
            if (nw < 0)
                err_sys("write failed");
            else
                err_quit("short write (%d/%d)", nw, n);
        }
    }

    fsync(ofd);
    exit(0);
}
```

Figure 14.20 Translate a file using ROT-13

The I/O portion of the program is straightforward: we read a block from the input file, translate it, and then write the block to the output file. We repeat this until we hit the end of file and `read` returns zero. The program in Figure 14.21 shows how to perform the same task using the equivalent asynchronous I/O functions.

```
#include "apue.h"
#include <ctype.h>
#include <fcntl.h>
#include <aio.h>
#include <errno.h>

#define BSZ 4096
#define NBUF 8

enum rwop {
    UNUSED = 0,
    READ_PENDING = 1,
    WRITE_PENDING = 2
};

struct buf {
    enum rwop     op;
    int           last;
    struct aiocb  aiocb;
    unsigned char data[BSZ];
};

struct buf bufs[NBUF];
```

```
unsigned char
translate(unsigned char c)
{
    /* same as before */
}

int
main(int argc, char* argv[])
{
    int                 ifd, ofd, i, j, n, err, numop;
    struct stat         sbuf;
    const struct aiocb  *aiolist[NBUF];
    off_t               off = 0;

    if (argc != 3)
        err_quit("usage: rot13 infile outfile");
    if ((ifd = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s", argv[1]);
    if ((ofd = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, FILE_MODE)) < 0)
        err_sys("can't create %s", argv[2]);
    if (fstat(ifd, &sbuf) < 0)
        err_sys("fstat failed");

    /* initialize the buffers */
    for (i = 0; i < NBUF; i++) {
        bufs[i].op = UNUSED;
        bufs[i].aiocb.aio_buf = bufs[i].data;
        bufs[i].aiocb.aio_sigevent.sigev_notify = SIGEV_NONE;
        aiolist[i] = NULL;
    }

    numop = 0;
    for (;;) {
        for (i = 0; i < NBUF; i++) {
            switch (bufs[i].op) {
            case UNUSED:
                /*
                 * Read from the input file if more data
                 * remains unread.
                 */
                if (off < sbuf.st_size) {
                    bufs[i].op = READ_PENDING;
                    bufs[i].aiocb.aio_fildes = ifd;
                    bufs[i].aiocb.aio_offset = off;
                    off += BSZ;
                    if (off >= sbuf.st_size)
                        bufs[i].last = 1;
                    bufs[i].aiocb.aio_nbytes = BSZ;
                    if (aio_read(&bufs[i].aiocb) < 0)
                        err_sys("aio_read failed");
                    aiolist[i] = &bufs[i].aiocb;
```

```
            numop++;
        }
        break;

    case READ_PENDING:
        if ((err = aio_error(&bufs[i].aiocb)) == EINPROGRESS)
            continue;
        if (err != 0) {
            if (err == -1)
                err_sys("aio_error failed");
            else
                err_exit(err, "read failed");
        }

        /*
         * A read is complete; translate the buffer
         * and write it.
         */
        if ((n = aio_return(&bufs[i].aiocb)) < 0)
            err_sys("aio_return failed");
        if (n != BSZ && !bufs[i].last)
            err_quit("short read (%d/%d)", n, BSZ);
        for (j = 0; j < n; j++)
            bufs[i].data[j] = translate(bufs[i].data[j]);
        bufs[i].op = WRITE_PENDING;
        bufs[i].aiocb.aio_fildes = ofd;
        bufs[i].aiocb.aio_nbytes = n;
        if (aio_write(&bufs[i].aiocb) < 0)
            err_sys("aio_write failed");
        /* retain our spot in aiolist */
        break;

    case WRITE_PENDING:
        if ((err = aio_error(&bufs[i].aiocb)) == EINPROGRESS)
            continue;
        if (err != 0) {
            if (err == -1)
                err_sys("aio_error failed");
            else
                err_exit(err, "write failed");
        }

        /*
         * A write is complete; mark the buffer as unused.
         */
        if ((n = aio_return(&bufs[i].aiocb)) < 0)
            err_sys("aio_return failed");
        if (n != bufs[i].aiocb.aio_nbytes)
            err_quit("short write (%d/%d)", n, BSZ);
        aiolist[i] = NULL;
        bufs[i].op = UNUSED;
```

```
                numop--;
                break;
            }
        }
        if (numop == 0) {
            if (off >= sbuf.st_size)
                break;
        } else {
            if (aio_suspend(aiolist, NBUF, NULL) < 0)
                err_sys("aio_suspend failed");
        }
    }

    bufs[0].aiocb.aio_fildes = ofd;
    if (aio_fsync(O_SYNC, &bufs[0].aiocb) < 0)
        err_sys("aio_fsync failed");
    exit(0);
}
```

Figure 14.21 Translate a file using ROT-13 and asynchronous I/O

Note that we use eight buffers, so we can have up to eight asynchronous I/O requests pending. Surprisingly, this might actually reduce performance—if the reads are presented to the file system out of order, it can defeat the operating system's read-ahead algorithm.

Before we can check the return value of an operation, we need to make sure the operation has completed. When `aio_error` returns a value other than `EINPROGRESS` or –1, we know the operation is complete. Excluding these values, if the return value is anything other than 0, then we know the operation failed. Once we've checked these conditions, it is safe to call `aio_return` to get the return value of the I/O operation.

As long as we have work to do, we can submit asynchronous I/O operations. When we have an unused AIO control block, we can submit an asynchronous read request. When a read completes, we translate the buffer contents and then submit an asynchronous write request. When all AIO control blocks are in use, we wait for an operation to complete by calling `aio_suspend`.

When we write a block to the output file, we retain the same offset at which we read the data from the input file. Consequently, the order of the writes doesn't matter. This strategy works only because each character in the input file has a corresponding character in the output file at the same offset; we neither add nor delete characters in the output file. (This insight might help solve Exercise 14.8.)

We don't use asynchronous notification in this example, because it is easier to use a synchronous programming model. If we had something else to do while the I/O operations were in progress, then the additional work could be folded into the `for` loop. If we needed to prevent this additional work from delaying the task of translating the file, however, then we might have to structure the code to use some form of asynchronous notification. With multiple tasks, we need to prioritize the tasks before deciding how the program should be structured. □

14.6 readv and writev Functions

The readv and writev functions let us read into and write from multiple noncontiguous buffers in a single function call. These operations are called *scatter read* and *gather write*.

```
#include <sys/uio.h>

ssize_t readv(int fd, const struct iovec *iov, int iovcnt);

ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

Both return: number of bytes read or written, –1 on error

The second argument to both functions is a pointer to an array of iovec structures:

```
struct iovec {
  void   *iov_base;  /* starting address of buffer */
  size_t  iov_len;   /* size of buffer */
};
```

The number of elements in the *iov* array is specified by *iovcnt*. It is limited to IOV_MAX (recall Figure 2.11). Figure 14.22 shows a diagram relating the arguments to these two functions and the iovec structure.

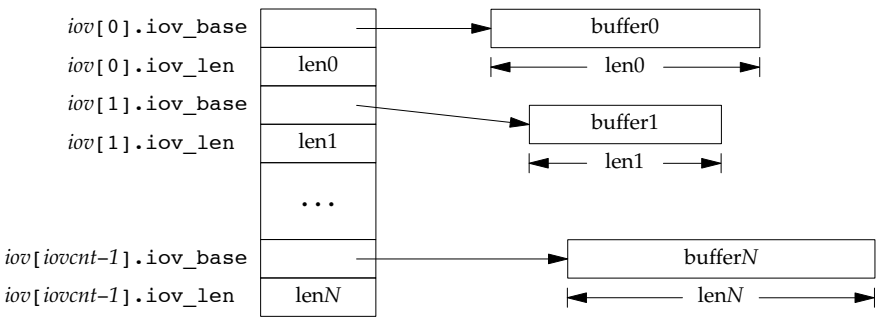


Figure 14.22 The iovec structure for readv and writev

The writev function gathers the output data from the buffers in order: *iov[0]*, *iov[1]*, through *iov[iovcnt–1]*; writev returns the total number of bytes output, which should normally equal the sum of all the buffer lengths.

The readv function scatters the data into the buffers in order, always filling one buffer before proceeding to the next. readv returns the total number of bytes that were read. A count of 0 is returned if there is no more data and the end of file is encountered.

These two functions originated in 4.2BSD and were later added to SVR4. These two functions are included in the XSI option of the Single UNIX Specification.

Example

In Section 20.8, in the function `_db_writeidx`, we need to write two buffers consecutively to a file. The second buffer to output is an argument passed by the caller, and the first buffer is one we create, containing the length of the second buffer and a file offset of other information in the file. There are three ways we can do this.

1. Call `write` twice, once for each buffer.
2. Allocate a buffer of our own that is large enough to contain both buffers, and copy both into the new buffer. We then call `write` once for this new buffer.
3. Call `writev` to output both buffers.

The solution we use in Section 20.8 is to use `writev`, but it's instructive to compare it to the other two solutions.

Figure 14.23 shows the results from the three methods just described.

Operation	Linux (Intel x86)			Mac OS X (Intel x86)		
	User	System	Clock	User	System	Clock
two `writes`	0.06	2.04	2.13	0.85	8.33	13.83
buffer copy, then one `write`	0.03	1.13	1.16	0.70	4.87	9.25
one `writev`	0.04	1.21	1.26	0.43	5.34	9.24

Figure 14.23 Timing results comparing `writev` and other techniques

The test program that we measured output a 100-byte header followed by 200 bytes of data. This was done 1,048,576 times, generating a 300-megabyte file. The test program has three separate cases—one for each of the techniques measured in Figure 14.23. We used `times` (Section 8.17) to obtain the user CPU time, system CPU time, and wall clock time before and after the writes. All three times are shown in seconds.

As we expect, the system time increases when we call `write` twice, compared to calling either `write` or `writev` once. This correlates with the results in Figure 3.6.

Next, note that the sum of the CPU times (user plus system) is slightly less when we do a buffer copy followed by a single `write` compared to a single call to `writev`. With the single `write`, we copy the buffers to a staging buffer at user level, and then the kernel will copy the data to its internal buffers when we call `write`. With `writev`, we should do less copying, because the kernel only needs to copy the data directly into its staging buffers. The fixed cost of using `writev` for such small amounts of data, however, is greater than the benefit. As the amount of data we need to copy increases, the more expensive it will be to copy the buffers in our program, and the `writev` alternative will be more attractive.

> Don't infer too much about the relative performance of Linux and Mac OS X from the numbers shown in Figure 14.23. The two computers were very different: they had different processor generations, different amounts of RAM, and disks with different speeds. To do an apples-to-apples comparison of one operating system to another, we need to use the same hardware for each operating system.

□

In summary, we should always try to use the fewest number of system calls necessary to get the job done. If we are writing small amounts of data, we will find it less expensive to copy the data ourselves and use a single `write` instead of using `writev`. We might find, however, that the performance benefits aren't worth the extra complexity cost needed to manage our own staging buffers.

14.7 `readn` and `writen` Functions

Pipes, FIFOs, and some devices—notably terminals and networks—have the following two properties.

1. A `read` operation may return less than asked for, even though we have not encountered the end of file. This is not an error, and we should simply continue reading from the device.
2. A `write` operation can return less than we specified. This may be caused by kernel output buffers becoming full, for example. Again, it's not an error, and we should continue writing the remainder of the data. (Normally, this short return from a `write` occurs only with a nonblocking descriptor or if a signal is caught.)

We'll never see this happen when reading or writing a disk file, except when the file system runs out of space or we hit our quota limit and we can't write all that we requested.

Generally, when we read from or write to a pipe, network device, or terminal, we need to take these characteristics into consideration. We can use the `readn` and `writen` functions to read and write *N* bytes of data, respectively, letting these functions handle a return value that's possibly less than requested. These two functions simply call `read` or `write` as many times as required to read or write the entire *N* bytes of data.

```
#include "apue.h"

ssize_t readn(int fd, void *buf, size_t nbytes);

ssize_t writen(int fd, void *buf, size_t nbytes);
```

Both return: number of bytes read or written, –1 on error

We define these functions as a convenience for later examples, similar to the error-handling routines used in many of the examples in this text. The `readn` and `writen` functions are not part of any standard.

We call `writen` whenever we're writing to one of the file types that we mentioned, but we call `readn` only when we know ahead of time that we will be receiving a certain number of bytes. Figure 14.24 shows implementations of `readn` and `writen` that we will use in later examples.

Note that if we encounter an error and have previously read or written any data, we return the amount of data transferred instead of the error. Similarly, if we reach the end

of file while reading, we return the number of bytes copied to the caller's buffer if we already read some data successfully and have not yet satisfied the amount requested.

```
#include "apue.h"

ssize_t             /* Read "n" bytes from a descriptor  */
readn(int fd, void *ptr, size_t n)
{
    size_t      nleft;
    ssize_t     nread;

    nleft = n;
    while (nleft > 0) {
        if ((nread = read(fd, ptr, nleft)) < 0) {
            if (nleft == n)
                return(-1); /* error, return -1 */
            else
                break;      /* error, return amount read so far */
        } else if (nread == 0) {
            break;          /* EOF */
        }
        nleft -= nread;
        ptr   += nread;
    }
    return(n - nleft);      /* return >= 0 */
}

ssize_t             /* Write "n" bytes to a descriptor  */
writen(int fd, const void *ptr, size_t n)
{
    size_t      nleft;
    ssize_t     nwritten;

    nleft = n;
    while (nleft > 0) {
        if ((nwritten = write(fd, ptr, nleft)) < 0) {
            if (nleft == n)
                return(-1); /* error, return -1 */
            else
                break;      /* error, return amount written so far */
        } else if (nwritten == 0) {
            break;
        }
        nleft -= nwritten;
        ptr   += nwritten;
    }
    return(n - nleft);      /* return >= 0 */
}
```

Figure 14.24 The `readn` and `writen` functions

14.8 Memory-Mapped I/O

Memory-mapped I/O lets us map a file on disk into a buffer in memory so that, when we fetch bytes from the buffer, the corresponding bytes of the file are read. Similarly, when we store data in the buffer, the corresponding bytes are automatically written to the file. This lets us perform I/O without using `read` or `write`.

> Memory-mapped I/O has been in use with virtual memory systems for many years. In 1981, 4.1BSD provided a different form of memory-mapped I/O with its `vread` and `vwrite` functions. These two functions were then removed in 4.2BSD and were intended to be replaced with the `mmap` function. The `mmap` function, however, was not included with 4.2BSD (for reasons described in Section 2.5 of McKusick et al. [1996]). Gingell, Moran, and Shannon [1987] describe one implementation of `mmap`. Version 4 of the Single UNIX Specification moved the `mmap` function from an option to the base specification. All POSIX-conforming systems are required to support it.

To use this feature, we have to tell the kernel to map a given file to a region in memory. This task is handled by the `mmap` function.

```
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flag, int fd, off_t off);
```

Returns: starting address of mapped region if OK, `MAP_FAILED` on error

The *addr* argument lets us specify the address where we want the mapped region to start. We normally set this value to 0 to allow the system to choose the starting address. The return value of this function is the starting address of the mapped area.

The *fd* argument is the file descriptor specifying the file that is to be mapped. We have to open this file before we can map it into the address space. The *len* argument is the number of bytes to map, and *off* is the starting offset in the file of the bytes to map. (Some restrictions on the value of *off* are described later.)

The *prot* argument specifies the protection of the mapped region.

prot	Description
`PROT_READ`	Region can be read.
`PROT_WRITE`	Region can be written.
`PROT_EXEC`	Region can be executed.
`PROT_NONE`	Region cannot be accessed.

Figure 14.25 Protection of memory-mapped region

We can specify the protection as either `PROT_NONE` or the bitwise OR of any combination of `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC`. The protection specified for a region can't allow more access than the `open` mode of the file. For example, we can't specify `PROT_WRITE` if the file was opened read-only.

Before looking at the *flag* argument, let's see what's going on here. Figure 14.26 shows a memory-mapped file. (Recall the memory layout of a typical process, shown in Figure 7.6.) In this figure, "start addr" is the return value from `mmap`. We have shown

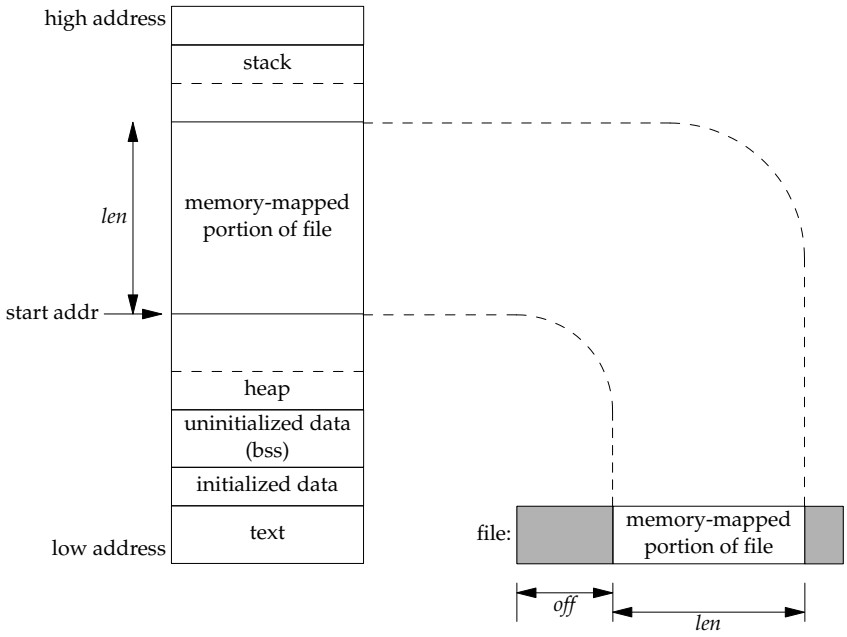


Figure 14.26 Example of a memory-mapped file

the mapped memory being somewhere between the heap and the stack: this is an implementation detail and may differ from one implementation to the next.

The *flag* argument affects various attributes of the mapped region.

`MAP_FIXED` The return value must equal *addr*. Use of this flag is discouraged, as it hinders portability. If this flag is not specified and if *addr* is nonzero, then the kernel uses *addr* as a hint of where to place the mapped region, but there is no guarantee that the requested address will be used. Maximum portability is obtained by specifying *addr* as 0.

> Support for the `MAP_FIXED` flag is optional on POSIX-conforming systems, but required on XSI-conforming systems.

`MAP_SHARED` This flag describes the disposition of store operations into the mapped region by this process. This flag specifies that store operations modify the mapped file—that is, a store operation is equivalent to a `write` to the file. Either this flag or the next (`MAP_PRIVATE`), but not both, must be specified.

`MAP_PRIVATE` This flag says that store operations into the mapped region cause a private copy of the mapped file to be created. All successive

references to the mapped region then reference the copy. (One use of this flag is for a debugger that maps the text portion of a program file but allows the user to modify the instructions. Any modifications affect the copy, not the original program file.)

Each implementation has additional `MAP_xxx` flag values, which are specific to that implementation. Check the `mmap`(2) manual page on your system for details.

The value of *off* and the value of *addr* (if `MAP_FIXED` is specified) are usually required to be multiples of the system's virtual memory page size. This value can be obtained from the `sysconf` function (Section 2.5.4) with an argument of `_SC_PAGESIZE` or `_SC_PAGE_SIZE`. Since *off* and *addr* are often specified as 0, this requirement is not a big deal.

> This requirement is usually imposed by the system implementations. Although the Single UNIX Specification no longer requires that this condition be satisfied, all the platforms covered in this book, except FreeBSD 8.0, have this requirement. FreeBSD 8.0 allows us to use any address alignment and offset alignment as long as the alignments match.

Since the starting offset of the mapped file is tied to the system's virtual memory page size, what happens if the length of the mapped region isn't a multiple of the page size? Assume that the file size is 12 bytes and that the system's page size is 512 bytes. In this case, the system normally provides a mapped region of 512 bytes, and the final 500 bytes of this region are set to 0. We can modify the final 500 bytes, but any changes we make to them are not reflected in the file. Thus we cannot append to a file with `mmap`. We must first grow the file, as we will see in Figure 14.27.

Two signals are normally used with mapped regions. `SIGSEGV` is normally used to indicate that we have tried to access memory that is not available to us. This signal can also be generated if we try to store into a mapped region that we specified to `mmap` as read-only. The `SIGBUS` signal can be generated if we access a portion of the mapped region that does not make sense at the time of the access. For example, assume that we map a file using the file's size, but before we reference the mapped region, the file's size is truncated by some other process. If we then try to access the memory-mapped region corresponding to the end portion of the file that was truncated, we'll receive `SIGBUS`.

A memory-mapped region is inherited by a child across a `fork` (since it's part of the parent's address space), but for the same reason, is not inherited by the new program across an `exec`.

We can change the permissions on an existing mapping by calling `mprotect`.

```
#include <sys/mman.h>

int mprotect(void *addr, size_t len, int prot);
```

Returns: 0 if OK, –1 on error

The legal values for *prot* are the same as those for `mmap` (Figure 14.25). Be aware that implementations may require the address argument to be an integral multiple of the system's page size.

When we modify pages that we've mapped into our address space using the `MAP_SHARED` flag, the changes aren't written back to the file immediately. Instead, the

kernel daemons decide when dirty pages are written back based on (a) system load and (b) configuration parameters meant to limit data loss in the event of a system failure. When the changes are written back, they are written in units of pages. Thus, if we modify only one byte in a page, when the change is written back to the file, the entire page will be written.

If the pages in a shared mapping have been modified, we can call `msync` to flush the changes to the file that backs the mapping. The `msync` function is similar to `fsync` (Section 3.13), but works on memory-mapped regions.

```
#include <sys/mman.h>

int msync(void *addr, size_t len, int flags);
```

Returns: 0 if OK, –1 on error

If the mapping is private, the file mapped is not modified. As with the other memory-mapped functions, the address must be aligned on a page boundary.

The *flags* argument allows us some control over how the memory is flushed. We can specify the `MS_ASYNC` flag to simply schedule the pages to be written. If we want to wait for the writes to complete before returning, we can use the `MS_SYNC` flag. Either `MS_ASYNC` or `MS_SYNC` must be specified.

An optional flag, `MS_INVALIDATE`, lets us tell the operating system to discard any pages that are out of sync with the underlying storage. Some implementations will discard all pages in the specified range when we use this flag, but this behavior is not required.

> The `msync` function is included in the XSI option in the Single UNIX Specification. As such, all UNIX systems must support it.

A memory-mapped region is automatically unmapped when the process terminates or we can unmap a region directly by calling the `munmap` function. Closing the file descriptor used when we mapped the region does not unmap the region.

```
#include <sys/mman.h>

int munmap(void *addr, size_t len);
```

Returns: 0 if OK, –1 on error

The `munmap` function does not affect the object that was mapped—that is, the call to `munmap` does not cause the contents of the mapped region to be written to the disk file. The updating of the disk file for a `MAP_SHARED` region happens automatically by the kernel's virtual memory algorithm sometime after we store into the memory-mapped region. Modifications to memory in a `MAP_PRIVATE` region are discarded when the region is unmapped.

Example

The program in Figure 14.27 copies a file (similar to the `cp`(1) command) using memory-mapped I/O.

```
#include "apue.h"
#include <fcntl.h>
#include <sys/mman.h>

#define COPYINCR (1024*1024*1024)   /* 1 GB */

int
main(int argc, char *argv[])
{
    int         fdin, fdout;
    void        *src, *dst;
    size_t      copysz;
    struct stat sbuf;
    off_t       fsz = 0;

    if (argc != 3)
        err_quit("usage: %s <fromfile> <tofile>", argv[0]);

    if ((fdin = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s for reading", argv[1]);

    if ((fdout = open(argv[2], O_RDWR | O_CREAT | O_TRUNC,
      FILE_MODE)) < 0)
        err_sys("can't creat %s for writing", argv[2]);

    if (fstat(fdin, &sbuf) < 0)         /* need size of input file */
        err_sys("fstat error");

    if (ftruncate(fdout, sbuf.st_size) < 0) /* set output file size */
        err_sys("ftruncate error");

    while (fsz < sbuf.st_size) {
        if ((sbuf.st_size - fsz) > COPYINCR)
            copysz = COPYINCR;
        else
            copysz = sbuf.st_size - fsz;

        if ((src = mmap(0, copysz, PROT_READ, MAP_SHARED,
          fdin, fsz)) == MAP_FAILED)
            err_sys("mmap error for input");
        if ((dst = mmap(0, copysz, PROT_READ | PROT_WRITE,
          MAP_SHARED, fdout, fsz)) == MAP_FAILED)
            err_sys("mmap error for output");

        memcpy(dst, src, copysz);   /* does the file copy */
        munmap(src, copysz);
        munmap(dst, copysz);
        fsz += copysz;
    }
    exit(0);
}
```

Figure 14.27 Copy a file using memory-mapped I/O

We first open both files and then call `fstat` to obtain the size of the input file. We need this size for the call to `mmap` for the input file, and we also need to set the size of the output file. We call `ftruncate` to set the size of the output file. If we don't set the output file's size, the call to `mmap` for the output file is successful, but the first reference to the associated memory region generates a `SIGBUS` signal.

We then call `mmap` for each file, to map the file into memory, and finally call `memcpy` to copy data from the input buffer to the output buffer. We copy at most 1 GB of data at a time to limit the amount of memory we use (it might not be possible to map the entire contents of a very large file if the system doesn't have enough memory). Before mapping the next sections of the files, we unmap the previous sections.

As the bytes of data are fetched from the input buffer (`src`), the input file is automatically read by the kernel; as the data is stored in the output buffer (`dst`), the data is automatically written to the output file.

> Exactly when the data is written to the file depends on the system's page management algorithms. Some systems have daemons that write dirty pages to disk slowly over time. If we want to ensure that the data is safely written to the file, we need to call `msync` with the `MS_SYNC` flag before exiting.

Let's compare this memory-mapped file copy to a copy that is done by calling `read` and `write` (with a buffer size of 8,192). Figure 14.28 shows the results. The times are given in seconds and the size of the file copied was 300 MB. Note that we don't synch the data to disk before exiting.

Operation	Linux 3.2.0 (Intel x86)			Solaris 10 (SPARC)		
	User	System	Clock	User	System	Clock
`read/write`	0.01	0.54	5.67	0.29	10.60	43.67
`mmap/memcpy`	0.08	0.65	22.54	1.89	8.56	38.42

Figure 14.28 Timing results comparing `read/write` versus `mmap/memcpy`

For both Linux 3.2.0 and Solaris 10, the total CPU time (user + system) is almost the same for both approaches. On Solaris, copying using `mmap` and `memcpy` takes more user time but less system time than copying using `read` and `write`. On Linux, the results are similar for the user time, but the system time for using `read` and `write` is slightly better than using `mmap` and `memcpy`. The two versions do the same work, but they go about it differently.

The major difference is that with `read` and `write`, we execute a lot more system calls and do more copying than with `mmap` and `memcpy`. With `read` and `write`, we copy the data from the kernel's buffer to the application's buffer (`read`), and then copy the data from the application's buffer to the kernel's buffer (`write`). With `mmap` and `memcpy`, we copy the data directly from one kernel buffer mapped into our address space into another kernel buffer mapped into our address space. This copying occurs as a result of page fault handling when we reference memory pages that don't yet exist (there is one fault per page read and one fault per page written). If the overhead for the

system call and extra copying differs from the page fault overhead, then one approach will perform better than the other.

On Linux 3.2.0, as far as elapsed time is concerned, the two versions of the program show a large difference in clock time: the version using `read` and `write` completes four times faster than the version using `mmap` and `memcpy`. However, on Solaris 10, the version with `mmap` and `memcpy` is faster than the version with `read` and `write`. If the CPU times are almost the same, then why would the clock times differ? One possibility is that we might have to wait longer for I/O to complete in one version. This wait time is not counted as CPU processing time. Another possibility is that some system processing might not be counted against our program—the processing done by system daemons to write pages to disk, for example. As we need to allocate pages for reading and writing, these system daemons will help make pages available. If the page writes are random instead of sequential, then it will take longer to write them out to disk, so we will need to wait longer before the pages become available for us to reuse. □

Depending on the system, memory-mapped I/O can be more efficient when copying one regular file to another. There are limitations. We can't use this technique to copy between certain devices (such as a network device or a terminal device), and we have to be careful if the size of the underlying file could change after we map it. Nevertheless, some applications can benefit from memory-mapped I/O, as it can often simplify the algorithms, since we manipulate memory instead of reading and writing a file. One example is the manipulation of a frame buffer device that references a bitmapped display.

Krieger, Stumm, and Unrau [1992] describe an alternative to the standard I/O library (Chapter 5) that uses memory-mapped I/O.

We return to memory-mapped I/O in Section 15.9, showing an example of how it can be used to provide shared memory between related processes.

14.9 Summary

In this chapter, we've described numerous advanced I/O functions, many of which are used in the examples in later chapters:

- Nonblocking I/O—issuing an I/O operation without letting it block
- Record locking (which we'll look at in more detail through an example, the database library in Chapter 20)
- I/O multiplexing—the `select` and `poll` functions (we'll use these in many of the later examples)
- Asynchronous I/O
- The `readv` and `writev` functions (also used in many of the later examples)
- Memory-mapped I/O (`mmap`)

Exercises

14.1 Write a test program that illustrates your system's behavior when a process is blocked while trying to write lock a range of a file and additional read-lock requests are made. Is the process requesting a write lock starved by the processes read locking the file?

14.2 Take a look at your system's headers and examine the implementation of `select` and the four `FD_` macros.

14.3 The system headers usually have a built-in limit on the maximum number of descriptors that the `fd_set` data type can handle. Assume that we need to increase this limit to handle up to 2,048 descriptors. How can we do this?

14.4 Compare the functions provided for signal sets (Section 10.11) and the `fd_set` descriptor sets. Also compare the implementation of the two on your system.

14.5 Implement the function `sleep_us`, which is similar to `sleep`, but waits for a specified number of microseconds. Use either `select` or `poll`. Compare this function to the BSD `usleep` function.

14.6 Can you implement the functions `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT`, and `WAIT_CHILD` from Figure 10.24 using advisory record locking instead of signals? If so, code and test your implementation.

14.7 Determine the capacity of a pipe using nonblocking writes. Compare this value with the value of `PIPE_BUF` from Chapter 2.

14.8 Rewrite the program in Figure 14.21 to make it a filter: read from the standard input and write to the standard output, but use the asynchronous I/O interfaces. What must you change to make it work properly? Keep in mind that you should get the same results whether the standard output is attached to a terminal, a pipe, or a regular file.

14.9 Recall Figure 14.23. Determine the break-even point on your system where using `writev` is faster than copying the data yourself and using a single `write`.

14.10 Run the program in Figure 14.27 to copy a file and determine whether the last-access time for the input file is updated.

14.11 In the program from Figure 14.27, `close` the input file after calling `mmap` to verify that closing the descriptor does not invalidate the memory-mapped I/O.

15

Interprocess Communication

15.1 Introduction

In Chapter 8, we described the process control primitives and saw how to work with multiple processes. But the only way for these processes to exchange information is by passing open files across a `fork` or an `exec` or through the file system. We'll now describe other techniques for processes to communicate with one another: interprocess communication (IPC).

In the past, UNIX System IPC was a hodgepodge of various approaches, few of which were portable across all UNIX system implementations. Through the POSIX and The Open Group (formerly X/Open) standardization efforts, the situation has since improved, but differences still exist. Figure 15.1 summarizes the various forms of IPC that are supported by the four implementations discussed in this text.

Note that the Single UNIX Specification (the "SUS" column) allows an implementation to support full-duplex pipes, but requires only half-duplex pipes. An implementation that supports full-duplex pipes will still work with correctly written applications that assume that the underlying operating system supports only half-duplex pipes. We use "(full)" instead of a bullet to show implementations that support half-duplex pipes by using full-duplex pipes.

In Figure 15.1, we show a bullet where basic functionality is supported. For full-duplex pipes, if the feature can be provided through UNIX domain sockets (Section 17.2), we show "UDS" in the column. Some implementations support the feature with pipes and UNIX domain sockets, so these entries have both "UDS" and a bullet.

The IPC interfaces introduced as part of the real-time extensions to POSIX.1 were included as options in the Single UNIX Specification. In SUSv4, the semaphore interfaces were moved from an option to the base specification.

IPC type	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
half-duplex pipes	•	(full)	•	•	(full)
FIFOs	•	•	•	•	•
full-duplex pipes	allowed	•, UDS	UDS	UDS	•, UDS
named full-duplex pipes	obsolescent	UDS	UDS	UDS	•, UDS
XSI message queues	XSI	•	•	•	•
XSI semaphores	XSI	•	•	•	•
XSI shared memory	XSI	•	•	•	•
message queues (real-time)	MSG option	•	•		•
semaphores	•	•	•	•	•
shared memory (real-time)	SHM option	•	•	•	•
sockets	•	•	•	•	•
STREAMS	obsolescent				•

Figure 15.1 Summary of UNIX System IPC

Named full-duplex pipes are provided as mounted STREAMS-based pipes, but are marked obsolescent in the Single UNIX Specification.

> Although support for STREAMS on Linux is available in the "Linux Fast-STREAMS" package from the OpenSS7 project, the package hasn't been updated recently. The latest release of the package from 2008 claims to work with kernels up to Linux 2.6.26.

The first ten forms of IPC in Figure 15.1 are usually restricted to IPC between processes on the same host. The final two rows—sockets and STREAMS—are the only two forms that are generally supported for IPC between processes on different hosts.

We have divided the discussion of IPC into three chapters. In this chapter, we examine classical IPC: pipes, FIFOs, message queues, semaphores, and shared memory. In the next chapter, we take a look at network IPC using the sockets mechanism. In Chapter 17, we take a look at some advanced features of IPC.

15.2 Pipes

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems. Pipes have two limitations.

1. Historically, they have been half duplex (i.e., data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
2. Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls `fork`, and the pipe is used between the parent and the child.

We'll see that FIFOs (Section 15.5) get around the second limitation, and that UNIX domain sockets (Section 17.2) get around both limitations.

Despite these limitations, half-duplex pipes are still the most commonly used form of IPC. Every time you type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one process to the standard input of the next using a pipe.

A pipe is created by calling the `pipe` function.

```
#include <unistd.h>

int pipe(int fd[2]);
```

Returns: 0 if OK, –1 on error

Two file descriptors are returned through the *fd* argument: *fd[0]* is open for reading, and *fd[1]* is open for writing. The output of *fd[1]* is the input for *fd[0]*.

> Originally in 4.3BSD and 4.4BSD, pipes were implemented using UNIX domain sockets. Even though UNIX domain sockets are full duplex by default, these operating systems hobbled the sockets used with pipes so that they operated in half-duplex mode only.
>
> POSIX.1 allows for implementations to support full-duplex pipes. For these implementations, *fd[0]* and *fd[1]* are open for both reading and writing.

Two ways to picture a half-duplex pipe are shown in Figure 15.2. The left half of the figure shows the two ends of the pipe connected in a single process. The right half of the figure emphasizes that the data in the pipe flows through the kernel.

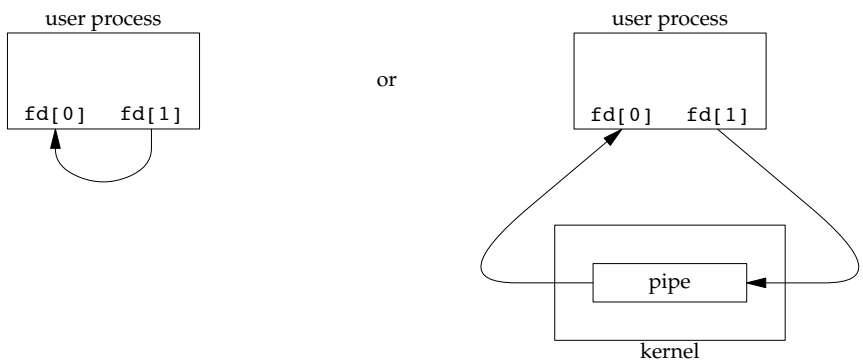


Figure 15.2 Two ways to view a half-duplex pipe

The `fstat` function (Section 4.2) returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the `S_ISFIFO` macro.

> POSIX.1 states that the `st_size` member of the `stat` structure is undefined for pipes. But when the `fstat` function is applied to the file descriptor for the read end of the pipe, many systems store in `st_size` the number of bytes available for reading in the pipe. This is, however, nonportable.

A pipe in a single process is next to useless. Normally, the process that calls `pipe` then calls `fork`, creating an IPC channel from the parent to the child, or vice versa. Figure 15.3 shows this scenario.

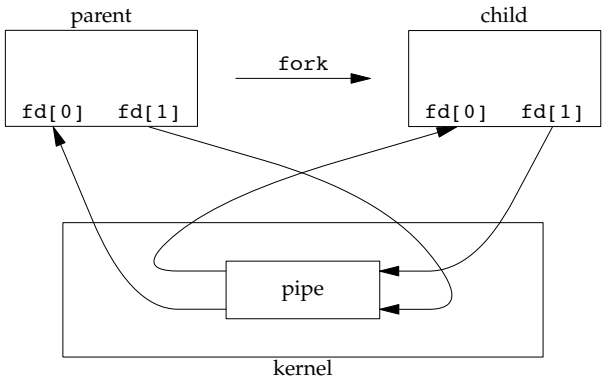


Figure 15.3 Half-duplex pipe after a `fork`

What happens after the `fork` depends on which direction of data flow we want. For a pipe from the parent to the child, the parent closes the read end of the pipe (`fd[0]`), and the child closes the write end (`fd[1]`). Figure 15.4 shows the resulting arrangement of descriptors.

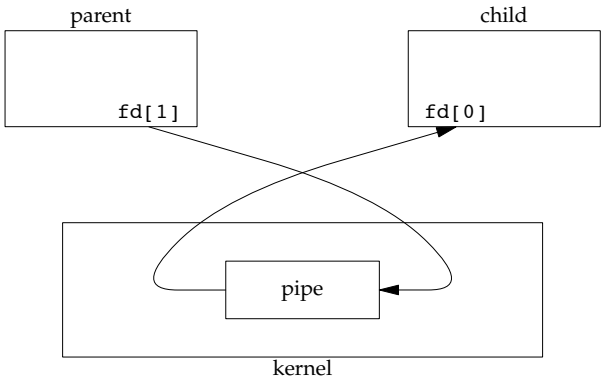


Figure 15.4 Pipe from parent to child

For a pipe from the child to the parent, the parent closes `fd[1]`, and the child closes `fd[0]`.

When one end of a pipe is closed, two rules apply.

1. If we `read` from a pipe whose write end has been closed, `read` returns 0 to indicate an end of file after all the data has been read. (Technically, we should say that this end of file is not generated until there are no more writers for the pipe. It's possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing. Normally, however, there is a single reader and a single writer for a pipe. When we get to FIFOs in the next section, we'll see that often there are multiple writers for a single FIFO.)

2. If we `write` to a pipe whose read end has been closed, the signal `SIGPIPE` is generated. If we either ignore the signal or catch it and return from the signal handler, `write` returns –1 with `errno` set to `EPIPE`.

When we're writing to a pipe (or FIFO), the constant `PIPE_BUF` specifies the kernel's pipe buffer size. A `write` of `PIPE_BUF` bytes or less will not be interleaved with the `write`s from other processes to the same pipe (or FIFO). But if multiple processes are writing to a pipe (or FIFO), and if we `write` more than `PIPE_BUF` bytes, the data might be interleaved with the data from the other writers. We can determine the value of `PIPE_BUF` by using `pathconf` or `fpathconf` (recall Figure 2.12).

Example

Figure 15.5 shows the code to create a pipe between a parent and its child and to send data down the pipe.

```
#include "apue.h"

int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {       /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                    /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

Figure 15.5 Send data from parent to child over a pipe

Note that the pipe direction here matches the orientation shown in Figure 15.4. □

In the previous example, we called `read` and `write` directly on the pipe descriptors. What is more interesting is to duplicate the pipe descriptors onto standard input or standard output. Often, the child then runs some other program, and that program can either read from its standard input (the pipe that we created) or write to its standard output (the pipe).

Example

Consider a program that displays some output that it has created, one page at a time. Rather than reinvent the pagination done by several UNIX system utilities, we want to invoke the user's favorite pager. To avoid writing all the data to a temporary file and calling `system` to display that file, we want to pipe the output directly to the pager. To do this, we create a pipe, `fork` a child process, set up the child's standard input to be the read end of the pipe, and `exec` the user's pager program. Figure 15.6 shows how to do this. (This example takes a command-line argument to specify the name of a file to display. Often, a program of this type would already have the data to display to the terminal in memory.)

```
#include "apue.h"
#include <sys/wait.h>

#define DEF_PAGER   "/bin/more"     /* default pager program */

int
main(int argc, char *argv[])
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    *pager, *argv0;
    char    line[MAXLINE];
    FILE    *fp;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");

    if ((fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);
    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                                   /* parent */
        close(fd[0]);       /* close read end */

        /* parent copies argv[1] to pipe */
        while (fgets(line, MAXLINE, fp) != NULL) {
            n = strlen(line);
            if (write(fd[1], line, n) != n)
                err_sys("write error to pipe");
        }
        if (ferror(fp))
            err_sys("fgets error");

        close(fd[1]);   /* close write end of pipe for reader */

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");
```

```
        exit(0);
    } else {                                        /* child */
        close(fd[1]);   /* close write end */
        if (fd[0] != STDIN_FILENO) {
            if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            close(fd[0]);   /* don't need this after dup2 */
        }

        /* get arguments for execl() */
        if ((pager = getenv("PAGER")) == NULL)
            pager = DEF_PAGER;
        if ((argv0 = strrchr(pager, '/')) != NULL)
            argv0++;        /* step past rightmost slash */
        else
            argv0 = pager;  /* no slash in pager */

        if (execl(pager, argv0, (char *)0) < 0)
            err_sys("execl error for %s", pager);
    }
    exit(0);
}
```

Figure 15.6 Copy file to pager program

Before calling `fork`, we create a pipe. After the `fork`, the parent closes its read end, and the child closes its write end. The child then calls `dup2` to have its standard input be the read end of the pipe. When the pager program is executed, its standard input will be the read end of the pipe.

When we duplicate one descriptor onto another (`fd[0]` onto standard input in the child), we have to be careful that the descriptor doesn't already have the desired value. If the descriptor already had the desired value and we called `dup2` and `close`, the single copy of the descriptor would be closed. (Recall the operation of `dup2` when its two arguments are equal, discussed in Section 3.12.) In this program, if standard input had not been opened by the shell, the `fopen` at the beginning of the program should have used descriptor 0, the lowest unused descriptor, so `fd[0]` should never equal standard input. Nevertheless, whenever we call `dup2` and `close` to duplicate one descriptor onto another, we'll always compare the descriptors first, as a defensive programming measure.

Note how we try to use the environment variable `PAGER` to obtain the name of the user's pager program. If this doesn't work, we use a default. This is a common usage of environment variables. □

Example

Recall the five functions `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT`, and `WAIT_CHILD` from Section 8.9. In Figure 10.24, we showed an implementation using signals. Figure 15.7 shows an implementation using pipes.

```
#include "apue.h"

static int  pfd1[2], pfd2[2];

void
TELL_WAIT(void)
{
    if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
        err_sys("pipe error");
}

void
TELL_PARENT(pid_t pid)
{
    if (write(pfd2[1], "c", 1) != 1)
        err_sys("write error");
}

void
WAIT_PARENT(void)
{
    char    c;

    if (read(pfd1[0], &c, 1) != 1)
        err_sys("read error");

    if (c != 'p')
        err_quit("WAIT_PARENT: incorrect data");
}

void
TELL_CHILD(pid_t pid)
{
    if (write(pfd1[1], "p", 1) != 1)
        err_sys("write error");
}

void
WAIT_CHILD(void)
{
    char    c;

    if (read(pfd2[0], &c, 1) != 1)
        err_sys("read error");

    if (c != 'c')
        err_quit("WAIT_CHILD: incorrect data");
}
```

Figure 15.7 Routines to let a parent and child synchronize

We create two pipes before the `fork`, as shown in Figure 15.8. The parent writes the character "p" across the top pipe when `TELL_CHILD` is called, and the child writes the character "c" across the bottom pipe when `TELL_PARENT` is called. The corresponding `WAIT_xxx` functions do a blocking `read` for the single character.

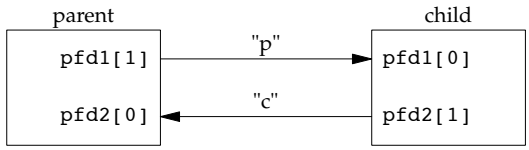


Figure 15.8 Using two pipes for parent–child synchronization

Note that each pipe has an extra reader, which doesn't matter. That is, in addition to the child reading from `pfd1[0]`, the parent has this end of the top pipe open for reading. This doesn't affect us, since the parent doesn't try to read from this pipe. □

15.3 `popen` and `pclose` Functions

Since a common operation is to create a pipe to another process to either read its output or send it input, the standard I/O library has historically provided the `popen` and `pclose` functions. These two functions handle all the dirty work that we've been doing ourselves: creating a pipe, `fork`ing a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
                                        Returns: file pointer if OK, NULL on error

int pclose(FILE *fp);
                                Returns: termination status of cmdstring, or –1 on error
```

The function `popen` does a `fork` and `exec` to execute the *cmdstring* and returns a standard I/O file pointer. If *type* is `"r"`, the file pointer is connected to the standard output of *cmdstring* (Figure 15.9).



Figure 15.9 Result of `fp = popen(`*cmdstring*`, "r")`

If *type* is `"w"`, the file pointer is connected to the standard input of *cmdstring*, as shown in Figure 15.10.



Figure 15.10 Result of `fp = popen(`*cmdstring*`, "w")`

One way to remember the final argument to `popen` is to remember that, like `fopen`, the returned file pointer is readable if *type* is `"r"` or writable if *type* is `"w"`.

The `pclose` function closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell. (We described the termination status in Section 8.6. The `system` function, described in Section 8.13, also returns the termination status.) If the shell cannot be executed, the termination status returned by `pclose` is as if the shell had executed `exit(127)`.

The *cmdstring* is executed by the Bourne shell, as in

```
sh -c cmdstring
```

This means that the shell expands any of its special characters in *cmdstring*. This allows us to say, for example,

```
fp = popen("ls *.c", "r");
```

or

```
fp = popen("cmd 2>&1", "r");
```

Example

Let's redo the program from Figure 15.6, using `popen`. This is shown in Figure 15.11.

```
#include "apue.h"
#include <sys/wait.h>

#define PAGER   "${PAGER:-more}" /* environment variable, or default */

int
main(int argc, char *argv[])
{
    char    line[MAXLINE];
    FILE    *fpin, *fpout;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if ((fpin = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);

    if ((fpout = popen(PAGER, "w")) == NULL)
        err_sys("popen error");

    /* copy argv[1] to pager */
```

```
    while (fgets(line, MAXLINE, fpin) != NULL) {
        if (fputs(line, fpout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (ferror(fpin))
        err_sys("fgets error");
    if (pclose(fpout) == -1)
        err_sys("pclose error");

    exit(0);
}
```

Figure 15.11 Copy file to pager program using popen

Using popen reduces the amount of code we have to write.

The shell command `${PAGER:-more}` says to use the value of the shell variable PAGER if it is defined and non-null; otherwise, use the string more. □

Example — popen and pclose Functions

Figure 15.12 shows our version of popen and pclose.

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/wait.h>

/*
 * Pointer to array allocated at run-time.
 */
static pid_t    *childpid = NULL;

/*
 * From our open_max(), Figure 2.17.
 */
static int      maxfd;

FILE *
popen(const char *cmdstring, const char *type)
{
    int     i;
    int     pfd[2];
    pid_t   pid;
    FILE    *fp;

    /* only allow "r" or "w" */
    if ((type[0] != 'r' && type[0] != 'w') || type[1] != 0) {
        errno = EINVAL;
        return(NULL);
    }
```

```
if (childpid == NULL) {     /* first time through */
    /* allocate zeroed out array for child pids */
    maxfd = open_max();
    if ((childpid = calloc(maxfd, sizeof(pid_t))) == NULL)
        return(NULL);
}

if (pipe(pfd) < 0)
    return(NULL);   /* errno set by pipe() */
if (pfd[0] >= maxfd || pfd[1] >= maxfd) {
    close(pfd[0]);
    close(pfd[1]);
    errno = EMFILE;
    return(NULL);
}

if ((pid = fork()) < 0) {
    return(NULL);   /* errno set by fork() */
} else if (pid == 0) {                          /* child */
    if (*type == 'r') {
        close(pfd[0]);
        if (pfd[1] != STDOUT_FILENO) {
            dup2(pfd[1], STDOUT_FILENO);
            close(pfd[1]);
        }
    } else {
        close(pfd[1]);
        if (pfd[0] != STDIN_FILENO) {
            dup2(pfd[0], STDIN_FILENO);
            close(pfd[0]);
        }
    }

    /* close all descriptors in childpid[] */
    for (i = 0; i < maxfd; i++)
        if (childpid[i] > 0)
            close(i);

    execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
    _exit(127);
}

/* parent continues... */
if (*type == 'r') {
    close(pfd[1]);
    if ((fp = fdopen(pfd[0], type)) == NULL)
        return(NULL);
} else {
    close(pfd[0]);
    if ((fp = fdopen(pfd[1], type)) == NULL)
        return(NULL);
```

```
    }

    childpid[fileno(fp)] = pid; /* remember child pid for this fd */
    return(fp);
}

int
pclose(FILE *fp)
{
    int     fd, stat;
    pid_t   pid;

    if (childpid == NULL) {
        errno = EINVAL;
        return(-1);     /* popen() has never been called */
    }

    fd = fileno(fp);
    if (fd >= maxfd) {
        errno = EINVAL;
        return(-1);     /* invalid file descriptor */
    }
    if ((pid = childpid[fd]) == 0) {
        errno = EINVAL;
        return(-1);     /* fp wasn't opened by popen() */
    }

    childpid[fd] = 0;
    if (fclose(fp) == EOF)
        return(-1);

    while (waitpid(pid, &stat, 0) < 0)
        if (errno != EINTR)
            return(-1); /* error other than EINTR from waitpid() */

    return(stat);   /* return child's termination status */
}
```

Figure 15.12 The popen and pclose functions

Although the core of popen is similar to the code we've used earlier in this chapter, there are many details that we need to take care of. First, each time popen is called, we have to remember the process ID of the child that we create and either its file descriptor or FILE pointer. We choose to save the child's process ID in the array childpid, which we index by the file descriptor. This way, when pclose is called with the FILE pointer as its argument, we call the standard I/O function fileno to get the file descriptor and then have the child process ID for the call to waitpid. Since it's possible for a given process to call popen more than once, we dynamically allocate the childpid array (the first time popen is called), with room for as many children as there are file descriptors.

Note that our open_max function from Figure 2.17 can return a guess of the maximum number of open files if this value is indeterminate for the system. We need to be careful not to use a pipe file descriptor whose value is larger than (or equal to) what the open_max function returns. In popen, if the value returned by open_max happens to be too small, we close the pipe file descriptors, set errno to EMFILE to indicate too many file descriptors are open, and return –1. In pclose, if the file descriptor corresponding to the file pointer argument is larger than expected, we set errno to EINVAL and return –1.

Calling pipe and fork and then duplicating the appropriate descriptors for each process in the popen function is similar to what we did earlier in this chapter.

POSIX.1 requires that popen close any streams that are still open in the child from previous calls to popen. To do this, we go through the childpid array in the child, closing any descriptors that are still open.

What happens if the caller of pclose has established a signal handler for SIGCHLD? The call to waitpid from pclose would return an error of EINTR. Since the caller is allowed to catch this signal (or any other signal that might interrupt the call to waitpid), we simply call waitpid again if it is interrupted by a caught signal.

Note that if the application calls waitpid and obtains the exit status of the child created by popen, we will call waitpid when the application calls pclose, find that the child no longer exists, and return –1 with errno set to ECHILD. This is the behavior required by POSIX.1 in this situation.

> Some early versions of pclose returned an error of EINTR if a signal interrupted the wait. Also, some early versions of pclose blocked or ignored the signals SIGINT, SIGQUIT, and SIGHUP during the wait. This is not allowed by POSIX.1.
>
> □

Note that popen should never be called by a set-user-ID or set-group-ID program. When it executes the command, popen does the equivalent of

```
execl("/bin/sh", "sh", "-c", command, NULL);
```

which executes the shell and *command* with the environment inherited by the caller. A malicious user can manipulate the environment so that the shell executes commands other than those intended, with the elevated permissions granted by the set-ID file mode.

One thing that popen is especially well suited for is executing simple filters to transform the input or output of the running command. Such is the case when a command wants to build its own pipeline.

Example

Consider an application that writes a prompt to standard output and reads a line from standard input. With the popen function, we can interpose a program between the application and its input to transform the input. Figure 15.13 shows the arrangement of processes in this situation.

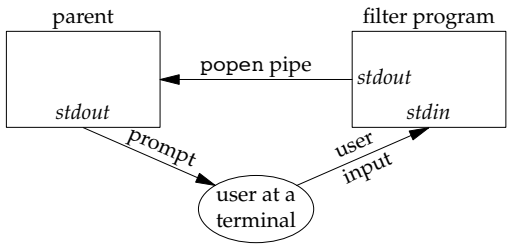


Figure 15.13 Transforming input using popen

The transformation could be pathname expansion, for example, or providing a history mechanism (remembering previously entered commands).

Figure 15.14 shows a simple filter to demonstrate this operation. The filter copies standard input to standard output, converting any uppercase character to lowercase. The reason we're careful to `fflush` standard output after writing a newline is discussed in the next section when we talk about coprocesses.

```
#include "apue.h"
#include <ctype.h>

int
main(void)
{
    int     c;

    while ((c = getchar()) != EOF) {
        if (isupper(c))
            c = tolower(c);
        if (putchar(c) == EOF)
            err_sys("output error");
        if (c == '\n')
            fflush(stdout);
    }
    exit(0);
}
```

Figure 15.14 Filter to convert uppercase characters to lowercase

We compile this filter into the executable file `myuclc`, which we then invoke from the program in Figure 15.15 using `popen`.

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char    line[MAXLINE];
    FILE    *fpin;
```

```
    if ((fpin = popen("myuclc", "r")) == NULL)
        err_sys("popen error");
    for ( ; ; ) {
        fputs("prompt> ", stdout);
        fflush(stdout);
        if (fgets(line, MAXLINE, fpin) == NULL) /* read from pipe */
            break;
        if (fputs(line, stdout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (pclose(fpin) == -1)
        err_sys("pclose error");
    putchar('\n');
    exit(0);
}
```

Figure 15.15 Invoke uppercase/lowercase filter to read commands

We need to call `fflush` after writing the prompt, because the standard output is normally line buffered, and the prompt does not contain a newline. □

15.4 Coprocesses

A UNIX system filter is a program that reads from standard input and writes to standard output. Filters are normally connected linearly in shell pipelines. A filter becomes a *coprocess* when the same program generates the filter's input and reads the filter's output.

The Korn shell provides coprocesses [Bolsky and Korn 1995]. The Bourne shell, the Bourne-again shell, and the C shell don't provide a way to connect processes together as coprocesses. A coprocess normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe. Although the shell syntax required to initiate a coprocess and connect its input and output to other processes is quite contorted (see pp. 62–63 of Bolsky and Korn [1995] for all the details), coprocesses are also useful from a C program.

Whereas `popen` gives us a one-way pipe to the standard input or from the standard output of another process, with a coprocess we have two one-way pipes to the other process: one to its standard input and one from its standard output. We want to write to its standard input, let it operate on the data, and then read from its standard output.

Example

Let's look at coprocesses with an example. The process creates two pipes: one is the standard input of the coprocess and the other is the standard output of the coprocess. Figure 15.16 shows this arrangement.

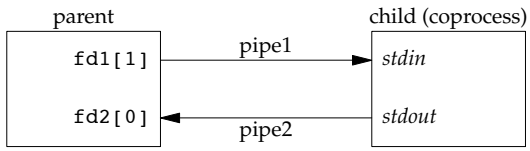


Figure 15.16 Driving a coprocess by writing its standard input and reading its standard output

The program in Figure 15.17 is a simple coprocess that reads two numbers from its standard input, computes their sum, and writes the sum to its standard output. (Coprocesses usually do more interesting work than we illustrate here. This example is admittedly contrived so that we can study the plumbing needed to connect the processes.)

```
#include "apue.h"

int
main(void)
{
    int     n, int1, int2;
    char    line[MAXLINE];

    while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
        line[n] = 0;        /* null terminate */
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            sprintf(line, "%d\n", int1 + int2);
            n = strlen(line);
            if (write(STDOUT_FILENO, line, n) != n)
                err_sys("write error");
        } else {
            if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
                err_sys("write error");
        }
    }
    exit(0);
}
```

Figure 15.17 Simple filter to add two numbers

We compile this program and leave the executable in the file `add2`.

The program in Figure 15.18 invokes the `add2` coprocess after reading two numbers from its standard input. The value from the coprocess is written to its standard output.

```
#include "apue.h"

static void sig_pipe(int);      /* our signal handler */

int
```

```
main(void)
{
    int     n, fd1[2], fd2[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("signal error");

    if (pipe(fd1) < 0 || pipe(fd2) < 0)
        err_sys("pipe error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                           /* parent */
        close(fd1[0]);
        close(fd2[1]);

        while (fgets(line, MAXLINE, stdin) != NULL) {
            n = strlen(line);
            if (write(fd1[1], line, n) != n)
                err_sys("write error to pipe");
            if ((n = read(fd2[0], line, MAXLINE)) < 0)
                err_sys("read error from pipe");
            if (n == 0) {
                err_msg("child closed pipe");
                break;
            }
            line[n] = 0;    /* null terminate */
            if (fputs(line, stdout) == EOF)
                err_sys("fputs error");
        }

        if (ferror(stdin))
            err_sys("fgets error on stdin");
        exit(0);
    } else {                                        /* child */
        close(fd1[1]);
        close(fd2[0]);
        if (fd1[0] != STDIN_FILENO) {
            if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            close(fd1[0]);
        }

        if (fd2[1] != STDOUT_FILENO) {
            if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO)
                err_sys("dup2 error to stdout");
            close(fd2[1]);
        }
        if (execl("./add2", "add2", (char *)0) < 0)
            err_sys("execl error");
```

```
    }
    exit(0);
}

static void
sig_pipe(int signo)
{
    printf("SIGPIPE caught\n");
    exit(1);
}
```

Figure 15.18 Program to drive the add2 filter

Here, we create two pipes, with the parent and the child closing the ends they don't need. We have to use two pipes: one for the standard input of the coprocess and one for its standard output. The child then calls `dup2` to move the pipe descriptors onto its standard input and standard output, before calling `execl`.

If we compile and run the program in Figure 15.18, it works as expected. Furthermore, if we `kill` the `add2` coprocess while the program in Figure 15.18 is waiting for our input and then enter two numbers, the signal handler is invoked when the program writes to the pipe that has no reader. (See Exercise 15.4.) □

Example

In the coprocess `add2` (Figure 15.17), we purposely used low-level I/O (UNIX system calls): `read` and `write`. What happens if we rewrite this coprocess to use standard I/O? Figure 15.19 shows the new version.

```
#include "apue.h"

int
main(void)
{
    int     int1, int2;
    char    line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL) {
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            if (printf("%d\n", int1 + int2) == EOF)
                err_sys("printf error");
        } else {
            if (printf("invalid args\n") == EOF)
                err_sys("printf error");
        }
    }
    exit(0);
}
```

Figure 15.19 Filter to add two numbers, using standard I/O

If we invoke this new coprocess from the program in Figure 15.18, it no longer works. The problem is the default standard I/O buffering. When the program in Figure 15.19 is invoked, the first `fgets` on the standard input causes the standard I/O library to allocate a buffer and choose the type of buffering. Since the standard input is a pipe, the standard I/O library defaults to fully buffered. The same thing happens with the standard output. While `add2` is blocked reading from its standard input, the program in Figure 15.18 is blocked reading from the pipe. We have a deadlock.

Here, we have control over the coprocess that's being run. We can change the program in Figure 15.19 by adding the following four lines before the `while` loop:

```
if (setvbuf(stdin, NULL, _IOLBF, 0) != 0)
    err_sys("setvbuf error");
if (setvbuf(stdout, NULL, _IOLBF, 0) != 0)
    err_sys("setvbuf error");
```

These lines cause `fgets` to return when a line is available and cause `printf` to do an `fflush` when a newline is output (refer to Section 5.4 for the details on standard I/O buffering). Making these explicit calls to `setvbuf` fixes the program in Figure 15.19.

If we aren't able to modify the program that we're piping the output into, other techniques are required. For example, if we use `awk`(1) as a coprocess from our program (instead of the `add2` program), the following won't work:

```
#! /bin/awk -f
{ print $1 + $2 }
```

The reason this won't work is again the standard I/O buffering. But in this case, we cannot change the way `awk` works (unless we have the source code for it). We are unable to modify the executable of `awk` in any way to change the way the standard I/O buffering is handled.

The solution for this general problem is to make the coprocess being invoked (`awk` in this case) think that its standard input and standard output are connected to a terminal. That causes the standard I/O routines in the coprocess to line buffer these two I/O streams, similar to what we did with the explicit calls to `setvbuf` previously. We use pseudo terminals to do this in Chapter 19. □

15.5 FIFOs

FIFOs are sometimes called named pipes. Unnamed pipes can be used only between related processes when a common ancestor has created the pipe. With FIFOs, however, unrelated processes can exchange data.

We saw in Chapter 4 that a FIFO is a type of file. One of the encodings of the `st_mode` member of the `stat` structure (Section 4.2) indicates that a file is a FIFO. We can test for this with the `S_ISFIFO` macro.

Creating a FIFO is similar to creating a file. Indeed, the *pathname* for a FIFO exists in the file system.

```
#include <sys/stat.h>

int mkfifo(const char *path, mode_t mode);

int mkfifoat(int fd, const char *path, mode_t mode);
```

Both return: 0 if OK, –1 on error

The specification of the *mode* argument is the same as for the open function (Section 3.3). The rules for the user and group ownership of the new FIFO are the same as we described in Section 4.6.

The mkfifoat function is similar to the mkfifo function, except that it can be used to create a FIFO in a location relative to the directory represented by the *fd* file descriptor argument. Like the other *at functions, there are three cases:

1. If the *path* parameter specifies an absolute pathname, then the *fd* parameter is ignored and the mkfifoat function behaves like the mkfifo function.
2. If the *path* parameter specifies a relative pathname and the *fd* parameter is a valid file descriptor for an open directory, the pathname is evaluated relative to this directory.
3. If the *path* parameter specifies a relative pathname and the *fd* parameter has the special value AT_FDCWD, the pathname is evaluated starting in the current working directory, and mkfifoat behaves like mkfifo.

Once we have used mkfifo or mkfifoat to create a FIFO, we open it using open. Indeed, the normal file I/O functions (e.g., close, read, write, unlink) all work with FIFOs.

> Applications can create FIFOs with the mknod and mknodat functions. Because POSIX.1 originally didn't include mknod, the mkfifo function was invented specifically for POSIX.1. The mknod and mknodat functions are included in the XSI option in POSIX.1.
>
> POSIX.1 also includes support for the mkfifo(1) command. All four platforms discussed in this text provide this command. As a result, we can create a FIFO using a shell command and then access it with the normal shell I/O redirection.

When we open a FIFO, the nonblocking flag (O_NONBLOCK) affects what happens.

- In the normal case (without O_NONBLOCK), an open for read-only blocks until some other process opens the FIFO for writing. Similarly, an open for write-only blocks until some other process opens the FIFO for reading.
- If O_NONBLOCK is specified, an open for read-only returns immediately. But an open for write-only returns –1 with errno set to ENXIO if no process has the FIFO open for reading.

As with a pipe, if we write to a FIFO that no process has open for reading, the signal SIGPIPE is generated. When the last writer for a FIFO closes the FIFO, an end of file is generated for the reader of the FIFO.

It is common to have multiple writers for a given FIFO. This means that we have to worry about atomic writes if we don't want the writes from multiple processes to be

interleaved. As with pipes, the constant `PIPE_BUF` specifies the maximum amount of data that can be written atomically to a FIFO.

There are two uses for FIFOs.

1. FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
2. FIFOs are used as rendezvous points in client–server applications to pass data between the clients and the servers.

We discuss each of these uses with an example.

Example — Using FIFOs to Duplicate Output Streams

FIFOs can be used to duplicate an output stream in a series of shell commands. This prevents writing the data to an intermediate disk file (similar to using pipes to avoid intermediate disk files). But whereas pipes can be used only for linear connections between processes, a FIFO has a name, so it can be used for nonlinear connections.

Consider a procedure that needs to process a filtered input stream twice. Figure 15.20 shows this arrangement.

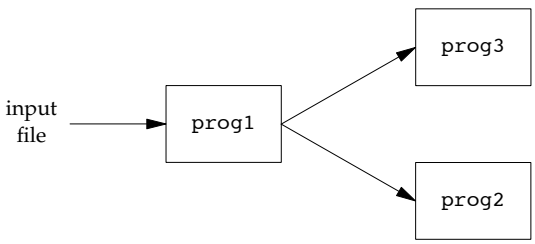


Figure 15.20 Procedure that processes a filtered input stream twice

With a FIFO and the UNIX program `tee`(1), we can accomplish this procedure without using a temporary file. (The `tee` program copies its standard input to both its standard output and the file named on its command line.)

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```

We create the FIFO and then start `prog3` in the background, reading from the FIFO. We then start `prog1` and use `tee` to send its input to both the FIFO and `prog2`. Figure 15.21 shows the process arrangement. □

Example — Client–Server Communication Using a FIFO

Another use for FIFOs is to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known

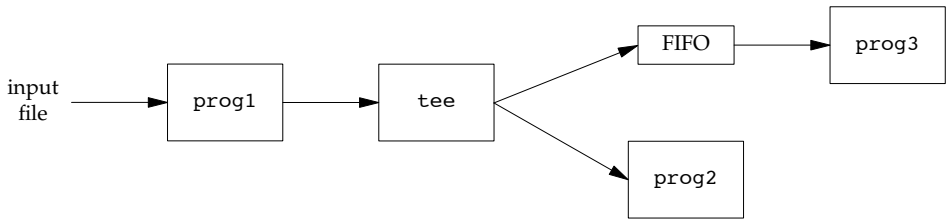


Figure 15.21 Using a FIFO and `tee` to send a stream to two different processes

FIFO that the server creates. (By "well-known," we mean that the pathname of the FIFO is known to all the clients that need to contact the server.) Figure 15.22 shows this arrangement.

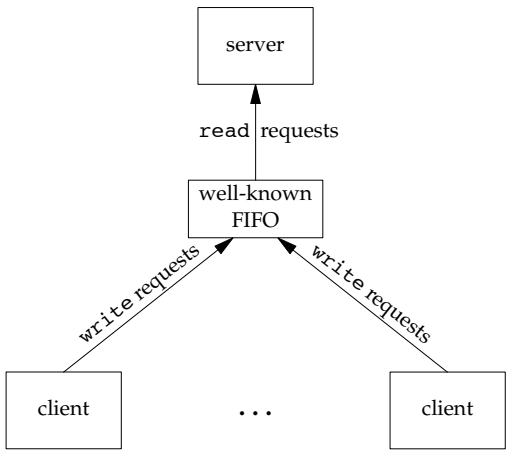


Figure 15.22 Clients sending requests to a server using a FIFO

Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than `PIPE_BUF` bytes in size. This prevents any interleaving of the client `write`s.

The problem in using FIFOs for this type of client–server communication is how to send replies back from the server to each client. A single FIFO can't be used, as the clients would never know when to read their response versus responses for other clients. One solution is for each client to send its process ID with the request. The server then creates a unique FIFO for each client, using a pathname based on the client's process ID. For example, the server can create a FIFO with the name `/tmp/serv1.XXXXX`, where `XXXXX` is replaced with the client's process ID. This arrangement is shown in Figure 15.23.

This arrangement works, although it is impossible for the server to tell whether a client crashes. A client crash leaves the client-specific FIFO in the file system. The

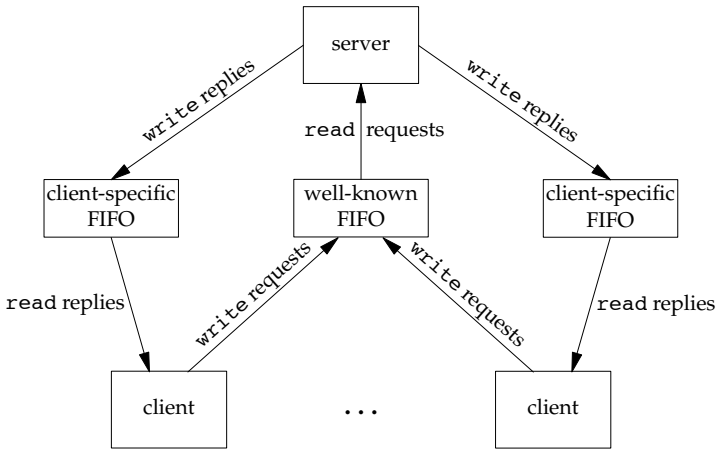


Figure 15.23 Client–server communication using FIFOs

server also must catch `SIGPIPE`, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.

With the arrangement shown in Figure 15.23, if the server opens its well-known FIFO read-only (since it only `read`s from it) each time the number of clients goes from 1 to 0, the server will `read` an end of file on the FIFO. To prevent the server from having to handle this case, a common trick is just to have the server `open` its well-known FIFO for read–write. (See Exercise 15.10.) □

15.6 XSI IPC

The three types of IPC that we call XSI IPC—message queues, semaphores, and shared memory—have many similarities. In this section, we cover these similar features; in the following sections, we look at the specific functions for each of the three IPC types.

> The XSI IPC functions are based closely on the System V IPC functions. These three types of IPC originated in the 1970s in an internal AT&T version of the UNIX System called "Columbus UNIX." These IPC features were later added to System V. They are often criticized for inventing their own namespace instead of using the file system.

15.6.1 Identifiers and Keys

Each *IPC structure* (message queue, semaphore, or shared memory segment) in the kernel is referred to by a non-negative integer *identifier*. To send a message to or fetch a message from a message queue, for example, all we need know is the identifier for the queue. Unlike file descriptors, IPC identifiers are not small integers. Indeed, when a

given IPC structure is created and then removed, the identifier associated with that structure continually increases until it reaches the maximum positive value for an integer, and then wraps around to 0.

The identifier is an internal name for an IPC object. Cooperating processes need an external naming scheme to be able to rendezvous using the same IPC object. For this purpose, an IPC object is associated with a *key* that acts as an external name.

Whenever an IPC structure is being created (by calling `msgget`, `semget`, or `shmget`), a key must be specified. The data type of this key is the primitive system data type `key_t`, which is often defined as a long integer in the header `<sys/types.h>`. This key is converted into an identifier by the kernel.

There are various ways for a client and a server to rendezvous at the same IPC structure.

1. The server can create a new IPC structure by specifying a key of `IPC_PRIVATE` and store the returned identifier somewhere (such as a file) for the client to obtain. The key `IPC_PRIVATE` guarantees that the server creates a new IPC structure. The disadvantage of this technique is that file system operations are required for the server to write the integer identifier to a file, and then for the clients to retrieve this identifier later.

   The `IPC_PRIVATE` key is also used in a parent–child relationship. The parent creates a new IPC structure specifying `IPC_PRIVATE`, and the resulting identifier is then available to the child after the `fork`. The child can pass the identifier to a new program as an argument to one of the `exec` functions.

2. The client and the server can agree on a key by defining the key in a common header, for example. The server then creates a new IPC structure specifying this key. The problem with this approach is that it's possible for the key to already be associated with an IPC structure, in which case the `get` function (`msgget`, `semget`, or `shmget`) returns an error. The server must handle this error, deleting the existing IPC structure, and try to create it again.

3. The client and the server can agree on a pathname and project ID (the project ID is a character value between 0 and 255) and call the function `ftok` to convert these two values into a key. This key is then used in step 2. The only service provided by `ftok` is a way of generating a key from a pathname and project ID.

```
#include <sys/ipc.h>

key_t ftok(const char *path, int id);
```

Returns: key if OK, `(key_t)`–1 on error

The *path* argument must refer to an existing file. Only the lower 8 bits of *id* are used when generating the key.

The key created by `ftok` is usually formed by taking parts of the `st_dev` and `st_ino` fields in the `stat` structure (Section 4.2) corresponding to the given pathname and combining them with the project ID. If two pathnames refer to two different files,

then ftok usually returns two different keys for the two pathnames. However, because both i-node numbers and keys are often stored in long integers, information loss can occur when creating a key. This means that two different pathnames to different files can generate the same key if the same project ID is used.

The three get functions (msgget, semget, and shmget) all have two similar arguments: a *key* and an integer *flag*. A new IPC structure is created (normally by a server) if either *key* is IPC_PRIVATE or *key* is not currently associated with an IPC structure of the particular type and the IPC_CREAT bit of *flag* is specified. To reference an existing queue (normally done by a client), *key* must equal the key that was specified when the queue was created, and IPC_CREAT must not be specified.

Note that it's never possible to specify IPC_PRIVATE to reference an existing queue, since this special *key* value always creates a new queue. To reference an existing queue that was created with a *key* of IPC_PRIVATE, we must know the associated identifier and then use that identifier in the other IPC calls (such as msgsnd and msgrcv), bypassing the get function.

If we want to create a new IPC structure, making sure that we don't reference an existing one with the same identifier, we must specify a *flag* with both the IPC_CREAT and IPC_EXCL bits set. Doing this causes an error return of EEXIST if the IPC structure already exists. (This is similar to an open that specifies the O_CREAT and O_EXCL flags.)

15.6.2 Permission Structure

XSI IPC associates an ipc_perm structure with each IPC structure. This structure defines the permissions and owner and includes at least the following members:

```
struct ipc_perm {
  uid_t  uid;  /* owner's effective user ID */
  gid_t  gid;  /* owner's effective group ID */
  uid_t  cuid; /* creator's effective user ID */
  gid_t  cgid; /* creator's effective group ID */
  mode_t mode; /* access modes */
  .
  .
  .
};
```

Each implementation includes additional members. See <sys/ipc.h> on your system for the complete definition.

All the fields are initialized when the IPC structure is created. At a later time, we can modify the uid, gid, and mode fields by calling msgctl, semctl, or shmctl. To change these values, the calling process must be either the creator of the IPC structure or the superuser. Changing these fields is similar to calling chown or chmod for a file.

The values in the mode field are similar to the values we saw in Figure 4.6, but there is nothing corresponding to execute permission for any of the IPC structures. Also, message queues and shared memory use the terms *read* and *write*, but semaphores use the terms *read* and *alter*. Figure 15.24 shows the six permissions for each form of IPC.

Permission	Bit
user-read	0400
user-write (alter)	0200
group-read	0040
group-write (alter)	0020
other-read	0004
other-write (alter)	0002

Figure 15.24 XSI IPC permissions

Some implementations define symbolic constants to represent each permission, but these constants are not standardized by the Single UNIX Specification.

15.6.3 Configuration Limits

All three forms of XSI IPC have built-in limits that we may encounter. Most of these limits can be changed by reconfiguring the kernel. We describe the limits when we describe each of the three forms of IPC.

> Each platform provides its own way to report and modify a particular limit. FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8 provide the `sysctl` command to view and modify kernel configuration parameters. On Solaris 10, changes to kernel IPC limits are made with the `prctl` command.
>
> On Linux, you can display the IPC-related limits by running `ipcs -l`. On FreeBSD and Mac OS X, the equivalent command is `ipcs -T`. On Solaris, you can discover the tunable parameters by running `sysdef -i`.

15.6.4 Advantages and Disadvantages

A fundamental problem with XSI IPC is that the IPC structures are systemwide and do not have a reference count. For example, if we create a message queue, place some messages on the queue, and then terminate, the message queue and its contents are not deleted. They remain in the system until specifically read or deleted by some process calling `msgrcv` or `msgctl`, by someone executing the `ipcrm`(1) command, or by the system being rebooted. Compare this with a pipe, which is completely removed when the last process to reference it terminates. With a FIFO, although the name stays in the file system until explicitly removed, any data left in a FIFO is removed when the last process to reference the FIFO terminates.

Another problem with XSI IPC is that these IPC structures are not known by names in the file system. We can't access them and modify their properties with the functions we described in Chapters 3 and 4. Almost a dozen new system calls (`msgget`, `semop`, `shmat`, and so on) were added to the kernel to support these IPC objects. We can't see the IPC objects with an `ls` command, we can't remove them with the `rm` command, and we can't change their permissions with the `chmod` command. Instead, two new commands—`ipcs`(1) and `ipcrm`(1)—were added.

Since these forms of IPC don't use file descriptors, we can't use the multiplexed I/O functions (`select` and `poll`) with them. This makes it harder to use more than one of these IPC structures at a time or to use any of these IPC structures with file or device I/O. For example, we can't have a server wait for a message to be placed on one of two message queues without some form of busy–wait loop.

An overview of a transaction processing system built using System V IPC is given in Andrade, Carges, and Kovach [1989]. They claim that the namespace used by System V IPC (the identifiers) is an advantage, not a problem as we said earlier, because using identifiers allows a process to send a message to a message queue with a single function call (`msgsnd`), whereas other forms of IPC normally require an `open`, `write`, and `close`. This argument is false. Clients still have to obtain the identifier for the server's queue somehow, to avoid using a key and calling `msgget`. The identifier assigned to a particular queue depends on how many other message queues exist when the queue is created and how many times the table in the kernel assigned to the new queue has been used since the kernel was bootstrapped. This is a dynamic value that can't be guessed or stored in a header. As we mentioned in Section 15.6.1, minimally a server has to write the assigned queue identifier to a file for its clients to read.

Other advantages listed by these authors for message queues are that they're reliable, flow controlled, and record oriented, and that they can be processed in other than first-in, first-out order. Figure 15.25 compares some of the features of these various forms of IPC.

IPC type	Connectionless?	Reliable?	Flow control?	Records?	Message types or priorities?
message queues	no	yes	yes	yes	yes
STREAMS	no	yes	yes	yes	yes
UNIX domain stream socket	no	yes	yes	no	no
UNIX domain datagram socket	yes	yes	no	yes	no
FIFOs (non-STREAMS)	no	yes	yes	no	no

Figure 15.25 Comparison of features of various forms of IPC

(We describe stream and datagram sockets in Chapter 16. We describe UNIX domain sockets in Section 17.2.) By "connectionless," we mean the ability to send a message without having to call some form of an open function first. As described previously, we don't consider message queues connectionless, since some technique is required to obtain the identifier for a queue. Since all these forms of IPC are restricted to a single host, all are reliable. When the messages are sent across a network, the possibility of messages being lost becomes a concern. "Flow control" means that the sender is put to sleep if there is a shortage of system resources (buffers) or if the receiver can't accept any more messages. When the flow control condition subsides (i.e., when there is room in the queue), the sender should automatically be awakened.

One feature that we don't show in Figure 15.25 is whether the IPC facility can automatically create a unique connection to a server for each client. We'll see in Chapter 17 that UNIX stream sockets provide this capability. The next three sections describe each of the three forms of XSI IPC in detail.

15.7 Message Queues

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. We'll call the message queue just a *queue* and its identifier a *queue ID*.

> The Single UNIX Specification message-passing option includes an alternative IPC message queue interface derived from the POSIX real-time extensions. We do not discuss it in this text.

A new queue is created or an existing queue opened by `msgget`. New messages are added to the end of a queue by `msgsnd`. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to `msgsnd` when the message is added to a queue. Messages are fetched from a queue by `msgrcv`. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

Each queue has the following `msqid_ds` structure associated with it:

```
struct msqid_ds {
  struct ipc_perm  msg_perm;     /* see Section 15.6.2 */
  msgqnum_t        msg_qnum;     /* # of messages on queue */
  msglen_t         msg_qbytes;   /* max # of bytes on queue */
  pid_t            msg_lspid;    /* pid of last msgsnd() */
  pid_t            msg_lrpid;    /* pid of last msgrcv() */
  time_t           msg_stime;    /* last-msgsnd() time */
  time_t           msg_rtime;    /* last-msgrcv() time */
  time_t           msg_ctime;    /* last-change time */
  .
  .
  .
};
```

This structure defines the current status of the queue. The members shown are the ones defined by the Single UNIX Specification. Implementations include additional fields not covered by the standard.

Description	Typical values			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
size in bytes of largest message we can send	16,384	8,192	16,384	derived
maximum size in bytes of a particular queue (i.e., the sum of all the sizes of messages on the queue)	2,048	16,384	2,048	65,536
maximum number of messages queues, systemwide	40	derived	40	128
maximum number of messages, systemwide	40	derived	40	8,192

Figure 15.26 System limits that affect message queues

Figure 15.26 lists the system limits that affect message queues. We show "derived" where a limit is derived from other limits. On Linux, for example, the maximum number of messages is based on the maximum number of queues and the maximum amount of data allowed on the queues. The maximum number of queues, in turn, is based on the amount of RAM installed in the system. Note that the queue maximum byte size limit further limits the maximum size of a message to be placed on a queue.

The first function normally called is `msgget` to either open an existing queue or create a new queue.

```
#include <sys/msg.h>

int msgget(key_t key, int flag);
```

Returns: message queue ID if OK, –1 on error

In Section 15.6.1, we described the rules for converting the *key* into an identifier and discussed whether a new queue is created or an existing queue is referenced. When a new queue is created, the following members of the `msqid_ds` structure are initialized.

- The `ipc_perm` structure is initialized as described in Section 15.6.2. The `mode` member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the values from Figure 15.24.
- `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, and `msg_rtime` are all set to 0.
- `msg_ctime` is set to the current time.
- `msg_qbytes` is set to the system limit.

On success, `msgget` returns the non-negative queue ID. This value is then used with the other three message queue functions.

The `msgctl` function performs various operations on a queue. This function and the related functions for semaphores and shared memory (`semctl` and `shmctl`) are the `ioctl`-like functions for XSI IPC (i.e., the garbage-can functions).

```
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Returns: 0 if OK, –1 on error

The *cmd* argument specifies the command to be performed on the queue specified by *msqid*.

`IPC_STAT` Fetch the `msqid_ds` structure for this queue, storing it in the structure pointed to by *buf*.

`IPC_SET` Copy the following fields from the structure pointed to by *buf* to the `msqid_ds` structure associated with this queue: `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode`, and `msg_qbytes`. This command can be executed only by a process whose effective user ID equals `msg_perm.cuid` or `msg_perm.uid` or by a process with superuser privileges. Only the superuser can increase the value of `msg_qbytes`.

`IPC_RMID` Remove the message queue from the system and any data still on the queue. This removal is immediate. Any other process still using the message queue will get an error of `EIDRM` on its next attempted operation on the queue. This command can be executed only by a process whose effective user ID equals `msg_perm.cuid` or `msg_perm.uid` or by a process with superuser privileges.

We'll see that these three commands (IPC_STAT, IPC_SET, and IPC_RMID) are also provided for semaphores and shared memory.

Data is placed onto a message queue by calling msgsnd.

```
#include <sys/msg.h>

int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

Returns: 0 if OK, –1 on error

As we mentioned earlier, each message is composed of a positive long integer type field, a non-negative length (*nbytes*), and the actual data bytes (corresponding to the length). Messages are always placed at the end of the queue.

The *ptr* argument points to a long integer that contains the positive integer message type, and it is immediately followed by the message data. (There is no message data if *nbytes* is 0.) If the largest message we send is 512 bytes, we can define the following structure:

```
struct mymesg {
  long  mtype;      /* positive message type */
  char  mtext[512]; /* message data, of length nbytes */
};
```

The *ptr* argument is then a pointer to a mymesg structure. The message type can be used by the receiver to fetch messages in an order other than first in, first out.

> Some platforms support both 32-bit and 64-bit environments. This affects the size of long integers and pointers. For example, on 64-bit SPARC systems, Solaris allows both 32-bit and 64-bit applications to coexist. If a 32-bit application were to exchange this structure over a pipe or a socket with a 64-bit application, problems would arise, because the size of a long integer is 4 bytes in a 32-bit application, but 8 bytes in a 64-bit application. This means that a 32-bit application will expect that the mtext field will start 4 bytes after the start of the structure, whereas a 64-bit application will expect the mtext field to start 8 bytes after the start of the structure. In this situation, part of the 64-bit application's mtype field will appear as part of the mtext field to the 32-bit application, and the first 4 bytes in the 32-bit application's mtext field will be interpreted as a part of the mtype field by the 64-bit application.
>
> This problem doesn't happen with XSI message queues, however. Solaris implements the 32-bit version of the IPC system calls with different entry points than the 64-bit version of the IPC system calls. The system calls know how to deal with a 32-bit application communicating with a 64-bit application, and treat the type field specially to avoid it interfering with the data portion of the message. The only potential problem is a loss of information when a 64-bit application sends a message with a value in the 8-byte type field that is larger than will fit in a 32-bit application's 4-byte type field. In this case, the 32-bit application will see a truncated type value.

A *flag* value of IPC_NOWAIT can be specified. This is similar to the nonblocking I/O flag for file I/O (Section 14.2). If the message queue is full (either the total number of messages on the queue equals the system limit, or the total number of bytes on the queue equals the system limit), specifying IPC_NOWAIT causes msgsnd to return immediately with an error of EAGAIN. If IPC_NOWAIT is not specified, we are blocked until there is room for the message, the queue is removed from the system, or a signal is caught and the signal handler returns. In the second case, an error of EIDRM is returned ("identifier removed"); in the last case, the error returned is EINTR.

Note how ungracefully the removal of a message queue is handled. Since a reference count is not maintained with each message queue (as there is for open files), the removal of a queue simply generates errors on the next queue operation by processes still using the queue. Semaphores handle this removal in the same fashion. In contrast, when a file is removed, the file's contents are not deleted until the last open descriptor for the file is closed.

When `msgsnd` returns successfully, the `msqid_ds` structure associated with the message queue is updated to indicate the process ID that made the call (`msg_lspid`), the time that the call was made (`msg_stime`), and that one more message is on the queue (`msg_qnum`).

Messages are retrieved from a queue by `msgrcv`.

```
#include <sys/msg.h>

ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

Returns: size of data portion of message if OK, –1 on error

As with `msgsnd`, the *ptr* argument points to a long integer (where the message type of the returned message is stored) followed by a data buffer for the actual message data. *nbytes* specifies the size of the data buffer. If the returned message is larger than *nbytes* and the `MSG_NOERROR` bit in *flag* is set, the message is truncated. (In this case, no notification is given to us that the message was truncated, and the remainder of the message is discarded.) If the message is too big and this *flag* value is not specified, an error of `E2BIG` is returned instead (and the message stays on the queue).

The *type* argument lets us specify which message we want.

type == 0 The first message on the queue is returned.

type > 0 The first message on the queue whose message type equals *type* is returned.

type < 0 The first message on the queue whose message type is the lowest value less than or equal to the absolute value of *type* is returned.

A nonzero *type* is used to read the messages in an order other than first in, first out. For example, the *type* could be a priority value if the application assigns priorities to the messages. Another use of this field is to contain the process ID of the client if a single message queue is being used by multiple clients and a single server (as long as a process ID fits in a long integer).

We can specify a *flag* value of `IPC_NOWAIT` to make the operation nonblocking, causing `msgrcv` to return –1 with `errno` set to `ENOMSG` if a message of the specified type is not available. If `IPC_NOWAIT` is not specified, the operation blocks until a message of the specified type is available, the queue is removed from the system (–1 is returned with `errno` set to `EIDRM`), or a signal is caught and the signal handler returns (causing `msgrcv` to return –1 with `errno` set to `EINTR`).

When `msgrcv` succeeds, the kernel updates the `msqid_ds` structure associated with the message queue to indicate the caller's process ID (`msg_lrpid`), the time of the call (`msg_rtime`), and that one less message is on the queue (`msg_qnum`).

Example — Timing Comparison of Message Queues and Full-Duplex Pipes

If we need a bidirectional flow of data between a client and a server, we can use either message queues or full-duplex pipes. (Recall from Figure 15.1 that full-duplex pipes are available through the UNIX domain sockets mechanism [Section 17.2], although some platforms provide a full-duplex pipe mechanism through the `pipe` function.)

Figure 15.27 shows a timing comparison of three of these techniques on Solaris: message queues, full-duplex (STREAMS) pipes, and UNIX domain sockets. The tests consisted of a program that created the IPC channel, called `fork`, and then sent about 200 megabytes of data from the parent to the child. The data was sent using 100,000 calls to `msgsnd`, with a message length of 2,000 bytes for the message queue, and 100,000 calls to `write`, with a length of 2,000 bytes for the full-duplex pipe and UNIX domain socket. The times are all in seconds.

Operation	User	System	Clock
message queue	0.58	4.16	5.09
full-duplex pipe	0.61	4.30	5.24
UNIX domain socket	0.59	5.58	7.49

Figure 15.27 Timing comparison of IPC alternatives on Solaris

These numbers show us that message queues, originally implemented to provide higher-than-normal-speed IPC, are no longer that much faster than other forms of IPC. (When message queues were implemented, the only other form of IPC available was half-duplex pipes.) When we consider the problems in using message queues (Section 15.6.4), we come to the conclusion that we shouldn't use them for new applications. □

15.8 Semaphores

A semaphore isn't a form of IPC similar to the others that we've described (pipes, FIFOs, and message queues). A semaphore is a counter used to provide access to a shared data object for multiple processes.

> The Single UNIX Specification includes an alternative set of semaphore interfaces that were originally part of its real-time extensions. We discuss these interfaces in Section 15.10.

To obtain a shared resource, a process needs to do the following:

1. Test the semaphore that controls the resource.
2. If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.

When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened.

To implement semaphores correctly, the test of a semaphore's value and the decrementing of this value must be an atomic operation. For this reason, semaphores are normally implemented inside the kernel.

A common form of semaphore is called a *binary semaphore*. It controls a single resource, and its value is initialized to 1. In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing.

XSI semaphores are, unfortunately, more complicated than this. Three features contribute to this unnecessary complication.

1. A semaphore is not simply a single non-negative value. Instead, we have to define a semaphore as a set of one or more semaphore values. When we create a semaphore, we specify the number of values in the set.
2. The creation of a semaphore (`semget`) is independent of its initialization (`semctl`). This is a fatal flaw, since we cannot atomically create a new semaphore set and initialize all the values in the set.
3. Since all forms of XSI IPC remain in existence even when no process is using them, we have to worry about a program that terminates without releasing the semaphores it has been allocated. The *undo* feature that we describe later is supposed to handle this.

The kernel maintains a `semid_ds` structure for each semaphore set:

```
struct semid_ds {
  struct ipc_perm  sem_perm;  /* see Section 15.6.2 */
  unsigned short   sem_nsems; /* # of semaphores in set */
  time_t           sem_otime; /* last-semop() time */
  time_t           sem_ctime; /* last-change time */
  :
};
```

The Single UNIX Specification defines the fields shown, but implementations can define additional members in the `semid_ds` structure.

Each semaphore is represented by an anonymous structure containing at least the following members:

```
struct {
  unsigned short  semval;   /* semaphore value, always >= 0 */
  pid_t           sempid;   /* pid for last operation */
  unsigned short  semncnt;  /* # processes awaiting semval>curval */
  unsigned short  semzcnt;  /* # processes awaiting semval==0 */
  :
};
```

Figure 15.28 lists the system limits that affect semaphore sets.

Description	Typical values			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
maximum value of any semaphore	32,767	32,767	32,767	65,535
maximum value of any semaphore's adjust-on-exit value	16,384	32,767	16,384	32,767
maximum number of semaphore sets, systemwide	10	128	87,381	128
maximum number of semaphores, systemwide	60	32,000	87,381	derived
maximum number of semaphores per semaphore set	60	250	87,381	512
maximum number of undo structures, systemwide	30	32,000	87,381	derived
maximum number of undo entries per undo structures	10	unlimited	10	derived
maximum number of operations per `semop` call	100	32	5	512

Figure 15.28 System limits that affect semaphores

When we want to use XSI semaphores, we first need to obtain a semaphore ID by calling the `semget` function.

```
#include <sys/sem.h>

int semget(key_t key, int nsems, int flag);
```

Returns: semaphore ID if OK, –1 on error

In Section 15.6.1, we described the rules for converting the *key* into an identifier and discussed whether a new set is created or an existing set is referenced. When a new set is created, the following members of the `semid_ds` structure are initialized.

- The `ipc_perm` structure is initialized as described in Section 15.6.2. The `mode` member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the values from Figure 15.24.
- `sem_otime` is set to 0.
- `sem_ctime` is set to the current time.
- `sem_nsems` is set to *nsems*.

The number of semaphores in the set is *nsems*. If a new set is being created (typically by the server), we must specify *nsems*. If we are referencing an existing set (a client), we can specify *nsems* as 0.

The `semctl` function is the catchall for various semaphore operations.

```
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ... /* union semun arg */ );
```

Returns: (see following)

The fourth argument is optional, depending on the command requested, and if present, is of type `semun`, a union of various command-specific arguments:

```
union semun {
  int               val;   /* for SETVAL */
  struct semid_ds  *buf;   /* for IPC_STAT and IPC_SET */
  unsigned short   *array; /* for GETALL and SETALL */
};
```

Note that the optional argument is the actual union, not a pointer to the union.

> Usually our application must define the `semun` union. However, on FreeBSD 8.0, this is defined for us in `<sys/sem.h>`.

The *cmd* argument specifies one of the following ten commands to be performed on the set specified by *semid*. The five commands that refer to one particular semaphore value use *semnum* to specify one member of the set. The value of *semnum* is between 0 and *nsems – 1*, inclusive.

`IPC_STAT` Fetch the `semid_ds` structure for this set, storing it in the structure pointed to by *arg.buf*.

`IPC_SET` Set the `sem_perm.uid`, `sem_perm.gid`, and `sem_perm.mode` fields from the structure pointed to by *arg.buf* in the `semid_ds` structure associated with this set. This command can be executed only by a process whose effective user ID equals `sem_perm.cuid` or `sem_perm.uid` or by a process with superuser privileges.

`IPC_RMID` Remove the semaphore set from the system. This removal is immediate. Any other process still using the semaphore will get an error of `EIDRM` on its next attempted operation on the semaphore. This command can be executed only by a process whose effective user ID equals `sem_perm.cuid` or `sem_perm.uid` or by a process with superuser privileges.

`GETVAL` Return the value of `semval` for the member *semnum*.

`SETVAL` Set the value of `semval` for the member *semnum*. The value is specified by *arg.val*.

`GETPID` Return the value of `sempid` for the member *semnum*.

`GETNCNT` Return the value of `semncnt` for the member *semnum*.

`GETZCNT` Return the value of `semzcnt` for the member *semnum*.

`GETALL` Fetch all the semaphore values in the set. These values are stored in the array pointed to by *arg.array*.

`SETALL` Set all the semaphore values in the set to the values pointed to by *arg.array*.

For all the `GET` commands other than `GETALL`, the function returns the corresponding value. For the remaining commands, the return value is 0 if the call succeeds. On error, the `semctl` function sets `errno` and returns –1.

The function `semop` atomically performs an array of operations on a semaphore set.

```
#include <sys/sem.h>

int semop(int semid, struct sembuf semoparray[], size_t nops);
```

Returns: 0 if OK, –1 on error

The *semoparray* argument is a pointer to an array of semaphore operations, represented by `sembuf` structures:

```
struct sembuf {
  unsigned short  sem_num;  /* member # in set (0, 1, ..., nsems-1) */
  short           sem_op;   /* operation (negative, 0, or positive) */
  short           sem_flg;  /* IPC_NOWAIT, SEM_UNDO */
};
```

The *nops* argument specifies the number of operations (elements) in the array.

The operation on each member of the set is specified by the corresponding `sem_op` value. This value can be negative, 0, or positive. (In the following discussion, we refer to the "undo" flag for a semaphore. This flag corresponds to the `SEM_UNDO` bit in the corresponding `sem_flg` member.)

1. The easiest case is when `sem_op` is positive. This case corresponds to the returning of resources by the process. The value of `sem_op` is added to the semaphore's value. If the undo flag is specified, `sem_op` is also subtracted from the semaphore's adjustment value for this process.
2. If `sem_op` is negative, we want to obtain resources that the semaphore controls.

   If the semaphore's value is greater than or equal to the absolute value of `sem_op` (the resources are available), the absolute value of `sem_op` is subtracted from the semaphore's value. This guarantees the resulting semaphore value is greater than or equal to 0. If the undo flag is specified, the absolute value of `sem_op` is also added to the semaphore's adjustment value for this process.

   If the semaphore's value is less than the absolute value of `sem_op` (the resources are not available), the following conditions apply.

   a. If `IPC_NOWAIT` is specified, `semop` returns with an error of `EAGAIN`.
   b. If `IPC_NOWAIT` is not specified, the `semncnt` value for this semaphore is incremented (since the caller is about to go to sleep), and the calling process is suspended until one of the following occurs.
      i. The semaphore's value becomes greater than or equal to the absolute value of `sem_op` (i.e., some other process has released some resources). The value of `semncnt` for this semaphore is decremented (since the calling process is done waiting), and the absolute value of `sem_op` is subtracted from the semaphore's value. If the undo flag is specified, the absolute value of `sem_op` is also added to the semaphore's adjustment value for this process.
      ii. The semaphore is removed from the system. In this case, the function returns an error of `EIDRM`.
      iii. A signal is caught by the process, and the signal handler returns. In this case, the value of `semncnt` for this semaphore is decremented (since the calling process is no longer waiting), and the function returns an error of `EINTR`.
3. If `sem_op` is 0, this means that the calling process wants to wait until the semaphore's value becomes 0.

If the semaphore's value is currently 0, the function returns immediately.

If the semaphore's value is nonzero, the following conditions apply.

a. If IPC_NOWAIT is specified, return is made with an error of EAGAIN.

b. If IPC_NOWAIT is not specified, the semzcnt value for this semaphore is incremented (since the caller is about to go to sleep), and the calling process is suspended until one of the following occurs.

   i. The semaphore's value becomes 0. The value of semzcnt for this semaphore is decremented (since the calling process is done waiting).

   ii. The semaphore is removed from the system. In this case, the function returns an error of EIDRM.

   iii. A signal is caught by the process, and the signal handler returns. In this case, the value of semzcnt for this semaphore is decremented (since the calling process is no longer waiting), and the function returns an error of EINTR.

The semop function operates atomically; it does either all the operations in the array or none of them.

Semaphore Adjustment on exit

As we mentioned earlier, it is a problem if a process terminates while it has resources allocated through a semaphore. Whenever we specify the SEM_UNDO flag for a semaphore operation and we allocate resources (a sem_op value less than 0), the kernel remembers how many resources we allocated from that particular semaphore (the absolute value of sem_op). When the process terminates, either voluntarily or involuntarily, the kernel checks whether the process has any outstanding semaphore adjustments and, if so, applies the adjustment to the corresponding semaphore.

If we set the value of a semaphore using semctl, with either the SETVAL or SETALL commands, the adjustment value for that semaphore in all processes is set to 0.

Example — Timing Comparison of Semaphores, Record Locking, and Mutexes

If we are sharing a single resource among multiple processes, we can use one of three techniques to coordinate access. We can use a a semaphore, record locking, or a mutex that is mapped into the address spaces of both processes. It's interesting to compare the timing differences between the three techniques.

With a semaphore, we create a semaphore set consisting of a single member and initialize the semaphore's value to 1. To allocate the resource, we call semop with a sem_op of –1; to release the resource, we perform a sem_op of +1. We also specify SEM_UNDO with each operation, to handle the case of a process that terminates without releasing its resource.

With record locking, we create an empty file and use the first byte of the file (which need not exist) as the lock byte. To allocate the resource, we obtain a write lock on the

byte; to release it, we unlock the byte. The record locking properties guarantee that if a process terminates while holding a lock, the kernel automatically releases the lock.

To use a mutex, we need both processes to map the same file into their address spaces and initialize a mutex at the same offset in the file using the `PTHREAD_PROCESS_SHARED` mutex attribute. To allocate the resource, we lock the mutex; to release the resource, we unlock the mutex. If a process terminates without releasing the mutex, recovery is difficult unless we use a robust mutex (recall the `pthread_mutex_consistent` function discussed in Section 12.4.1).

Figure 15.29 shows the time required to perform these three locking techniques on Linux. In each case, the resource was allocated and then released 1,000,000 times. This was done simultaneously by three different processes. The times in Figure 15.29 are the totals in seconds for all three processes.

Operation	User	System	Clock
semaphores with undo	0.50	6.08	7.55
advisory record locking	0.51	9.06	4.38
mutex in shared memory	0.21	0.40	0.25

Figure 15.29 Timing comparison of locking alternatives on Linux

On Linux, record locking is faster than semaphores, but mutexes in shared memory outperform both semaphores and record locking. If we're locking a single resource and don't need all the fancy features of XSI semaphores, record locking is preferred over semaphores. The reasons are that it is much simpler to use, it is faster (on this platform), and the system takes care of any lingering locks when a process terminates. Even though using a mutex in shared memory is the fastest option on this platform, we still prefer to use record locking, unless performance is the primary concern. There are two reasons for this. First, recovery from process termination is more difficult using a mutex in memory shared among multiple processes. Second, the *process-shared* mutex attribute isn't universally supported yet. In older versions of the Single UNIX Specification, it was optional. Although it is still optional in SUSv4, it is now required by all XSI-conforming implementations.

> Of the four platforms covered in this text, only Linux 3.2.0 and Solaris 10 currently support the *process-shared* mutex attribute.
>
> □

15.9 Shared Memory

Shared memory allows two or more processes to share a given region of memory. This is the fastest form of IPC, because the data does not need to be copied between the client and the server. The only trick in using shared memory is synchronizing access to a given region among multiple processes. If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done. Often, semaphores are used to synchronize shared memory access. (But as we saw at the end of the previous section, record locking or mutexes can also be used.)

The Single UNIX Specification shared memory objects option includes alternative interfaces, originally real-time extensions, to access shared memory. We don't discuss them in this text.

We've already seen one form of shared memory when multiple processes map the same file into their address spaces. The XSI shared memory differs from memory-mapped files in that there is no associated file. The XSI shared memory segments are anonymous segments of memory.

The kernel maintains a structure with at least the following members for each shared memory segment:

```
struct shmid_ds {
  struct ipc_perm  shm_perm;    /* see Section 15.6.2 */
  size_t           shm_segsz;   /* size of segment in bytes */
  pid_t            shm_lpid;    /* pid of last shmop() */
  pid_t            shm_cpid;    /* pid of creator */
  shmatt_t         shm_nattch;  /* number of current attaches */
  time_t           shm_atime;   /* last-attach time */
  time_t           shm_dtime;   /* last-detach time */
  time_t           shm_ctime;   /* last-change time */
  .
  .
  .
};
```

(Implementations add other structure members to support shared memory segments.)

The type `shmatt_t` is defined to be an unsigned integer at least as large as an `unsigned short`. Figure 15.30 lists the system limits that affect shared memory.

Description	Typical values			
	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
maximum size in bytes of a shared memory segment	33,554,432	32,768	4,194,304	derived
minimum size in bytes of a shared memory segment	1	1	1	1
maximum number of shared memory segments, systemwide	192	4,096	32	128
maximum number of shared memory segments, per process	128	4,096	8	128

Figure 15.30 System limits that affect shared memory

The first function called is usually `shmget`, to obtain a shared memory identifier.

```
#include <sys/shm.h>

int shmget(key_t key, size_t size, int flag);
```

Returns: shared memory ID if OK, –1 on error

In Section 15.6.1, we described the rules for converting the *key* into an identifier and whether a new segment is created or an existing segment is referenced. When a new segment is created, the following members of the `shmid_ds` structure are initialized.

- The `ipc_perm` structure is initialized as described in Section 15.6.2. The `mode` member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the values from Figure 15.24.

- `shm_lpid`, `shm_nattch`, `shm_atime`, and `shm_dtime` are all set to 0.
- `shm_ctime` is set to the current time.
- `shm_segsz` is set to the *size* requested.

The *size* parameter is the size of the shared memory segment in bytes. Implementations will usually round up this size to a multiple of the system's page size, but if an application specifies *size* as a value other than an integral multiple of the system's page size, the remainder of the last page will be unavailable for use. If a new segment is being created (typically by the server), we must specify its *size*. If we are referencing an existing segment (a client), we can specify *size* as 0. When a new segment is created, the contents of the segment are initialized with zeros.

The `shmctl` function is the catchall for various shared memory operations.

```
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Returns: 0 if OK, –1 on error

The *cmd* argument specifies one of the following five commands to be performed, on the segment specified by *shmid*.

`IPC_STAT` Fetch the `shmid_ds` structure for this segment, storing it in the structure pointed to by *buf*.

`IPC_SET` Set the following three fields from the structure pointed to by *buf* in the `shmid_ds` structure associated with this shared memory segment: `shm_perm.uid`, `shm_perm.gid`, and `shm_perm.mode`. This command can be executed only by a process whose effective user ID equals `shm_perm.cuid` or `shm_perm.uid` or by a process with superuser privileges.

`IPC_RMID` Remove the shared memory segment set from the system. Since an attachment count is maintained for shared memory segments (the `shm_nattch` field in the `shmid_ds` structure), the segment is not removed until the last process using the segment terminates or detaches it. Regardless of whether the segment is still in use, the segment's identifier is immediately removed so that `shmat` can no longer attach the segment. This command can be executed only by a process whose effective user ID equals `shm_perm.cuid` or `shm_perm.uid` or by a process with superuser privileges.

Two additional commands are provided by Linux and Solaris, but are not part of the Single UNIX Specification.

`SHM_LOCK` Lock the shared memory segment in memory. This command can be executed only by the superuser.

`SHM_UNLOCK` Unlock the shared memory segment. This command can be executed only by the superuser.

Once a shared memory segment has been created, a process attaches it to its address space by calling `shmat`.

```
#include <sys/shm.h>

void *shmat(int shmid, const void *addr, int flag);
```

Returns: pointer to shared memory segment if OK, –1 on error

The address in the calling process at which the segment is attached depends on the *addr* argument and whether the `SHM_RND` bit is specified in *flag*.

- If *addr* is 0, the segment is attached at the first available address selected by the kernel. This is the recommended technique.
- If *addr* is nonzero and `SHM_RND` is not specified, the segment is attached at the address given by *addr*.
- If *addr* is nonzero and `SHM_RND` is specified, the segment is attached at the address given by (*addr* – (*addr* modulus `SHMLBA`)). The `SHM_RND` command stands for "round." `SHMLBA` stands for "low boundary address multiple" and is always a power of 2. What the arithmetic does is round the address down to the next multiple of `SHMLBA`.

Unless we plan to run the application on only a single type of hardware (which is highly unlikely today), we should not specify the address where the segment is to be attached. Instead, we should specify an *addr* of 0 and let the system choose the address.

If the `SHM_RDONLY` bit is specified in *flag*, the segment is attached as read-only. Otherwise, the segment is attached as read–write.

The value returned by `shmat` is the address at which the segment is attached, or –1 if an error occurred. If `shmat` succeeds, the kernel will increment the `shm_nattch` counter in the `shmid_ds` structure associated with the shared memory segment.

When we're done with a shared memory segment, we call `shmdt` to detach it. Note that this does not remove the identifier and its associated data structure from the system. The identifier remains in existence until some process (often a server) specifically removes it by calling `shmctl` with a command of `IPC_RMID`.

```
#include <sys/shm.h>

int shmdt(const void *addr);
```

Returns: 0 if OK, –1 on error

The *addr* argument is the value that was returned by a previous call to `shmat`. If successful, `shmdt` will decrement the `shm_nattch` counter in the associated `shmid_ds` structure.

Example

Where a kernel places shared memory segments that are attached with an address of 0 is highly system dependent. Figure 15.31 shows a program that prints some information on where one particular system places various types of data.

```
#include "apue.h"
#include <sys/shm.h>

#define ARRAY_SIZE  40000
#define MALLOC_SIZE 100000
#define SHM_SIZE    100000
#define SHM_MODE    0600    /* user read/write */

char    array[ARRAY_SIZE];  /* uninitialized data = bss */

int
main(void)
{
    int     shmid;
    char    *ptr, *shmptr;

    printf("array[] from %p to %p\n", (void *)&array[0],
      (void *)&array[ARRAY_SIZE]);
    printf("stack around %p\n", (void *)&shmid);

    if ((ptr = malloc(MALLOC_SIZE)) == NULL)
        err_sys("malloc error");
    printf("malloced from %p to %p\n", (void *)ptr,
      (void *)ptr+MALLOC_SIZE);

    if ((shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)
        err_sys("shmget error");
    if ((shmptr = shmat(shmid, 0, 0)) == (void *)-1)
        err_sys("shmat error");
    printf("shared memory attached from %p to %p\n", (void *)shmptr,
      (void *)shmptr+SHM_SIZE);

    if (shmctl(shmid, IPC_RMID, 0) < 0)
        err_sys("shmctl error");

    exit(0);
}
```

Figure 15.31 Print where various types of data are stored

Running this program on a 64-bit Intel-based Linux system gives us the following output:

```
$ ./a.out
array[] from 0x6020c0 to 0x60bd00
stack around 0x7fff957b146c
malloced from 0x9e3010 to 0x9fb6b0
shared memory attached from 0x7fba578ab000 to 0x7fba578c36a0
```

Figure 15.32 shows a picture of this, similar to what we said was a typical memory layout in Figure 7.6. Note that the shared memory segment is placed well below the stack. □

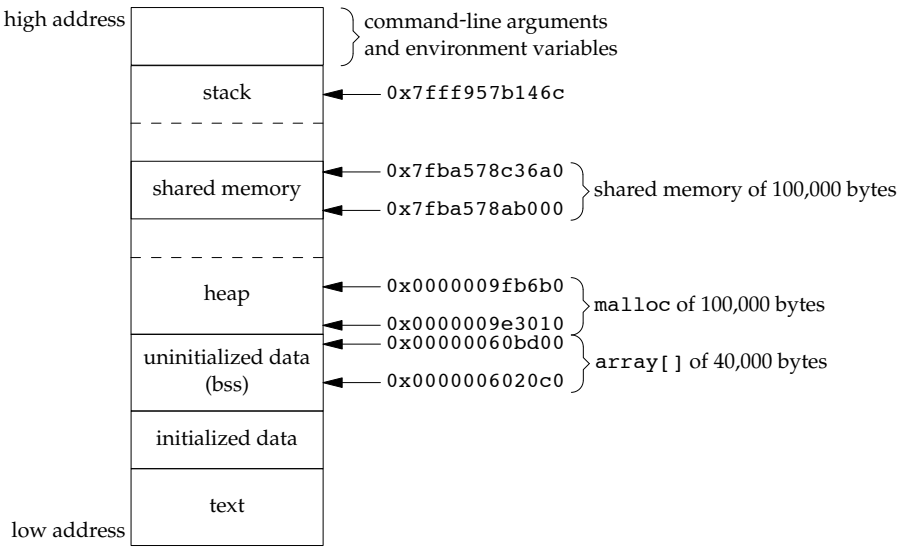


Figure 15.32 Memory layout on an Intel-based Linux system

Recall that the `mmap` function (Section 14.8) can be used to map portions of a file into the address space of a process. This is conceptually similar to attaching a shared memory segment using the `shmat` XSI IPC function. The main difference is that the memory segment mapped with `mmap` is backed by a file, whereas no file is associated with an XSI shared memory segment.

Example—Memory Mapping of `/dev/zero`

Shared memory can be used between unrelated processes. But if the processes are related, some implementations provide a different technique.

> The following technique works on FreeBSD 8.0, Linux 3.2.0, and Solaris 10. Mac OS X 10.6.8 currently doesn't support the mapping of character devices into the address space of a process.

The device `/dev/zero` is an infinite source of 0 bytes when read. This device also accepts any data that is written to it, ignoring the data. Our interest in this device for IPC arises from its special properties when it is memory mapped.

- An unnamed memory region is created whose size is the second argument to `mmap`, rounded up to the nearest page size on the system.
- The memory region is initialized to 0.
- Multiple processes can share this region if a common ancestor specifies the `MAP_SHARED` flag to `mmap`.

The program in Figure 15.33 is an example that uses this special device.

```
#include "apue.h"
#include <fcntl.h>
#include <sys/mman.h>

#define NLOOPS      1000
#define SIZE        sizeof(long)    /* size of shared memory area */

static int
update(long *ptr)
{
    return((*ptr)++);   /* return value before increment */
}

int
main(void)
{
    int     fd, i, counter;
    pid_t   pid;
    void    *area;

    if ((fd = open("/dev/zero", O_RDWR)) < 0)
        err_sys("open error");
    if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
      fd, 0)) == MAP_FAILED)
        err_sys("mmap error");
    close(fd);      /* can close /dev/zero now that it's mapped */

    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {           /* parent */
        for (i = 0; i < NLOOPS; i += 2) {
            if ((counter = update((long *)area)) != i)
                err_quit("parent: expected %d, got %d", i, counter);

            TELL_CHILD(pid);
            WAIT_CHILD();
        }
    } else {                        /* child */
        for (i = 1; i < NLOOPS + 1; i += 2) {
            WAIT_PARENT();

            if ((counter = update((long *)area)) != i)
                err_quit("child: expected %d, got %d", i, counter);

            TELL_PARENT(getppid());
        }
    }

    exit(0);
}
```

Figure 15.33 IPC between parent and child using memory mapped I/O of `/dev/zero`

The program opens the /dev/zero device and calls mmap, specifying a size of a long integer. Note that once the region is mapped, we can close the device. The process then creates a child. Since MAP_SHARED was specified in the call to mmap, writes to the memory-mapped region by one process are seen by the other process. (If we had specified MAP_PRIVATE instead, this example wouldn't work.)

The parent and the child then alternate running, incrementing a long integer in the shared memory-mapped region, using the synchronization functions from Section 8.9. The memory-mapped region is initialized to 0 by mmap. The parent increments it to 1, then the child increments it to 2, then the parent increments it to 3, and so on. Note that we have to use parentheses when we increment the value of the long integer in the update function, since we are incrementing the value and not the pointer.

The advantage of using /dev/zero in the manner that we've shown is that an actual file need not exist before we call mmap to create the mapped region. Mapping /dev/zero automatically creates a mapped region of the specified size. The disadvantage of this technique is that it works only between related processes. With related processes, however, it is probably simpler and more efficient to use threads (Chapters 11 and 12). Note that no matter which technique is used, we still need to synchronize access to the shared data. □

Example — Anonymous Memory Mapping

Many implementations provide anonymous memory mapping, a facility similar to the /dev/zero feature. To use this facility, we specify the MAP_ANON flag to mmap and specify the file descriptor as –1. The resulting region is anonymous (since it's not associated with a pathname through a file descriptor) and creates a memory region that can be shared with descendant processes.

> The anonymous memory-mapping facility is supported by all four platforms discussed in this text. Note, however, that Linux defines the MAP_ANONYMOUS flag for this facility, but defines the MAP_ANON flag to be the same value for improved application portability.

To modify the program in Figure 15.33 to use this facility, we make three changes: (a) remove the open of /dev/zero, (b) remove the close of fd, and (c) change the call to mmap to the following:

```
if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
                 MAP_ANON | MAP_SHARED, -1, 0)) == MAP_FAILED)
```

In this call, we specify the MAP_ANON flag and set the file descriptor to –1. The rest of the program from Figure 15.33 is unchanged. □

The last two examples illustrate sharing memory among multiple related processes. If shared memory is required between unrelated processes, there are two alternatives. Applications can use the XSI shared memory functions, or they can use mmap to map the same file into their address spaces using the MAP_SHARED flag.

15.10 POSIX Semaphores

The POSIX semaphore mechanism is one of three IPC mechanisms that originated with the real-time extensions to POSIX.1. The Single UNIX Specification placed the three mechanisms (message queues, semaphores, and shared memory) in option classes. Prior to SUSv4, the POSIX semaphore interfaces were included in the semaphores option. In SUSv4, these interfaces were moved to the base specification, but the message queue and shared memory interfaces remained optional.

The POSIX semaphore interfaces were meant to address several deficiencies with the XSI semaphore interfaces:

- The POSIX semaphore interfaces allow for higher-performance implementations compared to XSI semaphores.
- The POSIX semaphore interfaces are simpler to use: there are no semaphore sets, and several of the interfaces are patterned after familiar file system operations. Although there is no requirement that they be implemented in the file system, some systems do take this approach.
- The POSIX semaphores behave more gracefully when removed. Recall that when an XSI semaphore is removed, operations using the same semaphore identifier fail with `errno` set to `EIDRM`. With POSIX semaphores, operations continue to work normally until the last reference to the semaphore is released.

POSIX semaphores are available in two flavors: named and unnamed. They differ in how they are created and destroyed, but otherwise work the same. Unnamed semaphores exist in memory only and require that processes have access to the memory to be able to use the semaphores. This means they can be used only by threads in the same process or threads in different processes that have mapped the same memory extent into their address spaces. Named semaphores, in contrast, are accessed by name and can be used by threads in any processes that know their names.

To create a new named semaphore or use an existing one, we call the `sem_open` function.

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ... /* mode_t mode,
                unsigned int value */ );
```

Returns: Pointer to semaphore if OK, `SEM_FAILED` on error

When using an existing named semaphore, we specify only two arguments: the name of the semaphore and a zero value for the *oflag* argument. When the *oflag* argument has the `O_CREAT` flag set, we create a new named semaphore if it does not yet exist. If it already exists, it is opened for use, but no additional initialization takes place.

When we specify the `O_CREAT` flag, we need to provide two additional arguments. The *mode* argument specifies who can access the semaphore. It can take on the same values as the permission bits for opening a file: user-read, user-write, user-execute, group-read, group-write, group-execute, other-read, other-write, and other-execute. The resulting permissions assigned to the semaphore are modified by the caller's file

creation mask (Sections 4.5 and 4.8). Note, however, that only read and write access matter, but the interfaces don't allow us to specify the mode when we open an existing semaphore. Implementations usually open semaphores for both reading and writing.

The *value* argument is used to specify the initial value for the semaphore when we create it. It can take on any value from 0 to `SEM_VALUE_MAX` (Figure 2.9).

If we want to ensure that we are creating the semaphore, we can set the *oflag* argument to `O_CREAT|O_EXCL`. This will cause `sem_open` to fail if the semaphore already exists.

To promote portability, we must follow certain conventions when selecting a semaphore name.

- The first character in the name should be a slash (/). Although there is no requirement that an implementation of POSIX semaphores uses the file system, if the file system *is* used, we want to remove any ambiguity as to the starting point from which the name is interpreted.
- The name should contain no other slashes to avoid implementation-defined behavior. For example, if the file system is used, the names `/mysem` and `//mysem` would evaluate to the same filename, but if the implementation doesn't use the file system, the two names could be treated as different (consider what would happen if the implementation hashed the name to an integer value used to identify the semaphore).
- The maximum length of the semaphore name is implementation defined. The name should be no longer than `_POSIX_NAME_MAX` (Figure 2.8) characters, because this is the minimum acceptable limit to the maximum name length if the implementation does use the file system.

The `sem_open` function returns a semaphore pointer that we can pass to other semaphore functions when we want to operate on the semaphore. When we are done with the semaphore, we can call the `sem_close` function to release any resources associated with the semaphore.

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

Returns: 0 if OK, –1 on error

If our process exits without having first called `sem_close`, the kernel will close any open semaphores automatically. Note that this doesn't affect the state of the semaphore value—if we have incremented its value, this doesn't change just because we exit. Similarly, if we call `sem_close`, the semaphore value is unaffected. There is no mechanism equivalent to the `SEM_UNDO` flag found with XSI semaphores.

To destroy a named semaphore, we can use the `sem_unlink` function.

```
#include <semaphore.h>

int sem_unlink(const char *name);
```

Returns: 0 if OK, –1 on error

The sem_unlink function removes the name of the semaphore. If there are no open references to the semaphore, then it is destroyed. Otherwise, destruction is deferred until the last open reference is closed.

Unlike with XSI semaphores, we can only adjust the value of a POSIX semaphore by one with a single function call. Decrementing the count is analogous to locking a binary semaphore or acquiring a resource associated with a counting semaphore.

> Note that there is no distinction of semaphore type with POSIX semaphores. Whether we use a binary semaphore or a counting semaphore depends on how we initialize and use the semaphore. If a semaphore only ever has a value of 0 or 1, then it is a binary semaphore. When a binary semaphore has a value of 1, we say that it is "unlocked;" when it has a value of 0, we say that it is "locked."

To decrement the value of a semaphore, we can use either the sem_wait or sem_trywait function.

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);

int sem_wait(sem_t *sem);
```

Both return: 0 if OK, –1 on error

With the sem_wait function, we will block if the semaphore count is 0. We won't return until we have successfully decremented the semaphore count or are interrupted by a signal. We can use the sem_trywait function to avoid blocking. If the semaphore count is zero when we call sem_trywait, it will return –1 with errno set to EAGAIN instead of blocking.

A third alternative is to block for a bounded amount of time. We can use the sem_timedwait function for this purpose.

```
#include <semaphore.h>
#include <time.h>

int sem_timedwait(sem_t *restrict sem,
                  const struct timespec *restrict tsptr);
```

Returns: 0 if OK, –1 on error

The *tsptr* argument specifies the absolute time when we want to give up waiting for the semaphore. The timeout is based on the CLOCK_REALTIME clock (recall Figure 6.8). If the semaphore can be decremented immediately, then the value of the timeout doesn't matter—even though it might specify some time in the past, the attempt to decrement the semaphore will still succeed. If the timeout expires without being able to decrement the semaphore count, then sem_timedwait will return –1 with errno set to ETIMEDOUT.

To increment the value of a semaphore, we call the sem_post function. This is analogous to unlocking a binary semaphore or releasing a resource associated with a counting semaphore.

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Returns: 0 if OK, –1 on error

If a process is blocked in a call to `sem_wait` (or `sem_timedwait`) when we call `sem_post`, the process is awakened and the semaphore count that was just incremented by `sem_post` is decremented by `sem_wait` (or `sem_timedwait`).

When we want to use POSIX semaphores within a single process, it is easier to use unnamed semaphores. This only changes the way we create and destroy the semaphore. To create an unnamed semaphore, we call the `sem_init` function.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Returns: 0 if OK, –1 on error

The *pshared* argument indicates if we plan to use the semaphore with multiple processes. If so, we set it to a nonzero value. The *value* argument specifies the initial value of the semaphore.

Instead of returning a pointer to the semaphore like `sem_open` does, we need to declare a variable of type `sem_t` and pass its address to `sem_init` for initialization. If we plan to use the semaphore between two processes, we need to ensure that the *sem* argument points into the memory extent that is shared between the processes.

When we are done using the unnamed semaphore, we can discard it by calling the `sem_destroy` function.

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

Returns: 0 if OK, –1 on error

After calling `sem_destroy`, we can't use any of the semaphore functions with *sem* unless we reinitialize it by calling `sem_init` again.

One other function is available to allow us to retrieve the value of a semaphore. We call the `sem_getvalue` function for this purpose.

```
#include <semaphore.h>

int sem_getvalue(sem_t *restrict sem, int *restrict valp);
```

Returns: 0 if OK, –1 on error

On success, the integer pointed to by the *valp* argument will contain the value of the semaphore. Be aware, however, that the value of the semaphore can change by the time that we try to use the value we just read. Unless we use additional synchronization mechanisms to avoid this race, the `sem_getvalue` function is useful only for debugging.

The `sem_getvalue` function is not supported by Mac OS X 10.6.8.

Example

One of the motivations for introducing the POSIX semaphore interfaces was that they can be made to perform significantly better than the existing XSI semaphore interfaces. It is instructive to see if this goal was reached in existing systems, even though these systems were not designed to support real-time applications.

In Figure 15.34, we compare the performance of using XSI semaphores (without `SEM_UNDO`) and POSIX semaphores when 3 processes compete to allocate and release the semaphore 1,000,000 times on two platforms (Linux 3.2.0 and Solaris 10).

Operation	Solaris 10			Linux 3.2.0		
	User	System	Clock	User	System	Clock
XSI semaphores	11.85	15.85	27.91	0.33	5.93	7.33
POSIX semaphores	13.72	10.52	24.44	0.26	0.75	0.41

Figure 15.34 Timing comparison of semaphore implementations

In Figure 15.34, we can see that POSIX semaphores provide only a 12% improvement over XSI semaphores on Solaris, but on Linux the improvement is 94% (almost 18 times faster)! If we trace the programs, we find that the Linux implementation of POSIX semaphores maps the file into the process address space and performs individual semaphore operations without using system calls. □

Example

Recall from Figure 12.5 that the Single UNIX Specification doesn't define what happens when one thread locks a normal mutex and a different thread tries to unlock it, but that error-checking mutexes and recursive mutexes generate errors in this case. Because a binary semaphore can be used like a mutex, we can use a semaphore to create our own locking primitive to provide mutual exclusion.

Assuming we were to create our own lock that could be locked by one thread and unlocked by another, our lock structure might look like

```
struct slock {
  sem_t *semp;
  char   name[_POSIX_NAME_MAX];
};
```

The program in Figure 15.35 shows an implementation of a semaphore-based mutual exclusion primitive.

```
#include "slock.h"
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

struct slock *
```

```
s_alloc()
{
    struct slock *sp;
    static int cnt;

    if ((sp = malloc(sizeof(struct slock))) == NULL)
        return(NULL);
    do {
        snprintf(sp->name, sizeof(sp->name), "/%ld.%d", (long)getpid(),
          cnt++);
        sp->semp = sem_open(sp->name, O_CREAT|O_EXCL, S_IRWXU, 1);
    } while ((sp->semp == SEM_FAILED) && (errno == EEXIST));
    if (sp->semp == SEM_FAILED) {
        free(sp);
        return(NULL);
    }
    sem_unlink(sp->name);
    return(sp);
}

void
s_free(struct slock *sp)
{
    sem_close(sp->semp);
    free(sp);
}

int
s_lock(struct slock *sp)
{
    return(sem_wait(sp->semp));
}

int
s_trylock(struct slock *sp)
{
    return(sem_trywait(sp->semp));
}

int
s_unlock(struct slock *sp)
{
    return(sem_post(sp->semp));
}
```

Figure 15.35 Mutual exclusion using a POSIX semaphore

We create a name based on the process ID and a counter. We don't bother to protect the counter with a mutex, because if two racing threads call `s_alloc` at the same time and end up with the same name, using the `O_EXCL` flag in the call to `sem_open` will cause one to succeed and one to fail with `errno` set to `EEXIST`, so we just retry if this happens. Note that we unlink the semaphore after opening it. This destroys the name so that no other process can access it and simplifies cleanup when the process ends. □

15.11 Client–Server Properties

Let's detail some of the properties of clients and servers that are affected by the various types of IPC used between them. The simplest type of relationship is to have the client `fork` and `exec` the desired server. Two half-duplex pipes can be created before the `fork` to allow data to be transferred in both directions. Figure 15.16 is an example of this arrangement. The server that is executed can be a set-user-ID program, giving it special privileges. Also, the server can determine the real identity of the client by looking at its real user ID. (Recall from Section 8.10 that the real user ID and real group ID don't change across an `exec`.)

With this arrangement, we can build an *open server*. (We show an implementation of this client–server mechanism in Section 17.5.) It opens files for the client instead of the client calling the `open` function. This way, additional permission checking can be added, above and beyond the normal UNIX system user/group/other permissions. We assume that the server is a set-user-ID program, giving it additional permissions (root permission, perhaps). The server uses the real user ID of the client to determine whether to give it access to the requested file. This way, we can build a server that allows certain users permissions that they don't normally have.

In this example, since the server is a child of the parent, all the server can do is pass back the contents of the file to the parent. Although this works fine for regular files, it can't be used for special device files, for example. We would like to be able to have the server open the requested file and pass back the file descriptor. Whereas a parent can pass a child an open descriptor, a child cannot pass a descriptor back to the parent (unless special programming techniques are used, which we cover in Chapter 17).

We showed the next type of server in Figure 15.23. The server is a daemon process that is contacted using some form of IPC by all clients. We can't use pipes for this type of client–server arrangement. A form of named IPC is required, such as FIFOs or message queues. With FIFOs, we saw that an individual per-client FIFO is also required if the server is to send data back to the client. If the client–server application sends data only from the client to the server, a single well-known FIFO suffices. (The System V line printer spooler used this form of client–server arrangement. The client was the `lp`(1) command, and the server was the `lpsched` daemon process. A single FIFO was used, since the flow of data was only from the client to the server. Nothing was sent back to the client.)

Multiple possibilities exist with message queues.

1. A single queue can be used between the server and all the clients, using the type field of each message to indicate the message recipient. For example, the clients can send their requests with a type field of 1. Included in the request must be the client's process ID. The server then sends the response with the type field set to the client's process ID. The server receives only the messages with a type field of 1 (the fourth argument for `msgrcv`), and the clients receive only the messages with a type field equal to their process IDs.

2. Alternatively, an individual message queue can be used for each client. Before sending the first request to a server, each client creates its own message queue

with a key of IPC_PRIVATE. The server also has its own queue, with a key or identifier known to all clients. The client sends its first request to the server's well-known queue, and this request must contain the message queue ID of the client's queue. The server sends its first response to the client's queue, and all future requests and responses are exchanged on this queue.

One problem with this technique is that each client-specific queue usually has only a single message on it: a request for the server or a response for a client. This seems wasteful of a limited systemwide resource (a message queue), and a FIFO can be used instead. Another problem is that the server has to read messages from multiple queues. Neither select nor poll works with message queues.

Either of these two techniques using message queues can be implemented using shared memory segments and a synchronization method (a semaphore or record locking).

The problem with this type of client–server relationship (the client and the server being unrelated processes) is for the server to identify the client accurately. Unless the server is performing a nonprivileged operation, it is essential that the server know who the client is. This is required, for example, if the server is a set-user-ID program. Although all these forms of IPC go through the kernel, there is no facility provided by them to have the kernel identify the sender.

With message queues, if a single queue is used between the client and the server (so that only a single message is on the queue at a time, for example), the msg_lspid of the queue contains the process ID of the other process. But when writing the server, we want the effective user ID of the client, not its process ID. There is no portable way to obtain the effective user ID, given the process ID. (Naturally, the kernel maintains both values in the process table entry, but other than rummaging around through the kernel's memory, we can't obtain one, given the other.)

We'll use the following technique in Section 17.2 to allow the server to identify the client. The same technique can be used with FIFOs, message queues, semaphores, and shared memory. For the following description, assume that FIFOs are being used, as in Figure 15.23. The client must create its own FIFO and set the file access permissions of the FIFO so that only user-read and user-write are on. We assume that the server has superuser privileges (or else it probably wouldn't care about the client's true identity), so the server can still read and write to this FIFO. When the server receives the client's first request on the server's well-known FIFO (which must contain the identity of the client-specific FIFO), the server calls either stat or fstat on the client-specific FIFO. The server assumes that the effective user ID of the client is the owner of the FIFO (the st_uid field of the stat structure). The server verifies that only the user-read and user-write permissions are enabled. As another check, the server should look at the three times associated with the FIFO (the st_atime, st_mtime, and st_ctime fields of the stat structure) to verify that they are recent (no older than 15 or 30 seconds, for example). If a malicious client can create a FIFO with someone else as the owner and set the file's permission bits to user-read and user-write only, then the system has other fundamental security problems.

To use this technique with XSI IPC, recall that the `ipc_perm` structure associated with each message queue, semaphore, and shared memory segment identifies the creator of the IPC structure (the `cuid` and `cgid` fields). As with the example using FIFOs, the server should require the client to create the IPC structure and have the client set the access permissions to user-read and user-write only. The times associated with the IPC structure should also be verified by the server to be recent (since these IPC structures hang around until explicitly deleted).

We'll see in Section 17.3 that a far better way of doing this authentication is for the kernel to provide the effective user ID and effective group ID of the client. This is done by the socket subsystem when file descriptors are passed between processes.

15.12 Summary

We've detailed numerous forms of interprocess communication: pipes, named pipes (FIFOs), the three forms of IPC commonly called XSI IPC (message queues, semaphores, and shared memory), and an alternative semaphore mechanism provided by POSIX. Semaphores are really a synchronization primitive, not true IPC, and are often used to synchronize access to a shared resource, such as a shared memory segment. With pipes, we looked at the implementation of the `popen` function, at coprocesses, and at the pitfalls that can be encountered with the standard I/O library's buffering.

After comparing the timing of message queues versus full-duplex pipes, and semaphores versus record locking, we can make the following recommendations: learn pipes and FIFOs, since these two basic techniques can still be used effectively in numerous applications. Avoid using message queues and semaphores in any new applications. Full-duplex pipes and record locking should be considered instead, as they are far simpler. Shared memory still has its use, although the same functionality can be provided through the use of the `mmap` function (Section 14.8).

In the next chapter, we will look at network IPC, which can allow processes to communicate across machine boundaries.

Exercises

15.1 In the program shown in Figure 15.6, remove the `close` right before the `waitpid` at the end of the parent code. Explain what happens.

15.2 In the program in Figure 15.6, remove the `waitpid` at the end of the parent code. Explain what happens.

15.3 What happens if the argument to `popen` is a nonexistent command? Write a small program to test this.

15.4 In the program shown in Figure 15.18, remove the signal handler, execute the program, and then terminate the child. After entering a line of input, how can you tell that the parent was terminated by `SIGPIPE`?

15.5 In the program in Figure 15.18, use the standard I/O library for reading and writing the pipes instead of `read` and `write`.

15.6 The Rationale for POSIX.1 gives as one of the reasons for adding the `waitpid` function that most pre-POSIX.1 systems can't handle the following:

```
if ((fp = popen("/bin/true", "r")) == NULL)
    ...
if ((rc = system("sleep 100")) == -1)
    ...
if (pclose(fp) == -1)
    ...
```

What happens in this code if `waitpid` isn't available and `wait` is used instead?

15.7 Explain how `select` and `poll` handle an input descriptor that is a pipe, when the pipe is closed by the writer. To determine the answer, write two small test programs: one using `select` and one using `poll`.

Redo this exercise, looking at an output descriptor that is a pipe, when the read end is closed.

15.8 What happens if the *cmdstring* executed by `popen` with a *type* of `"r"` writes to its standard error?

15.9 Since `popen` invokes a shell to execute its *cmdstring* argument, what happens when `cmdstring` terminates? (Hint: Draw all the processes involved.)

15.10 POSIX.1 specifically states that opening a FIFO for read–write is undefined. Although most UNIX systems allow this, show another method for opening a FIFO for both reading and writing, without blocking.

15.11 Unless a file contains sensitive or confidential data, allowing other users to read the file causes no harm. (It is usually considered antisocial, however, to go snooping around in other people's files.) But what happens if a malicious process reads a message from a message queue that is being used by a server and several clients? What information does the malicious process need to know to read the message queue?

15.12 Write a program that does the following. Execute a loop five times: create a message queue, print the queue identifier, delete the message queue. Then execute the next loop five times: create a message queue with a key of `IPC_PRIVATE`, and place a message on the queue. After the program terminates, look at the message queues using `ipcs`(1). Explain what is happening with the queue identifiers.

15.13 Describe how to build a linked list of data objects in a shared memory segment. What would you store as the list pointers?

15.14 Draw a timeline of the program in Figure 15.33 showing the value of the variable `i` in both the parent and child, the value of the long integer in the shared memory region, and the value returned by the `update` function. Assume that the child runs first after the `fork`.

15.15 Redo the program in Figure 15.33 using the XSI shared memory functions from Section 15.9 instead of the shared memory-mapped region.

15.16 Redo the program in Figure 15.33 using the XSI semaphore functions from Section 15.8 to alternate between the parent and the child.

15.17 Redo the program in Figure 15.33 using advisory record locking to alternate between the parent and the child.

15.18 Redo the program in Figure 15.33 using the POSIX semaphore functions from Section 15.10 to alternate between the parent and the child.

16

Network IPC: Sockets

16.1 Introduction

In the previous chapter, we looked at pipes, FIFOs, message queues, semaphores, and shared memory—the classical methods of IPC provided by various UNIX systems. These mechanisms allow processes running on the same computer to communicate with one another. In this chapter, we look at the mechanisms that allow processes running on different computers (connected to a common network) to communicate with one another—network IPC.

In this chapter, we describe the socket network IPC interface, which can be used by processes to communicate with other processes, regardless of where they are running—on the same machine or on different machines. Indeed, this was one of the design goals of the socket interface. The same interfaces can be used for both intermachine communication and intramachine communication. Although the socket interface can be used to communicate using many different network protocols, we will restrict our discussion to the TCP/IP protocol suite in this chapter, since it is the de facto standard for communicating over the Internet.

The socket API as specified by POSIX.1 is based on the 4.4BSD socket interface. Although minor changes have been made over the years, the current socket interface closely resembles the interface when it was originally introduced in 4.2BSD in the early 1980s.

This chapter is only an overview of the socket API. Stevens, Fenner, and Rudoff [2004] discuss the socket interface in detail in the definitive text on network programming in the UNIX System.

16.2 Socket Descriptors

A socket is an abstraction of a communication endpoint. Just as they would use file descriptors to access files, applications use socket descriptors to access sockets. Socket descriptors are implemented as file descriptors in the UNIX System. Indeed, many of the functions that deal with file descriptors, such as `read` and `write`, will work with a socket descriptor.

To create a socket, we call the `socket` function.

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Returns: file (socket) descriptor if OK, –1 on error

The *domain* argument determines the nature of the communication, including the address format (described in more detail in the next section). Figure 16.1 summarizes the domains specified by POSIX.1. The constants start with `AF_` (for *address family*) because each domain has its own format for representing an address.

Domain	Description
`AF_INET`	IPv4 Internet domain
`AF_INET6`	IPv6 Internet domain (optional in POSIX.1)
`AF_UNIX`	UNIX domain
`AF_UNSPEC`	unspecified

Figure 16.1 Socket communication domains

We discuss the UNIX domain in Section 17.2. Most systems define the `AF_LOCAL` domain also, which is an alias for `AF_UNIX`. The `AF_UNSPEC` domain is a wildcard that represents "any" domain. Historically, some platforms provide support for additional network protocols, such as `AF_IPX` for the NetWare protocol family, but domain constants for these protocols are not defined by the POSIX.1 standard.

The *type* argument determines the type of the socket, which further determines the communication characteristics. The socket types defined by POSIX.1 are summarized in Figure 16.2, but implementations are free to add support for additional types.

Type	Description
`SOCK_DGRAM`	fixed-length, connectionless, unreliable messages
`SOCK_RAW`	datagram interface to IP (optional in POSIX.1)
`SOCK_SEQPACKET`	fixed-length, sequenced, reliable, connection-oriented messages
`SOCK_STREAM`	sequenced, reliable, bidirectional, connection-oriented byte streams

Figure 16.2 Socket types

The *protocol* argument is usually zero, to select the default protocol for the given domain and socket type. When multiple protocols are supported for the same domain and socket type, we can use the *protocol* argument to select a particular protocol. The default protocol for a `SOCK_STREAM` socket in the `AF_INET` communication domain is

TCP (Transmission Control Protocol). The default protocol for a SOCK_DGRAM socket in the AF_INET communication domain is UDP (User Datagram Protocol). Figure 16.3 lists the protocols defined for the Internet domain sockets.

Protocol	Description
IPPROTO_IP	IPv4 Internet Protocol
IPPROTO_IPV6	IPv6 Internet Protocol (optional in POSIX.1)
IPPROTO_ICMP	Internet Control Message Protocol
IPPROTO_RAW	Raw IP packets protocol (optional in POSIX.1)
IPPROTO_TCP	Transmission Control Protocol
IPPROTO_UDP	User Datagram Protocol

Figure 16.3 Protocols defined for Internet domain sockets

With a datagram (SOCK_DGRAM) interface, no logical connection needs to exist between peers for them to communicate. All you need to do is send a message addressed to the socket being used by the peer process.

A datagram, therefore, provides a connectionless service. A byte stream (SOCK_STREAM), in contrast, requires that, before you can exchange data, you set up a logical connection between your socket and the socket belonging to the peer with which you wish to communicate.

A datagram is a self-contained message. Sending a datagram is analogous to mailing someone a letter. You can mail many letters, but you can't guarantee the order of delivery, and some might get lost along the way. Each letter contains the address of the recipient, making the letter independent from all the others. Each letter can even go to different recipients.

In contrast, using a connection-oriented protocol for communicating with a peer is like making a phone call. First, you need to establish a connection by placing a phone call, but after the connection is in place, you can communicate bidirectionally with each other. The connection is a peer-to-peer communication channel over which you talk. Your words contain no addressing information, as a point-to-point virtual connection exists between both ends of the call, and the connection itself implies a particular source and destination.

A SOCK_STREAM socket provides a byte-stream service; applications are unaware of message boundaries. This means that when we read data from a SOCK_STREAM socket, it might not return the same number of bytes written by the sender. We will eventually get everything sent to us, but it might take several function calls.

A SOCK_SEQPACKET socket is just like a SOCK_STREAM socket except that we get a message-based service instead of a byte-stream service. This means that the amount of data received from a SOCK_SEQPACKET socket is the same amount as was written. The Stream Control Transmission Protocol (SCTP) provides a sequential packet service in the Internet domain.

A SOCK_RAW socket provides a datagram interface directly to the underlying network layer (which means IP in the Internet domain). Applications are responsible for building their own protocol headers when using this interface, because the transport protocols (TCP and UDP, for example) are bypassed. Superuser privileges are required

to create a raw socket to prevent malicious applications from creating packets that might bypass established security mechanisms.

Calling `socket` is similar to calling `open`. In both cases, you get a file descriptor that can be used for I/O. When you are done using the file descriptor, you call `close` to relinquish access to the file or socket and free up the file descriptor for reuse.

Although a socket descriptor is actually a file descriptor, you can't use a socket descriptor with every function that accepts a file descriptor argument. Figure 16.4 summarizes most of the functions we've described so far that are used with file descriptors and describes how they behave when used with socket descriptors. Unspecified and implementation-defined behavior usually means that the function doesn't work with socket descriptors. For example, `lseek` doesn't work with sockets, since sockets don't support the concept of a file offset.

Function	Behavior with socket
`close` (Section 3.3)	deallocates the socket
`dup`, `dup2` (Section 3.12)	duplicates the file descriptor as normal
`fchdir` (Section 4.23)	fails with `errno` set to `ENOTDIR`
`fchmod` (Section 4.9)	unspecified
`fchown` (Section 4.11)	implementation defined
`fcntl` (Section 3.14)	some commands supported, including `F_DUPFD`, `F_DUPFD_CLOEXEC`, `F_GETFD`, `F_GETFL`, `F_GETOWN`, `F_SETFD`, `F_SETFL`, and `F_SETOWN`
`fdatasync`, `fsync` (Section 3.13)	implementation defined
`fstat` (Section 4.2)	some `stat` structure members supported, but how left up to the implementation
`ftruncate` (Section 4.13)	unspecified
`ioctl` (Section 3.15)	some commands work, depending on underlying device driver
`lseek` (Section 3.6)	implementation defined (usually fails with `errno` set to `ESPIPE`)
`mmap` (Section 14.8)	unspecified
`poll` (Section 14.4.2)	works as expected
`pread` and `pwrite` (Section 3.11)	fails with `errno` set to `ESPIPE`
`read` (Section 3.7) and `readv` (Section 14.6)	equivalent to `recv` (Section 16.5) without any flags
`select` (Section 14.4.1)	works as expected
`write` (Section 3.8) and `writev` (Section 14.6)	equivalent to `send` (Section 16.5) without any flags

Figure 16.4 How file descriptor functions act with sockets

Communication on a socket is bidirectional. We can disable I/O on a socket with the `shutdown` function.

```
#include <sys/socket.h>

int shutdown(int sockfd, int how);
```

Returns: 0 if OK, –1 on error

If *how* is `SHUT_RD`, then reading from the socket is disabled. If *how* is `SHUT_WR`, then we can't use the socket for transmitting data. We can use `SHUT_RDWR` to disable both data transmission and reception.

Given that we can `close` a socket, why is `shutdown` needed? There are several reasons. First, `close` will deallocate the network endpoint only when the last active reference is closed. If we duplicate the socket (with `dup`, for example), the socket won't be deallocated until we close the last file descriptor referring to it. The `shutdown` function allows us to deactivate a socket independently of the number of active file descriptors referencing it. Second, it is sometimes convenient to shut a socket down in one direction only. For example, we can shut a socket down for writing if we want the process we are communicating with to be able to tell when we are done transmitting data, while still allowing us to use the socket to receive data sent to us by the process.

16.3 Addressing

In the previous section, we learned how to create and destroy a socket. Before we learn to do something useful with a socket, we need to learn how to identify the process with which we wish to communicate. Identifying the process has two components. The machine's network address helps us identify the computer on the network we wish to contact, and the service, represented by a *port number*, helps us identify the particular process on the computer.

16.3.1 Byte Ordering

When communicating with processes running on the same computer, we generally don't have to worry about byte ordering. The byte order is a characteristic of the processor architecture, dictating how bytes are ordered within larger data types, such as integers. Figure 16.5 shows how the bytes within a 32-bit integer are numbered.

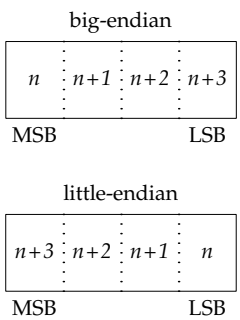


Figure 16.5 Byte order in a 32-bit integer

If the processor architecture supports *big-endian* byte order, then the highest byte address occurs in the least significant byte (LSB). *Little-endian* byte order is the opposite: the least significant byte contains the lowest byte address. Note that regardless of the byte ordering, the most significant byte (MSB) is always on the left, and the least significant byte is always on the right. Thus, if we were to assign a 32-bit integer the value `0x04030201`, the most significant byte would contain 4, and the least significant

byte would contain 1, regardless of the byte ordering. If we were then to cast a character pointer (`cp`) to the address of the integer, we would see a difference from the byte ordering. On a little-endian processor, `cp[0]` would refer to the least significant byte and contain 1; `cp[3]` would refer to the most significant byte and contain 4. Compare that to a big-endian processor, where `cp[0]` would contain 4, referring to the most significant byte, and `cp[3]` would contain 1, referring to the least significant byte. Figure 16.6 summarizes the byte ordering for the four platforms discussed in this text.

Operating system	Processor architecture	Byte order
FreeBSD 8.0	Intel Pentium	little-endian
Linux 3.2.0	Intel Core i5	little-endian
Mac OS X 10.6.8	Intel Core 2 Duo	little-endian
Solaris 10	Sun SPARC	big-endian

Figure 16.6 Byte order for test platforms

To confuse matters further, some processors can be configured for either little-endian or big-endian operation.

Network protocols specify a byte ordering so that heterogeneous computer systems can exchange protocol information without confusing the byte ordering. The TCP/IP protocol suite uses big-endian byte order. The byte ordering becomes visible to applications when they exchange formatted data. With TCP/IP, addresses are presented in network byte order, so applications sometimes need to translate them between the processor's byte order and the network byte order. This is common when printing an address in a human-readable form, for example.

Four functions are provided to convert between the processor byte order and the network byte order for TCP/IP applications.

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostint32);
                                    Returns: 32-bit integer in network byte order

uint16_t htons(uint16_t hostint16);
                                    Returns: 16-bit integer in network byte order

uint32_t ntohl(uint32_t netint32);
                                    Returns: 32-bit integer in host byte order

uint16_t ntohs(uint16_t netint16);
                                    Returns: 16-bit integer in host byte order
```

The `h` is for "host" byte order, and the `n` is for "network" byte order. The `l` is for "long" (i.e., 4-byte) integer, and the `s` is for "short" (i.e., 2-byte) integer. Although we include `<arpa/inet.h>` to use these functions, system implementations often declare these functions in other headers that are included by `<arpa/inet.h>`. It is also common for systems to implement these functions as macros.

16.3.2 Address Formats

An address identifies a socket endpoint in a particular communication domain. The address format is specific to the particular domain. So that addresses with different formats can be passed to the socket functions, the addresses are cast to a generic `sockaddr` address structure:

```
struct sockaddr {
  sa_family_t   sa_family;  /* address family */
  char          sa_data[];  /* variable-length address */
  ⋮
};
```

Implementations are free to add more members and define a size for the `sa_data` member. For example, on Linux, the structure is defined as

```
struct sockaddr {
  sa_family_t  sa_family;    /* address family */
  char         sa_data[14];  /* variable-length address */
};
```

But on FreeBSD, the structure is defined as

```
struct sockaddr {
  unsigned char  sa_len;       /* total length */
  sa_family_t    sa_family;    /* address family */
  char           sa_data[14];  /* variable-length address */
};
```

Internet addresses are defined in `<netinet/in.h>`. In the IPv4 Internet domain (`AF_INET`), a socket address is represented by a `sockaddr_in` structure:

```
struct in_addr {
  in_addr_t       s_addr;        /* IPv4 address */
};

struct sockaddr_in {
  sa_family_t    sin_family;    /* address family */
  in_port_t      sin_port;      /* port number */
  struct in_addr sin_addr;      /* IPv4 address */
};
```

The `in_port_t` data type is defined to be a `uint16_t`. The `in_addr_t` data type is defined to be a `uint32_t`. These integer data types specify the number of bits in the data type and are defined in `<stdint.h>`.

In contrast to the `AF_INET` domain, the IPv6 Internet domain (`AF_INET6`) socket address is represented by a `sockaddr_in6` structure:

```
struct in6_addr {
  uint8_t        s6_addr[16];    /* IPv6 address */
};
```

```
struct sockaddr_in6 {
  sa_family_t     sin6_family;    /* address family */
  in_port_t       sin6_port;      /* port number */
  uint32_t        sin6_flowinfo;  /* traffic class and flow info */
  struct in6_addr sin6_addr;      /* IPv6 address */
  uint32_t        sin6_scope_id;  /* set of interfaces for scope */
};
```

These are the definitions required by the Single UNIX Specification. Individual implementations are free to add more fields. For example, on Linux, the `sockaddr_in` structure is defined as

```
struct sockaddr_in {
  sa_family_t     sin_family;    /* address family */
  in_port_t       sin_port;      /* port number */
  struct in_addr  sin_addr;      /* IPv4 address */
  unsigned char   sin_zero[8];   /* filler */
};
```

where the `sin_zero` member is a filler field that should be set to all-zero values.

Note that although the `sockaddr_in` and `sockaddr_in6` structures are quite different, they are both passed to the socket routines cast to a `sockaddr` structure. In Section 17.2, we will see that the structure of a UNIX domain socket address is different from both of the Internet domain socket address formats.

It is sometimes necessary to print an address in a format that is understandable by a person instead of a computer. The BSD networking software included the `inet_addr` and `inet_ntoa` functions to convert between the binary address format and a string in dotted-decimal notation (a.b.c.d). These functions, however, work only with IPv4 addresses. Two new functions—`inet_ntop` and `inet_pton`—support similar functionality and work with both IPv4 and IPv6 addresses.

```
#include <arpa/inet.h>

const char *inet_ntop(int domain, const void *restrict addr,
                      char *restrict str, socklen_t size);
```

Returns: pointer to address string on success, `NULL` on error

```
int inet_pton(int domain, const char *restrict str,
              void *restrict addr);
```

Returns: 1 on success, 0 if the format is invalid, or –1 on error

The `inet_ntop` function converts a binary address in network byte order into a text string; `inet_pton` converts a text string into a binary address in network byte order. Only two *domain* values are supported: `AF_INET` and `AF_INET6`.

For `inet_ntop`, the *size* parameter specifies the size of the buffer (*str*) to hold the text string. Two constants are defined to make our job easier: `INET_ADDRSTRLEN` is large enough to hold a text string representing an IPv4 address, and `INET6_ADDRSTRLEN` is large enough to hold a text string representing an IPv6 address. For `inet_pton`, the *addr* buffer needs to be large enough to hold a 32-bit address if *domain* is `AF_INET` or large enough to hold a 128-bit address if *domain* is `AF_INET6`.

16.3.3 Address Lookup

Ideally, an application won't have to be aware of the internal structure of a socket address. If an application simply passes socket addresses around as `sockaddr` structures and doesn't rely on any protocol-specific features, then the application will work with many different protocols that provide the same type of service.

Historically, the BSD networking software has provided interfaces to access the various network configuration information. In Section 6.7, we briefly discussed the networking data files and the functions used to access them. In this section, we discuss them in a little more detail and introduce the newer functions used to look up addressing information.

The network configuration information returned by these functions can be kept in a number of places. This information can be kept in static files (e.g., `/etc/hosts`, `/etc/services`), or it can be managed by a name service, such as DNS (Domain Name System) or NIS (Network Information Service). Regardless of where the information is kept, the same functions can be used to access it.

The hosts known by a given computer system are found by calling `gethostent`.

```
#include <netdb.h>

struct hostent *gethostent(void);
                                        Returns: pointer if OK, NULL on error

void sethostent(int stayopen);

void endhostent(void);
```

If the host database file isn't already open, `gethostent` will open it. The `gethostent` function returns the next entry in the file. The `sethostent` function will open the file or rewind it if it is already open. When the *stayopen* argument is set to a nonzero value, the file remains open after calling `gethostent`. The `endhostent` function can be used to close the file.

When `gethostent` returns, we get a pointer to a `hostent` structure, which might point to a static data buffer that is overwritten each time we call `gethostent`. The `hostent` structure is defined to have at least the following members:

```
struct hostent {
  char   *h_name;       /* name of host */
  char  **h_aliases;    /* pointer to alternate host name array */
  int     h_addrtype;   /* address type */
  int     h_length;     /* length in bytes of address */
  char  **h_addr_list;  /* pointer to array of network addresses */
  .
  .
  .
};
```

The addresses returned are in network byte order.

Two additional functions—`gethostbyname` and `gethostbyaddr`—originally were included with the `hostent` functions, but are now considered to be obsolete. They were removed from Version 4 of the Single UNIX Specification. We'll see replacements for them shortly.

We can get network names and numbers with a similar set of interfaces.

```
#include <netdb.h>

struct netent *getnetbyaddr(uint32_t net, int type);

struct netent *getnetbyname(const char *name);

struct netent *getnetent(void);
```

All return: pointer if OK, NULL on error

```
void setnetent(int stayopen);

void endnetent(void);
```

The `netent` structure contains at least the following fields:

```
struct netent {
  char     *n_name;      /* network name */
  char    **n_aliases;   /* alternate network name array pointer */
  int       n_addrtype;  /* address type */
  uint32_t  n_net;       /* network number */
  :
};
```

The network number is returned in network byte order. The address type is one of the address family constants (`AF_INET`, for example).

We can map between protocol names and numbers with the following functions.

```
#include <netdb.h>

struct protoent *getprotobyname(const char *name);

struct protoent *getprotobynumber(int proto);

struct protoent *getprotoent(void);
```

All return: pointer if OK, NULL on error

```
void setprotoent(int stayopen);

void endprotoent(void);
```

The `protoent` structure as defined by POSIX.1 has at least the following members:

```
struct protoent {
  char   *p_name;     /* protocol name */
  char  **p_aliases;  /* pointer to alternate protocol name array */
  int     p_proto;    /* protocol number */
  :
};
```

Services are represented by the port number portion of the address. Each service is offered on a unique, well-known port number. We can map a service name to a port

number with `getservbyname`, map a port number to a service name with `getservbyport`, or scan the services database sequentially with `getservent`.

```
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);

struct servent *getservbyport(int port, const char *proto);

struct servent *getservent(void);
```

All return: pointer if OK, `NULL` on error

```
void setservent(int stayopen);

void endservent(void);
```

The `servent` structure is defined to have at least the following members:

```
struct servent {
  char   *s_name;     /* service name */
  char  **s_aliases;  /* pointer to alternate service name array */
  int     s_port;     /* port number */
  char   *s_proto;    /* name of protocol */
  :
};
```

POSIX.1 defines several new functions to allow an application to map from a host name and a service name to an address, and vice versa. These functions replace the older `gethostbyname` and `gethostbyaddr` functions.

The `getaddrinfo` function allows us to map a host name and a service name to an address.

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *restrict host,
                const char *restrict service,
                const struct addrinfo *restrict hint,
                struct addrinfo **restrict res);
```

Returns: 0 if OK, nonzero error code on error

```
void freeaddrinfo(struct addrinfo *ai);
```

We need to provide the host name, the service name, or both. If we provide only one name, the other should be a null pointer. The host name can be either a node name or the host address in dotted-decimal notation.

The `getaddrinfo` function returns a linked list of `addrinfo` structures. We can use `freeaddrinfo` to free one or more of these structures, depending on how many structures are linked together using the `ai_next` field in the structures.

The addrinfo structure is defined to include at least the following members:

```
struct addrinfo {
  int               ai_flags;      /* customize behavior */
  int               ai_family;     /* address family */
  int               ai_socktype;   /* socket type */
  int               ai_protocol;   /* protocol */
  socklen_t         ai_addrlen;    /* length in bytes of address */
  struct sockaddr  *ai_addr;       /* address */
  char             *ai_canonname;  /* canonical name of host */
  struct addrinfo  *ai_next;       /* next in list */
  .
  .
  .
};
```

We can supply an optional *hint* to select addresses that meet certain criteria. The hint is a template used for filtering addresses and uses only the ai_family, ai_flags, ai_protocol, and ai_socktype fields. The remaining integer fields must be set to 0, and the pointer fields must be null. Figure 16.7 summarizes the flags we can use in the ai_flags field to customize how addresses and names are treated.

Flag	Description
AI_ADDRCONFIG	Query for whichever address type (IPv4 or IPv6) is configured.
AI_ALL	Look for both IPv4 and IPv6 addresses (used only with AI_V4MAPPED).
AI_CANONNAME	Request a canonical name (as opposed to an alias).
AI_NUMERICHOST	The host address is specified in numeric format; don't try to translate it.
AI_NUMERICSERV	The service is specified as a numeric port number; don't try to translate it.
AI_PASSIVE	Socket address is intended to be bound for listening.
AI_V4MAPPED	If no IPv6 addresses are found, return IPv4 addresses mapped in IPv6 format.

Figure 16.7 Flags for addrinfo structure

If getaddrinfo fails, we can't use perror or strerror to generate an error message. Instead, we need to call gai_strerror to convert the error code returned into an error message.

```
#include <netdb.h>

const char *gai_strerror(int error);
```

Returns: a pointer to a string describing the error

The getnameinfo function converts an address into host and service names.

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *restrict addr, socklen_t alen,
                char *restrict host, socklen_t hostlen,
                char *restrict service, socklen_t servlen, int flags);
```

Returns: 0 if OK, nonzero on error

The socket address (*addr*) is translated into a host name and a service name. If *host* is non-null, it points to a buffer *hostlen* bytes long that will be used to return the host name. Similarly, if *service* is non-null, it points to a buffer *servlen* bytes long that will be used to return the service name.

The *flags* argument gives us some control over how the translation is done. Figure 16.8 summarizes the supported flags.

Flag	Description
NI_DGRAM	The service is datagram based instead of stream based.
NI_NAMEREQD	If the host name can't be found, treat this as an error.
NI_NOFQDN	Return only the node name portion of the fully qualified domain name for local hosts.
NI_NUMERICHOST	Return the numeric form of the host address instead of the name.
NI_NUMERICSCOPE	For IPv6, return the numeric form of the scope ID instead of the name.
NI_NUMERICSERV	Return the numeric form of the service address (i.e., the port number) instead of the name.

Figure 16.8 Flags for the `getnameinfo` function

Example

Figure 16.9 illustrates the use of the `getaddrinfo` function.

```
#include "apue.h"
#if defined(SOLARIS)
#include <netinet/in.h>
#endif
#include <netdb.h>
#include <arpa/inet.h>
#if defined(BSD)
#include <sys/socket.h>
#include <netinet/in.h>
#endif

void
print_family(struct addrinfo *aip)
{
    printf(" family ");
    switch (aip->ai_family) {
    case AF_INET:
        printf("inet");
        break;
    case AF_INET6:
        printf("inet6");
        break;
    case AF_UNIX:
        printf("unix");
        break;
    case AF_UNSPEC:
```

```
            printf("unspecified");
            break;
        default:
            printf("unknown");
        }
    }

    void
    print_type(struct addrinfo *aip)
    {
        printf(" type ");
        switch (aip->ai_socktype) {
        case SOCK_STREAM:
            printf("stream");
            break;
        case SOCK_DGRAM:
            printf("datagram");
            break;
        case SOCK_SEQPACKET:
            printf("seqpacket");
            break;
        case SOCK_RAW:
            printf("raw");
            break;
        default:
            printf("unknown (%d)", aip->ai_socktype);
        }
    }

    void
    print_protocol(struct addrinfo *aip)
    {
        printf(" protocol ");
        switch (aip->ai_protocol) {
        case 0:
            printf("default");
            break;
        case IPPROTO_TCP:
            printf("TCP");
            break;
        case IPPROTO_UDP:
            printf("UDP");
            break;
        case IPPROTO_RAW:
            printf("raw");
            break;
        default:
            printf("unknown (%d)", aip->ai_protocol);
        }
    }
```

```
void
print_flags(struct addrinfo *aip)
{
    printf("flags");
    if (aip->ai_flags == 0) {
        printf(" 0");
    } else {
        if (aip->ai_flags & AI_PASSIVE)
            printf(" passive");
        if (aip->ai_flags & AI_CANONNAME)
            printf(" canon");
        if (aip->ai_flags & AI_NUMERICHOST)
            printf(" numhost");
        if (aip->ai_flags & AI_NUMERICSERV)
            printf(" numserv");
        if (aip->ai_flags & AI_V4MAPPED)
            printf(" v4mapped");
        if (aip->ai_flags & AI_ALL)
            printf(" all");
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo     *ailist, *aip;
    struct addrinfo     hint;
    struct sockaddr_in  *sinp;
    const char          *addr;
    int                 err;
    char                abuf[INET_ADDRSTRLEN];

    if (argc != 3)
        err_quit("usage: %s nodename service", argv[0]);
    hint.ai_flags = AI_CANONNAME;
    hint.ai_family = 0;
    hint.ai_socktype = 0;
    hint.ai_protocol = 0;
    hint.ai_addrlen = 0;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(argv[1], argv[2], &hint, &ailist)) != 0)
        err_quit("getaddrinfo error: %s", gai_strerror(err));
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        print_flags(aip);
        print_family(aip);
        print_type(aip);
        print_protocol(aip);
        printf("\n\thost %s", aip->ai_canonname?aip->ai_canonname:"-");
        if (aip->ai_family == AF_INET) {
```

```
            sinp = (struct sockaddr_in *)aip->ai_addr;
            addr = inet_ntop(AF_INET, &sinp->sin_addr, abuf,
                INET_ADDRSTRLEN);
            printf(" address %s", addr?addr:"unknown");
            printf(" port %d", ntohs(sinp->sin_port));
        }
        printf("\n");
    }
    exit(0);
}
```

Figure 16.9 Print host and service information

This program illustrates the use of the `getaddrinfo` function. If multiple protocols provide the given service for the given host, the program will print more than one entry. In this example, we print out the address information only for the protocols that work with IPv4 (`ai_family` equals `AF_INET`). If we wanted to restrict the output to the `AF_INET` protocol family, we could set the `ai_family` field in the hint.

When we run the program on one of the test systems, we get

```
$ ./a.out harry nfs
flags canon family inet type stream protocol TCP
     host harry address 192.168.1.99 port 2049
flags canon family inet type datagram protocol UDP
     host harry address 192.168.1.99 port 2049
```

□

16.3.4 Associating Addresses with Sockets

The address associated with a client's socket is of little interest, and we can let the system choose a default address for us. For a server, however, we need to associate a well-known address with the server's socket on which client requests will arrive. Clients need a way to discover the address to use to contact a server, and the simplest scheme is for a server to reserve an address and register it in `/etc/services` or with a name service.

We use the `bind` function to associate an address with a socket.

```
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr, socklen_t len);
```

Returns: 0 if OK, –1 on error

There are several restrictions on the address we can use:

- The address we specify must be valid for the machine on which the process is running; we can't specify an address belonging to some other machine.
- The address must match the format supported by the address family we used to create the socket.

- The port number in the address cannot be less than 1,024 unless the process has the appropriate privilege (i.e., is the superuser).
- Usually, only one socket endpoint can be bound to a given address, although some protocols allow duplicate bindings.

For the Internet domain, if we specify the special IP address `INADDR_ANY` (defined in `<netinet/in.h>`), the socket endpoint will be bound to all the system's network interfaces. This means that we can receive packets from any of the network interface cards installed in the system. We'll see in the next section that the system will choose an address and bind it to our socket for us if we call `connect` or `listen` without first binding an address to the socket.

We can use the `getsockname` function to discover the address bound to a socket.

```
#include <sys/socket.h>

int getsockname(int sockfd, struct sockaddr *restrict addr,
                socklen_t *restrict alenp);
```

Returns: 0 if OK, –1 on error

Before calling `getsockname`, we set *alenp* to point to an integer containing the size of the `sockaddr` buffer. On return, the integer is set to the size of the address returned. If the address won't fit in the buffer provided, the address is silently truncated. If no address is currently bound to the socket, the results are undefined.

If the socket is connected to a peer, we can find out the peer's address by calling the `getpeername` function.

```
#include <sys/socket.h>

int getpeername(int sockfd, struct sockaddr *restrict addr,
                socklen_t *restrict alenp);
```

Returns: 0 if OK, –1 on error

Other than returning the peer's address, the `getpeername` function is identical to the `getsockname` function.

16.4 Connection Establishment

If we're dealing with a connection-oriented network service (`SOCK_STREAM` or `SOCK_SEQPACKET`), then before we can exchange data, we need to create a connection between the socket of the process requesting the service (the client) and the process providing the service (the server). We use the `connect` function to create a connection.

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t len);
```

Returns: 0 if OK, –1 on error

The address we specify with `connect` is the address of the server with which we wish to communicate. If *sockfd* is not bound to an address, `connect` will bind a default address for the caller.

When we try to connect to a server, the connect request might fail for several reasons. For a connect request to succeed, the machine to which we are trying to connect must be up and running, the server must be bound to the address we are trying to contact, and there must be room in the server's pending connect queue (we'll learn more about this shortly). Thus, applications must be able to handle `connect` error returns that might be caused by transient conditions.

Example

Figure 16.10 shows one way to handle transient `connect` errors. These errors are likely with a server that is running on a heavily loaded system.

```
#include "apue.h"
#include <sys/socket.h>

#define MAXSLEEP 128

int
connect_retry(int sockfd, const struct sockaddr *addr, socklen_t alen)
{
    int numsec;

    /*
     * Try to connect with exponential backoff.
     */
    for (numsec = 1; numsec <= MAXSLEEP; numsec <<= 1) {
        if (connect(sockfd, addr, alen) == 0) {
            /*
             * Connection accepted.
             */
            return(0);
        }

        /*
         * Delay before trying again.
         */
        if (numsec <= MAXSLEEP/2)
            sleep(numsec);
    }
    return(-1);
}
```

Figure 16.10 Connect with retry

This function shows what is known as an *exponential backoff* algorithm. If the call to `connect` fails, the process goes to sleep for a short time and then tries again, increasing the delay each time through the loop, up to a maximum delay of about 2 minutes.

There is a problem with the code shown in Figure 16.10: it isn't portable. This technique works on Linux and Solaris, but doesn't work as expected on FreeBSD and Mac OS X. If the first connection attempt fails, BSD-based socket implementations continue to fail successive connection attempts when the same socket descriptor is used with TCP. This is a case of a protocol-specific behavior leaking through the (protocol-independent) socket interface and becoming visible to applications. The reason for this is historical, and thus the Single UNIX Specification warns that the state of a socket is undefined if `connect` fails.

Because of this, portable applications need to close the socket if `connect` fails. If we want to retry, we have to open a new socket. This more portable technique is shown in Figure 16.11.

```
#include "apue.h"
#include <sys/socket.h>

#define MAXSLEEP 128

int
connect_retry(int domain, int type, int protocol,
              const struct sockaddr *addr, socklen_t alen)
{
    int numsec, fd;

    /*
     * Try to connect with exponential backoff.
     */
    for (numsec = 1; numsec <= MAXSLEEP; numsec <<= 1) {
        if ((fd = socket(domain, type, protocol)) < 0)
            return(-1);
        if (connect(fd, addr, alen) == 0) {
            /*
             * Connection accepted.
             */
            return(fd);
        }
        close(fd);

        /*
         * Delay before trying again.
         */
        if (numsec <= MAXSLEEP/2)
            sleep(numsec);
    }
    return(-1);
}
```

Figure 16.11 Portable connect with retry

Note that because we might have to establish a new socket, it makes no sense to pass a socket descriptor to the `connect_retry` function. Instead of returning an indication of success, we now return a connected socket descriptor to the caller. □

If the socket descriptor is in nonblocking mode, which we discuss further in Section 16.8, `connect` will return –1 with `errno` set to the special error code `EINPROGRESS` if the connection can't be established immediately. The application can use either `poll` or `select` to determine when the file descriptor is writable. At this point, the connection is complete.

The `connect` function can also be used with a connectionless network service (`SOCK_DGRAM`). This might seem like a contradiction, but it is an optimization instead. If we call `connect` with a `SOCK_DGRAM` socket, the destination address of all messages we send is set to the address we specified in the `connect` call, relieving us from having to provide the address every time we transmit a message. In addition, we will receive datagrams only from the address we've specified.

A server announces that it is willing to accept connect requests by calling the `listen` function.

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

Returns: 0 if OK, –1 on error

The *backlog* argument provides a hint to the system regarding the number of outstanding connect requests that it should enqueue on behalf of the process. The actual value is determined by the system, but the upper limit is specified as `SOMAXCONN` in `<sys/socket.h>`.

> On Solaris, the `SOMAXCONN` value in `<sys/socket.h>` is ignored. The particular maximum depends on the implementation of each protocol. For TCP, the default is 128.

Once the queue is full, the system will reject additional connect requests, so the *backlog* value must be chosen based on the expected load of the server and the amount of processing it must do to accept a connect request and start the service.

Once a server has called `listen`, the socket used can receive connect requests. We use the `accept` function to retrieve a connect request and convert it into a connection.

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *restrict addr,
           socklen_t *restrict len);
```

Returns: file (socket) descriptor if OK, –1 on error

The file descriptor returned by `accept` is a socket descriptor that is connected to the client that called `connect`. This new socket descriptor has the same socket type and address family as the original socket (*sockfd*). The original socket passed to `accept` is not associated with the connection, but instead remains available to receive additional connect requests.

If we don't care about the client's identity, we can set the *addr* and *len* parameters to `NULL`. Otherwise, before calling `accept`, we need to set the *addr* parameter to a buffer large enough to hold the address and set the integer pointed to by *len* to the size of the buffer in bytes. On return, `accept` will fill in the client's address in the buffer and update the integer pointed to by *len* to reflect the size of the address.

If no connect requests are pending, `accept` will block until one arrives. If *sockfd* is in nonblocking mode, `accept` will return –1 and set `errno` to either `EAGAIN` or `EWOULDBLOCK`.

All four platforms discussed in this text define `EAGAIN` to be the same as `EWOULDBLOCK`.

If a server calls `accept` and no connect request is present, the server will block until one arrives. Alternatively, a server can use either `poll` or `select` to wait for a connect request to arrive. In this case, a socket with pending connect requests will appear to be readable.

Example

Figure 16.12 shows a function we can use to allocate and initialize a socket for use by a server process.

```
#include "apue.h"
#include <errno.h>
#include <sys/socket.h>

int
initserver(int type, const struct sockaddr *addr, socklen_t alen,
  int qlen)
{
    int fd;
    int err = 0;

    if ((fd = socket(addr->sa_family, type, 0)) < 0)
        return(-1);
    if (bind(fd, addr, alen) < 0)
        goto errout;
    if (type == SOCK_STREAM || type == SOCK_SEQPACKET) {
        if (listen(fd, qlen) < 0)
            goto errout;
    }
    return(fd);

errout:
    err = errno;
    close(fd);
    errno = err;
    return(-1);
}
```

Figure 16.12 Initialize a socket endpoint for use by a server

We'll see that TCP has some strange rules regarding address reuse that make this example inadequate. Figure 16.22 shows a version of this function that bypasses these rules, solving the major drawback with this version. □

16.5 Data Transfer

Since a socket endpoint is represented as a file descriptor, we can use `read` and `write` to communicate with a socket, as long as it is connected. Recall that a datagram socket can be "connected" if we set the default peer address using the `connect` function. Using `read` and `write` with socket descriptors is significant, because it means that we can pass socket descriptors to functions that were originally designed to work with local files. We can also arrange to pass the socket descriptors to child processes that execute programs that know nothing about sockets.

Although we can exchange data using `read` and `write`, that is about all we can do with these two functions. If we want to specify options, receive packets from multiple clients, or send out-of-band data, we need to use one of the six socket functions designed for data transfer.

Three functions are available for sending data, and three are available for receiving data. First, we'll look at the ones used to send data.

The simplest one is `send`. It is similar to `write`, but allows us to specify flags to change how the data we want to transmit is treated.

```
#include <sys/socket.h>

ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags);
```

Returns: number of bytes sent if OK, –1 on error

Like `write`, the socket has to be connected to use `send`. The *buf* and *nbytes* arguments have the same meaning as they do with `write`.

Unlike `write`, however, `send` supports a fourth *flags* argument. Three flags are defined by the Single UNIX Specification, but it is common for implementations to support additional ones. They are summarized in Figure 16.13.

If `send` returns success, it doesn't necessarily mean that the process at the other end of the connection receives the data. All we are guaranteed is that when `send` succeeds, the data has been delivered to the network drivers without error.

With a protocol that supports message boundaries, if we try to send a single message larger than the maximum supported by the protocol, `send` will fail with `errno` set to `EMSGSIZE`. With a byte-stream protocol, `send` will block until the entire amount of data has been transmitted.

The `sendto` function is similar to `send`. The difference is that `sendto` allows us to specify a destination address to be used with connectionless sockets.

```
#include <sys/socket.h>

ssize_t sendto(int sockfd, const void *buf, size_t nbytes, int flags,
               const struct sockaddr *destaddr, socklen_t destlen);
```

Returns: number of bytes sent if OK, –1 on error

With a connection-oriented socket, the destination address is ignored, as the destination is implied by the connection. With a connectionless socket, we can't use `send` unless

the destination address is first set by calling `connect`, so `sendto` gives us an alternate way to send a message.

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
`MSG_CONFIRM`	Provide feedback to the link layer to keep address mapping valid.			•		
`MSG_DONTROUTE`	Don't route packet outside of local network.		•	•	•	•
`MSG_DONTWAIT`	Enable nonblocking operation (equivalent to using `O_NONBLOCK`).		•	•	•	•
`MSG_EOF`	Shut the sender side of the socket down after sending data.		•		•	
`MSG_EOR`	Mark the end of the record if supported by protocol.	•	•	•	•	•
`MSG_MORE`	Delay sending the packet to allow more data to be written.			•		
`MSG_NOSIGNAL`	Don't generate `SIGPIPE` when writing to an unconnected socket.	•	•	•		
`MSG_OOB`	Send out-of-band data if supported by protocol (see Section 16.7).	•	•	•	•	•

Figure 16.13 Flags used with `send` socket calls

We have one more choice when transmitting data over a socket. We can call `sendmsg` with a `msghdr` structure to specify multiple buffers from which to transmit data, similar to the `writev` function (Section 14.6).

```
#include <sys/socket.h>

ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

Returns: number of bytes sent if OK, –1 on error

POSIX.1 defines the `msghdr` structure to have at least the following members:

```
struct msghdr {
  void          *msg_name;        /* optional address */
  socklen_t      msg_namelen;     /* address size in bytes */
  struct iovec  *msg_iov;         /* array of I/O buffers */
  int            msg_iovlen;      /* number of elements in array */
  void          *msg_control;     /* ancillary data */
  socklen_t      msg_controllen;  /* number of ancillary bytes */
  int            msg_flags;       /* flags for received message */
  .
  .
  .
};
```

We saw the `iovec` structure in Section 14.6. We'll see the use of ancillary data in Section 17.4.

The recv function is similar to read, but allows us to specify some options to control how we receive the data.

```
#include <sys/socket.h>

ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);
```

Returns: length of message in bytes,
0 if no messages are available and peer has done an orderly shutdown,
or –1 on error

The flags that can be passed to recv are summarized in Figure 16.14. Only three are defined by the Single UNIX Specification.

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
MSG_CMSG_CLOEXEC	Set the close-on-exec flag for file descriptors received over a UNIX domain socket (see Section 17.4).			•		
MSG_DONTWAIT	Enable nonblocking operation (equivalent to using O_NONBLOCK).		•	•		•
MSG_ERRQUEUE	Receive error information as ancillary data.			•		
MSG_OOB	Retrieve out-of-band data if supported by protocol (see Section 16.7).	•	•	•	•	•
MSG_PEEK	Return packet contents without consuming the packet.	•	•	•	•	•
MSG_TRUNC	Request that the real length of the packet be returned, even if it was truncated.			•		
MSG_WAITALL	Wait until all data is available (SOCK_STREAM only).	•	•	•	•	•

Figure 16.14 Flags used with recv socket calls

When we specify the MSG_PEEK flag, we can peek at the next data to be read without actually consuming it. The next call to read or one of the recv functions will return the same data we peeked at.

With SOCK_STREAM sockets, we can receive less data than we requested. The MSG_WAITALL flag inhibits this behavior, preventing recv from returning until all the data we requested has been received. With SOCK_DGRAM and SOCK_SEQPACKET sockets, the MSG_WAITALL flag provides no change in behavior, because these message-based socket types already return an entire message in a single read.

If the sender has called shutdown (Section 16.2) to end transmission, or if the network protocol supports orderly shutdown by default and the sender has closed the socket, then recv will return 0 when we have received all the data.

If we are interested in the identity of the sender, we can use `recvfrom` to obtain the source address from which the data was sent.

```
#include <sys/socket.h>

ssize_t recvfrom(int sockfd, void *restrict buf, size_t len, int flags,
                 struct sockaddr *restrict addr,
                 socklen_t *restrict addrlen);
```

Returns: length of message in bytes, 0 if no messages are available and peer has done an orderly shutdown, or –1 on error

If *addr* is non-null, it will contain the address of the socket endpoint from which the data was sent. When calling `recvfrom`, we need to set the *addrlen* parameter to point to an integer containing the size in bytes of the socket buffer to which *addr* points. On return, the integer is set to the actual size of the address in bytes.

Because it allows us to retrieve the address of the sender, `recvfrom` is typically used with connectionless sockets. Otherwise, `recvfrom` behaves identically to `recv`.

To receive data into multiple buffers, similar to `readv` (Section 14.6), or if we want to receive ancillary data (Section 17.4), we can use `recvmsg`.

```
#include <sys/socket.h>

ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

Returns: length of message in bytes, 0 if no messages are available and peer has done an orderly shutdown, or –1 on error

The `msghdr` structure (which we saw used with `sendmsg`) is used by `recvmsg` to specify the input buffers to be used to receive the data. We can set the *flags* argument to change the default behavior of `recvmsg`. On return, the `msg_flags` field of the `msghdr` structure is set to indicate various characteristics of the data received. (The `msg_flags` field is ignored on entry to `recvmsg`.) The possible values on return from `recvmsg` are summarized in Figure 16.15. We'll see an example that uses `recvmsg` in Chapter 17.

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
`MSG_CTRUNC`	Control data was truncated.	•	•	•	•	•
`MSG_EOR`	End of record was received.	•	•	•	•	•
`MSG_ERRQUEUE`	Error information was received as ancillary data.			•		
`MSG_OOB`	Out-of-band data was received.	•	•	•	•	•
`MSG_TRUNC`	Normal data was truncated.	•	•	•	•	•

Figure 16.15 Flags returned in `msg_flags` by `recvmsg`

Example — Connection-Oriented Client

Figure 16.16 shows a client command that communicates with a server to obtain the output from a system's `uptime` command. We call this service "remote uptime" (or "ruptime" for short).

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <sys/socket.h>

#define BUFLEN      128

extern int connect_retry(int, int, int, const struct sockaddr *,
    socklen_t);

void
print_uptime(int sockfd)
{
    int     n;
    char    buf[BUFLEN];

    while ((n = recv(sockfd, buf, BUFLEN, 0)) > 0)
        write(STDOUT_FILENO, buf, n);
    if (n < 0)
        err_sys("recv error");
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int             sockfd, err;

    if (argc != 2)
        err_quit("usage: ruptime hostname");
    memset(&hint, 0, sizeof(hint));
    hint.ai_socktype = SOCK_STREAM;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(argv[1], "ruptime", &hint, &ailist)) != 0)
        err_quit("getaddrinfo error: %s", gai_strerror(err));
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = connect_retry(aip->ai_family, SOCK_STREAM, 0,
          aip->ai_addr, aip->ai_addrlen)) < 0) {
            err = errno;
        } else {
            print_uptime(sockfd);
            exit(0);
        }
    }
```

```
    err_exit(err, "can't connect to %s", argv[1]);
}
```

Figure 16.16 Client command to get uptime from server

This program connects to a server, reads the string sent by the server, and prints the string on the standard output. Since we're using a `SOCK_STREAM` socket, we can't be guaranteed that we will read the entire string in one call to `recv`, so we need to repeat the call until it returns 0.

The `getaddrinfo` function might return more than one candidate address for us to use if the server supports multiple network interfaces or multiple network protocols. We try each one in turn, giving up when we find one that allows us to connect to the service. We use the `connect_retry` function from Figure 16.11 to establish a connection with the server. □

Example — Connection-Oriented Server

Figure 16.17 shows the server that provides the `uptime` command's output to the client program from Figure 16.16.

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <syslog.h>
#include <sys/socket.h>

#define BUFLEN  128
#define QLEN 10

#ifndef HOST_NAME_MAX
#define HOST_NAME_MAX 256
#endif

extern int initserver(int, const struct sockaddr *, socklen_t, int);

void
serve(int sockfd)
{
    int     clfd;
    FILE    *fp;
    char    buf[BUFLEN];

    set_cloexec(sockfd);
    for (;;) {
        if ((clfd = accept(sockfd, NULL, NULL)) < 0) {
            syslog(LOG_ERR, "ruptimed: accept error: %s",
              strerror(errno));
            exit(1);
        }
        set_cloexec(clfd);
        if ((fp = popen("/usr/bin/uptime", "r")) == NULL) {
```

```
            sprintf(buf, "error: %s\n", strerror(errno));
            send(clfd, buf, strlen(buf), 0);
        } else {
            while (fgets(buf, BUFLEN, fp) != NULL)
                send(clfd, buf, strlen(buf), 0);
            pclose(fp);
        }
        close(clfd);
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int             sockfd, err, n;
    char            *host;

    if (argc != 1)
        err_quit("usage: ruptimed");
    if ((n = sysconf(_SC_HOST_NAME_MAX)) < 0)
        n = HOST_NAME_MAX;  /* best guess */
    if ((host = malloc(n)) == NULL)
        err_sys("malloc error");
    if (gethostname(host, n) < 0)
        err_sys("gethostname error");
    daemonize("ruptimed");
    memset(&hint, 0, sizeof(hint));
    hint.ai_flags = AI_CANONNAME;
    hint.ai_socktype = SOCK_STREAM;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
        syslog(LOG_ERR, "ruptimed: getaddrinfo error: %s",
          gai_strerror(err));
        exit(1);
    }
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
          aip->ai_addrlen, QLEN)) >= 0) {
            serve(sockfd);
            exit(0);
        }
    }
    exit(1);
}
```

Figure 16.17 Server program to provide system uptime

To find its address, the server needs to get the name of the host on which it is running. If the maximum host name length is indeterminate, we use `HOST_NAME_MAX` instead. If the system doesn't define `HOST_NAME_MAX`, we define it ourselves. POSIX.1 requires the maximum host name length to be at least 255 bytes, not including the terminating null, so we define `HOST_NAME_MAX` to be 256 to include the terminating null.

The server gets the host name by calling `gethostname` and looks up the address for the remote uptime service. Multiple addresses can be returned, but we simply choose the first one for which we can establish a passive socket endpoint (i.e., one used only to listen for connect requests). Handling multiple addresses is left as an exercise.

We use the `initserver` function from Figure 16.12 to initialize the socket endpoint on which we will wait for connect requests to arrive. (Actually, we use the version from Figure 16.22; we'll see why when we discuss socket options in Section 16.6.) □

Example — Alternative Connection-Oriented Server

Previously, we stated that using file descriptors to access sockets was significant, because it allowed programs that knew nothing about networking to be used in a networked environment. The version of the server shown in Figure 16.18 illustrates this point. Instead of reading the output of the `uptime` command and sending it to the client, the server arranges to have the standard output and standard error of the `uptime` command be the socket endpoint connected to the client.

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <syslog.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/wait.h>

#define QLEN 10

#ifndef HOST_NAME_MAX
#define HOST_NAME_MAX 256
#endif

extern int initserver(int, const struct sockaddr *, socklen_t, int);

void
serve(int sockfd)
{
    int     clfd, status;
    pid_t   pid;

    set_cloexec(sockfd);
    for (;;) {
        if ((clfd = accept(sockfd, NULL, NULL)) < 0) {
            syslog(LOG_ERR, "ruptimed: accept error: %s",
              strerror(errno));
```

```
            exit(1);
        }
        if ((pid = fork()) < 0) {
            syslog(LOG_ERR, "ruptimed: fork error: %s",
              strerror(errno));
            exit(1);
        } else if (pid == 0) {  /* child */
            /*
             * The parent called daemonize (Figure 13.1), so
             * STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO
             * are already open to /dev/null.  Thus, the call to
             * close doesn't need to be protected by checks that
             * clfd isn't already equal to one of these values.
             */
            if (dup2(clfd, STDOUT_FILENO) != STDOUT_FILENO ||
              dup2(clfd, STDERR_FILENO) != STDERR_FILENO) {
                syslog(LOG_ERR, "ruptimed: unexpected error");
                exit(1);
            }
            close(clfd);
            execl("/usr/bin/uptime", "uptime", (char *)0);
            syslog(LOG_ERR, "ruptimed: unexpected return from exec: %s",
              strerror(errno));
        } else {        /* parent */
            close(clfd);
            waitpid(pid, &status, 0);
        }
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int             sockfd, err, n;
    char            *host;

    if (argc != 1)
        err_quit("usage: ruptimed");
    if ((n = sysconf(_SC_HOST_NAME_MAX)) < 0)
        n = HOST_NAME_MAX;  /* best guess */
    if ((host = malloc(n)) == NULL)
        err_sys("malloc error");
    if (gethostname(host, n) < 0)
        err_sys("gethostname error");
    daemonize("ruptimed");
    memset(&hint, 0, sizeof(hint));
    hint.ai_flags = AI_CANONNAME;
    hint.ai_socktype = SOCK_STREAM;
    hint.ai_canonname = NULL;
```

```
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
        syslog(LOG_ERR, "ruptimed: getaddrinfo error: %s",
          gai_strerror(err));
        exit(1);
    }
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
          aip->ai_addrlen, QLEN)) >= 0) {
            serve(sockfd);
            exit(0);
        }
    }
    exit(1);
}
```

Figure 16.18 Server program illustrating command writing directly to socket

Instead of using `popen` to run the `uptime` command and reading the output from the pipe connected to the command's standard output, we use `fork` to create a child process and then use `dup2` to arrange that the child's copy of `STDIN_FILENO` is open to `/dev/null` and that both `STDOUT_FILENO` and `STDERR_FILENO` are open to the socket endpoint. When we execute `uptime`, the command writes the results to its standard output, which is connected to the socket, and the data is sent back to the `ruptime` client command.

The parent can safely close the file descriptor connected to the client, because the child still has it open. The parent waits for the child to complete before proceeding, so that the child doesn't become a zombie. Since it shouldn't take too long to run the `uptime` command, the parent can afford to wait for the child to exit before accepting the next connect request. This strategy might not be appropriate if the child takes a long time, however. □

The previous examples have used connection-oriented sockets. But how do we choose the appropriate type? When do we use a connection-oriented socket, and when do we use a connectionless socket? The answer depends on how much work we want to do and how much tolerance we have for errors.

With a connectionless socket, packets can arrive out of order, so if we can't fit all our data in one packet, we will have to worry about ordering in our application. The maximum packet size is a characteristic of the communication protocol. Also, with a connectionless socket, the packets can be lost. If our application can't tolerate this loss, we should use connection-oriented sockets.

Tolerating packet loss means that we have two choices. If we intend to have reliable communication with our peer, we have to number our packets and request retransmission from the peer application when we detect a missing packet. We also have to identify duplicate packets and discard them, since a packet might be delayed and appear to be lost, but show up after we have requested retransmission.

The other choice we have is to deal with the error by letting the user retry the command. For simple applications this might be adequate, but for complex applications it usually isn't a viable alternative. Thus, it is generally better to use connection-oriented sockets in this case.

The drawbacks to connection-oriented sockets are that more work and time are needed to establish a connection, and each connection consumes more resources from the operating system.

Example — Connectionless Client

The program in Figure 16.19 is a version of the `uptime` client command that uses the datagram socket interface.

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <sys/socket.h>

#define BUFLEN      128
#define TIMEOUT     20

void
sigalrm(int signo)
{
}

void
print_uptime(int sockfd, struct addrinfo *aip)
{
    int     n;
    char    buf[BUFLEN];

    buf[0] = 0;
    if (sendto(sockfd, buf, 1, 0, aip->ai_addr, aip->ai_addrlen) < 0)
        err_sys("sendto error");
    alarm(TIMEOUT);
    if ((n = recvfrom(sockfd, buf, BUFLEN, 0, NULL, NULL)) < 0) {
        if (errno != EINTR)
            alarm(0);
        err_sys("recv error");
    }
    alarm(0);
    write(STDOUT_FILENO, buf, n);
}

int
main(int argc, char *argv[])
{
    struct addrinfo     *ailist, *aip;
    struct addrinfo     hint;
```

```
    int                 sockfd, err;
    struct sigaction    sa;

    if (argc != 2)
        err_quit("usage: ruptime hostname");
    sa.sa_handler = sigalrm;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGALRM, &sa, NULL) < 0)
        err_sys("sigaction error");
    memset(&hint, 0, sizeof(hint));
    hint.ai_socktype = SOCK_DGRAM;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(argv[1], "ruptime", &hint, &ailist)) != 0)
        err_quit("getaddrinfo error: %s", gai_strerror(err));

    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = socket(aip->ai_family, SOCK_DGRAM, 0)) < 0) {
            err = errno;
        } else {
            print_uptime(sockfd, aip);
            exit(0);
        }
    }

    fprintf(stderr, "can't contact %s: %s\n", argv[1], strerror(err));
    exit(1);
}
```

Figure 16.19 Client command using datagram service

The `main` function for the datagram-based client is similar to the one for the connection-oriented client, with the addition of installing a signal handler for `SIGALRM`. We use the `alarm` function to avoid blocking indefinitely in the call to `recvfrom`.

With the connection-oriented protocol, we needed to connect to the server before exchanging data. The arrival of the connect request was enough for the server to determine that it needed to provide service to a client. But with the datagram-based protocol, we need a way to notify the server that we want it to perform its service on our behalf. In this example, we simply send the server a 1-byte message. The server will receive it, get our address from the packet, and use this address to transmit its response. If the server offered multiple services, we could use this request message to indicate the service we want, but since the server does only one thing, the content of the 1-byte message doesn't matter.

If the server isn't running, the client will block indefinitely in the call to `recvfrom`. With the connection-oriented example, the `connect` call will fail if the server isn't running. To avoid blocking indefinitely, we set an alarm clock before calling `recvfrom`.

□

Example — Connectionless Server

The program in Figure 16.20 is the datagram version of the `uptime` server.

```
#include "apue.h"
#include <netdb.h>
#include <errno.h>
#include <syslog.h>
#include <sys/socket.h>

#define BUFLEN      128
#define MAXADDRLEN  256

#ifndef HOST_NAME_MAX
#define HOST_NAME_MAX 256
#endif

extern int initserver(int, const struct sockaddr *, socklen_t, int);

void
serve(int sockfd)
{
    int             n;
    socklen_t       alen;
    FILE            *fp;
    char            buf[BUFLEN];
    char            abuf[MAXADDRLEN];
    struct sockaddr *addr = (struct sockaddr *)abuf;

    set_cloexec(sockfd);
    for (;;) {
        alen = MAXADDRLEN;
        if ((n = recvfrom(sockfd, buf, BUFLEN, 0, addr, &alen)) < 0) {
            syslog(LOG_ERR, "ruptimed: recvfrom error: %s",
              strerror(errno));
            exit(1);
        }
        if ((fp = popen("/usr/bin/uptime", "r")) == NULL) {
            sprintf(buf, "error: %s\n", strerror(errno));
            sendto(sockfd, buf, strlen(buf), 0, addr, alen);
        } else {
            if (fgets(buf, BUFLEN, fp) != NULL)
                sendto(sockfd, buf, strlen(buf), 0, addr, alen);
            pclose(fp);
        }
    }
}

int
main(int argc, char *argv[])
{
    struct addrinfo *ailist, *aip;
    struct addrinfo hint;
    int             sockfd, err, n;
    char            *host;
```

```
    if (argc != 1)
        err_quit("usage: ruptimed");
    if ((n = sysconf(_SC_HOST_NAME_MAX)) < 0)
        n = HOST_NAME_MAX;  /* best guess */
    if ((host = malloc(n)) == NULL)
        err_sys("malloc error");
    if (gethostname(host, n) < 0)
        err_sys("gethostname error");
    daemonize("ruptimed");
    memset(&hint, 0, sizeof(hint));
    hint.ai_flags = AI_CANONNAME;
    hint.ai_socktype = SOCK_DGRAM;
    hint.ai_canonname = NULL;
    hint.ai_addr = NULL;
    hint.ai_next = NULL;
    if ((err = getaddrinfo(host, "ruptime", &hint, &ailist)) != 0) {
        syslog(LOG_ERR, "ruptimed: getaddrinfo error: %s",
          gai_strerror(err));
        exit(1);
    }
    for (aip = ailist; aip != NULL; aip = aip->ai_next) {
        if ((sockfd = initserver(SOCK_DGRAM, aip->ai_addr,
          aip->ai_addrlen, 0)) >= 0) {
            serve(sockfd);
            exit(0);
        }
    }
    exit(1);
}
```

Figure 16.20 Server providing system uptime over datagrams

The server blocks in `recvfrom` for a request for service. When a request arrives, we save the requester's address and use `popen` to run the `uptime` command. We send the output back to the client using the `sendto` function, with the destination address set to the requester's address. □

16.6 Socket Options

The socket mechanism provides two socket-option interfaces for us to control the behavior of sockets. One interface is used to set an option, and another interface allows us to query the state of an option. We can get and set three kinds of options:

1. Generic options that work with all socket types
2. Options that are managed at the socket level, but depend on the underlying protocols for support
3. Protocol-specific options unique to each individual protocol

The Single UNIX Specification defines only the socket-layer options (the first two option types in the preceding list).

We can set a socket option with the `setsockopt` function.

```
#include <sys/socket.h>

int setsockopt(int sockfd, int level, int option, const void *val,
               socklen_t len);
```

Returns: 0 if OK, –1 on error

The *level* argument identifies the protocol to which the option applies. If the option is a generic socket-level option, then *level* is set to `SOL_SOCKET`. Otherwise, *level* is set to the number of the protocol that controls the option. Examples are `IPPROTO_TCP` for TCP options and `IPPROTO_IP` for IP options. Figure 16.21 summarizes the generic socket-level options defined by the Single UNIX Specification.

Option	Type of *val* argument	Description
`SO_ACCEPTCONN`	`int`	Return whether a socket is enabled for listening (`getsockopt` only).
`SO_BROADCAST`	`int`	Broadcast datagrams if `*val` is nonzero.
`SO_DEBUG`	`int`	Debugging in network drivers enabled if `*val` is nonzero.
`SO_DONTROUTE`	`int`	Bypass normal routing if `*val` is nonzero.
`SO_ERROR`	`int`	Return and clear pending socket error (`getsockopt` only).
`SO_KEEPALIVE`	`int`	Periodic keep-alive messages enabled if `*val` is nonzero.
`SO_LINGER`	`struct linger`	Delay time when unsent messages exist and socket is closed.
`SO_OOBINLINE`	`int`	Out-of-band data placed inline with normal data if `*val` is nonzero.
`SO_RCVBUF`	`int`	The size in bytes of the receive buffer.
`SO_RCVLOWAT`	`int`	The minimum amount of data in bytes to return on a receive call.
`SO_RCVTIMEO`	`struct timeval`	The timeout value for a socket receive call.
`SO_REUSEADDR`	`int`	Reuse addresses in `bind` if `*val` is nonzero.
`SO_SNDBUF`	`int`	The size in bytes of the send buffer.
`SO_SNDLOWAT`	`int`	The minimum amount of data in bytes to transmit in a send call.
`SO_SNDTIMEO`	`struct timeval`	The timeout value for a socket send call.
`SO_TYPE`	`int`	Identify the socket type (`getsockopt` only).

Figure 16.21 Socket options

The *val* argument points to a data structure or an integer, depending on the option. Some options are on/off switches. If the integer is nonzero, then the option is enabled. If the integer is zero, then the option is disabled. The *len* argument specifies the size of the object to which *val* points.

We can find out the current value of an option with the `getsockopt` function.

```
#include <sys/socket.h>

int getsockopt(int sockfd, int level, int option, void *restrict val,
               socklen_t *restrict lenp);
```

Returns: 0 if OK, –1 on error

The *lenp* argument is a pointer to an integer. Before calling `getsockopt`, we set the integer to the size of the buffer where the option is to be copied. If the actual size of the option is greater than this size, the option is silently truncated. If the actual size of the option is less than this size, then the integer is updated with the actual size on return.

Example

The function in Figure 16.12 fails to operate properly when the server terminates and we try to restart it immediately. Normally, the implementation of TCP will prevent us from binding the same address until a timeout expires, which is usually on the order of several minutes. Luckily, the `SO_REUSEADDR` socket option allows us to bypass this restriction, as illustrated in Figure 16.22.

```
#include "apue.h"
#include <errno.h>
#include <sys/socket.h>

int
initserver(int type, const struct sockaddr *addr, socklen_t alen,
  int qlen)
{
    int fd, err;
    int reuse = 1;

    if ((fd = socket(addr->sa_family, type, 0)) < 0)
        return(-1);
    if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &reuse,
      sizeof(int)) < 0)
        goto errout;
    if (bind(fd, addr, alen) < 0)
        goto errout;
    if (type == SOCK_STREAM || type == SOCK_SEQPACKET)
        if (listen(fd, qlen) < 0)
            goto errout;
    return(fd);

errout:
    err = errno;
    close(fd);
    errno = err;
    return(-1);
}
```

Figure 16.22 Initialize a socket endpoint for use by a server with address reuse

To enable the `SO_REUSEADDR` option, we set an integer to a nonzero value and pass the address of the integer as the *val* argument to `setsockopt`. We set the *len* argument to the size of an integer to indicate the size of the object to which *val* points. □

16.7 Out-of-Band Data

Out-of-band data is an optional feature supported by some communication protocols, allowing higher-priority delivery of data than normal. Out-of-band data is sent ahead of any data that is already queued for transmission. TCP supports out-of-band data, but UDP doesn't. The socket interface to out-of-band data is heavily influenced by TCP's implementation of out-of-band data.

TCP refers to out-of-band data as "urgent" data. TCP supports only a single byte of urgent data, but allows urgent data to be delivered out of band from the normal data delivery mechanisms. To generate urgent data, we specify the `MSG_OOB` flag to any of the three `send` functions. If we send more than one byte with the `MSG_OOB` flag, the last byte will be treated as the urgent-data byte.

When urgent data is received, we are sent the `SIGURG` signal if we have arranged for signal generation by the socket. In Sections 3.14 and 14.5.2, we saw that we could use the `F_SETOWN` command to `fcntl` to set the ownership of a socket. If the third argument to `fcntl` is positive, it specifies a process ID. If it is a negative value other than –1, it represents the process group ID. Thus, we can arrange that our process receive signals from a socket by calling

```
fcntl(sockfd, F_SETOWN, pid);
```

The `F_GETOWN` command can be used to retrieve the current socket ownership. As with the `F_SETOWN` command, a negative value represents a process group ID and a positive value represents a process ID. Thus, the call

```
owner = fcntl(sockfd, F_GETOWN, 0);
```

will return with `owner` equal to the ID of the process configured to receive signals from the socket if `owner` is positive and with the absolute value of `owner` equal to the ID of the process group configured to receive signals from the socket if `owner` is negative.

TCP supports the notion of an *urgent mark*: the point in the normal data stream where the urgent data would go. We can choose to receive the urgent data inline with the normal data if we use the `SO_OOBINLINE` socket option. To help us identify when we have reached the urgent mark, we can use the `sockatmark` function.

```
#include <sys/socket.h>

int sockatmark(int sockfd);
```

Returns: 1 if at mark, 0 if not at mark, –1 on error

When the next byte to be read is at the urgent mark, `sockatmark` will return 1.

When out-of-band data is present in a socket's read queue, the `select` function (Section 14.4.1) will return the file descriptor as having an exception condition pending. We can choose to receive the urgent data inline with the normal data, or we can use the `MSG_OOB` flag with one of the `recv` functions to receive the urgent data ahead of any other queue data. TCP queues only one byte of urgent data. If another urgent byte arrives before we receive the current one, the existing one is discarded.

16.8 Nonblocking and Asynchronous I/O

Normally, the `recv` functions will block when no data is immediately available. Similarly, the `send` functions will block when there is not enough room in the socket's output queue to send the message. This behavior changes when the socket is in nonblocking mode. In this case, these functions will fail instead of blocking, setting `errno` to either `EWOULDBLOCK` or `EAGAIN`. When this happens, we can use either `poll` or `select` to determine when we can receive or transmit data.

The Single UNIX Specification includes support for a general asynchronous I/O mechanism (recall Section 14.5). The socket mechanism has its own way of handling asynchronous I/O, but this isn't standardized in the Single UNIX Specification. Some texts refer to the classic socket-based asynchronous I/O mechanism as ''signal-based I/O'' to distinguish it from the general asynchronous I/O mechanism found in the Single UNIX Specification.

With socket-based asynchronous I/O, we can arrange to be sent the `SIGIO` signal when we can read data from a socket or when space becomes available in a socket's write queue. Enabling asynchronous I/O is a two-step process.

1. Establish socket ownership so signals can be delivered to the proper processes.
2. Inform the socket that we want it to signal us when I/O operations won't block.

We can accomplish the first step in three ways.

1. Use the `F_SETOWN` command with `fcntl`.
2. Use the `FIOSETOWN` command with `ioctl`.
3. Use the `SIOCSPGRP` command with `ioctl`.

To accomplish the second step, we have two choices.

1. Use the `F_SETFL` command with `fcntl` and enable the `O_ASYNC` file flag.
2. Use the `FIOASYNC` command with `ioctl`.

We have several options, but they are not universally supported. Figure 16.23 summarizes the support for these options provided by the platforms discussed in this text.

Mechanism	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
`fcntl(fd, F_SETOWN, pid)`	•	•	•	•	•
`ioctl(fd, FIOSETOWN, pid)`		•	•	•	•
`ioctl(fd, SIOCSPGRP, pid)`		•	•	•	•
`fcntl(fd, F_SETFL, flags|O_ASYNC)`		•	•	•	
`ioctl(fd, FIOASYNC, &n);`		•	•	•	•

Figure 16.23 Socket asynchronous I/O management commands

16.9 Summary

In this chapter, we looked at the IPC mechanisms that allow processes to communicate with other processes on different machines as well as within the same machine. We discussed how socket endpoints are named and how we can discover the addresses to use when contacting servers.

We presented examples of clients and servers that use connectionless (i.e., datagram-based) sockets and connection-oriented sockets. We briefly discussed asynchronous and nonblocking socket I/O and the interfaces used to manage socket options.

In the next chapter, we will look at some advanced IPC topics, including how we can use sockets to pass file descriptors between processes running on the same machine.

Exercises

16.1 Write a program to determine your system's byte ordering.

16.2 Write a program to print out which `stat` structure members are supported for sockets on at least two different platforms, and describe how the results differ.

16.3 The program in Figure 16.17 provides service on only a single endpoint. Modify the program to support service on multiple endpoints (each with a different address) at the same time.

16.4 Write a client program and a server program to return the number of processes currently running on a specified host computer.

16.5 In the program in Figure 16.18, the server waits for the child to execute the `uptime` command and exit before accepting the next connect request. Redesign the server so that the time to service one request doesn't delay the processing of incoming connect requests.

16.6 Write two library routines: one to enable asynchronous (signal-based) I/O on a socket and one to disable asynchronous I/O on a socket. Use Figure 16.23 to make sure that the functions work on all platforms with as many socket types as possible.

17

Advanced IPC

17.1 Introduction

In the previous two chapters, we discussed various forms of IPC, including pipes and sockets. In this chapter, we look at an advanced form of IPC—the UNIX domain socket mechanism—and see what we can do with it. With this form of IPC, we can pass open file descriptors between processes running on the same computer system, server processes can associate names with their file descriptors, and client processes running on the same system can use these names to rendezvous with the servers. We'll also see how the operating system provides a unique IPC channel per client.

17.2 UNIX Domain Sockets

UNIX domain sockets are used to communicate with processes running on the same machine. Although Internet domain sockets can be used for this same purpose, UNIX domain sockets are more efficient. UNIX domain sockets only copy data; they have no protocol processing to perform, no network headers to add or remove, no checksums to calculate, no sequence numbers to generate, and no acknowledgements to send.

UNIX domain sockets provide both stream and datagram interfaces. The UNIX domain datagram service is reliable, however. Messages are neither lost nor delivered out of order. UNIX domain sockets are like a cross between sockets and pipes. You can use the network-oriented socket interfaces with them, or you can use the `socketpair` function to create a pair of unnamed, connected, UNIX domain sockets.

```
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol, int sockfd[2]);
```

Returns: 0 if OK, –1 on error

Although the interface is sufficiently general to allow `socketpair` to be used with other domains, operating systems typically provide support only for the UNIX domain.

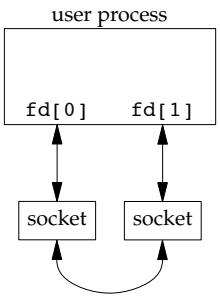


Figure 17.1 A socket pair

A pair of connected UNIX domain sockets acts like a full-duplex pipe: both ends are open for reading and writing (see Figure 17.1). We'll refer to these as "fd-pipes" to distinguish them from normal, half-duplex pipes.

Example — `fd_pipe` Function

Figure 17.2 shows the `fd_pipe` function, which uses the `socketpair` function to create a pair of connected UNIX domain stream sockets.

```
#include "apue.h"
#include <sys/socket.h>

/*
 * Returns a full-duplex pipe (a UNIX domain socket) with
 * the two file descriptors returned in fd[0] and fd[1].
 */
int
fd_pipe(int fd[2])
{
    return(socketpair(AF_UNIX, SOCK_STREAM, 0, fd));
}
```

Figure 17.2 Creating a full-duplex pipe

Some BSD-based systems use UNIX domain sockets to implement pipes. But when `pipe` is called, the write end of the first descriptor and the read end of the second descriptor are both closed. To get a full-duplex pipe, we must call `socketpair` directly. □

Example — Polling XSI Message Queues with the Help of UNIX Domain Sockets

In Section 15.6.4, we said one of the problems with using XSI message queues is that we can't use `poll` or `select` with them, because they aren't associated with file descriptors. However, sockets *are* associated with file descriptors, and we can use them to notify us when messages arrive. We'll use one thread per message queue. Each thread will block in a call to `msgrcv`. When a message arrives, the thread will write it down one end of a UNIX domain socket. Our application will use the other end of the socket to receive the message when `poll` indicates data can be read from the socket.

The program in Figure 17.3 illustrates this technique. The `main` function creates the message queues and UNIX domain sockets and starts one thread to service each queue. Then it uses an infinite loop to poll one end of the sockets. When a socket is readable, it reads from the socket and writes the message on the standard output.

```
#include "apue.h"
#include <poll.h>
#include <pthread.h>
#include <sys/msg.h>
#include <sys/socket.h>

#define NQ      3       /* number of queues */
#define MAXMSZ  512     /* maximum message size */
#define KEY     0x123   /* key for first message queue */

struct threadinfo {
    int qid;
    int fd;
};

struct mymesg {
    long mtype;
    char mtext[MAXMSZ];
};

void *
helper(void *arg)
{
    int                 n;
    struct mymesg       m;
    struct threadinfo   *tip = arg;

    for(;;) {
        memset(&m, 0, sizeof(m));
        if ((n = msgrcv(tip->qid, &m, MAXMSZ, 0, MSG_NOERROR)) < 0)
            err_sys("msgrcv error");
        if (write(tip->fd, m.mtext, n) < 0)
            err_sys("write error");
    }
}

int
main()
```

```
{
    int                 i, n, err;
    int                 fd[2];
    int                 qid[NQ];
    struct pollfd       pfd[NQ];
    struct threadinfo   ti[NQ];
    pthread_t           tid[NQ];
    char                buf[MAXMSZ];

    for (i = 0; i < NQ; i++) {
        if ((qid[i] = msgget((KEY+i), IPC_CREAT|0666)) < 0)
            err_sys("msgget error");

        printf("queue ID %d is %d\n", i, qid[i]);

        if (socketpair(AF_UNIX, SOCK_DGRAM, 0, fd) < 0)
            err_sys("socketpair error");
        pfd[i].fd = fd[0];
        pfd[i].events = POLLIN;
        ti[i].qid = qid[i];
        ti[i].fd = fd[1];
        if ((err = pthread_create(&tid[i], NULL, helper, &ti[i])) != 0)
            err_exit(err, "pthread_create error");
    }

    for (;;) {
        if (poll(pfd, NQ, -1) < 0)
            err_sys("poll error");
        for (i = 0; i < NQ; i++) {
            if (pfd[i].revents & POLLIN) {
                if ((n = read(pfd[i].fd, buf, sizeof(buf))) < 0)
                    err_sys("read error");
                buf[n] = 0;
                printf("queue id %d, message %s\n", qid[i], buf);
            }
        }
    }

    exit(0);
}
```

Figure 17.3 Poll for XSI messages using UNIX domain sockets

Note that we use datagram (`SOCK_DGRAM`) sockets instead of stream sockets. This allows us to retain message boundaries so when we read from the socket, we read only one message at a time.

This technique allows us to use either `poll` or `select` (indirectly) with message queues. As long as the costs of one thread per queue and copying each message two extra times (once to write it to the socket and once to read it from the socket) are acceptable, this technique will make it easier to use XSI message queues.

Well use the program shown in Figure 17.4 to send messages to our test program from Figure 17.3.

```
#include "apue.h"
#include <sys/msg.h>

#define MAXMSZ 512

struct mymesg {
    long mtype;
    char mtext[MAXMSZ];
};

int
main(int argc, char *argv[])
{
    key_t key;
    long qid;
    size_t nbytes;
    struct mymesg m;

    if (argc != 3) {
        fprintf(stderr, "usage: sendmsg KEY message\n");
        exit(1);
    }
    key = strtol(argv[1], NULL, 0);
    if ((qid = msgget(key, 0)) < 0)
        err_sys("can't open queue key %s", argv[1]);
    memset(&m, 0, sizeof(m));
    strncpy(m.mtext, argv[2], MAXMSZ-1);
    nbytes = strlen(m.mtext);
    m.mtype = 1;
    if (msgsnd(qid, &m, nbytes, 0) < 0)
        err_sys("can't send message");
    exit(0);
}
```

Figure 17.4 Post a message to an XSI message queue

This program takes two arguments: the key associated with the queue and a string to be sent as the body of the message. When we send messages to the server, it prints them as shown below.

```
$ ./pollmsg &                          run the server in the background
[1] 12814
$ queue ID 0 is 196608
queue ID 1 is 196609
queue ID 2 is 196610
$ ./sendmsg 0x123 "hello, world"       send a message to the first queue
queue id 196608, message hello, world
$ ./sendmsg 0x124 "just a test"        send a message to the second queue
queue id 196609, message just a test
$ ./sendmsg 0x125 "bye"                send a message to the third queue
queue id 196610, message bye
```

□

17.2.1 Naming UNIX Domain Sockets

Although the socketpair function creates sockets that are connected to each other, the individual sockets don't have names. This means that they can't be addressed by unrelated processes.

In Section 16.3.4, we learned how to bind an address to an Internet domain socket. Just as with Internet domain sockets, UNIX domain sockets can be named and used to advertise services. The address format used with UNIX domain sockets differs from that used with Internet domain sockets, however.

Recall from Section 16.3 that socket address formats differ from one implementation to the next. An address for a UNIX domain socket is represented by a sockaddr_un structure. On Linux 3.2.0 and Solaris 10, the sockaddr_un structure is defined in the header <sys/un.h> as

```
struct sockaddr_un {
    sa_family_t  sun_family;     /* AF_UNIX */
    char         sun_path[108];  /* pathname */
};
```

On FreeBSD 8.0 and Mac OS X 10.6.8, however, the sockaddr_un structure is defined as

```
struct sockaddr_un {
    unsigned char  sun_len;        /* sockaddr length */
    sa_family_t    sun_family;     /* AF_UNIX */
    char           sun_path[104];  /* pathname */
};
```

The sun_path member of the sockaddr_un structure contains a pathname. When we bind an address to a UNIX domain socket, the system creates a file of type S_IFSOCK with the same name.

This file exists only as a means of advertising the socket name to clients. The file can't be opened or otherwise used for communication by applications.

If the file already exists when we try to bind the same address, the bind request will fail. When we close the socket, this file is not automatically removed, so we need to make sure that we unlink it before our application exits.

Example

The program in Figure 17.5 shows an example of binding an address to a UNIX domain socket.

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>

int
main(void)
{
    int fd, size;
```

```
    struct sockaddr_un un;

    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, "foo.socket");
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        err_sys("socket failed");
    size = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);
    if (bind(fd, (struct sockaddr *)&un, size) < 0)
        err_sys("bind failed");
    printf("UNIX domain socket bound\n");
    exit(0);
}
```

Figure 17.5 Binding an address to a UNIX domain socket

When we run this program, the `bind` request succeeds. If we run the program a second time, however, we get an error, because the file already exists. The program won't succeed again until we remove the file.

```
$ ./a.out                                          run the program
UNIX domain socket bound
$ ls -l foo.socket                                 look at the socket file
srwxr-xr-x  1 sar       0 May 18 00:44 foo.socket
$ ./a.out                                          try to run the program again
bind failed: Address already in use
$ rm foo.socket                                    remove the socket file
$ ./a.out                                          run the program a third time
UNIX domain socket bound                           now it succeeds
```

The way we determine the size of the address to bind is to calculate the offset of the `sun_path` member in the `sockaddr_un` structure and add to it the length of the pathname, not including the terminating null byte. Since implementations vary in which members precede `sun_path` in the `sockaddr_un` structure, we use the `offsetof` macro from `<stddef.h>` (included by `apue.h`) to calculate the offset of the `sun_path` member from the start of the structure. If you look in `<stddef.h>`, you'll see a definition similar to the following:

```
#define offsetof(TYPE, MEMBER)   ((int)&((TYPE *)0)->MEMBER)
```

The expression evaluates to an integer, which is the starting address of the member, assuming that the structure begins at address 0. □

17.3 Unique Connections

A server can arrange for unique UNIX domain connections to clients using the standard `bind`, `listen`, and `accept` functions. Clients use `connect` to contact the server; after the connect request is accepted by the server, a unique connection exists between the client and the server. This style of operation is the same that we illustrated with Internet domain sockets in Figures 16.16 and 16.17.

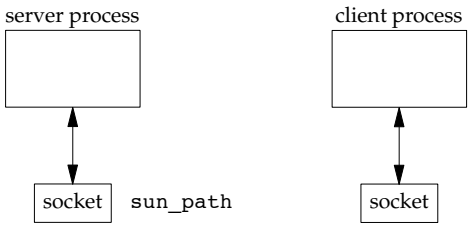


Figure 17.6 Client and server sockets before a `connect`

Figure 17.6 shows a client process and a server process before a connection exists between the two. The server has bound its socket to a `sockaddr_un` address and is listening for connection requests. Figure 17.7 shows the unique connection between the client and server after the server has accepted the client's connection request.

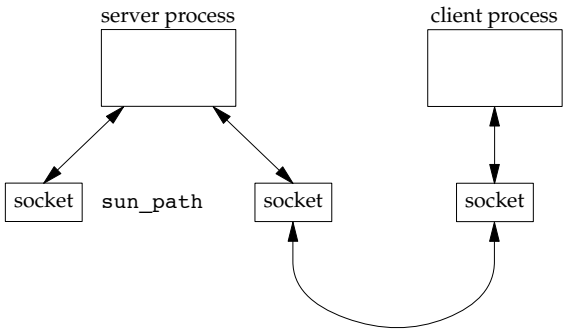


Figure 17.7 Client and server sockets after a `connect`

We will now develop three functions that can be used to create unique connections between unrelated processes running on the same machine. These functions mimic the connection-oriented socket functions discussed in Section 16.4. We use UNIX domain sockets for the underlying communication mechanism here.

```
#include "apue.h"

int serv_listen(const char *name);
```

Returns: file descriptor to listen on if OK, negative value on error

```
int serv_accept(int listenfd, uid_t *uidptr);
```

Returns: new file descriptor if OK, negative value on error

```
int cli_conn(const char *name);
```

Returns: file descriptor if OK, negative value on error

The `serv_listen` function (Figure 17.8) can be used by a server to announce its willingness to listen for client connect requests on a well-known name (some pathname in the file system). Clients will use this name when they want to connect to the server. The return value is the server's UNIX domain socket used to receive client connection requests.

The `serv_accept` function (Figure 17.9) is used by a server to wait for a client's connect request to arrive. When one arrives, the system automatically creates a new UNIX domain socket, connects it to the client's socket, and returns the new socket to the server. Additionally, the effective user ID of the client is stored in the memory to which *uidptr* points.

A client calls `cli_conn` (Figure 17.10) to connect to a server. The *name* argument specified by the client must be the same name that was advertised by the server's call to `serv_listen`. On return, the client gets a file descriptor connected to the server.

Figure 17.8 shows the `serv_listen` function.

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>

#define QLEN    10

/*
 * Create a server endpoint of a connection.
 * Returns fd if all OK, <0 on error.
 */
int
serv_listen(const char *name)
{
    int                 fd, len, err, rval;
    struct sockaddr_un  un;

    if (strlen(name) >= sizeof(un.sun_path)) {
        errno = ENAMETOOLONG;
        return(-1);
    }

    /* create a UNIX domain stream socket */
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-2);

    unlink(name);   /* in case it already exists */

    /* fill in socket address structure */
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, name);
    len = offsetof(struct sockaddr_un, sun_path) + strlen(name);

    /* bind the name to the descriptor */
    if (bind(fd, (struct sockaddr *)&un, len) < 0) {
        rval = -3;
```

```
        goto errout;
    }

    if (listen(fd, QLEN) < 0) { /* tell kernel we're a server */
        rval = -4;
        goto errout;
    }
    return(fd);

errout:
    err = errno;
    close(fd);
    errno = err;
    return(rval);
}
```

Figure 17.8 The `serv_listen` function

First, we create a single UNIX domain socket by calling `socket`. We then fill in a `sockaddr_un` structure with the well-known pathname to be assigned to the socket. This structure is the argument to `bind`. Note that we don't need to set the `sun_len` field present on some platforms, because the operating system sets this for us, deriving it from the address length we pass to the `bind` function.

Finally, we call `listen` (Section 16.4) to tell the kernel that the process will be acting as a server awaiting connections from clients. When a connect request from a client arrives, the server calls the `serv_accept` function (Figure 17.9).

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <time.h>
#include <errno.h>

#define STALE   30  /* client's name can't be older than this (sec) */

/*
 * Wait for a client connection to arrive, and accept it.
 * We also obtain the client's user ID from the pathname
 * that it must bind before calling us.
 * Returns new fd if all OK, <0 on error
 */
int
serv_accept(int listenfd, uid_t *uidptr)
{
    int                 clifd, err, rval;
    socklen_t           len;
    time_t              staletime;
    struct sockaddr_un  un;
    struct stat         statbuf;
    char                *name;

    /* allocate enough space for longest name plus terminating null */
```

```
    if ((name = malloc(sizeof(un.sun_path + 1))) == NULL)
        return(-1);
    len = sizeof(un);
    if ((clifd = accept(listenfd, (struct sockaddr *)&un, &len)) < 0) {
        free(name);
        return(-2);     /* often errno=EINTR, if signal caught */
    }

    /* obtain the client's uid from its calling address */
    len -= offsetof(struct sockaddr_un, sun_path); /* len of pathname */
    memcpy(name, un.sun_path, len);
    name[len] = 0;          /* null terminate */
    if (stat(name, &statbuf) < 0) {
        rval = -3;
        goto errout;
    }

#ifdef  S_ISSOCK    /* not defined for SVR4 */
    if (S_ISSOCK(statbuf.st_mode) == 0) {
        rval = -4;      /* not a socket */
        goto errout;
    }
#endif

    if ((statbuf.st_mode & (S_IRWXG | S_IRWXO)) ||
        (statbuf.st_mode & S_IRWXU) != S_IRWXU) {
          rval = -5;    /* is not rwx------ */
          goto errout;
    }

    staletime = time(NULL) - STALE;
    if (statbuf.st_atime < staletime ||
        statbuf.st_ctime < staletime ||
        statbuf.st_mtime < staletime) {
          rval = -6;    /* i-node is too old */
          goto errout;
    }

    if (uidptr != NULL)
        *uidptr = statbuf.st_uid;   /* return uid of caller */
    unlink(name);       /* we're done with pathname now */
    free(name);
    return(clifd);

errout:
    err = errno;
    close(clifd);
    free(name);
    errno = err;
    return(rval);
}
```

Figure 17.9 The `serv_accept` function

The server blocks in the call to `accept`, waiting for a client to call `cli_conn`. When `accept` returns, its return value is a brand-new descriptor that is connected to the client. Additionally, the pathname that the client assigned to its socket (the name that contained the client's process ID) is returned by `accept`, through the second argument (the pointer to the `sockaddr_un` structure). We copy this pathname and ensure that it is null terminated (if the pathname takes up all available space in the `sun_path` member of the `sockaddr_un` structure, there won't be room for the terminating null byte). Then we call `stat` to verify that the pathname is indeed a socket and that the permissions allow only user-read, user-write, and user-execute. We also verify that the three times associated with the socket are no older than 30 seconds. (Recall from Section 6.10 that the `time` function returns the current time and date in seconds past the Epoch.)

If all these checks are OK, we assume that the identity of the client (its effective user ID) is the owner of the socket. Although this check isn't perfect, it's the best we can do with current systems. (It would be better if the kernel returned the effective user ID to us through a parameter to `accept`.)

The client initiates the connection to the server by calling the `cli_conn` function (Figure 17.10).

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>

#define CLI_PATH    "/var/tmp/"
#define CLI_PERM    S_IRWXU         /* rwx for user only */

/*
 * Create a client endpoint and connect to a server.
 * Returns fd if all OK, <0 on error.
 */
int
cli_conn(const char *name)
{
    int                 fd, len, err, rval;
    struct sockaddr_un  un, sun;
    int                 do_unlink = 0;

    if (strlen(name) >= sizeof(un.sun_path)) {
        errno = ENAMETOOLONG;
        return(-1);
    }

    /* create a UNIX domain stream socket */
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        return(-1);

    /* fill socket address structure with our address */
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    sprintf(un.sun_path, "%s%05ld", CLI_PATH, (long)getpid());
```

```
    len = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);

    unlink(un.sun_path);        /* in case it already exists */
    if (bind(fd, (struct sockaddr *)&un, len) < 0) {
        rval = -2;
        goto errout;
    }
    if (chmod(un.sun_path, CLI_PERM) < 0) {
        rval = -3;
        do_unlink = 1;
        goto errout;
    }

    /* fill socket address structure with server's address */
    memset(&sun, 0, sizeof(sun));
    sun.sun_family = AF_UNIX;
    strcpy(sun.sun_path, name);
    len = offsetof(struct sockaddr_un, sun_path) + strlen(name);
    if (connect(fd, (struct sockaddr *)&sun, len) < 0) {
        rval = -4;
        do_unlink = 1;
        goto errout;
    }
    return(fd);

errout:
    err = errno;
    close(fd);
    if (do_unlink)
        unlink(un.sun_path);
    errno = err;
    return(rval);
}
```

Figure 17.10 The `cli_conn` function

We call `socket` to create the client's end of a UNIX domain socket. We then fill in a `sockaddr_un` structure with a client-specific name.

We don't let the system choose a default address for us, because the server would be unable to distinguish one client from another (if we don't explicitly bind a name to a UNIX domain socket, the kernel implicitly binds an address to it on our behalf and no file is created in the file system to represent the socket). Instead, we bind our own address—a step we usually don't take when developing a client program that uses sockets.

The last five characters of the pathname we bind are made from the process ID of the client. We call `unlink`, just in case the pathname already exists. We then call `bind` to assign a name to the client's socket. This creates a socket file in the file system with the same name as the bound pathname. We call `chmod` to turn off all permissions other than user-read, user-write, and user-execute. In `serv_accept`, the server checks these permissions and the user ID of the socket to verify the client's identity.

We then have to fill in another `sockaddr_un` structure, this time with the well-known pathname of the server. Finally, we call the `connect` function to initiate the connection with the server.

17.4 Passing File Descriptors

Passing an open file descriptor between processes is a powerful technique. It can lead to different ways of designing client–server applications. It allows one process (typically a server) to do everything that is required to open a file (involving such details as translating a network name to a network address, dialing a modem, and negotiating locks for the file) and simply pass back to the calling process a descriptor that can be used with all the I/O functions. All the details involved in opening the file or device are hidden from the client.

We must be more specific about what we mean by "passing an open file descriptor" from one process to another. Recall Figure 3.8, which showed two processes that have opened the same file. Although they share the same v-node, each process has its own file table entry.

When we pass an open file descriptor from one process to another, we want the passing process and the receiving process to share the same file table entry. Figure 17.11 shows the desired arrangement.

Technically, we are passing a pointer to an open file table entry from one process to another. This pointer is assigned the first available descriptor in the receiving process. (Saying that we are passing an open descriptor mistakenly gives the impression that the descriptor number in the receiving process is the same as in the sending process, which usually isn't true.) Having two processes share an open file table is exactly what happens after a `fork` (recall Figure 8.2).

What normally happens when a descriptor is passed from one process to another is that the sending process, after passing the descriptor, then closes the descriptor. Closing the descriptor by the sender doesn't really close the file or device, since the descriptor is still considered open by the receiving process (even if the receiver hasn't specifically received the descriptor yet).

We define the following three functions that we use in this chapter to send and receive file descriptors. Later in this section, we'll show the code for these three functions.

```
#include "apue.h"

int send_fd(int fd, int fd_to_send);

int send_err(int fd, int status, const char *errmsg);
```

Both return: 0 if OK, –1 on error

```
int recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t));
```

Returns: file descriptor if OK, negative value on error

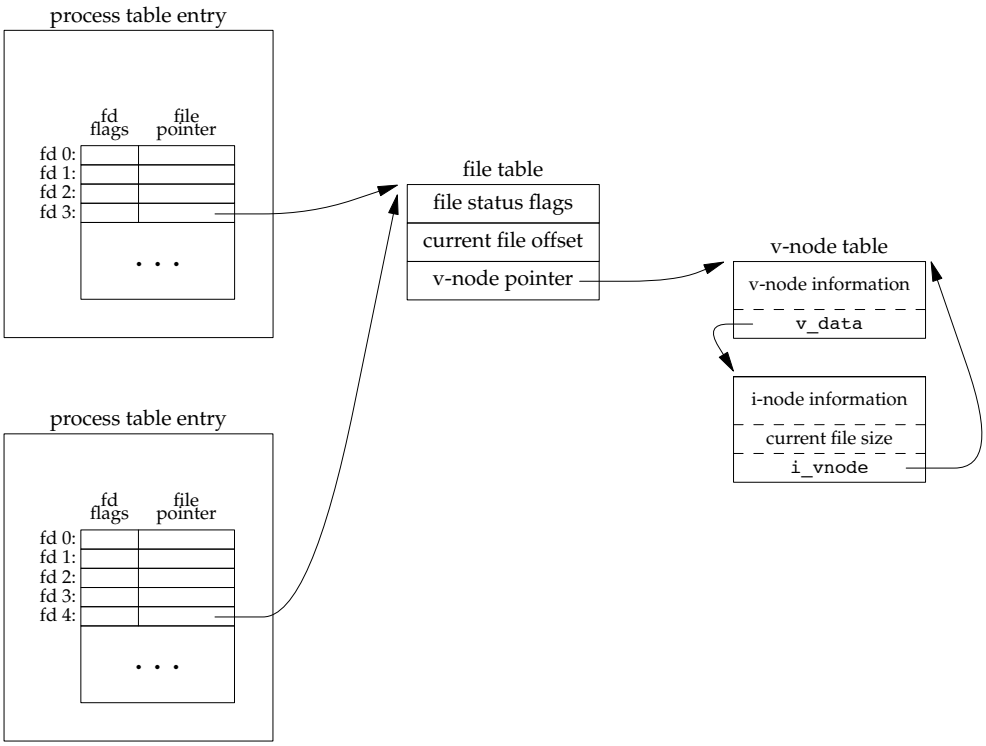


Figure 17.11 Passing an open file from the top process to the bottom process

A process (normally a server) that wants to pass a descriptor to another process calls either `send_fd` or `send_err`. The process waiting to receive the descriptor (the client) calls `recv_fd`.

The `send_fd` function sends the descriptor *fd_to_send* across using the UNIX domain socket represented by *fd*. The `send_err` function sends the *errmsg* using *fd*, followed by the *status* byte. The value of *status* must be in the range –1 through –255.

Clients call `recv_fd` to receive a descriptor. If all is OK (the sender called `send_fd`), the non-negative descriptor is returned as the value of the function. Otherwise, the value returned is the *status* that was sent by `send_err` (a negative value in the range –1 through –255). Additionally, if an error message was sent by the server, the client's *userfunc* is called to process the message. The first argument to *userfunc* is the constant `STDERR_FILENO`, followed by a pointer to the error message and its length. The return value from *userfunc* is the number of bytes written or a negative number on error. Often, the client specifies the normal `write` function as the *userfunc*.

We implement our own protocol that is used by these three functions. To send a descriptor, `send_fd` sends two bytes of 0, followed by the actual descriptor. To send an error, `send_err` sends the *errmsg*, followed by a byte of 0, followed by the absolute value of the *status* byte (1 through 255). The `recv_fd` function reads everything on the

socket until it encounters a null byte. Any characters read up to this point are passed to the caller's *userfunc*. The next byte read by `recv_fd` is the status byte. If the status byte is 0, a descriptor was passed; otherwise, there is no descriptor to receive.

The function `send_err` calls the `send_fd` function after writing the error message to the socket. This is shown in Figure 17.12.

```
#include "apue.h"

/*
 * Used when we had planned to send an fd using send_fd(),
 * but encountered an error instead.  We send the error back
 * using the send_fd()/recv_fd() protocol.
 */
int
send_err(int fd, int errcode, const char *msg)
{
    int     n;

    if ((n = strlen(msg)) > 0)
        if (writen(fd, msg, n) != n)    /* send the error message */
            return(-1);

    if (errcode >= 0)
        errcode = -1;   /* must be negative */

    if (send_fd(fd, errcode) < 0)
        return(-1);

    return(0);
}
```

Figure 17.12 The `send_err` function

To exchange file descriptors using UNIX domain sockets, we call the `sendmsg`(2) and `recvmsg`(2) functions (Section 16.5). Both functions take a pointer to a `msghdr` structure that contains all the information on what to send or receive. The structure on your system might look similar to the following:

```
struct msghdr {
    void         *msg_name;        /* optional address */
    socklen_t     msg_namelen;     /* address size in bytes */
    struct iovec *msg_iov;         /* array of I/O buffers */
    int           msg_iovlen;      /* number of elements in array */
    void         *msg_control;     /* ancillary data */
    socklen_t     msg_controllen;  /* number of ancillary bytes */
    int           msg_flags;       /* flags for received message */
};
```

The first two elements are normally used for sending datagrams on a network connection, where the destination address can be specified with each datagram. The next two elements allow us to specify an array of buffers (scatter read or gather write), as we described for the `readv` and `writev` functions (Section 14.6). The `msg_flags` field contains flags describing the message received, as summarized in Figure 16.15.

Two elements deal with the passing or receiving of control information. The `msg_control` field points to a `cmsghdr` (control message header) structure, and the `msg_controllen` field contains the number of bytes of control information.

```
struct cmsghdr {
    socklen_t  cmsg_len;    /* data byte count, including header */
    int        cmsg_level;  /* originating protocol */
    int        cmsg_type;   /* protocol-specific type */
    /* followed by the actual control message data */
};
```

To send a file descriptor, we set `cmsg_len` to the size of the `cmsghdr` structure, plus the size of an integer (the descriptor). The `cmsg_level` field is set to `SOL_SOCKET`, and `cmsg_type` is set to `SCM_RIGHTS`, to indicate that we are passing access rights. (`SCM` stands for *socket-level control message*.) Access rights can be passed only across a UNIX domain socket. The descriptor is stored right after the `cmsg_type` field, using the macro `CMSG_DATA` to obtain the pointer to this integer.

Three macros are used to access the control data, and one macro is used to help calculate the value to be used for `cmsg_len`.

```
#include <sys/socket.h>

unsigned char *CMSG_DATA(struct cmsghdr *cp);
```

Returns: pointer to data associated with `cmsghdr` structure

```
struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *mp);
```

Returns: pointer to first `cmsghdr` structure associated with the `msghdr` structure, or `NULL` if none exists

```
struct cmsghdr *CMSG_NXTHDR(struct msghdr *mp,
                            struct cmsghdr *cp);
```

Returns: pointer to next `cmsghdr` structure associated with the `msghdr` structure given the current `cmsghdr` structure, or `NULL` if we're at the last one

```
unsigned int CMSG_LEN(unsigned int nbytes);
```

Returns: size to allocate for data object *nbytes* large

The Single UNIX Specification defines the first three macros, but omits `CMSG_LEN`.

The `CMSG_LEN` macro returns the number of bytes needed to store a data object of size *nbytes*, after adding the size of the `cmsghdr` structure, adjusting for any alignment constraints required by the processor architecture, and rounding up.

The program in Figure 17.13 is the `send_fd` function, which passes a file descriptor over a UNIX domain socket. In the `sendmsg` call, we send both the protocol data (the null and the status byte) and the descriptor.

```
#include "apue.h"
#include <sys/socket.h>

/* size of control buffer to send/recv one file descriptor */
#define CONTROLLEN  CMSG_LEN(sizeof(int))

static struct cmsghdr   *cmptr = NULL;  /* malloc'ed first time */

/*
 * Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status.
 */
int
send_fd(int fd, int fd_to_send)
{
    struct iovec    iov[1];
    struct msghdr   msg;
    char            buf[2]; /* send_fd()/recv_fd() 2-byte protocol */

    iov[0].iov_base = buf;
    iov[0].iov_len  = 2;
    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;

    if (fd_to_send < 0) {
        msg.msg_control    = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd_to_send;   /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        cmptr->cmsg_level  = SOL_SOCKET;
        cmptr->cmsg_type   = SCM_RIGHTS;
        cmptr->cmsg_len    = CONTROLLEN;
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        *(int *)CMSG_DATA(cmptr) = fd_to_send;      /* the fd to pass */
        buf[1] = 0;     /* zero status means OK */
    }

    buf[0] = 0;         /* null byte flag to recv_fd() */
    if (sendmsg(fd, &msg, 0) != 2)
        return(-1);
    return(0);
}
```

Figure 17.13 Sending a file descriptor over a UNIX domain socket

To receive a descriptor (Figure 17.14), we allocate enough room for a `cmsghdr` structure and a descriptor, set `msg_control` to point to the allocated area, and call `recvmsg`. We use the `CMSG_LEN` macro to calculate the amount of space needed.

We read from the socket until we read the null byte that precedes the final status byte. Everything up to this null byte is an error message from the sender.

```
#include "apue.h"
#include <sys/socket.h>     /* struct msghdr */

/* size of control buffer to send/recv one file descriptor */
#define CONTROLLEN  CMSG_LEN(sizeof(int))

static struct cmsghdr   *cmptr = NULL;      /* malloc'ed first time */

/*
 * Receive a file descriptor from a server process.  Also, any data
 * received is passed to (*userfunc)(STDERR_FILENO, buf, nbytes).
 * We have a 2-byte protocol for receiving the fd from send_fd().
 */
int
recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t))
{
    int             newfd, nr, status;
    char            *ptr;
    char            buf[MAXLINE];
    struct iovec    iov[1];
    struct msghdr   msg;

    status = -1;
    for ( ; ; ) {
        iov[0].iov_base = buf;
        iov[0].iov_len  = sizeof(buf);
        msg.msg_iov     = iov;
        msg.msg_iovlen  = 1;
        msg.msg_name    = NULL;
        msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        if ((nr = recvmsg(fd, &msg, 0)) < 0) {
            err_ret("recvmsg error");
            return(-1);
        } else if (nr == 0) {
            err_ret("connection closed by server");
            return(-1);
        }

        /*
         * See if this is the final data with null & status.  Null
         * is next to last byte of buffer; status byte is last byte.
         * Zero status means there is a file descriptor to receive.
         */
```

```
        for (ptr = buf; ptr < &buf[nr]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nr-1])
                    err_dump("message format error");
                status = *ptr & 0xFF;   /* prevent sign extension */
                if (status == 0) {
                    if (msg.msg_controllen != CONTROLLEN)
                        err_dump("status = 0 but no fd");
                    newfd = *(int *)CMSG_DATA(cmptr);
                } else {
                    newfd = -status;
                }
                nr -= 2;
            }
        }
        if (nr > 0 && (*userfunc)(STDERR_FILENO, buf, nr) != nr)
            return(-1);
        if (status >= 0)    /* final data has arrived */
            return(newfd);  /* descriptor, or -status */
    }
}
```

Figure 17.14 Receiving a file descriptor over a UNIX domain socket

Note that we are always prepared to receive a descriptor (we set `msg_control` and `msg_controllen` before each call to `recvmsg`), but only if `msg_controllen` is nonzero on return did we actually receive a descriptor.

Recall the hoops we needed to jump through to determine the identity of the caller in the `serv_accept` function (Figure 17.9). It would have been better for the kernel to pass us the credentials of the caller on return from the call to `accept`. Some UNIX domain socket implementations provide similar functionality when exchanging messages, but their interfaces differ.

> FreeBSD 8.0 and Linux 3.2.0 provide support for sending credentials over UNIX domain sockets, but they do it differently. Mac OS X 10.6.8 is derived in part from FreeBSD, but has credential passing disabled. Solaris 10 doesn't support sending credentials over UNIX domain sockets. However, it supports the ability to obtain the credentials of a process passing a file descriptor over a STREAMS pipe, although we do not discuss the details here.

With FreeBSD, credentials are transmitted as a `cmsgcred` structure:

```
#define CMGROUP_MAX 16

struct cmsgcred {
    pid_t  cmcred_pid;                  /* sender's process ID */
    uid_t  cmcred_uid;                  /* sender's real UID */
    uid_t  cmcred_euid;                 /* sender's effective UID */
    gid_t  cmcred_gid;                  /* sender's real GID */
    short  cmcred_ngroups;              /* number of groups */
    gid_t  cmcred_groups[CMGROUP_MAX];  /* groups */
};
```

When we transmit credentials, we need to reserve space only for the `cmsgcred` structure. The kernel will fill in this structure for us to prevent an application from pretending to have a different identity.

On Linux, credentials are transmitted as a `ucred` structure:

```
struct ucred {
    pid_t  pid;  /* sender's process ID */
    uid_t  uid;  /* sender's user ID */
    gid_t  gid;  /* sender's group ID */
};
```

Unlike FreeBSD, Linux requires that we initialize this structure before transmission. The kernel will ensure that applications either use values that correspond to the caller or have the appropriate privilege to use other values.

Figure 17.15 shows the `send_fd` function updated to include the credentials of the sending process.

```
#include "apue.h"
#include <sys/socket.h>

#if defined(SCM_CREDS)          /* BSD interface */
#define CREDSTRUCT      cmsgcred
#define SCM_CREDTYPE    SCM_CREDS
#elif defined(SCM_CREDENTIALS)  /* Linux interface */
#define CREDSTRUCT      ucred
#define SCM_CREDTYPE    SCM_CREDENTIALS
#else
#error passing credentials is unsupported!
#endif

/* size of control buffer to send/recv one file descriptor */
#define RIGHTSLEN   CMSG_LEN(sizeof(int))
#define CREDSLEN    CMSG_LEN(sizeof(struct CREDSTRUCT))
#define CONTROLLEN  (RIGHTSLEN + CREDSLEN)

static struct cmsghdr   *cmptr = NULL;  /* malloc'ed first time */

/*
 * Pass a file descriptor to another process.
 * If fd<0, then -fd is sent back instead as the error status.
 */
int
send_fd(int fd, int fd_to_send)
{
    struct CREDSTRUCT   *credp;
    struct cmsghdr      *cmp;
    struct iovec        iov[1];
    struct msghdr       msg;
    char                buf[2]; /* send_fd/recv_ufd 2-byte protocol */

    iov[0].iov_base = buf;
    iov[0].iov_len  = 2;
    msg.msg_iov     = iov;
    msg.msg_iovlen  = 1;
```

```
    msg.msg_name    = NULL;
    msg.msg_namelen = 0;
    msg.msg_flags = 0;
    if (fd_to_send < 0) {
        msg.msg_control    = NULL;
        msg.msg_controllen = 0;
        buf[1] = -fd_to_send;   /* nonzero status means error */
        if (buf[1] == 0)
            buf[1] = 1; /* -256, etc. would screw up protocol */
    } else {
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        cmp = cmptr;
        cmp->cmsg_level  = SOL_SOCKET;
        cmp->cmsg_type   = SCM_RIGHTS;
        cmp->cmsg_len    = RIGHTSLEN;
        *(int *)CMSG_DATA(cmp) = fd_to_send;    /* the fd to pass */
        cmp = CMSG_NXTHDR(&msg, cmp);
        cmp->cmsg_level  = SOL_SOCKET;
        cmp->cmsg_type   = SCM_CREDTYPE;
        cmp->cmsg_len    = CREDSLEN;
        credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);
#if defined(SCM_CREDENTIALS)
        credp->uid = geteuid();
        credp->gid = getegid();
        credp->pid = getpid();
#endif
        buf[1] = 0;     /* zero status means OK */
    }
    buf[0] = 0;         /* null byte flag to recv_ufd() */
    if (sendmsg(fd, &msg, 0) != 2)
        return(-1);
    return(0);
}
```

Figure 17.15 Sending credentials over UNIX domain sockets

Note that we need to initialize the credentials structure only on Linux.

The function in Figure 17.16 is a modified version of `recv_fd`, called `recv_ufd`, that returns the user ID of the sender through a reference parameter.

```
#include "apue.h"
#include <sys/socket.h>     /* struct msghdr */
#include <sys/un.h>

#if defined(SCM_CREDS)          /* BSD interface */
#define CREDSTRUCT      cmsgcred
#define CR_UID          cmcred_uid
#define SCM_CREDTYPE    SCM_CREDS
#elif defined(SCM_CREDENTIALS)  /* Linux interface */
```

```
#define CREDSTRUCT      ucred
#define CR_UID          uid
#define CREDOPT         SO_PASSCRED
#define SCM_CREDTYPE    SCM_CREDENTIALS
#else
#error passing credentials is unsupported!
#endif

/* size of control buffer to send/recv one file descriptor */
#define RIGHTSLEN   CMSG_LEN(sizeof(int))
#define CREDSLEN    CMSG_LEN(sizeof(struct CREDSTRUCT))
#define CONTROLLEN  (RIGHTSLEN + CREDSLEN)

static struct cmsghdr   *cmptr = NULL;      /* malloc'ed first time */

/*
 * Receive a file descriptor from a server process.  Also, any data
 * received is passed to (*userfunc)(STDERR_FILENO, buf, nbytes).
 * We have a 2-byte protocol for receiving the fd from send_fd().
 */
int
recv_ufd(int fd, uid_t *uidptr,
         ssize_t (*userfunc)(int, const void *, size_t))
{
    struct cmsghdr      *cmp;
    struct CREDSTRUCT   *credp;
    char                *ptr;
    char                buf[MAXLINE];
    struct iovec        iov[1];
    struct msghdr       msg;
    int                 nr;
    int                 newfd = -1;
    int                 status = -1;
#if defined(CREDOPT)
    const int           on = 1;

    if (setsockopt(fd, SOL_SOCKET, CREDOPT, &on, sizeof(int)) < 0) {
        err_ret("setsockopt error");
        return(-1);
    }
#endif
    for ( ; ; ) {
        iov[0].iov_base = buf;
        iov[0].iov_len  = sizeof(buf);
        msg.msg_iov     = iov;
        msg.msg_iovlen  = 1;
        msg.msg_name    = NULL;
        msg.msg_namelen = 0;
        if (cmptr == NULL && (cmptr = malloc(CONTROLLEN)) == NULL)
            return(-1);
        msg.msg_control    = cmptr;
        msg.msg_controllen = CONTROLLEN;
        if ((nr = recvmsg(fd, &msg, 0)) < 0) {
```

```
            err_ret("recvmsg error");
            return(-1);
        } else if (nr == 0) {
            err_ret("connection closed by server");
            return(-1);
        }

        /*
         * See if this is the final data with null & status.  Null
         * is next to last byte of buffer; status byte is last byte.
         * Zero status means there is a file descriptor to receive.
         */
        for (ptr = buf; ptr < &buf[nr]; ) {
            if (*ptr++ == 0) {
                if (ptr != &buf[nr-1])
                    err_dump("message format error");
                status = *ptr & 0xFF;   /* prevent sign extension */
                if (status == 0) {
                    if (msg.msg_controllen != CONTROLLEN)
                        err_dump("status = 0 but no fd");

                    /* process the control data */
                    for (cmp = CMSG_FIRSTHDR(&msg);
                      cmp != NULL; cmp = CMSG_NXTHDR(&msg, cmp)) {
                        if (cmp->cmsg_level != SOL_SOCKET)
                            continue;
                        switch (cmp->cmsg_type) {
                        case SCM_RIGHTS:
                            newfd = *(int *)CMSG_DATA(cmp);
                            break;
                        case SCM_CREDTYPE:
                            credp = (struct CREDSTRUCT *)CMSG_DATA(cmp);
                            *uidptr = credp->CR_UID;
                        }
                    }
                } else {
                    newfd = -status;
                }
                nr -= 2;
            }
        }
        if (nr > 0 && (*userfunc)(STDERR_FILENO, buf, nr) != nr)
            return(-1);
        if (status >= 0)    /* final data has arrived */
            return(newfd);  /* descriptor, or -status */
    }
}
```

Figure 17.16 Receiving credentials over UNIX domain sockets

On FreeBSD, we specify `SCM_CREDS` to transmit credentials; on Linux, we use `SCM_CREDENTIALS`.

17.5 An Open Server, Version 1

Using file descriptor passing, we now develop an open server—a program that is executed by a process to open one or more files. Instead of sending the contents of the file back to the calling process, however, this server sends back an open file descriptor. As a result, the open server can work with any type of file (such as a device or a socket) and not simply regular files. The client and server exchange a minimum amount of information using IPC: the filename and open mode sent by the client, and the descriptor returned by the server. The contents of the file are not exchanged using IPC.

There are several advantages in designing the server to be a separate executable program (either one that is executed by the client, as we develop in this section, or a daemon server, which we develop in the next section).

- The server can easily be contacted by any client, similar to the client calling a library function. We are not hard-coding a particular service into the application, but designing a general facility that others can reuse.
- If we need to change the server, only a single program is affected. Conversely, updating a library function can require that all programs that call the function be updated (i.e., relinked with the link editor). Shared libraries can simplify this updating (Section 7.7).
- The server can be a set-user-ID program, providing it with additional permissions that the client does not have. Note that a library function (or shared library function) can't provide this capability.

The client process creates an fd-pipe and then calls `fork` and `exec` to invoke the server. The client sends requests across the fd-pipe using one end, and the server sends back responses over the fd-pipe using the other end.

We define the following application protocol between the client and the server.

1. The client sends a request of the form "`open` <*pathname*> <*openmode*>\0" across the fd-pipe to the server. The <*openmode*> is the numeric value, in ASCII decimal, of the second argument to the `open` function. This request string is terminated by a null byte.
2. The server sends back an open descriptor or an error by calling either `send_fd` or `send_err`.

This is an example of a process sending an open descriptor to its parent. In Section 17.6, we'll modify this example to use a single daemon server, where the server sends a descriptor to a completely unrelated process.

We first have the header, `open.h` (Figure 17.17), which includes the standard headers and defines the function prototypes.

```
#include "apue.h"
#include <errno.h>

#define CL_OPEN "open"          /* client's request for server */

int     csopen(char *, int);
```

Figure 17.17 The `open.h` header

The `main` function (Figure 17.18) is a loop that reads a pathname from standard input and copies the file to standard output. The function calls `csopen` to contact the open server and return an open descriptor.

```
#include    "open.h"
#include    <fcntl.h>

#define BUFFSIZE    8192

int
main(int argc, char *argv[])
{
    int     n, fd;
    char    buf[BUFFSIZE];
    char    line[MAXLINE];

    /* read filename to cat from stdin */
    while (fgets(line, MAXLINE, stdin) != NULL) {
        if (line[strlen(line) - 1] == '\n')
            line[strlen(line) - 1] = 0; /* replace newline with null */

        /* open the file */
        if ((fd = csopen(line, O_RDONLY)) < 0)
            continue;   /* csopen() prints error from server */

        /* and cat to stdout */
        while ((n = read(fd, buf, BUFFSIZE)) > 0)
            if (write(STDOUT_FILENO, buf, n) != n)
                err_sys("write error");
        if (n < 0)
            err_sys("read error");
        close(fd);
    }

    exit(0);
}
```

Figure 17.18 The client `main` function, version 1

The function `csopen` (Figure 17.19) does the `fork` and `exec` of the server, after creating the fd-pipe.

```
#include    "open.h"
#include    <sys/uio.h>     /* struct iovec */

/*
 * Open the file by sending the "name" and "oflag" to the
 * connection server and reading a file descriptor back.
 */
int
csopen(char *name, int oflag)
{
    pid_t           pid;
    int             len;
```

```
    char            buf[10];
    struct iovec    iov[3];
    static int      fd[2] = { -1, -1 };

    if (fd[0] < 0) {    /* fork/exec our open server first time */
        if (fd_pipe(fd) < 0) {
            err_ret("fd_pipe error");
            return(-1);
        }
        if ((pid = fork()) < 0) {
            err_ret("fork error");
            return(-1);
        } else if (pid == 0) {      /* child */
            close(fd[0]);
            if (fd[1] != STDIN_FILENO &&
              dup2(fd[1], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            if (fd[1] != STDOUT_FILENO &&
              dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
                err_sys("dup2 error to stdout");
            if (execl("./opend", "opend", (char *)0) < 0)
                err_sys("execl error");
        }
        close(fd[1]);               /* parent */
    }
    sprintf(buf, " %d", oflag);     /* oflag to ascii */
    iov[0].iov_base = CL_OPEN " ";      /* string concatenation */
    iov[0].iov_len  = strlen(CL_OPEN) + 1;
    iov[1].iov_base = name;
    iov[1].iov_len  = strlen(name);
    iov[2].iov_base = buf;
    iov[2].iov_len  = strlen(buf) + 1;  /* +1 for null at end of buf */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
    if (writev(fd[0], &iov[0], 3) != len) {
        err_ret("writev error");
        return(-1);
    }

    /* read descriptor, returned errors handled by write() */
    return(recv_fd(fd[0], write));
}
```

Figure 17.19 The csopen function, version 1

The child closes one end of the fd-pipe, and the parent closes the other. For the server that it executes, the child also duplicates its end of the fd-pipe onto its standard input and standard output. (Another option would have been to pass the ASCII representation of the descriptor `fd[1]` as an argument to the server.)

The parent sends to the server the request containing the pathname and open mode. Finally, the parent calls `recv_fd` to return either the descriptor or an error. If an error is returned by the server, `write` is called to output the message to standard error.

Now let's look at the open server. It is the program `opend` that is executed by the client in Figure 17.19. First, we have the `opend.h` header (Figure 17.20), which includes the standard headers and declares the global variables and function prototypes.

```
#include "apue.h"
#include <errno.h>

#define CL_OPEN "open"          /* client's request for server */

extern char  errmsg[];  /* error message string to return to client */
extern int   oflag;     /* open() flag: O_xxx ... */
extern char *pathname;  /* of file to open() for client */

int      cli_args(int, char **);
void     handle_request(char *, int, int);
```

Figure 17.20 The `opend.h` header, version 1

The `main` function (Figure 17.21) reads the requests from the client on the fd-pipe (its standard input) and calls the function `handle_request`.

```
#include    "opend.h"

char     errmsg[MAXLINE];
int      oflag;
char    *pathname;

int
main(void)
{
    int     nread;
    char    buf[MAXLINE];

    for ( ; ; ) {   /* read arg buffer from client, process request */
        if ((nread = read(STDIN_FILENO, buf, MAXLINE)) < 0)
            err_sys("read error on stream pipe");
        else if (nread == 0)
            break;      /* client has closed the stream pipe */
        handle_request(buf, nread, STDOUT_FILENO);
    }
    exit(0);
}
```

Figure 17.21 The server `main` function, version 1

The function `handle_request` in Figure 17.22 does all the work. It calls the function `buf_args` to break up the client's request into a standard `argv`-style argument list and calls the function `cli_args` to process the client's arguments. If all is OK, `open` is called to open the file, and then `send_fd` sends the descriptor back to the client across the fd-pipe (its standard output). If an error is encountered, `send_err` is called to send back an error message, using the client–server protocol that we described earlier.

```
#include    "opend.h"
#include    <fcntl.h>

void
handle_request(char *buf, int nread, int fd)
{
    int     newfd;

    if (buf[nread-1] != 0) {
        snprintf(errmsg, MAXLINE-1,
          "request not null terminated: %*.*s\n", nread, nread, buf);
        send_err(fd, -1, errmsg);
        return;
    }
    if (buf_args(buf, cli_args) < 0) {  /* parse args & set options */
        send_err(fd, -1, errmsg);
        return;
    }
    if ((newfd = open(pathname, oflag)) < 0) {
        snprintf(errmsg, MAXLINE-1, "can't open %s: %s\n", pathname,
          strerror(errno));
        send_err(fd, -1, errmsg);
        return;
    }
    if (send_fd(fd, newfd) < 0)     /* send the descriptor */
        err_sys("send_fd error");
    close(newfd);       /* we're done with descriptor */
}
```

Figure 17.22 The `handle_request` function, version 1

The client's request is a null-terminated string of white-space-separated arguments. The function `buf_args` in Figure 17.23 breaks this string into a standard `argv`-style argument list and calls a user function to process the arguments. We use the ISO C function `strtok` to tokenize the string into separate arguments.

```
#include "apue.h"

#define MAXARGC     50  /* max number of arguments in buf */
#define WHITE   " \t\n" /* white space for tokenizing arguments */

/*
 * buf[] contains white-space-separated arguments.  We convert it to an
 * argv-style array of pointers, and call the user's function (optfunc)
 * to process the array.  We return -1 if there's a problem parsing buf,
 * else we return whatever optfunc() returns.  Note that user's buf[]
 * array is modified (nulls placed after each token).
 */
int
buf_args(char *buf, int (*optfunc)(int, char **))
{
```

```
    char    *ptr, *argv[MAXARGC];
    int     argc;

    if (strtok(buf, WHITE) == NULL)     /* an argv[0] is required */
        return(-1);
    argv[argc = 0] = buf;
    while ((ptr = strtok(NULL, WHITE)) != NULL) {
        if (++argc >= MAXARGC-1)    /* -1 for room for NULL at end */
            return(-1);
        argv[argc] = ptr;
    }
    argv[++argc] = NULL;

    /*
     * Since argv[] pointers point into the user's buf[],
     * user's function can just copy the pointers, even
     * though argv[] array will disappear on return.
     */
    return((*optfunc)(argc, argv));
}
```

Figure 17.23 The `buf_args` function

The server's function that is called by `buf_args` is `cli_args` (Figure 17.24). It verifies that the client sent the right number of arguments and stores the pathname and open mode in global variables.

```
#include    "opend.h"

/*
 * This function is called by buf_args(), which is called by
 * handle_request().  buf_args() has broken up the client's
 * buffer into an argv[]-style array, which we now process.
 */
int
cli_args(int argc, char **argv)
{
    if (argc != 3 || strcmp(argv[0], CL_OPEN) != 0) {
        strcpy(errmsg, "usage: <pathname> <oflag>\n");
        return(-1);
    }
    pathname = argv[1];     /* save ptr to pathname to open */
    oflag = atoi(argv[2]);
    return(0);
}
```

Figure 17.24 The `cli_args` function

This completes the open server that is invoked by a `fork` and `exec` from the client. A single fd-pipe is created before the `fork` and is used to communicate between the client and the server. With this arrangement, we have one server per client.

17.6 An Open Server, Version 2

In the previous section, we developed an open server that was invoked by a `fork` and `exec` by the client, demonstrating how we can pass file descriptors from a child to a parent. In this section, we develop an open server as a daemon process. One server handles all clients. We expect this design to be more efficient, since a `fork` and an `exec` are avoided. We use a UNIX domain socket connection between the client and the server and demonstrate passing file descriptors between unrelated processes. We'll use the three functions `serv_listen`, `serv_accept`, and `cli_conn` introduced in Section 17.3. This server also demonstrates how a single server can handle multiple clients, using both the `select` and `poll` functions from Section 14.4.

This version of the client is similar to the client from Section 17.5. Indeed, the file `main.c` is identical (Figure 17.18). We add the following line to the `open.h` header (Figure 17.17):

```
#define CS_OPEN "/tmp/opend.socket"  /* server's well-known name */
```

The file `open.c` does change from Figure 17.19, since we now call `cli_conn` instead of doing the `fork` and `exec`. This is shown in Figure 17.25.

```
#include    "open.h"
#include    <sys/uio.h>     /* struct iovec */

/*
 * Open the file by sending the "name" and "oflag" to the
 * connection server and reading a file descriptor back.
 */
int
csopen(char *name, int oflag)
{
    int             len;
    char            buf[12];
    struct iovec    iov[3];
    static int      csfd = -1;

    if (csfd < 0) {     /* open connection to conn server */
        if ((csfd = cli_conn(CS_OPEN)) < 0) {
            err_ret("cli_conn error");
            return(-1);
        }
    }

    sprintf(buf, " %d", oflag);     /* oflag to ascii */
    iov[0].iov_base = CL_OPEN " ";  /* string concatenation */
    iov[0].iov_len  = strlen(CL_OPEN) + 1;
    iov[1].iov_base = name;
    iov[1].iov_len  = strlen(name);
    iov[2].iov_base = buf;
    iov[2].iov_len  = strlen(buf) + 1;  /* null always sent */
    len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
```

```
    if (writev(csfd, &iov[0], 3) != len) {
        err_ret("writev error");
        return(-1);
    }

    /* read back descriptor; returned errors handled by write() */
    return(recv_fd(csfd, write));
}
```

Figure 17.25 The csopen function, version 2

The protocol from the client to the server remains the same.

Next, we'll look at the server. The header `opend.h` (Figure 17.26) includes the standard headers and declares the global variables and the function prototypes.

```
#include "apue.h"
#include <errno.h>

#define CS_OPEN "/tmp/opend.socket" /* well-known name */
#define CL_OPEN "open"              /* client's request for server */

extern int   debug;     /* nonzero if interactive (not daemon) */
extern char  errmsg[];  /* error message string to return to client */
extern int   oflag;     /* open flag: O_xxx ... */
extern char *pathname;  /* of file to open for client */

typedef struct {    /* one Client struct per connected client */
  int   fd;         /* fd, or -1 if available */
  uid_t uid;
} Client;

extern Client   *client;        /* ptr to malloc'ed array */
extern int       client_size;   /* # entries in client[] array */

int      cli_args(int, char **);
int      client_add(int, uid_t);
void     client_del(int);
void     loop(void);
void     handle_request(char *, int, int, uid_t);
```

Figure 17.26 The opend.h header, version 2

Since this server handles all clients, it must maintain the state of each client connection. This is done with the `client` array declared in the `opend.h` header. Figure 17.27 defines three functions that manipulate this array.

```
#include    "opend.h"

#define NALLOC  10      /* # client structs to alloc/realloc for */

static void
client_alloc(void)      /* alloc more entries in the client[] array */
```

```
{
    int     i;

    if (client == NULL)
        client = malloc(NALLOC * sizeof(Client));
    else
        client = realloc(client, (client_size+NALLOC)*sizeof(Client));
    if (client == NULL)
        err_sys("can't alloc for client array");

    /* initialize the new entries */
    for (i = client_size; i < client_size + NALLOC; i++)
        client[i].fd = -1;  /* fd of -1 means entry available */

    client_size += NALLOC;
}

/*
 * Called by loop() when connection request from a new client arrives.
 */
int
client_add(int fd, uid_t uid)
{
    int     i;

    if (client == NULL)     /* first time we're called */
        client_alloc();
again:
    for (i = 0; i < client_size; i++) {
        if (client[i].fd == -1) {   /* find an available entry */
            client[i].fd = fd;
            client[i].uid = uid;
            return(i);  /* return index in client[] array */
        }
    }

    /* client array full, time to realloc for more */
    client_alloc();
    goto again;     /* and search again (will work this time) */
}

/*
 * Called by loop() when we're done with a client.
 */
void
client_del(int fd)
{
    int     i;

    for (i = 0; i < client_size; i++) {
        if (client[i].fd == fd) {
            client[i].fd = -1;
```

```
            return;
        }
    }
    log_quit("can't find client entry for fd %d", fd);
}
```

Figure 17.27 Functions to manipulate `client` array

The first time `client_add` is called, it calls `client_alloc`, which in turn calls `malloc` to allocate space for ten entries in the array. After these ten entries are all in use, a later call to `client_add` causes `realloc` to allocate additional space. By dynamically allocating space this way, we have not limited the size of the `client` array at compile time to some value that we guessed and put into a header. These functions call the `log_` functions (Appendix B) if an error occurs, since we assume that the server is a daemon.

Normally the server will run as a daemon, but we want to provide an option that allows it to be run in the foreground, with diagnostic messages sent to the standard error. This should make the server easier to test and debug, especially if we don't have permission to read the log file where the diagnostic messages are normally written. We'll use a command-line option to control whether the server runs in the foreground or as a daemon in the background.

It is important that all commands on a system follow the same conventions, because this makes them easier to use. If someone is familiar with the way command-line options are formed with one command, it would create more chances for mistakes if another command followed different conventions.

This problem is sometimes visible when dealing with white space on the command line. Some commands require that an option be separated from its argument by white space, but other commands require the argument to follow immediately after its option, without any intervening spaces. Without a consistent set of rules to follow, users either have to memorize the syntax of all commands or resort to a trial-and-error process when invoking them.

The Single UNIX Specification includes a set of conventions and guidelines that promote consistent command-line syntax. They include such suggestions as "Restrict each command-line option to a single alphanumeric character" and "All options should be preceded by a – character."

Luckily, the `getopt` function exists to help command developers process command-line options in a consistent manner.

```
#include <unistd.h>

int getopt(int argc, char * const argv[], const char *options);

extern int optind, opterr, optopt;
extern char *optarg;
```

Returns: the next option character, or
–1 when all options have been processed

The *argc* and *argv* arguments are the same ones passed to the `main` function of the program. The *options* argument is a string containing the option characters supported by the command. If an option character is followed by a colon, then the option takes an argument. Otherwise, the option exists by itself. For example, if the usage statement for a command was

```
command [-i] [-u username] [-z] filename
```

we would pass `"iu:z"` as the *options* string to `getopt`.

The `getopt` function is normally used in a loop that terminates when `getopt` returns –1. During each iteration of the loop, `getopt` will return the next option processed. It is up to the application to sort out any conflict in options, however; `getopt` simply parses the options and enforces a standard format.

When it encounters an invalid option, `getopt` returns a question mark instead of the character. If an option's argument is missing, `getopt` will also return a question mark, but if the first character in the options string is a colon, `getopt` returns a colon instead. The special pattern `--` will cause `getopt` to stop processing options and return –1. This allows users to provide command arguments that start with a minus sign but aren't options. For example, if you have a file named `-bar`, you can't remove it by typing

```
rm -bar
```

because `rm` will try to interpret `-bar` as options. The way to remove the file is to type

```
rm -- -bar
```

The `getopt` function supports four external variables.

`optarg` If an option takes an argument, `getopt` sets `optarg` to point to the option's argument string when an option is processed.

`opterr` If an option error is encountered, `getopt` will print an error message by default. To disable this behavior, applications can set `opterr` to 0.

`optind` The index in the `argv` array of the next string to be processed. It starts at 1 and is incremented for each argument processed by `getopt`.

`optopt` If an error is encountered during options processing, `getopt` will set `optopt` to point to the option string that caused the error.

The open server's `main` function (Figure 17.28) defines the global variables, processes the command-line options, and calls the function `loop`. If we invoke the server with the `-d` option, the server runs interactively instead of as a daemon. This option is used when testing the server.

```
#include    "opend.h"
#include    <syslog.h>

int      debug, oflag, client_size, log_to_stderr;
char     errmsg[MAXLINE];
char    *pathname;
```

```
Client  *client = NULL;

int
main(int argc, char *argv[])
{
    int     c;

    log_open("open.serv", LOG_PID, LOG_USER);

    opterr = 0;     /* don't want getopt() writing to stderr */
    while ((c = getopt(argc, argv, "d")) != EOF) {
        switch (c) {
        case 'd':       /* debug */
            debug = log_to_stderr = 1;
            break;

        case '?':
            err_quit("unrecognized option: -%c", optopt);
        }
    }

    if (debug == 0)
        daemonize("opend");

    loop();     /* never returns */
}
```

Figure 17.28 The server `main` function, version 2

The function `loop` is the server's infinite loop. We'll show two versions of this function. Figure 17.29 shows one version that uses `select`; Figure 17.30 shows another version that uses `poll`.

```
#include    "opend.h"
#include    <sys/select.h>

void
loop(void)
{
    int     i, n, maxfd, maxi, listenfd, clifd, nread;
    char    buf[MAXLINE];
    uid_t   uid;
    fd_set  rset, allset;

    FD_ZERO(&allset);

    /* obtain fd to listen for client requests on */
    if ((listenfd = serv_listen(CS_OPEN)) < 0)
        log_sys("serv_listen error");
    FD_SET(listenfd, &allset);
    maxfd = listenfd;
    maxi = -1;
```

```
    for ( ; ; ) {
        rset = allset;  /* rset gets modified each time around */
        if ((n = select(maxfd + 1, &rset, NULL, NULL, NULL)) < 0)
            log_sys("select error");

        if (FD_ISSET(listenfd, &rset)) {
            /* accept new client request */
            if ((clifd = serv_accept(listenfd, &uid)) < 0)
                log_sys("serv_accept error: %d", clifd);
            i = client_add(clifd, uid);
            FD_SET(clifd, &allset);
            if (clifd > maxfd)
                maxfd = clifd;  /* max fd for select() */
            if (i > maxi)
                maxi = i;   /* max index in client[] array */
            log_msg("new connection: uid %d, fd %d", uid, clifd);
            continue;
        }

        for (i = 0; i <= maxi; i++) {   /* go through client[] array */
            if ((clifd = client[i].fd) < 0)
                continue;
            if (FD_ISSET(clifd, &rset)) {
                /* read argument buffer from client */
                if ((nread = read(clifd, buf, MAXLINE)) < 0) {
                    log_sys("read error on fd %d", clifd);
                } else if (nread == 0) {
                    log_msg("closed: uid %d, fd %d",
                      client[i].uid, clifd);
                    client_del(clifd);  /* client has closed cxn */
                    FD_CLR(clifd, &allset);
                    close(clifd);
                } else {    /* process client's request */
                    handle_request(buf, nread, clifd, client[i].uid);
                }
            }
        }
    }
}
```

Figure 17.29 The `loop` function using `select`

This function calls `serv_listen` (Figure 17.8) to create the server's endpoint for the client connections. The remainder of the function is a loop that starts with a call to `select`. Two conditions can be true after `select` returns.

1. The descriptor `listenfd` can be ready for reading, which means that a new client has called `cli_conn`. To handle this, we call `serv_accept` (Figure 17.9) and then update the `client` array and associated bookkeeping information for the new client. (We keep track of the highest descriptor number for the first

argument to `select`. We also keep track of the highest index in use in the `client` array.)

2. An existing client's connection can be ready for reading. This means that the client has either terminated or sent a new request. We find out about a client termination by `read` returning 0 (end of file). If `read` returns a value greater than 0, there is a new request to process, which we handle by calling `handle_request`.

We keep track of which descriptors are currently in use in the `allset` descriptor set. As new clients connect to the server, the appropriate bit is turned on in this descriptor set. The appropriate bit is turned off when the client terminates.

We always know when a client terminates, whether the termination is voluntary or not, since all the client's descriptors (including the connection to the server) are automatically closed by the kernel. This differs from the XSI IPC mechanisms.

The `loop` function that uses `poll` is shown in Figure 17.30.

```
#include    "opend.h"
#include    <poll.h>

#define NALLOC  10  /* # pollfd structs to alloc/realloc */

static struct pollfd *
grow_pollfd(struct pollfd *pfd, int *maxfd)
{
    int             i;
    int             oldmax = *maxfd;
    int             newmax = oldmax + NALLOC;

    if ((pfd = realloc(pfd, newmax * sizeof(struct pollfd))) == NULL)
        err_sys("realloc error");
    for (i = oldmax; i < newmax; i++) {
        pfd[i].fd = -1;
        pfd[i].events = POLLIN;
        pfd[i].revents = 0;
    }
    *maxfd = newmax;
    return(pfd);
}

void
loop(void)
{
    int             i, listenfd, clifd, nread;
    char            buf[MAXLINE];
    uid_t           uid;
    struct pollfd   *pollfd;
    int             numfd = 1;
    int             maxfd = NALLOC;

    if ((pollfd = malloc(NALLOC * sizeof(struct pollfd))) == NULL)
        err_sys("malloc error");
```

```
    for (i = 0; i < NALLOC; i++) {
        pollfd[i].fd = -1;
        pollfd[i].events = POLLIN;
        pollfd[i].revents = 0;
    }

    /* obtain fd to listen for client requests on */
    if ((listenfd = serv_listen(CS_OPEN)) < 0)
        log_sys("serv_listen error");
    client_add(listenfd, 0);    /* we use [0] for listenfd */
    pollfd[0].fd = listenfd;

    for ( ; ; ) {
        if (poll(pollfd, numfd, -1) < 0)
            log_sys("poll error");

        if (pollfd[0].revents & POLLIN) {
            /* accept new client request */
            if ((clifd = serv_accept(listenfd, &uid)) < 0)
                log_sys("serv_accept error: %d", clifd);
            client_add(clifd, uid);

            /* possibly increase the size of the pollfd array */
            if (numfd == maxfd)
                pollfd = grow_pollfd(pollfd, &maxfd);
            pollfd[numfd].fd = clifd;
            pollfd[numfd].events = POLLIN;
            pollfd[numfd].revents = 0;
            numfd++;
            log_msg("new connection: uid %d, fd %d", uid, clifd);
        }

        for (i = 1; i < numfd; i++) {
            if (pollfd[i].revents & POLLHUP) {
                goto hungup;
            } else if (pollfd[i].revents & POLLIN) {
                /* read argument buffer from client */
                if ((nread = read(pollfd[i].fd, buf, MAXLINE)) < 0) {
                    log_sys("read error on fd %d", pollfd[i].fd);
                } else if (nread == 0) {
hungup:
                    /* the client closed the connection */
                    log_msg("closed: uid %d, fd %d",
                      client[i].uid, pollfd[i].fd);
                    client_del(pollfd[i].fd);
                    close(pollfd[i].fd);
                    if (i < (numfd-1)) {
                        /* pack the array */
                        pollfd[i].fd = pollfd[numfd-1].fd;
                        pollfd[i].events = pollfd[numfd-1].events;
                        pollfd[i].revents = pollfd[numfd-1].revents;
                        i--;    /* recheck this entry */
```

```
                    }
                    numfd--;
                } else {        /* process client's request */
                    handle_request(buf, nread, pollfd[i].fd,
                      client[i].uid);
                }
            }
        }
    }
}
```

Figure 17.30 The `loop` function using `poll`

To allow for as many clients as there are possible open descriptors, we dynamically allocate space for the array of `pollfd` structures using the same strategy as used in the `client_alloc` function for the `client` array (see Figure 17.27).

We use the first entry (index 0) of the `pollfd` array for the `listenfd` descriptor. The arrival of a new client connection is indicated by a `POLLIN` on the `listenfd` descriptor. As before, we call `serv_accept` to accept the connection.

For an existing client, we have to handle two different events from `poll`: a client termination is indicated by `POLLHUP`, and a new request from an existing client is indicated by `POLLIN`. The client can close its end of the connection while there is still data to be read from the server's end of the connection. Even though the endpoint is marked as hung up, the server can read all the data queued on its end. But with this server, when we receive the hangup from the client, we can `close` the connection to the client, effectively throwing away any queued data. There is no reason to process any requests still remaining, since we can't send any responses back.

As with the `select` version of this function, new requests from a client are handled by calling the `handle_request` function (Figure 17.31). This function is similar to the earlier version (Figure 17.22). It calls the same function, `buf_args` (Figure 17.23), that calls `cli_args` (Figure 17.24), but since it runs from a daemon process, it logs error messages instead of printing them on the standard error.

```
#include    "opend.h"
#include    <fcntl.h>

void
handle_request(char *buf, int nread, int clifd, uid_t uid)
{
    int     newfd;

    if (buf[nread-1] != 0) {
        snprintf(errmsg, MAXLINE-1,
          "request from uid %d not null terminated: %*.*s\n",
          uid, nread, nread, buf);
        send_err(clifd, -1, errmsg);
        return;
    }
    log_msg("request: %s, from uid %d", buf, uid);
```

```
    /* parse the arguments, set options */
    if (buf_args(buf, cli_args) < 0) {
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

    if ((newfd = open(pathname, oflag)) < 0) {
        snprintf(errmsg, MAXLINE-1, "can't open %s: %s\n",
          pathname, strerror(errno));
        send_err(clifd, -1, errmsg);
        log_msg(errmsg);
        return;
    }

    /* send the descriptor */
    if (send_fd(clifd, newfd) < 0)
        log_sys("send_fd error");
    log_msg("sent fd %d over fd %d for %s", newfd, clifd, pathname);
    close(newfd);       /* we're done with descriptor */
}
```

Figure 17.31 The `request` function, version 2

This completes the second version of the open server, which uses a single daemon to handle all the client requests.

17.7 Summary

The key points in this chapter are the ability to pass file descriptors between processes and the ability of a server to accept unique connections from clients. Although all platforms provide support for UNIX domain sockets (refer back to Figure 15.1), we've seen that there are differences in each implementation, which makes it more difficult for us to develop portable applications.

We used UNIX domain sockets throughout this chapter. We saw how to use them to implement a full-duplex pipe and how they can be used to adapt the I/O multiplexing functions from Section 14.4 to work indirectly with XSI message queues.

We presented two versions of an open server. One version was invoked directly by the client, using `fork` and `exec`. The second was a daemon server that handled all client requests. Both versions used the file descriptor passing and receiving functions.

We also saw how to use the `getopt` function to enforce consistent command-line processing for our programs. The final version of the open server used the `getopt` function, the client–server connection functions introduced in Section 17.3, and the I/O multiplexing functions from Section 14.4.

Exercises

17.1 We chose to use UNIX domain datagram sockets in Figure 17.3, because they retain message boundaries. Describe the changes that would be necessary to use regular pipes instead. How can we avoid copying the messages two extra times?

17.2 Write the following program using the file descriptor passing functions from this chapter and the parent–child synchronization routines from Section 8.9. The program calls `fork`, and the child `open`s an existing file and passes the open descriptor to the parent. The child then positions the file using `lseek` and notifies the parent. The parent reads the file's current offset and prints it for verification. If the file was passed from the child to the parent as we described, they should be sharing the same file table entry, so each time the child changes the file's current offset, that change should also affect the parent's descriptor. Have the child position the file to a different offset and notify the parent again.

17.3 In Figures 17.20 and 17.21, we differentiated between declaring and defining the global variables. What is the difference?

17.4 Recode the `buf_args` function (Figure 17.23), removing the compile-time limit on the size of the `argv` array. Use dynamic memory allocation.

17.5 Describe ways to optimize the function `loop` in Figure 17.29 and Figure 17.30. Implement your optimizations.

17.6 In the `serv_listen` function (Figure 17.8), we unlink the name of the file representing the UNIX domain socket if the file already exists. To avoid unintentionally removing a file that isn't a socket, we could call `stat` first to verify the file type. Explain the two problems with this approach.

17.7 Describe two possible ways to pass more than one file descriptor with a single call to `sendmsg`. Try them out to see if they are supported by your operating system.

18

Terminal I/O

18.1 Introduction

The handling of terminal I/O is a messy area, regardless of the operating system. The UNIX System is no exception. The manual page for terminal I/O is usually one of the longest in most editions of the programmer's manuals.

With the UNIX System, a schism formed in the late 1970s when System III developed a different set of terminal routines from those of Version 7. The System III style of terminal I/O continued through System V, and the Version 7 style became the standard for the BSD-derived systems. As with signals, this difference between the two worlds has been conquered by POSIX.1. In this chapter, we look at all the POSIX.1 terminal functions and some of the platform-specific additions.

Part of the complexity of the terminal I/O system occurs because people use terminal I/O for so many different things: terminals, hard-wired lines between computers, modems, printers, and so on.

18.2 Overview

Terminal I/O has two modes:

1. Canonical mode input processing. In this mode, terminal input is processed as lines. The terminal driver returns at most one line per read request.
2. Noncanonical mode input processing. The input characters are not assembled into lines.

If we don't do anything special, canonical mode is the default. For example, if the shell redirects standard input to the terminal and we use `read` and `write` to copy standard input to standard output, the terminal is in canonical mode, and each `read` returns at most one line. Programs that manipulate the entire screen, such as the `vi` editor, use noncanonical mode, since the commands may be single characters and are not terminated by newlines. Also, this editor doesn't want processing by the system of the special characters, since they may overlap with the editor commands. For example, the Control-D character is often the end-of-file character for the terminal, but it's also a `vi` command to scroll down one-half screen.

> The Version 7 and older BSD-style terminal drivers supported three modes for terminal input: (a) cooked mode (the input is collected into lines, and the special characters are processed), (b) raw mode (the input is not assembled into lines, and there is no processing of special characters), and (c) cbreak mode (the input is not assembled into lines, but some of the special characters are processed). Figure 18.20 shows a POSIX.1 function that places a terminal in cbreak or raw mode.

POSIX.1 defines 11 special input characters, 9 of which we can change. We've been using some of these throughout the text—the end-of-file character (usually Control-D) and the suspend character (usually Control-Z), for example. Section 18.3 describes each of these characters.

We can think of a terminal device as being controlled by a terminal driver, usually within the kernel. Each terminal device has an input queue and an output queue, shown in Figure 18.1.

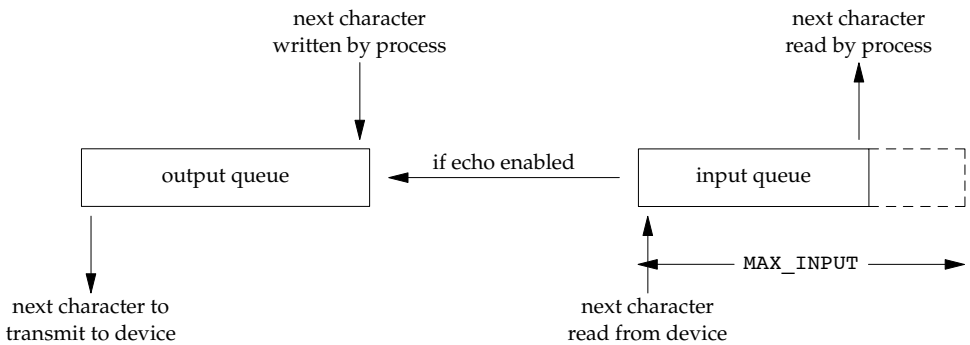


Figure 18.1 Logical picture of input and output queues for a terminal device

There are several points to consider from this picture.

- If echoing is enabled, there is an implied link between the input queue and the output queue.
- The size of the input queue, `MAX_INPUT` (see Figure 2.12), is finite. When the input queue for a particular device fills, the system behavior is implementation dependent. Most UNIX systems echo the bell character when this happens.

- There is another input limit, `MAX_CANON`, that we don't show here. This limit is the maximum number of bytes in a canonical input line.
- Although the size of the output queue is finite, no constants defining that size are accessible to the program, because when the output queue starts to fill up, the kernel simply puts the writing process to sleep until room is available.
- We'll see how the `tcflush` flush function allows us to flush either the input queue or the output queue. Similarly, when we describe the `tcsetattr` function, we'll see how we can tell the system to change the attributes of a terminal device only after the output queue is empty. (We want to do this, for example, if we're changing the output attributes.) We can also tell the system to discard everything in the input queue when changing the terminal attributes. (We want to do this if we're changing the input attributes or changing between canonical and noncanonical modes, so that previously entered characters aren't interpreted in the wrong mode.)

Most UNIX systems implement all the canonical processing in a module called the *terminal line discipline*. We can think of this module as a box that sits between the kernel's generic read and write functions and the actual device driver (see Figure 18.2).

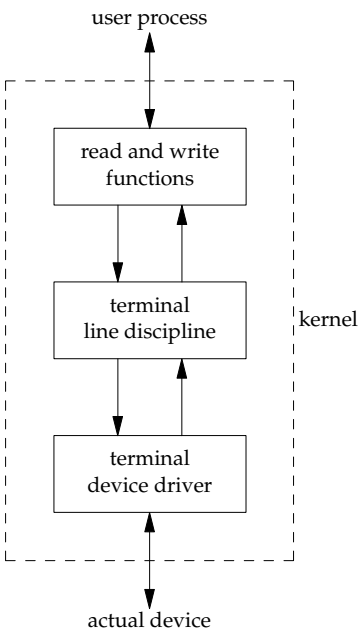


Figure 18.2 Terminal line discipline

By isolating the canonical processing in a separate module, all terminal drivers can support canonical processing consistently. We'll return to this picture when we discuss pseudo terminals in Chapter 19.

All the terminal device characteristics that we can examine and change are contained in a `termios` structure. This structure is defined in the header `<termios.h>`, which we use throughout this chapter:

```
struct termios {
  tcflag_t  c_iflag;    /* input flags */
  tcflag_t  c_oflag;    /* output flags */
  tcflag_t  c_cflag;    /* control flags */
  tcflag_t  c_lflag;    /* local flags */
  cc_t      c_cc[NCCS]; /* control characters */
};
```

Roughly speaking, the input flags control the input of characters by the terminal device driver (e.g., strip eighth bit on input, enable input parity checking), the output flags control the driver output (e.g., perform output processing, map newline to CR/LF), the control flags affect the RS-232 serial lines (e.g., ignore modem status lines, one or two stop bits per character), and the local flags affect the interface between the driver and the user (e.g., echo on or off, visually erase characters, enable terminal-generated signals, job control stop signal for background output).

The type `tcflag_t` is big enough to hold each of the flag values and is often defined as an `unsigned int` or an `unsigned long`. The `c_cc` array contains all the special characters that we can change. `NCCS` is the number of elements in this array and is typically between 15 and 20 (since most implementations of the UNIX System support more than the 11 POSIX-defined special characters). The `cc_t` type is large enough to hold each special character and is typically an `unsigned char`.

> Versions of System V that predated the POSIX standard had a header named `<termio.h>` and a structure named `termio`. POSIX.1 added an `s` to the names, to differentiate them from their predecessors.

Figures 18.3 through 18.6 list all the terminal flags that we can change to affect the characteristics of a terminal device. Note that even though the Single UNIX Specification defines a common subset that all platforms start from, all the implementations have their own additions. Most of these additions come from the historical differences between the systems. We'll discuss each of these flag values in detail in Section 18.5.

Given all the options available, how do we examine and change these characteristics of a terminal device? Figure 18.7 summarizes the various functions defined by the Single UNIX Specification that operate on terminal devices. (All the functions listed are part of the base POSIX specification. We described `tcgetpgrp`, `tcgetsid`, and `tcsetpgrp` in Section 9.7.)

Note that the Single UNIX Specification doesn't use the classic `ioctl` on terminal devices. Instead, it uses the 13 functions shown in Figure 18.7. The reason is that the `ioctl` function for terminal devices uses a different data type for its final argument, which depends on the action being performed. This makes type checking of the arguments impossible.

Although only 13 functions operate on terminal devices, the first two functions in Figure 18.7 (`tcgetattr` and `tcsetattr`) manipulate almost 70 different flags (see

Figures 18.3 through 18.6). The handling of terminal devices is complicated by the large number of options available for terminal devices and the challenge of trying to determine which options are required for a particular device (be it a terminal, modem, printer, or whatever).

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
CBAUDEXT	extended baud rate					•
CCAR_OFLOW	DCD flow control of output		•		•	
CCTS_OFLOW	CTS flow control of output		•		•	•
CDSR_OFLOW	DSR flow control of output		•		•	
CDTR_IFLOW	DTR flow control of input		•		•	
CIBAUDEXT	extended input baud rate					•
CIGNORE	ignore control flags		•		•	
CLOCAL	ignore modem status lines	•	•	•	•	•
CMSPAR	mark or space parity			•		
CREAD	enable receiver	•	•	•	•	•
CRTSCTS	enable hardware flow control		•	•	•	•
CRTS_IFLOW	RTS flow control of input		•		•	•
CRTSXOFF	enable input hardware flow control					•
CSIZE	character size mask	•	•	•	•	•
CSTOPB	send two stop bits, else one	•	•	•	•	•
HUPCL	hang up on last close	•	•	•	•	•
MDMBUF	same as CCAR_OFLOW		•		•	
PARENB	parity enable	•	•	•	•	•
PAREXT	mark or space parity					•
PARODD	odd parity, else even	•	•	•	•	•

Figure 18.3 `c_cflag` terminal flags

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
BRKINT	generate SIGINT on BREAK	•	•	•	•	•
ICRNL	map CR to NL on input	•	•	•	•	•
IGNBRK	ignore BREAK condition	•	•	•	•	•
IGNCR	ignore CR	•	•	•	•	•
IGNPAR	ignore characters with parity errors	•	•	•	•	•
IMAXBEL	ring bell on input queue full		•	•	•	•
INLCR	map NL to CR on input	•	•	•	•	•
INPCK	enable input parity checking	•	•	•	•	•
ISTRIP	strip eighth bit off input characters	•	•	•	•	•
IUCLC	map uppercase to lowercase on input			•		•
IUTF8	input is UTF-8			•	•	
IXANY	enable any characters to restart output	•	•	•	•	•
IXOFF	enable start/stop input flow control	•	•	•	•	•
IXON	enable start/stop output flow control	•	•	•	•	•
PARMRK	mark parity errors	•	•	•	•	•

Figure 18.4 `c_iflag` terminal flags

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
ALTWERASE	use alternate WERASE algorithm		•		•	
ECHO	enable echo	•	•	•	•	•
ECHOCTL	echo control chars as ^(Char)		•	•	•	•
ECHOE	visually erase chars	•	•	•	•	•
ECHOK	echo kill	•	•	•	•	•
ECHOKE	visual erase for kill		•	•	•	•
ECHONL	echo NL	•	•	•	•	•
ECHOPRT	visual erase mode for hard copy		•	•	•	•
EXTPROC	external character processing		•	•	•	
FLUSHO	output being flushed		•	•	•	•
ICANON	canonical input	•	•	•	•	•
IEXTEN	enable extended input char processing	•	•	•	•	•
ISIG	enable terminal-generated signals	•	•	•	•	•
NOFLSH	disable flush after interrupt or quit	•	•	•	•	•
NOKERNINFO	no kernel output from STATUS		•		•	
PENDIN	retype pending input		•	•	•	•
TOSTOP	send SIGTTOU for background output	•	•	•	•	•
XCASE	canonical upper/lower presentation			•		•

Figure 18.5 `c_lflag` terminal flags

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
BSDLY	backspace delay mask	XSI		•		•
CRDLY	CR delay mask	XSI		•		•
FFDLY	form feed delay mask	XSI		•		•
NLDLY	NL delay mask	XSI		•		•
OCRNL	map CR to NL on output	XSI	•	•		•
OFDEL	fill is DEL, else NUL	XSI		•		•
OFILL	use fill character for delay	XSI		•		•
OLCUC	map lowercase to uppercase on output			•		•
ONLCR	map NL to CR-NL	XSI	•	•	•	•
ONLRET	NL performs CR function	XSI	•	•		•
ONOCR	no CR output at column 0	XSI	•	•		•
ONOEOT	discard EOTs (^D) on output		•		•	
OPOST	perform output processing	•	•	•	•	•
OXTABS	expand tabs to spaces		•		•	
TABDLY	horizontal tab delay mask	XSI	•	•		•
VTDLY	vertical tab delay mask	XSI		•		•

Figure 18.6 `c_oflag` terminal flags

Function	Description
`tcgetattr`	fetch attributes (`termios` structure)
`tcsetattr`	set attributes (`termios` structure)
`cfgetispeed`	get input speed
`cfgetospeed`	get output speed
`cfsetispeed`	set input speed
`cfsetospeed`	set output speed
`tcdrain`	wait for all output to be transmitted
`tcflow`	suspend transmit or receive
`tcflush`	flush pending input and/or output
`tcsendbreak`	send BREAK character
`tcgetpgrp`	get foreground process group ID
`tcsetpgrp`	set foreground process group ID
`tcgetsid`	get process group ID of session leader for controlling TTY

Figure 18.7 Summary of terminal I/O functions

The relationships among the 13 functions shown in Figure 18.7 are illustrated in Figure 18.8.

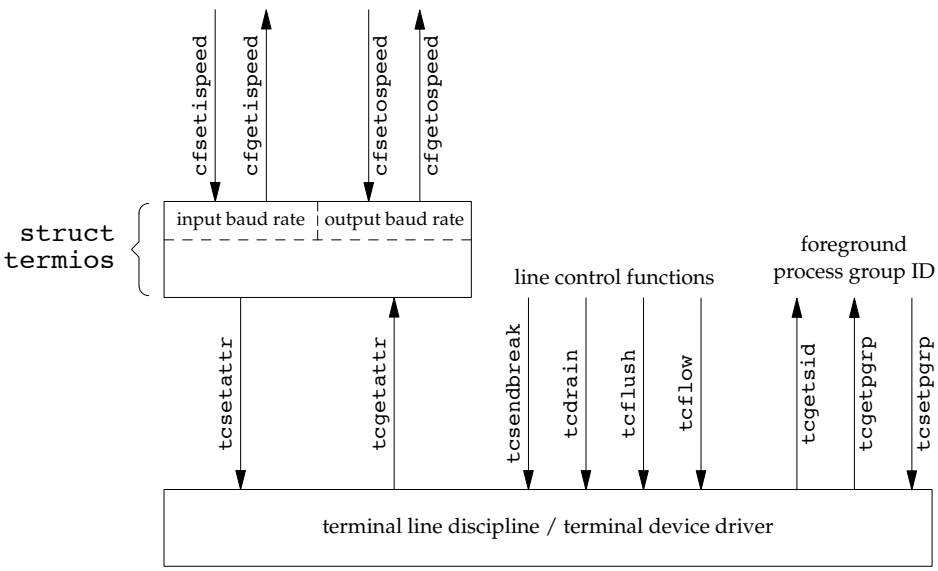


Figure 18.8 Relationships among the terminal-related functions

POSIX.1 doesn't specify where in the `termios` structure the baud rate information is stored; that is an implementation detail. Some systems, such as Solaris, store this information in the `c_cflag` field. Linux and BSD-derived systems, such as FreeBSD and Mac OS X, have two separate fields in the structure: one for the input speed and one for the output speed.

18.3 Special Input Characters

POSIX.1 defines 11 characters that are handled specially on input. Implementations define additional special characters. Figure 18.9 summarizes these special characters.

Character	Description	c_cc subscript	Enabled by field	Enabled by flag	Typical value	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
CR	carriage return	(can't change)	c_lflag	ICANON	\r	•	•	•	•	•
DISCARD	discard output	VDISCARD	c_lflag	IEXTEN	^O		•	•	•	•
DSUSP	delayed suspend (SIGTSTP)	VDSUSP	c_lflag	ISIG	^Y		•		•	•
EOF	end of file	VEOF	c_lflag	ICANON	^D	•	•	•	•	•
EOL	end of line	VEOL	c_lflag	ICANON		•	•	•	•	•
EOL2	alternate end of line	VEOL2	c_lflag	ICANON			•	•	•	•
ERASE	backspace one character	VERASE	c_lflag	ICANON	^H, ^?	•	•	•	•	•
ERASE2	alternate backspace character	VERASE2	c_lflag	ICANON	^H, ^?		•			
INTR	interrupt signal (SIGINT)	VINTR	c_lflag	ISIG	^?, ^C	•	•	•	•	•
KILL	erase line	VKILL	c_lflag	ICANON	^U	•	•	•	•	•
LNEXT	literal next	VLNEXT	c_lflag	IEXTEN	^V		•	•	•	•
NL	line feed (newline)	(can't change)	c_lflag	ICANON	\n	•	•	•	•	•
QUIT	quit signal (SIGQUIT)	VQUIT	c_lflag	ISIG	^\	•	•	•	•	•
REPRINT	reprint all input	VREPRINT	c_lflag	ICANON	^R		•	•	•	•
START	resume output	VSTART	c_iflag	IXON/IXOFF	^Q	•	•	•	•	•
STATUS	status request	VSTATUS	c_lflag	ICANON	^T		•		•	
STOP	stop output	VSTOP	c_iflag	IXON/IXOFF	^S	•	•	•	•	•
SUSP	suspend signal (SIGTSTP)	VSUSP	c_lflag	ISIG	^Z	•	•	•	•	•
WERASE	backspace one word	VWERASE	c_lflag	ICANON	^W		•	•	•	•

Figure 18.9 Summary of special terminal input characters

Of the 11 POSIX.1 special characters, we can change 9 of them to almost any value that we like. The exceptions are the newline and carriage return characters (\n and \r, respectively) and perhaps the STOP and START characters (depends on the implementation). To do this, we modify the appropriate entry in the c_cc array of the termios structure. The elements in this array are referred to by name, with each name beginning with a V (the third column in Figure 18.9).

POSIX.1 allows us to disable these characters. If we set the value of an entry in the c_cc array to the value of _POSIX_VDISABLE, then we disable the corresponding special character.

> In early versions of the Single UNIX Specification, support for _POSIX_VDISABLE was optional. It is now required.
>
> All four platforms discussed in this text support this feature. Linux 3.2.0 and Solaris 10 define _POSIX_VDISABLE as 0; FreeBSD 8.0 and Mac OS X 10.6.8 define it as 0xff.
>
> Some earlier UNIX systems disabled a feature if the corresponding special input character was 0.

Example

Before describing all the special characters in detail, let's look at a small program that changes them. The program in Figure 18.10 disables the interrupt character and sets the end-of-file character to Control-B.

```
#include "apue.h"
#include <termios.h>

int
main(void)
{
    struct termios  term;
    long            vdisable;

    if (isatty(STDIN_FILENO) == 0)
        err_quit("standard input is not a terminal device");

    if ((vdisable = fpathconf(STDIN_FILENO, _PC_VDISABLE)) < 0)
        err_quit("fpathconf error or _POSIX_VDISABLE not in effect");

    if (tcgetattr(STDIN_FILENO, &term) < 0) /* fetch tty state */
        err_sys("tcgetattr error");

    term.c_cc[VINTR] = vdisable;    /* disable INTR character */
    term.c_cc[VEOF]  = 2;           /* EOF is Control-B */

    if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &term) < 0)
        err_sys("tcsetattr error");

    exit(0);
}
```

Figure 18.10 Disable interrupt character and change end-of-file character

Note the following points regarding this program.

- We modify the terminal characters only if standard input is a terminal device. We call `isatty` (Section 18.9) to check this.
- We fetch the `_POSIX_VDISABLE` value using `fpathconf`.
- The function `tcgetattr` (Section 18.4) fetches a `termios` structure from the kernel. After we've modified this structure, we call `tcsetattr` to set the attributes. The only attributes that change are the ones we specifically modified.
- Disabling the interrupt key is different from ignoring the interrupt signal. The program in Figure 18.10 simply disables the special character that causes the terminal driver to generate `SIGINT`. We can still use the `kill` function to send the signal to the process. □

We now describe each of the special characters in more detail. We call these the special input characters, but two of the characters, STOP and START (Control-S and Control-Q), are also handled specially when output. Note that when recognized by the terminal driver and processed specially, most of these special characters are then discarded: they are not returned to the process in a read operation. The exceptions are the newline characters (NL, EOL, EOL2) and the carriage return (CR).

CR The carriage return character. We cannot change this character. This character is recognized on input in canonical mode. When both `ICANON` (canonical mode) and `ICRNL` (map CR to NL) are set and `IGNCR` (ignore CR) is not set, the CR character is translated to NL and has the same effect as a NL character. This character is returned to the reading process (perhaps after being translated to a NL).

DISCARD The discard character. This character, recognized on input in extended mode (`IEXTEN`), causes subsequent output to be discarded until another DISCARD character is entered or the discard condition is cleared (see the `FLUSHO` option). This character is discarded when processed (i.e., it is not passed to the process).

DSUSP The delayed-suspend job-control character. This character is recognized on input in extended mode (`IEXTEN`) if job control is supported and if the `ISIG` flag is set. Like the SUSP character, this delayed-suspend character generates the `SIGTSTP` signal that is sent to all processes in the foreground process group (refer to Figure 9.7). However, the delayed-suspend character generates a signal only when a process reads from the controlling terminal, not when the character is typed. This character is discarded when processed (i.e., it is not passed to the process).

EOF The end-of-file character. This character is recognized on input in canonical mode (`ICANON`). When we type this character, all bytes waiting to be read are immediately passed to the reading process. If no bytes are waiting to be read, a count of 0 is returned. Entering an EOF character at the beginning of the line is the normal way to indicate the end of file to a program. This character is discarded when processed in canonical mode (i.e., it is not passed to the process).

EOL The additional line delimiter character, like NL. This character is recognized on input in canonical mode (`ICANON`) and is returned to the reading process; however, this character is not normally used.

EOL2 Another line delimiter character, like NL. This character is treated identically to the EOL character.

ERASE The erase character (backspace). This character is recognized on input in canonical mode (`ICANON`) and erases the previous character in the line, not erasing beyond the beginning of the line. The erase character is discarded when processed in canonical mode (i.e., it is not passed to the process).

ERASE2 The alternate erase character (backspace). This character is treated exactly like the erase character (ERASE).

INTR The interrupt character. This character is recognized on input if the `ISIG` flag is set and generates the `SIGINT` signal that is sent to all processes in the foreground process group (refer to Figure 9.7). This character is discarded when processed (i.e., it is not passed to the process).

KILL The kill character. (The name "kill" is overused; recall the `kill` function used to send a signal to a process. This character should be called the line-erase character; it has nothing to do with signals.) It is recognized on input in canonical mode (`ICANON`). It erases the entire line and is discarded when processed (i.e., it is not passed to the process).

LNEXT The literal-next character. This character is recognized on input in extended mode (`IEXTEN`) and causes any special meaning of the next character to be ignored. This works for all special characters listed in this section. We can use this character to type any character to a program. The LNEXT character is discarded when processed, but the next character entered is passed to the process.

NL The newline character, also called the line delimiter. We cannot change this character. It is recognized on input in canonical mode (`ICANON`). This character is returned to the reading process.

QUIT The quit character. This character is recognized on input if the `ISIG` flag is set. The quit character generates the `SIGQUIT` signal, which is sent to all processes in the foreground process group (refer to Figure 9.7). This character is discarded when processed (i.e., it is not passed to the process).

Recall from Figure 10.1 that the difference between INTR and QUIT is that the QUIT character not only terminates the process by default, but also generates a `core` file.

REPRINT The reprint character. This character is recognized on input in extended, canonical mode (both `IEXTEN` and `ICANON` flags set) and causes all unread input to be output (reechoed). This character is discarded when processed (i.e., it is not passed to the process).

START The start character. This character is recognized on input if the `IXON` flag is set and is automatically generated as output if the `IXOFF` flag is set. A received START character with `IXON` set causes stopped output (from a previously entered STOP character) to restart. In this case, the START character is discarded when processed (i.e., it is not passed to the process).

When `IXOFF` is set, the terminal driver automatically generates a START character to resume input that it had previously stopped, when the new input will not overflow the input buffer.

STATUS The BSD status-request character. This character is recognized on input in extended, canonical mode (both `IEXTEN` and `ICANON` flags set) and generates the `SIGINFO` signal, which is sent to all processes in the foreground process group (refer to Figure 9.7). Additionally, if the `NOKERNINFO` flag is not set, status information on the foreground process group is displayed on the terminal. This character is discarded when processed (i.e., it is not passed to the process).

STOP The stop character. This character is recognized on input if the `IXON` flag is set and is automatically generated as output if the `IXOFF` flag is set. A received STOP character with `IXON` set stops the output. In this case, the STOP character is discarded when processed (i.e., it is not passed to the process). The stopped output is restarted when a START character is entered.

When `IXOFF` is set, the terminal driver automatically generates a STOP character to prevent the input buffer from overflowing.

SUSP The suspend job-control character. This character is recognized on input if job control is supported and if the `ISIG` flag is set. The suspend character generates the `SIGTSTP` signal, which is sent to all processes in the foreground process group (refer to Figure 9.7). This character is discarded when processed (i.e., it is not passed to the process).

WERASE The word-erase character. This character is recognized on input in extended, canonical mode (both `IEXTEN` and `ICANON` flags set) and causes the previous word to be erased. First, it skips backward over any white space (spaces or tabs), then skips backward over the previous token, leaving the cursor positioned where the first character of the previous token was located. Normally, the previous token ends when a white space character is encountered. We can change this behavior, however, by setting the `ALTWERASE` flag. This flag causes the previous token to end when the first nonalphanumeric character is encountered. The word-erase character is discarded when processed (i.e., it is not passed to the process).

Another "character" that we need to define for terminal devices is the BREAK character. BREAK is not really a character, but rather a condition that occurs during asynchronous serial data transmission. A BREAK condition is signaled to the device driver in various ways, depending on the serial interface.

> Most old serial terminals have a key labeled BREAK that generates the BREAK condition, which is why most people think of BREAK as a character. Some newer terminal keyboards don't have a BREAK key. On PCs, the break key might be mapped for another purpose. For example, the Windows command interpreter can be interrupted by pressing Control-BREAK.

For asynchronous serial data transmission, a BREAK is a sequence of zero-valued bits that continues for longer than the time required to send one byte. The entire sequence of zero-valued bits is considered a single BREAK. In Section 18.8, we'll see how to send a BREAK with the `tcsendbreak` function.

18.4 Getting and Setting Terminal Attributes

To get and set a `termios` structure, we call two functions: `tcgetattr` and `tcsetattr`. This is how we examine and modify the various option flags and special characters to make the terminal operate the way we want it to.

```
#include <termios.h>

int tcgetattr(int fd, struct termios *termptr);

int tcsetattr(int fd, int opt, const struct termios *termptr);
```

Both return: 0 if OK, –1 on error

Both functions take a pointer to a `termios` structure and either return the current terminal attributes or set the terminal's attributes. Since these two functions operate only on terminal devices, `errno` is set to `ENOTTY` and –1 is returned if *fd* does not refer to a terminal device.

The argument *opt* for `tcsetattr` lets us specify when we want the new terminal attributes to take effect. This argument is specified as one of the following constants.

`TCSANOW` The change occurs immediately.

`TCSADRAIN` The change occurs after all output has been transmitted. This option should be used if we are changing the output parameters.

`TCSAFLUSH` The change occurs after all output has been transmitted. Furthermore, when the change takes place, all input data that has not been read is discarded (flushed).

The return status of `tcsetattr` can be confusing to use correctly. This function returns OK if it was able to perform *any* of the requested actions, even if it couldn't perform all the requested actions. If the function returns OK, it is our responsibility to see whether all the requested actions were performed. This means that after we call `tcsetattr` to set the desired attributes, we need to call `tcgetattr` and compare the actual terminal's attributes to the desired attributes to detect any differences.

What are the attributes of a terminal we open for the first time? The answer is "it depends." Some systems might initialize the terminal attributes to implementation-defined values. Other systems might leave the attributes with the values they had the last time that the terminal was used. If we want to be sure that the terminal behavior conforms to the standard, we can open the terminal device with the `O_TTY_INIT` flag (see Section 3.3). This will ensure that when we call `tcgetattr`, any nonstandard portions of the `termios` structure will be initialized so the terminal will behave as expected when we change the attributes and call `tcsetattr`.

18.5 Terminal Option Flags

In this section, we list all the various terminal option flags, expanding the descriptions from Figures 18.3 through 18.6. This list is alphabetical and indicates in which of the four terminal flag fields the option appears. (The field that controls a given option is usually not apparent from the option name alone.) We also note whether each option is defined by the Single UNIX Specification and list the platforms that support it.

All the flags listed specify one or more bits that we turn on or clear, unless we call the flag a *mask*. A mask defines multiple bits grouped together from which a set of values is defined. We have a defined name for the mask and a name for each value. For example, to set the character size, we first zero the bits using the character-size mask `CSIZE`, and then set one of the values `CS5`, `CS6`, `CS7`, or `CS8`.

The six delay values supported by Linux and Solaris are also masks: `BSDLY`, `CRDLY`, `FFDLY`, `NLDLY`, `TABDLY`, and `VTDLY`. Refer to the `termio`(7I) manual page on Solaris for the length of each delay value. In all cases, a delay mask of 0 means no delay. If a delay is specified, the `OFILL` and `OFDEL` flags determine whether the driver does an actual delay or whether fill characters are transmitted instead.

Example

Figure 18.11 demonstrates the use of these masks to extract a value and to set a value.

```
#include "apue.h"
#include <termios.h>

int
main(void)
{
    struct termios  term;

    if (tcgetattr(STDIN_FILENO, &term) < 0)
        err_sys("tcgetattr error");

    switch (term.c_cflag & CSIZE) {
    case CS5:
        printf("5 bits/byte\n");
        break;
    case CS6:
        printf("6 bits/byte\n");
        break;
    case CS7:
        printf("7 bits/byte\n");
        break;
    case CS8:
        printf("8 bits/byte\n");
        break;
    default:
        printf("unknown bits/byte\n");
    }

    term.c_cflag &= ~CSIZE;     /* zero out the bits */
    term.c_cflag |= CS8;        /* set 8 bits/byte */
    if (tcsetattr(STDIN_FILENO, TCSANOW, &term) < 0)
        err_sys("tcsetattr error");

    exit(0);
}
```

Figure 18.11 Example of `tcgetattr` and `tcsetattr`

□

We now describe each of the flags.

ALTWERASE (c_lflag, FreeBSD, Mac OS X) If set, an alternate word-erase algorithm is used when the WERASE character is entered. Instead of moving backward until the previous white space character, this flag causes the WERASE character to move backward until the first nonalphanumeric character is encountered.

BRKINT (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If this flag is set and IGNBRK is not set, the input and output queues are flushed when a BREAK is received, and a SIGINT signal is generated. This signal is generated for the foreground process group if the terminal device is a controlling terminal.

If neither IGNBRK nor BRKINT is set, then a BREAK is read as a single character \0, unless PARMRK is set; in that case the BREAK is read as the 3-byte sequence \377, \0, \0.

BSDLY (c_oflag, XSI, Linux, Solaris) Backspace delay mask. The values for the mask are BS0 or BS1.

CBAUDEXT (c_cflag, Solaris) Extended baud rates. Used to enable baud rates greater than B38400. (We discuss baud rates in Section 18.7.)

CCAR_OFLOW (c_cflag, FreeBSD, Mac OS X) Enable hardware flow control of the output using the RS-232 modem carrier signal Data-Carrier-Detect (DCD). This is the same as the old MDMBUF flag.

CCTS_OFLOW (c_cflag, FreeBSD, Mac OS X, Solaris) Enable hardware flow control of the output using the Clear-To-Send (CTS) RS-232 signal.

CDSR_OFLOW (c_cflag, FreeBSD, Mac OS X) Flow control the output according to the Data-Set-Ready (DSR) RS-232 signal.

CDTR_IFLOW (c_cflag, FreeBSD, Mac OS X) Flow control the input according to the Data-Terminal-Ready (DTR) RS-232 signal.

CIBAUDEXT (c_cflag, Solaris) Extended input baud rates. Used to enable input baud rates greater than B38400. (We discuss baud rates in Section 18.7.)

CIGNORE (c_cflag, FreeBSD, Mac OS X) Ignore control flags.

CLOCAL (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the modem status lines are ignored. This usually means that the device is directly attached. When this flag is not set, an open of a terminal device usually blocks until the modem answers a call and establishes a connection, for example.

CMSPAR (c_oflag, Linux) Select mark or space parity. If PARODD is set, the parity bit is always 1 (mark parity). Otherwise, the parity bit is always 0 (space parity).

CRDLY (c_oflag, XSI, Linux, Solaris) Carriage return delay mask. Possible values for the mask are CR0, CR1, CR2, and CR3.

`CREAD` (`c_cflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the receiver is enabled and characters can be received.

`CRTSCTS` (`c_cflag`, FreeBSD, Linux, Mac OS X, Solaris) Behavior depends on platform. For Solaris, enables outbound hardware flow control if set. On the other three platforms, enables both inbound and outbound hardware flow control (equivalent to `CCTS_OFLOW|CRTS_IFLOW`).

`CRTS_IFLOW` (`c_cflag`, FreeBSD, Mac OS X, Solaris) Request-To-Send (RTS) flow control of input.

`CRTSXOFF` (`c_cflag`, Solaris) If set, inbound hardware flow control is enabled. The state of the Request-To-Send RS-232 signal controls the flow control.

`CSIZE` (`c_cflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) This field is a mask that specifies the number of bits per byte for both transmission and reception. This size does not include the parity bit, if any. The values for the field defined by this mask are `CS5`, `CS6`, `CS7`, and `CS8`, for 5, 6, 7, and 8 bits per byte, respectively.

`CSTOPB` (`c_cflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, two stop bits are used; otherwise, one stop bit is used.

`ECHO` (`c_lflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, input characters are echoed back to the terminal device. Input characters can be echoed in either canonical or noncanonical mode.

`ECHOCTL` (`c_lflag`, FreeBSD, Linux, Mac OS X, Solaris) If set and if `ECHO` is set, ASCII control characters (those characters in the range 0 through octal 37, inclusive) other than the ASCII TAB, the ASCII NL, and the START and STOP characters are echoed as `^`*X*, where *X* is the character formed by adding octal 100 to the control character. For example, the ASCII Control-A character (octal 1) is echoed as `^A` and the ASCII DELETE character (octal 177) is echoed as `^?`. If this flag is not set, the ASCII control characters are echoed as themselves. As with the `ECHO` flag, this flag affects the echoing of control characters in both canonical and noncanonical modes.

Be aware that some systems echo the EOF character differently, since its typical value is Control-D. (Control-D is the ASCII EOT character, which can cause some terminals to hang up.) Check your manual.

`ECHOE` (`c_lflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set and if `ICANON` is set, the ERASE character erases the last character in the current line from the display. This is usually done in the terminal driver by writing the three-character sequence backspace, space, backspace.

If the WERASE character is supported, `ECHOE` causes the previous word to be erased using one or more of the same three-character sequence.

If the `ECHOPRT` flag is supported, the actions described here for `ECHOE` assume that the `ECHOPRT` flag is not set.

ECHOK (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set and if ICANON is set, the KILL character erases the current line from the display or outputs the NL character (to emphasize that the entire line was erased).

If the ECHOKE flag is supported, this description of ECHOK assumes that ECHOKE is not set.

ECHOKE (c_lflag, FreeBSD, Linux, Mac OS X, Solaris) If set and if ICANON is set, the KILL character is echoed by erasing each character on the line. The way in which each character is erased is selected by the ECHOE and ECHOPRT flags.

ECHONL (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set and if ICANON is set, the NL character is echoed, even if ECHO is not set.

ECHOPRT (c_lflag, FreeBSD, Linux, Mac OS X, Solaris) If set and if both ICANON and ECHO are set, then the ERASE character (and WERASE character, if supported) cause all the characters being erased to be printed as they are erased. This is often useful on a hard-copy terminal to see exactly which characters are being deleted.

EXTPROC (c_lflag, FreeBSD, Linux, Mac OS X) If set, canonical character processing is performed external to the operating system. This can be the case if the serial communication peripheral card can offload the host processor by doing some of the line discipline processing. This can also be the case when using pseudo terminals (Chapter 19).

FFDLY (c_oflag, XSI, Linux, Solaris) Form feed delay mask. The values for the mask are FF0 or FF1.

FLUSHO (c_lflag, FreeBSD, Linux, Mac OS X, Solaris) If set, output is being flushed. This flag is set when we type the DISCARD character; the flag is cleared when we type another DISCARD character. We can also set or clear this condition by setting or clearing this terminal flag.

HUPCL (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the modem control lines are lowered (i.e., the modem connection is broken) when the last process closes the device.

ICANON (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, canonical mode is in effect (Section 18.10). This enables the following characters: EOF, EOL, EOL2, ERASE, KILL, REPRINT, STATUS, and WERASE. The input characters are assembled into lines.

If canonical mode is not enabled, read requests are satisfied directly from the input queue. A read does not return until at least MIN bytes have been received or the timeout value TIME has expired between bytes. Refer to Section 18.11 for additional details.

ICRNL (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set and if IGNCR is not set, a received CR character is translated into a NL character.

IEXTEN (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the extended, implementation-defined special characters are recognized and processed.

IGNBRK (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) When set, a BREAK condition on input is ignored. See BRKINT for a way to have a BREAK condition either generate a SIGINT signal or be read as data.

IGNCR (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, a received CR character is ignored. If this flag is not set, it is possible to translate the received CR into a NL character if the ICRNL flag is set.

IGNPAR (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) When set, an input byte with a framing error (other than a BREAK) or an input byte with a parity error is ignored.

IMAXBEL (c_iflag, FreeBSD, Linux, Mac OS X, Solaris) Ring bell when input queue is full.

INLCR (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, a received NL character is translated into a CR character.

INPCK (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) When set, input parity checking is enabled. If INPCK is not set, input parity checking is disabled.

Parity "generation and detection" and "input parity checking" are two different things. The generation and detection of parity bits is controlled by the PARENB flag. Setting this flag usually causes the device driver for the serial interface to generate parity for outgoing characters and to verify the parity of incoming characters. The flag PARODD determines whether the parity should be odd or even. If an input character arrives with the wrong parity, then the state of the INPCK flag is checked. If this flag is set, then the IGNPAR flag is checked (to see whether the input byte with the parity error should be ignored); if the byte should not be ignored, then the PARMRK flag is checked to see which characters should be passed to the reading process.

ISIG (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the input characters are compared against the special characters that cause the terminal-generated signals to be generated (INTR, QUIT, SUSP, and DSUSP); if equal, the corresponding signal is generated.

ISTRIP (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) When set, valid input bytes are stripped to 7 bits. When this flag is not set, all 8 bits are processed.

IUCLC (c_iflag, Linux, Solaris) Map uppercase to lowercase on input.

`IUTF8` (`c_iflag`, Linux, Mac OS X) Allow character erase processing to work with UTF-8 multibyte characters.

`IXANY` (`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) Enable any characters to restart output.

`IXOFF` (`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, start–stop input control is enabled. When it notices that the input queue is getting full, the terminal driver outputs a STOP character. This character should be recognized by the device that is sending the data and cause the device to stop. Later, when the characters on the input queue have been processed, the terminal driver will output a START character. This should cause the device to resume sending data.

`IXON` (`c_iflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, start–stop output control is enabled. When the terminal driver receives a STOP character, output stops. While the output is stopped, the next START character resumes the output. If this flag is not set, the START and STOP characters are read by the process as normal characters.

`MDMBUF` (`c_cflag`, FreeBSD, Mac OS X) Flow control the output according to the modem carrier flag. This is the old name for the `CCAR_OFLOW` flag.

`NLDLY` (`c_oflag`, XSI, Linux, Solaris) Newline delay mask. The values for the mask are `NL0` or `NL1`.

`NOFLSH` (`c_lflag`, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) By default, when the terminal driver generates the `SIGINT` and `SIGQUIT` signals, both the input and output queues are flushed. Also, when it generates the `SIGSUSP` signal, the input queue is flushed. If the `NOFLSH` flag is set, this normal flushing of the queues does not occur when these signals are generated.

`NOKERNINFO` (`c_lflag`, FreeBSD, Mac OS X) When set, this flag prevents the STATUS character from printing information on the foreground process group. Regardless of whether this flag is set, however, the STATUS character still causes the `SIGINFO` signal to be sent to the foreground process group.

`OCRNL` (`c_oflag`, XSI, FreeBSD, Linux, Solaris) If set, map CR to NL on output.

`OFDEL` (`c_oflag`, XSI, Linux, Solaris) If set, the output fill character is ASCII DEL; otherwise, it's ASCII NUL. See the `OFILL` flag.

`OFILL` (`c_oflag`, XSI, Linux, Solaris) If set, fill characters (either ASCII DEL or ASCII NUL; see the `OFDEL` flag) are transmitted for a delay, instead of using a timed delay. See the six delay masks: `BSDLY`, `CRDLY`, `FFDLY`, `NLDLY`, `TABDLY`, and `VTDLY`.

`OLCUC` (`c_oflag`, Linux, Solaris) If set, map lowercase characters to uppercase characters on output.

ONLCR (c_oflag, XSI, FreeBSD, Linux, Mac OS X, Solaris) If set, map NL to CR-NL on output.

ONLRET (c_oflag, XSI, FreeBSD, Linux, Solaris) If set, the NL character is assumed to perform the carriage return function on output.

ONOCR (c_oflag, XSI, FreeBSD, Linux, Solaris) If set, a CR is not output at column 0.

ONOEOT (c_oflag, FreeBSD, Mac OS X) If set, EOT (^D) characters are discarded on output. This may be necessary on some terminals that interpret Control-D as a hangup.

OPOST (c_oflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, implementation-defined output processing takes place. Refer to Figure 18.6 for the various implementation-defined flags for the c_oflag field.

OXTABS (c_oflag, FreeBSD, Mac OS X) If set, tabs are expanded to spaces on output. This produces the same effect as setting the horizontal tab delay (TABDLY) to XTABS or TAB3.

PARENB (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, parity generation is enabled for outgoing characters, and parity checking is performed on incoming characters. The parity is odd if PARODD is set; otherwise, it is even parity. See also the discussion of the INPCK, IGNPAR, and PARMRK flags.

PAREXT (c_cflag, Solaris) Select mark or space parity. If PARODD is set, the parity bit is always 1 (mark parity). Otherwise, the parity bit is always 0 (space parity).

PARMRK (c_iflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) When set and if IGNPAR is not set, a byte with a framing error (other than a BREAK) or a byte with a parity error is read by the process as the three-character sequence \377, \0, *X*, where *X* is the byte received in error. If ISTRIP is not set, a valid \377 is passed to the process as \377, \377. If neither IGNPAR nor PARMRK is set, a byte with a framing error (other than a BREAK) or with a parity error is read as a single character \0.

PARODD (c_cflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set, the parity for outgoing and incoming characters is odd parity. Otherwise, the parity is even parity. Note that the PARENB flag controls the generation and detection of parity.

The PARODD flag also controls whether mark or space parity is used when either the CMSPAR or PAREXT flag is set.

PENDIN (c_lflag, FreeBSD, Linux, Mac OS X, Solaris) If set, any input that has not been read is reprinted by the system when the next character is input. This action is similar to what happens when we type the REPRINT character.

TABDLY (c_oflag, XSI, Linux, Mac OS X, Solaris) Horizontal tab delay mask. The values for the mask are TAB0, TAB1, TAB2, or TAB3.

The value XTABS is equal to TAB3. This value causes the system to expand tabs into spaces. The system assumes a tab stop every eight spaces, and we can't change this assumption.

TOSTOP (c_lflag, POSIX.1, FreeBSD, Linux, Mac OS X, Solaris) If set and if the implementation supports job control, the SIGTTOU signal is sent to the process group of a background process that tries to write to its controlling terminal. By default, this signal stops all the processes in the process group. This signal is not generated by the terminal driver if the background process that is writing to the controlling terminal is either ignoring or blocking the signal.

VTDLY (c_oflag, XSI, Linux, Solaris) Vertical tab delay mask. The values for the mask are VT0 and VT1.

XCASE (c_lflag, Linux, Solaris) If set and if ICANON is also set, the terminal is assumed to be uppercase only, and all input is converted to lowercase. To input an uppercase character, precede it with a backslash. Similarly, the system outputs an uppercase character by preceding it with a backslash. (This option flag is obsolete today, since most, if not all, uppercase-only terminals have disappeared.)

18.6 stty Command

All the options described in the previous section can be examined and changed from within a program with the tcgetattr and tcsetattr functions (Section 18.4) or from the command line (or a shell script) with the stty(1) command. This command is simply an interface to the first six functions that we listed in Figure 18.7. If we execute this command with its -a option, it displays all the terminal options:

```
$ stty -a
speed 9600 baud; 25 rows; 80 columns;
lflags: icanon isig iexten echo echoe -echok echoke -echonl echoctl
        -echoprt -altwerase -noflsh -tostop -flusho pendin -nokerninfo
        -extproc
iflags: -istrip icrnl -inlcr -igncr ixon -ixoff ixany imaxbel -ignbrk
        brkint -inpck -ignpar -parmrk
oflags: opost onlcr -ocrnl -oxtabs -onocr -onlret
cflags: cread cs8 -parenb -parodd hupcl -clocal -cstopb -crtscts
        -dsrflow -dtrflow -mdmbuf
cchars: discard = ^O; dsusp = ^Y; eof = ^D; eol = <undef>;
        eol2 = <undef>; erase = ^H; erase2 = ^?; intr = ^C; kill = ^U;
        lnext = ^V; min = 1; quit = ^; reprint = ^R; start = ^Q;
        status = ^T; stop = ^S; susp = ^Z; time = 0; werase = ^W;
```

Option names preceded by a hyphen are disabled. The last four lines display the current settings for each of the terminal special characters (Section 18.3). The first line displays the number of rows and columns for the current terminal window; we discuss the terminal window size in Section 18.12.

The `stty` command uses its standard input to get and set the terminal option flags. Although some older implementations used standard output, POSIX.1 requires that the standard input be used. All four implementations discussed in this text provide versions of `stty` that operate on the standard input. This means that we can type

```
stty -a </dev/tty1a
```

if we are interested in discovering the settings on the terminal named `tty1a`.

18.7 Baud Rate Functions

The term *baud rate* is a historical term that should be referred to today as "bits per second." Although most terminal devices use the same baud rate for both input and output, the capability exists to set the two rates to different values, if the hardware allows this.

```
#include <termios.h>

speed_t cfgetispeed(const struct termios *termptr);

speed_t cfgetospeed(const struct termios *termptr);

                                        Both return: baud rate value

int cfsetispeed(struct termios *termptr, speed_t speed);

int cfsetospeed(struct termios *termptr, speed_t speed);

                                        Both return: 0 if OK, –1 on error
```

The return value from the two `cfget` functions and the *speed* argument to the two `cfset` functions are one of the following constants: `B50`, `B75`, `B110`, `B134`, `B150`, `B200`, `B300`, `B600`, `B1200`, `B1800`, `B2400`, `B4800`, `B9600`, `B19200`, or `B38400`. The constant `B0` means "hang up." If `B0` is specified as the output baud rate when `tcsetattr` is called, the modem control lines are no longer asserted.

Most systems define additional baud rate values, such as `B57600` and `B115200`.

To use these functions, we must realize that the input and output baud rates are stored in the device's `termios` structure, as shown in Figure 18.8. Before calling either of the `cfget` functions, we first have to obtain the device's `termios` structure using `tcgetattr`. Similarly, after calling either of the two `cfset` functions, all we've done is set the baud rate in a `termios` structure. For this change to affect the device, we have to call `tcsetattr`. If there is an error in either of the baud rates that we set, we may not find out about the error until we call `tcsetattr`.

The four baud rate functions exist to insulate applications from differences in the way that implementations represent baud rates in the `termios` structure. Linux and BSD-derived platforms tend to store baud rates as numeric values equal to the rates (i.e., 9,600 baud is stored as the value 9,600), whereas System V–derived platforms (such as Solaris) tend to encode the baud rate in a bitmask. The speed values we get from the `cfget` functions and pass to the `cfset` functions are untranslated from their representation as they are stored in the `termios` structure.

18.8 Line Control Functions

The following four functions provide line control capability for terminal devices. All four require that *fd* refer to a terminal device; otherwise, –1 is returned with `errno` set to `ENOTTY`.

```
#include <termios.h>

int tcdrain(int fd);

int tcflow(int fd, int action);

int tcflush(int fd, int queue);

int tcsendbreak(int fd, int duration);
```

All four return: 0 if OK, –1 on error

The `tcdrain` function waits for all output to be transmitted. The `tcflow` function gives us control over both input and output flow control. The *action* argument must be one of the following four values:

`TCOOFF` Output is suspended.

`TCOON` Output that was previously suspended is restarted.

`TCIOFF` The system transmits a STOP character, which should cause the terminal device to stop sending data.

`TCION` The system transmits a START character, which should cause the terminal device to resume sending data.

The `tcflush` function lets us flush (throw away) either the input buffer (data that has been received by the terminal driver, which we have not read) or the output buffer (data that we have written, which has not yet been transmitted). The *queue* argument must be one of the following three constants:

`TCIFLUSH` The input queue is flushed.

`TCOFLUSH` The output queue is flushed.

`TCIOFLUSH` Both the input and the output queues are flushed.

The `tcsendbreak` function transmits a continuous stream of zero bits for a specified duration. If the *duration* argument is 0, the transmission lasts between 0.25 second and 0.5 second. POSIX.1 specifies that if *duration* is nonzero, the transmission time is implementation dependent.

18.9 Terminal Identification

Historically, the name of the controlling terminal in most versions of the UNIX System has been `/dev/tty`. POSIX.1 provides a runtime function that we can call to determine the name of the controlling terminal.

```
#include <stdio.h>

char *ctermid(char *ptr);
```

Returns: pointer to name of controlling terminal on success, pointer to empty string on error

If *ptr* is non-null, it is assumed to point to an array of at least `L_ctermid` bytes, and the name of the controlling terminal of the process is stored in the array. The constant `L_ctermid` is defined in `<stdio.h>`. If *ptr* is a null pointer, the function allocates room for the array (usually as a static variable). Again, the name of the controlling terminal of the process is stored in the array.

In both cases, the starting address of the array is returned as the value of the function. Since most UNIX systems use `/dev/tty` as the name of the controlling terminal, this function is intended to aid portability to other operating systems.

All four platforms described in this text return the string `/dev/tty` when we call `ctermid`.

Example — `ctermid` Function

Figure 18.12 shows an implementation of the POSIX.1 `ctermid` function.

```
#include    <stdio.h>
#include    <string.h>

static char ctermid_name[L_ctermid];

char *
ctermid(char *str)
{
    if (str == NULL)
        str = ctermid_name;
    return(strcpy(str, "/dev/tty"));    /* strcpy() returns str */
}
```

Figure 18.12 Implementation of POSIX.1 `ctermid` function

Note that we can't protect against overrunning the caller's buffer, because we have no way to determine its size. □

Two functions that are more interesting for a UNIX system are `isatty`, which returns true if a file descriptor refers to a terminal device, and `ttyname`, which returns the pathname of the terminal device that is open on a file descriptor.

```
#include <unistd.h>

int isatty(int fd);
```

Returns: 1 (true) if terminal device, 0 (false) otherwise

```
char *ttyname(int fd);
```

Returns: pointer to pathname of terminal, `NULL` on error

Example — `isatty` Function

The `isatty` function is trivial to implement, as we show in Figure 18.13. We simply try one of the terminal-specific functions (that doesn't change anything if it succeeds) and look at the return value.

```
#include    <termios.h>

int
isatty(int fd)
{
    struct termios  ts;

    return(tcgetattr(fd, &ts) != -1); /* true if no error (is a tty) */
}
```

Figure 18.13 Implementation of POSIX.1 `isatty` function

We test our `isatty` function with the program in Figure 18.14.

```
#include "apue.h"

int
main(void)
{
    printf("fd 0: %s\n", isatty(0) ? "tty" : "not a tty");
    printf("fd 1: %s\n", isatty(1) ? "tty" : "not a tty");
    printf("fd 2: %s\n", isatty(2) ? "tty" : "not a tty");
    exit(0);
}
```

Figure 18.14 Test the `isatty` function

When we run the program from Figure 18.14, we get the following output:

```
$ ./a.out
fd 0: tty
fd 1: tty
fd 2: tty
$ ./a.out </etc/passwd 2>/dev/null
fd 0: not a tty
fd 1: tty
fd 2: not a tty
```

□

Example — `ttyname` Function

The `ttyname` function (Figure 18.15) is longer, as we have to search all the device entries, looking for a match.

```
#include    <sys/stat.h>
#include    <dirent.h>
#include    <limits.h>
#include    <string.h>
#include    <termios.h>
#include    <unistd.h>
#include    <stdlib.h>

struct devdir {
    struct devdir   *d_next;
    char            *d_name;
};

static struct devdir    *head;
static struct devdir    *tail;
static char             pathname[_POSIX_PATH_MAX + 1];

static void
add(char *dirname)
{
    struct devdir   *ddp;
    int             len;

    len = strlen(dirname);

    /*
     * Skip ., .., and /dev/fd.
     */
    if ((dirname[len-1] == '.') && (dirname[len-2] == '/' ||
      (dirname[len-2] == '.' && dirname[len-3] == '/')))
        return;
    if (strcmp(dirname, "/dev/fd") == 0)
        return;
    if ((ddp = malloc(sizeof(struct devdir))) == NULL)
        return;
```

```
    if ((ddp->d_name = strdup(dirname)) == NULL) {
        free(ddp);
        return;
    }

    ddp->d_next = NULL;
    if (tail == NULL) {
        head = ddp;
        tail = ddp;
    } else {
        tail->d_next = ddp;
        tail = ddp;
    }
}

static void
cleanup(void)
{
    struct devdir   *ddp, *nddp;

    ddp = head;
    while (ddp != NULL) {
        nddp = ddp->d_next;
        free(ddp->d_name);
        free(ddp);
        ddp = nddp;
    }
    head = NULL;
    tail = NULL;
}

static char *
searchdir(char *dirname, struct stat *fdstatp)
{
    struct stat     devstat;
    DIR             *dp;
    int             devlen;
    struct dirent   *dirp;

    strcpy(pathname, dirname);
    if ((dp = opendir(dirname)) == NULL)
        return(NULL);
    strcat(pathname, "/");
    devlen = strlen(pathname);
    while ((dirp = readdir(dp)) != NULL) {
        strncpy(pathname + devlen, dirp->d_name,
          _POSIX_PATH_MAX - devlen);

        /*
         * Skip aliases.
         */
        if (strcmp(pathname, "/dev/stdin") == 0 ||
```

```
            strcmp(pathname, "/dev/stdout") == 0 ||
            strcmp(pathname, "/dev/stderr") == 0)
                continue;
            if (stat(pathname, &devstat) < 0)
                continue;
            if (S_ISDIR(devstat.st_mode)) {
                add(pathname);
                continue;
            }
            if (devstat.st_ino == fdstatp->st_ino &&
              devstat.st_dev == fdstatp->st_dev) {  /* found a match */
                closedir(dp);
                return(pathname);
            }
        }

        closedir(dp);
        return(NULL);
}

char *
ttyname(int fd)
{
        struct stat     fdstat;
        struct devdir   *ddp;
        char            *rval;

        if (isatty(fd) == 0)
            return(NULL);
        if (fstat(fd, &fdstat) < 0)
            return(NULL);
        if (S_ISCHR(fdstat.st_mode) == 0)
            return(NULL);

        rval = searchdir("/dev", &fdstat);
        if (rval == NULL) {
            for (ddp = head; ddp != NULL; ddp = ddp->d_next)
                if ((rval = searchdir(ddp->d_name, &fdstat)) != NULL)
                    break;
        }

        cleanup();
        return(rval);
}
```

Figure 18.15 Implementation of POSIX.1 `ttyname` function

The technique is to read the `/dev` directory, looking for an entry with the same device number and i-node number. Recall from Section 4.24 that each file system has a unique device number (the `st_dev` field in the `stat` structure, from Section 4.2), and each directory entry in that file system has a unique i-node number (the `st_ino` field in the `stat` structure). We assume in this function that when we hit a matching device

number and matching i-node number, we've located the desired directory entry. We could also verify that the two entries have matching `st_rdev` fields (the major and minor device numbers for the terminal device) and that the directory entry is a character special file. However, since we've already verified that the file descriptor argument is both a terminal device and a character special file, and since a matching device number and i-node number pair is unique on a UNIX system, there is no need for the additional comparisons.

The name of our terminal might reside in a subdirectory in `/dev`. Thus, we might need to search the entire file system tree under `/dev`. We skip several directories that might produce incorrect or odd-looking results: `/dev/.`, `/dev/..`, and `/dev/fd`. We also skip the aliases `/dev/stdin`, `/dev/stdout`, and `/dev/stderr`, since they are symbolic links to files in `/dev/fd`.

We can test this implementation with the program shown in Figure 18.16.

```
#include "apue.h"

int
main(void)
{
    char *name;

    if (isatty(0)) {
        name = ttyname(0);
        if (name == NULL)
            name = "undefined";
    } else {
        name = "not a tty";
    }
    printf("fd 0: %s\n", name);

    if (isatty(1)) {
        name = ttyname(1);
        if (name == NULL)
            name = "undefined";
    } else {
        name = "not a tty";
    }
    printf("fd 1: %s\n", name);

    if (isatty(2)) {
        name = ttyname(2);
        if (name == NULL)
            name = "undefined";
    } else {
        name = "not a tty";
    }
    printf("fd 2: %s\n", name);

    exit(0);
}
```

Figure 18.16 Test the `ttyname` function

Running the program from Figure 18.16 gives us

```
$ ./a.out < /dev/console 2> /dev/null
fd 0: /dev/console
fd 1: /dev/ttys001
fd 2: not a tty
```

□

18.10 Canonical Mode

Canonical mode is simple: we issue a read, and the terminal driver returns when a line has been entered. Several conditions cause the read to return.

- The read returns when the requested number of bytes have been read. We don't have to read a complete line. If we read a partial line, no information is lost; the next read starts where the previous read stopped.
- The read returns when a line delimiter is encountered. Recall from Section 18.3 that the following characters are interpreted as end of line in canonical mode: NL, EOL, EOL2, and EOF. Also, recall from Section 18.5 that if `ICRNL` is set and if `IGNCR` is not set, then the CR character also terminates a line, since it acts just like the NL character.

  Of these five line delimiters, one (EOF) is discarded by the terminal driver when it's processed. The other four are returned to the caller as the last character of the line.
- The read also returns if a signal is caught and if the function is not automatically restarted (Section 10.5).

Example—`getpass` Function

We now examine the function `getpass`, which reads a password of some type from the user at a terminal. This function is called by the `login`(1) and `crypt`(1) programs. To read the password, the function must turn off echoing, but it can leave the terminal in canonical mode, as whatever we type as the password forms a complete line. Figure 18.17 shows a typical implementation on a UNIX system.

There are several points to consider in this example.

- Instead of hard-wiring `/dev/tty` into the program, we call the function `ctermid` to open the controlling terminal.
- We read and write only to the controlling terminal and return an error if we can't open this device for reading and writing. There are other conventions to use. The version of `getpass` in the GNU C library reads from standard input and writes to standard error if the controlling terminal can't be opened for reading and writing. The Solaris version fails if it can't open the controlling terminal.

```
#include    <signal.h>
#include    <stdio.h>
#include    <termios.h>

#define MAX_PASS_LEN    8       /* max #chars for user to enter */

char *
getpass(const char *prompt)
{
    static char     buf[MAX_PASS_LEN + 1];  /* null byte at end */
    char            *ptr;
    sigset_t        sig, osig;
    struct termios  ts, ots;
    FILE            *fp;
    int             c;

    if ((fp = fopen(ctermid(NULL), "r+")) == NULL)
        return(NULL);
    setbuf(fp, NULL);

    sigemptyset(&sig);
    sigaddset(&sig, SIGINT);        /* block SIGINT */
    sigaddset(&sig, SIGTSTP);       /* block SIGTSTP */
    sigprocmask(SIG_BLOCK, &sig, &osig);    /* and save mask */

    tcgetattr(fileno(fp), &ts);     /* save tty state */
    ots = ts;                       /* structure copy */
    ts.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);
    tcsetattr(fileno(fp), TCSAFLUSH, &ts);
    fputs(prompt, fp);

    ptr = buf;
    while ((c = getc(fp)) != EOF && c != '\n')
        if (ptr < &buf[MAX_PASS_LEN])
            *ptr++ = c;
    *ptr = 0;           /* null terminate */
    putc('\n', fp);     /* we echo a newline */

    tcsetattr(fileno(fp), TCSAFLUSH, &ots); /* restore TTY state */
    sigprocmask(SIG_SETMASK, &osig, NULL);  /* restore mask */
    fclose(fp);         /* done with /dev/tty */
    return(buf);
}
```

Figure 18.17 Implementation of `getpass` function

- We block the two signals `SIGINT` and `SIGTSTP`. If we didn't do this, entering the INTR character would abort the program and leave the terminal with echoing disabled. Similarly, entering the SUSP character would stop the program and return to the shell with echoing disabled. We choose to block the signals while we have echoing disabled. If they are generated while we're reading the password, they are held until we return. There are other ways to

handle these signals. Some versions just ignore `SIGINT` (saving its previous action) while in `getpass`, resetting the action for this signal to its previous value before returning. This means that any occurrence of the signal while it's ignored is lost. Other versions catch `SIGINT` (saving its previous action) and if the signal is caught, send themselves the signal with the `kill` function after resetting the terminal state and signal action. None of the versions of `getpass` catch, ignore, or block `SIGQUIT`, so entering the QUIT character aborts the program and probably leaves the terminal with echoing disabled.

- Be aware that some shells, notably the Korn shell, turn echoing back on whenever they read interactive input. These shells are the ones that provide command-line editing and therefore manipulate the state of the terminal every time we enter an interactive command. So, if we invoke this program under one of these shells and abort it with the QUIT character, it may reenable echoing for us. Other shells that don't provide this form of command-line editing, such as the Bourne shell, will abort the program and leave the terminal in no-echo mode. If we do this to our terminal, the `stty` command can reenable echoing.
- We use standard I/O to read and write the controlling terminal. We specifically set the stream to be unbuffered; otherwise, there might be some interactions between the writing and reading of the stream (we would need some calls to `fflush`). We could have also used unbuffered I/O (Chapter 3), but we would have to simulate the `getc` function using `read`.
- We store only up to eight characters as the password. Any additional characters that are entered are ignored.

The program in Figure 18.18 calls `getpass` and prints what we enter to let us verify that the ERASE and KILL characters work (as they should in canonical mode).

```
#include "apue.h"

char    *getpass(const char *);

int
main(void)
{
    char    *ptr;

    if ((ptr = getpass("Enter password:")) == NULL)
        err_sys("getpass error");
    printf("password: %s\n", ptr);

    /* now use password (probably encrypt it) ... */

    while (*ptr != 0)
        *ptr++ = 0;     /* zero it out when we're done with it */
    exit(0);
}
```

Figure 18.18 Call the `getpass` function

Whenever a program that calls `getpass` is done with the cleartext password, the program should zero it out in memory, just to be safe. If the program were to generate a `core` file that others might be able to read or if some other process were somehow able to read our memory, they might be able to read the cleartext password. (By "cleartext," we mean the password that we type at the prompt that is printed by `getpass`. Most UNIX system programs then modify this cleartext password, turning it into an "encrypted" password. The `pw_passwd` field in the password file (Section 6.2), for example, contains the encrypted password, not the cleartext password.) □

18.11 Noncanonical Mode

Noncanonical mode is specified by turning off the `ICANON` flag in the `c_lflag` field of the `termios` structure. In noncanonical mode, the input data is not assembled into lines. The following special characters (Section 18.3) are not processed: ERASE, KILL, EOF, NL, EOL, EOL2, CR, REPRINT, STATUS, and WERASE.

As we said, understanding canonical mode is easy: the system returns up to one line at a time. But with noncanonical mode, how does the system know when to return data to us? If it returned one byte at a time, overhead would be excessive. (Recall Figure 3.6, which showed the overhead in reading one byte at a time. Each time we doubled the amount of data returned, we halved the system call overhead.) The system can't always return multiple bytes at a time, since sometimes we don't know how much data to read until we start reading it.

The solution is to tell the system to return when either a specified amount of data has been read or after a given amount of time has passed. This technique uses two variables in the `c_cc` array in the `termios` structure: MIN and TIME. These two elements of the array are indexed by the names `VMIN` and `VTIME`.

MIN specifies the minimum number of bytes before a `read` returns. TIME specifies the number of tenths of a second to wait for data to arrive. There are four cases.

Case A: MIN > 0, TIME > 0

TIME specifies an interbyte timer that is started only when the first byte is received. If MIN bytes are received before the timer expires, `read` returns MIN bytes. If the timer expires before MIN bytes are received, `read` returns the bytes received. (At least one byte is returned if the timer expires, because the timer is not started until the first byte is received.) In this case, the caller blocks until the first byte is received. If data is already available when `read` is called, it is as if the data had been received immediately after the `read`.

Case B: MIN > 0, TIME == 0

The `read` does not return until MIN bytes have been received. This can cause a `read` to block indefinitely.

Case C: MIN == 0, TIME > 0

TIME specifies a read timer that is started when `read` is called. (Compare this to case A, in which a nonzero TIME represented an interbyte timer that was not started until the first byte was received.) The `read` returns when a single byte is received or when the timer expires. If the timer expires, `read` returns 0.

Case D: MIN == 0, TIME == 0

If some data is available, `read` returns up to the number of bytes requested. If no data is available, `read` returns 0 immediately.

Realize in all these cases that MIN is only a minimum. If the program requests more than MIN bytes of data, it's possible to receive up to the requested amount. This also applies to cases C and D, in which MIN is 0.

Figure 18.19 summarizes the four cases for noncanonical input. In this figure, *nbytes* is the third argument to `read` (the maximum number of bytes to return).

	MIN > 0	MIN == 0
TIME > 0	**A:** `read` returns [MIN, *nbytes*] before timer expires; `read` returns [1, MIN) if timer expires. (TIME = interbyte timer. Caller can block indefinitely.)	**C:** `read` returns [1, *nbytes*] before timer expires; `read` returns 0 if timer expires. (TIME = `read` timer.)
TIME == 0	**B**: `read` returns [MIN, *nbytes*] when available. (Caller can block indefinitely.)	**D**: `read` returns [0, *nbytes*] immediately.

Figure 18.19 Four cases for noncanonical input

Be aware that POSIX.1 allows the subscripts `VMIN` and `VTIME` to have the same values as `VEOF` and `VEOL`, respectively. Indeed, Solaris does this for backward compatibility with older versions of System V. This creates a portability problem, however. In going from noncanonical to canonical mode, we must now restore `VEOF` and `VEOL` as well. If `VMIN` equals `VEOF` and we don't restore their values, when we set `VMIN` to its typical value of 1, the end-of-file character becomes Control-A. The easiest way around this problem is to save the entire `termios` structure when going into noncanonical mode and restore it when going back to canonical mode.

Example

The program in Figure 18.20 defines the `tty_cbreak` and `tty_raw` functions that set the terminal in *cbreak mode* and *raw mode*. (The terms *cbreak* and *raw* come from the Version 7 terminal driver.) We can reset the terminal to its original state (the state before either of these functions was called) by calling the function `tty_reset`.

If we've called `tty_cbreak`, we need to call `tty_reset` before calling `tty_raw`. The same goes for calling `tty_cbreak` after calling `tty_raw`. This improves the chances that the terminal will be left in a usable state if we encounter any errors.

Two additional functions are provided: `tty_atexit` can be established as an exit handler to ensure that the terminal mode is reset by `exit`, and `tty_termios` returns a pointer to the original canonical mode `termios` structure.

```
#include "apue.h"
#include <termios.h>
#include <errno.h>

static struct termios       save_termios;
static int                  ttysavefd = -1;
static enum { RESET, RAW, CBREAK }  ttystate = RESET;

int
tty_cbreak(int fd)  /* put terminal into a cbreak mode */
{
    int             err;
    struct termios  buf;

    if (ttystate != RESET) {
        errno = EINVAL;
        return(-1);
    }
    if (tcgetattr(fd, &buf) < 0)
        return(-1);
    save_termios = buf; /* structure copy */

    /*
     * Echo off, canonical mode off.
     */
    buf.c_lflag &= ~(ECHO | ICANON);

    /*
     * Case B: 1 byte at a time, no timer.
     */
    buf.c_cc[VMIN] = 1;
    buf.c_cc[VTIME] = 0;
    if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
        return(-1);

    /*
     * Verify that the changes stuck.  tcsetattr can return 0 on
     * partial success.
     */
    if (tcgetattr(fd, &buf) < 0) {
        err = errno;
        tcsetattr(fd, TCSAFLUSH, &save_termios);
        errno = err;
        return(-1);
    }
    if ((buf.c_lflag & (ECHO | ICANON)) || buf.c_cc[VMIN] != 1 ||
      buf.c_cc[VTIME] != 0) {
```

```
        /*
         * Only some of the changes were made.  Restore the
         * original settings.
         */
        tcsetattr(fd, TCSAFLUSH, &save_termios);
        errno = EINVAL;
        return(-1);
    }

    ttystate = CBREAK;
    ttysavefd = fd;
    return(0);
}

int
tty_raw(int fd)     /* put terminal into a raw mode */
{
    int             err;
    struct termios  buf;

    if (ttystate != RESET) {
        errno = EINVAL;
        return(-1);
    }
    if (tcgetattr(fd, &buf) < 0)
        return(-1);
    save_termios = buf; /* structure copy */

    /*
     * Echo off, canonical mode off, extended input
     * processing off, signal chars off.
     */
    buf.c_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);

    /*
     * No SIGINT on BREAK, CR-to-NL off, input parity
     * check off, don't strip 8th bit on input, output
     * flow control off.
     */
    buf.c_iflag &= ~(BRKINT | ICRNL | INPCK | ISTRIP | IXON);

    /*
     * Clear size bits, parity checking off.
     */
    buf.c_cflag &= ~(CSIZE | PARENB);

    /*
     * Set 8 bits/char.
     */
    buf.c_cflag |= CS8;

    /*
```

```
         * Output processing off.
         */
        buf.c_oflag &= ~(OPOST);

        /*
         * Case B: 1 byte at a time, no timer.
         */
        buf.c_cc[VMIN] = 1;
        buf.c_cc[VTIME] = 0;
        if (tcsetattr(fd, TCSAFLUSH, &buf) < 0)
            return(-1);

        /*
         * Verify that the changes stuck.  tcsetattr can return 0 on
         * partial success.
         */
        if (tcgetattr(fd, &buf) < 0) {
            err = errno;
            tcsetattr(fd, TCSAFLUSH, &save_termios);
            errno = err;
            return(-1);
        }
        if ((buf.c_lflag & (ECHO | ICANON | IEXTEN | ISIG)) ||
          (buf.c_iflag & (BRKINT | ICRNL | INPCK | ISTRIP | IXON)) ||
          (buf.c_cflag & (CSIZE | PARENB | CS8)) != CS8 ||
          (buf.c_oflag & OPOST) || buf.c_cc[VMIN] != 1 ||
          buf.c_cc[VTIME] != 0) {
            /*
             * Only some of the changes were made.  Restore the
             * original settings.
             */
            tcsetattr(fd, TCSAFLUSH, &save_termios);
            errno = EINVAL;
            return(-1);
        }

        ttystate = RAW;
        ttysavefd = fd;
        return(0);
    }

    int
    tty_reset(int fd)       /* restore terminal's mode */
    {
        if (ttystate == RESET)
            return(0);
        if (tcsetattr(fd, TCSAFLUSH, &save_termios) < 0)
            return(-1);
        ttystate = RESET;
        return(0);
    }
```

```
void
tty_atexit(void)        /* can be set up by atexit(tty_atexit) */
{
    if (ttysavefd >= 0)
        tty_reset(ttysavefd);
}

struct termios *
tty_termios(void)       /* let caller see original tty state */
{
    return(&save_termios);
}
```

Figure 18.20 Set terminal mode to cbreak or raw

Our definition of cbreak mode is the following:

- Noncanonical mode. As we mentioned at the beginning of this section, this mode turns off some input character processing. It does not turn off signal handling, so the user can always type one of the characters that triggers a terminal-generated signal. Be aware that the caller should catch these signals; otherwise, there's a chance that the signal will terminate the program, and the terminal will be left in cbreak mode.

  As a general rule, whenever we write a program that changes the terminal mode, we should catch most signals. This allows us to reset the terminal mode before terminating.
- Echo off.
- One byte at a time input. To do this, we set MIN to 1 and TIME to 0. This is case B from Figure 18.19. A `read` won't return until at least one byte is available.

We define raw mode as follows:

- Noncanonical mode. We also turn off processing of the signal-generating characters (`ISIG`) and the extended input character processing (`IEXTEN`). Additionally, we disable a BREAK character from generating a signal, by turning off `BRKINT`.
- Echo off.
- We disable the CR-to-NL mapping on input (`ICRNL`), input parity detection (`INPCK`), the stripping of the eighth bit on input (`ISTRIP`), and output flow control (`IXON`).
- Eight-bit characters (`CS8`), and parity checking is disabled (`PARENB`).
- All output processing is disabled (`OPOST`).
- One byte at a time input (MIN = 1, TIME = 0).

The program in Figure 18.21 tests raw and cbreak modes.

```
#include "apue.h"

static void
sig_catch(int signo)
{
    printf("signal caught\n");
    tty_reset(STDIN_FILENO);
    exit(0);
}

int
main(void)
{
    int     i;
    char    c;

    if (signal(SIGINT, sig_catch) == SIG_ERR)   /* catch signals */
        err_sys("signal(SIGINT) error");
    if (signal(SIGQUIT, sig_catch) == SIG_ERR)
        err_sys("signal(SIGQUIT) error");
    if (signal(SIGTERM, sig_catch) == SIG_ERR)
        err_sys("signal(SIGTERM) error");

    if (tty_raw(STDIN_FILENO) < 0)
        err_sys("tty_raw error");
    printf("Enter raw mode characters, terminate with DELETE\n");
    while ((i = read(STDIN_FILENO, &c, 1)) == 1) {
        if ((c &= 255) == 0177)     /* 0177 = ASCII DELETE */
            break;
        printf("%o\n", c);
    }
    if (tty_reset(STDIN_FILENO) < 0)
        err_sys("tty_reset error");
    if (i <= 0)
        err_sys("read error");
    if (tty_cbreak(STDIN_FILENO) < 0)
        err_sys("tty_cbreak error");
    printf("\nEnter cbreak mode characters, terminate with SIGINT\n");
    while ((i = read(STDIN_FILENO, &c, 1)) == 1) {
        c &= 255;
        printf("%o\n", c);
    }
    if (tty_reset(STDIN_FILENO) < 0)
        err_sys("tty_reset error");
    if (i <= 0)
        err_sys("read error");

    exit(0);
}
```

Figure 18.21 Test raw and cbreak terminal modes

Running the program in Figure 18.21, we can see what happens with these two terminal modes:

```
$ ./a.out
Enter raw mode characters, terminate with DELETE
                                                    4
                                                     33
                                                       133
                                                          61
                                                            70
                                                              176
                                 type DELETE
Enter cbreak mode characters, terminate with SIGINT
1                                type Control-A
10                               type backspace
signal caught                    type interrupt key
```

In raw mode, the characters entered were Control-D (04) and the special function key F7. On the terminal being used, this function key generated five characters: *ESC* (033), *[* (0133), *1* (061), *8* (070), and *~* (0176). Note that with the output processing turned off in raw mode (`~OPOST`), we do not get a carriage return output after each character. Also note that special-character processing is disabled in cbreak mode (so, for example, Control-D, the end-of-file character, and backspace aren't handled specially), whereas the terminal-generated signals are still processed. □

18.12 Terminal Window Size

Most UNIX systems provide a way to keep track of the current terminal window size and to have the kernel notify the foreground process group when the size changes. The kernel maintains a `winsize` structure for every terminal and pseudo terminal:

```
struct winsize {
  unsigned short  ws_row;     /* rows, in characters */
  unsigned short  ws_col;     /* columns, in characters */
  unsigned short  ws_xpixel;  /* horizontal size, pixels (unused) */
  unsigned short  ws_ypixel;  /* vertical size, pixels (unused) */
};
```

The rules for this structure are as follows:

- We can fetch the current value of this structure using an `ioctl` (Section 3.15) of `TIOCGWINSZ`.
- We can store a new value of this structure in the kernel using an `ioctl` of `TIOCSWINSZ`. If this new value differs from the current value stored in the kernel, a `SIGWINCH` signal is sent to the foreground process group. (Note from Figure 10.1 that the default action for this signal is to be ignored.)

- Other than storing the current value of the structure and generating a signal when the value changes, the kernel does nothing else with this structure. Interpreting the structure is entirely up to the application.

This feature is provided to notify applications (such as the `vi` editor) when the window size changes. When it receives the signal, the application can fetch the new size and redraw the screen.

Example

Figure 18.22 shows a program that prints the current window size and goes to sleep. Each time the window size changes, `SIGWINCH` is caught and the new size is printed. We have to terminate this program with a signal.

```
#include "apue.h"
#include <termios.h>
#ifndef TIOCGWINSZ
#include <sys/ioctl.h>
#endif

static void
pr_winsize(int fd)
{
    struct winsize  size;

    if (ioctl(fd, TIOCGWINSZ, (char *) &size) < 0)
        err_sys("TIOCGWINSZ error");
    printf("%d rows, %d columns\n", size.ws_row, size.ws_col);
}

static void
sig_winch(int signo)
{
    printf("SIGWINCH received\n");
    pr_winsize(STDIN_FILENO);
}

int
main(void)
{
    if (isatty(STDIN_FILENO) == 0)
        exit(1);
    if (signal(SIGWINCH, sig_winch) == SIG_ERR)
        err_sys("signal error");
    pr_winsize(STDIN_FILENO);   /* print initial size */
    for ( ; ; )                 /* and sleep forever */
        pause();
}
```

Figure 18.22 Print window size

Running the program in Figure 18.22 on a windowed terminal gives us

```
$ ./a.out
35 rows, 80 columns          initial size
SIGWINCH received            change window size: signal is caught
40 rows, 123 columns
SIGWINCH received            and again
42 rows, 33 columns
^C $                         type the interrupt key to terminate
```

□

18.13 termcap, terminfo, and curses

`termcap` stands for "terminal capability," and it refers to the text file `/etc/termcap` and a set of routines used to read this file. The `termcap` scheme was developed at Berkeley to support the `vi` editor. The `termcap` file contains descriptions of various terminals: which features the terminal supports (e.g., how many lines and rows, whether the terminal support backspace) and how to make the terminal perform certain operations (e.g., clear the screen, move the cursor to a given location). Taking this information out of the compiled program and placing it into a text file that can easily be edited allows the `vi` editor to run on many different terminals.

The routines that support the `termcap` file were eventually extracted from the `vi` editor and placed into a separate `curses` library. Many features were added to make this library usable for any program that wanted to manipulate the screen.

The `termcap` scheme was not perfect. As more and more terminals were added to the data file, it took longer to scan the file, looking for a specific terminal. The data file also used two-character names to identify the various terminal attributes. These deficiencies led to development of the `terminfo` scheme and its associated `curses` library. The terminal descriptions in `terminfo` are basically compiled versions of a textual description and can be located faster at runtime. `terminfo` appeared with SVR2 and has been included in all System V releases since then.

> Historically, System V–based systems used `terminfo`, and BSD-derived systems used `termcap`, but it is now common for systems to provide both. Mac OS X, however, supports only `terminfo`.

A description of `terminfo` and the `curses` library is provided by Goodheart [1991], but this book is currently out of print. Strang [1986] describes the Berkeley version of the `curses` library. Strang, Mui, and O'Reilly [1988] provide a description of `termcap` and `terminfo`.

> The `ncurses` library, a free version that is compatible with the SVR4 `curses` interface, can be found at `http://invisible-island.net/ncurses/ncurses.html`. It can also be found at `http://www.gnu.org/software/ncurses`.

Neither `termcap` nor `terminfo`, by itself, addresses the problems we've been looking at in this chapter: changing the terminal's mode, changing one of the terminal special characters, handling the window size, and so on. What they do provide is a way

to perform typical operations (clear the screen, move the cursor) on a wide variety of terminals. On the other hand, `curses` does help with some of the details that we've addressed in this chapter. Functions are provided by `curses` to set raw mode, set cbreak mode, turn echo on and off, and the like. Note that the `curses` library is designed for character-based dumb terminals, which have mostly been replaced by pixel-based graphics terminals today.

18.14 Summary

Terminals have many features and options, most of which we're able to change to suit our needs. In this chapter, we described numerous functions that change a terminal's operation—namely, special input characters and the option flags. We also looked at all the terminal special characters and the many options that can be set or reset for a terminal device.

There are two modes of terminal input—canonical (line at a time) and noncanonical. We showed examples of both modes and provided functions that map between the POSIX.1 terminal options and the older BSD cbreak and raw modes. We also described how to fetch and change the window size of a terminal.

Exercises

18.1 Write a program that calls `tty_raw` and terminates (without resetting the terminal mode). If your system provides the `reset`(1) command (all four systems described in this text do), use it to restore the terminal mode.

18.2 The `PARODD` flag in the `c_cflag` field allows us to specify even or odd parity. The BSD `tip` program, however, also allows the parity bit to be 0 or 1. How does it do this?

18.3 If your system's `stty`(1) command outputs the MIN and TIME values, do the following exercise. Log in to the system twice and start the `vi` editor from one login. Use the `stty` command from your other login to determine which values `vi` sets MIN and TIME to (since `vi` sets the terminal to noncanonical mode). (If you are running a windowing system on your terminal, you can do this same test by logging in once and using two separate windows instead.)

This page intentionally left blank

19

Pseudo Terminals

19.1 Introduction

In Chapter 9, we saw that terminal logins come in through a terminal device, automatically providing terminal semantics. A terminal line discipline (Figure 18.2) exists between the terminal and the programs that we run, so we can set the terminal's special characters (e.g., backspace, line erase, interrupt) and the like. When a login arrives on a network connection, however, a terminal line discipline is not automatically provided between the incoming network connection and the login shell. Figure 9.5 showed that a *pseudo terminal* device driver is used to provide terminal semantics.

In addition to network logins, pseudo terminals have other uses that we explore in this chapter. We start with an overview on how to use pseudo terminals, followed by a discussion of specific use cases. Next we provide functions to create pseudo terminals on various platforms, and then we use these functions to write a program that we call `pty`. We'll show various uses of this program: making a transcript of all the character input and output on the terminal (the `script`(1) program) and running coprocesses to avoid the buffering problems we encountered in the program from Figure 15.19.

19.2 Overview

The term *pseudo terminal* implies that it looks like a terminal to an application program, but it's not a real terminal. Figure 19.1 shows the typical arrangement of the processes involved when a pseudo terminal is being used. The key points in this figure are the following.

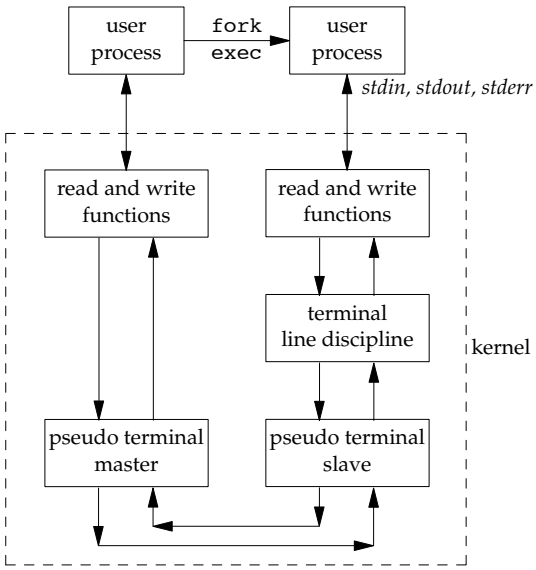


Figure 19.1 Typical arrangement of processes using a pseudo terminal

- Normally, a process opens the pseudo terminal master and then calls `fork`. The child establishes a new session, opens the corresponding pseudo terminal slave, duplicates the file descriptor to the standard input, standard output, and standard error, and then calls `exec`. The pseudo terminal slave becomes the controlling terminal for the child process.
- It appears to the user process above the slave that its standard input, standard output, and standard error are a terminal device. The process can issue all the terminal I/O functions from Chapter 18 on these descriptors. But since the slave isn't a real terminal device, functions that don't make sense (e.g., change the baud rate, send a break character, set odd parity) are just ignored.
- Anything written to the master appears as input to the slave, and vice versa. Indeed, all the input to the slave comes from the user process above the pseudo terminal master. This behaves like a bidirectional pipe, but with the terminal line discipline module above the slave, we have additional capabilities over a plain pipe.

Figure 19.1 shows what a pseudo terminal looks like on a FreeBSD, Mac OS X, or Linux system. In Section 19.3, we show how to open these devices.

Under Solaris, a pseudo terminal is built using the STREAMS subsystem. Figure 19.2 details the arrangement of the pseudo terminal STREAMS modules under Solaris. The two STREAMS modules that are shown as dashed boxes are optional. The `pckt` and `ptem` modules help provide semantics specific to pseudo terminals. The other two modules (`ldterm` and `ttcompat`) provide line discipline processing. In Section 19.3, we show how to build this arrangement of STREAMS modules.

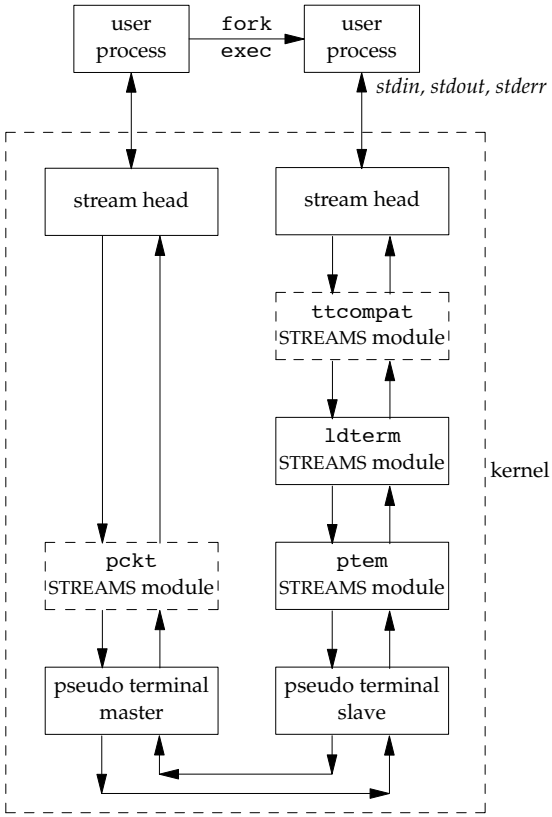


Figure 19.2 Arrangement of pseudo terminals under Solaris

From this point on, we'll simplify the figures by not showing the "read and write functions" from Figure 19.1 or the "stream head" from Figure 19.2. We'll also use the abbreviation PTY for pseudo terminal and lump all the STREAMS modules above the slave PTY in Figure 19.2 into a box called "terminal line discipline," as in Figure 19.1.

We'll now examine some of the typical uses of pseudo terminals.

Network Login Servers

Pseudo terminals are built into servers that provide network logins. The typical examples are the `telnetd` and `rlogind` servers. Chapter 15 of Stevens [1990] details the steps involved in the `rlogin` service. Once the login shell is running on the remote host, we have the arrangement shown in Figure 19.3. A similar arrangement is used by the `telnetd` server.

We show two calls to `exec` between the `rlogind` server and the login shell, because the `login` program is usually between the two to validate the user.

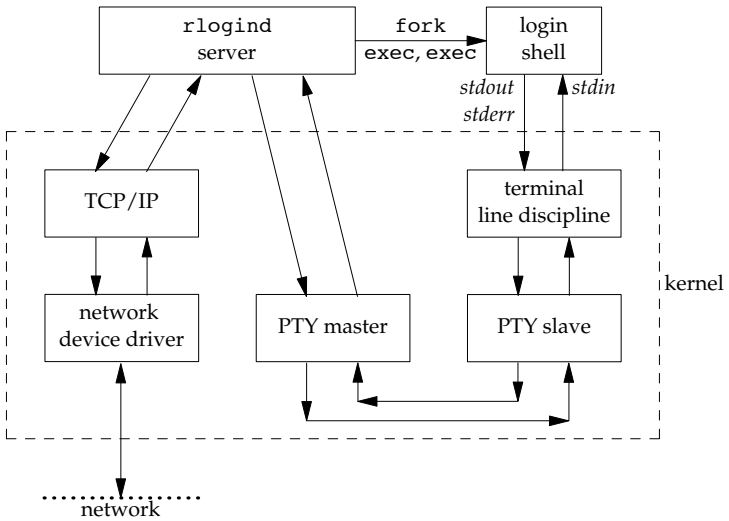


Figure 19.3 Arrangement of processes for `rlogind` server

A key point in Figure 19.3 is that the process driving the PTY master is normally reading and writing another I/O stream at the same time. In this example, the other I/O stream is the TCP/IP box. This implies that the process must be using some form of I/O multiplexing (Section 14.4), such as `select` or `poll`, or must be divided into two processes or threads.

Windowing System Terminal Emulation

Windowing systems typically provide a terminal emulator so that we can use a shell to run our programs from a familiar command-line environment. The terminal emulator acts as an intermediary between a shell and the window manager. Each shell executes in its own window. This arrangement (with two shells running in different windows) is shown in Figure 19.4.

The shell runs with its standard input, standard output, and standard error attached to the slave side of the PTY. The terminal emulator program opens the master side of the PTY. Besides acting as an interface to the windowing subsystem, the terminal emulator is responsible for emulating a particular type of terminal, which means it needs to respond to the escape codes associated with the type of device it is emulating. These codes are listed in the `termcap` and `terminfo` databases.

When a user resizes a terminal emulator window, the window manager informs the terminal emulator. The terminal emulator issues a `TIOCSWINSZ ioctl` command on the master side of the PTY to set the window size for the slave side. If the new size differs from the current size, the kernel sends a `SIGWINCH` signal to the foreground

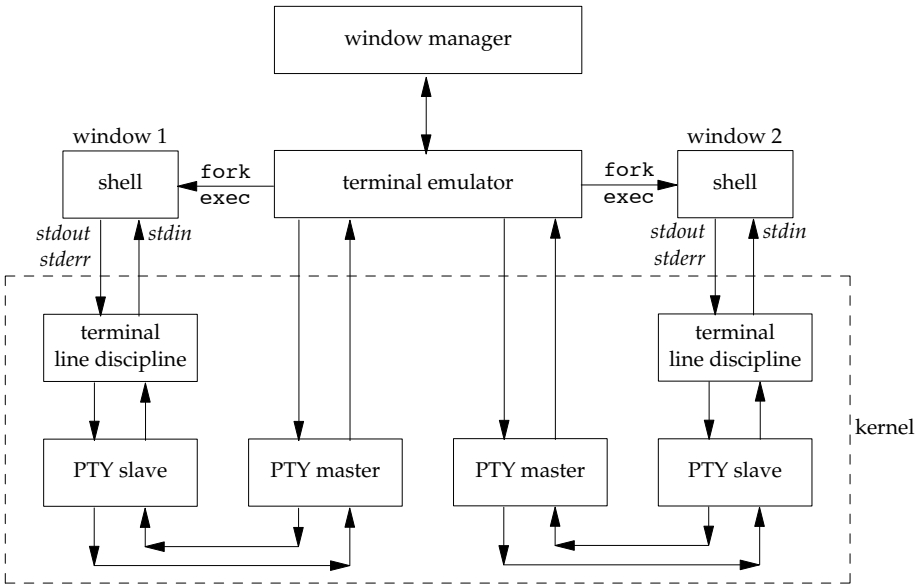


Figure 19.4 Arrangement of processes for windowing system

process group of the PTY slave. If the application needs to redraw the screen when the window is resized, it can catch the `SIGWINCH` signal, issue the `TIOCGWINSZ ioctl` command to get the new screen dimensions, and redraw the screen.

`script` Program

The `script`(1) program that is supplied with most UNIX systems makes a copy in a file of everything that is input and output during a terminal session. The program does this by placing itself between the terminal and a new invocation of our login shell. Figure 19.5 details the interactions involved in the `script` program. Here, we specifically show that the `script` program is normally run from a login shell, which then waits for `script` to terminate.

While `script` is running, everything output by the terminal line discipline above the PTY slave is copied to the script file (usually called `typescript`). Since our keystrokes are normally echoed by that line discipline module, the script file also contains our input. The script file won't contain any passwords that we enter, however, since passwords aren't echoed.

> While writing the first edition of this book, Rich Stevens used the `script` program to capture the output of the example programs. This avoided typographical errors that could have occurred if he had copied the program output by hand. The drawback to using `script`, however, is having to deal with control characters that are present in the script file.

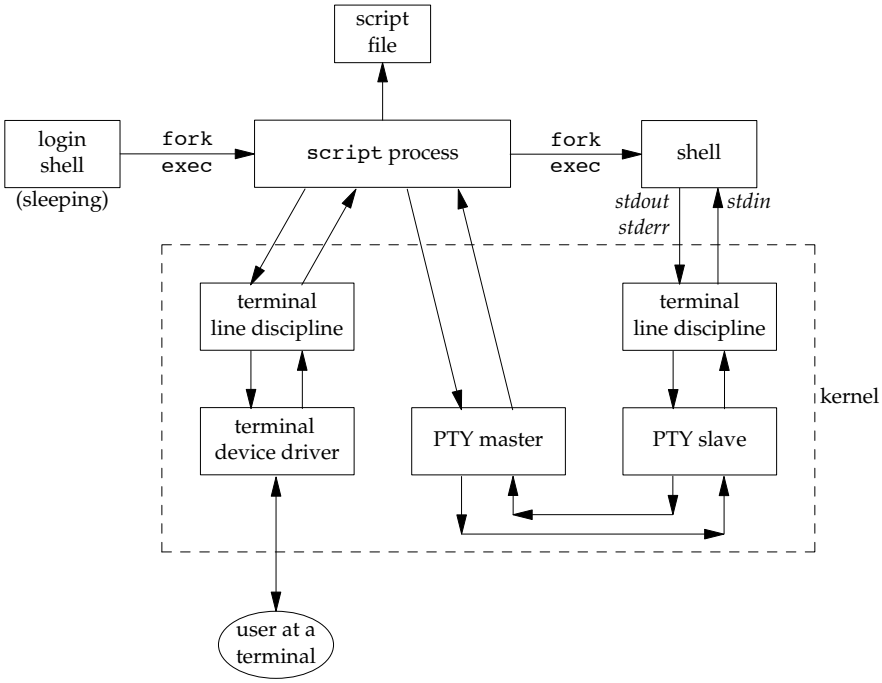


Figure 19.5 The `script` program

After developing the general `pty` program in Section 19.5, we'll see that a trivial shell script turns it into a version of the `script` program.

`expect` Program

Pseudo terminals can be used to drive interactive programs in noninteractive modes. Numerous programs are hard-wired to require a terminal to run. One example is the `passwd`(1) command, which requires that the user enter a password in response to a prompt.

Rather than modify all the interactive programs to support a batch mode of operation, a better solution is to provide a way to drive any interactive program from a script. The `expect` program [Libes 1990, 1991, 1994] provides a way to do this. It uses pseudo terminals to run other programs, similar to the `pty` program in Section 19.5. But `expect` also provides a programming language to examine the output of the program being run to make decisions about what to send the program as input. When an interactive program is being run from a script, we can't just copy everything from the script to the program, and vice versa. Instead, we have to send the program some input, look at its output, and decide what to send it next.

Running Coprocesses

In the coprocess example in Figure 15.19, we couldn't invoke a coprocess that used the standard I/O library for its input and output, because when we talked to the coprocess across a pipe, the standard I/O library fully buffered the standard input and standard output, leading to a deadlock. If the coprocess is a compiled program for which we don't have the source code, we can't add `fflush` statements to solve this problem. Figure 15.16 showed a process driving a coprocess. What we need to do is place a pseudo terminal between the two processes, as shown in Figure 19.6, to trick the coprocess into thinking that it is being driven from a terminal instead of from another process.

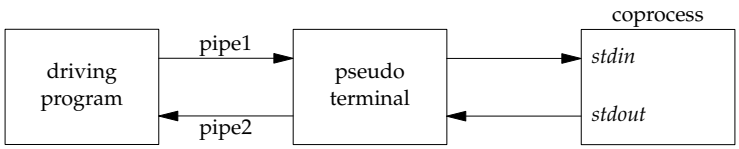


Figure 19.6 Driving a coprocess using a pseudo terminal

Now the standard input and standard output of the coprocess look like a terminal device, so the standard I/O library will set these two streams to be line buffered.

The parent can obtain a pseudo terminal between itself and the coprocess in two ways. (The parent in this case could be the program in Figure 15.18, which used two pipes to communicate with the coprocess.) One way is for the parent to call the `pty_fork` function directly (Section 19.4) instead of calling `fork`. Another is to `exec` the `pty` program (Section 19.5) with the coprocess as its argument. We'll look at these two solutions after showing the `pty` program.

Watching the Output of Long-Running Programs

If we have a program that runs for a long time, we can easily run it in the background using any of the standard shells. Unfortunately, if we redirect its standard output to a file, and if it doesn't generate much output, we can't easily monitor its progress, because the standard I/O library will fully buffer its standard output. All that we'll see are blocks of output written by the standard I/O library to the output file, possibly in chunks as large as 8,192 bytes.

If we have the source code, we can insert calls to `fflush` to force the standard I/O buffers to be flushed at select points or change the buffering mode to line buffered using `setvbuf`. If we don't have the source code, however, we can run the program under the `pty` program, making its standard I/O library think that its standard output is a terminal. Figure 19.7 shows this arrangement, where we have called the slow output program `slowout`. The `fork/exec` arrow from the login shell to the `pty` process is shown as a dashed arrow to emphasize that the `pty` process is running as a background job.

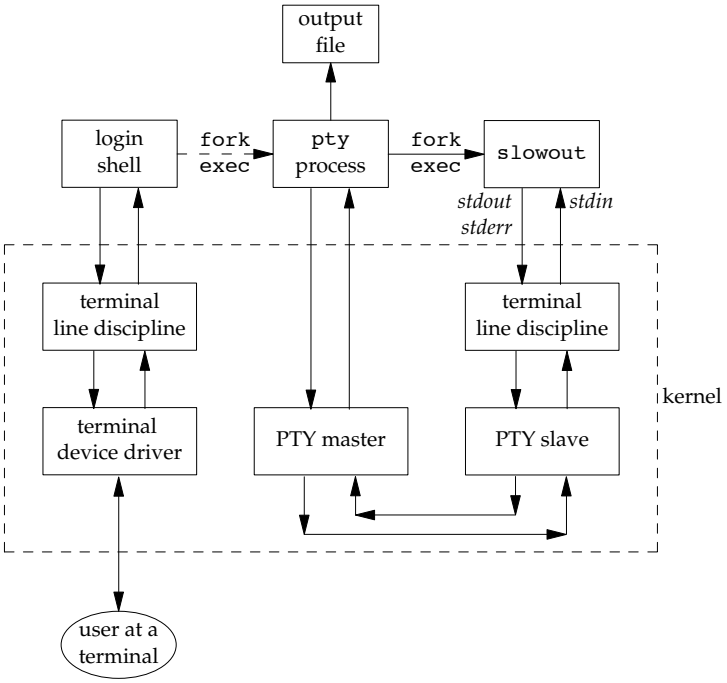


Figure 19.7 Running a slow output program using a pseudo terminal

19.3 Opening Pseudo-Terminal Devices

PTYs act like physical terminal devices so that applications are unaware of which type of device they are using. However, applications don't need to set the `O_TTY_INIT` flag when opening PTY device files. The Single UNIX Specification already requires that implementations initialize the slave side of a PTY device when it is first opened so that any nonstandard `termios` flags needed for the device to operate as expected are set. This requirement is intended to allow the PTY device to operate properly with POSIX-conforming applications that call `tcgetattr` and `tcsetattr`.

The way we open a PTY device differs among platforms. The Single UNIX Specification includes several functions as part of the XSI option in an attempt to unify the methods. These extensions are based on the functions originally provided to manage STREAMS-based PTYs in System V Release 4. The `posix_openpt` function is provided as a portable way to open an available PTY master device.

```
#include <stdlib.h>
#include <fcntl.h>

int posix_openpt(int oflag);
```

Returns: file descriptor of next available PTY master if OK, –1 on error

The *oflag* argument is a bitmask that specifies how the master device is to be opened, similar to the same argument used with open(2). Not all open flags are supported, however. With posix_openpt, we can specify O_RDWR to open the master device for reading and writing, and we can specify O_NOCTTY to prevent the master device from becoming a controlling terminal for the caller. All other open flags result in unspecified behavior.

Before a slave pseudo terminal device can be used, its permissions need to be set so that it is accessible to applications. The grantpt function does just this. It sets the user ID of the slave's device node to the caller's real user ID and sets the node's group ID to an unspecified value, usually some group that has access to terminal devices. The permissions are set to allow read and write access to individual owners and write access to group owners (0620).

Implementations commonly set the group ownership of the slave PTY device to group tty. Programs that need permission to write to all active terminals on the system are set-group-ID to the group tty. Examples of such programs are wall(1) and write(1). Because the group write permission is enabled on slave PTY devices, these programs can write to them.

```
#include <stdlib.h>

int grantpt(int fd);

int unlockpt(int fd);
```

Both return: 0 on success, –1 on error

To change permission on the slave device node, grantpt might need to fork and exec a set-user-ID program (/usr/lib/pt_chmod on Solaris, for example). Thus, the behavior is unspecified if the caller is catching SIGCHLD.

The unlockpt function is used to grant access to the slave pseudo terminal device, thereby allowing applications to open the device. By preventing others from opening the slave device, applications setting up the devices have an opportunity to initialize the slave and master devices properly before they can be used.

Note that in both grantpt and unlockpt, the file descriptor argument is the file descriptor associated with the master pseudo terminal device.

The ptsname function is used to find the pathname of the slave pseudo terminal device, given the file descriptor of the master. This allows applications to identify the slave independent of any particular conventions that might be followed by a given platform. Note that the name returned might be stored in static memory, so it can be overwritten on successive calls.

```
#include <stdlib.h>

char *ptsname(int fd);
```

Returns: pointer to name of PTY slave if OK, NULL on error

Figure 19.8 summarizes the pseudo terminal functions in the Single UNIX Specification and indicates which functions are supported by the platforms discussed in this text.

On FreeBSD, `grantpt` and `unlockpt` do nothing other than argument validation; the PTYs are created dynamically with the correct permissions. Note that FreeBSD defines the `O_NOCTTY` flag only for compatibility with applications that call `posix_openpt`. FreeBSD does not allocate a controlling terminal as a side effect of opening a terminal device, so the `O_NOCTTY` flag has no effect.

Function	Description	XSI	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
`grantpt`	change permissions of slave PTY device	•	•	•	•	•
`posix_openpt`	open a master PTY device	•	•	•	•	•
`ptsname`	return name of slave PTY device	•	•	•	•	•
`unlockpt`	allow slave PTY device to be opened	•	•	•	•	•

Figure 19.8 XSI pseudo terminal functions

The Single UNIX Specification has improved portability in this area, but differences remain. We provide two functions that handle all the details: `ptym_open` to open the next available PTY master device and `ptys_open` to open the corresponding slave device.

```
#include "apue.h"

int ptym_open(char *pts_name, int pts_namesz);
```

Returns: file descriptor of PTY master if OK, –1 on error

```
int ptys_open(char *pts_name);
```

Returns: file descriptor of PTY slave if OK, –1 on error

Normally, we don't call these two functions directly; instead, the function `pty_fork` (Section 19.4) calls them and also `fork`s a child process.

The `ptym_open` function opens the next available PTY master. The caller must allocate an array to hold the name of the slave; if the call succeeds, the name of the corresponding slave is returned through *pts_name*. This name is then passed to `ptys_open`, which opens the slave device. The length of the buffer in bytes is passed in *pts_namesz* so that the `ptym_open` function doesn't copy a string that is longer than the buffer.

The reason for providing two functions to open the two devices will become obvious when we show the `pty_fork` function. Normally, a process calls `ptym_open` to open the master and obtain the name of the slave. The process then `fork`s, and the child calls `ptys_open` to open the slave after calling `setsid` to establish a new session. This is how the slave becomes the controlling terminal for the child.

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#if defined(SOLARIS)
#include <stropts.h>
#endif
```

```
int
ptym_open(char *pts_name, int pts_namesz)
{
    char    *ptr;
    int     fdm, err;

    if ((fdm = posix_openpt(O_RDWR)) < 0)
        return(-1);
    if (grantpt(fdm) < 0)       /* grant access to slave */
        goto errout;
    if (unlockpt(fdm) < 0)      /* clear slave's lock flag */
        goto errout;
    if ((ptr = ptsname(fdm)) == NULL)   /* get slave's name */
        goto errout;

    /*
     * Return name of slave.  Null terminate to handle
     * case where strlen(ptr) > pts_namesz.
     */
    strncpy(pts_name, ptr, pts_namesz);
    pts_name[pts_namesz - 1] = '\0';
    return(fdm);            /* return fd of master */
errout:
    err = errno;
    close(fdm);
    errno = err;
    return(-1);
}

int
ptys_open(char *pts_name)
{
    int fds;
#if defined(SOLARIS)
    int err, setup;
#endif

    if ((fds = open(pts_name, O_RDWR)) < 0)
        return(-1);

#if defined(SOLARIS)
    /*
     * Check if stream is already set up by autopush facility.
     */
    if ((setup = ioctl(fds, I_FIND, "ldterm")) < 0)
        goto errout;

    if (setup == 0) {
        if (ioctl(fds, I_PUSH, "ptem") < 0)
            goto errout;
        if (ioctl(fds, I_PUSH, "ldterm") < 0)
            goto errout;
```

```
        if (ioctl(fds, I_PUSH, "ttcompat") < 0) {
errout:
            err = errno;
            close(fds);
            errno = err;
            return(-1);
        }
    }
#endif
    return(fds);
}
```

Figure 19.9 Pseudo-terminal open functions

The `ptym_open` function uses the XSI PTY functions to find and open an unused PTY master device and initialize the corresponding PTY slave device. The `ptys_open` function opens the slave PTY device. On a Solaris system, however, we might need to take additional steps before the slave PTY device will behave like a terminal.

On Solaris, after opening the slave device, we might need to push three STREAMS modules onto the slave's stream. Together, the pseudo terminal emulation module (`ptem`) and the terminal line discipline module (`ldterm`) act like a real terminal. The `ttcompat` module provides compatibility for older V7, 4BSD, and Xenix `ioctl` calls. It's an optional module, but since it's automatically pushed for network logins, we push it onto the slave's stream.

The reason why we might *not* need to push these three modules is that they might be there already. The STREAMS system supports a facility known as *autopush*, which allows an administrator to configure a list of modules to be pushed onto a stream whenever a particular device is opened (see Rago [1993] for more details). We use the `I_FIND` `ioctl` command to see whether `ldterm` is already present on the stream. If so, we assume that the stream has been configured by the autopush mechanism and avoid pushing the modules a second time.

Linux, Mac OS X, and Solaris follow the historical System V behavior: if the caller is a session leader that does not already have a controlling terminal, the call to `open` allocates the PTY slave as the controlling terminal. If we didn't want this to happen, we could specify the `O_NOCTTY` flag for `open`. However, on FreeBSD, the `open` of the slave PTY does not have the side effect of allocating the device as the controlling terminal. In the next section, we'll see how to allocate the controlling terminal when running on FreeBSD.

19.4 `pty_fork` Function

We now use the two functions from the previous section, `ptym_open` and `ptys_open`, to write a new function that we call `pty_fork`. This new function combines the opening of the master and the slave with a call to `fork`, establishing the child as a session leader with a controlling terminal.

```
#include "apue.h"
#include <termios.h>

pid_t pty_fork(int *ptrfdm, char *slave_name, int slave_namesz,
               const struct termios *slave_termios,
               const struct winsize *slave_winsize);
```

Returns: 0 in child, process ID of child in parent, –1 on error

The file descriptor of the PTY master is returned through the *ptrfdm* pointer.

If *slave_name* is non-null, the name of the slave device is stored at that location. The caller is responsible for allocating the storage pointed to by this argument.

If the pointer *slave_termios* is non-null, the system uses the referenced structure to initialize the terminal line discipline of the slave. If this pointer is null, the system sets the slave's `termios` structure to an implementation-defined initial state. Similarly, if the *slave_winsize* pointer is non-null, the referenced structure initializes the slave's window size. If this pointer is null, the `winsize` structure is normally initialized to 0.

Figure 19.10 shows the code for this function. It works on all four platforms described in this text, calling the `ptym_open` and `ptys_open` functions.

After opening the PTY master, `fork` is called. As we mentioned before, we want to wait to call `ptys_open` until in the child and after calling `setsid` to establish a new session. When it calls `setsid`, the child is not a process group leader, so the three steps listed in Section 9.5 occur: (a) a new session is created with the child as the session leader, (b) a new process group is created for the child, and (c) the child loses any association it might have had with its previous controlling terminal. Under Linux, Mac OS X, and Solaris, the slave becomes the controlling terminal of this new session when `ptys_open` is called. Under FreeBSD, we have to use the `TIOCSCTTY ioctl` command to allocate the controlling terminal. (Recall Figure 9.8—the other three platforms also support `TIOCSCTTY`, but we need to call it only on FreeBSD.)

The two structures `termios` and `winsize` are then initialized in the child. Finally, the slave file descriptor is duplicated onto standard input, standard output, and standard error in the child. This means that whatever process the caller `exec`s from the child will have these three descriptors connected to the slave PTY (its controlling terminal).

After the call to `fork`, the parent just returns the PTY master descriptor and the process ID of the child. In the next section, we use the `pty_fork` function in the `pty` program.

```
#include "apue.h"
#include <termios.h>

pid_t
pty_fork(int *ptrfdm, char *slave_name, int slave_namesz,
         const struct termios *slave_termios,
         const struct winsize *slave_winsize)
{
    int     fdm, fds;
```

```
    pid_t   pid;
    char    pts_name[20];

    if ((fdm = ptym_open(pts_name, sizeof(pts_name))) < 0)
        err_sys("can't open master pty: %s, error %d", pts_name, fdm);

    if (slave_name != NULL) {
        /*
         * Return name of slave.  Null terminate to handle case
         * where strlen(pts_name) > slave_namesz.
         */
        strncpy(slave_name, pts_name, slave_namesz);
        slave_name[slave_namesz - 1] = '\0';
    }

    if ((pid = fork()) < 0) {
        return(-1);
    } else if (pid == 0) {      /* child */
        if (setsid() < 0)
            err_sys("setsid error");

        /*
         * System V acquires controlling terminal on open().
         */
        if ((fds = ptys_open(pts_name)) < 0)
            err_sys("can't open slave pty");
        close(fdm);     /* all done with master in child */

#if defined(BSD)
        /*
         * TIOCSCTTY is the BSD way to acquire a controlling terminal.
         */
        if (ioctl(fds, TIOCSCTTY, (char *)0) < 0)
            err_sys("TIOCSCTTY error");
#endif
        /*
         * Set slave's termios and window size.
         */
        if (slave_termios != NULL) {
            if (tcsetattr(fds, TCSANOW, slave_termios) < 0)
                err_sys("tcsetattr error on slave pty");
        }
        if (slave_winsize != NULL) {
            if (ioctl(fds, TIOCSWINSZ, slave_winsize) < 0)
                err_sys("TIOCSWINSZ error on slave pty");
        }

        /*
         * Slave becomes stdin/stdout/stderr of child.
         */
        if (dup2(fds, STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");
```

```
        if (dup2(fds, STDOUT_FILENO) != STDOUT_FILENO)
            err_sys("dup2 error to stdout");
        if (dup2(fds, STDERR_FILENO) != STDERR_FILENO)
            err_sys("dup2 error to stderr");
        if (fds != STDIN_FILENO && fds != STDOUT_FILENO &&
          fds != STDERR_FILENO)
            close(fds);
        return(0);      /* child returns 0 just like fork() */
    } else {                    /* parent */
        *ptrfdm = fdm;  /* return fd of master */
        return(pid);    /* parent returns pid of child */
    }
}
```

Figure 19.10 The `pty_fork` function

19.5 `pty` Program

The goal in writing the `pty` program is to be able to type

```
pty prog arg1 arg2
```

instead of

```
prog arg1 arg2
```

When we use `pty` to execute another program, that program is executed in a session of its own, connected to a pseudo terminal.

Let's look at the source code for the `pty` program. The first file (Figure 19.11) contains the `main` function. It calls the `pty_fork` function from the previous section.

```
#include "apue.h"
#include <termios.h>

#ifdef LINUX
#define OPTSTR "+d:einv"
#else
#define OPTSTR "d:einv"
#endif

static void set_noecho(int);    /* at the end of this file */
void        do_driver(char *);  /* in the file driver.c */
void        loop(int, int);     /* in the file loop.c */

int
main(int argc, char *argv[])
{
    int             fdm, c, ignoreeof, interactive, noecho, verbose;
    pid_t           pid;
    char            *driver;
    char            slave_name[20];
```

```
struct termios  orig_termios;
struct winsize  size;

interactive = isatty(STDIN_FILENO);
ignoreeof = 0;
noecho = 0;
verbose = 0;
driver = NULL;

opterr = 0;     /* don't want getopt() writing to stderr */
while ((c = getopt(argc, argv, OPTSTR)) != EOF) {
    switch (c) {
    case 'd':       /* driver for stdin/stdout */
        driver = optarg;
        break;

    case 'e':       /* noecho for slave pty's line discipline */
        noecho = 1;
        break;

    case 'i':       /* ignore EOF on standard input */
        ignoreeof = 1;
        break;

    case 'n':       /* not interactive */
        interactive = 0;
        break;

    case 'v':       /* verbose */
        verbose = 1;
        break;

    case '?':
        err_quit("unrecognized option: -%c", optopt);
    }
}
if (optind >= argc)
    err_quit("usage: pty [ -d driver -einv ] program [ arg ... ]");

if (interactive) {  /* fetch current termios and window size */
    if (tcgetattr(STDIN_FILENO, &orig_termios) < 0)
        err_sys("tcgetattr error on stdin");
    if (ioctl(STDIN_FILENO, TIOCGWINSZ, (char *) &size) < 0)
        err_sys("TIOCGWINSZ error");
    pid = pty_fork(&fdm, slave_name, sizeof(slave_name),
      &orig_termios, &size);
} else {
    pid = pty_fork(&fdm, slave_name, sizeof(slave_name),
      NULL, NULL);
}

if (pid < 0) {
    err_sys("fork error");
```

```
    } else if (pid == 0) {      /* child */
        if (noecho)
            set_noecho(STDIN_FILENO);   /* stdin is slave pty */

        if (execvp(argv[optind], &argv[optind]) < 0)
            err_sys("can't execute: %s", argv[optind]);
    }

    if (verbose) {
        fprintf(stderr, "slave name = %s\n", slave_name);
        if (driver != NULL)
            fprintf(stderr, "driver = %s\n", driver);
    }

    if (interactive && driver == NULL) {
        if (tty_raw(STDIN_FILENO) < 0)  /* user's tty to raw mode */
            err_sys("tty_raw error");
        if (atexit(tty_atexit) < 0)     /* reset user's tty on exit */
            err_sys("atexit error");
    }

    if (driver)
        do_driver(driver);  /* changes our stdin/stdout */

    loop(fdm, ignoreeof);   /* copies stdin -> ptym, ptym -> stdout */

    exit(0);
}

static void
set_noecho(int fd)      /* turn off echo (for slave pty) */
{
    struct termios  stermios;

    if (tcgetattr(fd, &stermios) < 0)
        err_sys("tcgetattr error");

    stermios.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);

    /*
     * Also turn off NL to CR/NL mapping on output.
     */
    stermios.c_oflag &= ~(ONLCR);

    if (tcsetattr(fd, TCSANOW, &stermios) < 0)
        err_sys("tcsetattr error");
}
```

Figure 19.11 The `main` function for the `pty` program

In the next section, we'll look at the various command-line options when we examine different uses of the `pty` program. The `getopt` function helps us parse command-line arguments in a consistent manner. To enforce POSIX behavior on Linux systems, we set the first character of the option string to a plus sign.

Before calling `pty_fork`, we fetch the current values for the `termios` and `winsize` structures, passing these as arguments to `pty_fork`. This way, the PTY slave assumes the same initial state as the current terminal.

After returning from `pty_fork`, the child optionally turns off echoing for the slave PTY and then calls `execvp` to execute the program specified on the command line. All remaining command-line arguments are passed as arguments to this program.

The parent optionally sets the user's terminal to raw mode. In this case, the parent also sets an exit handler to reset the terminal state when `exit` is called. We describe the `do_driver` function in the next section.

The parent then calls the function `loop` (Figure 19.12), which copies everything received from the standard input to the PTY master and everything from the PTY master to standard output. For variety, we have coded it in two processes this time, although a single process using `select`, `poll`, or multiple threads would also work.

```
#include "apue.h"

#define BUFFSIZE    512

static void sig_term(int);
static volatile sig_atomic_t    sigcaught;  /* set by signal handler */

void
loop(int ptym, int ignoreeof)
{
    pid_t   child;
    int     nread;
    char    buf[BUFFSIZE];

    if ((child = fork()) < 0) {
        err_sys("fork error");
    } else if (child == 0) {    /* child copies stdin to ptym */
        for ( ; ; ) {
            if ((nread = read(STDIN_FILENO, buf, BUFFSIZE)) < 0)
                err_sys("read error from stdin");
            else if (nread == 0)
                break;      /* EOF on stdin means we're done */
            if (writen(ptym, buf, nread) != nread)
                err_sys("writen error to master pty");
        }

        /*
         * We always terminate when we encounter an EOF on stdin,
         * but we notify the parent only if ignoreeof is 0.
         */
        if (ignoreeof == 0)
            kill(getppid(), SIGTERM);   /* notify parent */
        exit(0);    /* and terminate; child can't return */
    }

    /*
     * Parent copies ptym to stdout.
     */
```

```
    if (signal_intr(SIGTERM, sig_term) == SIG_ERR)
        err_sys("signal_intr error for SIGTERM");

    for ( ; ; ) {
        if ((nread = read(ptym, buf, BUFFSIZE)) <= 0)
            break;      /* signal caught, error, or EOF */
        if (writen(STDOUT_FILENO, buf, nread) != nread)
            err_sys("writen error to stdout");
    }

    /*
     * There are three ways to get here: sig_term() below caught the
     * SIGTERM from the child, we read an EOF on the pty master (which
     * means we have to signal the child to stop), or an error.
     */
    if (sigcaught == 0) /* tell child if it didn't send us the signal */
        kill(child, SIGTERM);

    /*
     * Parent returns to caller.
     */
}

/*
 * The child sends us SIGTERM when it gets EOF on the pty slave or
 * when read() fails.  We probably interrupted the read() of ptym.
 */
static void
sig_term(int signo)
{
    sigcaught = 1;      /* just set flag and return */
}
```

Figure 19.12 The loop function

Note that because we use two processes, one has to notify the other when it terminates. We use the SIGTERM signal for this notification.

19.6 Using the pty Program

We'll now look at various examples with the pty program, seeing the need for the command-line options.

If our shell is the Korn shell, we can execute the command

```
pty ksh
```

and get a brand-new invocation of the shell, running under a pseudo terminal.

If the file ttyname is the program we showed in Figure 18.16, we can run the pty program as follows:

```
$ who
sar  console  May 19 16:47
sar  ttys000  May 19 16:47
sar  ttys001  May 19 16:48
sar  ttys002  May 19 16:48
sar  ttys003  May 19 16:49
sar  ttys004  May 19 16:49          ttys004 is the highest PTY currently in use
$ pty ttyname                       run program in Figure 18.16 from PTY
fd 0: /dev/ttys005                  ttys005 is the next available PTY
fd 1: /dev/ttys005
fd 2: /dev/ttys005
```

`utmp` File

In Section 6.8, we described the `utmp` file that records all users currently logged in to a UNIX system. The question is whether a user running a program on a pseudo terminal is considered logged in. In the case of remote logins, with `telnetd` and `rlogind`, obviously an entry should be made in the `utmp` file for the user logged in on the pseudo terminal. There is little agreement, however, whether users running a shell on a pseudo terminal from a window system or from a program, such as `script`, should have entries made in the `utmp` file. Some systems record these; others don't. If a system doesn't record these entries in the `utmp` file, the `who`(1) program normally won't show the corresponding pseudo terminals as being used.

Unless the `utmp` file has other-write permission enabled (which is considered to be a security hole), random programs that use pseudo terminals won't be able to write to this file.

Job Control Interaction

If we run a job-control shell under `pty`, it works normally. For example,

```
pty ksh
```

runs the Korn shell under `pty`. We can run programs under this new shell and use job control just as we do with our login shell. But if we run an interactive program other than a job-control shell under `pty`, as in

```
pty cat
```

everything is fine until we type the job-control suspend character. At that point, the job-control character is echoed as `^Z` and is ignored. Under earlier BSD-based systems, the `cat` process terminates, the `pty` process terminates, and we're back to our original shell. To understand what's going on here, we need to examine all the processes involved, their process groups, and sessions. Figure 19.13 shows the arrangement when `pty cat` is running.

When we type the suspend character (Control-Z), it is recognized by the line discipline module beneath the `cat` process, since `pty` puts the terminal (beneath the `pty` parent) into raw mode. But the kernel won't stop the `cat` process, because it

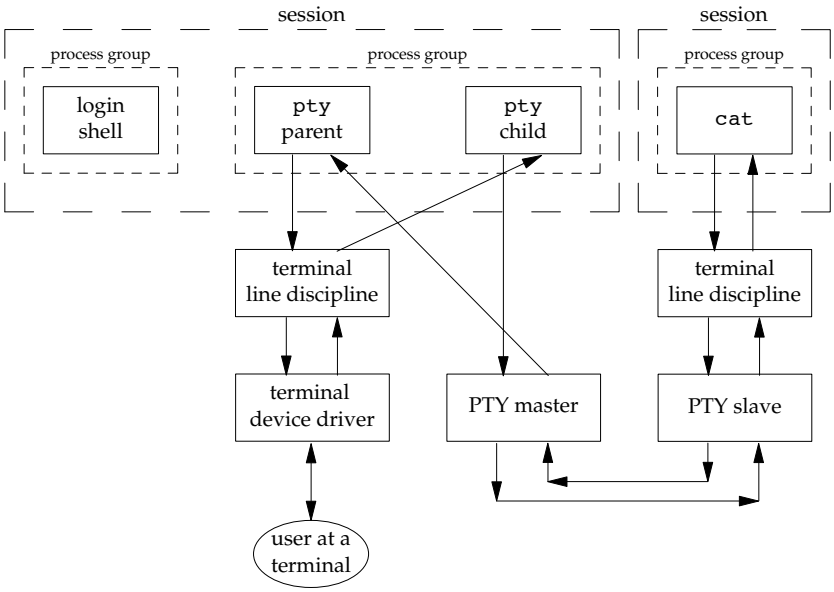


Figure 19.13 Process groups and sessions for `pty cat`

belongs to an orphaned process group (Section 9.10). The parent of `cat` is the `pty` parent, and it belongs to another session.

Historically, implementations have handled this condition differently. POSIX.1 says only that the `SIGTSTP` signal can't be delivered to the process. Systems derived from 4.3BSD delivered `SIGKILL` instead, which the process can't even catch. In 4.4BSD, this behavior was changed to conform to POSIX.1. Instead of sending `SIGKILL`, the 4.4BSD kernel silently discards the `SIGTSTP` signal if it has the default disposition and is to be delivered to a process in an orphaned process group. Most current implementations follow this behavior.

When we use `pty` to run a job-control shell, the jobs invoked by this new shell are never members of an orphaned process group, because the job-control shell always belongs to the same session. In that case, the Control-Z that we type is sent to the process invoked by the shell, not to the shell itself.

The only way to avoid this inability of the process invoked by `pty` to handle job-control signals is to add yet another command-line flag to `pty`, telling it to recognize the job control suspend character itself (in the `pty` child) instead of letting the character get all the way through to the other line discipline.

Watching the Output of Long-Running Programs

Another example of job control interaction with the `pty` program is found in the configuration illustrated in Figure 19.7. If we run the program that generates output slowly as

```
pty slowout > file.out &
```

the `pty` process is stopped immediately when the child tries to read from its standard input (the terminal). The reason is that the job is a background job and gets job-control stopped when it tries to access the terminal. If we redirect standard input so that `pty` doesn't try to read from the terminal, as in

```
pty slowout < /dev/null > file.out &
```

the `pty` program stops immediately because it reads an end of file on its standard input and terminates. The solution for this problem is the `-i` option, which says to ignore an end of file on the standard input:

```
pty -i slowout < /dev/null > file.out &
```

This flag causes the `pty` child in Figure 19.12 to exit when the end of file is encountered, but the child doesn't tell the parent to terminate. Instead, the parent continues copying the PTY slave output to standard output (the file `file.out` in the example).

`script` Program

Using the `pty` program, we can implement the `script`(1) program as the following shell script:

```
#!/bin/sh
pty "${SHELL:-/bin/sh}" | tee typescript
```

Once we run this shell script, we can execute the `ps` command to see all the process relationships. Figure 19.14 details these relationships.

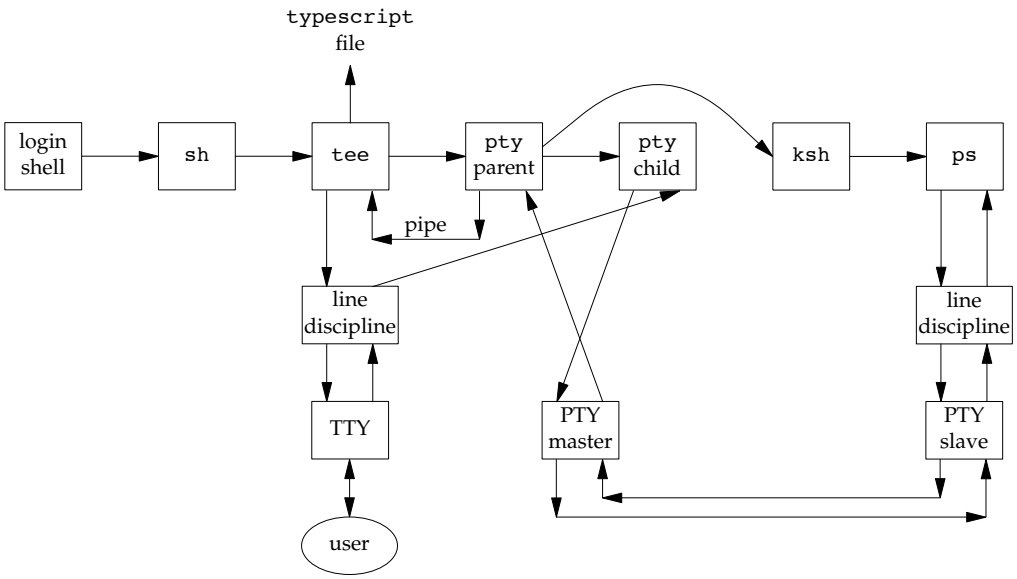


Figure 19.14 Arrangement of processes for `script` shell script

In this example, we assume that the SHELL variable is the Korn shell (probably /bin/ksh). As we mentioned earlier, script copies only what is output by the new shell (and any processes that it invokes), but since the line discipline module above the PTY slave normally has echo enabled, most of what we type is also written to the typescript file.

Running Coprocesses

In Figure 15.18, the coprocess couldn't use the standard I/O functions, because standard input and standard output do not refer to a terminal, so the standard I/O functions treat them as fully buffered. If we run the coprocess under pty by replacing the line

```
if (execl("./add2", "add2", (char *)0) < 0)
```

with

```
if (execl("./pty", "pty", "-e", "add2", (char *)0) < 0)
```

the program now works, even if the coprocess uses standard I/O.

Figure 19.15 shows the arrangement of processes when we run the coprocess with a pseudo terminal as its input and output. It is an expansion of Figure 19.6, showing all the process connections and data flow. The box labeled "driving program" is the program from Figure 15.18, with the execl changed as described previously.

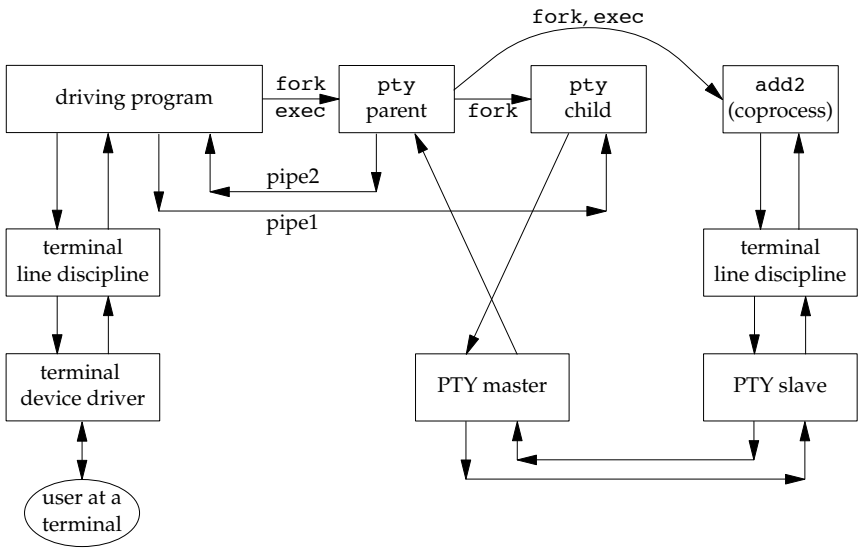


Figure 19.15 Running a coprocess with a pseudo terminal as its input and output

This example shows the need for the -e (no echo) option for the pty program. The pty program is not running interactively, because its standard input is not connected to a terminal. In Figure 19.11, the interactive flag defaults to false, since the call to

`isatty` returns false. This means that the line discipline above the actual terminal remains in canonical mode with echo enabled. By specifying the `-e` option, we turn off echo in the line discipline module above the PTY slave. If we don't do this, everything we type is echoed twice—by both line discipline modules.

We also have the `-e` option turn off the `ONLCR` flag in the `termios` structure to prevent all the output from the coprocess from being terminated with a carriage return and a newline.

Testing this example on different systems revealed another problem that we alluded to in Section 14.7 when we described the `readn` and `writen` functions. The amount of data returned by a `read`, when the descriptor refers to something other than an ordinary disk file, can differ between implementations. This coprocess example using `pty` gave unexpected results that were tracked down to the `read` function on the pipe in the program from Figure 15.18, which returned less than a line. The solution was to not use the program shown in Figure 15.18, but rather to use the version of this program from Exercise 15.5 that was modified to use the standard I/O library, with the standard I/O streams for both pipes set to line buffering. With this approach, the `fgets` function does as many `read`s as required to obtain a complete line. The `while` loop in Figure 15.18 assumes that each line sent to the coprocess causes one line to be returned.

Driving Interactive Programs Noninteractively

Although it's tempting to think that `pty` can run any coprocess, even a coprocess that is interactive, it doesn't work. The problem is that `pty` just copies everything on its standard input to the PTY and everything from the PTY to its standard output, never looking at what it sends or what it gets back.

As an example, we can run the `telnet` command under `pty`, talking directly to the remote host:

```
pty telnet 192.168.1.3
```

Doing this provides no benefit over just typing `telnet 192.168.1.3`, but we would like to run the `telnet` program from a script, perhaps to check some condition on the remote host. If the file `telnet.cmd` contains the four lines

```
sar
passwd
uptime
exit
```

the first line is the user name we use to log in to the remote host, the second line is the password, the third line is a command we'd like to run, and the fourth line terminates the session. But if we run this script as

```
pty -i < telnet.cmd telnet 192.168.1.3
```

it doesn't do what we want. Instead, the contents of the file `telnet.cmd` are sent to the remote host before it has a chance to prompt us for an account name and password. When it turns off echoing to read the password, `login` uses the `tcsetattr` option, which discards any data already queued. Thus, the data we send is thrown away.

When we run the telnet program interactively, we wait for the remote host to prompt for a password before we type it, but the pty program doesn't know to do this. This is why it takes a more sophisticated program than pty, such as expect, to drive an interactive program from a script file.

Even running pty from the program in Figure 15.18, as we showed earlier, doesn't help, because the program in Figure 15.18 assumes that each line it writes to the pipe generates exactly one line on the other pipe. With an interactive program, one line of input may generate many lines of output. Furthermore, the program in Figure 15.18 always sent a line to the coprocess before reading from it. This strategy won't work when we want to read from the coprocess before sending it anything.

There are a few ways to proceed from here to be able to drive an interactive program from a script. We could add a command language and interpreter to pty, but a reasonable command language would probably be ten times larger than the pty program. Another option is to take a command language and use the pty_fork function to invoke interactive programs. This is what the expect program does.

We'll take a different path here and just provide an option (-d) to allow pty to be connected to a driver process for its input and output. The standard output of the driver is pty's standard input, and vice versa. This is similar to a coprocess, but on "the other side" of pty. The resulting arrangement of processes is almost identical to Figure 19.15, but in the current scenario, pty does the fork and exec of the driver process. Also, instead of two half-duplex pipes, we'll use a single bidirectional pipe between pty and the driver process.

Figure 19.16 shows the source for the do_driver function, which is called by the main function of pty (Figure 19.11) when the -d option is specified.

```
#include "apue.h"

void
do_driver(char *driver)
{
    pid_t   child;
    int     pipe[2];

    /*
     * Create a full-duplex pipe to communicate with the driver.
     */
    if (fd_pipe(pipe) < 0)
        err_sys("can't create stream pipe");

    if ((child = fork()) < 0) {
        err_sys("fork error");
    } else if (child == 0) {        /* child */
        close(pipe[1]);

        /* stdin for driver */
        if (dup2(pipe[0], STDIN_FILENO) != STDIN_FILENO)
            err_sys("dup2 error to stdin");

        /* stdout for driver */
        if (dup2(pipe[0], STDOUT_FILENO) != STDOUT_FILENO)
```

```
            err_sys("dup2 error to stdout");
        if (pipe[0] != STDIN_FILENO && pipe[0] != STDOUT_FILENO)
            close(pipe[0]);

        /* leave stderr for driver alone */
        execlp(driver, driver, (char *)0);
        err_sys("execlp error for: %s", driver);
    }

    close(pipe[0]);     /* parent */
    if (dup2(pipe[1], STDIN_FILENO) != STDIN_FILENO)
        err_sys("dup2 error to stdin");
    if (dup2(pipe[1], STDOUT_FILENO) != STDOUT_FILENO)
        err_sys("dup2 error to stdout");
    if (pipe[1] != STDIN_FILENO && pipe[1] != STDOUT_FILENO)
        close(pipe[1]);

    /*
     * Parent returns, but with stdin and stdout connected
     * to the driver.
     */
}
```

Figure 19.16 The `do_driver` function for the `pty` program

By writing our own driver program that is invoked by `pty`, we can drive interactive programs in any way desired. Even though it has its standard input and standard output connected to `pty`, the driver process can still interact with the user by reading and writing `/dev/tty`. This solution still isn't as general as the `expect` program, but it provides a useful option to `pty` for fewer than 50 lines of code.

19.7 Advanced Features

Pseudo terminals have some additional capabilities that we briefly mention here. These capabilities are further documented in Sun Microsystems [2005] and the BSD `pts`(4) manual page.

Packet Mode

Packet mode lets the PTY master learn of state changes in the PTY slave. On Solaris, this mode is enabled by pushing the STREAMS module `pckt` onto the PTY master side. We showed this optional module in Figure 19.2. On FreeBSD, Linux, and Mac OS X, this mode is enabled with the `TIOCPKT` `ioctl` command.

The details of packet mode differ between Solaris and the other platforms. Under Solaris, the process reading the PTY master has to call `getmsg` to fetch the messages from the stream head, because the `pckt` module converts certain events into nondata STREAMS messages. With the other platforms, each `read` from the PTY master returns a status byte followed by optional data.

Regardless of the implementation details, the purpose of packet mode is to inform the process reading the PTY master when the following events occur at the line discipline module above the PTY slave: when the read queue is flushed, when the write queue is flushed, when output is stopped (e.g., Control-S), when output is restarted, when XON/XOFF flow control is enabled after being disabled, and when XON/XOFF flow control is disabled after being enabled. These events are used, for example, by the `rlogin` client and the `rlogind` server.

Remote Mode

A PTY master can set the PTY slave to remote mode by issuing the `TIOCREMOTE ioctl` command. Although Mac OS X 10.6.8 and Solaris 10 use the same command to enable and disable this feature, under Solaris the third argument to `ioctl` is an integer, whereas with Mac OS X, it is a pointer to an integer. (FreeBSD 8.0 and Linux 3.2.0 don't support this command.)

When it sets this mode, the PTY master is telling the PTY slave's line discipline not to perform any processing of the data that it receives from the PTY master, regardless of the canonical/noncanonical flag in the slave's `termios` structure. Remote mode is intended for an application, such as a window manager, that does its own line editing.

Window Size Changes

The process above the PTY master can issue the `TIOCSWINSZ ioctl` command to set the window size of the slave. If the new size differs from the current size, a `SIGWINCH` signal is sent to the foreground process group of the PTY slave.

Signal Generation

The process reading and writing the PTY master can send signals to the process group of the PTY slave. Under Solaris 10, this is done using the `TIOCSIGNAL ioctl` command. With FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8, the `ioctl` command is `TIOCSIG`. In both cases, the third argument is set to the signal number.

19.8 Summary

We started this chapter with an overview of how to use pseudo terminals and a look at some use cases. We continued by examining the code required to set up a pseudo terminal under the four platforms discussed in this text. We then used this code to provide the generic `pty_fork` function that can be used by many different applications. We used this function as the basis for a small program (`pty`), which we then used to explore many of the properties of pseudo terminals.

Pseudo terminals are used daily on most UNIX systems to provide network logins. We've examined other uses for pseudo terminals as well, ranging from the `script` program to driving interactive programs from a batch script.

Exercises

19.1 When we remotely log in to a BSD system using either `telnet` or `rlogin`, the ownership of the PTY slave and its permissions are set, as we described in Section 19.3. How does this happen?

19.2 Use the `pty` program to determine the values used by your system to initialize a slave PTY's `termios` structure and `winsize` structure.

19.3 Recode the `loop` function (Figure 19.12) as a single process using either `select` or `poll`.

19.4 In the child process after `pty_fork` returns, standard input, standard output, and standard error are all open for read–write. Can you change standard input to be read-only and the other two to be write-only?

19.5 In Figure 19.13, identify which process groups are in the foreground and which are in the background, and identify the session leaders.

19.6 In Figure 19.13, in what order do the processes terminate when we type the end-of-file character? Verify this with process accounting, if possible.

19.7 The `script`(1) program normally adds to the beginning of the output file a line with the starting time, and to the end of the output file another line with the ending time. Add these features to the simple shell script that we showed.

19.8 Explain why the contents of the file `data` are output to the terminal in the following example, even though the program `ttyname` (Figure 18.16) only generates output and never reads its input.

```
$ cat data                      a file with two lines
hello,
world
$ pty -i < data ttyname         -i says ignore eof on stdin
hello,                          where did these two lines come from?
world
fd 0: /dev/ttys005              we expect these three lines from ttyname
fd 1: /dev/ttys005
fd 2: /dev/ttys005
```

19.9 Write a program that calls `pty_fork` and have the child `exec` another program that you will write. The new program that the child `exec`s must catch `SIGTERM` and `SIGWINCH`. When it catches a signal, the program should print that it did; for the latter signal, it should also print the terminal's window size. Then have the parent process send the `SIGTERM` signal to the process group of the PTY slave with the `ioctl` command we described in Section 19.7. Read back from the slave to verify that the signal was caught. Follow this with the parent setting the window size of the PTY slave, and then read back the slave's output again. Have the parent `exit` and determine whether the slave process also terminates; if so, how does it terminate?

20

A Database Library

20.1 Introduction

During the early 1980s, the UNIX System was considered a hostile environment for running multiuser database systems. (See Stonebraker [1981] and Weinberger [1982].) Earlier systems, such as Version 7, did indeed present large obstacles, since they did not provide any form of IPC (other than half-duplex pipes) and did not provide any form of byte-range locking. Many of these deficiencies were remedied, however. By the late 1980s, the UNIX System had evolved to provide a suitable environment for running reliable, multiuser database systems. Since then, numerous commercial firms have offered these types of database systems.

In this chapter, we develop a simple, multiuser database library of C functions that any program can call to fetch and store records in a database. (Such a database is often called a *key–value store*.) This library of C functions is usually only one part of a complete database system. We do not develop the other pieces, such as a query language, leaving these items to the many textbooks on database systems. Our interest is the parts of the UNIX System interface required by a database library, and how they relate to the topics we've already covered (such as record—byte-range—locking, in Section 14.3).

20.2 History

One popular library of database functions in the UNIX System is the `dbm`(3) library. This library was developed by Ken Thompson and uses a dynamic hashing scheme. It was originally provided with Version 7, appears in all BSD releases, and was also

provided in SVR4's BSD-compatibility library [AT&T 1990c]. The BSD developers extended the `dbm` library and called it `ndbm`. The `ndbm` library was included in BSD as well as in SVR4. The `ndbm` functions are standardized in the XSI option of the Single UNIX Specification.

Seltzer and Yigit [1991] provide a detailed history of the dynamic hashing algorithm used by the `dbm` library and other implementations of this library, including `gdbm`, the GNU version of the `dbm` library. Unfortunately, a basic limitation of all these implementations is that none allows concurrent updating of the database by multiple processes. These implementations provide no type of concurrency controls (such as record locking).

4.4BSD provided a new `db`(3) library that supports three forms of access: (a) record oriented, (b) hashing, and (c) a B-tree. Again, no form of concurrency was provided (as was plainly stated in the BUGS section of the `db`(3) manual page).

> Oracle (`http://www.oracle.com`) provides versions of the `db` library that do support concurrent access, locking, and transactions.

Most commercial database libraries do provide the concurrency controls required for multiple processes to update a database simultaneously. These systems typically use advisory locking, as we described in Section 14.3, but they often implement their own locking primitives to avoid the overhead of a system call to acquire an uncontested lock. These commercial systems usually implement their database using B+ trees [Comer 1979] or some dynamic hashing technique, such as linear hashing [Litwin 1980] or extendible hashing [Fagin et al. 1979].

Figure 20.1 summarizes the database libraries commonly found in the four operating systems described in this book. Note that on Linux, the `gdbm` library provides support for both `dbm` and `ndbm` functions.

Library	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
`dbm`	XSI		`gdbm`		•
`ndbm`		•	`gdbm`	•	•
`db`		•	•	•	

Figure 20.1 Support for database libraries on various platforms

20.3 The Library

The library we develop in this chapter will be similar to the `ndbm` library, but we'll add the concurrency control mechanisms to allow multiple processes to update the same database at the same time. We first describe the C interface to the database library, then in the next section describe the actual implementation.

When we open a database, we are returned a handle (an opaque pointer) representing the database. We'll pass this handle to the remaining database functions.

```
#include "apue_db.h"

DBHANDLE db_open(const char *pathname, int oflag, ... /* int mode */);

                                        Returns: database handle if OK, NULL on error

void db_close(DBHANDLE db);
```

If `db_open` is successful, two files are created: *pathname.idx* is the index file, and *pathname.dat* is the data file. The *oflag* argument is used as the second argument to `open` (Section 3.3) to specify how the files are to be opened (e.g., read-only, read–write, create file if it doesn't exist). The *mode* argument is used as the third argument to `open` (the file access permissions) if the database files are created.

When we're done with a database, we call `db_close`. It closes the index file and the data file and releases any memory that it allocated for internal buffers.

When we store a new record in the database, we have to specify the key for the record and the data associated with the key. If the database contained personnel records, the key could be the employee ID, and the data could be the employee's name, address, telephone number, date of hire, and the like. Our implementation requires that the key for each record be unique. (We can't have two employee records with the same employee ID, for example.)

```
#include "apue_db.h"

int db_store(DBHANDLE db, const char *key, const char *data,
             int flag);

                                        Returns: 0 if OK, nonzero on error (see following)
```

The *key* and *data* arguments are null-terminated character strings. The only restriction on these two strings is that neither can contain null bytes. They may, for example, contain newlines.

The *flag* argument can be `DB_INSERT` (to insert a new record), `DB_REPLACE` (to replace an existing record), or `DB_STORE` (to either insert or replace a record, whichever is appropriate). These three constants are defined in the `apue_db.h` header. If we specify either `DB_INSERT` or `DB_STORE` and the record does not exist, a new record is inserted. If we specify either `DB_REPLACE` or `DB_STORE` and the record already exists, the existing record is replaced with the new record. If we specify `DB_REPLACE` and the record doesn't exist, we set `errno` to `ENOENT` and return –1 without adding the new record. If we specify `DB_INSERT` and the record already exists, no record is inserted. In this case, the return value is 1 to distinguish it from a normal error return (–1).

We can fetch any record from the database by specifying its *key*.

```
#include "apue_db.h"

char *db_fetch(DBHANDLE db, const char *key);

                                        Returns: pointer to data if OK, NULL if record not found
```

The return value is a pointer to the data that was stored with the *key*, if the record is found. We can also delete a record from the database by specifying its *key*.

```
#include "apue_db.h"

int db_delete(DBHANDLE db, const char *key);
```

Returns: 0 if OK, –1 if record not found

In addition to fetching a record by specifying its key, we can go through the entire database, reading each record in turn. To do this, we first call `db_rewind` to rewind the database to the first record and then call `db_nextrec` in a loop to read each sequential record.

```
#include "apue_db.h"

void db_rewind(DBHANDLE db);

char *db_nextrec(DBHANDLE db, char *key);
```

Returns: pointer to data if OK, `NULL` on end of file

If *key* is a non-null pointer, `db_nextrec` returns the key by copying it to the memory starting at that location.

There is no order to the records returned by `db_nextrec`. All we're guaranteed is that we'll read each record in the database once. If we store three records with keys of A, B, and C, in that order, we have no idea in which order `db_nextrec` will return the three records. It might return B, then A, then C, or some other (apparently random) order. The actual order depends on the implementation of the database.

These seven functions provide the interface to the database library. We now describe the actual implementation that we have chosen.

20.4 Implementation Overview

Database access libraries often use two files to store the information: an index file and a data file. The index file contains the actual index value (the key) and a pointer to the corresponding data record in the data file. Numerous techniques can be used to organize the index file so that it can be searched quickly and efficiently for any key: hashing and B+ trees are popular. We have chosen to use a fixed-size hash table with chaining for the index file. We mentioned in the description of `db_open` that we create two files: one with a suffix of `.idx` and one with a suffix of `.dat`.

We store the key and the index as null-terminated character strings; they cannot contain arbitrary binary data. Some database systems store numerical data in a binary format (1, 2, or 4 bytes for an integer, for example) to save storage space. This complicates the functions and requires more work to make the database files portable between different computer systems. For example, if a network has two systems that use different formats for storing binary integers, we need to account for this difference if we want both systems to access the database. (It is not at all uncommon today to have systems with different architectures sharing files on a network.) Storing all the records, both keys and data, as character strings simplifies everything. It does require additional disk space, but that is a small cost for portability.

With `db_store`, only one record for each key is allowed. Some database systems allow a key to have multiple records and then provide a way to access all the records associated with a given key. Additionally, we have only a single index file, meaning that each data record can have only a single key (we don't support secondary keys). Some database systems allow each record to have multiple keys and often use one index file per key. Each time a new record is inserted or deleted, all index files must be updated accordingly. (An example of a file with multiple indexes is an employee file. We could have one index whose key is the employee ID and another whose key is the employee's Social Security number. Having an index whose key is the employee name could be a problem, as names are not always unique.)

Figure 20.2 shows a general picture of the database implementation.

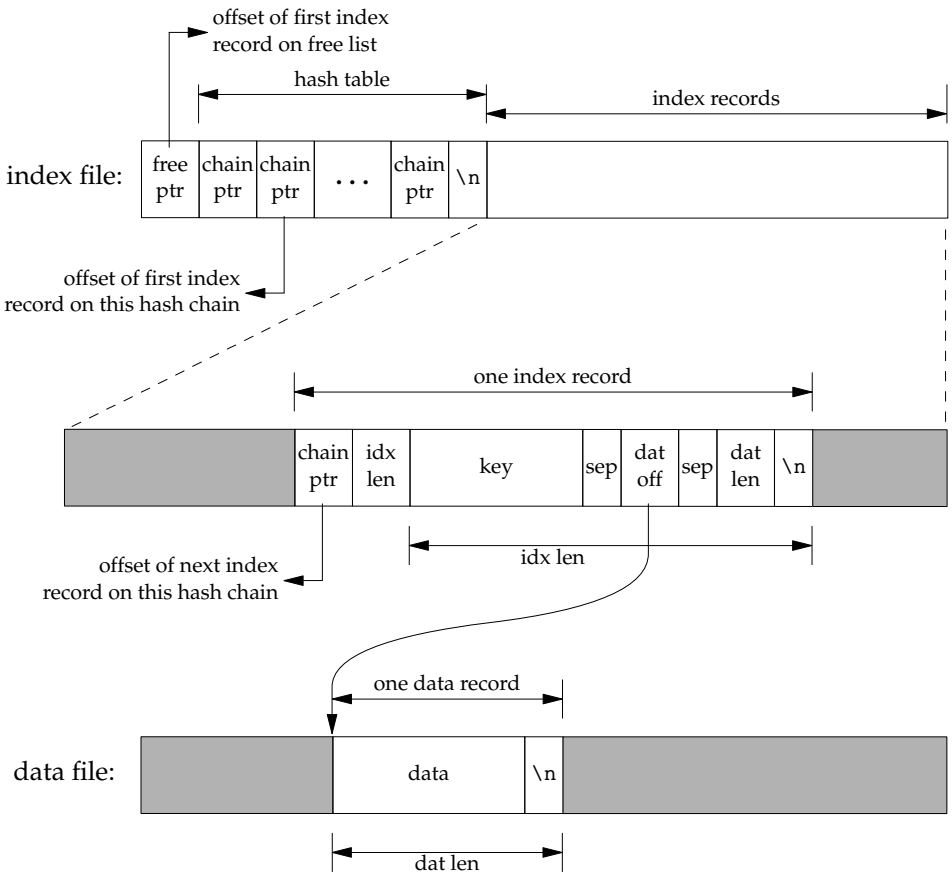


Figure 20.2 Arrangement of index file and data file

The index file consists of three portions: the free-list pointer, the hash table, and the index records. In Figure 20.2, all the fields called *ptr* are simply file offsets stored as an ASCII number.

To find a record in the database given its key, `db_fetch` calculates the hash value of the key, which leads to one hash chain in the hash table. (The *chain ptr* field could be 0, indicating an empty chain.) We then follow this hash chain, which is a linked list of all the index records with this hash value. When we encounter a *chain ptr* value of 0, we've hit the end of the hash chain.

Let's look at an actual database file. The program in Figure 20.3 creates a new database and writes three records to it. Since we store all the fields in the database as ASCII characters, we can look at the actual index file and data file using any of the standard UNIX System tools:

```
$ ls -l db4.*
-rw-r--r--  1 sar        28 Oct 19 21:33 db4.dat
-rw-r--r--  1 sar        72 Oct 19 21:33 db4.idx
$ cat db4.idx
   0  53  35   0
   0  10Alpha:0:6
   0  10beta:6:14
  17  11gamma:20:8
$ cat db4.dat
data1
Data for beta
record3
```

To keep this example small, we have set the size of each *ptr* field to four ASCII characters; the number of hash chains is set to 3. Since each *ptr* is a file offset, a four-character field limits the total size of the index file and data file to 10,000 bytes. When we do some performance measurements of the database system in Section 20.9, we set the size of each *ptr* field to six characters (allowing file sizes up to 1 million bytes) and the number of hash chains to more than 100.

The first line in the index file

```
   0  53  35   0
```

consists of the free-list pointer (0, the free list is empty) and the three hash chain pointers (53, 35, and 0). The next line

```
   0  10Alpha:0:6
```

shows the format of each index record. The first field (0) is the four-character chain pointer. This record is the end of its hash chain. The next field (10) is the four-character *idx len*, the length of the remainder of this index record. We read each index record using two `read`s: one to read the two fixed-size fields (the *chain ptr* and *idx len*) and another to read the remaining (variable-length) portion. The remaining three fields—*key*, *dat off*, and *dat len*—are delimited by a separator character (a colon in this case). We need the separator character, since each of these three fields is variable length. The separator character can't appear in the key. Finally, a newline terminates the index record. The newline isn't required, since *idx len* contains the length of the record. We store the newline to separate each index record so we can use the normal UNIX System tools, such as `cat` and `more`, with the index file. The *key* is the value that we specified

when we wrote the record to the database. The data offset (0) and data length (6) refer to the data file. We can see that the data record does start at offset 0 in the data file and has a length of 6 bytes.

```
#include "apue.h"
#include "apue_db.h"
#include <fcntl.h>

int
main(void)
{
    DBHANDLE    db;

    if ((db = db_open("db4", O_RDWR | O_CREAT | O_TRUNC,
      FILE_MODE)) == NULL)
        err_sys("db_open error");

    if (db_store(db, "Alpha", "data1", DB_INSERT) != 0)
        err_quit("db_store error for alpha");
    if (db_store(db, "beta", "Data for beta", DB_INSERT) != 0)
        err_quit("db_store error for beta");
    if (db_store(db, "gamma", "record3", DB_INSERT) != 0)
        err_quit("db_store error for gamma");

    db_close(db);
    exit(0);
}
```

Figure 20.3 Create a database and write three records to it

(As with the index file, we automatically append a newline to each data record, so we can use the normal UNIX System tools with the file. This newline at the end is not returned to the caller by `db_fetch`.)

If we follow the three hash chains in this example, we see that the first record on the first hash chain is at offset 53 (`gamma`). The next record on this chain is at offset 17 (`alpha`), and this is the last record on the chain. The first record on the second hash chain is at offset 35 (`beta`), and it's the last record on the chain. The third hash chain is empty.

Note that the order of the keys in the index file and the order of their corresponding records in the data file is the same as the order of the calls to `db_store` in Figure 20.3. Since the `O_TRUNC` flag was specified for `db_open`, the index file and the data file were both truncated and the database initialized from scratch. In this case, `db_store` just appends the new index records and data records to the end of the corresponding file. We'll see later that `db_store` can also reuse portions of these two files that correspond to deleted records.

The choice of a fixed-size hash table for the index is a compromise. It allows fast access as long as each hash chain isn't too long. We want to be able to search for any key quickly, but we don't want to complicate the data structures by using either a B-tree or dynamic hashing. Dynamic hashing has the advantage that any data record can be

located with only two disk accesses (see Litwin [1980] or Fagin et al. [1979] for details). B-trees have the advantage of traversing the database in (sorted) key order (something that we can't do with the `db_nextrec` function using a hash table.)

20.5 Centralized or Decentralized?

Given multiple processes accessing the same database, we can implement the functions in two ways:

1. Centralized. Have a single process that is the database manager, and have it be the only process that accesses the database. The functions contact this central process using some form of IPC.
2. Decentralized. Have each function apply the required concurrency controls (locking) and then issue its own I/O function calls.

Database systems have been built using each of these techniques. Given adequate locking routines, the decentralized implementation is usually faster, because IPC is avoided. Figure 20.4 depicts the operation of the centralized approach.

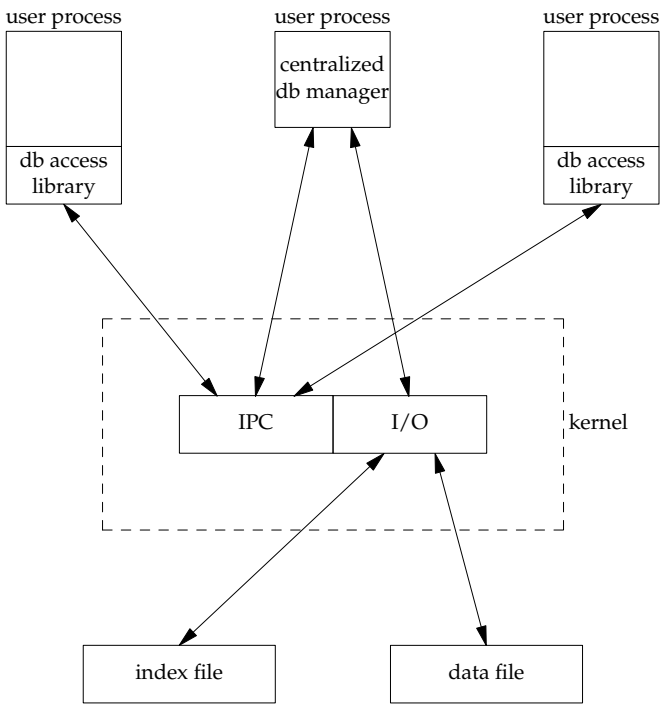


Figure 20.4 Centralized approach for database access

We purposely show the IPC going through the kernel, as most forms of message passing under the UNIX System operate this way. (Shared memory, as described in Section 15.9, avoids this copying of the data.) With the centralized approach, a record is read by the central process and then passed to the requesting process using IPC. This is a disadvantage of this design. Note that the centralized database manager is the only process that does I/O with the database files.

The centralized approach has the advantage that customer tuning of its operation may be possible. For example, we might be able to assign different priorities to different processes through the centralized process. This could affect the scheduling of I/O operations by the centralized process. With the decentralized approach, this is more difficult to do. We are usually at the mercy of the kernel's disk I/O scheduling policy and locking policy; that is, if three processes are waiting for a lock to become available, we cannot tell which process gets the lock next.

Another advantage of the centralized approach is that recovery is easier than with the decentralized approach. All the state information is in one place in the centralized approach, so if the database processes are killed, we have only one place to look to identify the outstanding transactions we need to resolve to restore the database to a consistent state.

The decentralized approach is shown in Figure 20.5. This is the design that we'll implement in this chapter.

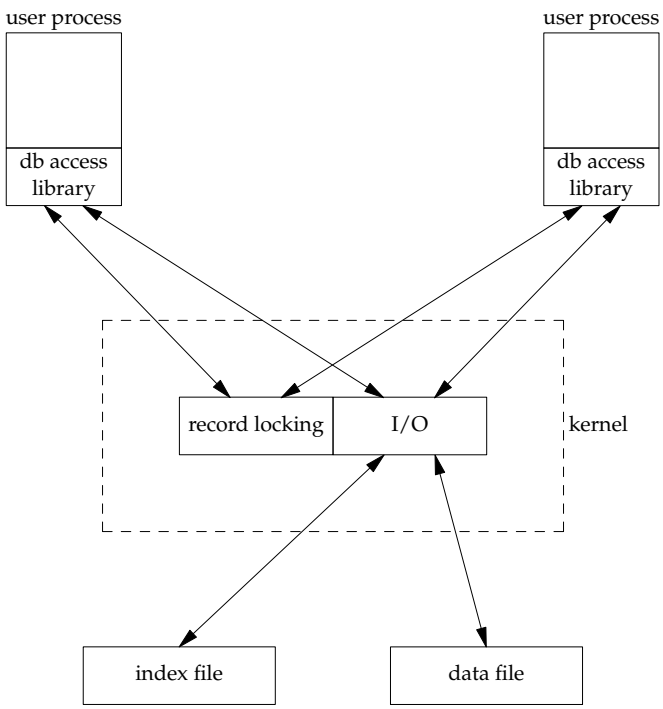


Figure 20.5 Decentralized approach for database access

The user processes that call the functions in the database library to perform I/O are considered cooperating processes, since they use byte-range locking to provide concurrent access.

20.6 Concurrency

We purposely chose a two-file implementation (an index file and a data file) because that is a common implementation technique (it simplifies space management in the files). It requires us to handle the locking interactions of both files. But there are numerous ways to handle the locking of these two files.

Coarse-Grained Locking

The simplest form of locking is to use one of the two files as a lock for the entire database and to require the caller to obtain this lock before operating on the database. We call this *coarse-grained locking*. For example, we can say that the process with a read lock on byte 0 of the index file has read access to the entire database. A process with a write lock on byte 0 of the index file has write access to the entire database. We can use the normal UNIX System byte-range locking semantics to allow any number of readers at one time, but only one writer at a time. (Recall Figure 14.3.) The functions `db_fetch` and `db_nextrec` require a read lock, and `db_delete`, `db_store`, and `db_open` all require a write lock. (The reason `db_open` requires a write lock is that if the file is being created, it has to write the empty free list and hash chains at the front of the index file.)

The problem with coarse-grained locking is that it limits concurrency. If a process is adding a record to one hash chain, another process should be able to read a record on a different hash chain.

Fine-Grained Locking

We enhance coarse-grained locking to allow more concurrency and call this *fine-grained locking*. We first require a reader or a writer to obtain a read lock or a write lock on the hash chain for a given record. We allow any number of readers at one time on any hash chain but only a single writer on a hash chain. Next, a writer needing to access the free list (either `db_delete` or `db_store`) must obtain a write lock on the free list. Finally, whenever it appends a new record to the end of either the index file or the data file, `db_store` has to obtain a write lock on that portion of the file.

We expect fine-grained locking to provide more concurrency than coarse-grained locking. In Section 20.9, we show some actual measurements. In Section 20.8, we show the source code for our implementation of fine-grained locking and discuss the details of implementing locking. (Coarse-grained locking is merely a simplification of the locking that we show.)

In the source code, we call `read`, `readv`, `write`, and `writev` directly. We do not use the standard I/O library. Although it is possible to use byte-range locking with the standard I/O library, careful handling of buffering is required. We don't want an

`fgets`, for example, to return data that was read into a standard I/O buffer 10 minutes ago if the data was modified by another process 5 minutes ago.

Our discussion of concurrency is predicated on the simple needs of the database library. Commercial systems often have additional requirements. See Chapter 16 of Date [2004] for additional details on concurrency.

20.7 Building the Library

The database library consists of two files: a public C header file and a C source file. We can build a static library using the commands

```
gcc -I../include -Wall -c db.c
ar rsv libapue_db.a db.o
```

Applications that want to link with `libapue_db.a` will also need to link with `libapue.a`, since we use some of our common functions in the database library.

If, on the other hand, we want to build a dynamic shared library version of the database library, we can use the following commands:

```
gcc -I../include -Wall -fPIC -c db.c
gcc -shared -Wl,-soname,libapue_db.so.1 -o libapue_db.so.1 \
    -L../lib -lapue -lc db.o
```

The resulting shared library, `libapue_db.so.1`, needs to be placed in a common directory where the dynamic linker/loader can find it. Alternatively, we can place it in a private directory and modify our `LD_LIBRARY_PATH` environment variable to include the private directory in the search path of the dynamic linker/loader.

> The steps used to build shared libraries vary among platforms. Here, we have shown how to do it on a Linux system with the GNU C compiler.

20.8 Source Code

We start by showing the `apue_db.h` header. This header is included by the library source code and all applications that call the library.

For the remainder of this text, we depart from the style of the previous examples in several ways. First, because the source code example is longer than usual, we number the lines. This makes it easier to link the discussion with the corresponding source code. Second, we place the description of the source code immediately below the source code on the same page.

> This style was inspired by John Lions in his book documenting the UNIX Version 6 operating system source code [Lions 1977, 1996]. It simplifies the task of studying large amounts of source code.

Note that we do not bother to number blank lines. Although this departs from the normal behavior of such tools as `pr`(1), we have nothing interesting to say about blank lines.

```
 1  #ifndef _APUE_DB_H
 2  #define _APUE_DB_H

 3  typedef    void *  DBHANDLE;

 4  DBHANDLE  db_open(const char *, int, ...);
 5  void      db_close(DBHANDLE);
 6  char     *db_fetch(DBHANDLE, const char *);
 7  int       db_store(DBHANDLE, const char *, const char *, int);
 8  int       db_delete(DBHANDLE, const char *);
 9  void      db_rewind(DBHANDLE);
10  char     *db_nextrec(DBHANDLE, char *);

11  /*
12   * Flags for db_store().
13   */
14  #define DB_INSERT     1    /* insert new record only */
15  #define DB_REPLACE    2    /* replace existing record */
16  #define DB_STORE      3    /* replace or insert */

17  /*
18   * Implementation limits.
19   */
20  #define IDXLEN_MIN    6    /* key, sep, start, sep, length, \n */
21  #define IDXLEN_MAX 1024    /* arbitrary */
22  #define DATLEN_MIN    2    /* data byte, newline */
23  #define DATLEN_MAX 1024    /* arbitrary */

24  #endif /* _APUE_DB_H */
```

[1–3] We use the `_APUE_DB_H` symbol to ensure that the contents of the header file are included only once. The `DBHANDLE` type represents an active reference to the database and is used to isolate applications from the implementation details of the database. Compare this technique with the way the standard I/O library exposes the `FILE` structure to applications.

[4–10] Next, we declare the prototypes for the database library's public functions. Since this header is included by applications that want to use the library, we don't declare the prototypes for the library's private functions here.

[11–24] The legal flags that can be passed to the `db_store` function are defined next, followed by fundamental limits of the implementation. These limits can be changed, if desired, to support bigger databases.

The minimum index record length is specified by `IDXLEN_MIN`. This represents a 1-byte key, a 1-byte separator, a 1-byte starting offset, another 1-byte separator, a 1-byte length, and a terminating newline character. (Recall the format of an index record from Figure 20.2.) An index record will usually be larger than `IDXLEN_MIN` bytes, but this is the bare minimum size.

The next file is `db.c`, the C source file for the library. For simplicity, we include all functions in a single file. This has the advantage that we can hide private functions by declaring them as `static`.

```
 1  #include "apue.h"
 2  #include "apue_db.h"
 3  #include <fcntl.h>      /* open & db_open flags */
 4  #include <stdarg.h>
 5  #include <errno.h>
 6  #include <sys/uio.h>    /* struct iovec */

 7  /*
 8   * Internal index file constants.
 9   * These are used to construct records in the
10   * index file and data file.
11   */
12  #define IDXLEN_SZ     4    /* index record length (ASCII chars) */
13  #define SEP         ':'    /* separator char in index record */
14  #define SPACE       ' '    /* space character */
15  #define NEWLINE     '\n'   /* newline character */

16  /*
17   * The following definitions are for hash chains and free
18   * list chain in the index file.
19   */
20  #define PTR_SZ        7    /* size of ptr field in hash chain */
21  #define PTR_MAX 9999999    /* max file offset = 10**PTR_SZ - 1 */
22  #define NHASH_DEF   137    /* default hash table size */
23  #define FREE_OFF      0    /* free list offset in index file */
24  #define HASH_OFF PTR_SZ    /* hash table offset in index file */

25  typedef unsigned long  DBHASH; /* hash values */
26  typedef unsigned long  COUNT;  /* unsigned counter */
```

[1–6] We include `apue.h` because we use some of the functions from our private library. In turn, `apue.h` includes several standard header files, including `<stdio.h>` and `<unistd.h>`. We include `<stdarg.h>` because the `db_open` function uses the variable-argument functions declared by `<stdarg.h>`.

[7–26] The size of an index record is specified by `IDXLEN_SZ`. We use some characters, such as colon and newline, as delimiters in the database. We use the space character as "white out" when we delete a record.

Some of the values that we have defined as constants could also be made variable, with some added complexity in the implementation. For example, we set the size of the hash table to 137 entries. A better technique would be to let the caller specify this as an argument to `db_open`, based on the expected size of the database. We would then have to store this size at the beginning of the index file.

```
27  /*
28   * Library's private representation of the database.
29   */
30  typedef struct {
31    int    idxfd;  /* fd for index file */
32    int    datfd;  /* fd for data file */
33    char  *idxbuf; /* malloc'ed buffer for index record */
34    char  *datbuf; /* malloc'ed buffer for data record*/
35    char  *name;   /* name db was opened under */
36    off_t  idxoff; /* offset in index file of index record */
37                   /* key is at (idxoff + PTR_SZ + IDXLEN_SZ) */
38    size_t idxlen; /* length of index record */
39                   /* excludes IDXLEN_SZ bytes at front of record */
40                   /* includes newline at end of index record */
41    off_t  datoff; /* offset in data file of data record */
42    size_t datlen; /* length of data record */
43                   /* includes newline at end */
44    off_t  ptrval; /* contents of chain ptr in index record */
45    off_t  ptroff; /* chain ptr offset pointing to this idx record */
46    off_t  chainoff; /* offset of hash chain for this index record */
47    off_t  hashoff;  /* offset in index file of hash table */
48    DBHASH nhash;    /* current hash table size */
49    COUNT  cnt_delok;    /* delete OK */
50    COUNT  cnt_delerr;   /* delete error */
51    COUNT  cnt_fetchok;  /* fetch OK */
52    COUNT  cnt_fetcherr; /* fetch error */
53    COUNT  cnt_nextrec;  /* nextrec */
54    COUNT  cnt_stor1;    /* store: DB_INSERT, no empty, appended */
55    COUNT  cnt_stor2;    /* store: DB_INSERT, found empty, reused */
56    COUNT  cnt_stor3;    /* store: DB_REPLACE, diff len, appended */
57    COUNT  cnt_stor4;    /* store: DB_REPLACE, same len, overwrote */
58    COUNT  cnt_storerr;  /* store error */
59  } DB;
```

[27–48] The `DB` structure is where we keep all the information for each open database. The `DBHANDLE` value that is returned by `db_open` and used by all the other functions is really just a pointer to one of these structures, but we hide that from the callers.

Since we store pointers and lengths as ASCII in the database, we convert these to numeric values and save them in the `DB` structure. We also save the hash table size even though it is fixed, just in case we decide to enhance the library to allow callers to specify the size when the database is created (see Exercise 20.7).

[49–59] The last ten fields in the `DB` structure count both successful and unsuccessful operations. If we want to analyze the performance of our database, we can write a function to return these statistics, but for now we only maintain the counters.

```
60   /*
61    * Internal functions.
62    */
63   static DB     *_db_alloc(int);
64   static void    _db_dodelete(DB *);
65   static int     _db_find_and_lock(DB *, const char *, int);
66   static int     _db_findfree(DB *, int, int);
67   static void    _db_free(DB *);
68   static DBHASH  _db_hash(DB *, const char *);
69   static char   *_db_readdat(DB *);
70   static off_t   _db_readidx(DB *, off_t);
71   static off_t   _db_readptr(DB *, off_t);
72   static void    _db_writedat(DB *, const char *, off_t, int);
73   static void    _db_writeidx(DB *, const char *, off_t, int, off_t);
74   static void    _db_writeptr(DB *, off_t, off_t);

75   /*
76    * Open or create a database.  Same arguments as open(2).
77    */
78   DBHANDLE
79   db_open(const char *pathname, int oflag, ...)
80   {
81      DB          *db;
82      int         len, mode;
83      size_t      i;
84      char        asciiptr[PTR_SZ + 1],
85                  hash[(NHASH_DEF + 1) * PTR_SZ + 2];
86                      /* +2 for newline and null */
87      struct stat statbuff;

88      /*
89       * Allocate a DB structure, and the buffers it needs.
90       */
91      len = strlen(pathname);
92      if ((db = _db_alloc(len)) == NULL)
93          err_dump("db_open: _db_alloc error for DB");
```

[60–74] We have chosen to name all the user-callable (public) functions starting with `db_` and all the internal (private) functions starting with `_db_`. The public functions were declared in the library's header file, `apue_db.h`. We declare the internal functions as `static` so they are visible only to functions residing in the same file (the file containing the library implementation).

[75–93] The `db_open` function has the same arguments as `open`(2). If the caller wants to create the database files, the optional third argument specifies the file permissions. The `db_open` function opens the index file and the data file, initializing the index file, if necessary. The function starts by calling `_db_alloc` to allocate and initialize a `DB` structure.

```
 94      db->nhash   = NHASH_DEF;/* hash table size */
 95      db->hashoff = HASH_OFF; /* offset in index file of hash table */
 96      strcpy(db->name, pathname);
 97      strcat(db->name, ".idx");

 98      if (oflag & O_CREAT) {
 99          va_list ap;

100          va_start(ap, oflag);
101          mode = va_arg(ap, int);
102          va_end(ap);

103          /*
104           * Open index file and data file.
105           */
106          db->idxfd = open(db->name, oflag, mode);
107          strcpy(db->name + len, ".dat");
108          db->datfd = open(db->name, oflag, mode);
109      } else {
110          /*
111           * Open index file and data file.
112           */
113          db->idxfd = open(db->name, oflag);
114          strcpy(db->name + len, ".dat");
115          db->datfd = open(db->name, oflag);
116      }

117      if (db->idxfd < 0 || db->datfd < 0) {
118          _db_free(db);
119          return(NULL);
120      }
```

[94–97] We continue to initialize the `DB` structure. The pathname passed in by the caller specifies the prefix of the database filenames. We append the suffix `.idx` to create the name for the database index file.

[98–108] If the caller wants to create the database files, we use the variable argument functions from `<stdarg.h>` to find the optional third argument. Then we use `open` to create and open the index file and data file. Note that the filename of the data file starts with the same prefix as the index file but has `.dat` as a suffix instead.

[109–116] If the caller doesn't specify the `O_CREAT` flag, then we're opening existing database files. In this case, we simply call `open` with two arguments.

[117–120] If an error occurs while we are opening or creating either database file, we call `_db_free` to clean up the `DB` structure and then return `NULL` to the caller. If one `open` succeeded and one failed, `_db_free` will take care of closing the open file descriptor, as we shall see shortly.

```
121     if ((oflag & (O_CREAT | O_TRUNC)) == (O_CREAT | O_TRUNC)) {
122         /*
123          * If the database was created, we have to initialize
124          * it.  Write lock the entire file so that we can stat
125          * it, check its size, and initialize it, atomically.
126          */
127         if (writew_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
128             err_dump("db_open: writew_lock error");

129         if (fstat(db->idxfd, &statbuff) < 0)
130             err_sys("db_open: fstat error");

131         if (statbuff.st_size == 0) {
132             /*
133              * We have to build a list of (NHASH_DEF + 1) chain
134              * ptrs with a value of 0.  The +1 is for the free
135              * list pointer that precedes the hash table.
136              */
137             sprintf(asciiptr, "%*d", PTR_SZ, 0);
```

[121–130] We encounter locking if the database is being created. Consider two processes trying to create the same database at about the same time. Assume that the first process calls `fstat` and is blocked by the kernel after `fstat` returns. The second process calls `db_open`, finds that the length of the index file is 0, and initializes the free list and hash chain. The second process then writes one record to the database. At this point, the second process is blocked, and the first process continues executing right after the call to `fstat`. The first process finds the size of the index file to be 0 (since `fstat` was called before the second process initialized the index file), so the first process initializes the free list and hash chain, wiping out the record that the second process stored in the database. The way to prevent this is to use locking. We use the macros `readw_lock`, `writew_lock`, and `un_lock` from Section 14.3.

[131–137] If the size of the index file is 0, we have just created it, so we need to initialize the free list and hash chain pointers it contains. Note that we use the format string `%*d` to convert a database pointer from an integer to an ASCII string. (We'll use this type of format again in `_db_writeidx` and `_db_writeptr`.) This format tells `sprintf` to take the `PTR_SZ` argument and use it as the minimum field width for the next argument, which is 0 in this instance (here we are initializing the pointers to 0, since we are creating a new database). This has the effect of forcing the string created to be at least `PTR_SZ` characters (padded on the left with spaces). In `_db_writeidx` and `_db_writeptr`, we will pass a pointer value instead of zero, but we will first verify that the pointer value isn't greater than `PTR_MAX`, to guarantee that every pointer string we write to the database occupies exactly `PTR_SZ` (7) characters.

```
138             hash[0] = 0;
139             for (i = 0; i < NHASH_DEF + 1; i++)
140                 strcat(hash, asciiptr);
141             strcat(hash, "\n");
142             i = strlen(hash);
143             if (write(db->idxfd, hash, i) != i)
144                 err_dump("db_open: index file init write error");
145         }
146         if (un_lock(db->idxfd, 0, SEEK_SET, 0) < 0)
147             err_dump("db_open: un_lock error");
148     }
149     db_rewind(db);
150     return(db);
151   }
152   /*
153    * Allocate & initialize a DB structure and its buffers.
154    */
155   static DB *
156   _db_alloc(int namelen)
157   {
158     DB      *db;
159     /*
160      * Use calloc, to initialize the structure to zero.
161      */
162     if ((db = calloc(1, sizeof(DB))) == NULL)
163         err_dump("_db_alloc: calloc error for DB");
164     db->idxfd = db->datfd = -1;             /* descriptors */
165     /*
166      * Allocate room for the name.
167      * +5 for ".idx" or ".dat" plus null at end.
168      */
169     if ((db->name = malloc(namelen + 5)) == NULL)
170         err_dump("_db_alloc: malloc error for name");
```

[138–151] We continue to initialize the newly created database. We build the hash table and write it to the index file. Then we unlock the index file, reset the database file pointers, and return a pointer to the `DB` structure as the opaque handle for the caller to use with the other database functions.

[152–164] The `_db_alloc` function is called by `db_open` to allocate storage for the `DB` structure, an index buffer, and a data buffer. We use `calloc` to allocate memory to hold the `DB` structure and ensure that it is initialized to all zeros. Since this has the side effect of setting the database file descriptors to zero, we need to reset them to –1 to indicate that they are not yet valid.

[165–170] We allocate space to hold the name of the database file. We use this buffer to create both filenames by changing the suffix to refer to either the index file or the data file, as we saw in `db_open`.

```
171     /*
172      * Allocate an index buffer and a data buffer.
173      * +2 for newline and null at end.
174      */
175     if ((db->idxbuf = malloc(IDXLEN_MAX + 2)) == NULL)
176         err_dump("_db_alloc: malloc error for index buffer");
177     if ((db->datbuf = malloc(DATLEN_MAX + 2)) == NULL)
178         err_dump("_db_alloc: malloc error for data buffer");
179     return(db);
180   }
181   /*
182    * Relinquish access to the database.
183    */
184   void
185   db_close(DBHANDLE h)
186   {
187     _db_free((DB *)h);  /* closes fds, free buffers & struct */
188   }
189   /*
190    * Free up a DB structure, and all the malloc'ed buffers it
191    * may point to.  Also close the file descriptors if still open.
192    */
193   static void
194   _db_free(DB *db)
195   {
196     if (db->idxfd >= 0)
197         close(db->idxfd);
198     if (db->datfd >= 0)
199         close(db->datfd);
```

[171–180] We allocate space for buffers for the index and data files. The buffer sizes are defined in `apue_db.h`. An enhancement to the database library would be to allow these buffers to expand as required. We could keep track of the size of these two buffers and call `realloc` whenever we find we need a bigger buffer. Finally, we return a pointer to the `DB` structure that we allocated.

[181–188] The `db_close` function is a wrapper that casts a database handle to a `DB` structure pointer, passing it to `_db_free` to release any resources and free the `DB` structure.

[189–199] The `_db_free` function is called by `db_open` if an error occurs while opening the index file or data file and is also called by `db_close` when an application is done using the database. If the file descriptor for the database index file is valid, we close it. The same is done with the file descriptor for the data file. (Recall that when we allocate a new `DB` structure in `_db_alloc`, we initialize each file descriptor to –1. If we are unable to open one of the database files, the corresponding file descriptor will still be set to –1, and we will avoid trying to close it.)

```
200     if (db->idxbuf != NULL)
201         free(db->idxbuf);
202     if (db->datbuf != NULL)
203         free(db->datbuf);
204     if (db->name != NULL)
205         free(db->name);
206     free(db);
207   }
208   /*
209    * Fetch a record.  Return a pointer to the null-terminated data.
210    */
211   char *
212   db_fetch(DBHANDLE h, const char *key)
213   {
214     DB      *db = h;
215     char    *ptr;
216     if (_db_find_and_lock(db, key, 0) < 0) {
217         ptr = NULL;             /* error, record not found */
218         db->cnt_fetcherr++;
219     } else {
220         ptr = _db_readdat(db);  /* return pointer to data */
221         db->cnt_fetchok++;
222     }
223     /*
224      * Unlock the hash chain that _db_find_and_lock locked.
225      */
226     if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
227         err_dump("db_fetch: un_lock error");
228     return(ptr);
229   }
```

[200–207] Next, we free any dynamically allocated buffers. We can safely pass a null pointer to `free`, so we don't need to check the value of each buffer pointer beforehand, but we do so anyway because we consider it better style to free only those objects that we allocated. (Not all deallocator functions are as forgiving as `free`.) Finally, we free the memory backing the `DB` structure.

[208–218] The `db_fetch` function is used to read a record given its key. We first try to find the record by calling `_db_find_and_lock`. If the record can't be found, we set the return value (`ptr`) to `NULL` and increment the count of unsuccessful record searches. Because `_db_find_and_lock` returns with the database index file locked, we can't return until we unlock it.

[219–229] If the record is found, we call `_db_readdat` to read the corresponding data record and increment the count of the successful record searches. Before returning, we unlock the index file by calling `un_lock`. Then we return a pointer to the record found (or `NULL` if the record wasn't found).

```
230   /*
231    * Find the specified record.  Called by db_delete, db_fetch,
232    * and db_store.  Returns with the hash chain locked.
233    */
234   static int
235   _db_find_and_lock(DB *db, const char *key, int writelock)
236   {
237     off_t   offset, nextoffset;
238     /*
239      * Calculate the hash value for this key, then calculate the
240      * byte offset of corresponding chain ptr in hash table.
241      * This is where our search starts.  First we calculate the
242      * offset in the hash table for this key.
243      */
244     db->chainoff = (_db_hash(db, key) * PTR_SZ) + db->hashoff;
245     db->ptroff = db->chainoff;
246     /*
247      * We lock the hash chain here.  The caller must unlock it
248      * when done.  Note we lock and unlock only the first byte.
249      */
250     if (writelock) {
251         if (writew_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
252             err_dump("_db_find_and_lock: writew_lock error");
253     } else {
254         if (readw_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
255             err_dump("_db_find_and_lock: readw_lock error");
256     }
257     /*
258      * Get the offset in the index file of first record
259      * on the hash chain (can be 0).
260      */
261     offset = _db_readptr(db, db->ptroff);
```

[230–237] The `_db_find_and_lock` function is used internally by the library to find a record given its key. We set the `writelock` parameter to a nonzero value if we want to acquire a write lock on the index file while we search for the record. If we set `writelock` to zero, we read lock the index file while we search it.

[238–256] We prepare to traverse a hash chain in `_db_find_and_lock`. We convert the key into a hash value, which we use to calculate the starting address of the hash chain in the file (`chainoff`). We wait for the lock to be granted before going through the hash chain. Note that we lock only the first byte in the start of the hash chain. This increases concurrency by allowing multiple processes to search different hash chains at the same time.

[257–261] We call `_db_readptr` to read the first pointer in the hash chain. If this returns zero, the hash chain is empty.

```
262     while (offset != 0) {
263         nextoffset = _db_readidx(db, offset);
264         if (strcmp(db->idxbuf, key) == 0)
265             break;       /* found a match */
266         db->ptroff = offset; /* offset of this (unequal) record */
267         offset = nextoffset; /* next one to compare */
268     }
269     /*
270      * offset == 0 on error (record not found).
271      */
272     return(offset == 0 ? -1 : 0);
273   }

274   /*
275    * Calculate the hash value for a key.
276    */
277   static DBHASH
278   _db_hash(DB *db, const char *key)
279   {
280     DBHASH      hval = 0;
281     char        c;
282     int         i;

283     for (i = 1; (c = *key++) != 0; i++)
284         hval += c * i;      /* ascii char times its 1-based index */
285     return(hval % db->nhash);
286   }
```

[262–268] In the `while` loop, we go through each index record on the hash chain, comparing keys. We call `_db_readidx` to read each index record. It populates the `idxbuf` field with the key of the current record. If `_db_readidx` returns zero, we've reached the last entry in the chain.

[269–273] If `offset` is zero after the loop, we've reached the end of a hash chain without finding a matching key, so we return –1. Otherwise, we found a match (and exited the loop with the `break` statement), so we return success (0). In this case, the `ptroff` field contains the address of the previous index record, `datoff` contains the address of the data record, and `datlen` contains the size of the data record. As we make our way through the hash chain, we save the previous index record that points to the current index record. We'll use this when we delete a record, since we have to modify the chain pointer of the previous record to delete the current record.

[274–286] `_db_hash` calculates the hash value for a given key. It multiplies each ASCII character times its 1-based index and divides the result by the number of hash table entries. The remainder from the division is the hash value for this key. Recall that the number of hash table entries is 137, which is a prime number. According to Knuth [1998], prime hashes generally provide good distribution characteristics.

```
287   /*
288    * Read a chain ptr field from anywhere in the index file:
289    * the free list pointer, a hash table chain ptr, or an
290    * index record chain ptr.
291    */
292   static off_t
293   _db_readptr(DB *db, off_t offset)
294   {
295     char    asciiptr[PTR_SZ + 1];

296     if (lseek(db->idxfd, offset, SEEK_SET) == -1)
297         err_dump("_db_readptr: lseek error to ptr field");
298     if (read(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
299         err_dump("_db_readptr: read error of ptr field");
300     asciiptr[PTR_SZ] = 0;       /* null terminate */
301     return(atol(asciiptr));
302   }

303   /*
304    * Read the next index record.  We start at the specified offset
305    * in the index file.  We read the index record into db->idxbuf
306    * and replace the separators with null bytes.  If all is OK we
307    * set db->datoff and db->datlen to the offset and length of the
308    * corresponding data record in the data file.
309    */
310   static off_t
311   _db_readidx(DB *db, off_t offset)
312   {
313     ssize_t             i;
314     char            *ptr1, *ptr2;
315     char            asciiptr[PTR_SZ + 1], asciilen[IDXLEN_SZ + 1];
316     struct iovec    iov[2];
```

[287–302] `_db_readptr` reads any one of three different chain pointers: (a) the pointer at the beginning of the index file that points to the first index record on the free list, (b) the pointers in the hash table that point to the first index record on each hash chain, and (c) the pointers that are stored at the beginning of each index record (whether the index record is part of a hash chain or on the free list). We convert the pointer from ASCII to a long integer before returning it. No locking is done by this function; that is up to the caller.

[303–316] The `_db_readidx` function is used to read the record at the specified offset from the index file. On success, the function will return the offset of the next record in the list. In this case, the function will populate several fields in the `DB` structure: `idxoff` contains the offset of the current record in the index file, `ptrval` contains the offset of the next index entry in the list, `idxlen` contains the length of the current index record, `idxbuf` contains the actual index record, `datoff` contains the offset of the record in the data file, and `datlen` contains the length of the data record.

```
317     /*
318      * Position index file and record the offset.  db_nextrec
319      * calls us with offset==0, meaning read from current offset.
320      * We still need to call lseek to record the current offset.
321      */
322     if ((db->idxoff = lseek(db->idxfd, offset,
323       offset == 0 ? SEEK_CUR : SEEK_SET)) == -1)
324         err_dump("_db_readidx: lseek error");

325     /*
326      * Read the ascii chain ptr and the ascii length at
327      * the front of the index record.  This tells us the
328      * remaining size of the index record.
329      */
330     iov[0].iov_base = asciiptr;
331     iov[0].iov_len  = PTR_SZ;
332     iov[1].iov_base = asciilen;
333     iov[1].iov_len  = IDXLEN_SZ;
334     if ((i = readv(db->idxfd, &iov[0], 2)) != PTR_SZ + IDXLEN_SZ) {
335         if (i == 0 && offset == 0)
336             return(-1);     /* EOF for db_nextrec */
337         err_dump("_db_readidx: readv error of index record");
338     }

339     /*
340      * This is our return value; always >= 0.
341      */
342     asciiptr[PTR_SZ] = 0;        /* null terminate */
343     db->ptrval = atol(asciiptr); /* offset of next key in chain */

344     asciilen[IDXLEN_SZ] = 0;     /* null terminate */
345     if ((db->idxlen = atoi(asciilen)) < IDXLEN_MIN ||
346       db->idxlen > IDXLEN_MAX)
347         err_dump("_db_readidx: invalid length");
```

[317–324] We start by seeking to the index file offset provided by the caller. We record the offset in the `DB` structure, so even if the caller wants to read the record at the current file offset (by setting `offset` to 0), we still need to call `lseek` to determine the current offset. Since an index record will never be stored at offset 0 in the index file, we can safely overload the value of 0 to mean "read from the current offset."

[325–338] We call `readv` to read the two fixed-length fields at the beginning of the index record: the chain pointer to the next index record and the size of the variable-length index record that follows.

[339–347] We convert the offset of the next record to an integer and store it in the `ptrval` field (this will be used as the return value for this function). Then we convert the length of the index record into an integer and save it in the `idxlen` field.

```
348     /*
349      * Now read the actual index record.  We read it into the key
350      * buffer that we malloced when we opened the database.
351      */
352     if ((i = read(db->idxfd, db->idxbuf, db->idxlen)) != db->idxlen)
353         err_dump("_db_readidx: read error of index record");
354     if (db->idxbuf[db->idxlen-1] != NEWLINE)    /* sanity check */
355         err_dump("_db_readidx: missing newline");
356     db->idxbuf[db->idxlen-1] = 0;    /* replace newline with null */
357     /*
358      * Find the separators in the index record.
359      */
360     if ((ptr1 = strchr(db->idxbuf, SEP)) == NULL)
361         err_dump("_db_readidx: missing first separator");
362     *ptr1++ = 0;                /* replace SEP with null */
363     if ((ptr2 = strchr(ptr1, SEP)) == NULL)
364         err_dump("_db_readidx: missing second separator");
365     *ptr2++ = 0;                /* replace SEP with null */
366     if (strchr(ptr2, SEP) != NULL)
367         err_dump("_db_readidx: too many separators");
368     /*
369      * Get the starting offset and length of the data record.
370      */
371     if ((db->datoff = atol(ptr1)) < 0)
372         err_dump("_db_readidx: starting offset < 0");
373     if ((db->datlen = atol(ptr2)) <= 0 || db->datlen > DATLEN_MAX)
374         err_dump("_db_readidx: invalid length");
375     return(db->ptrval);     /* return offset of next key in chain */
376 }
```

[348–356] We read the variable-length index record into the `idxbuf` field in the `DB` structure. The record should be terminated with a newline, which we replace with a null byte. If the index file is corrupt, we terminate and drop core by calling `err_dump`.

[357–367] We separate the index record into three fields: the key, the offset of the corresponding data record, and the length of the data record. The `strchr` function finds the first occurrence of the specified character in the given string. Here we look for the character that separates fields in the record (`SEP`, which we define to be a colon).

[368–376] We convert the data record offset and length into integers and store them in the `DB` structure. Then we return the offset of the next record in the hash chain. Note that we do not read the data record; that task is left to the caller. In the `db_fetch` function, for example, we don't read the data record until `_db_find_and_lock` has read the index record that matches the key that we're looking for.

```
377   /*
378    * Read the current data record into the data buffer.
379    * Return a pointer to the null-terminated data buffer.
380    */
381   static char *
382   _db_readdat(DB *db)
383   {
384     if (lseek(db->datfd, db->datoff, SEEK_SET) == -1)
385         err_dump("_db_readdat: lseek error");
386     if (read(db->datfd, db->datbuf, db->datlen) != db->datlen)
387         err_dump("_db_readdat: read error");
388     if (db->datbuf[db->datlen-1] != NEWLINE)    /* sanity check */
389         err_dump("_db_readdat: missing newline");
390     db->datbuf[db->datlen-1] = 0; /* replace newline with null */
391     return(db->datbuf);     /* return pointer to data record */
392   }
393   /*
394    * Delete the specified record.
395    */
396   int
397   db_delete(DBHANDLE h, const char *key)
398   {
399     DB      *db = h;
400     int     rc = 0;         /* assume record will be found */
401     if (_db_find_and_lock(db, key, 1) == 0) {
402         _db_dodelete(db);
403         db->cnt_delok++;
404     } else {
405         rc = -1;            /* not found */
406         db->cnt_delerr++;
407     }
408     if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
409         err_dump("db_delete: un_lock error");
410     return(rc);
411   }
```

[377–392] The `_db_readdat` function populates the `datbuf` field in the `DB` structure with the contents of the data record, expecting that the `datoff` and `datlen` fields will have been properly initialized already.

[393–411] The `db_delete` function is used to delete a record given its key. We use `_db_find_and_lock` to determine whether the record exists in the database. If it does, we call `_db_dodelete` to do the work needed to delete the record. The third argument to `_db_find_and_lock` controls whether the chain is read locked or write locked. Here we are requesting a write lock, since we will potentially change the list. Since `_db_find_and_lock` returns with the lock still held, we need to unlock it, regardless of whether the record was found.

```
412   /*
413    * Delete the current record specified by the DB structure.
414    * This function is called by db_delete and db_store, after
415    * the record has been located by _db_find_and_lock.
416    */
417   static void
418   _db_dodelete(DB *db)
419   {
420     int     i;
421     char    *ptr;
422     off_t   freeptr, saveptr;

423     /*
424      * Set data buffer and key to all blanks.
425      */
426     for (ptr = db->datbuf, i = 0; i < db->datlen - 1; i++)
427         *ptr++ = SPACE;
428     *ptr = 0;   /* null terminate for _db_writedat */
429     ptr = db->idxbuf;
430     while (*ptr)
431         *ptr++ = SPACE;

432     /*
433      * We have to lock the free list.
434      */
435     if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
436         err_dump("_db_dodelete: writew_lock error");

437     /*
438      * Write the data record with all blanks.
439      */
440     _db_writedat(db, db->datbuf, db->datoff, SEEK_SET);
```

[412–431] The `_db_dodelete` function does all the work necessary to delete a record from the database. (This function is also called by `db_store`.) Most of the function just updates two linked lists: the free list and the hash chain for this key. When a record is deleted, we set its key and data record to blanks. This fact is used by `db_nextrec`, which we'll examine later in this section.

[432–440] We call `writew_lock` to write lock the free list. This step prevents two processes that are deleting records at the same time, on two different hash chains, from interfering with each other. Since we'll add the deleted record to the free list, which changes the free-list pointer, only one process at a time can be doing this.

We write the all-blank data record by calling `_db_writedat`. Note that there is no need for `_db_writedat` to lock the data file in this case. Since `db_delete` has write locked the hash chain for this record, we know that no other process is reading or writing this particular data record.

```
441     /*
442      * Read the free list pointer.  Its value becomes the
443      * chain ptr field of the deleted index record.  This means
444      * the deleted record becomes the head of the free list.
445      */
446     freeptr = _db_readptr(db, FREE_OFF);
447     /*
448      * Save the contents of index record chain ptr,
449      * before it's rewritten by _db_writeidx.
450      */
451     saveptr = db->ptrval;
452     /*
453      * Rewrite the index record.  This also rewrites the length
454      * of the index record, the data offset, and the data length,
455      * none of which has changed, but that's OK.
456      */
457     _db_writeidx(db, db->idxbuf, db->idxoff, SEEK_SET, freeptr);
458     /*
459      * Write the new free list pointer.
460      */
461     _db_writeptr(db, FREE_OFF, db->idxoff);
462     /*
463      * Rewrite the chain ptr that pointed to this record being
464      * deleted.  Recall that _db_find_and_lock sets db->ptroff to
465      * point to this chain ptr.  We set this chain ptr to the
466      * contents of the deleted record's chain ptr, saveptr.
467      */
468     _db_writeptr(db, db->ptroff, saveptr);
469     if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
470         err_dump("_db_dodelete: un_lock error");
471 }
```

[441–461] We read the free-list pointer and then update the index record so that its next record pointer is set to the first record on the free list. (If the free list was empty, this new chain pointer is 0.) We have already cleared the key. Then we update the free-list pointer with the offset of the index record we are deleting. This means that the free list is handled on a last-in, first-out basis; that is, deleted records are added to the front of the free list (although we remove entries from the free list on a first-fit basis).

We don't have a separate free list for each file. When we add a deleted index record to the free list, the index record still points to the deleted data record. There are better ways to do this, in exchange for added complexity.

[462–471] We update the previous record in the hash chain to point to the record after the one we are deleting, thus removing the deleted record from the hash chain. Finally, we unlock the free list.

```
472   /*
473    * Write a data record.  Called by _db_dodelete (to write
474    * the record with blanks) and db_store.
475    */
476   static void
477   _db_writedat(DB *db, const char *data, off_t offset, int whence)
478   {
479     struct iovec    iov[2];
480     static char     newline = NEWLINE;

481     /*
482      * If we're appending, we have to lock before doing the lseek
483      * and write to make the two an atomic operation.  If we're
484      * overwriting an existing record, we don't have to lock.
485      */
486     if (whence == SEEK_END) /* we're appending, lock entire file */
487         if (writew_lock(db->datfd, 0, SEEK_SET, 0) < 0)
488             err_dump("_db_writedat: writew_lock error");

489     if ((db->datoff = lseek(db->datfd, offset, whence)) == -1)
490         err_dump("_db_writedat: lseek error");
491     db->datlen = strlen(data) + 1;  /* datlen includes newline */

492     iov[0].iov_base = (char *) data;
493     iov[0].iov_len  = db->datlen - 1;
494     iov[1].iov_base = &newline;
495     iov[1].iov_len  = 1;
496     if (writev(db->datfd, &iov[0], 2) != db->datlen)
497         err_dump("_db_writedat: writev error of data record");

498     if (whence == SEEK_END)
499         if (un_lock(db->datfd, 0, SEEK_SET, 0) < 0)
500             err_dump("_db_writedat: un_lock error");
501   }
```

[472–491] We call `_db_writedat` to write a data record. When we delete a record, we use `_db_writedat` to overwrite the record with blanks; `_db_writedat` doesn't need to lock the data file, because `db_delete` has write locked the hash chain for this record. Thus, no other process could be reading or writing this particular data record. When we cover `db_store` later in this section, we'll encounter the case in which `_db_writedat` is appending to the data file and has to lock it.

We seek to the location where we want to write the data record. The amount to write is the record size plus 1 byte for the terminating newline we add.

[492–501] We set up the `iovec` array and call `writev` to write the data record and newline. We can't assume that the caller's buffer has room at the end for us to append the newline, so we write the newline from a separate buffer. If we are appending a record to the file, we release the lock we acquired earlier.

```
502   /*
503    * Write an index record.  _db_writedat is called before
504    * this function to set the datoff and datlen fields in the
505    * DB structure, which we need to write the index record.
506    */
507   static void
508   _db_writeidx(DB *db, const char *key,
509                off_t offset, int whence, off_t ptrval)
510   {
511     struct iovec    iov[2];
512     char            asciiptrlen[PTR_SZ + IDXLEN_SZ + 1];
513     int             len;

514     if ((db->ptrval = ptrval) < 0 || ptrval > PTR_MAX)
515         err_quit("_db_writeidx: invalid ptr: %d", ptrval);
516     sprintf(db->idxbuf, "%s%c%lld%c%ld\n", key, SEP,
517       (long long)db->datoff, SEP, (long)db->datlen);
518     len = strlen(db->idxbuf);
519     if (len < IDXLEN_MIN || len > IDXLEN_MAX)
520         err_dump("_db_writeidx: invalid length");
521     sprintf(asciiptrlen, "%*lld%*d", PTR_SZ, (long long)ptrval,
522       IDXLEN_SZ, len);

523     /*
524      * If we're appending, we have to lock before doing the lseek
525      * and write to make the two an atomic operation.  If we're
526      * overwriting an existing record, we don't have to lock.
527      */
528     if (whence == SEEK_END)     /* we're appending */
529         if (writew_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
530           SEEK_SET, 0) < 0)
531             err_dump("_db_writeidx: writew_lock error");
```

[502–522] The `_db_writeidx` function is called to write an index record. After validating the next pointer in the chain, we create the index record and store the second half of it in `idxbuf`. We need the size of this portion of the index record to create the first half of the index record, which we store in the local variable `asciiptrlen`.

Note that we use casts to force the size of the arguments in the `sprintf` statements to match the format specifications. This is because the size of the `off_t` and `size_t` data types can vary among platforms. Even a 32-bit system can provide 64-bit file offsets, so we can't make any assumptions about the size of the `off_t` data type.

[523–531] As with `_db_writedat`, this function deals with locking only when a new index record is being appended to the index file. When `_db_dodelete` calls this function, we're rewriting an existing index record. In this case, the caller has write locked the hash chain, so no additional locking is required.

```
532     /*
533      * Position the index file and record the offset.
534      */
535     if ((db->idxoff = lseek(db->idxfd, offset, whence)) == -1)
536         err_dump("_db_writeidx: lseek error");

537     iov[0].iov_base = asciiptrlen;
538     iov[0].iov_len  = PTR_SZ + IDXLEN_SZ;
539     iov[1].iov_base = db->idxbuf;
540     iov[1].iov_len  = len;
541     if (writev(db->idxfd, &iov[0], 2) != PTR_SZ + IDXLEN_SZ + len)
542         err_dump("_db_writeidx: writev error of index record");

543     if (whence == SEEK_END)
544         if (un_lock(db->idxfd, ((db->nhash+1)*PTR_SZ)+1,
545           SEEK_SET, 0) < 0)
546             err_dump("_db_writeidx: un_lock error");
547   }

548   /*
549    * Write a chain ptr field somewhere in the index file:
550    * the free list, the hash table, or in an index record.
551    */
552   static void
553   _db_writeptr(DB *db, off_t offset, off_t ptrval)
554   {
555     char    asciiptr[PTR_SZ + 1];

556     if (ptrval < 0 || ptrval > PTR_MAX)
557         err_quit("_db_writeptr: invalid ptr: %d", ptrval);
558     sprintf(asciiptr, "%*lld", PTR_SZ, (long long)ptrval);

559     if (lseek(db->idxfd, offset, SEEK_SET) == -1)
560         err_dump("_db_writeptr: lseek error to ptr field");
561     if (write(db->idxfd, asciiptr, PTR_SZ) != PTR_SZ)
562         err_dump("_db_writeptr: write error of ptr field");
563   }
```

[532–547] We seek to the location where we want to write the index record and save this offset in the `idxoff` field of the `DB` structure. Since we built the index record in two separate buffers, we use `writev` to store it in the index file. If we were appending to the file, we release the lock we acquired before seeking. This makes the seek and the write an atomic operation from the perspective of concurrently running processes appending new records to the same database.

[548–563] `_db_writeptr` is used to write a chain pointer to the index file. We validate that the chain pointer is within bounds, then convert it to an ASCII string. We seek to the specified offset in the index file and write the pointer.

```
564   /*
565    * Store a record in the database.  Return 0 if OK, 1 if record
566    * exists and DB_INSERT specified, -1 on error.
567    */
568   int
569   db_store(DBHANDLE h, const char *key, const char *data, int flag)
570   {
571     DB      *db = h;
572     int     rc, keylen, datlen;
573     off_t   ptrval;
574     if (flag != DB_INSERT && flag != DB_REPLACE &&
575       flag != DB_STORE) {
576         errno = EINVAL;
577         return(-1);
578     }
579     keylen = strlen(key);
580     datlen = strlen(data) + 1;      /* +1 for newline at end */
581     if (datlen < DATLEN_MIN || datlen > DATLEN_MAX)
582         err_dump("db_store: invalid data length");
583     /*
584      * _db_find_and_lock calculates which hash table this new record
585      * goes into (db->chainoff), regardless of whether it already
586      * exists or not. The following calls to _db_writeptr change the
587      * hash table entry for this chain to point to the new record.
588      * The new record is added to the front of the hash chain.
589      */
590     if (_db_find_and_lock(db, key, 1) < 0) { /* record not found */
591         if (flag == DB_REPLACE) {
592             rc = -1;
593             db->cnt_storerr++;
594             errno = ENOENT;     /* error, record does not exist */
595             goto doreturn;
596         }
```

[564–582] We use `db_store` to add a record to the database. We first validate the flag value we are passed. Then we make sure that the length of the data record is valid. If it isn't, we drop core and exit. This is OK for an example, but if we were building a production-quality library, we'd return an error status instead, which would give the application a chance to recover.

[583–596] We call `_db_find_and_lock` to see if the record already exists. It is OK if the record doesn't exist and either `DB_INSERT` or `DB_STORE` is specified, or if the record already exists and either `DB_REPLACE` or `DB_STORE` is specified. Replacing an existing record implies that the keys are identical but that the data records probably differ. Note that the final argument to `_db_find_and_lock` specifies that the hash chain must be write locked, as we will probably be modifying this hash chain.

```
597         /*
598          * _db_find_and_lock locked the hash chain for us; read
599          * the chain ptr to the first index record on hash chain.
600          */
601         ptrval = _db_readptr(db, db->chainoff);

602         if (_db_findfree(db, keylen, datlen) < 0) {
603             /*
604              * Can't find an empty record big enough. Append the
605              * new record to the ends of the index and data files.
606              */
607             _db_writedat(db, data, 0, SEEK_END);
608             _db_writeidx(db, key, 0, SEEK_END, ptrval);

609             /*
610              * db->idxoff was set by _db_writeidx.  The new
611              * record goes to the front of the hash chain.
612              */
613             _db_writeptr(db, db->chainoff, db->idxoff);
614             db->cnt_stor1++;
615         } else {
616             /*
617              * Reuse an empty record. _db_findfree removed it from
618              * the free list and set both db->datoff and db->idxoff.
619              * Reused record goes to the front of the hash chain.
620              */
621             _db_writedat(db, data, db->datoff, SEEK_SET);
622             _db_writeidx(db, key, db->idxoff, SEEK_SET, ptrval);
623             _db_writeptr(db, db->chainoff, db->idxoff);
624             db->cnt_stor2++;
625         }
```

[597–601] After we call `_db_find_and_lock`, the code divides into four cases. In the first two, no record was found, so we are adding a new record. We read the offset of the first entry on the hash list.

[602–614] Case 1: we call `_db_findfree` to search the free list for a deleted record with the same size key and same size data. If no such record is found, we have to append the new record to the ends of the index and data files. We call `_db_writedat` to write the data part, `_db_writeidx` to write the index part, and `_db_writeptr` to place the new record on the front of the hash chain. We increment a count (`cnt_stor1`) of the number of times we executed this case to allow us to characterize the behavior of the database.

[615–625] Case 2: `_db_findfree` found an empty record with the correct sizes and removed it from the free list (we'll see the implementation of `_db_findfree` shortly). We write the data and index portions of the new record and add the record to the front of the hash chain as we did in case 1. The `cnt_stor2` field counts how many times we've executed this case.

```
626     } else {                        /* record found */
627         if (flag == DB_INSERT) {
628             rc = 1;     /* error, record already in db */
629             db->cnt_storerr++;
630             goto doreturn;
631         }
632         /*
633          * We are replacing an existing record.  We know the new
634          * key equals the existing key, but we need to check if
635          * the data records are the same size.
636          */
637         if (datlen != db->datlen) {
638             _db_dodelete(db);   /* delete the existing record */
639             /*
640              * Reread the chain ptr in the hash table
641              * (it may change with the deletion).
642              */
643             ptrval = _db_readptr(db, db->chainoff);
644             /*
645              * Append new index and data records to end of files.
646              */
647             _db_writedat(db, data, 0, SEEK_END);
648             _db_writeidx(db, key, 0, SEEK_END, ptrval);
649             /*
650              * New record goes to the front of the hash chain.
651              */
652             _db_writeptr(db, db->chainoff, db->idxoff);
653             db->cnt_stor3++;
654         } else {
```

[626–631] Now we reach the two cases in which a record with the same key already exists in the database. If the caller isn't replacing the record, we set the return code to indicate that a record exists, increment the count of the number of store errors, and jump to the end of the function, where we handle the common return logic.

[632–654] Case 3: an existing record is being replaced, and the length of the new data record differs from the length of the existing one. We call `_db_dodelete` to delete the existing record. Recall that this places the deleted record at the head of the free list. Then we append the new record to the ends of the data and index files by calling `_db_writedat` and `_db_writeidx`. (There are other ways to handle this case. We could try to find a deleted record that has the correct data size.) The new record is added to the front of the hash chain by calling `_db_writeptr`. The `cnt_stor3` counter in the `DB` structure records the number of times we've executed this case.

```
655             /*
656              * Same size data, just replace data record.
657              */
658             _db_writedat(db, data, db->datoff, SEEK_SET);
659             db->cnt_stor4++;
660         }
661     }
662     rc = 0;     /* OK */
663   doreturn: /* unlock hash chain locked by _db_find_and_lock */
664     if (un_lock(db->idxfd, db->chainoff, SEEK_SET, 1) < 0)
665         err_dump("db_store: un_lock error");
666     return(rc);
667   }
668   /*
669    * Try to find a free index record and accompanying data record
670    * of the correct sizes.  We're only called by db_store.
671    */
672   static int
673   _db_findfree(DB *db, int keylen, int datlen)
674   {
675     int     rc;
676     off_t   offset, nextoffset, saveoffset;
677     /*
678      * Lock the free list.
679      */
680     if (writew_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
681         err_dump("_db_findfree: writew_lock error");
682     /*
683      * Read the free list pointer.
684      */
685     saveoffset = FREE_OFF;
686     offset = _db_readptr(db, saveoffset);
```

[655–661] Case 4: An existing record is being replaced, and the length of the new data record equals the length of the existing data record. This is the easiest case; we simply rewrite the data record and increment the counter (`cnt_stor4`) for this case.

[662–667] In the normal case, we set the return code to indicate success and fall through to the common return logic. We unlock the hash chain that was locked as a result of calling `_db_find_and_lock` and return to the caller.

[668–686] The `_db_findfree` function tries to find a free index record and associated data record of the specified sizes. We need to write lock the free list to avoid interfering with any other processes using the free list. After locking the free list, we get the pointer address at the head of the list.

```
687     while (offset != 0) {
688         nextoffset = _db_readidx(db, offset);
689         if (strlen(db->idxbuf) == keylen && db->datlen == datlen)
690             break;      /* found a match */
691         saveoffset = offset;
692         offset = nextoffset;
693     }

694     if (offset == 0) {
695         rc = -1;    /* no match found */
696     } else {
697         /*
698          * Found a free record with matching sizes.
699          * The index record was read in by _db_readidx above,
700          * which sets db->ptrval.  Also, saveoffset points to
701          * the chain ptr that pointed to this empty record on
702          * the free list.  We set this chain ptr to db->ptrval,
703          * which removes the empty record from the free list.
704          */
705         _db_writeptr(db, saveoffset, db->ptrval);
706         rc = 0;

707         /*
708          * Notice also that _db_readidx set both db->idxoff
709          * and db->datoff.  This is used by the caller, db_store,
710          * to write the new index record and data record.
711          */
712     }

713     /*
714      * Unlock the free list.
715      */
716     if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
717         err_dump("_db_findfree: un_lock error");
718     return(rc);
719 }
```

[687–693] The `while` loop in `_db_findfree` goes through the free list, looking for a record with matching key and data sizes. In this simple implementation, we reuse a deleted record only if the key length and data length equal the lengths for the new record being inserted. There are a variety of better ways to reuse this deleted space, in exchange for added complexity.

[694–712] If we can't find an available record of the requested key and data sizes, we set the return code to indicate failure. Otherwise, we write the previous record's chain pointer to point to the next chain pointer value of the record we have found. This removes the record from the free list.

[713–719] Once we are done with the free list, we release the write lock. Then we return the status to the caller.

```
720   /*
721    * Rewind the index file for db_nextrec.
722    * Automatically called by db_open.
723    * Must be called before first db_nextrec.
724    */
725   void
726   db_rewind(DBHANDLE h)
727   {
728     DB      *db = h;
729     off_t   offset;

730     offset = (db->nhash + 1) * PTR_SZ;  /* +1 for free list ptr */

731     /*
732      * We're just setting the file offset for this process
733      * to the start of the index records; no need to lock.
734      * +1 below for newline at end of hash table.
735      */
736     if ((db->idxoff = lseek(db->idxfd, offset+1, SEEK_SET)) == -1)
737         err_dump("db_rewind: lseek error");
738   }

739   /*
740    * Return the next sequential record.
741    * We just step our way through the index file, ignoring deleted
742    * records.  db_rewind must be called before this function is
743    * called the first time.
744    */
745   char *
746   db_nextrec(DBHANDLE h, char *key)
747   {
748     DB      *db = h;
749     char    c;
750     char    *ptr;
```

[720–738] The `db_rewind` function is used to reset the database to "the beginning;" we set the file offset for the index file to point to the first record in the index file (immediately following the hash table). (Recall the structure of the index file from Figure 20.2.)

[739–750] The `db_nextrec` function returns the next record in the database. The return value is a pointer to the data buffer. If the caller provides a non-null value for the `key` parameter, the corresponding key is copied to this address. The caller is responsible for allocating a buffer big enough to store the key. A buffer whose size is `IDXLEN_MAX` bytes is large enough to hold any key.

Records are returned sequentially, in the order that they happen to be stored in the database file. Thus, the records are not sorted by key value. Also, because we do not follow the hash chains, we can come across records that have been deleted, but we will not return these to the caller.

```
751     /*
752      * We read lock the free list so that we don't read
753      * a record in the middle of its being deleted.
754      */
755     if (readw_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
756         err_dump("db_nextrec: readw_lock error");

757     do {
758         /*
759          * Read next sequential index record.
760          */
761         if (_db_readidx(db, 0) < 0) {
762             ptr = NULL;     /* end of index file, EOF */
763             goto doreturn;
764         }

765         /*
766          * Check if key is all blank (empty record).
767          */
768         ptr = db->idxbuf;
769         while ((c = *ptr++) != 0  &&  c == SPACE)
770             ;   /* skip until null byte or nonblank */
771     } while (c == 0);   /* loop until a nonblank key is found */

772     if (key != NULL)
773         strcpy(key, db->idxbuf);    /* return key */
774     ptr = _db_readdat(db);  /* return pointer to data buffer */
775     db->cnt_nextrec++;

776   doreturn:
777     if (un_lock(db->idxfd, FREE_OFF, SEEK_SET, 1) < 0)
778         err_dump("db_nextrec: un_lock error");
779     return(ptr);
780   }
```

[751–756] We first need to read lock the free list so that no other processes can remove a record while we are reading it.

[757–771] We call `_db_readidx` to read the next record. We pass in an offset of 0 to tell `_db_readidx` to continue reading from the current offset. Since we are reading the index file sequentially, we can come across records that have been deleted. We want to return only valid records, so we skip any record whose key is all spaces (recall that `_db_dodelete` clears a key by setting it to all spaces).

[772–780] When we find a valid key, we copy it to the caller's buffer if one was supplied. Then we read the data record and set the return value to point to the internal buffer containing the data record. We increment a statistics counter, unlock the free list, and return the pointer to the data record.

The normal use of db_rewind and db_nextrec is in a loop of the form

```
db_rewind(db);
while ((ptr = db_nextrec(db, key)) != NULL) {
    /* process record */
}
```

As we warned earlier, there is no order to the returned records; they are not in key order.

If the database is being modified while db_nextrec is called from a loop, the records returned by db_nextrec are simply a snapshot of a changing database at some point in time. db_nextrec always returns a "correct" record when it is called; that is, it won't return a record that was deleted. But it is possible for a record returned by db_nextrec to be deleted immediately after db_nextrec returns. Similarly, if a deleted record is reused right after db_nextrec skips over the deleted record, we won't see that new record unless we rewind the database and go through it again. If it's important to obtain an accurate "frozen" snapshot of the database using db_nextrec, then no insertions or deletions can be going on at the same time.

Look at the locking used by db_nextrec. We're not going through any hash chain, and we can't determine the hash chain that a record belongs on. Therefore, it is possible for an index record to be in the process of being deleted when db_nextrec is reading the record. To prevent this race, db_nextrec read locks the free list, thereby avoiding any interaction with _db_dodelete and _db_findfree.

Before we conclude our study of the db.c source file, we need to describe the locking when new index records or data records are appended to the end of the file. In cases 1 and 3, db_store calls both _db_writeidx and _db_writedat with a third argument of 0 and a fourth argument of SEEK_END. This fourth argument is the flag to these two functions, indicating that the new record is being appended to the file. The technique used by _db_writeidx is to write lock the index file from the end of the hash chain to the end of file. This won't interfere with any other readers or writers of the database (since they will lock a hash chain), but it does prevent other callers of db_store from trying to append at the same time. The technique used by _db_writedat is to write lock the entire data file. Again, this won't interfere with other readers or writers of the database (since they don't even try to lock the data file), but it does prevent other callers of db_store from trying to append to the data file at the same time. (See Exercise 20.3.)

20.9 Performance

We wrote a test program to test the database library and to obtain some timing measurements of the database access patterns of typical applications. This program takes two command-line arguments: the number of children to create and the number of database records (*nrec*) for each child to write to the database. The program then creates an empty database (by calling db_open), forks the number of child processes, and waits for all the children to terminate. Each child performs the following steps.

1. Write *nrec* records to the database.
2. Read the *nrec* records back by key value.
3. Perform the following loop *nrec* × 5 times.
   a. Read a random record.
   b. Every 37 times through the loop, delete a random record.
   c. Every 11 times through the loop, insert a new record and read the record back.
   d. Every 17 times through the loop, replace a random record with a new record. Every other one of these replacements is a record with the same size data; the alternate is a record with a longer data portion.
4. Delete all the records that this child wrote. Every time a record is deleted, ten random records are looked up.

The number of operations performed on the database is counted by the `cnt_xxx` variables in the `DB` structure, which were incremented in the functions. The number of operations differs from one child to the next, since the random-number generator used to select records is initialized in each child to the child's process ID. A typical count of the operations performed in each child is shown in Figure 20.6.

Operation	Calls to `fcntl` (per operation)		Operation count (*nrec* = 2,000)
	Coarse-grained locking	Fine-grained locking	
`db_store`, `DB_INSERT`, no empty record, appended	2	8	2,920
`db_store`, `DB_INSERT`, empty record reused	2	4	468
`db_store`, `DB_REPLACE`, different data length, appended	2	8	405
`db_store`, `DB_REPLACE`, equal data length	2	2	416
`db_store`, record not found	2	2	71
`db_fetch`, record found	2	2	32,873
`db_fetch`, record not found	2	2	2,966
`db_delete`, record found	2	4	3,388
`db_delete`, record not found	2	2	422

Figure 20.6 Typical count of operations performed by each child

We performed about ten times more fetches than stores or deletions, which is probably typical of many database applications.

Each child is performing these operations (fetching, storing, and deleting) only with the records that the child wrote. The concurrency controls are being exercised because all the children are operating on the same database (albeit different records in the same database). The total number of records in the database increases in proportion to the number of children. (With one child, *nrec* records are originally written to the database. With two children, *nrec* × 2 records are originally written, and so on.)

To test the concurrency provided by coarse-grained locking versus fine-grained locking and to compare the three types of locking (no locking, advisory locking, and

mandatory locking), we ran three versions of the test program. The first version used the source code shown in Section 20.8, which we've called fine-grained locking. The second version changed the locking calls to implement coarse-grained locking, as described in Section 20.6. The third version had all locking calls removed, so we could measure the overhead involved in locking. We can run the first and second versions (fine-grained locking and coarse-grained locking) using either advisory or mandatory locking, by changing the permission bits on the database files. (In all the tests reported in this section, we measured the times for mandatory locking using only the implementation of fine-grained locking.)

All the timing tests in this section were done on an Intel Core-i5 system running Linux 3.2.0. This system has four cores, which allows up to four processes to run concurrently.

Single-Process Results

Figure 20.7 shows the results when only a single child process ran, with an *nrec* of 2,000, 6,000, and 12,000.

nrec	No locking			Advisory locking						Mandatory locking		
				Coarse-grained locking			Fine-grained locking			Fine-grained locking		
	User	Sys	Clock	User	Sys	Clock	User	Sys	Clock	User	Sys	Clock
2,000	0.10	0.22	0.33	0.17	0.33	0.51	0.13	0.38	0.51	0.14	0.43	0.58
6,000	0.59	1.32	1.91	0.88	2.13	3.03	0.90	2.14	3.05	0.99	2.52	3.53
12,000	4.37	9.58	13.97	5.38	12.60	18.01	5.34	12.63	18.01	5.53	15.03	20.60

Figure 20.7 Single child, varying *nrec*, different locking techniques

The last 12 columns give the corresponding times in seconds. In all cases, the user CPU time plus the system CPU time approximately equals the clock time. This set of tests was CPU limited and not disk limited.

The six columns under "Advisory locking" are almost equal for each row. This makes sense because for a single process; there is no difference between coarse-grained locking and fine-grained locking, except for the extra calls to `fcntl`.

Comparing no locking with advisory locking, we see that adding the locking calls increases the system CPU time by 32% to 73%. Even though the locks are never used (since only a single process is running), the system call overhead in the calls to `fcntl` adds time. Also note that the user CPU time is about the same for all four versions of locking. Since the user code is almost equivalent (except for the number of calls to `fcntl`), this makes sense.

The final point to note from Figure 20.7 is that mandatory locking increases the system CPU time by 13% to 19% compared to advisory locking. Since the number of locking calls is the same for advisory fine-grained locking and mandatory fine-grained locking, the additional system call overhead must be from the reads and writes.

The next test was to try the no-locking program with multiple children. The results, as expected, were random errors. Normally, records that were added to the database couldn't be found, and the test program aborted. Different errors occurred every time the test program was run. This illustrates a classic race condition: multiple processes updating the same file without using any form of locking.

Multiple-Process Results

The final set of measurements looks mainly at the differences between coarse-grained locking and fine-grained locking. As we said earlier, intuitively, we expect fine-grained locking to provide additional concurrency, since there is less time that portions of the database are locked from other processes. Figure 20.8 shows the results for an *nrec* of 2,000, varying the number of children from 1 to 16.

# Proc	Advisory locking							Mandatory locking			
	Coarse-grained locking			Fine-grained locking			Δ Clock	Fine-grained locking			Δ Sys
	User	Sys	Clock	User	Sys	Clock	Percent	User	Sys	Clock	Percent
1	0.14	0.35	0.50	0.14	0.35	0.50	0	0.15	0.42	0.58	20
2	0.60	1.43	1.88	0.54	1.36	1.10	71	0.65	2.01	1.59	48
3	0.97	2.67	3.18	1.37	3.73	2.20	45	1.62	5.67	3.28	52
4	2.38	6.17	5.59	2.83	8.15	4.07	37	3.29	12.35	6.31	52
5	3.72	10.17	8.37	4.28	11.86	6.09	37	4.96	18.47	9.49	56
6	5.02	14.52	11.52	6.04	17.46	8.89	30	6.66	26.38	13.22	51
7	7.00	20.16	15.84	8.06	23.23	11.88	33	9.12	36.13	18.09	56
8	9.12	26.20	20.31	10.50	30.50	15.48	31	11.81	47.20	23.49	55
9	11.60	33.91	25.64	13.40	37.80	19.29	33	14.54	60.23	29.66	59
10	14.28	42.24	31.35	16.39	47.01	23.74	32	17.84	74.05	36.27	58
11	17.37	51.12	37.50	19.71	56.59	28.57	31	21.57	90.14	44.10	59
12	20.70	60.48	44.24	23.47	66.10	33.34	33	25.57	108.94	53.11	65
13	25.13	70.67	51.96	27.70	77.76	39.21	33	29.71	133.31	63.07	71
14	28.40	82.23	59.88	32.34	91.45	46.22	30	34.22	155.80	73.86	70
15	32.23	94.26	68.30	36.32	102.97	51.82	32	39.05	180.66	84.14	75
16	37.24	107.87	78.67	42.17	118.20	59.72	32	44.11	208.28	96.82	76

Figure 20.8 Comparison of various locking techniques, *nrec* = 2,000

All times are in seconds and are the total for the parent and all its children. There are many items to consider from this data.

The first thing to notice is that the sum of the user and system times exceeds the clock time when multiple processes are used. This seems odd at first, but is normal when multiple cores are present. What happens is that all concurrently executing processes accumulate time as they execute; the CPU processing times shown are the sum of the times of all the cores used by the program. Because we can run multiple processes at the same time (one per core), the CPU processing times can exceed the clock time.

The eighth column, labeled "Δ Clock," is the percentage difference between the clock times from advisory coarse-grained locking and advisory fine-grained locking. This is a measurement of how much concurrency we obtain by going from coarse-

grained locking to fine-grained locking. On the system used for these tests, coarse-grained locking is the same as fine-grained locking for one process, but becomes more expensive (by about 30%) with multiple processes.

We would like the clock time to decrease from coarse-grained to fine-grained locking, which it does as soon as we start using multiple processes. However, we expect the system time to remain higher for fine-grained locking for any number of processes, because we are issuing more `fcntl` calls with fine-grained locking than with coarse-grained locking. If we total the number of `fcntl` calls in Figure 20.6, we have an average of 87,858 for coarse-grained locking and 115,520 for fine-grained locking. We expect this increase of 31% more calls to `fcntl` to result in increased system time for fine-grained locking. Therefore, the decrease in system time for fine-grained locking with two processes, and the relatively small increase with more than two processes, is puzzling.

There are two reasons for this behavior. First, recall from Figure 20.7 that there is no significant difference between coarse-grained locking times and fine-grained locking times when there is no contention for the locks. This shows that the CPU overhead of the extra `fcntl` calls doesn't affect the performance of the test program. The second reason is that with coarse-grained locking, we hold locks for longer periods of time, thus increasing the likelihood that other processes will block on a lock. With fine-grained locking, the locking is done over shorter intervals, so there is less chance that processes will block. If we count the number of times `fcntl` blocks, we will see that processes block more frequently with coarse-grained locking. For example, with four processes, coarse-grained locking blocks almost five times more frequently than with fine-grained locking. The extra work that processes need to do to put themselves to sleep and wake up more often with coarse-grained locking increases the system time, reducing the difference in system times between the two locking approaches.

The final column in Figure 20.8, labeled "Δ Sys," is the percentage increase in the system CPU time from advisory fine-grained locking to mandatory fine-grained locking. These percentages show that mandatory locking adds significantly (between 20% and 76%) to the system time as concurrency increases.

Since the user code for all these tests is almost identical (there are some additional `fcntl` calls for both advisory fine-grained and mandatory fine-grained locking), we expect the user CPU times to be the same across any row.

> The first time we ran these tests, we measured the user times for coarse-grained locking to be almost twice as long as the times for fine-grained locking when multiple processes competed for the locks. Because the two versions of the database are the same, except for the number of calls to `fcntl`, this made no sense. After investigating, we discovered that because there was more contention with coarse-grained locking, processes were waiting longer, and the operating system decided to reduce the CPU clock frequency to save power. With fine-grained locking, there was more activity, so the system increased the CPU clock frequency. This (artificially) made the coarse-grained locking tests run more slowly than the fine-grained tests. After disabling the frequency scaling feature, we measured the performance of the test without this bias, and the difference in user times was much smaller.

The values in the first row of Figure 20.8 are similar to those for an *nrec* of 2,000 in Figure 20.7. This corresponds to our expectation.

Figure 20.9 is a graph of the data from Figure 20.8 for advisory fine-grained locking. We plot the clock time as the number of processes goes from 1 to 16. We also plot the user CPU time divided by the number of processes and the system CPU time divided by the number of processes.

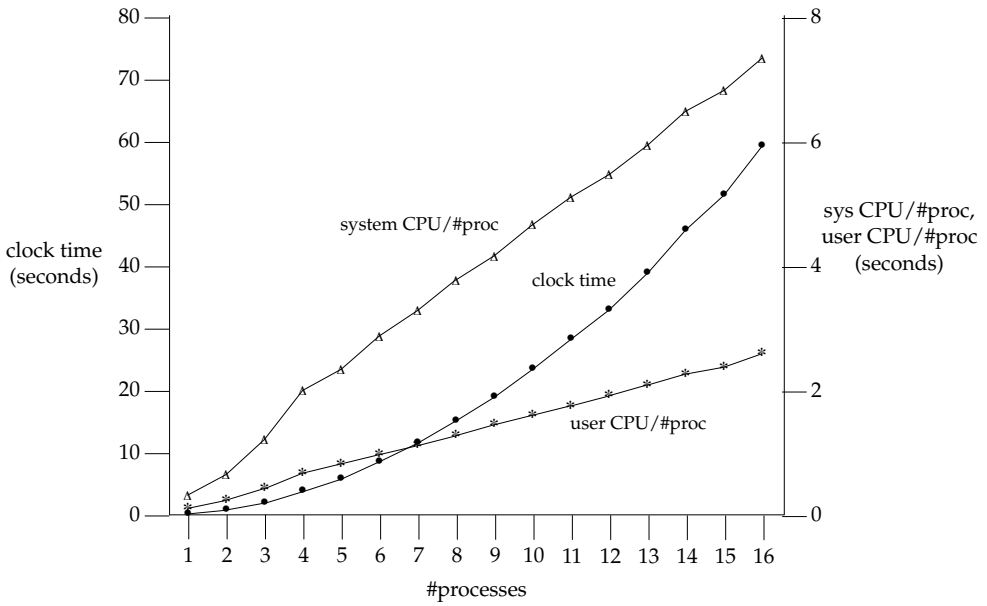


Figure 20.9 Values from Figure 20.8 for advisory fine-grained locking

Note that both CPU times, divided by the number of processes, are linear but that the plot of the clock time is nonlinear. The probable reason is the added amount of CPU time used by the operating system to switch between processes as the number of processes increases. This operating system overhead would show up as an increased clock time, but shouldn't affect the CPU times of the individual processes.

The reason the user CPU time increases with the number of processes is that there are more records in the database. Each hash chain is getting longer, so it takes the `_db_find_and_lock` function longer, on average, to find a record.

20.10 Summary

This chapter has taken a long look at the design and implementation of a database library. Although we've kept the library small and simple for presentation purposes, it contains the record locking required to allow concurrent access by multiple processes.

We've also looked at the performance of this library with various numbers of processes using no locking, advisory locking (fine-grained and coarse-grained), and mandatory locking. With a single process, we saw that advisory locking adds between

29% and 59% to the clock time over no locking and that mandatory locking adds about another 15% over advisory locking.

Exercises

20.1 The locking in `_db_dodelete` is somewhat conservative. For example, we could allow more concurrency by not write locking the free list until we really need to; that is, the call to `writew_lock` could be moved between the calls to `_db_writedat` and `_db_readptr`. What happens if we do this?

20.2 If `db_nextrec` did not read lock the free list and a record that it was reading was also in the process of being deleted, describe how `db_nextrec` could return the correct key but an all-blank (hence incorrect) data record. (Hint: Look at `_db_dodelete`.)

20.3 At the end of Section 20.8, we described the locking performed by `_db_writeidx` and `_db_writedat`. We said that this locking didn't interfere with other readers and writers except those making calls to `db_store`. Is this true if mandatory locking is being used?

20.4 How would you integrate the `fsync` function into this database library?

20.5 In `db_store`, we write the data record before the index record. What happens if you do it in the opposite order?

20.6 Create a new database and write some number of records to the database. Write a program that calls `db_nextrec` to read each record in the database, and call `_db_hash` to calculate the hash value for each record. Print a histogram of the number of records on each hash chain. Is the hashing function in `_db_hash` adequate?

20.7 Modify the database functions so that the number of hash chains in the index file can be specified when the database is created.

20.8 Compare the performance of the database functions when the database is (a) on the same host as the test program and (b) on a different host accessed via NFS. Does the record locking provided by the database library still work?

20.9 The database reuses free list records only if the sizes of the key buffer and data buffer match the needed sizes exactly. Modify the database to allow larger buffer sizes on the free list to satisfy the request. How do you have to change the persistent format of the database to support this feature?

20.10 After implementing a solution to Exercise 20.9, write a tool to convert one database format to the other.

This page intentionally left blank

21

Communicating with a Network Printer

21.1 Introduction

We now develop a program that can communicate with a network printer. These printers are connected to multiple computers via Ethernet and often support PostScript files as well as plain text files. Applications generally use the Internet Printing Protocol (IPP) to communicate with these printers, although some support alternative communication protocols.

We are about to describe two programs: a print spooler daemon that sends jobs to a printer and a command to submit print jobs to the spooler daemon. Since the print spooler has to do multiple things (e.g., communicate with clients submitting jobs, communicate with the printer, read files, scan directories), this gives us a chance to use many of the functions from earlier chapters. For example, we use threads (Chapters 11 and 12) to simplify the design of the print spooler and sockets (Chapter 16) to communicate between the program used to schedule a file to be printed and the print spooler, and also between the print spooler and the network printer.

21.2 The Internet Printing Protocol

IPP specifies the communication rules for building network-based printing systems. By embedding an IPP server inside a printer with an Ethernet card, the printer can service requests from many computer systems. These computer systems need not be located on the same physical network, however. IPP is built on top of standard Internet protocols, so any computer that can create a TCP/IP connection to the printer can submit a print job.

IPP is specified in a series of documents (Requests For Comments, or RFCs) available at `http://www.ietf.org/rfc.html`. Proposed draft standards are developed by the Printer Working Group, which is associated with the IEEE. These drafts are available at `http://www.pwg.org/ipp`. The main documents are listed in Figure 21.1, although many other documents are available to further specify administrative procedures, job attributes, and the like.

Document	Title
RFC 2567	Design Goals for an Internet Printing Protocol
RFC 2568	Rationale for the Structure of the Model and Protocol for the Internet Printing Protocol
RFC 2911	Internet Printing Protocol/1.1: Model and Semantics
RFC 2910	Internet Printing Protocol/1.1: Encoding and Transport
RFC 3196	Internet Printing Protocol/1.1: Implementor's Guide
Candidate Standard 5100.12-2011	Internet Printing Protocol Version 2.0, Second Edition

Figure 21.1 Primary IPP documents

Candidate Standard 5100.12-2011 specifies all features that implementations must support to conform to different versions of the IPP standard. There are many proposed extensions to the IPP protocol (specific features are defined in other IPP-related documents). These features are divided into groups to create different conformance levels; each level is a different protocol version. For compatibility, each higher level of conformance requires that implementations meet most of the requirements defined by lower versions of the standard. In this chapter, we will use IPP version 1.1 in our simple example.

IPP is built on top of HTTP, the Hypertext Transfer Protocol (Section 21.3). HTTP, in turn, is built on top of TCP/IP. Figure 21.2 shows the structure of an IPP message.

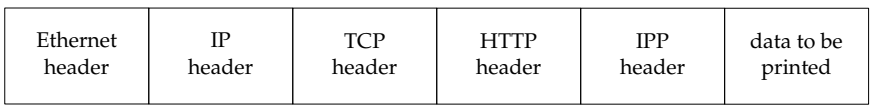


Figure 21.2 Structure of an IPP message

IPP is a request–response protocol. A client sends a request message to a server, and the server answers with a response message. The IPP header contains a field that indicates the requested operation. Operations are defined to submit print jobs, cancel print jobs, get job attributes, get printer attributes, pause and restart the printer, place a job on hold, and release a held job.

Figure 21.3 shows the structure of an IPP message header. The first 2 bytes are the IPP version number. For protocol version 1.1, each byte has a value of 1. For a protocol request, the next 2 bytes contain a value identifying the type of operation requested. For a protocol response, these 2 bytes contain a status code instead.

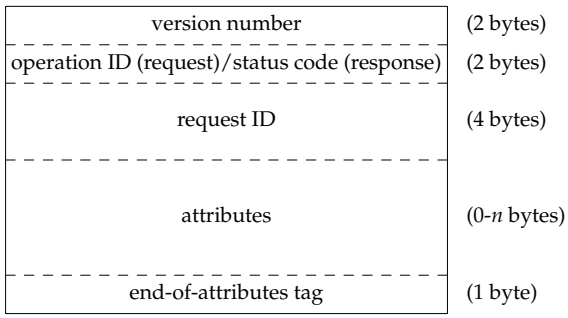


Figure 21.3 Structure of an IPP header

The next 4 bytes contain an integer identifying the request, which allows requests to be matched up with responses. Optional attributes follow this, terminated by an end-of-attributes tag. Any data that might be associated with the request follows immediately after the end-of-attributes tag.

In the header, integers are stored as signed, two's-complement, binary values in big-endian byte order (i.e., network byte order). Attributes are stored in groups. Each group starts with a single byte identifying the group. Within each group, an attribute is generally represented as a 1-byte tag, followed by a 2-byte name length, followed by the name of the attribute, followed by a 2-byte value length, and finally by the value itself. The value can be encoded as a string, a binary integer, or a more complex structure, such as a date/timestamp.

Figure 21.4 shows how the `attributes-charset` attribute would be encoded with a value of `utf-8`.

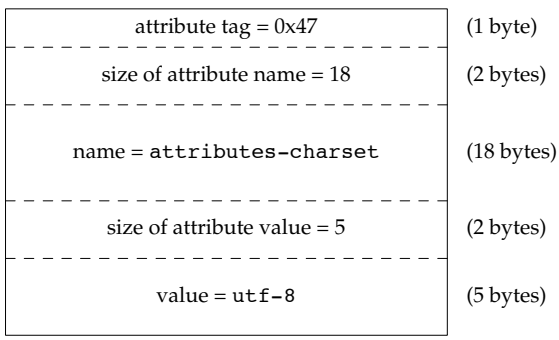


Figure 21.4 Sample IPP attribute encoding

Depending on the operation requested, some attributes are required to be provided in the request message, whereas others are optional. For example, Figure 21.5 shows the attributes defined for a print-job request.

Attribute	Status	Description
attributes-charset	required	the character set used by attributes of type text or name
attributes-natural-language	required	the natural language used by attributes of type text or name
printer-uri	required	the printer's Universal Resource Identifier
requesting-user-name	optional	name of user submitting job (used for authentication, if enabled)
job-name	optional	name of job used to distinguish between multiple jobs
ipp-attribute-fidelity	optional	if true, tells printer to reject job if all attributes can't be met; otherwise, printer does its best to print the job
document-name	optional	the name of the document (suitable for printing in a banner, for example)
document-format	optional	the format of the document (e.g., plaintext, PostScript)
document-natural-language	optional	the natural language of the document
compression	optional	the algorithm used to compress the document data
job-k-octets	optional	size of the document in 1,024-octet units
job-impressions	optional	number of impressions (images imposed on a page) submitted in this job
job-media-sheets	optional	number of sheets printed by this job

Figure 21.5 Attributes of print-job request

The IPP header contains a mixture of text and binary data. Attribute names are stored as text, but sizes are stored as binary integers. This complicates the process of building and parsing the header, since we need to worry about such things as network byte order and our host processor's ability to address an integer on an arbitrary byte boundary. A better alternative would have been to design the header to contain text only. This simplifies processing at the cost of slightly larger protocol messages.

21.3 The Hypertext Transfer Protocol

Version 1.1 of HTTP is specified in RFC 2616. HTTP is also a request–response protocol. A request message contains a start line, followed by header lines, a blank line, and an optional entity body. The entity body contains the IPP header and data in this case.

HTTP headers are ASCII, with each line terminated by a carriage return (\r) and a line feed (\n). The start line consists of a *method* that indicates which operation the client is requesting, a Uniform Resource Locator (URL) that describes the server and protocol, and a string indicating the HTTP version. The only method used by IPP is POST, which is used to send data to a server.

The header lines specify attributes, such as the format and length of the entity body. A header line consists of an attribute name followed by a colon, optional white space, and the attribute value, and is terminated by a carriage return and a line feed. For example, to specify that the entity body contains an IPP message, we include the header line

```
Content-Type: application/ipp
```

The following is a sample HTTP header for a print request submitted to the author's Xerox Phaser 8560 printer:

```
POST /ipp HTTP/1.1^M
Content-Length: 21931^M
Content-Type: application/ipp^M
Host: phaser8560:631^M
^M
```

The `Content-Length` line specifies the size in bytes of the amount of data in the HTTP message. This excludes the size of the HTTP header, but includes the size of the IPP header. The `Host` line specifies the host name and port number of the server to which the message is being sent.

The `^M` at the end of the each line is the carriage return that precedes the line feed. The line feed doesn't show up as a printable character. Note that the last line of the header is empty, except for the carriage return and line feed.

The start line in an HTTP response message contains a version string followed by a numeric status code and a status message, terminated by a carriage return and a line feed. The remainder of the HTTP response message has the same format as the request message: headers followed by a blank line and an optional entity body.

In response to a print request, the printer might send us the following message:

```
HTTP/1.1 200 OK^M
Content-Type: application/ipp^M
Cache-Control: no-cache, no-store, must-revalidate^M
Expires: THU, 26 OCT 1995 00:00:00 GMT^M
Content-Length: 215^M
Server: Allegro-Software-RomPager/4.34^M
^M
```

As far as our print spooler is concerned, all we care about in this message is the first line: it tells us whether the request succeeded or failed using a numeric error code and a short string. The remainder of the message contains additional information to control caching by nodes that might sit in between the client and the server and to indicate the software version running on the server.

21.4 Printer Spooling

The programs that we develop in this chapter form the basis of a simple printer spooler. A simple user command sends a file to the printer spooler; the spooler saves it to disk, queues the request, and ultimately sends the file to the printer.

All UNIX Systems provide at least one print spooling system. FreeBSD ships LPD, the BSD print spooling system (see `lpd`(8) and Chapter 13 of Stevens [1990]). Linux and Mac OS X include CUPS, the Common UNIX Printing System (see `cupsd`(8)). Solaris ships with the standard System V printer spooler (see `lp`(1) and `lpsched`(1M)). In this

chapter, our interest is not in these spooling systems per se, but rather in communicating with a network printer. We need to develop a spooling system to solve the problem of multiuser access to a single resource (the printer).

We use a simple command that reads a file and sends it to the printer spooler daemon. The command has one option to force the file to be treated as plaintext (the default assumes that the file is PostScript). We call this command `print`.

In our printer spooler daemon, `printd`, we use multiple threads to divide up the work that the daemon needs to accomplish.

- One thread listens on a socket for new print requests arriving from clients running the `print` command.
- A separate thread is spawned for each client to copy the file to be printed to a spooling area.
- One thread communicates with the printer, sending it queued jobs one at a time.
- One thread handles signals.

Figure 21.6 shows how these components fit together.

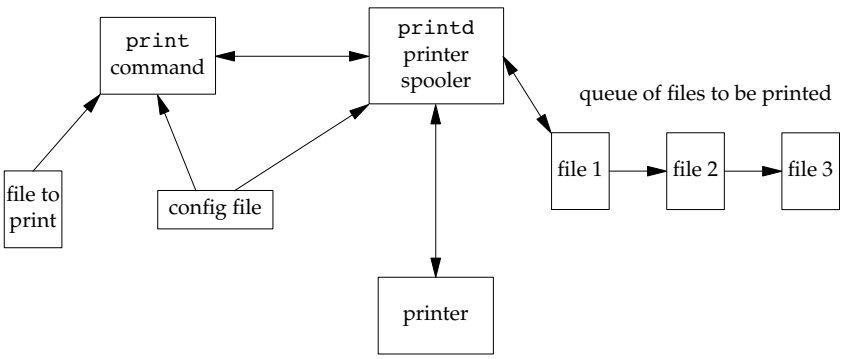


Figure 21.6 Printer spooling components

The print configuration file is `/etc/printer.conf`. It identifies the host name of the server running the printer spooling daemon and the host name of the network printer. The spooling daemon is identified by a line starting with the `printserver` keyword, followed by white space and the host name of the server. The printer is identified by a line starting with the `printer` keyword, followed by white space and the host name of the printer.

A sample printer configuration file might contain the following lines:

```
printserver   fujin
printer       phaser8560
```

where `fujin` is the host name of the computer system running the printer spooling daemon, and `phaser8560` is the host name of the network printer. We assume these

names are listed in `/etc/hosts` or registered with whatever name service we are using, so that we can translate the names to network addresses.

We can run the `print` command on the same machine where the printer spooling daemon is running, or we can run it from any machine on the same network. We only need to configure the `printserver` field in `/etc/printer.conf` in the latter case, because only the daemon needs to know the name of the printer.

Security

Programs that run with superuser privileges have the potential to open a computer system up to attack. Such programs usually aren't more vulnerable than any other program, but when compromised can lead to attackers obtaining full access to your system.

The printer spooling daemon in this chapter starts out with superuser privileges in this example to be able to bind a socket to a privileged TCP port number. To make the daemon less vulnerable to attack, we can

- Design the daemon to conform to the principles of least privilege (Section 8.11). After we obtain a socket bound to a privileged port address, we can change the user and group IDs of the daemon to something other than `root` (`lp`, for example). All the files and directories used to store queued print jobs should be owned by this nonprivileged user. This way, the daemon, if compromised, will provide the attacker with access only to the printing subsystem. This is still a concern, but it is far less serious than an attacker getting full access to your system.
- Audit the daemon's source code for all known potential vulnerabilities, such as buffer overruns.
- Log unexpected or suspicious behavior so that an administrator can take note and investigate further.

21.5 Source Code

The source code for this chapter comprises five files, not including some of the common library routines we've used in earlier chapters:

`ipp.h`	Header file containing IPP definitions
`print.h`	Header containing common constants, data structure definitions, and utility routine declarations
`util.c`	Utility routines used by the two programs
`print.c`	The C source file for the command used to print a file
`printd.c`	The C source file for the printer spooling daemon

We will study each file in the order listed.

We start with the `ipp.h` header file.

```
 1  #ifndef _IPP_H
 2  #define _IPP_H
 3  /*
 4   * Defines parts of the IPP protocol between the scheduler
 5   * and the printer.  Based on RFC2911 and RFC2910.
 6   */
 7  /*
 8   * Status code classes.
 9   */
10  #define STATCLASS_OK(x)     ((x) >= 0x0000 && (x) <= 0x00ff)
11  #define STATCLASS_INFO(x)   ((x) >= 0x0100 && (x) <= 0x01ff)
12  #define STATCLASS_REDIR(x)  ((x) >= 0x0300 && (x) <= 0x03ff)
13  #define STATCLASS_CLIERR(x) ((x) >= 0x0400 && (x) <= 0x04ff)
14  #define STATCLASS_SRVERR(x) ((x) >= 0x0500 && (x) <= 0x05ff)
15  /*
16   * Status codes.
17   */
18  #define STAT_OK             0x0000  /* success */
19  #define STAT_OK_ATTRIGN     0x0001  /* OK; some attrs ignored */
20  #define STAT_OK_ATTRCON     0x0002  /* OK; some attrs conflicted */
21  #define STAT_CLI_BADREQ     0x0400  /* invalid client request */
22  #define STAT_CLI_FORBID     0x0401  /* request is forbidden */
23  #define STAT_CLI_NOAUTH     0x0402  /* authentication required */
24  #define STAT_CLI_NOPERM     0x0403  /* client not authorized */
25  #define STAT_CLI_NOTPOS     0x0404  /* request not possible */
26  #define STAT_CLI_TIMOUT     0x0405  /* client too slow */
27  #define STAT_CLI_NOTFND     0x0406  /* no object found for URI */
28  #define STAT_CLI_OBJGONE    0x0407  /* object no longer available */
29  #define STAT_CLI_TOOBIG     0x0408  /* requested entity too big */
30  #define STAT_CLI_TOOLNG     0x0409  /* attribute value too large */
31  #define STAT_CLI_BADFMT     0x040a  /* unsupported doc format */
32  #define STAT_CLI_NOTSUP     0x040b  /* attributes not supported */
33  #define STAT_CLI_NOSCHM     0x040c  /* URI scheme not supported */
34  #define STAT_CLI_NOCHAR     0x040d  /* charset not supported */
35  #define STAT_CLI_ATTRCON    0x040e  /* attributes conflicted */
36  #define STAT_CLI_NOCOMP     0x040f  /* compression not supported */
37  #define STAT_CLI_COMPERR    0x0410  /* data can't be decompressed */
38  #define STAT_CLI_FMTERR     0x0411  /* document format error */
39  #define STAT_CLI_ACCERR     0x0412  /* error accessing data */
```

[1–14] We start the `ipp.h` header with the standard `#ifdef` to prevent errors when it is included twice in the same file. Then we define the classes of IPP status codes (see Section 13 in RFC 2911).

[15–39] We define specific status codes based on RFC 2911. We don't use these codes in the program shown here; their use is left as an exercise (See Exercise 21.1).

```
40   #define STAT_SRV_INTERN   0x0500  /* unexpected internal error */
41   #define STAT_SRV_NOTSUP   0x0501  /* operation not supported */
42   #define STAT_SRV_UNAVAIL  0x0502  /* service unavailable */
43   #define STAT_SRV_BADVER   0x0503  /* version not supported */
44   #define STAT_SRV_DEVERR   0x0504  /* device error */
45   #define STAT_SRV_TMPERR   0x0505  /* temporary error */
46   #define STAT_SRV_REJECT   0x0506  /* server not accepting jobs */
47   #define STAT_SRV_TOOBUSY  0x0507  /* server too busy */
48   #define STAT_SRV_CANCEL   0x0508  /* job has been canceled */
49   #define STAT_SRV_NOMULTI  0x0509  /* multi-doc jobs unsupported */

50   /*
51    * Operation IDs
52    */
53   #define OP_PRINT_JOB         0x02
54   #define OP_PRINT_URI         0x03
55   #define OP_VALIDATE_JOB      0x04
56   #define OP_CREATE_JOB        0x05
57   #define OP_SEND_DOC          0x06
58   #define OP_SEND_URI          0x07
59   #define OP_CANCEL_JOB        0x08
60   #define OP_GET_JOB_ATTR      0x09
61   #define OP_GET_JOBS          0x0a
62   #define OP_GET_PRINTER_ATTR  0x0b
63   #define OP_HOLD_JOB          0x0c
64   #define OP_RELEASE_JOB       0x0d
65   #define OP_RESTART_JOB       0x0e
66   #define OP_PAUSE_PRINTER     0x10
67   #define OP_RESUME_PRINTER    0x11
68   #define OP_PURGE_JOBS        0x12

69   /*
70    * Attribute Tags.
71    */
72   #define TAG_OPERATION_ATTR   0x01  /* operation attributes tag */
73   #define TAG_JOB_ATTR         0x02  /* job attributes tag */
74   #define TAG_END_OF_ATTR      0x03  /* end of attributes tag */
75   #define TAG_PRINTER_ATTR     0x04  /* printer attributes tag */
76   #define TAG_UNSUPP_ATTR      0x05  /* unsupported attributes tag */
```

[40–49] We continue to define status codes. The ones in the range `0x500` to `0x5ff` are server error codes. All codes are described in Sections 13.1.1 through 13.1.5 in RFC 2911.

[50–68] We define the various operation IDs next. There is one ID for each task defined by IPP (see Section 4.4.15 in RFC 2911). In our example, we will use only the print-job operation.

[69–76] The attribute tags delimit the attribute groups in the IPP request and response messages. The tag values are defined in Section 3.5.1 of RFC 2910.

```
 77   /*
 78    * Value Tags.
 79    */
 80   #define TAG_UNSUPPORTED      0x10 /* unsupported value */
 81   #define TAG_UNKNOWN          0x12 /* unknown value */
 82   #define TAG_NONE             0x13 /* no value */
 83   #define TAG_INTEGER          0x21 /* integer */
 84   #define TAG_BOOLEAN          0x22 /* boolean */
 85   #define TAG_ENUM             0x23 /* enumeration */
 86   #define TAG_OCTSTR           0x30 /* octetString */
 87   #define TAG_DATETIME         0x31 /* dateTime */
 88   #define TAG_RESOLUTION       0x32 /* resolution */
 89   #define TAG_INTRANGE         0x33 /* rangeOfInteger */
 90   #define TAG_TEXTWLANG        0x35 /* textWithLanguage */
 91   #define TAG_NAMEWLANG        0x36 /* nameWithLanguage */
 92   #define TAG_TEXTWOLANG       0x41 /* textWithoutLanguage */
 93   #define TAG_NAMEWOLANG       0x42 /* nameWithoutLanguage */
 94   #define TAG_KEYWORD          0x44 /* keyword */
 95   #define TAG_URI              0x45 /* URI */
 96   #define TAG_URISCHEME        0x46 /* uriScheme */
 97   #define TAG_CHARSET          0x47 /* charset */
 98   #define TAG_NATULANG         0x48 /* naturalLanguage */
 99   #define TAG_MIMETYPE         0x49 /* mimeMediaType */

100   struct ipp_hdr {
101     int8_t  major_version;  /* always 1 */
102     int8_t  minor_version;  /* always 1 */
103     union {
104         int16_t op; /* operation ID */
105         int16_t st; /* status */
106     } u;
107     int32_t request_id;     /* request ID */
108     char    attr_group[1];  /* start of optional attributes group */
109     /* optional data follows */
110   };

111   #define operation u.op
112   #define status u.st

113   #endif /* _IPP_H */
```

[77–99] The value tags indicate the format of individual attributes and parameters. They are defined in Section 3.5.2 of RFC 2910.

[100–113] We define the structure of an IPP header. Request messages start with the same header as response messages, except that the operation ID in the request is replaced by a status code in the response.

We end the header file with a `#endif` to match the `#ifdef` at the start of the file.

The next file is the `print.h` header.

```
 1   #ifndef _PRINT_H
 2   #define _PRINT_H

 3   /*
 4    * Print server header file.
 5    */
 6   #include <sys/socket.h>
 7   #include <arpa/inet.h>
 8   #include <netdb.h>
 9   #include <errno.h>

10   #define CONFIG_FILE    "/etc/printer.conf"
11   #define SPOOLDIR       "/var/spool/printer"
12   #define JOBFILE        "jobno"
13   #define DATADIR        "data"
14   #define REQDIR         "reqs"

15   #if defined(BSD)
16   #define LPNAME         "daemon"
17   #elif defined(MACOS)
18   #define LPNAME         "_lp"
19   #else
20   #define LPNAME         "lp"
21   #endif
```

[1–9] We include all header files that an application might need if it included this header. This makes it easy for applications to include `print.h` without having to track down all the header dependencies.

[10–14] We define the files and directories for the implementation. The configuration file containing the host names of the printer spooling daemon and the network-attached printer is `/etc/printer.conf`. Copies of the files to be printed will be stored in the directory `/var/spool/printer/data` and control information for each request will be stored in the directory `/var/spool/printer/reqs`. The file containing the next job number is `/var/spool/printer/jobno`.

The directories must be created by an administrator and be owned by the same user account under which the printer spooling daemon runs. The daemon won't try to create these directories if they don't exist, because the daemon would need root privileges to create directories in `/var/spool`. We design the daemon to do as little as possible while running as root to minimize the chance of creating a security hole.

[15–21] Next, we define the account name under which the printer spooling daemon will run. On Linux and Solaris, this name is `lp`. On Mac OS X, the name is `_lp`. FreeBSD, however, doesn't define a separate account for the printer spooling daemon, so we use the account reserved for system daemons.

```
22   #define FILENMSZ        64
23   #define FILEPERM        (S_IRUSR|S_IWUSR)

24   #define USERNM_MAX      64
25   #define JOBNM_MAX       256
26   #define MSGLEN_MAX      512

27   #ifndef HOST_NAME_MAX
28   #define HOST_NAME_MAX   256
29   #endif

30   #define IPP_PORT        631
31   #define QLEN            10

32   #define IBUFSZ          512    /* IPP header buffer size */
33   #define HBUFSZ          512    /* HTTP header buffer size */
34   #define IOBUFSZ         8192   /* data buffer size */

35   #ifndef ETIME
36   #define ETIME ETIMEDOUT
37   #endif

38   extern int getaddrlist(const char *, const char *,
39     struct addrinfo **);
40   extern char *get_printserver(void);
41   extern struct addrinfo *get_printaddr(void);
42   extern ssize_t tread(int, void *, size_t, unsigned int);
43   extern ssize_t treadn(int, void *, size_t, unsigned int);
44   extern int connect_retry(int, int, int, const struct sockaddr *,
45     socklen_t);
46   extern int initserver(int, const struct sockaddr *, socklen_t,
47     int);
```

[22–34] Next, we define limits and constants. `FILEPERM` is the permissions used when creating copies of files submitted to be printed. The permissions are restrictive because we don't want ordinary users to be able to read each other's files while they are waiting to be printed. We define `HOST_NAME_MAX` as the largest host name we will support if we are unable to determine the system's limit with `sysconf`.

IPP is defined to use port 631. The `QLEN` is the backlog parameter we pass to `listen` (see Section 16.4 for details).

[35–37] Some platforms don't define the error `ETIME`, so we define it to an alternate error code that makes sense for these systems. This is the error code we return when a read times out (we don't want the server to block indefinitely reading from a socket).

[38–47] Next, we declare all the public routines contained in `util.c` (we'll look at these shortly). Note that the `connect_retry` function, from Figure 16.11, and the `initserver` function, from Figure 16.22, are not included in `util.c`.

```
48  /*
49   * Structure describing a print request.
50   */
51  struct printreq {
52     uint32_t size;              /* size in bytes */
53     uint32_t flags;             /* see below */
54     char usernm[USERNM_MAX];    /* user's name */
55     char jobnm[JOBNM_MAX];      /* job's name */
56  };

57  /*
58   * Request flags.
59   */
60  #define PR_TEXT        0x01    /* treat file as plain text */

61  /*
62   * The response from the spooling daemon to the print command.
63   */
64  struct printresp {
65     uint32_t retcode;           /* 0=success, !0=error code */
66     uint32_t jobid;             /* job ID */
67     char msg[MSGLEN_MAX];       /* error message */
68  };

69  #endif /* _PRINT_H */
```

[48–69] The `printreq` and `printresp` structures define the protocol between the print command and the printer spooling daemon. The `print` command sends a `printreq` structure specifying the size of the job in bytes, job characteristics, the user name, and the job name to the printer spooling daemon. The daemon responds with a `printresp` structure containing a return code, the job ID, and an error message if the request failed.

The `PR_TEXT` job characteristic indicates that the file being printed should be treated as plaintext (instead of PostScript). We define a bitmask of flags instead of defining a separate field for each flag. Although only one flag value is currently defined, we could extend the protocol in the future to add more characteristics. For example, we could add a flag to request double-sided printing. We have room for 31 additional flags without requiring that we change the size of the structure. Changing the size of the structure means that we might introduce a compatibility problem between the client and the server unless we upgrade both at the same time. An alternative approach is to add a version number to the messages to allow the structures to change with each version.

Note that we define all integers in the protocol structures with an explicit size. This helps avoid misaligned structure elements when a client has a different long integer size than the server.

The next file we will look at is `util.c`, the file containing utility routines.

```
 1  #include "apue.h"
 2  #include "print.h"
 3  #include <ctype.h>
 4  #include <sys/select.h>

 5  #define MAXCFGLINE 512
 6  #define MAXKWLEN   16
 7  #define MAXFMTLEN  16

 8  /*
 9   * Get the address list for the given host and service and
10   * return through ailistpp.  Returns 0 on success or an error
11   * code on failure.  Note that we do not set errno if we
12   * encounter an error.
13   *
14   * LOCKING: none.
15   */
16  int
17  getaddrlist(const char *host, const char *service,
18    struct addrinfo **ailistpp)
19  {
20     int             err;
21     struct addrinfo hint;

22     hint.ai_flags = AI_CANONNAME;
23     hint.ai_family = AF_INET;
24     hint.ai_socktype = SOCK_STREAM;
25     hint.ai_protocol = 0;
26     hint.ai_addrlen = 0;
27     hint.ai_canonname = NULL;
28     hint.ai_addr = NULL;
29     hint.ai_next = NULL;
30     err = getaddrinfo(host, service, &hint, ailistpp);
31     return(err);
32  }
```

[1–7] We first define the limits needed by the functions in this file. `MAXCFGLINE` is the maximum size of a line in the printer configuration file, `MAXKWLEN` is the maximum size of a keyword in the configuration file, and `MAXFMTLEN` is the maximum size of the format string we pass to `sscanf`.

[8–32] The first function is `getaddrlist`. It is a wrapper for `getaddrinfo` (Section 16.3.3), since we always call `getaddrinfo` with the same hint structure. Note that we do not need mutex locking in this function. The `LOCKING` comment at the beginning of each function is intended only for documenting multithreaded locking. This comment lists the assumptions, if any, that are made regarding the locking, tells which locks the function might acquire or release, and tells which locks must be held to call the function.

```
33  /*
34   * Given a keyword, scan the configuration file for a match
35   * and return the string value corresponding to the keyword.
36   *
37   * LOCKING: none.
38   */
39  static char *
40  scan_configfile(char *keyword)
41  {
42      int             n, match;
43      FILE            *fp;
44      char            keybuf[MAXKWLEN], pattern[MAXFMTLEN];
45      char            line[MAXCFGLINE];
46      static char     valbuf[MAXCFGLINE];

47      if ((fp = fopen(CONFIG_FILE, "r")) == NULL)
48          log_sys("can't open %s", CONFIG_FILE);
49      sprintf(pattern, "%%%ds %%%ds", MAXKWLEN-1, MAXCFGLINE-1);
50      match = 0;
51      while (fgets(line, MAXCFGLINE, fp) != NULL) {
52          n = sscanf(line, pattern, keybuf, valbuf);
53          if (n == 2 && strcmp(keyword, keybuf) == 0) {
54              match = 1;
55              break;
56          }
57      }
58      fclose(fp);
59      if (match != 0)
60          return(valbuf);
61      else
62          return(NULL);
63  }
```

[33–46] The `scan_configfile` function searches through the printer configuration file for the specified keyword.

[47–63] We open the configuration file for reading and build the format string corresponding to the search pattern. The notation `%%%ds` builds a format specifier that limits the string size so we don't overrun the buffers used to store the strings on the stack. We read the file one line at a time and scan for two strings separated by white space; if we find them, we compare the first string with the keyword. If we find a match or we reach the end of the file, the loop ends and we close the file. If the keyword matches, we return a pointer to the buffer containing the string after the keyword; otherwise, we return `NULL`.

The string returned is stored in a static buffer (`valbuf`), which can be overwritten on successive calls. Thus, `scan_configfile` can't be called by a multithreaded application unless we take care to avoid calling it from multiple threads at the same time.

```
64   /*
65    * Return the host name running the print server or NULL on error.
66    *
67    * LOCKING: none.
68    */
69   char *
70   get_printserver(void)
71   {
72      return(scan_configfile("printserver"));
73   }
74   /*
75    * Return the address of the network printer or NULL on error.
76    *
77    * LOCKING: none.
78    */
79   struct addrinfo *
80   get_printaddr(void)
81   {
82      int             err;
83      char            *p;
84      struct addrinfo *ailist;
85      if ((p = scan_configfile("printer")) != NULL) {
86          if ((err = getaddrlist(p, "ipp", &ailist)) != 0) {
87              log_msg("no address information for %s", p);
88              return(NULL);
89          }
90          return(ailist);
91      }
92      log_msg("no printer address specified");
93      return(NULL);
94   }
```

[64–73] The `get_printserver` function is simply a wrapper function that calls `scan_configfile` to find the name of the computer system where the printer spooling daemon is running.

[74–94] We use the `get_printaddr` function to get the address of the network printer. It is similar to the previous function except that when we find the name of the printer in the configuration file, we use the name to find the corresponding network address.

Both `get_printserver` and `get_printaddr` call `scan_configfile`. If it can't open the printer configuration file, `scan_configfile` calls `log_sys` to print an error message and exit. Although `get_printserver` is meant to be called from a client command and `get_printaddr` is meant to be called from the daemon, having both call `log_sys` is OK, because we can arrange for the log functions to print to the standard error instead of to the log file by setting a global variable.

```
 95   /*
 96    * "Timed" read - timout specifies the # of seconds to wait before
 97    * giving up (5th argument to select controls how long to wait for
 98    * data to be readable).  Returns # of bytes read or -1 on error.
 99    *
100    * LOCKING: none.
101    */
102   ssize_t
103   tread(int fd, void *buf, size_t nbytes, unsigned int timout)
104   {
105     int             nfds;
106     fd_set          readfds;
107     struct timeval  tv;

108     tv.tv_sec = timout;
109     tv.tv_usec = 0;
110     FD_ZERO(&readfds);
111     FD_SET(fd, &readfds);
112     nfds = select(fd+1, &readfds, NULL, NULL, &tv);
113     if (nfds <= 0) {
114         if (nfds == 0)
115             errno = ETIME;
116         return(-1);
117     }
118     return(read(fd, buf, nbytes));
119   }
```

[95–107] We provide a function called `tread` to read a specified number of bytes, but block for at most *timout* seconds before giving up. This function is useful when reading from a socket or a pipe. If we don't receive data before the specified time limit, we return –1 with `errno` set to `ETIME`. If data is available within the time limit, we return at most *nbytes* bytes of data, but we can return less than requested if all the data doesn't arrive in time.

We use `tread` to prevent denial-of-service attacks on the printer spooling daemon. A malicious user might repeatedly try to connect to the daemon without sending it data, just to prevent other users from being able to submit print jobs. By giving up after a reasonable amount of time, we prevent this from happening. The tricky part is selecting a suitable timeout value that is large enough to prevent premature failures when the system is under load and tasks are taking longer to complete. If we choose a value that is too large, however, we might enable denial-of-service attacks by allowing the daemon to consume too many resources to process the pending requests.

[108–119] We use `select` to wait for the specified file descriptor to be readable. If the time limit expires before data is available to be read, `select` returns 0, so we set `errno` to `ETIME` in this case. If `select` fails or times out, we return –1. Otherwise, we return whatever data is available.

```
120   /*
121    * "Timed" read - timout specifies the number of seconds to wait
122    * per read call before giving up, but read exactly nbytes bytes.
123    * Returns number of bytes read or -1 on error.
124    *
125    * LOCKING: none.
126    */
127   ssize_t
128   treadn(int fd, void *buf, size_t nbytes, unsigned int timout)
129   {
130     size_t  nleft;
131     ssize_t nread;

132     nleft = nbytes;
133     while (nleft > 0) {
134         if ((nread = tread(fd, buf, nleft, timout)) < 0) {
135             if (nleft == nbytes)
136                 return(-1); /* error, return -1 */
137             else
138                 break;      /* error, return amount read so far */
139         } else if (nread == 0) {
140             break;          /* EOF */
141         }
142         nleft -= nread;
143         buf += nread;
144     }
145     return(nbytes - nleft);      /* return >= 0 */
146   }
```

[120–146] We also provide a variation of `tread`, called `treadn`, that reads exactly the number of bytes requested. This is similar to the `readn` function described in Section 14.7, but with the addition of the timeout parameter.

To read exactly *nbytes* bytes, we have to be prepared to make multiple calls to `read`. The difficult part is trying to apply a single timeout value to multiple calls to `read`. We don't want to use an alarm, because signals can be messy to deal with in multithreaded applications. We can't rely on the system updating the `timeval` structure on return from `select` to indicate the amount of time left, because many platforms do not support this behavior (Section 14.4.1). Thus, we compromise and define the timeout value in this case to apply to an individual `read` call. Instead of limiting the total amount of time we wait, it limits the amount of time we'll wait in every iteration of the loop. The maximum time we can wait is bounded by (*nbytes* × *timout*) seconds (worst case, we'll receive only 1 byte at a time).

We use `nleft` to record the number of bytes remaining to be read. If `tread` fails and we have received data in a previous iteration, we break out of the `while` loop and return the number of bytes read; otherwise, we return –1.

The command used to submit a print job is shown next. The C source file is `print.c`.

```
 1  /*
 2   * The client command for printing documents.  Opens the file
 3   * and sends it to the printer spooling daemon.  Usage:
 4   *     print [-t] filename
 5   */
 6  #include "apue.h"
 7  #include "print.h"
 8  #include <fcntl.h>
 9  #include <pwd.h>

10  /*
11   * Needed for logging funtions.
12   */
13  int log_to_stderr = 1;

14  void submit_file(int, int, const char *, size_t, int);

15  int
16  main(int argc, char *argv[])
17  {
18      int             fd, sfd, err, text, c;
19      struct stat     sbuf;
20      char            *host;
21      struct addrinfo *ailist, *aip;

22      err = 0;
23      text = 0;
24      while ((c = getopt(argc, argv, "t")) != -1) {
25          switch (c) {
26          case 't':
27              text = 1;
28              break;

29          case '?':
30              err = 1;
31              break;
32          }
33      }
```

[1–14] We need to define an integer called `log_to_stderr` to be able to use the log functions in our library. If this integer is set to a nonzero value, error messages will be sent to the standard error stream instead of to a log file. Although we don't use any logging functions in `print.c`, we do link `util.o` with `print.o` to build the executable `print` command, and `util.c` contains functions for both user commands and daemons.

[15–33] We support one option, `-t`, to force the file to be printed as text (instead of as a PostScript program, for example). We use the `getopt` function (introduced in Section 17.6) to process the command options.

```
34      if (err || (optind != argc - 1))
35          err_quit("usage: print [-t] filename");
36      if ((fd = open(argv[optind], O_RDONLY)) < 0)
37          err_sys("print: can't open %s", argv[optind]);
38      if (fstat(fd, &sbuf) < 0)
39          err_sys("print: can't stat %s", argv[optind]);
40      if (!S_ISREG(sbuf.st_mode))
41          err_quit("print: %s must be a regular file", argv[optind]);

42      /*
43       * Get the hostname of the host acting as the print server.
44       */
45      if ((host = get_printserver()) == NULL)
46          err_quit("print: no print server defined");
47      if ((err = getaddrlist(host, "print", &ailist)) != 0)
48          err_quit("print: getaddrinfo error: %s", gai_strerror(err));

49      for (aip = ailist; aip != NULL; aip = aip->ai_next) {
50          if ((sfd = connect_retry(AF_INET, SOCK_STREAM, 0,
51            aip->ai_addr, aip->ai_addrlen)) < 0) {
52              err = errno;
```

[34–41] When `getopt` completes processing the command options, it leaves the variable `optind` set to the index of the first nonoptional argument. If this is any value other than the index of the last argument, then the wrong number of arguments was specified (we support only one nonoptional argument). Our error processing includes checks to ensure that we can open the file to be printed and that it is a regular file (as opposed to a directory or other type of file).

[42–48] We get the name of the host where the printer spooling daemon is running by calling the `get_printserver` function from `util.c`. Then we translate the host name into a network address by calling `getaddrlist` (also from `util.c`).

Note that we specify the service as "print." As part of installing the printer spooling daemon on a system, we need to make sure that `/etc/services` (or the equivalent database) has an entry for the printer service. When we select a port number for the daemon, it would be a good idea to select one that is privileged, to prevent malicious users from writing applications that pretend to be a printer spooling daemon but instead steal copies of the files we try to print. This means that the port number should be less than 1,024 (recall Section 16.3.4) and that our daemon will have to run with superuser privileges to allow it to bind to a reserved port.

[49–52] We try to connect to the daemon using one address at a time from the list returned by `getaddrinfo`. We will try to send the file to the daemon using the first address to which we can connect.

```
53          } else {
54              submit_file(fd, sfd, argv[optind], sbuf.st_size, text);
55              exit(0);
56          }
57      }
58      err_exit(err, "print: can't contact %s", host);
59  }

60  /*
61   * Send a file to the printer daemon.
62   */
63  void
64  submit_file(int fd, int sockfd, const char *fname, size_t nbytes,
65              int text)
66  {
67      int                 nr, nw, len;
68      struct passwd       *pwd;
69      struct printreq     req;
70      struct printresp    res;
71      char                buf[IOBUFSZ];

72      /*
73       * First build the header.
74       */
75      if ((pwd = getpwuid(geteuid())) == NULL) {
76          strcpy(req.usernm, "unknown");
77      } else {
78          strncpy(req.usernm, pwd->pw_name, USERNM_MAX-1);
79          req.usernm[USERNM_MAX-1] = '\0';
80      }
```

[53–59] If we are able to connect to the printer spooling daemon, we call `submit_file` to transmit the file we want to print to the daemon. Then we exit with a value of 0 to indicate success. If we can't connect to any of the addresses, we call `err_exit` to print an error message and exit with a value of 1 to indicate failure. (Appendix B contains the source code for `err_exit` and the other error routines.)

[60–80] The `submit_file` function sends a print request to the daemon and reads the response. First, we build the `printreq` request header. We use `geteuid` to get the caller's effective user ID and pass this to `getpwuid` to look for the user in the system's password file. We copy the user's name to the request header or use the string `unknown` if we can't identify the user. We use `strncpy` when copying the name from the password file to avoid writing past the end of the user name buffer in the request header. If the name is longer than the size of the buffer, `strncpy` won't store a terminating null byte in the buffer, so we need to do it ourselves.

```
 81      req.size = htonl(nbytes);

 82      if (text)
 83          req.flags = htonl(PR_TEXT);
 84      else
 85          req.flags = 0;

 86      if ((len = strlen(fname)) >= JOBNM_MAX) {
 87          /*
 88           * Truncate the filename (+-5 accounts for the leading
 89           * four characters and the terminating null).
 90           */
 91          strcpy(req.jobnm, "... ");
 92          strncat(req.jobnm, &fname[len-JOBNM_MAX+5], JOBNM_MAX-5);
 93      } else {
 94          strcpy(req.jobnm, fname);
 95      }

 96      /*
 97       * Send the header to the server.
 98       */
 99      nw = writen(sockfd, &req, sizeof(struct printreq));
100      if (nw != sizeof(struct printreq)) {
101          if (nw < 0)
102              err_sys("can't write to print server");
103          else
104              err_quit("short write (%d/%d) to print server",
105                nw, sizeof(struct printreq));
106      }
```

[81–95] We store the size of the file to be printed in the header after converting it to network byte order. Then we do the same with the `PR_TEXT` flag if the file is to be printed as plaintext. By translating these integers to network byte order, we can run the print command on a client system while the printer spooling daemon is running on another computer system. If these systems use processors with different byte ordering, then the commands will still work. (We discussed byte ordering in Section 16.3.1.)

We set the job name to the name of the file being printed. If the name is longer than will fit in the job name field in the message, we copy only the last portion of the name that will fit. This effectively truncates the beginning portion of the name. In this case, we prepend an ellipsis to indicate that there were more characters than would fit in the field.

[96–106] We send the request header to the daemon using `writen`. (Recall that we introduced the `writen` function in Figure 14.24.) The `writen` function uses multiple calls to `write`, if necessary, to transmit the specified amount. If the `writen` function returns an error or transmits less than we requested, we print an error message and exit.

```
107     /*
108      * Now send the file.
109      */
110     while ((nr = read(fd, buf, IOBUFSZ)) != 0) {
111         nw = writen(sockfd, buf, nr);
112         if (nw != nr) {
113             if (nw < 0)
114                 err_sys("can't write to print server");
115             else
116                 err_quit("short write (%d/%d) to print server",
117                   nw, nr);
118         }
119     }

120     /*
121      * Read the response.
122      */
123     if ((nr = readn(sockfd, &res, sizeof(struct printresp))) !=
124       sizeof(struct printresp))
125         err_sys("can't read response from server");
126     if (res.retcode != 0) {
127         printf("rejected: %s\n", res.msg);
128         exit(1);
129     } else {
130         printf("job ID %ld\n", (long)ntohl(res.jobid));
131     }
132 }
```

[107–119] After sending the header to the daemon, we send the file to be printed. We read the file `IOBUFSZ` bytes at a time and use `writen` to send the data to the daemon. As with the header, if the write fails or we write less than we expect, we print an error message and exit.

[120–132] Once we have sent the file to be printed to the print spooling daemon, we read the daemon's response. If the print request failed, the return code (`retcode`) will be nonzero, so we print the textual error message included in the response. If the request succeeded, we print the job ID so that the user knows how to refer to the request in the future. (Writing a command to cancel a pending print request is left as an exercise; the job ID can be used in the cancellation request to identify the job to be removed from the print queue. See Exercise 21.5.) When `submit_file` returns to the `main` function, we exit, indicating success.

Note that a successful response from the daemon does not mean that the printer was able to print the file; it merely means that the daemon successfully added the print job to the queue.

This completes our look at the print command. The last file we will look at is the C source file for the printer spooling daemon.

```
 1  /*
 2   * Print server daemon.
 3   */
 4  #include "apue.h"
 5  #include <fcntl.h>
 6  #include <dirent.h>
 7  #include <ctype.h>
 8  #include <pwd.h>
 9  #include <pthread.h>
10  #include <strings.h>
11  #include <sys/select.h>
12  #include <sys/uio.h>

13  #include "print.h"
14  #include "ipp.h"

15  /*
16   * These are for the HTTP response from the printer.
17   */
18  #define HTTP_INFO(x)   ((x) >= 100 && (x) <= 199)
19  #define HTTP_SUCCESS(x) ((x) >= 200 && (x) <= 299)

20  /*
21   * Describes a print job.
22   */
23  struct job {
24     struct job      *next;      /* next in list */
25     struct job      *prev;      /* previous in list */
26     int32_t          jobid;     /* job ID */
27     struct printreq  req;       /* copy of print request */
28  };

29  /*
30   * Describes a thread processing a client request.
31   */
32  struct worker_thread {
33     struct worker_thread  *next;    /* next in list */
34     struct worker_thread  *prev;    /* previous in list */
35     pthread_t              tid;     /* thread ID */
36     int                    sockfd;  /* socket */
37  };
```

[1–19] The printer spooling daemon includes the IPP header file that we saw earlier, because the daemon needs to communicate with the printer using this protocol. The `HTTP_INFO` and `HTTP_SUCCESS` macros define the status of the HTTP request (recall that IPP is built on top of HTTP). Section 10 in RFC 2616 defines the HTTP status codes.

[20–37] The `job` and `worker_thread` structures are used by the spooling daemon to keep track of print jobs and threads accepting print requests, respectively.

```
38  /*
39   * Needed for logging.
40   */
41  int                     log_to_stderr = 0;

42  /*
43   * Printer-related stuff.
44   */
45  struct addrinfo         *printer;
46  char                    *printer_name;
47  pthread_mutex_t         configlock = PTHREAD_MUTEX_INITIALIZER;
48  int                     reread;

49  /*
50   * Thread-related stuff.
51   */
52  struct worker_thread    *workers;
53  pthread_mutex_t         workerlock = PTHREAD_MUTEX_INITIALIZER;
54  sigset_t                mask;

55  /*
56   * Job-related stuff.
57   */
58  struct job              *jobhead, *jobtail;
59  int                     jobfd;
```

[38–41] Our logging functions require that we define the `log_to_stderr` variable and set it to 0 to force log messages to be sent to the system log instead of to the standard error. In `print.c`, we defined `log_to_stderr` and set it to 1, even though we don't use the log functions in the user command. We could have avoided this by splitting the utility functions into two separate files: one for the server and one for the client commands.

[42–48] We use the global variable `printer` to hold the network address of the printer. We store the host name of the printer in `printer_name`. The `configlock` mutex protects access to the `reread` variable, which is used to indicate that the daemon needs to reread the configuration file, presumably because an administrator changed the printer or its network address.

[49–54] Next, we define the thread-related variables. We use `workers` as the head of a doubly linked list of threads that are receiving files from clients. This list is protected by the mutex `workerlock`. The signal mask used by the threads is held in the variable `mask`.

[55–59] For the list of pending jobs, we define `jobhead` to be the start of the list and `jobtail` to be the tail of the list. This list is also doubly linked, but we need to add jobs to the end of the list, so we must remember a pointer to the list tail. With the list of worker threads, the order doesn't matter, so we can add them to the head of the list and don't need to remember the tail pointer. `jobfd` is the file descriptor for the job file.

```
60  int32_t                 nextjob;
61  pthread_mutex_t         joblock = PTHREAD_MUTEX_INITIALIZER;
62  pthread_cond_t          jobwait = PTHREAD_COND_INITIALIZER;
63  /*
64   * Function prototypes.
65   */
66  void        init_request(void);
67  void        init_printer(void);
68  void        update_jobno(void);
69  int32_t     get_newjobno(void);
70  void        add_job(struct printreq *, int32_t);
71  void        replace_job(struct job *);
72  void        remove_job(struct job *);
73  void        build_qonstart(void);
74  void        *client_thread(void *);
75  void        *printer_thread(void *);
76  void        *signal_thread(void *);
77  ssize_t     readmore(int, char **, int, int *);
78  int         printer_status(int, struct job *);
79  void        add_worker(pthread_t, int);
80  void        kill_workers(void);
81  void        client_cleanup(void *);
82  /*
83   * Main print server thread.  Accepts connect requests from
84   * clients and spawns additional threads to service requests.
85   *
86   * LOCKING: none.
87   */
88  int
89  main(int argc, char *argv[])
90  {
91      pthread_t           tid;
92      struct addrinfo     *ailist, *aip;
93      int                 sockfd, err, i, n, maxfd;
94      char                *host;
95      fd_set              rendezvous, rset;
96      struct sigaction    sa;
97      struct passwd       *pwdp;
```

[60–62] `nextjob` is the ID of the next print job to be received. The `joblock` mutex protects the linked list of jobs, as well as the condition represented by the `jobwait` condition variable.

[63–81] We declare the function prototypes for the remaining functions in this file. Doing this up front allows us to place the functions in the file without worrying about the order in which each is called.

[82–97] The `main` function for the printer spooling daemon has two tasks to perform: initialize the daemon and then process connect requests from clients.

```
 98      if (argc != 1)
 99          err_quit("usage: printd");
100      daemonize("printd");

101      sigemptyset(&sa.sa_mask);
102      sa.sa_flags = 0;
103      sa.sa_handler = SIG_IGN;
104      if (sigaction(SIGPIPE, &sa, NULL) < 0)
105          log_sys("sigaction failed");
106      sigemptyset(&mask);
107      sigaddset(&mask, SIGHUP);
108      sigaddset(&mask, SIGTERM);
109      if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
110          log_sys("pthread_sigmask failed");

111      n = sysconf(_SC_HOST_NAME_MAX);
112      if (n < 0)  /* best guess */
113          n = HOST_NAME_MAX;
114      if ((host = malloc(n)) == NULL)
115          log_sys("malloc error");
116      if (gethostname(host, n) < 0)
117          log_sys("gethostname error");

118      if ((err = getaddrlist(host, "print", &ailist)) != 0) {
119          log_quit("getaddrinfo error: %s", gai_strerror(err));
120          exit(1);
121      }
```

[98–100] The daemon doesn't have any options (the only argument is the command name itself), so if `argc` is not 1, we call `err_quit` to print an error message and exit. We call the `daemonize` function from Figure 13.1 to become a daemon. After this point, we can't print error messages to standard error; we need to log them instead.

[101–110] We arrange to ignore `SIGPIPE`. We will be writing to socket file descriptors, and we don't want a write error to trigger `SIGPIPE`, because the default action is to kill the process. Next, we set the signal mask of the thread to include `SIGHUP` and `SIGTERM`. All threads we create will inherit this signal mask. We'll send the `SIGHUP` signal to the daemon to tell it to reread its configuration file. We'll send the `SIGTERM` signal to the daemon to tell it to clean up and exit gracefully.

[111–117] We call `sysconf` to get the maximum size of a host name. If `sysconf` fails or the limit is undefined, we use `HOST_NAME_MAX` as a best guess. Sometimes, this constant is defined for us by the platform, but if it isn't, we chose our own value in `print.h`. We allocate memory to hold the host name and call `gethostname` to retrieve it.

[118–121] Next, we try to find the network address that the daemon is supposed to use to provide the printer spooling service.

```
122     FD_ZERO(&rendezvous);
123     maxfd = -1;
124     for (aip = ailist; aip != NULL; aip = aip->ai_next) {
125         if ((sockfd = initserver(SOCK_STREAM, aip->ai_addr,
126           aip->ai_addrlen, QLEN)) >= 0) {
127             FD_SET(sockfd, &rendezvous);
128             if (sockfd > maxfd)
129                 maxfd = sockfd;
130         }
131     }
132     if (maxfd == -1)
133         log_quit("service not enabled");

134     pwdp = getpwnam(LPNAME);
135     if (pwdp == NULL)
136         log_sys("can't find user %s", LPNAME);
137     if (pwdp->pw_uid == 0)
138         log_quit("user %s is privileged", LPNAME);
139     if (setgid(pwdp->pw_gid) < 0 || setuid(pwdp->pw_uid) < 0)
140         log_sys("can't change IDs to user %s", LPNAME);

141     init_request();
142     init_printer();
```

[122–131] We clear the `rendezvous` `fd_set` variable that we will use with `select` to wait for client connect requests. We initialize the maximum file descriptor to –1 so that the first file descriptor we allocate is sure to be greater than `maxfd`. For each network address on which we need to provide service, we call `initserver` (from Figure 16.22) to allocate and initialize a socket. If `initserver` succeeds, we add the file descriptor to the `fd_set`; if it is greater than the maximum, we set `maxfd` equal to the socket file descriptor.

[132–133] If `maxfd` is still –1 after stepping through the list of `addrinfo` structures, we can't enable the printer spooling service, so we log a message and exit.

[134–140] Our daemon needs superuser privileges to bind a socket to a reserved port number. Now that this is done, we can lower its privileges by changing its user and group IDs to the ones associated with the `LPNAME` account. We follow the principles of least privilege to avoid exposing the system to any potential vulnerabilities in the daemon. We call `getpwnam` to find the password entry for the daemon. If no such user account exists, or if it exists with the same user ID as the superuser, we log an error message and exit. Otherwise, we change both the real and effective IDs by calling `setgid` and `setuid`. To avoid exposing our system, we choose to provide no service at all if we can't reduce our privileges.

[141–142] We call `init_request` to initialize the job requests and ensure that only one copy of the daemon is running, and we call `init_printer` to initialize the printer information (we'll see both of these functions shortly).

```
143     err = pthread_create(&tid, NULL, printer_thread, NULL);
144     if (err == 0)
145         err = pthread_create(&tid, NULL, signal_thread, NULL);
146     if (err != 0)
147         log_exit(err, "can't create thread");
148     build_qonstart();
149     log_msg("daemon initialized");
150     for (;;) {
151         rset = rendezvous;
152         if (select(maxfd+1, &rset, NULL, NULL, NULL) < 0)
153             log_sys("select failed");
154         for (i = 0; i <= maxfd; i++) {
155             if (FD_ISSET(i, &rset)) {
156                 /*
157                  * Accept the connection and handle the request.
158                  */
159                 if ((sockfd = accept(i, NULL, NULL)) < 0)
160                     log_ret("accept failed");
161                 pthread_create(&tid, NULL, client_thread,
162                   (void *)((long)sockfd));
163             }
164         }
165     }
166     exit(1);
167 }
```

[143–149] We create one thread to handle signals and one thread to communicate with the printer. (By restricting printer communication to one thread, we can simplify the locking of the printer-related data structures.) Then we call `build_qonstart` to search the directories in `/var/spool/printer` for any pending jobs. For each job that we find on disk, we will create a structure to let the printer thread know that it should send the file to the printer. At this point, we are done setting up the daemon, so we log a message to indicate that the daemon has initialized successfully.

[150–167] We copy the `rendezvous fd_set` structure to `rset` and call `select` to wait for one of the file descriptors to become readable. We have to copy `rendezvous`, because `select` will modify the `fd_set` structure that we pass to it to include only those file descriptors that satisfy the event. Since the sockets have been initialized for use by a server, a readable file descriptor means that a connect request is pending. After `select` returns, we check `rset` for a readable file descriptor. If we find one, we call `accept` to accept the connection. If this fails, we log an error message and continue checking for more readable file descriptors. Otherwise, we create a thread to handle the client connection. The `main` thread loops, farming requests out to other threads for processing, and should never reach the `exit` statement.

```
168   /*
169    * Initialize the job ID file.  Use a record lock to prevent
170    * more than one printer daemon from running at a time.
171    *
172    * LOCKING: none, except for record-lock on job ID file.
173    */
174   void
175   init_request(void)
176   {
177     int     n;
178     char    name[FILENMSZ];

179     sprintf(name, "%s/%s", SPOOLDIR, JOBFILE);
180     jobfd = open(name, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR);
181     if (write_lock(jobfd, 0, SEEK_SET, 0) < 0)
182         log_quit("daemon already running");

183     /*
184      * Reuse the name buffer for the job counter.
185      */
186     if ((n = read(jobfd, name, FILENMSZ)) < 0)
187         log_sys("can't read job file");
188     if (n == 0)
189         nextjob = 1;
190     else
191         nextjob = atol(name);
192   }
```

[168–182] The `init_request` function does two things: it places a record lock on the job file, `/var/spool/printer/jobno`, and it reads the file to determine the next job number to assign. We place a write lock on the entire file to indicate that the daemon is running. If someone tries to start additional copies of the printer spooling daemon while one is already running, these additional daemons will fail to obtain the write lock and will exit. Thus, only one copy of the daemon can be running at a time. (Recall that we used this technique in Figure 13.6; we discussed the `write_lock` macro in Section 14.3.)

[183–192] The job file contains an ASCII integer string representing the next job number. If the file was just created and therefore is empty, we set `nextjob` to 1. Otherwise, we use `atol` to convert the string to an integer and use this value as the next job number. We leave `jobfd` open to the job file so that we can update the job number as jobs are created. We can't close the file, because this would release the write lock that we've placed on it.

On a system where a long integer is 64 bits wide, we need a buffer at least 21 bytes in size to fit a string representing the largest possible long integer. We can safely reuse the filename buffer, because `FILENMSZ` is defined to be 64 in `print.h`.

```
193   /*
194    * Initialize printer information from configuration file.
195    *
196    * LOCKING: none.
197    */
198   void
199   init_printer(void)
200   {
201     printer = get_printaddr();
202     if (printer == NULL)
203         exit(1);    /* message already logged */
204     printer_name = printer->ai_canonname;
205     if (printer_name == NULL)
206         printer_name = "printer";
207     log_msg("printer is %s", printer_name);
208   }
209   /*
210    * Update the job ID file with the next job number.
211    * Doesn't handle wrap-around of job number.
212    *
213    * LOCKING: none.
214    */
215   void
216   update_jobno(void)
217   {
218     char    buf[32];
219     if (lseek(jobfd, 0, SEEK_SET) == -1)
220         log_sys("can't seek in job file");
221     sprintf(buf, "%d", nextjob);
222     if (write(jobfd, buf, strlen(buf)) < 0)
223         log_sys("can't update job file");
224   }
```

[193–208] The `init_printer` function is used to set the printer name and address. We get the printer address by calling `get_printaddr` (from `util.c`). If this fails, we exit. The `get_printaddr` function logs its own message when it is unable to find the printer's address. If a printer address is found, however, we set the printer name to the `ai_canonname` field in the `addrinfo` structure. If this field is null, we set the printer name to a default value of `printer`. Note that we log the name of the printer we are using to aid administrators in diagnosing problems with the spooling system.

[209–224] The `update_jobno` function is used to write the next job number to the job file, `/var/spool/printer/jobno`. We seek to the beginning of the file, convert the integer job number into a string, and write it to the file. On error, we log a message and exit. The job number increases monotonically; handling wrap-around is left as an exercise (see Exercise 21.9).

```
225   /*
226    * Get the next job number.
227    *
228    * LOCKING: acquires and releases joblock.
229    */
230   int32_t
231   get_newjobno(void)
232   {
233     int32_t jobid;

234     pthread_mutex_lock(&joblock);
235     jobid = nextjob++;
236     if (nextjob <= 0)
237         nextjob = 1;
238     pthread_mutex_unlock(&joblock);
239     return(jobid);
240   }

241   /*
242    * Add a new job to the list of pending jobs.  Then signal
243    * the printer thread that a job is pending.
244    *
245    * LOCKING: acquires and releases joblock.
246    */
247   void
248   add_job(struct printreq *reqp, int32_t jobid)
249   {
250     struct job  *jp;

251     if ((jp = malloc(sizeof(struct job))) == NULL)
252         log_sys("malloc failed");
253     memcpy(&jp->req, reqp, sizeof(struct printreq));
```

[225–240] The `get_newjobno` function is used to get the next job number. We first lock the `joblock` mutex. We increment the `nextjob` variable and handle the case where it wraps around. Then we unlock the mutex and return the value `nextjob` had before we incremented it. Multiple threads can call `get_newjobno` at the same time; we need to serialize access to the next job number so that each thread gets a unique job number. (Refer to Figure 11.9 to see what could happen if we don't serialize the threads in this case.)

[241–253] The `add_job` function is used to add a new print request to the end of the list of pending print jobs. We start by allocating space for the `job` structure. If this fails, we log an error message and exit. At this point, the print request is stored safely on disk; when the printer spooling daemon is restarted, it will pick the request up. After we allocate memory for the new job, we copy the request structure from the client into the job structure. Recall from `print.h` that a `job` structure consists of a pair of list pointers, a job ID, and a copy of the `printreq` structure sent to us by the client `print` command.

```
254     jp->jobid = jobid;
255     jp->next = NULL;
256     pthread_mutex_lock(&joblock);
257     jp->prev = jobtail;
258     if (jobtail == NULL)
259         jobhead = jp;
260     else
261         jobtail->next = jp;
262     jobtail = jp;
263     pthread_mutex_unlock(&joblock);
264     pthread_cond_signal(&jobwait);
265   }

266   /*
267    * Replace a job back on the head of the list.
268    *
269    * LOCKING: acquires and releases joblock.
270    */
271   void
272   replace_job(struct job *jp)
273   {
274     pthread_mutex_lock(&joblock);
275     jp->prev = NULL;
276     jp->next = jobhead;
277     if (jobhead == NULL)
278         jobtail = jp;
279     else
280         jobhead->prev = jp;
281     jobhead = jp;
282     pthread_mutex_unlock(&joblock);
283   }
```

[254–265] We save the job ID and lock the `joblock` mutex to gain exclusive access to the linked list of print jobs. We are about to add the new job structure to the end of the list. We set the new structure's previous pointer to the last job on the list. If the list is empty, we set `jobhead` to point to the new structure. Otherwise, we set the next pointer in the last entry on the list to point to the new structure. Then we set `jobtail` to point to the new structure. We unlock the mutex and signal the printer thread that another job is available.

[266–283] The `replace_job` function is used to insert a job at the head of the pending job list. We acquire the `joblock` mutex, set the previous pointer in the `job` structure to `NULL`, and set the next pointer in the `job` structure to point to the head of the list. If the list is empty, we set `jobtail` to point to the `job` structure we are replacing. Otherwise, we set the previous pointer in the first `job` structure on the list to point to the `job` structure we are replacing. Then we set the `jobhead` pointer to the `job` structure we are replacing. Finally, we release the `joblock` mutex.

```
284   /*
285    * Remove a job from the list of pending jobs.
286    *
287    * LOCKING: caller must hold joblock.
288    */
289   void
290   remove_job(struct job *target)
291   {
292     if (target->next != NULL)
293         target->next->prev = target->prev;
294     else
295         jobtail = target->prev;
296     if (target->prev != NULL)
297         target->prev->next = target->next;
298     else
299         jobhead = target->next;
300   }

301   /*
302    * Check the spool directory for pending jobs on start-up.
303    *
304    * LOCKING: none.
305    */
306   void
307   build_qonstart(void)
308   {
309     int             fd, err, nr;
310     int32_t         jobid;
311     DIR             *dirp;
312     struct dirent   *entp;
313     struct printreq req;
314     char            dname[FILENMSZ], fname[FILENMSZ];

315     sprintf(dname, "%s/%s", SPOOLDIR, REQDIR);
316     if ((dirp = opendir(dname)) == NULL)
317         return;
```

[284–300] `remove_job` removes a job from the list of pending jobs given a pointer to the job to be removed. The caller must already hold the `joblock` mutex. If the next pointer is non-null, we set the next entry's previous pointer to the target's previous pointer. Otherwise, the entry is the last one on the list, so we set `jobtail` to the target's previous pointer. If the target's previous pointer is non-null, we set the previous entry's next pointer equal to the target's next pointer. Otherwise, this is the first entry in the list, so we set `jobhead` to point to the next entry in the list after the target.

[301–317] When the daemon starts, it calls `build_qonstart` to build an in-memory list of print jobs from the disk files stored in `/var/spool/printer/reqs`. If we can't open the directory, no print jobs are pending, so we return.

```
318     while ((entp = readdir(dirp)) != NULL) {
319         /*
320          * Skip "." and ".."
321          */
322         if (strcmp(entp->d_name, ".") == 0 ||
323           strcmp(entp->d_name, "..") == 0)
324             continue;
325         /*
326          * Read the request structure.
327          */
328         sprintf(fname, "%s/%s/%s", SPOOLDIR, REQDIR, entp->d_name);
329         if ((fd = open(fname, O_RDONLY)) < 0)
330             continue;
331         nr = read(fd, &req, sizeof(struct printreq));
332         if (nr != sizeof(struct printreq)) {
333             if (nr < 0)
334                 err = errno;
335             else
336                 err = EIO;
337             close(fd);
338             log_msg("build_qonstart: can't read %s: %s",
339               fname, strerror(err));
340             unlink(fname);
341             sprintf(fname, "%s/%s/%s", SPOOLDIR, DATADIR,
342               entp->d_name);
343             unlink(fname);
344             continue;
345         }
346         jobid = atol(entp->d_name);
347         log_msg("adding job %d to queue", jobid);
348         add_job(&req, jobid);
349     }
350     closedir(dirp);
351 }
```

[318–324] We read each entry in the directory, one at a time. We skip the entries for dot and dot-dot.

[325–345] For each entry, we create the full pathname of the file and open it for reading. If the `open` call fails, we just skip the file. Otherwise, we read the `printreq` structure stored in it. If we don't read the entire structure, we close the file, log an error message, and unlink the file. Then we create the full pathname of the corresponding data file and unlink it, too.

[346–351] If we were able to read a complete `printreq` structure, we convert the filename into a job ID (the name of the file is its job ID), log a message, and then add the request to the list of pending print jobs. When we are done reading the directory, `readdir` will return `NULL`, and we close the directory and return.

```
352   /*
353    * Accept a print job from a client.
354    *
355    * LOCKING: none.
356    */
357   void *
358   client_thread(void *arg)
359   {
360     int                 n, fd, sockfd, nr, nw, first;
361     int32_t             jobid;
362     pthread_t           tid;
363     struct printreq     req;
364     struct printresp    res;
365     char                name[FILENMSZ];
366     char                buf[IOBUFSZ];

367     tid = pthread_self();
368     pthread_cleanup_push(client_cleanup, (void *)((long)tid));
369     sockfd = (long)arg;
370     add_worker(tid, sockfd);

371     /*
372      * Read the request header.
373      */
374     if ((n = treadn(sockfd, &req, sizeof(struct printreq), 10)) !=
375       sizeof(struct printreq)) {
376         res.jobid = 0;
377         if (n < 0)
378             res.retcode = htonl(errno);
379         else
380             res.retcode = htonl(EIO);
381         strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
382         writen(sockfd, &res, sizeof(struct printresp));
383         pthread_exit((void *)1);
384     }
```

[352–370] The `client_thread` is spawned from the `main` thread when a connect request is accepted. Its job is to receive the file to be printed from the client `print` command. We create a separate thread for each client print request.

The first thing we do is install a thread cleanup handler (see Section 11.5 for a discussion of thread cleanup handlers). The cleanup handler is `client_cleanup`, which we will see later. It takes a single argument: our thread ID. Then we call `add_worker` to create a `worker_thread` structure and add it to the list of active client threads.

[371–384] At this point, we are done with the thread's initialization tasks, so we read the request header from the client. If the client sends less data than we expect or we encounter an error, we respond with a message indicating the reason for the error and call `pthread_exit` to terminate the thread.

```
385     req.size = ntohl(req.size);
386     req.flags = ntohl(req.flags);
387     /*
388      * Create the data file.
389      */
390     jobid = get_newjobno();
391     sprintf(name, "%s/%s/%d", SPOOLDIR, DATADIR, jobid);
392     fd = creat(name, FILEPERM);
393     if (fd < 0) {
394         res.jobid = 0;
395         res.retcode = htonl(errno);
396         log_msg("client_thread: can't create %s: %s", name,
397           strerror(res.retcode));
398         strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
399         writen(sockfd, &res, sizeof(struct printresp));
400         pthread_exit((void *)1);
401     }
402     /*
403      * Read the file and store it in the spool directory.
404      * Try to figure out if the file is a PostScript file
405      * or a plain text file.
406      */
407     first = 1;
408     while ((nr = tread(sockfd, buf, IOBUFSZ, 20)) > 0) {
409         if (first) {
410             first = 0;
411             if (strncmp(buf, "%!PS", 4) != 0)
412                 req.flags |= PR_TEXT;
413         }
```

[385–401] We convert the integer fields in the request header to host byte order and call `get_newjobno` to reserve the next job ID for this print request. We create the job data file, named `/var/spool/printer/data/`*jobid*, where *jobid* is the request's job ID. We use permissions that prevent others from being able read the files (`FILEPERM` is defined as `S_IRUSR|S_IWUSR` in `print.h`). If we can't create the file, we log an error message, send a failure response back to the client, and terminate the thread by calling `pthread_exit`.

[402–413] We read the file contents from the client, with the intention of writing the contents out to our private copy of the data file. But before we write anything, we need to check if this is a PostScript file the first time through the loop. If the file doesn't begin with the pattern `%!PS`, we can assume that the file is plaintext, so we set the `PR_TEXT` flag in the request header in this case. (Recall that the client can also set this flag if the `-t` flag is included when the `print` command is executed.) Although PostScript programs are not required to start with the pattern `%!PS`, the document formatting guidelines (Adobe Systems [1999]) strongly recommends that they do.

```
414         nw = write(fd, buf, nr);
415         if (nw != nr) {
416             res.jobid = 0;
417             if (nw < 0)
418                 res.retcode = htonl(errno);
419             else
420                 res.retcode = htonl(EIO);
421             log_msg("client_thread: can't write %s: %s", name,
422               strerror(res.retcode));
423             close(fd);
424             strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
425             writen(sockfd, &res, sizeof(struct printresp));
426             unlink(name);
427             pthread_exit((void *)1);
428         }
429     }
430     close(fd);
431     /*
432      * Create the control file.  Then write the
433      * print request information to the control
434      * file.
435      */
436     sprintf(name, "%s/%s/%d", SPOOLDIR, REQDIR, jobid);
437     fd = creat(name, FILEPERM);
438     if (fd < 0) {
439         res.jobid = 0;
440         res.retcode = htonl(errno);
441         log_msg("client_thread: can't create %s: %s", name,
442           strerror(res.retcode));
443         strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
444         writen(sockfd, &res, sizeof(struct printresp));
445         sprintf(name, "%s/%s/%d", SPOOLDIR, DATADIR, jobid);
446         unlink(name);
447         pthread_exit((void *)1);
448     }
```

[414–430] We write the data that we read from the client to the data file. If `write` fails, we log an error message, close the file descriptor for the data file, send an error message back to the client, delete the data file, and terminate the thread by calling `pthread_exit`. Note that we do not explicitly close the socket file descriptor. This is done for us by our thread cleanup handler as part of the processing that occurs when we call `pthread_exit`.

When we receive all the data to be printed, we close the file descriptor for the data file.

[431–448] Next, we create a file, `/var/spool/printer/reqs/`*jobid*, to remember the print request. If this fails, we log an error message, send an error response to the client, remove the data file, and terminate the thread.

```
449     nw = write(fd, &req, sizeof(struct printreq));
450     if (nw != sizeof(struct printreq)) {
451         res.jobid = 0;
452         if (nw < 0)
453             res.retcode = htonl(errno);
454         else
455             res.retcode = htonl(EIO);
456         log_msg("client_thread: can't write %s: %s", name,
457           strerror(res.retcode));
458         close(fd);
459         strncpy(res.msg, strerror(res.retcode), MSGLEN_MAX);
460         writen(sockfd, &res, sizeof(struct printresp));
461         unlink(name);
462         sprintf(name, "%s/%s/%d", SPOOLDIR, DATADIR, jobid);
463         unlink(name);
464         pthread_exit((void *)1);
465     }
466     close(fd);
467     /*
468      * Send response to client.
469      */
470     res.retcode = 0;
471     res.jobid = htonl(jobid);
472     sprintf(res.msg, "request ID %d", jobid);
473     writen(sockfd, &res, sizeof(struct printresp));
474     /*
475      * Notify the printer thread, clean up, and exit.
476      */
477     log_msg("adding job %d to queue", jobid);
478     add_job(&req, jobid);
479     pthread_cleanup_pop(1);
480     return((void *)0);
481 }
```

[449–465] We write the `printreq` structure to the control file. On error, we log a message, close the descriptor for the control file, send a failure response back to the client, remove the data and control files, and terminate the thread.

[466–473] We close the file descriptor for the control file and send a message containing the job ID and a successful status (`retcode` set to 0) back to the client.

[474–481] We call `add_job` to add the received job to the list of pending print jobs and call `pthread_cleanup_pop` to complete the cleanup processing. The thread terminates when we return.

Note that before the thread exits, we must close any file descriptors we no longer need. Unlike with process termination, file descriptors are not closed automatically when a thread ends if other threads exist in the process. If we didn't close unneeded file descriptors, we'd eventually run out of resources.

```
482   /*
483    * Add a worker to the list of worker threads.
484    *
485    * LOCKING: acquires and releases workerlock.
486    */
487   void
488   add_worker(pthread_t tid, int sockfd)
489   {
490     struct worker_thread    *wtp;

491     if ((wtp = malloc(sizeof(struct worker_thread))) == NULL) {
492         log_ret("add_worker: can't malloc");
493         pthread_exit((void *)1);
494     }
495     wtp->tid = tid;
496     wtp->sockfd = sockfd;
497     pthread_mutex_lock(&workerlock);
498     wtp->prev = NULL;
499     wtp->next = workers;
500     if (workers == NULL)
501         workers = wtp;
502     else
503         workers->prev = wtp;
504     pthread_mutex_unlock(&workerlock);
505   }

506   /*
507    * Cancel (kill) all outstanding workers.
508    *
509    * LOCKING: acquires and releases workerlock.
510    */
511   void
512   kill_workers(void)
513   {
514     struct worker_thread    *wtp;

515     pthread_mutex_lock(&workerlock);
516     for (wtp = workers; wtp != NULL; wtp = wtp->next)
517         pthread_cancel(wtp->tid);
518     pthread_mutex_unlock(&workerlock);
519   }
```

[482–505] `add_worker` adds a `worker_thread` structure to the list of active threads. We allocate memory for the structure, initialize it, lock the `workerlock` mutex, add the structure to the head of the list, and unlock the mutex.

[506–519] The `kill_workers` function walks the list of worker threads and cancels each one. We hold the `workerlock` mutex while we walk the list. Recall that `pthread_cancel` merely schedules a thread for cancellation; actual cancellation happens when each thread reaches the next cancellation point.

```
520   /*
521    * Cancellation routine for the worker thread.
522    *
523    * LOCKING: acquires and releases workerlock.
524    */
525   void
526   client_cleanup(void *arg)
527   {
528     struct worker_thread    *wtp;
529     pthread_t               tid;

530     tid = (pthread_t)((long)arg);
531     pthread_mutex_lock(&workerlock);
532     for (wtp = workers; wtp != NULL; wtp = wtp->next) {
533         if (wtp->tid == tid) {
534             if (wtp->next != NULL)
535                 wtp->next->prev = wtp->prev;
536             if (wtp->prev != NULL)
537                 wtp->prev->next = wtp->next;
538             else
539                 workers = wtp->next;
540             break;
541         }
542     }
543     pthread_mutex_unlock(&workerlock);
544     if (wtp != NULL) {
545         close(wtp->sockfd);
546         free(wtp);
547     }
548   }
```

[520–542] The `client_cleanup` function is the thread cleanup handler for the worker threads that communicate with client commands. This function is called when the thread calls `pthread_exit`, calls `pthread_cleanup_pop` with a nonzero argument, or responds to a cancellation request. The argument is the thread ID of the thread terminating.

We lock the `workerlock` mutex and search the list of worker threads until we find a matching thread ID. When we find a match, we remove the worker thread structure from the list and stop the search.

[543–548] We unlock the `workerlock` mutex, close the socket file descriptor used by the thread to communicate with the client, and free the memory backing the `worker_thread` structure.

Since we try to acquire the `workerlock` mutex, if a thread reaches a cancellation point while the `kill_workers` function is still walking the list, we will have to wait until `kill_workers` releases the mutex before we can proceed.

```
549   /*
550    * Deal with signals.
551    *
552    * LOCKING: acquires and releases configlock.
553    */
554   void *
555   signal_thread(void *arg)
556   {
557     int     err, signo;

558     for (;;) {
559         err = sigwait(&mask, &signo);
560         if (err != 0)
561             log_quit("sigwait failed: %s", strerror(err));
562         switch (signo) {
563         case SIGHUP:
564             /*
565              * Schedule to re-read the configuration file.
566              */
567             pthread_mutex_lock(&configlock);
568             reread = 1;
569             pthread_mutex_unlock(&configlock);
570             break;

571         case SIGTERM:
572             kill_workers();
573             log_msg("terminate with signal %s", strsignal(signo));
574             exit(0);

575         default:
576             kill_workers();
577             log_quit("unexpected signal %d", signo);
578         }
579     }
580   }
```

[549–562] The `signal_thread` function is run by the thread that is responsible for handling signals. In the `main` function, we initialized the signal mask to include `SIGHUP` and `SIGTERM`. Here, we call `sigwait` to wait for one of these signals to occur. If `sigwait` fails, we log an error message and exit.

[563–570] If we receive `SIGHUP`, we acquire the `configlock` mutex, set the `reread` variable to 1, and release the mutex. This tells the printer daemon to reread the configuration file on the next iteration in its processing loop.

[571–574] If we receive `SIGTERM`, we call `kill_workers` to kill all the worker threads, log a message, and call `exit` to terminate the process.

[575–580] If we receive a signal we are not expecting, we kill the worker threads and call `log_quit` to log an error message and exit.

```
581   /*
582    * Add an option to the IPP header.
583    *
584    * LOCKING: none.
585    */
586   char *
587   add_option(char *cp, int tag, char *optname, char *optval)
588   {
589     int     n;
590     union {
591         int16_t s;
592         char c[2];
593     }       u;

594     *cp++ = tag;
595     n = strlen(optname);
596     u.s = htons(n);
597     *cp++ = u.c[0];
598     *cp++ = u.c[1];
599     strcpy(cp, optname);
600     cp += n;
601     n = strlen(optval);
602     u.s = htons(n);
603     *cp++ = u.c[0];
604     *cp++ = u.c[1];
605     strcpy(cp, optval);
606     return(cp + n);
607   }
```

[581–593] The `add_option` function is used to add an option to the IPP header that we build to send to the printer. Recall from Figure 21.4 that the format of an attribute is a 1-byte tag describing the type of the attribute, followed by the length of the attribute name stored in binary as a 2-byte integer, followed by the name, the size of the attribute value, and finally the value itself.

IPP makes no attempt to control the alignment of the binary integers embedded in the header. Some processor architectures, such as the SPARC, can't load an integer from an arbitrary address. This means that we can't store the integers in the header by casting a pointer to `int16_t` to the address in the header where the integer is to be stored. Instead, we need to copy the integer 1 byte at a time. This is why we define the `union` containing a 16-bit integer and 2 bytes.

[594–607] We store the tag in the header and convert the length of the attribute name to network byte order. We copy the length 1 byte at a time to the header. Then we copy the attribute name. We repeat this process for the attribute value and return the address in the header where the next part of the header should begin.

```
608   /*
609    * Single thread to communicate with the printer.
610    *
611    * LOCKING: acquires and releases joblock and configlock.
612    */
613   void *
614   printer_thread(void *arg)
615   {
616     struct job      *jp;
617     int             hlen, ilen, sockfd, fd, nr, nw, extra;
618     char            *icp, *hcp, *p;
619     struct ipp_hdr  *hp;
620     struct stat     sbuf;
621     struct iovec    iov[2];
622     char            name[FILENMSZ];
623     char            hbuf[HBUFSZ];
624     char            ibuf[IBUFSZ];
625     char            buf[IOBUFSZ];
626     char            str[64];
627     struct timespec ts = { 60, 0 };     /* 1 minute */

628     for (;;) {
629         /*
630          * Get a job to print.
631          */
632         pthread_mutex_lock(&joblock);
633         while (jobhead == NULL) {
634             log_msg("printer_thread: waiting...");
635             pthread_cond_wait(&jobwait, &joblock);
636         }
637         remove_job(jp = jobhead);
638         log_msg("printer_thread: picked up job %d", jp->jobid);
639         pthread_mutex_unlock(&joblock);
640         update_jobno();
```

[608–627] The `printer_thread` function is run by the thread that communicates with the network printer. We'll use `icp` and `ibuf` to build the IPP header. We'll use `hcp` and `hbuf` to build the HTTP header. We need to build the headers in separate buffers. The HTTP header includes a length field in ASCII, and we won't know how much space to reserve for it until we assemble the IPP header. We'll use `writev` to write these two headers in one call.

[628–640] The printer thread runs in an infinite loop that waits for jobs to transmit to the printer. We use the `joblock` mutex to protect the list of jobs. If a job is not pending, we use `pthread_cond_wait` to wait for one to arrive. When a job is ready, we remove it from the list by calling `remove_job`. We still hold the mutex at this point, so we release it and call `update_jobno` to write the next job number to `/var/spool/printer/jobno`.

```
641         /*
642          * Check for a change in the config file.
643          */
644         pthread_mutex_lock(&configlock);
645         if (reread) {
646             freeaddrinfo(printer);
647             printer = NULL;
648             printer_name = NULL;
649             reread = 0;
650             pthread_mutex_unlock(&configlock);
651             init_printer();
652         } else {
653             pthread_mutex_unlock(&configlock);
654         }
655         /*
656          * Send job to printer.
657          */
658         sprintf(name, "%s/%s/%d", SPOOLDIR, DATADIR, jp->jobid);
659         if ((fd = open(name, O_RDONLY)) < 0) {
660             log_msg("job %d canceled - can't open %s: %s",
661               jp->jobid, name, strerror(errno));
662             free(jp);
663             continue;
664         }
665         if (fstat(fd, &sbuf) < 0) {
666             log_msg("job %d canceled - can't fstat %s: %s",
667               jp->jobid, name, strerror(errno));
668             free(jp);
669             close(fd);
670             continue;
671         }
```

[641–654] Now that we have a job to print, we check for a change in the configuration file. We lock the `configlock` mutex and check the `reread` variable. If it is nonzero, then we free the old printer `addrinfo` list, clear the pointers, unlock the mutex, and call `init_printer` to reinitialize the printer information. Since only this context looks at and potentially changes the printer information after the `main` thread initialized it, we don't need any synchronization other than using the `configlock` mutex to protect the state of the `reread` flag.

Note that although we acquire and release two different mutex locks in this function, we never hold both at the same time, so we don't need to establish a lock hierarchy (Section 11.6.2).

[655–671] If we can't open the data file, we log an error message, free the `job` structure, and continue. After opening the file, we call `fstat` to find the size of the file. If this fails, we log an error message, clean up, and continue.

```
672         if ((sockfd = connect_retry(AF_INET, SOCK_STREAM, 0,
673           printer->ai_addr, printer->ai_addrlen)) < 0) {
674             log_msg("job %d deferred - can't contact printer: %s",
675               jp->jobid, strerror(errno));
676             goto defer;
677         }
678         /*
679          * Set up the IPP header.
680          */
681         icp = ibuf;
682         hp = (struct ipp_hdr *)icp;
683         hp->major_version = 1;
684         hp->minor_version = 1;
685         hp->operation = htons(OP_PRINT_JOB);
686         hp->request_id = htonl(jp->jobid);
687         icp += offsetof(struct ipp_hdr, attr_group);
688         *icp++ = TAG_OPERATION_ATTR;
689         icp = add_option(icp, TAG_CHARSET, "attributes-charset",
690           "utf-8");
691         icp = add_option(icp, TAG_NATULANG,
692           "attributes-natural-language", "en-us");
693         sprintf(str, "http://%s/ipp", printer_name);
694         icp = add_option(icp, TAG_URI, "printer-uri", str);
695         icp = add_option(icp, TAG_NAMEWOLANG,
696           "requesting-user-name", jp->req.usernm);
697         icp = add_option(icp, TAG_NAMEWOLANG, "job-name",
698           jp->req.jobnm);
```

[672–677] We open a stream socket connected to the printer. If the `connect_retry` call fails, we jump down to `defer`, where we will clean up, delay, and try again later.

[678–698] Next, we set up the IPP header. The operation is a print-job request. We use `htons` to convert the 2-byte operation ID from host to network byte order and `htonl` to convert the 4-byte job ID from host to network byte order. After the initial portion of the header, we set the tag value to indicate that operation attributes follow. We call `add_option` to add attributes to the message. Figure 21.5 lists the required and optional attributes for print-job requests; the first three are required. We specify the character set to be UTF-8, which the printer must support. We specify the language as `en-us`, which represents U.S. English. Another required attribute is the printer Universal Resource Identifier (URI); we set it to `http://`*printer_name*`/ipp`.

The `requesting-user-name` attribute is recommended, but not required. The `job-name` attribute is optional. Recall that the `print` command sends the name of the file being printed as the job name, which can help users distinguish among multiple pending jobs.

```
699         if (jp->req.flags & PR_TEXT) {
700             p = "text/plain";
701             extra = 1;
702         } else {
703             p = "application/postscript";
704             extra = 0;
705         }
706         icp = add_option(icp, TAG_MIMETYPE, "document-format", p);
707         *icp++ = TAG_END_OF_ATTR;
708         ilen = icp - ibuf;

709         /*
710          * Set up the HTTP header.
711          */
712         hcp = hbuf;
713         sprintf(hcp, "POST /ipp HTTP/1.1\r\n");
714         hcp += strlen(hcp);
715         sprintf(hcp, "Content-Length: %ld\r\n",
716           (long)sbuf.st_size + ilen + extra);
717         hcp += strlen(hcp);
718         strcpy(hcp, "Content-Type: application/ipp\r\n");
719         hcp += strlen(hcp);
720         sprintf(hcp, "Host: %s:%d\r\n", printer_name, IPP_PORT);
721         hcp += strlen(hcp);
722         *hcp++ = '\r';
723         *hcp++ = '\n';
724         hlen = hcp - hbuf;
```

[699–708] The last attribute we supply is the `document-format`. If we omit it, the printer will interpret the file using some default format. For a PostScript printer, this is probably PostScript, but some printers can autosense the format and choose between PostScript, plaintext, or PCL (Hewlett-Packard's Printer Command Language). If the `PR_TEXT` flag is set, we set the format to `text/plain`. Otherwise, we set it to `application/postscript`. Then we delimit the end of the attributes portion of the header with an end-of-attributes tag and calculate the size of the IPP header.

The `extra` integer counts any extra characters we might need to transmit to the printer. As we shall see shortly, we need to send an extra character to be able to print plain text reliably. We need to account for this extra character when we calculate the content length.

[709–724] Now that we know the IPP header size, we can set up the HTTP header. We set the `Content-Length` to the size in bytes of the IPP header plus the size of the file to be printed plus any extra characters we might need to send. The `Content-Type` is `application/ipp`. We mark the end of the HTTP header with a carriage return and a line feed. Finally, we calculate the size of the HTTP header.

```
725         /*
726          * Write the headers first.  Then send the file.
727          */
728         iov[0].iov_base = hbuf;
729         iov[0].iov_len = hlen;
730         iov[1].iov_base = ibuf;
731         iov[1].iov_len = ilen;
732         if (writev(sockfd, iov, 2) != hlen + ilen) {
733             log_ret("can't write to printer");
734             goto defer;
735         }
736         if (jp->req.flags & PR_TEXT) {
737             /*
738              * Hack: allow PostScript to be printed as plain text.
739              */
740             if (write(sockfd, "\b", 1) != 1) {
741                 log_ret("can't write to printer");
742                 goto defer;
743             }
744         }
745         while ((nr = read(fd, buf, IOBUFSZ)) > 0) {
746             if ((nw = writen(sockfd, buf, nr)) != nr) {
747                 if (nw < 0)
748                   log_ret("can't write to printer");
749                 else
750                   log_msg("short write (%d/%d) to printer", nw, nr);
751                 goto defer;
752             }
753         }
```

[725–735] We set the first element of the `iovec` array to refer to the HTTP header and the second element to refer to the IPP header. Then we use `writev` to send both headers to the printer. If the write fails or we write less than we requested, we log a message and jump to `defer`, where we will clean up and delay before trying again.

[736–744] Even if we specify plaintext, the Phaser 8560 will try to autosense the document format. To prevent it from recognizing the beginning of a file we want to print as plaintext, the first character we send is a backspace. This character doesn't show up in the printout and defeats the printer's ability to autosense the file format. This allows us to print the source to a PostScript file instead of printing the image resulting from the PostScript file.

[745–753] We send the data file to the printer in `IOBUFSZ` chunks. `write` can send less than we requested when the socket buffers are full, so we use `writen` to handle this case. We don't worry about this condition when we write the headers, because they are small. However, the file to print could be large.

```
754         if (nr < 0) {
755             log_ret("can't read %s", name);
756             goto defer;
757         }
758         /*
759          * Read the response from the printer.
760          */
761         if (printer_status(sockfd, jp)) {
762             unlink(name);
763             sprintf(name, "%s/%s/%d", SPOOLDIR, REQDIR, jp->jobid);
764             unlink(name);
765             free(jp);
766             jp = NULL;
767         }
768     defer:
769         close(fd);
770         if (sockfd >= 0)
771             close(sockfd);
772         if (jp != NULL) {
773             replace_job(jp);
774             nanosleep(&ts, NULL);
775         }
776       }
777     }
778     /*
779      * Read data from the printer, possibly increasing the buffer.
780      * Returns offset of end of data in buffer or -1 on failure.
781      *
782      * LOCKING: none.
783      */
784     ssize_t
785     readmore(int sockfd, char **bpp, int off, int *bszp)
```

[754–757] When we reach the end of the file, `read` will return 0. However, if `read` fails, we log an error message and jump to `defer`.

[758–767] After sending the file to the printer, we call `printer_status` to read the printer's response to our request. On success, `printer_status` returns a nonzero value and we delete the data and control files. Then we free the `job` structure, set its pointer to `NULL`, and fall through to the `defer` label.

[768–777] At the `defer` label, we close the file descriptor for the open data file. If the socket descriptor is valid, we close it. On error, `jp` will point to the job structure for the job we are trying to print, so we place the job back on the head of the pending job list and delay for 1 minute. On success, `jp` is `NULL`, so we simply go back to the top of the loop to get the next job to print.

[778–785] The `readmore` function is used to read part of the response message from the printer.

```
786   {
787     ssize_t nr;
788     char    *bp = *bpp;
789     int     bsz = *bszp;
790     if (off >= bsz) {
791         bsz += IOBUFSZ;
792         if ((bp = realloc(*bpp, bsz)) == NULL)
793             log_sys("readmore: can't allocate bigger read buffer");
794         *bszp = bsz;
795         *bpp = bp;
796     }
797     if ((nr = tread(sockfd, &bp[off], bsz-off, 1)) > 0)
798         return(off+nr);
799     else
800         return(-1);
801   }
802   /*
803    * Read and parse the response from the printer.  Return 1
804    * if the request was successful, and 0 otherwise.
805    *
806    * LOCKING: none.
807    */
808   int
809   printer_status(int sfd, struct job *jp)
810   {
811     int             i, success, code, len, found, bufsz, datsz;
812     int32_t         jobid;
813     ssize_t         nr;
814     char            *bp, *cp, *statcode, *reason, *contentlen;
815     struct ipp_hdr  *hp;
816     /*
817      * Read the HTTP header followed by the IPP response header.
818      * They can be returned in multiple read attempts.  Use the
819      * Content-Length specifier to determine how much to read.
820      */
```

[786–801] If we're at the end of the buffer, we reallocate a bigger buffer and return the new starting address and size through the `bpp` and `bszp` parameters, respectively. We read as much as the buffer will hold, starting at the end of the data already in the buffer, and return the new end-of-data offset in the buffer. If the `read` fails or the timeout expires, we return –1.

[802–820] The `printer_status` function reads the printer's response to a print-job request. We don't know how the printer will respond; it might send a response in multiple messages, send the complete response in one message, or include intermediate acknowledgements, such as HTTP `100 Continue` messages. We need to handle all these possibilities.

```
821     success = 0;
822     bufsz = IOBUFSZ;
823     if ((bp = malloc(IOBUFSZ)) == NULL)
824         log_sys("printer_status: can't allocate read buffer");

825     while ((nr = tread(sfd, bp, bufsz, 5)) > 0) {
826         /*
827          * Find the status.  Response starts with "HTTP/x.y"
828          * so we can skip the first 8 characters.
829          */
830         cp = bp + 8;
831         datsz = nr;
832         while (isspace((int)*cp))
833             cp++;
834         statcode = cp;
835         while (isdigit((int)*cp))
836             cp++;
837         if (cp == statcode) { /* Bad format; log it and move on */
838             log_msg(bp);
839         } else {
840             *cp++ = '\0';
841             reason = cp;
842             while (*cp != '\r' && *cp != '\n')
843                 cp++;
844             *cp = '\0';
845             code = atoi(statcode);
846             if (HTTP_INFO(code))
847                 continue;
848             if (!HTTP_SUCCESS(code)) { /* probable error: log it */
849                 bp[datsz] = '\0';
850                 log_msg("error: %s", reason);
851                 break;
852             }
```

[821–838] We allocate a buffer and read from the printer, expecting a response to be available within about 5 seconds. We skip the `HTTP/1.1` string and any white space that starts the message. The numeric status code should follow. If it doesn't, we log the contents of the message.

[839–844] If we have found a numeric status code in the response, we convert the first nondigit character following the status code to a null byte (this character should be some form of white space). The reason string (a text message) should follow. We search for the terminating carriage return or line feed, also terminating the text string with a null byte.

[845–852] We call the `atoi` function to convert the status code string into an integer. If this is an informational message only, we ignore it and continue the loop to read more. We expect to see either a success message or an error message. If we get an error message, we log the error and break out of the loop.

```
853             /*
854              * HTTP request was okay, but still need to check
855              * IPP status.  Search for the Content-Length.
856              */
857             i = cp - bp;
858             for (;;) {
859                 while (*cp != 'C' && *cp != 'c' && i < datsz) {
860                     cp++;
861                     i++;
862                 }
863                 if (i >= datsz) {   /* get more header */
864                     if ((nr = readmore(sfd, &bp, i, &bufsz)) < 0) {
865                         goto out;
866                     } else {
867                         cp = &bp[i];
868                         datsz += nr;
869                     }
870                 }
871                 if (strncasecmp(cp, "Content-Length:", 15) == 0) {
872                     cp += 15;
873                     while (isspace((int)*cp))
874                         cp++;
875                     contentlen = cp;
876                     while (isdigit((int)*cp))
877                         cp++;
878                     *cp++ = '\0';
879                     i = cp - bp;
880                     len = atoi(contentlen);
881                     break;
882                 } else {
883                     cp++;
884                     i++;
885                 }
886             }
```

[853–870] If the HTTP request succeeds, we need to check the IPP status. We search through the message until we find the `Content-Length` attribute. HTTP header keywords are case insensitive, so we need to check both lowercase and uppercase characters. If we run out of buffer space, we call `readmore`, which uses `realloc` to increase the buffer size. Because the buffer address might change, we need to adjust `cp` to point to the correct place in the buffer.

[871–886] We use the `strncasecmp` function to do a case-insensitive comparison. If we find the `Content-Length` attribute string, we search for its value. We convert this numeric string into an integer and break out of the `for` loop. If the comparison fails, we continue searching the buffer byte by byte. If we reach the end of the buffer without finding the `Content-Length` attribute, we read more from the printer and continue the search.

```
887             if (i >= datsz) {   /* get more header */
888                 if ((nr = readmore(sfd, &bp, i, &bufsz)) < 0) {
889                     goto out;
890                 } else {
891                     cp = &bp[i];
892                     datsz += nr;
893                 }
894             }
895             found = 0;
896             while (!found) {  /* look for end of HTTP header */
897                 while (i < datsz - 2) {
898                     if (*cp == '\n' && *(cp + 1) == '\r' &&
899                       *(cp + 2) == '\n') {
900                         found = 1;
901                         cp += 3;
902                         i += 3;
903                         break;
904                     }
905                     cp++;
906                     i++;
907                 }
908                 if (i >= datsz) {   /* get more header */
909                     if ((nr = readmore(sfd, &bp, i, &bufsz)) < 0) {
910                         goto out;
911                     } else {
912                         cp = &bp[i];
913                         datsz += nr;
914                     }
915                 }
916             }
917             if (datsz - i < len) {  /* get more header */
918                 if ((nr = readmore(sfd, &bp, i, &bufsz)) < 0) {
919                     goto out;
920                 } else {
921                     cp = &bp[i];
922                     datsz += nr;
```

[887–916] We now know the length of the message (specified by the `Content-Length` attribute). If we've exhausted the contents of the buffer, we read more from the printer. Next we search for the end of the HTTP header (a blank line). If we find it, we set the `found` flag and skip the blank line. Whenever we call `readmore`, we set `cp` to point to the same offset in the buffer that it had previously just in case the buffer address changed when it was reallocated.

[917–922] When we find the end of the HTTP header, we calculate the number of bytes that the HTTP header consumed. If the amount we've read minus the size of the HTTP header is not equal to the amount of data in the IPP message (the value we calculated from the content length), then we read some more.

```
923                 }
924             }

925             hp = (struct ipp_hdr *)cp;
926             i = ntohs(hp->status);
927             jobid = ntohl(hp->request_id);

928             if (jobid != jp->jobid) {
929                 /*
930                  * Different jobs.  Ignore it.
931                  */
932                 log_msg("jobid %d status code %d", jobid, i);
933                 break;
934             }

935             if (STATCLASS_OK(i))
936                 success = 1;
937             break;
938         }
939     }

940   out:
941     free(bp);
942     if (nr < 0) {
943         log_msg("jobid %d: error reading printer response: %s",
944           jobid, strerror(errno));
945     }
946     return(success);
947   }
```

[923–927] We get the status and job ID from the IPP header in the message. Both are stored as integers in network byte order, so we need to convert them to the host byte order by calling `ntohs` and `ntohl`, respectively.

[928–939] If the job IDs don't match, then this is not our response, so we log a message and break out of the outer `while` loop. If the IPP status indicates success, then we save the return value and break out of the loop.

[940–947] Before we return, we free the buffer we used to hold the response message. We return 1 if the print request was successful and 0 if it failed.

This concludes our look at the extended example in this chapter. The programs in this chapter were tested with a Xerox Phaser 8560 network-attached PostScript printer. Unfortunately, this printer doesn't disable its autosense feature when we set the document format to `text/plain`. This led us to use a hack to trick the printer so that it wouldn't autosense the document format when we wanted to treat a document as plaintext. An alternative is to print the source to a PostScript program using a utility such as `a2ps`(1), which encapsulates the PostScript program before printing.

21.6 Summary

This chapter has examined in detail two complete programs: a print spooler daemon that sends a print job to a network printer and a command that can be used to submit a job to be printed to the spooling daemon. This has given us a chance to see many features that we described in earlier chapters used in real programs: threads, I/O multiplexing, file I/O, reading directories, socket I/O, and signals.

Exercises

21.1 Translate the IPP error code values listed in `ipp.h` into error messages. Then modify the print spooler daemon to log a message at the end of the `printer_status` function when the IPP header indicates a printer error.

21.2 Add support to the `print` command and the `printd` daemon to allow users to request double-sided printing. Do the same for landscape and portrait page orientation.

21.3 Modify the print spooler daemon so that when it starts, it contacts the printer to find out which features are supported by the printer so that the daemon doesn't request an option that isn't supported.

21.4 Write a command to report on the status of pending print jobs.

21.5 Write a command to cancel a pending print job. Use the job ID as the argument to the command to specify which job to cancel. How can you prevent one user from canceling another user's print jobs?

21.6 Add support for multiple printers to the printer spooler. Include a way to move print jobs from one printer to another.

21.7 In the printer daemon, explain why we don't need to prod the printer thread when the signal handling thread catches `SIGHUP` and sets `reread` to 1.

21.8 In the `printer_status` function, we search for the length of the IPP message by looking for the `Content-Length` HTTP attribute. This technique won't work with a printer that responds using a chunked transfer encoding. Check RFC 2616 to see how a chunked message is formatted and then modify `printer_status` to support this form of response as well.

21.9 In the `update_jobno` function, when the next job number wraps around from the maximum positive value to 1 (see `get_newjobno` to see how this can happen), we can write a smaller number over a larger one. This can result in the daemon reading an incorrect number when it restarts. What is a simple solution to this problem?

This page intentionally left blank

Appendix A

Function Prototypes

This appendix contains the function prototypes for the standard ISO C, POSIX, and UNIX System functions described in the text. Often, we want to see only the arguments to a function ("Which argument is the file pointer for `fgets`?") or only the return value ("Does `sprintf` return a pointer or a count?"). These prototypes also show which headers need to be included to obtain the definitions of any special constants and to obtain the ISO C function prototype to help detect any compile-time errors.

The page number reference for each function prototype appears to the right of the first header file listed for the function. The page number reference gives the page containing the prototype for the function. That page should be consulted for additional information on the function.

Some functions are supported by only a few of the platforms described in this text. In addition, some platforms support function flags that other platforms don't support. In these cases, we usually list the platforms for which support is provided. In a few cases, however, we list platforms that lack support.

```
void      abort(void);
```
`<stdlib.h>` p. 365

This function never returns

```
int       accept(int sockfd, struct sockaddr *restrict addr,
              socklen_t *restrict len);
```
`<sys/socket.h>` p. 608

Returns: file (socket) descriptor if OK, –1 on error

```
int       access(const char *path, int mode);
                  <unistd.h>                                          p. 102
                  mode: R_OK, W_OK, X_OK, F_OK
                  Returns: 0 if OK, –1 on error

int       aio_cancel(int fd, struct aiocb *aiocb);
                  <aio.h>                                             p. 514
                  Returns: AIO_ALLDONE, AIO_CANCELED,
                           AIO_NOTCANCELED, or –1 on error

int       aio_error(const struct aiocb *aiocb);
                  <aio.h>                                             p. 513
                  Returns: 0 if operation succeeded, EINPROGRESS if
                           operation is still in progress, error code if
                           operation failed, or –1 on error

int       aio_fsync(int op, struct aiocb *aiocb);
                  <aio.h>                                             p. 513
                  Returns: 0 if OK, –1 on error

int       aio_read(struct aiocb *aiocb);
                  <aio.h>                                             p. 512
                  Returns: 0 if OK, –1 on error

ssize_t   aio_return(const struct aiocb *aiocb);
                  <aio.h>                                             p. 513
                  Returns: result of asynchronous operation, –1 on error

int       aio_suspend(const struct aiocb *const list[], int nent,
                      const struct timespec *timeout);
                  <aio.h>                                             p. 514
                  Returns: 0 if OK, –1 on error

int       aio_write(struct aiocb *aiocb);
                  <aio.h>                                             p. 512
                  Returns: 0 if OK, –1 on error

unsigned
int       alarm(unsigned int seconds);
                  <unistd.h>                                          p. 338
                  Returns: 0 or number of seconds until previously set alarm

int       atexit(void (*func)(void));
                  <stdlib.h>                                          p. 200
                  Returns: 0 if OK, nonzero on error
```

```
int       bind(int sockfd, const struct sockaddr *addr, socklen_t len);
                    <sys/socket.h>                                    p. 604
                    Returns: 0 if OK, –1 on error

void     *calloc(size_t nobj, size_t size);
                    <stdlib.h>                                        p. 207
                    Returns: non-null pointer if OK, NULL on error

speed_t   cfgetispeed(const struct termios *termptr);
                    <termios.h>                                       p. 692
                    Returns: baud rate value

speed_t   cfgetospeed(const struct termios *termptr);
                    <termios.h>                                       p. 692
                    Returns: baud rate value

int       cfsetispeed(struct termios *termptr, speed_t speed);
                    <termios.h>                                       p. 692
                    Returns: 0 if OK, –1 on error

int       cfsetospeed(struct termios *termptr, speed_t speed);
                    <termios.h>                                       p. 692
                    Returns: 0 if OK, –1 on error

int       chdir(const char *path);
                    <unistd.h>                                        p. 135
                    Returns: 0 if OK, –1 on error

int       chmod(const char *path, mode_t mode);
                    <sys/stat.h>                                      p. 106
                    mode: S_IS[UG]ID, S_ISVTX,
                          S_I[RWX](USR|GRP|OTH)
                    Returns: 0 if OK, –1 on error

int       chown(const char *path, uid_t owner, gid_t group);
                    <unistd.h>                                        p. 109
                    Returns: 0 if OK, –1 on error

void      clearerr(FILE *fp);
                    <stdio.h>                                         p. 151

int       clock_getres(clockid_t clock_id, struct timespec *tsp);
                    <sys/time.h>                                      p. 190
                    clock_id: CLOCK_REALTIME, CLOCK_MONOTONIC,
                              CLOCK_PROCESS_CPUTIME_ID,
                              CLOCK_THREAD_CPUTIME_ID
                    Returns: 0 if OK, –1 on error
```

```
int       clock_gettime(clockid_t clock_id, struct timespec *tsp);
```
<sys/time.h> p. 189

clock_id: CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID

Returns: 0 if OK, –1 on error

```
int       clock_nanosleep(clockid_t clock_id, int flags,
                          const struct timespec *reqtp,
                          struct timespec *remtp);
```
<time.h> p. 375

clock_id: CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID

flags: TIMER_ABSTIME

Returns: 0 if slept for requested time, error number on failure

```
int       clock_settime(clockid_t clock_id, const struct timespec *tsp);
```
<sys/time.h> p. 190

clock_id: CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID

Returns: 0 if OK, –1 on error

```
int       close(int fd);
```
<unistd.h> p. 66

Returns: 0 if OK, –1 on error

```
int       closedir(DIR *dp);
```
<dirent.h> p. 130

Returns: 0 if OK, –1 on error

```
void      closelog(void);
```
<syslog.h> p. 470

```
unsigned
char     *CMSG_DATA(struct cmsghdr *cp);
```
<sys/socket.h> p. 645

Returns: pointer to data associated with cmsghdr structure

```
struct
cmsghdr  *CMSG_FIRSTHDR(struct msghdr *mp);
```
<sys/socket.h> p. 645

Returns: pointer to first cmsghdr structure associated with the msghdr structure, or NULL if none exists

```
unsigned
int       CMSG_LEN(unsigned int nbytes);
                    <sys/socket.h>                                    p. 645
                    Returns: size to allocate for data object nbytes large

struct
cmsghdr  *CMSG_NXTHDR(struct msghdr *mp, struct cmsghdr *cp);
                    <sys/socket.h>                                    p. 645
                    Returns: pointer to next cmsghdr structure associated with
                             the msghdr structure given the current cmsghdr
                             structure, or NULL if we're at the last one

int       connect(int sockfd, const struct sockaddr *addr,
                  socklen_t len);
                    <sys/socket.h>                                    p. 605
                    Returns: 0 if OK, –1 on error

int       creat(const char *path, mode_t mode);
                    <fcntl.h>                                         p. 66
                    mode: S_IS[UG]ID, S_ISVTX,
                          S_I[RWX](USR|GRP|OTH)
                    Returns: file descriptor opened for write-only if OK,
                             –1 on error

char     *ctermid(char *ptr);
                    <stdio.h>                                         p. 694
                    Returns: pointer to name of controlling terminal on success,
                             pointer to empty string on error

int       dprintf(int fd, const char *restrict format, ...);
                    <stdio.h>                                         p. 159
                    Returns: number of characters output if OK, negative value
                             if output error

int       dup(int fd);
                    <unistd.h>                                        p. 79
                    Returns: new file descriptor if OK, –1 on error

int       dup2(int fd, int fd2);
                    <unistd.h>                                        p. 79
                    Returns: new file descriptor if OK, –1 on error

void      endgrent(void);
                    <grp.h>                                           p. 183

void      endhostent(void);
                    <netdb.h>                                         p. 597
```

```
void      endnetent(void);
                    <netdb.h>                                          p. 598

void      endprotoent(void);
                    <netdb.h>                                          p. 598

void      endpwent(void);
                    <pwd.h>                                            p. 180

void      endservent(void);
                    <netdb.h>                                          p. 599

void      endspent(void);
                    <shadow.h>                                         p. 182
                    Platforms: Linux 3.2.0, Solaris 10

int       execl(const char *path, const char *arg0, ...
                /* (char *) 0 */ );
                    <unistd.h>                                         p. 249
                    Returns: –1 on error, no return on success

int       execle(const char *path, const char *arg0, ... /* (char *) 0,
                 char *const envp[] */ );
                    <unistd.h>                                         p. 249
                    Returns: –1 on error, no return on success

int       execlp(const char *filename, const char *arg0, ...
                 /* (char *) 0 */ );
                    <unistd.h>                                         p. 249
                    Returns: –1 on error, no return on success

int       execv(const char *path, char *const argv[]);
                    <unistd.h>                                         p. 249
                    Returns: –1 on error, no return on success

int       execve(const char *path, char *const argv[],
                 char *const envp[]);
                    <unistd.h>                                         p. 249
                    Returns: –1 on error, no return on success

int       execvp(const char *filename, char *const argv[]);
                    <unistd.h>                                         p. 249
                    Returns: –1 on error, no return on success

void      _Exit(int status);
                    <stdlib.h>                                         p. 198
                    This function never returns
```

```
void      _exit(int status);
                  <unistd.h>                                          p. 198
                  This function never returns

void      exit(int status);
                  <stdlib.h>                                          p. 198
                  This function never returns

int       faccessat(int fd, const char *path, int mode, int flag);
                  <unistd.h>                                          p. 102
                  mode: R_OK, W_OK, X_OK, F_OK
                  flag: AT_EACCESS
                  Returns: 0 if OK, –1 on error

int       fchdir(int fd);
                  <unistd.h>                                          p. 135
                  Returns: 0 if OK, –1 on error

int       fchmod(int fd, mode_t mode);
                  <sys/stat.h>                                        p. 106
                  mode: S_IS[UG]ID, S_ISVTX,
                        S_I[RWX](USR|GRP|OTH)
                  Returns: 0 if OK, –1 on error

int       fchmodat(int fd, const char *path, mode_t mode, int flag);
                  <sys/stat.h>                                        p. 106
                  mode: S_IS[UG]ID, S_ISVTX,
                        S_I[RWX](USR|GRP|OTH)
                  flag: AT_SYMLINK_NOFOLLOW
                  Returns: 0 if OK, –1 on error

int       fchown(int fd, uid_t owner, gid_t group);
                  <unistd.h>                                          p. 109
                  Returns: 0 if OK, –1 on error

int       fchownat(int fd, const char *path, uid_t owner,
                   gid_t group, int flag);
                  <unistd.h>                                          p. 109
                  flag: AT_SYMLINK_NOFOLLOW
                  Returns: 0 if OK, –1 on error

int       fclose(FILE *fp);
                  <stdio.h>                                           p. 150
                  Returns: 0 if OK, EOF on error
```

```
int       fcntl(int fd, int cmd, ... /* int arg */ );
                  <fcntl.h>                                          p. 82
                  cmd: F_DUPFD, F_DUPFD_CLOEXEC, F_GETFD,
                       F_SETFD, F_GETFL, F_SETFL, F_GETOWN,
                       F_SETOWN, F_GETLK, F_SETLK, F_SETLKW
                  Returns: depends on cmd if OK, –1 on error

int       fdatasync(int fd);
                  <unistd.h>                                         p. 81
                  Returns: 0 if OK, –1 on error
                  Platforms: Linux 3.2.0, Solaris 10

void      FD_CLR(int fd, fd_set *fdset);
                  <sys/select.h>                                     p. 503

int       FD_ISSET(int fd, fd_set *fdset);
                  <sys/select.h>                                     p. 503
                  Returns: nonzero if fd is in set, 0 otherwise

FILE     *fdopen(int fd, const char *type);
                  <stdio.h>                                          p. 148
                  type: "r", "w", "a", "r+", "w+", "a+"
                  Returns: file pointer if OK, NULL on error

DIR      *fdopendir(int fd);
                  <dirent.h>                                         p. 130
                  Returns: pointer if OK, NULL on error

void      FD_SET(int fd, fd_set *fdset);
                  <sys/select.h>                                     p. 503

void      FD_ZERO(fd_set *fdset);
                  <sys/select.h>                                     p. 503

int       feof(FILE *fp);
                  <stdio.h>                                          p. 151
                  Returns: nonzero (true) if end of file on stream,
                           0 (false) otherwise

int       ferror(FILE *fp);
                  <stdio.h>                                          p. 151
                  Returns: nonzero (true) if error on stream, 0 (false) otherwise

int       fexecve(int fd, char *const argv[], char *const envp[]);
                  <unistd.h>                                         p. 249
                  Returns: –1 on error, no return on success
```

```
int       fflush(FILE *fp);
                  <stdio.h>                                          p. 147
                  Returns: 0 if OK, EOF on error

int       fgetc(FILE *fp);
                  <stdio.h>                                          p. 150
                  Returns: next character if OK, EOF on end of file or error

int       fgetpos(FILE *restrict fp, fpos_t *restrict pos);
                  <stdio.h>                                          p. 158
                  Returns: 0 if OK, nonzero on error

char     *fgets(char *restrict buf, int n, FILE *restrict fp);
                  <stdio.h>                                          p. 152
                  Returns: buf if OK, NULL on end of file or error

int       fileno(FILE *fp);
                  <stdio.h>                                          p. 164
                  Returns: file descriptor associated with the stream, –1 on error

void      flockfile(FILE *fp);
                  <stdio.h>                                          p. 443

FILE     *fmemopen(void *restrict buf, size_t size,
                   const char *restrict type);
                  <stdio.h>                                          p. 171
                  type: "r", "w", "a", "r+", "w+", "a+"
                  Returns: stream pointer if OK, NULL on error

FILE     *fopen(const char *restrict path, const char *restrict type);
                  <stdio.h>                                          p. 148
                  type: "r", "w", "a", "r+", "w+", "a+"
                  Returns: file pointer if OK, NULL on error

pid_t     fork(void);
                  <unistd.h>                                         p. 229
                  Returns: 0 in child, process ID of child in parent, –1 on error

long      fpathconf(int fd, int name);
                  <unistd.h>                                         p. 42
                  name: _PC_ASYNC_IO, _PC_CHOWN_RESTRICTED,
                        _PC_FILESIZEBITS, _PC_LINK_MAX,
                        _PC_MAX_CANON, _PC_MAX_INPUT,
                        _PC_NAME_MAX, _PC_NO_TRUNC, _PC_PATH_MAX,
                        _PC_PIPE_BUF, _PC_PRIO_IO, _PC_SYMLINK_MAX,
                        _PC_SYNC_IO, _PC_TIMESTAMP_RESOLUTION,
                        _PC_2_SYMLINKS, _PC_VDISABLE
                  Returns: corresponding value if OK, –1 on error
```

```
int       fprintf(FILE *restrict fp, const char *restrict format, ...);
                  <stdio.h>                                          p. 159
                  Returns: number of characters output if OK, negative value
                        if output error

int       fputc(int c, FILE *fp);
                  <stdio.h>                                          p. 152
                  Returns: c if OK, EOF on error

int       fputs(const char *restrict str, FILE *restrict fp);
                  <stdio.h>                                          p. 153
                  Returns: non-negative value if OK, EOF on error

size_t    fread(void *restrict ptr, size_t size, size_t nobj,
                FILE *restrict fp);
                  <stdio.h>                                          p. 156
                  Returns: number of objects read

void      free(void *ptr);
                  <stdlib.h>                                         p. 207

void      freeaddrinfo(struct addrinfo *ai);
                  <sys/socket.h>                                     p. 599
                  <netdb.h>

FILE     *freopen(const char *restrict path, const char *restrict type,
                  FILE *restrict fp);
                  <stdio.h>                                          p. 148
                  type: "r", "w", "a", "r+", "w+", "a+"
                  Returns: file pointer if OK, NULL on error

int       fscanf(FILE *restrict fp, const char *restrict format, ...);
                  <stdio.h>                                          p. 162
                  Returns: number of input items assigned, EOF if input
                        error or end of file before any conversion

int       fseek(FILE *fp, long offset, int whence);
                  <stdio.h>                                          p. 158
                  whence: SEEK_SET, SEEK_CUR, SEEK_END
                  Returns: 0 if OK, –1 on error

int       fseeko(FILE *fp, off_t offset, int whence);
                  <stdio.h>                                          p. 158
                  whence: SEEK_SET, SEEK_CUR, SEEK_END
                  Returns: 0 if OK, –1 on error
```

```
int       fsetpos(FILE *fp, const fpos_t *pos);
                  <stdio.h>                                          p. 158
                  Returns: 0 if OK, nonzero on error

int       fstat(int fd, struct stat *buf);
                  <sys/stat.h>                                       p. 93
                  Returns: 0 if OK, –1 on error

int       fstatat(int fd, const char *restrict path,
                struct stat *restrict buf, int flag);
                  <sys/stat.h>                                       p. 93
                  flag: AT_SYMLINK_NOFOLLOW
                  Returns: 0 if OK, –1 on error

int       fsync(int fd);
                  <unistd.h>                                         p. 81
                  Returns: 0 if OK, –1 on error

long      ftell(FILE *fp);
                  <stdio.h>                                          p. 158
                  Returns: current file position indicator if OK, –1L on error

off_t     ftello(FILE *fp);
                  <stdio.h>                                          p. 158
                  Returns: current file position indicator if OK,
                           (off_t)-1 on error

key_t     ftok(const char *path, int id);
                  <sys/ipc.h>                                        p. 557
                  Returns: key if OK, (key_t)-1 on error

int       ftruncate(int fd, off_t length);
                  <unistd.h>                                         p. 112
                  Returns: 0 if OK, –1 on error

int       ftrylockfile(FILE *fp);
                  <stdio.h>                                          p. 443
                  Returns: 0 if OK, nonzero if lock can't be acquired

void      funlockfile(FILE *fp);
                  <stdio.h>                                          p. 443

int       futimens(int fd, const struct timespec times[2]);
                  <sys/stat.h>                                       p. 126
                  Returns: 0 if OK, –1 on error
```

```
int       fwide(FILE *fp, int mode);
                  <stdio.h>                                          p. 144
                  <wchar.h>
                  Returns: positive if stream is wide oriented, negative if stream
                           is byte oriented, or 0 if stream has no orientation

size_t    fwrite(const void *restrict ptr, size_t size, size_t nobj,
                 FILE *restrict fp);
                  <stdio.h>                                          p. 156
                  Returns: number of objects written

const
char     *gai_strerror(int error);
                  <netdb.h>                                          p. 600
                  Returns: a pointer to a string describing the error

int       getaddrinfo(const char *restrict host,
                      const char *restrict service,
                      const struct addrinfo *restrict hint,
                      struct addrinfo **restrict res);
                  <sys/socket.h>                                     p. 599
                  <netdb.h>
                  Returns: 0 if OK, nonzero error code on error

int       getc(FILE *fp);
                  <stdio.h>                                          p. 150
                  Returns: next character if OK, EOF on end of file or error

int       getchar(void);
                  <stdio.h>                                          p. 150
                  Returns: next character if OK, EOF on end of file or error

int       getchar_unlocked(void);
                  <stdio.h>                                          p. 444
                  Returns: the next character if OK, EOF on end of file or error

int       getc_unlocked(FILE *fp);
                  <stdio.h>                                          p. 444
                  Returns: the next character if OK, EOF on end of file or error

char     *getcwd(char *buf, size_t size);
                  <unistd.h>                                         p. 136
                  Returns: buf if OK, NULL on error

gid_t     getegid(void);
                  <unistd.h>                                         p. 228
                  Returns: effective group ID of calling process
```

```
char     *getenv(const char *name);
                    <stdlib.h>                                    p. 210
                    Returns: pointer to value associated with name,
                          NULL if not found

uid_t     geteuid(void);
                    <unistd.h>                                    p. 228
                    Returns: effective user ID of calling process

gid_t     getgid(void);
                    <unistd.h>                                    p. 228
                    Returns: real group ID of calling process

struct
group    *getgrent(void);
                    <grp.h>                                       p. 183
                    Returns: pointer if OK, NULL on error or end of file

struct
group    *getgrgid(gid_t gid);
                    <grp.h>                                       p. 182
                    Returns: pointer if OK, NULL on error

struct
group    *getgrnam(const char *name);
                    <grp.h>                                       p. 182
                    Returns: pointer if OK, NULL on error

int       getgroups(int gidsetsize, gid_t grouplist[]);
                    <unistd.h>                                    p. 184
                    Returns: number of supplementary group IDs if OK,
                          –1 on error

struct
hostent  *gethostent(void);
                    <netdb.h>                                     p. 597
                    Returns: pointer if OK, NULL on error

int       gethostname(char *name, int namelen);
                    <unistd.h>                                    p. 188
                    Returns: 0 if OK, –1 on error

char     *getlogin(void);
                    <unistd.h>                                    p. 275
                    Returns: pointer to string giving login name if OK,
                          NULL on error
```

```
int       getnameinfo(const struct sockaddr *restrict addr,
                      socklen_t alen, char *restrict host,
                      socklen_t hostlen, char *restrict service,
                      socklen_t servlen, unsigned int flags);
```
<sys/socket.h> p. 600
<netdb.h>
flags: NI_DGRAM, NI_NAMEREQD, NI_NOFQDN, NI_NUMERICHOST, NI_NUMERICSCOPE, NI_NUMERICSERV
Returns: 0 if OK, nonzero on error

```
struct
netent    *getnetbyaddr(uint32_t net, int type);
```
<netdb.h> p. 598
Returns: pointer if OK, NULL on error

```
struct
netent    *getnetbyname(const char *name);
```
<netdb.h> p. 598
Returns: pointer if OK, NULL on error

```
struct
netent    *getnetent(void);
```
<netdb.h> p. 598
Returns: pointer if OK, NULL on error

```
int       getopt(int argc, char * const argv[], const char *options);
```
<fcntl.h> p. 662
extern int opterr, optind, optopt;
extern char *optarg;
Returns: the next option character, or –1 when all options have been processed

```
int       getpeername(int sockfd, struct sockaddr *restrict addr,
                      socklen_t *restrict alenp);
```
<sys/socket.h> p. 605
Returns: 0 if OK, –1 on error

```
pid_t     getpgid(pid_t pid);
```
<unistd.h> p. 294
Returns: process group ID if OK, –1 on error

```
pid_t     getpgrp(void);
```
<unistd.h> p. 293
Returns: process group ID of calling process

```
pid_t     getpid(void);
                        <unistd.h>                                    p. 228
                        Returns: process ID of calling process

pid_t     getppid(void);
                        <unistd.h>                                    p. 228
                        Returns: parent process ID of calling process

int       getpriority(int which, id_t who);
                        <sys/resource.h>                              p. 277
                        which: PRIO_PROCESS, PRIO_PGRP, PRIO_USER
                        Returns: nice value between –NZERO and NZERO–1 if OK,
                                 –1 on error

struct
protoent *getprotobyname(const char *name);
                        <netdb.h>                                     p. 598
                        Returns: pointer if OK, NULL on error

struct
protoent *getprotobynumber(int proto);
                        <netdb.h>                                     p. 598
                        Returns: pointer if OK, NULL on error

struct
protoent *getprotoent(void);
                        <netdb.h>                                     p. 598
                        Returns: pointer if OK, NULL on error

struct
passwd   *getpwent(void);
                        <pwd.h>                                       p. 180
                        Returns: pointer if OK, NULL on error or end of file

struct
passwd   *getpwnam(const char *name);
                        <pwd.h>                                       p. 179
                        Returns: pointer if OK, NULL on error

struct
passwd   *getpwuid(uid_t uid);
                        <pwd.h>                                       p. 179
                        Returns: pointer if OK, NULL on error
```

```
int       getrlimit(int resource, struct rlimit *rlptr);
                    <sys/resource.h>                                    p. 220
                    resource: RLIMIT_CORE, RLIMIT_CPU,
                              RLIMIT_DATA, RLIMIT_FSIZE,
                              RLIMIT_NOFILE, RLIMIT_STACK,
                              RLIMIT_AS (FreeBSD 8.0, Linux 3.2.0,
                                         Solaris 10),
                              RLIMIT_MEMLOCK (FreeBSD 8.0, Linux 3.2.0,
                                              Mac OS X 10.6.8),
                              RLIMIT_MSGQUEUE (Linux 3.2.0),
                              RLIMIT_NICE (Linux 3.2.0),
                              RLIMIT_NPROC (FreeBSD 8.0, Linux 3.2.0,
                                            Mac OS X 10.6.8),
                              RLIMIT_NPTS (FreeBSD 8.0),
                              RLIMIT_RSS (FreeBSD 8.0, Linux 3.2.0,
                                          Mac OS X 10.6.8),
                              RLIMIT_SBSIZE (FreeBSD 8.0),
                              RLIMIT_SIGPENDING (Linux 3.2.0),
                              RLIMIT_SWAP (FreeBSD 8.0),
                              RLIMIT_VMEM (Solaris 10)
                    Returns: 0 if OK, –1 on error

char      *gets(char *buf);
                    <stdio.h>                                           p. 152
                    Returns: buf if OK, NULL on end of file or error

struct
servent   *getservbyname(const char *name, const char *proto);
                    <netdb.h>                                           p. 599
                    Returns: pointer if OK, NULL on error

struct
servent   *getservbyport(int port, const char *proto);
                    <netdb.h>                                           p. 599
                    Returns: pointer if OK, NULL on error

struct
servent   *getservent(void);
                    <netdb.h>                                           p. 599
                    Returns: pointer if OK, NULL on error

pid_t     getsid(pid_t pid);
                    <unistd.h>                                          p. 296
                    Returns: session leader's process group ID if OK,
                             –1 on error

int       getsockname(int sockfd, struct sockaddr *restrict addr,
                      socklen_t *restrict alenp);
                    <sys/socket.h>                                      p. 605
                    Returns: 0 if OK, –1 on error
```

```
int       getsockopt(int sockfd, int level, int option, void *restrict val,
                     socklen_t *restrict lenp);
                       <sys/socket.h>                                    p. 624
                       Returns: 0 if OK, –1 on error

struct
spwd     *getspent(void);
                       <shadow.h>                                        p. 182
                       Returns: pointer if OK, NULL on error
                       Platforms: Linux 3.2.0, Solaris 10

struct
spwd     *getspnam(const char *name);
                       <shadow.h>                                        p. 182
                       Returns: pointer if OK, NULL on error
                       Platforms: Linux 3.2.0, Solaris 10

int       gettimeofday(struct timeval *restrict tp,
                       void *restrict tzp);
                       <sys/time.h>                                      p. 190
                       Returns: 0 always

uid_t     getuid(void);
                       <unistd.h>                                        p. 228
                       Returns: real user ID of calling process

struct
tm       *gmtime(const time_t *calptr);
                       <time.h>                                          p. 192
                       Returns: pointer to broken-down time, NULL on error

int       grantpt(int fd);
                       <stdlib.h>                                        p. 723
                       Returns: 0 on success, –1 on error

uint32_t  htonl(uint32_t hostint32);
                       <arpa/inet.h>                                     p. 594
                       Returns: 32-bit integer in network byte order

uint16_t  htons(uint16_t hostint16);
                       <arpa/inet.h>                                     p. 594
                       Returns: 16-bit integer in network byte order
```

```
const
char     *inet_ntop(int domain, const void *restrict addr,
                    char *restrict str, socklen_t size);
                       <arpa/inet.h>                                    p. 596
                       Returns: pointer to address string on success, NULL on error

int      inet_pton(int domain, const char *restrict str,
                   void *restrict addr);
                       <arpa/inet.h>                                    p. 596
                       Returns: 1 on success, 0 if the format is invalid, or –1 on error

int      initgroups(const char *username, gid_t basegid);
                       <grp.h>    /* Linux & Solaris */                 p. 184
                       <unistd.h> /* FreeBSD & Mac OS X */
                       Returns: 0 if OK, –1 on error

int      ioctl(int fd, int request, ...);
                       <unistd.h>     /* System V */                    p. 87
                       <sys/ioctl.h>  /* BSD and Linux */
                       Returns: –1 on error, something else if OK

int      isatty(int fd);
                       <unistd.h>                                       p. 695
                       Returns: 1 (true) if terminal device, 0 (false) otherwise

int      kill(pid_t pid, int signo);
                       <signal.h>                                       p. 337
                       Returns: 0 if OK, –1 on error

int      lchown(const char *path, uid_t owner, gid_t group);
                       <unistd.h>                                       p. 109
                       Returns: 0 if OK, –1 on error

int      link(const char *existingpath, const char *newpath);
                       <unistd.h>                                       p. 116
                       Returns: 0 if OK, –1 on error

int      linkat(int efd, const char *existingpath, int nfd,
                const char *newpath, int flag);
                       <unistd.h>                                       p. 116
                       flag: AT_SYMLINK_NOFOLLOW
                       Returns: 0 if OK, –1 on error
```

```
int       lio_listio(int mode,
                     struct aiocb *restrict const list[restrict],
                     int nent, struct sigevent *restrict sigev);
                      <aio.h>                                          p. 515
                      mode: LIO_NOWAIT, LIO_WAIT
                      Returns: 0 if OK, –1 on error

int       listen(int sockfd, int backlog);
                      <sys/socket.h>                                   p. 608
                      Returns: 0 if OK, –1 on error

struct
tm       *localtime(const time_t *calptr);
                      <time.h>                                         p. 192
                      Returns: pointer to broken-down time, NULL on error

void      longjmp(jmp_buf env, int val);
                      <setjmp.h>                                       p. 215
                      This function never returns

off_t     lseek(int fd, off_t offset, int whence);
                      <unistd.h>                                       p. 67
                      whence: SEEK_SET, SEEK_CUR, SEEK_END
                      Returns: new file offset if OK, –1 on error

int       lstat(const char *restrict path,
                struct stat *restrict buf);
                      <sys/stat.h>                                     p. 93
                      Returns: 0 if OK, –1 on error

void     *malloc(size_t size);
                      <stdlib.h>                                       p. 207
                      Returns: non-null pointer if OK, NULL on error

int       mkdir(const char *path, mode_t mode);
                      <sys/stat.h>                                     p. 129
                      mode: S_IS[UG]ID, S_ISVTX,
                            S_I[RWX](USR|GRP|OTH)
                      Returns: 0 if OK, –1 on error

int       mkdirat(int fd, const char *path, mode_t mode);
                      <sys/stat.h>                                     p. 129
                      mode: S_IS[UG]ID, S_ISVTX,
                            S_I[RWX](USR|GRP|OTH)
                      Returns: 0 if OK, –1 on error
```

```
char     *mkdtemp(char *template);
                    <stdlib.h>                                          p. 169
                    Returns: pointer to directory name if OK, NULL on error

int      mkfifo(const char *path, mode_t mode);
                    <sys/stat.h>                                        p. 553
                    mode: S_IS[UG]ID, S_ISVTX,
                          S_I[RWX](USR|GRP|OTH)
                    Returns: 0 if OK, –1 on error

int      mkfifoat(int fd, const char *path, mode_t mode);
                    <sys/stat.h>                                        p. 553
                    mode: S_IS[UG]ID, S_ISVTX,
                          S_I[RWX](USR|GRP|OTH)
                    Returns: 0 if OK, –1 on error

int      mkstemp(char *template);
                    <stdlib.h>                                          p. 169
                    Returns: file descriptor if OK, –1 on error

time_t   mktime(struct tm *tmptr);
                    <time.h>                                            p. 192
                    Returns: calendar time if OK, –1 on error

void     *mmap(void *addr, size_t len, int prot, int flag, int fd,
              off_t off);
                    <sys/mman.h>                                        p. 525
                    prot: PROT_READ, PROT_WRITE, PROT_EXEC,
                          PROT_NONE
                    flag: MAP_FIXED, MAP_SHARED, MAP_PRIVATE
                    Returns: starting address of mapped region if OK,
                             MAP_FAILED on error

int      mprotect(void *addr, size_t len, int prot);
                    <sys/mman.h>                                        p. 527
                    Returns: 0 if OK, –1 on error

int      msgctl(int msqid, int cmd, struct msqid_ds *buf);
                    <sys/msg.h>                                         p. 562
                    cmd: IPC_STAT, IPC_SET, IPC_RMID
                    Returns: 0 if OK, –1 on error

int      msgget(key_t key, int flag);
                    <sys/msg.h>                                         p. 562
                    flag: IPC_CREAT, IPC_EXCL
                    Returns: message queue ID if OK, –1 on error
```

```
ssize_t   msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
                    <sys/msg.h>                                          p. 564
                    flag: IPC_NOWAIT, MSG_NOERROR
                    Returns: size of data portion of message if OK, –1 on error

int       msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
                    <sys/msg.h>                                          p. 563
                    flag: IPC_NOWAIT
                    Returns: 0 if OK, –1 on error

int       msync(void *addr, size_t len, int flags);
                    <sys/mman.h>                                         p. 528
                    flag: MS_ASYNC, MS_INVALIDATE, MS_SYNC
                    Returns: 0 if OK, –1 on error

int       munmap(void *addr, size_t len);
                    <sys/mman.h>                                         p. 528
                    Returns: 0 if OK, –1 on error

int       nanosleep(const struct timespec *reqtp,
                    struct timespec *remtp);
                    <time.h>                                             p. 374
                    Returns: 0 if slept for requested time, –1 on error

int       nice(int incr);
                    <unistd.h>                                           p. 276
                    Returns: new nice value – NZERO if OK, –1 on error

uint32_t  ntohl(uint32_t netint32);
                    <arpa/inet.h>                                        p. 594
                    Returns: 32-bit integer in host byte order

uint16_t  ntohs(uint16_t netint16);
                    <arpa/inet.h>                                        p. 594
                    Returns: 16-bit integer in host byte order

int       open(const char *path, int oflag, ... /* mode_t mode */ );
                    <fcntl.h>                                            p. 62
                    oflag: O_RDONLY, O_WRONLY, O_RDWR, O_EXEC,
                           O_SEARCH;
                           O_APPEND, O_CLOEXEC, O_CREAT,
                           O_DIRECTORY, O_DSYNC, O_EXCL,
                           O_NOCTTY, O_NOFOLLOW, O_NONBLOCK,
                           O_RSYNC, O_SYNC, O_TRUNC, O_TTY_INIT
                    mode: S_IS[UG]ID, S_ISVTX,
                          S_I[RWX](USR|GRP|OTH)
                    Returns: file descriptor if OK, –1 on error
                    Platforms: O_FSYNC flag on FreeBSD 8.0 and Mac OS X 10.6.8
```

```
int       openat(int fd, const char *path, int oflag, ...
                  /* mode_t mode */ );
                        <fcntl.h>                                   p. 62
                        oflag: O_RDONLY, O_WRONLY, O_RDWR, O_EXEC,
                               O_SEARCH;
                               O_APPEND, O_CLOEXEC, O_CREAT,
                               O_DIRECTORY, O_DSYNC, O_EXCL,
                               O_NOCTTY, O_NOFOLLOW, O_NONBLOCK,
                               O_RSYNC, O_SYNC, O_TRUNC, O_TTY_INIT
                        mode: S_IS[UG]ID, S_ISVTX,
                               S_I[RWX](USR|GRP|OTH)
                        Returns: file descriptor if OK, –1 on error
                        Platforms: O_FSYNC flag on FreeBSD 8.0 and Mac OS X 10.6.8

DIR      *opendir(const char *path);
                        <dirent.h>                                  p. 130
                        Returns: pointer if OK, NULL on error

void      openlog(const char *ident, int option, int facility);
                        <syslog.h>                                  p. 470
                        option: LOG_CONS, LOG_NDELAY, LOG_NOWAIT,
                                LOG_ODELAY, LOG_PERROR, LOG_PID
                        facility: LOG_AUTH, LOG_AUTHPRIV, LOG_CRON,
                                  LOG_DAEMON, LOG_FTP, LOG_KERN,
                                  LOG_LOCAL[0-7], LOG_LPR, LOG_MAIL,
                                  LOG_NEWS, LOG_SYSLOG, LOG_USER, LOG_UUCP

FILE     *open_memstream(char **bufp, size_t *sizep);
                        <stdio.h>                                   p. 173
                        Returns: stream pointer if OK, NULL on error

FILE     *open_wmemstream(wchar_t **bufp, size_t *sizep);
                        <wchar.h>                                   p. 173
                        Returns: stream pointer if OK, NULL on error

long      pathconf(const char *path, int name);
                        <unistd.h>                                  p. 42
                        name: _PC_ASYNC_IO, _PC_CHOWN_RESTRICTED,
                              _PC_FILESIZEBITS, _PC_LINK_MAX,
                              _PC_MAX_CANON, _PC_MAX_INPUT,
                              _PC_NAME_MAX, _PC_NO_TRUNC, _PC_PATH_MAX,
                              _PC_PIPE_BUF, _PC_PRIO_IO, _PC_SYMLINK_MAX,
                              _PC_SYNC_IO, _PC_TIMESTAMP_RESOLUTION,
                              _PC_2_SYMLINKS, _PC_VDISABLE
                        Returns: corresponding value if OK, –1 on error

int       pause(void);
                        <unistd.h>                                  p. 338
                        Returns: –1 with errno set to EINTR
```

```
int       pclose(FILE *fp);
                  <stdio.h>                                          p. 541
                  Returns: termination status of popen cmdstring, –1 on error

void      perror(const char *msg);
                  <stdio.h>                                          p. 15

int       pipe(int fd[2]);
                  <unistd.h>                                         p. 535
                  Returns: 0 if OK, –1 on error

int       poll(struct pollfd fdarray[], nfds_t nfds, int timeout);
                  <poll.h>                                           p. 506
                  Returns: count of ready descriptors, 0 on timeout, –1 on error

FILE     *popen(const char *cmdstring, const char *type);
                  <stdio.h>                                          p. 541
                  type: "r", "w"
                  Returns: file pointer if OK, NULL on error

int       posix_openpt(int oflag);
                  <stdlib.h>                                         p. 722
                  <fcntl.h>
                  oflag: O_RWDR, O_NOCTTY
                  Returns: file descriptor of next available PTY master if OK,
                           –1 on error

ssize_t   pread(int fd, void *buf, size_t nbytes, off_t offset);
                  <unistd.h>                                         p. 78
                  Returns: number of bytes read, 0 if end of file, –1 on error

int       printf(const char *restrict format, ...);
                  <stdio.h>                                          p. 159
                  Returns: number of characters output if OK, negative value
                           if output error

int       pselect(int maxfdp1, fd_set *restrict readfds,
                  fd_set *restrict writefds, fd_set *restrict exceptfds,
                  const struct timespec *restrict tsptr,
                  const sigset_t *restrict sigmask);
                  <sys/select.h>                                     p. 506
                  Returns: count of ready descriptors, 0 on timeout, –1 on error

void      psiginfo(const siginfo_t *info, const char *msg);
                  <signal.h>                                         p. 379
```

```
void      psignal(int signo, const char *msg);
                  <signal.h>                                        p. 379
                  <siginfo.h>    /* on Solaris */

int       pthread_atfork(void (*prepare)(void), void (*parent)(void),
                    void (*child)(void));
                  <pthread.h>                                       p. 458
                  Returns: 0 if OK, error number on failure

int       pthread_attr_destroy(pthread_attr_t *attr);
                  <pthread.h>                                       p. 427
                  Returns: 0 if OK, error number on failure

int       pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);
                  <pthread.h>                                       p. 428
                  Returns: 0 if OK, error number on failure

int       pthread_attr_getguardsize(const pthread_attr_t
                                *restrict attr,
                              size_t *restrict guardsize);
                  <pthread.h>                                       p. 430
                  Returns: 0 if OK, error number on failure

int       pthread_attr_getstack(const pthread_attr_t *restrict attr,
                          void **restrict stackaddr,
                          size_t *restrict stacksize);
                  <pthread.h>                                       p. 429
                  Returns: 0 if OK, error number on failure

int       pthread_attr_getstacksize(const pthread_attr_t
                                *restrict attr,
                              size_t *restrict stacksize);
                  <pthread.h>                                       p. 430
                  Returns: 0 if OK, error number on failure

int       pthread_attr_init(pthread_attr_t *attr);
                  <pthread.h>                                       p. 427
                  Returns: 0 if OK, error number on failure

int       pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);
                  <pthread.h>                                       p. 428
                  detachstate: PTHREAD_CREATE_DETACHED,
                               PTHREAD_CREATE_JOINABLE
                  Returns: 0 if OK, error number on failure
```

```
int     pthread_attr_setguardsize(pthread_attr_t *attr,
                                  size_t guardsize);
```
<pthread.h> p. 430
Returns: 0 if OK, error number on failure

```
int     pthread_attr_setstack(const pthread_attr_t *attr,
                              void *stackaddr, size_t *stacksize);
```
<pthread.h> p. 429
Returns: 0 if OK, error number on failure

```
int     pthread_attr_setstacksize(pthread_attr_t *attr,
                                  size_t stacksize);
```
<pthread.h> p. 430
Returns: 0 if OK, error number on failure

```
int     pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
```
<pthread.h> p. 441
Returns: 0 if OK, error number on failure

```
int     pthread_barrierattr_getpshared(const pthread_barrierattr_t
                                         *restrict attr,
                                       int *restrict pshared);
```
<pthread.h> p. 441
Returns: 0 if OK, error number on failure

```
int     pthread_barrierattr_init(pthread_barrierattr_t *attr);
```
<pthread.h> p. 441
Returns: 0 if OK, error number on failure

```
int     pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
                                       int pshared);
```
<pthread.h> p. 441
pshared: PTHREAD_PROCESS_PRIVATE, PTHREAD_PROCESS_SHARED
Returns: 0 if OK, error number on failure

```
int     pthread_barrier_destroy(pthread_barrier_t *barrier);
```
<pthread.h> p. 418
Returns: 0 if OK, error number on failure

```
int     pthread_barrier_init(pthread_barrier_t *restrict barrier,
                             const pthread_barrierattr_t *
                               restrict attr,
                             unsigned int count);
```
<pthread.h> p. 418
Returns: 0 if OK, error number on failure

```
int       pthread_barrier_wait(pthread_barrier_t *barrier);
```
<pthread.h> p. 419

Returns: 0 or PTHREAD_BARRIER_SERIAL_THREAD if OK, error number on failure

```
int       pthread_cancel(pthread_t tid);
```
<pthread.h> p. 393

Returns: 0 if OK, error number on failure

```
void      pthread_cleanup_pop(int execute);
```
<pthread.h> p. 394

```
void      pthread_cleanup_push(void (*rtn)(void *), void *arg);
```
<pthread.h> p. 394

```
int       pthread_condattr_destroy(pthread_condattr_t *attr);
```
<pthread.h> p. 440

Returns: 0 if OK, error number on failure

```
int       pthread_condattr_getclock(const pthread_condattr_t
                                    *restrict attr,
                                  clockid_t *restrict clock_id);
```
<pthread.h> p. 441

Returns: 0 if OK, error number on failure

```
int       pthread_condattr_getpshared(const pthread_condattr_t
                                      *restrict attr,
                                    int *restrict pshared);
```
<pthread.h> p. 440

Returns: 0 if OK, error number on failure

```
int       pthread_condattr_init(pthread_condattr_t *attr);
```
<pthread.h> p. 440

Returns: 0 if OK, error number on failure

```
int       pthread_condattr_setclock(pthread_condattr_t *attr,
                                    clockid_t clock_id);
```
<pthread.h> p. 441

Returns: 0 if OK, error number on failure

```
int       pthread_condattr_setpshared(pthread_condattr_t *attr,
                                      int pshared);
```
<pthread.h> p. 440

pshared: PTHREAD_PROCESS_PRIVATE, PTHREAD_PROCESS_SHARED

Returns: 0 if OK, error number on failure

```
int     pthread_cond_broadcast(pthread_cond_t *cond);
```
<pthread.h> p. 415
Returns: 0 if OK, error number on failure

```
int     pthread_cond_destroy(pthread_cond_t *cond);
```
<pthread.h> p. 414
Returns: 0 if OK, error number on failure

```
int     pthread_cond_init(pthread_cond_t *restrict cond,
                  const pthread_condattr_t *restrict attr);
```
<pthread.h> p. 414
Returns: 0 if OK, error number on failure

```
int     pthread_cond_signal(pthread_cond_t *cond);
```
<pthread.h> p. 415
Returns: 0 if OK, error number on failure

```
int     pthread_cond_timedwait(pthread_cond_t *restrict cond,
                       pthread_mutex_t *restrict mutex,
                       const struct timespec
                         *restrict timeout);
```
<pthread.h> p. 414
Returns: 0 if OK, error number on failure

```
int     pthread_cond_wait(pthread_cond_t *restrict cond,
                  pthread_mutex_t *restrict mutex);
```
<pthread.h> p. 414
Returns: 0 if OK, error number on failure

```
int     pthread_create(pthread_t *restrict tidp,
                 const pthread_attr_t *restrict attr,
                 void *(*start_rtn)(void *),
                 void *restrict arg);
```
<pthread.h> p. 385
Returns: 0 if OK, error number on failure

```
int     pthread_detach(pthread_t tid);
```
<pthread.h> p. 397
Returns: 0 if OK, error number on failure

```
int     pthread_equal(pthread_t tid1, pthread_t tid2);
```
<pthread.h> p. 385
Returns: nonzero if equal, 0 otherwise

```
void    pthread_exit(void *rval_ptr);
```
<pthread.h> p. 389

```
void     *pthread_getspecific(pthread_key_t key);
```
<pthread.h> p. 449
Returns: thread-specific data value or NULL if no value has been associated with the key

```
int      pthread_join(pthread_t thread, void **rval_ptr);
```
<pthread.h> p. 389
Returns: 0 if OK, error number on failure

```
int      pthread_key_create(pthread_key_t *keyp,
                    void (*destructor)(void *));
```
<pthread.h> p. 447
Returns: 0 if OK, error number on failure

```
int      pthread_key_delete(pthread_key_t key);
```
<pthread.h> p. 448
Returns: 0 if OK, error number on failure

```
int      pthread_kill(pthread_t thread, int signo);
```
<signal.h> p. 455
Returns: 0 if OK, error number on failure

```
int      pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```
<pthread.h> p. 431
Returns: 0 if OK, error number on failure

```
int      pthread_mutexattr_getpshared(const pthread_mutexattr_t
                                *restrict attr,
                              int *restrict pshared);
```
<pthread.h> p. 431
Returns: 0 if OK, error number on failure

```
int      pthread_mutexattr_getrobust(const pthread_mutexattr_t
                                 *restrict attr,
                               int *restrict robust);
```
<pthread.h> p. 432
Returns: 0 if OK, error number on failure

```
int      pthread_mutexattr_gettype(const pthread_mutexattr_t
                                *restrict attr,
                              int *restrict type);
```
<pthread.h> p. 434
Returns: 0 if OK, error number on failure

```
int      pthread_mutexattr_init(pthread_mutexattr_t *attr);
```
<pthread.h> p. 431
Returns: 0 if OK, error number on failure

```
int     pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
                                     int pshared);
```
<pthread.h> p. 431

pshared: PTHREAD_PROCESS_PRIVATE,
PTHREAD_PROCESS_SHARED

Returns: 0 if OK, error number on failure

```
int     pthread_mutexattr_setrobust(pthread_mutexattr_t *attr,
                                    int robust);
```
<pthread.h> p. 432

robust: PTHREAD_MUTEX_ROBUST,
PTHREAD_MUTEX_STALLED

Returns: 0 if OK, error number on failure

```
int     pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```
<pthread.h> p. 434

type: PTHREAD_MUTEX_NORMAL,
PTHREAD_MUTEX_ERRORCHECK,
PTHREAD_MUTEX_RECURSIVE,
PTHREAD_MUTEX_DEFAULT

Returns: 0 if OK, error number on failure

```
int     pthread_mutex_consistent(pthread_mutex_t * mutex);
```
<pthread.h> p. 433

Returns: 0 if OK, error number on failure

```
int     pthread_mutex_destroy(pthread_mutex_t *mutex);
```
<pthread.h> p. 400

Returns: 0 if OK, error number on failure

```
int     pthread_mutex_init(pthread_mutex_t *restrict mutex,
                           const pthread_mutexattr_t *restrict attr);
```
<pthread.h> p. 400

Returns: 0 if OK, error number on failure

```
int     pthread_mutex_lock(pthread_mutex_t *mutex);
```
<pthread.h> p. 400

Returns: 0 if OK, error number on failure

```
int     pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
                                const struct timespec *restrict tsptr);
```
<pthread.h> p. 407
<time.h>

Returns: 0 if OK, error number on failure

```
int     pthread_mutex_trylock(pthread_mutex_t *mutex);
```
<pthread.h> p. 400

Returns: 0 if OK, error number on failure

```
int     pthread_mutex_unlock(pthread_mutex_t *mutex);
```
<pthread.h> p. 400
Returns: 0 if OK, error number on failure

```
int     pthread_once(pthread_once_t *initflag, void (*initfn)(void));
```
<pthread.h> p. 448
`pthread_once_t initflag = PTHREAD_ONCE_INIT;`
Returns: 0 if OK, error number on failure

```
int     pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```
<pthread.h> p. 439
Returns: 0 if OK, error number on failure

```
int     pthread_rwlockattr_getpshared(const pthread_rwlockattr_t
                                      *restrict attr,
                                    int *restrict pshared);
```
<pthread.h> p. 440
Returns: 0 if OK, error number on failure

```
int     pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```
<pthread.h> p. 439
Returns: 0 if OK, error number on failure

```
int     pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
                                      int pshared);
```
<pthread.h> p. 440
pshared: `PTHREAD_PROCESS_PRIVATE, PTHREAD_PROCESS_SHARED`
Returns: 0 if OK, error number on failure

```
int     pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```
<pthread.h> p. 409
Returns: 0 if OK, error number on failure

```
int     pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                            const pthread_rwlockattr_t
                              *restrict attr);
```
<pthread.h> p. 409
Returns: 0 if OK, error number on failure

```
int     pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```
<pthread.h> p. 410
Returns: 0 if OK, error number on failure

```
int       pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
                                     const struct timespec
                                       *restrict tsptr);
```
<pthread.h> p. 413
<time.h>
Returns: 0 if OK, error number on failure

```
int       pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
                                     const struct timespec
                                       *restrict tsptr);
```
<pthread.h> p. 413
<time.h>
Returns: 0 if OK, error number on failure

```
int       pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```
<pthread.h> p. 410
Returns: 0 if OK, error number on failure

```
int       pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```
<pthread.h> p. 410
Returns: 0 if OK, error number on failure

```
int       pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```
<pthread.h> p. 410
Returns: 0 if OK, error number on failure

```
int       pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```
<pthread.h> p. 410
Returns: 0 if OK, error number on failure

```
pthread_t pthread_self(void);
```
<pthread.h> p. 385
Returns: thread ID of the calling thread

```
int       pthread_setcancelstate(int state, int *oldstate);
```
<pthread.h> p. 451
state: PTHREAD_CANCEL_ENABLE,
PTHREAD_CANCEL_DISABLE
Returns: 0 if OK, error number on failure

```
int       pthread_setcanceltype(int type, int *oldtype);
```
<pthread.h> p. 453
type: PTHREAD_CANCEL_DEFERRED,
PTHREAD_CANCEL_ASYNCHRONOUS
Returns: 0 if OK, error number on failure

```
int       pthread_setspecific(pthread_key_t key, const void *value);
                    <pthread.h>                                          p. 449
                    Returns: 0 if OK, error number on failure

int       pthread_sigmask(int how, const sigset_t *restrict set,
                        sigset_t *restrict oset);
                    <signal.h>                                           p. 454
                    how: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
                    Returns: 0 if OK, error number on failure

int       pthread_spin_destroy(pthread_spinlock_t *lock);
                    <pthread.h>                                          p. 417
                    Returns: 0 if OK, error number on failure

int       pthread_spin_init(pthread_spinlock_t *lock, int pshared);
                    <pthread.h>                                          p. 417
                    pshared: PTHREAD_PROCESS_PRIVATE,
                             PTHREAD_PROCESS_SHARED
                    Returns: 0 if OK, error number on failure

int       pthread_spin_lock(pthread_spinlock_t *lock);
                    <pthread.h>                                          p. 418
                    Returns: 0 if OK, error number on failure

int       pthread_spin_trylock(pthread_spinlock_t *lock);
                    <pthread.h>                                          p. 418
                    Returns: 0 if OK, error number on failure

int       pthread_spin_unlock(pthread_spinlock_t *lock);
                    <pthread.h>                                          p. 418
                    Returns: 0 if OK, error number on failure

void      pthread_testcancel(void);
                    <pthread.h>                                          p. 453

char     *ptsname(int fd);
                    <stdlib.h>                                           p. 723
                    Returns: pointer to name of PTY slave if OK, NULL on error

int       putc(int c, FILE *fp);
                    <stdio.h>                                            p. 152
                    Returns: c if OK, EOF on error

int       putchar(int c);
                    <stdio.h>                                            p. 152
                    Returns: c if OK, EOF on error
```

```
int       putchar_unlocked(int c);
                    <stdio.h>                                   p. 444
                    Returns: c if OK, EOF on error

int       putc_unlocked(int c, FILE *fp);
                    <stdio.h>                                   p. 444
                    Returns: c if OK, EOF on error

int       putenv(char *str);
                    <stdlib.h>                                  p. 212
                    Returns: 0 if OK, nonzero on error

int       puts(const char *str);
                    <stdio.h>                                   p. 153
                    Returns: non-negative value if OK, EOF on error

ssize_t   pwrite(int fd, const void *buf, size_t nbytes, off_t offset);
                    <unistd.h>                                  p. 78
                    Returns: number of bytes written if OK, –1 on error

int       raise(int signo);
                    <signal.h>                                  p. 337
                    Returns: 0 if OK, nonzero on error

ssize_t   read(int fd, void *buf, size_t nbytes);
                    <unistd.h>                                  p. 71
                    Returns: number of bytes read if OK, 0 if end of file,
                             –1 on error

struct
dirent    *readdir(DIR *dp);
                    <dirent.h>                                  p. 130
                    Returns: pointer if OK, NULL at end of directory or error

ssize_t   readlink(const char *restrict path, char *restrict buf,
                   size_t bufsize);
                    <unistd.h>                                  p. 123
                    Returns: number of bytes read if OK, –1 on error

ssize_t   readlinkat(int fd, const char* restrict path,
                     char *restrict buf, size_t bufsize);
                    <unistd.h>                                  p. 123
                    Returns: number of bytes read if OK, –1 on error

ssize_t   readv(int fd, const struct iovec *iov, int iovcnt);
                    <sys/uio.h>                                 p. 521
                    Returns: number of bytes read if OK, 0 if end of file,
                             –1 on error
```

```
void     *realloc(void *ptr, size_t newsize);
                    <stdlib.h>                                          p. 207
                    Returns: non-null pointer if OK, NULL on error

ssize_t  recv(int sockfd, void *buf, size_t nbytes, int flags);
                    <sys/socket.h>                                      p. 612
                    flags: MSG_PEEK, MSG_OOB, MSG_WAITALL,
                           MSG_CMSG_CLOEXEC (Linux 3.2.0),
                           MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0,
                                         Solaris 10),
                           MSG_ERRQUEUE (Linux 3.2.0),
                           MSG_TRUNC (Linux 3.2.0)
                    Returns: length of message in bytes, 0 if no messages are
                             available and peer has done an orderly shutdown,
                             or –1 on error

ssize_t  recvfrom(int sockfd, void *restrict buf, size_t len, int flags,
                  struct sockaddr *restrict addr,
                  socklen_t *restrict addrlen);
                    <sys/socket.h>                                      p. 613
                    flags: MSG_PEEK, MSG_OOB, MSG_WAITALL
                           MSG_CMSG_CLOEXEC (Linux 3.2.0),
                           MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0,
                                         Solaris 10),
                           MSG_ERRQUEUE (Linux 3.2.0),
                           MSG_TRUNC (Linux 3.2.0)
                    Returns: length of message in bytes, 0 if no messages are
                             available and peer has done an orderly shutdown,
                             or –1 on error

ssize_t  recvmsg(int sockfd, struct msghdr *msg, int flags);
                    <sys/socket.h>                                      p. 613
                    flags: MSG_PEEK, MSG_OOB, MSG_WAITALL
                           MSG_CMSG_CLOEXEC (Linux 3.2.0),
                           MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0,
                                         Solaris 10),
                           MSG_ERRQUEUE (Linux 3.2.0),
                           MSG_TRUNC (Linux 3.2.0)
                    Returns: length of message in bytes, 0 if no messages are
                             available and peer has done an orderly shutdown,
                             or –1 on error

int      remove(const char *path);
                    <stdio.h>                                           p. 119
                    Returns: 0 if OK, –1 on error

int      rename(const char *oldname, const char *newname);
                    <stdio.h>                                           p. 119
                    Returns: 0 if OK, –1 on error
```

```
int       renameat(int oldfd, const char *oldname, int newfd,
                   const char *newname);
                      <stdio.h>                                    p. 119
                      Returns: 0 if OK, –1 on error

void      rewind(FILE *fp);
                      <stdio.h>                                    p. 158

void      rewinddir(DIR *dp);
                      <dirent.h>                                   p. 130

int       rmdir(const char *path);
                      <unistd.h>                                   p. 130
                      Returns: 0 if OK, –1 on error

int       scanf(const char *restrict format, ...);
                      <stdio.h>                                    p. 162
                      Returns: number of input items assigned, EOF if input error or
                               end of file before any conversion

void      seekdir(DIR *dp, long loc);
                      <dirent.h>                                   p. 130

int       select(int maxfdp1, fd_set *restrict readfds,
                 fd_set *restrict writefds, fd_set *restrict exceptfds,
                 struct timeval *restrict tvptr);
                      <sys/select.h>                               p. 502
                      Returns: count of ready descriptors, 0 on timeout, –1 on error

int       sem_close(sem_t *sem);
                      <semaphore.h>                                p. 580
                      Returns: 0 if OK, –1 on error

int       semctl(int semid, int semnum, int cmd, ...
                 /* union semun arg */ );
                      <sys/sem.h>                                  p. 567
                      cmd: IPC_STAT, IPC_SET, IPC_RMID, GETPID,
                           GETNCNT, GETZCNT, GETVAL, SETVAL,
                           GETALL, SETALL
                      Returns: (depends on command), –1 on error

int       sem_destroy(sem_t *sem);
                      <semaphore.h>                                p. 582
                      Returns: 0 if OK, –1 on error
```

```
int       semget(key_t key, int nsems, int flag);
                  <sys/sem.h>                                          p. 567
                  flag: IPC_CREAT, IPC_EXCL
                  Returns: semaphore ID if OK, –1 on error

int       sem_getvalue(sem_t *restrict sem, int *restrict valp);
                  <semaphore.h>                                        p. 582
                  Returns: 0 if OK, –1 on error

int       sem_init(sem_t *sem, int pshared, unsigned int value);
                  <semaphore.h>                                        p. 582
                  Returns: 0 if OK, –1 on error

int       semop(int semid, struct sembuf semoparray[], size_t nops);
                  <sys/sem.h>                                          p. 568
                  Returns: 0 if OK, –1 on error

sem_t    *sem_open(const char *name, int oflag, ... /* mode_t mode,
                    unsigned int value */ );
                  <semaphore.h>                                        p. 579
                  flag: IPC_CREAT, IPC_EXCL
                  Returns: pointer to semaphore if OK, SEM_FAILED on error

int       sem_post(sem_t *sem);
                  <semaphore.h>                                        p. 582
                  Returns: 0 if OK, –1 on error

int       sem_timedwait(sem_t *restrict sem,
                    const struct timespec *restrict tsptr);
                  <semaphore.h>                                        p. 581
                  <time.h>
                  Returns: 0 if OK, –1 on error

int       sem_trywait(sem_t *sem);
                  <semaphore.h>
                  Returns: 0 if OK, –1 on error

int       sem_unlink(const char *name);
                  <semaphore.h>                                        p. 580
                  Returns: 0 if OK, –1 on error

int       sem_wait(sem_t *sem);
                  <semaphore.h>                                        p. 581
                  Returns: 0 if OK, –1 on error
```

```
ssize_t   send(int sockfd, const void *buf, size_t nbytes, int flags);
                    <sys/socket.h>                                          p. 610
                    flags: MSG_EOR, MSG_OOB, MSG_NOSIGNAL
                           MSG_CONFIRM (Linux 3.2.0),
                           MSG_DONTROUTE (FreeBSD 8.0, Linux 3.2.0,
                                          Mac OS X 10.6.8, Solaris 10),
                           MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0,
                                         Mac OS X 10.6.8, Solaris 10),
                           MSG_EOF (FreeBSD 8.0, Mac OS X 10.6.8),
                           MSG_MORE (Linux 3.2.0)
                    Returns: number of bytes sent if OK, –1 on error

ssize_t   sendmsg(int sockfd, const struct msghdr *msg, int flags);
                    <sys/socket.h>                                          p. 611
                    flags: MSG_EOR, MSG_OOB, MSG_NOSIGNAL
                           MSG_CONFIRM (Linux 3.2.0),
                           MSG_DONTROUTE (FreeBSD 8.0, Linux 3.2.0,
                                          Mac OS X 10.6.8, Solaris 10),
                           MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0,
                                         Mac OS X 10.6.8, Solaris 10),
                           MSG_EOF (FreeBSD 8.0, Mac OS X 10.6.8),
                           MSG_MORE (Linux 3.2.0)
                    Returns: number of bytes sent if OK, –1 on error

ssize_t   sendto(int sockfd, const void *buf, size_t nbytes, int flags,
                 const struct sockaddr *destaddr, socklen_t destlen);
                    <sys/socket.h>                                          p. 610
                    flags: MSG_EOR, MSG_OOB, MSG_NOSIGNAL
                           MSG_CONFIRM (Linux 3.2.0),
                           MSG_DONTROUTE (FreeBSD 8.0, Linux 3.2.0,
                                          Mac OS X 10.6.8, Solaris 10),
                           MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0,
                                         Mac OS X 10.6.8, Solaris 10),
                           MSG_EOF (FreeBSD 8.0, Mac OS X 10.6.8),
                           MSG_MORE (Linux 3.2.0)
                    Returns: number of bytes sent if OK, –1 on error

void      setbuf(FILE *restrict fp, char *restrict buf);
                    <stdio.h>                                               p. 146

int       setegid(gid_t gid);
                    <unistd.h>                                              p. 258
                    Returns: 0 if OK, –1 on error

int       setenv(const char *name, const char *value, int rewrite);
                    <stdlib.h>                                              p. 212
                    Returns: 0 if OK, –1 on error
```

```
int       seteuid(uid_t uid);
                  <unistd.h>                                          p. 258
                  Returns: 0 if OK, –1 on error

int       setgid(gid_t gid);
                  <unistd.h>                                          p. 256
                  Returns: 0 if OK, –1 on error

void      setgrent(void);
                  <grp.h>                                             p. 183

int       setgroups(int ngroups, const gid_t grouplist[]);
                  <grp.h>    /* Linux */                              p. 184
                  <unistd.h> /* FreeBSD, Mac OS X, and Solaris */
                  Returns: 0 if OK, –1 on error

void      sethostent(int stayopen);
                  <netdb.h>                                           p. 597

int       setjmp(jmp_buf env);
                  <setjmp.h>                                          p. 215
                  Returns: 0 if called directly, nonzero if returning from
                           a call to longjmp

int       setlogmask(int maskpri);
                  <syslog.h>                                          p. 470
                  Returns: previous log priority mask value

void      setnetent(int stayopen);
                  <netdb.h>                                           p. 598

int       setpgid(pid_t pid, pid_t pgid);
                  <unistd.h>                                          p. 294
                  Returns: 0 if OK, –1 on error

int       setpriority(int which, id_t who, int value);
                  <sys/resource.h>                                    p. 277
                  which: PRIO_PROCESS, PRIO_PGRP, PRIO_USER
                  Returns: 0 if OK, –1 on error

void      setprotoent(int stayopen);
                  <netdb.h>                                           p. 598

void      setpwent(void);
                  <pwd.h>                                             p. 180
```

```
int       setregid(gid_t rgid, gid_t egid);
                   <unistd.h>                                          p. 257
                   Returns: 0 if OK, –1 on error

int       setreuid(uid_t ruid, uid_t euid);
                   <unistd.h>                                          p. 257
                   Returns: 0 if OK, –1 on error

int       setrlimit(int resource, const struct rlimit *rlptr);
                   <sys/resource.h>                                    p. 220
                   resource: RLIMIT_CORE, RLIMIT_CPU,
                             RLIMIT_DATA, RLIMIT_FSIZE,
                             RLIMIT_NOFILE, RLIMIT_STACK,
                             RLIMIT_AS (FreeBSD 8.0, Linux 3.2.0,
                                        Solaris 10),
                             RLIMIT_MEMLOCK (FreeBSD 8.0, Linux 3.2.0,
                                             Mac OS X 10.6.8),
                             RLIMIT_MSGQUEUE (Linux 3.2.0),
                             RLIMIT_NICE (Linux 3.2.0),
                             RLIMIT_NPROC (FreeBSD 8.0, Linux 3.2.0,
                                           Mac OS X 10.6.8),
                             RLIMIT_NPTS (FreeBSD 8.0),
                             RLIMIT_RSS (FreeBSD 8.0, Linux 3.2.0,
                                         Mac OS X 10.6.8),
                             RLIMIT_SBSIZE (FreeBSD 8.0),
                             RLIMIT_SIGPENDING (Linux 3.2.0),
                             RLIMIT_SWAP (FreeBSD 8.0),
                             RLIMIT_VMEM (Solaris 10)
                   Returns: 0 if OK, –1 on error

void      setservent(int stayopen);
                   <netdb.h>                                           p. 599

pid_t     setsid(void);
                   <unistd.h>                                          p. 295
                   Returns: process group ID if OK, –1 on error

int       setsockopt(int sockfd, int level, int option, const void *val,
                     socklen_t len);
                   <sys/socket.h>                                      p. 624
                   Returns: 0 if OK, –1 on error

void      setspent(void);
                   <shadow.h>                                          p. 182
                   Platforms: Linux 3.2.0, Solaris 10

int       setuid(uid_t uid);
                   <unistd.h>                                          p. 256
                   Returns: 0 if OK, –1 on error
```

```
int       setvbuf(FILE *restrict fp, char *restrict buf, int mode,
                  size_t size);
                       <stdio.h>                                          p. 146
                       mode: _IOFBF, _IOLBF, _IONBF
                       Returns: 0 if OK, nonzero on error

void     *shmat(int shmid, const void *addr, int flag);
                       <sys/shm.h>                                        p. 574
                       flag: SHM_RND, SHM_RDONLY
                       Returns: pointer to shared memory segment if OK, –1 on error

int       shmctl(int shmid, int cmd, struct shmid_ds *buf);
                       <sys/shm.h>                                        p. 573
                       cmd: IPC_STAT, IPC_SET, IPC_RMID,
                            SHM_LOCK (Linux 3.2.0, Solaris 10),
                            SHM_UNLOCK (Linux 3.2.0, Solaris 10)
                       Returns: 0 if OK, –1 on error

int       shmdt(const void *addr);
                       <sys/shm.h>                                        p. 574
                       Returns: 0 if OK, –1 on error

int       shmget(key_t key, size_t size, int flag);
                       <sys/shm.h>                                        p. 572
                       flag: IPC_CREAT, IPC_EXCL
                       Returns: non-negative shared memory ID if OK, –1 on error

int       shutdown(int sockfd, int how);
                       <sys/socket.h>                                     p. 592
                       how: SHUT_RD, SHUT_WR, SHUT_RDWR
                       Returns: 0 if OK, –1 on error

int       sig2str(int signo, char *str);
                       <signal.h>                                         p. 380
                       Returns: 0 if OK, –1 on error
                       Platforms: Solaris 10

int       sigaction(int signo, const struct sigaction *restrict act,
                    struct sigaction *restrict oact);
                       <signal.h>                                         p. 350
                       Returns: 0 if OK, –1 on error

int       sigaddset(sigset_t *set, int signo);
                       <signal.h>                                         p. 344
                       Returns: 0 if OK, –1 on error
```

```
int       sigdelset(sigset_t *set, int signo);
                    <signal.h>                                          p. 344
                    Returns: 0 if OK, –1 on error

int       sigemptyset(sigset_t *set);
                    <signal.h>                                          p. 344
                    Returns: 0 if OK, –1 on error

int       sigfillset(sigset_t *set);
                    <signal.h>                                          p. 344
                    Returns: 0 if OK, –1 on error

int       sigismember(const sigset_t *set, int signo);
                    <signal.h>                                          p. 344
                    Returns: 1 if true, 0 if false, –1 on error

void      siglongjmp(sigjmp_buf env, int val);
                    <setjmp.h>                                          p. 356
                    This function never returns

void    (*signal(int signo, void (*func)(int)))(int);
                    <signal.h>                                          p. 323
                    Returns: previous disposition of signal if OK,
                          SIG_ERR on error

int       sigpending(sigset_t *set);
                    <signal.h>                                          p. 347
                    Returns: 0 if OK, –1 on error

int       sigprocmask(int how, const sigset_t *restrict set,
                    sigset_t *restrict oset);
                    <signal.h>                                          p. 346
                    how: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
                    Returns: 0 if OK, –1 on error

int       sigqueue(pid_t pid, int signo, const union sigval value)
                    <signal.h>                                          p. 376
                    Returns: 0 if OK, –1 on error

int       sigsetjmp(sigjmp_buf env, int savemask);
                    <setjmp.h>                                          p. 356
                    Returns: 0 if called directly, nonzero if returning from a call
                          to siglongjmp
```

```
int       sigsuspend(const sigset_t *sigmask);
                     <signal.h>                                    p. 359
                     Returns: –1 with errno set to EINTR

int       sigwait(const sigset_t *restrict set, int *restrict signop);
                     <signal.h>                                    p. 454
                     Returns: 0 if OK, error number on failure

unsigned
int       sleep(unsigned int seconds);
                     <unistd.h>                                    p. 373
                     Returns: 0 or number of unslept seconds

int       snprintf(char *restrict buf, size_t n,
                   const char *restrict format, ...);
                     <stdio.h>                                     p. 159
                     Returns: number of characters that would have been stored
                              in array if buffer was large enough, negative value
                              if encoding error

int       sockatmark(int sockfd);
                     <sys/socket.h>                                p. 626
                     Returns: 1 if at mark, 0 if not at mark, –1 on error

int       socket(int domain, int type, int protocol);
                     <sys/socket.h>                                p. 590
                     type: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET
                     Returns: file (socket) descriptor if OK, –1 on error

int       socketpair(int domain, int type, int protocol, int sockfd[2]);
                     <sys/socket.h>                                p. 630
                     type: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET
                     Returns: 0 if OK, –1 on error

int       sprintf(char *restrict buf, const char *restrict format, ...);
                     <stdio.h>                                     p. 159
                     Returns: number of characters stored in array if OK, negative
                              value if encoding error

int       sscanf(const char *restrict buf,
                 const char *restrict format, ...);
                     <stdio.h>                                     p. 162
                     Returns: number of input items assigned, EOF if input error or
                              end of file before any conversion
```

```
int       stat(const char *restrict path, struct stat *restrict buf);
                    <sys/stat.h>                                        p. 93
                    Returns: 0 if OK, –1 on error

int       str2sig(const char *str, int *signop);
                    <signal.h>                                          p. 380
                    Returns: 0 if OK, –1 on error
                    Platforms: Solaris 10

char     *strerror(int errnum);
                    <string.h>                                          p. 15
                    Returns: pointer to message string

size_t    strftime(char *restrict buf, size_t maxsize,
                   const char *restrict format,
                   const struct tm *restrict tmptr);
                    <time.h>                                            p. 192
                    Returns: number of characters stored in array if room,
                          0 otherwise

size_t    strftime_l(char *restrict buf, size_t maxsize,
                     const char *restrict format,
                     const struct tm *restrict tmptr, locale_t locale);
                    <time.h>                                            p. 192
                    Returns: number of characters stored in array if room,
                          0 otherwise

char     *strptime(const char *restrict buf, const char *restrict format,
                   struct tm *restrict tmptr);
                    <time.h>                                            p. 195
                    Returns: pointer to one character past last character parsed,
                          NULL otherwise

char     *strsignal(int signo);
                    <string.h>                                          p. 380
                    Returns: a pointer to a string describing the signal

int       symlink(const char *actualpath, const char *sympath);
                    <unistd.h>                                          p. 123
                    Returns: 0 if OK, –1 on error

int       symlinkat(const char *actualpath, int fd, const char *sympath);
                    <unistd.h>                                          p. 123
                    Returns: 0 if OK, –1 on error

void      sync(void);
                    <unistd.h>                                          p. 81
```

```
long      sysconf(int name);
                  <unistd.h>                                          p. 42
                  name: _SC_ARG_MAX, _SC_ASYNCHRONOUS_IO,
                        _SC_ATEXIT_MAX, _SC_BARRIERS,
                        _SC_CHILD_MAX, _SC_CLK_TCK,
                        _SC_CLOCK_SELECTION, _SC_COLL_WEIGHTS_MAX,
                        _SC_DELAYTIMER_MAX, _SC_HOST_NAME_MAX,
                        _SC_IOV_MAX, _SC_JOB_CONTROL,
                        _SC_LINE_MAX, _SC_LOGIN_NAME_MAX,
                        _SC_MAPPED_FILED, _SC_MEMORY_PROTECTION,
                        _SC_NGROUPS_MAX,_SC_OPEN_MAX,
                        _SC_PAGESIZE, _SC_PAGE_SIZE,
                        _SC_READER_WRITER_LOCKS,
                        _SC_REALTIME_SIGNALS, _SC_RE_DUP_MAX,
                        _SC_RTSIG_MAX, _SC_SAVED_IDS,
                        _SC_SEMAPHORES, _SC_SEM_NSEMS_MAX,
                        _SC_SEM_VALUE_MAX, _SC_SHELL,
                        _SC_SIGQUEUE_MAX, _SC_SPIN_LOCKS,
                        _SC_STREAM_MAX, _SC_SYMLOOP_MAX,
                        _SC_THREAD_SAFE_FUNCTIONS,
                        _SC_THREADS, _SC_TIMER_MAX,
                        _SC_TIMERS, _SC_TTY_NAME_MAX,
                        _SC_TZNAME_MAX, _SC_VERSION,
                        _SC_XOPEN_CRYPT, _SC_XOPEN_REALTIME,
                        _SC_XOPEN_REALTIME_THREADS, _SC_XOPEN_SHM
                        _SC_XOPEN_VERSION
                  Returns: corresponding value if OK, –1 on error

void      syslog(int priority, char *format, ...);
                  <syslog.h>                                          p. 470

int       system(const char *cmdstring);
                  <stdlib.h>                                          p. 265
                  Returns: termination status of shell

int       tcdrain(int fd);
                  <termios.h>                                         p. 693
                  Returns: 0 if OK, –1 on error

int       tcflow(int fd, int action);
                  <termios.h>                                         p. 693
                  action: TCOOFF, TCOON, TCIOFF, TCION
                  Returns: 0 if OK, –1 on error

int       tcflush(int fd, int queue);
                  <termios.h>                                         p. 693
                  queue: TCIFLUSH, TCOFLUSH, TCIOFLUSH
                  Returns: 0 if OK, –1 on error
```

```
int       tcgetattr(int fd, struct termios *termptr);
```
<termios.h> p. 683
Returns: 0 if OK, –1 on error

```
pid_t     tcgetpgrp(int fd);
```
<unistd.h> p. 298
Returns: process group ID of foreground process group if OK, –1 on error

```
pid_t     tcgetsid(int fd);
```
<termios.h> p. 299
Returns: session leader's process group ID if OK, –1 on error

```
int       tcsendbreak(int fd, int duration);
```
<termios.h> p. 693
Returns: 0 if OK, –1 on error

```
int       tcsetattr(int fd, int opt, const struct termios *termptr);
```
<termios.h> p. 683
opt: TCSANOW, TCSADRAIN, TCSAFLUSH
Returns: 0 if OK, –1 on error

```
int       tcsetpgrp(int fd, pid_t pgrpid);
```
<unistd.h> p. 298
Returns: 0 if OK, –1 on error

```
long      telldir(DIR *dp);
```
<dirent.h> p. 130
Returns: current location in directory associated with *dp*

```
time_t    time(time_t *calptr);
```
<time.h> p. 189
Returns: value of time if OK, –1 on error

```
clock_t   times(struct tms *buf);
```
<sys/times.h> p. 280
Returns: elapsed wall clock time in clock ticks if OK, –1 on error

```
FILE     *tmpfile(void);
```
<stdio.h> p. 167
Returns: file pointer if OK, NULL on error

```
char     *tmpnam(char *ptr);
```
<stdio.h> p. 167
Returns: pointer to unique pathname, NULL on error

```
int       truncate(const char *path, off_t length);
                    <unistd.h>                                          p. 112
                    Returns: 0 if OK, –1 on error

char     *ttyname(int fd);
                    <unistd.h>                                          p. 695
                    Returns: pointer to pathname of terminal, NULL on error

mode_t    umask(mode_t cmask);
                    <sys/stat.h>                                        p. 104
                    Returns: previous file mode creation mask

int       uname(struct utsname *name);
                    <sys/utsname.h>                                     p. 187
                    Returns: non-negative value if OK, –1 on error

int       ungetc(int c, FILE *fp);
                    <stdio.h>                                           p. 151
                    Returns: c if OK, EOF on error

int       unlink(const char *path);
                    <unistd.h>                                          p. 117
                    Returns: 0 if OK, –1 on error

int       unlinkat(int fd, const char *path, int flag);
                    <unistd.h>                                          p. 117
                    flag: AT_REMOVEDIR
                    Returns: 0 if OK, –1 on error

int       unlockpt(int fd);
                    <stdlib.h>                                          p. 723
                    Returns: 0 on success, –1 on error

int       unsetenv(const char *name);
                    <stdlib.h>                                          p. 212
                    Returns: 0 if OK, –1 on error

int       utimensat(int fd, const char *path,
                    const struct timespec times[2], int flag);
                    <sys/stat.h>                                        p. 126
                    flag: AT_SYMLINK_NOFOLLOW
                    Returns: 0 if OK, –1 on error

int       utimes(const char *path, const struct timeval times[2]);
                    <sys/time.h>                                        p. 127
                    Returns: 0 if OK, –1 on error
```

```
int     vdprintf(int fd, const char *restrict format, va_list arg);
```
<stdarg.h> p. 161
<stdio.h>
Returns: number of characters output if OK, negative value if output error

```
int     vfprintf(FILE *restrict fp, const char *restrict format,
             va_list arg);
```
<stdarg.h> p. 161
<stdio.h>
Returns: number of characters output if OK, negative value if output error

```
int     vfscanf(FILE *restrict fp, const char *restrict format,
            va_list arg);
```
<stdarg.h> p. 163
<stdio.h>
Returns: number of input items assigned, EOF if input error or end of file before any conversion

```
int     vprintf(const char *restrict format, va_list arg);
```
<stdarg.h> p. 161
<stdio.h>
Returns: number of characters output if OK, negative value if output error

```
int     vscanf(const char *restrict format, va_list arg);
```
<stdarg.h> p. 163
<stdio.h>
Returns: number of input items assigned, EOF if input error or end of file before any conversion

```
int     vsnprintf(char *restrict buf, size_t n,
              const char *restrict format, va_list arg);
```
<stdarg.h> p. 161
<stdio.h>
Returns: number of characters that would have been stored in array if buffer was large enough, negative value if encoding error

```
int     vsprintf(char *restrict buf, const char *restrict format,
             va_list arg);
```
<stdarg.h> p. 161
<stdio.h>
Returns: number of characters stored in array if OK, negative value if encoding error

```
int       vsscanf(const char *restrict buf, const char *restrict format,
                  va_list arg);
```
<stdarg.h> p. 163
<stdio.h>
Returns: number of input items assigned, EOF if input error or end of file before any conversion

```
void      vsyslog(int priority, const char *format, va_list arg);
```
<syslog.h> p. 472
<stdarg.h>
Platforms: FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, Solaris 10

```
pid_t     wait(int *statloc);
```
<sys/wait.h> p. 238
Returns: process ID if OK, 0, or –1 on error

```
int       waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```
<sys/wait.h> p. 244
idtype: P_PID, P_PGID, P_ALL
options: WCONTINUED, WEXITED, WNOHANG, WNOWAIT, WSTOPPED
Returns: 0 if OK, –1 on error
Platforms: Linux 3.2.0, Solaris 10

```
pid_t     waitpid(pid_t pid, int *statloc, int options);
```
<sys/wait.h> p. 238
options: WCONTINUED, WNOHANG, WUNTRACED
Returns: process ID if OK, 0, or –1 on error

```
pid_t     wait3(int *statloc, int options, struct rusage *rusage);
```
<sys/types.h> p. 245
<sys/wait.h>
<sys/time.h>
<sys/resource.h>
options: WNOHANG, WUNTRACED
Returns: process ID if OK, 0, or –1 on error
Platforms: FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, Solaris 10

```
pid_t     wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```
<sys/types.h> p. 245
<sys/wait.h>
<sys/time.h>
<sys/resource.h>
options: WNOHANG, WUNTRACED
Returns: process ID if OK, 0, or –1 on error
Platforms: FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, Solaris 10

```
ssize_t   write(int fd, const void *buf, size_t nbytes);
                        <unistd.h>                                    p. 72
                        Returns: number of bytes written if OK, –1 on error

ssize_t   writev(int fd, const struct iovec *iov, int iovcnt);
                        <sys/uio.h>                                   p. 521
                        Returns: number of bytes written if OK, –1 on error
```

This page intentionally left blank

Appendix B

Miscellaneous Source Code

B.1 Our Header File

Most programs in the text include the header `apue.h`, shown in Figure B.1. It defines constants (such as `MAXLINE`) and prototypes for our own functions.

Most programs need to include the following headers: `<stdio.h>`, `<stdlib.h>` (for the `exit` function prototype), and `<unistd.h>` (for all the standard UNIX function prototypes). So our header automatically includes these system headers, along with `<string.h>`. This also reduces the size of all the program listings in the text.

```
/*
 * Our own header, to be included before all standard system headers.
 */
#ifndef _APUE_H
#define _APUE_H

#define _POSIX_C_SOURCE 200809L

#if defined(SOLARIS)        /* Solaris 10 */
#define _XOPEN_SOURCE 600
#else
#define _XOPEN_SOURCE 700
#endif

#include <sys/types.h>      /* some systems still require this */
#include <sys/stat.h>
#include <sys/termios.h>    /* for winsize */
```

```
#if defined(MACOS) || !defined(TIOCGWINSZ)
#include <sys/ioctl.h>
#endif

#include <stdio.h>      /* for convenience */
#include <stdlib.h>     /* for convenience */
#include <stddef.h>     /* for offsetof */
#include <string.h>     /* for convenience */
#include <unistd.h>     /* for convenience */
#include <signal.h>     /* for SIG_ERR */

#define MAXLINE 4096            /* max line length */

/*
 * Default file access permissions for new files.
 */
#define FILE_MODE   (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

/*
 * Default permissions for new directories.
 */
#define DIR_MODE    (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)

typedef void    Sigfunc(int);   /* for signal handlers */

#define min(a,b)    ((a) < (b) ? (a) : (b))
#define max(a,b)    ((a) > (b) ? (a) : (b))

/*
 * Prototypes for our own functions.
 */
char    *path_alloc(size_t *);              /* Figure 2.16 */
long     open_max(void);                    /* Figure 2.17 */

int      set_cloexec(int);                  /* Figure 13.9 */
void     clr_fl(int, int);
void     set_fl(int, int);                  /* Figure 3.12 */

void     pr_exit(int);                      /* Figure 8.5 */

void     pr_mask(const char *);             /* Figure 10.14 */
Sigfunc *signal_intr(int, Sigfunc *);       /* Figure 10.19 */

void     daemonize(const char *);           /* Figure 13.1 */

void     sleep_us(unsigned int);            /* Exercise 14.5 */
ssize_t  readn(int, void *, size_t);        /* Figure 14.24 */
ssize_t  writen(int, const void *, size_t); /* Figure 14.24 */

int      fd_pipe(int *);                    /* Figure 17.2 */
int      recv_fd(int, ssize_t (*func)(int,
                const void *, size_t));     /* Figure 17.14 */
```

```
int      send_fd(int, int);                 /* Figure 17.13 */
int      send_err(int, int,
                  const char *);            /* Figure 17.12 */
int      serv_listen(const char *);         /* Figure 17.8 */
int      serv_accept(int, uid_t *);         /* Figure 17.9 */
int      cli_conn(const char *);            /* Figure 17.10 */
int      buf_args(char *, int (*func)(int,
                  char **));                /* Figure 17.23 */

int      tty_cbreak(int);                   /* Figure 18.20 */
int      tty_raw(int);                      /* Figure 18.20 */
int      tty_reset(int);                    /* Figure 18.20 */
void     tty_atexit(void);                  /* Figure 18.20 */
struct termios  *tty_termios(void);         /* Figure 18.20 */

int      ptym_open(char *, int);            /* Figure 19.9 */
int      ptys_open(char *);                 /* Figure 19.9 */
#ifdef  TIOCGWINSZ
pid_t    pty_fork(int *, char *, int, const struct termios *,
                  const struct winsize *);  /* Figure 19.10 */
#endif

int     lock_reg(int, int, int, off_t, int, off_t); /* Figure 14.5 */

#define read_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))

pid_t   lock_test(int, int, off_t, int, off_t);     /* Figure 14.6 */

#define is_read_lockable(fd, offset, whence, len) \
            (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
            (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)

void    err_msg(const char *, ...);         /* Appendix B */
void    err_dump(const char *, ...) __attribute__((noreturn));
void    err_quit(const char *, ...) __attribute__((noreturn));
void    err_cont(int, const char *, ...);
void    err_exit(int, const char *, ...) __attribute__((noreturn));
void    err_ret(const char *, ...);
void    err_sys(const char *, ...) __attribute__((noreturn));

void    log_msg(const char *, ...);         /* Appendix B */
```

```
void    log_open(const char *, int, int);
void    log_quit(const char *, ...) __attribute__((noreturn));
void    log_ret(const char *, ...);
void    log_sys(const char *, ...) __attribute__((noreturn));
void    log_exit(int, const char *, ...) __attribute__((noreturn));

void    TELL_WAIT(void);        /* parent/child from Section 8.9 */
void    TELL_PARENT(pid_t);
void    TELL_CHILD(pid_t);
void    WAIT_PARENT(void);
void    WAIT_CHILD(void);

#endif  /* _APUE_H */
```

Figure B.1 Our header: `apue.h`

The reasons we include our header before all the normal system headers are to allow us to define anything that might be required by headers before they are included, to control the order in which header files are included, and to allow us to redefine anything that needs to be fixed up to hide the differences between systems.

B.2 Standard Error Routines

Two sets of error functions are used in most of the examples throughout the text to handle error conditions. One set begins with `err_` and outputs an error message to standard error. The other set begins with `log_` and is intended for daemon processes (Chapter 13) that probably have no controlling terminal.

The reason for defining our own error functions is to let us write our error handling with a single line of C code, as in

```
if (error condition)
        err_dump(printf format with any number of arguments);
```

instead of

```
if (error condition) {
        char  buf[200];

        sprintf(buf, printf format with any number of arguments);
        perror(buf);
        abort();
}
```

Our error functions use the variable-length argument list facility from ISO C. See Section 7.3 of Kernighan and Ritchie [1988] for additional details. Be aware that this ISO C facility differs from the `varargs` facility provided by earlier systems (such as SVR3 and 4.3BSD). The names of the macros are the same, but the arguments to some of the macros have changed.

Figure B.2 summarizes the differences between the various error functions.

Function	Adds string from `strerror`?	Parameter to `strerror`	Terminate?
`err_dump`	yes	`errno`	`abort();`
`err_exit`	yes	explicit parameter	`exit(1);`
`err_msg`	no		`return;`
`err_quit`	no		`exit(1);`
`err_ret`	yes	`errno`	`return;`
`err_sys`	yes	`errno`	`exit(1);`
`err_cont`	yes	explicit parameter	`return;`
`log_msg`	no		`return;`
`log_quit`	no		`exit(2);`
`log_ret`	yes	`errno`	`return;`
`log_sys`	yes	`errno`	`exit(2);`
`log_exit`	yes	explicit parameter	`exit(2);`

Figure B.2 Our standard error functions

Figure B.3 shows the error functions that output to standard error.

```
#include "apue.h"
#include <errno.h>      /* for definition of errno */
#include <stdarg.h>     /* ISO C variable aruments */

static void err_doit(int, int, const char *, va_list);

/*
 * Nonfatal error related to a system call.
 * Print a message and return.
 */
void
err_ret(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error related to a system call.
 * Print a message and terminate.
 */
void
err_sys(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
```

```
    va_end(ap);
    exit(1);
}

/*
 * Nonfatal error unrelated to a system call.
 * Error code passed as explict parameter.
 * Print a message and return.
 */
void
err_cont(int error, const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, error, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error unrelated to a system call.
 * Error code passed as explict parameter.
 * Print a message and terminate.
 */
void
err_exit(int error, const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, error, fmt, ap);
    va_end(ap);
    exit(1);
}

/*
 * Fatal error related to a system call.
 * Print a message, dump core, and terminate.
 */
void
err_dump(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
    abort();        /* dump core and terminate */
    exit(1);        /* shouldn't get here */
}

/*
 * Nonfatal error unrelated to a system call.
```

```
 * Print a message and return.
 */
void
err_msg(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(0, 0, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error unrelated to a system call.
 * Print a message and terminate.
 */
void
err_quit(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(0, 0, fmt, ap);
    va_end(ap);
    exit(1);
}

/*
 * Print a message and return to caller.
 * Caller specifies "errnoflag".
 */
static void
err_doit(int errnoflag, int error, const char *fmt, va_list ap)
{
    char    buf[MAXLINE];

    vsnprintf(buf, MAXLINE-1, fmt, ap);
    if (errnoflag)
        snprintf(buf+strlen(buf), MAXLINE-strlen(buf)-1, ": %s",
          strerror(error));
    strcat(buf, "\n");
    fflush(stdout);     /* in case stdout and stderr are the same */
    fputs(buf, stderr);
    fflush(NULL);       /* flushes all stdio output streams */
}
```

Figure B.3 Error functions that output to standard error

Figure B.4 shows the `log_XXX` error functions. These require the caller to define the variable `log_to_stderr` and set it nonzero if the process is not running as a daemon. In this case, the error messages are sent to standard error. If the `log_to_stderr` flag is 0, the `syslog` facility (Section 13.4) is used.

```
/*
 * Error routines for programs that can run as a daemon.
 */

#include "apue.h"
#include <errno.h>      /* for definition of errno */
#include <stdarg.h>     /* ISO C variable arguments */
#include <syslog.h>

static void log_doit(int, int, int, const char *, va_list ap);

/*
 * Caller must define and set this: nonzero if
 * interactive, zero if daemon
 */
extern int  log_to_stderr;

/*
 * Initialize syslog(), if running as daemon.
 */
void
log_open(const char *ident, int option, int facility)
{
    if (log_to_stderr == 0)
        openlog(ident, option, facility);
}

/*
 * Nonfatal error related to a system call.
 * Print a message with the system's errno value and return.
 */
void
log_ret(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(1, errno, LOG_ERR, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error related to a system call.
 * Print a message and terminate.
 */
void
log_sys(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(1, errno, LOG_ERR, fmt, ap);
```

```
    va_end(ap);
    exit(2);
}

/*
 * Nonfatal error unrelated to a system call.
 * Print a message and return.
 */
void
log_msg(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(0, 0, LOG_ERR, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error unrelated to a system call.
 * Print a message and terminate.
 */
void
log_quit(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(0, 0, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}

/*
 * Fatal error related to a system call.
 * Error number passed as an explicit parameter.
 * Print a message and terminate.
 */
void
log_exit(int error, const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(1, error, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}

/*
 * Print a message and return to caller.
 * Caller specifies "errnoflag" and "priority".
 */
```

```
static void
log_doit(int errnoflag, int error, int priority, const char *fmt,
         va_list ap)
{
    char    buf[MAXLINE];

    vsnprintf(buf, MAXLINE-1, fmt, ap);
    if (errnoflag)
        snprintf(buf+strlen(buf), MAXLINE-strlen(buf)-1, ": %s",
          strerror(error));
    strcat(buf, "\n");
    if (log_to_stderr) {
        fflush(stdout);
        fputs(buf, stderr);
        fflush(stderr);
    } else {
        syslog(priority, "%s", buf);
    }
}
```

Figure B.4 Error functions for daemons

Appendix C

Solutions to Selected Exercises

Chapter 1

1.1 For this exercise, we use the following two arguments for the `ls`(1) command: `-i` prints the i-node number of the file or directory (we say more about i-nodes in Section 4.14), and `-d` prints information about a directory instead of information on all the files in the directory.

Execute the following:

```
$ ls -ldi /etc/. /etc/..          -i says print i-node number
 162561 drwxr-xr-x  66 root    4096 Feb  5 03:59 /etc/./
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /etc/../
$ ls -ldi /. /..                  both . and .. have i-node number 2
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /./
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /../
```

1.2 The UNIX System is a multiprogramming, or multitasking, system. Other processes were running at the time this program was run.

1.3 Since the *msg* argument to `perror` is a pointer, `perror` could modify the string that *msg* points to. The qualifier `const`, however, says that `perror` does not modify what the pointer points to. On the other hand, the error number

argument to `strerror` is an integer, and since C passes all arguments by value, the `strerror` function couldn't modify this value even if it wanted to. (If the handling of function arguments in C is not clear, you should review Section 5.2 of Kernighan and Ritchie [1988].)

1.4 During the year 2038. We can solve the problem by making the `time_t` data type a 64-bit integer. If it is currently a 32-bit integer, applications will have to be recompiled to work properly. But the problem is worse. Some file systems and backup media store times in 32-bit integers. These would need to be updated as well, but we still need to be able to read the old format.

1.5 Approximately 248 days.

Chapter 2

2.1 The following technique is used by FreeBSD. The primitive data types that can appear in multiple headers are defined in the header `<machine/_types.h>`. For example:

```
#ifndef _MACHINE__TYPES_H_
#define _MACHINE__TYPES_H_

typedef int             __int32_t;
typedef unsigned int    __uint32_t;
    .
    .
    .

typedef __uint32_t      __size_t;
    .
    .
    .

#endif  /* _MACHINE__TYPES_H_ */
```

In each of the headers that can define the `size_t` primitive system data type, we have the sequence

```
#ifndef  _SIZE_T_DECLARED
typedef __size_t       size_t;
#define _SIZE_T_DECLARED
#endif
```

This way, the `typedef` for `size_t` is executed only once.

2.3 If `OPEN_MAX` is indeterminate or ridiculously large (i.e., equal to `LONG_MAX`), we can use `getrlimit` to get the per-process maximum for open file descriptors. Since the per-process limit can be modified, we can't cache the value obtained from the previous call (it might have changed). See Figure C.1.

```
#include "apue.h"
#include <limits.h>
#include <sys/resource.h>

#define OPEN_MAX_GUESS  256

long
open_max(void)
{
    long openmax;
    struct rlimit rl;

    if ((openmax = sysconf(_SC_OPEN_MAX)) < 0 ||
      openmax == LONG_MAX) {
        if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
            err_sys("can't get file limit");
        if (rl.rlim_max == RLIM_INFINITY)
            openmax = OPEN_MAX_GUESS;
        else
            openmax = rl.rlim_max;
    }
    return(openmax);
}
```

Figure C.1 Alternative method for identifying the largest possible file descriptor

Chapter 3

3.1 All disk I/O goes through the kernel's block buffers (also called the kernel's buffer cache). The exception to this is I/O on a raw disk device, which we aren't considering. (Some systems also provide a *direct I/O* option to allow applications to bypass the kernel buffers, but we aren't considering this option either.) Chapter 3 of Bach [1986] describes the operation of this buffer cache. Since the data that we `read` or `write` is buffered by the kernel, the term *unbuffered I/O* refers to the lack of automatic buffering in the user process with these two functions. Each `read` or `write` invokes a single system call.

3.3 Each call to `open` gives us a new file table entry. However, since both `open`s reference the same file, both file table entries point to the same v-node table entry. The call to `dup` references the existing file table entry. We show this in Figure C.2. An `F_SETFD` on `fd1` affects only the file descriptor flags for `fd1`, but an `F_SETFL` on `fd1` affects the file table entry that both `fd1` and `fd2` point to.

3.4 If `fd` is 1, then the `dup2(fd, 1)` returns 1 without closing file descriptor 1. (Remember our discussion of this in Section 3.12.) After the three calls to `dup2`, all three descriptors point to the same file table entry. Nothing needs to be closed.

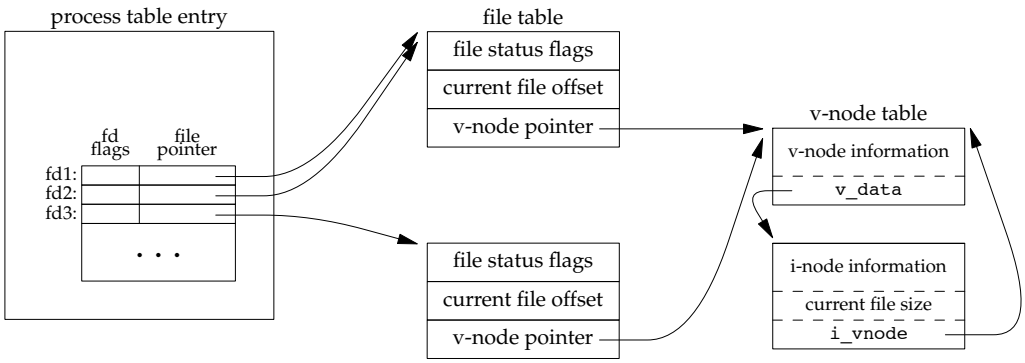


Figure C.2 Result of dup and open

If `fd` is 3, however, after the three calls to `dup2`, four descriptors are pointing to the same file table entry. In this case, we need to close descriptor 3.

3.5 Since the shells process their command line from left to right, the command

```
./a.out > outfile 2>&1
```

first sets standard output to `outfile` and then dups standard output onto descriptor 2 (standard error). The result is that standard output and standard error are set to the same file. Descriptors 1 and 2 both point to the same file table entry. With

```
./a.out 2>&1 > outfile
```

however, the `dup` is executed first, causing descriptor 2 to be the terminal (assuming that the command is run interactively). Then standard output is redirected to the file `outfile`. The result is that descriptor 1 points to the file table entry for `outfile`, and descriptor 2 points to the file table entry for the terminal.

3.6 You can still `lseek` and `read` anywhere in the file, but a `write` automatically resets the file offset to the end of file before the data is written. This makes it impossible to `write` anywhere other than at the end of file.

Chapter 4

4.1 If `stat` is called, it always tries to follow a symbolic link (Figure 4.17), so the program will never print a file type of "symbolic link." For the example shown in the text, where `/dev/cdrom` is a symbolic link to `/dev/sr0`, `stat` reports that `/dev/cdrom` is a block special file, not a symbolic link. If the symbolic link points to a nonexistent file, `stat` returns an error.

4.2 All permissions are turned off:

```
$ umask 777
$ date > temp.foo
$ ls -l temp.foo
----------  1 sar         29 Feb  5 14:06 temp.foo
```

4.3 The following shows what happens when user-read permission is turned off:

```
$ date > foo
$ chmod u-r foo                    turn off user-read permission
$ ls -l foo                        verify the file's permissions
--w-r--r--  1 sar         29 Feb  5 14:21 foo
$ cat foo                          and try to read it
cat: foo: Permission denied
```

4.4 If we try, using either `open` or `creat`, to create a file that already exists, the file's access permission bits are not changed. We can verify this by running the program from Figure 4.9:

```
$ rm foo bar                       delete the files in case they already exist
$ date > foo                       create them with some data
$ date > bar
$ chmod a-r foo bar                turn off all read permissions
$ ls -l foo bar                    verify their permissions
--w-------  1 sar         29 Feb  5 14:25 bar
--w-------  1 sar         29 Feb  5 14:25 foo
$ ./a.out                          run program from Figure 4.9
$ ls -l foo bar                    check permissions and sizes
--w-------  1 sar          0 Feb  5 14:26 bar
--w-------  1 sar          0 Feb  5 14:26 foo
```

Note that the permissions didn't change but that the files were truncated.

4.5 The size of a directory should never be 0, since there should always be entries for dot and dot-dot. The size of a symbolic link is the number of characters in the pathname contained in the symbolic link, and this pathname must always contain at least one character.

4.7 The kernel has a default setting for the file access permission bits when it creates a new `core` file. In this example, it was `rw-r--r--`. This default value may or may not be modified by the `umask` value. The shell also has a default setting for the file access permission bits when it creates a new file for redirection. In this example, it was `rw-rw-rw-`, and this value is always modified by our current `umask`. In this example, our `umask` was 02.

4.8 We can't use `du`, because it requires either the name of the file, as in

```
du tempfile
```

or a directory name, as in

```
du .
```

But when the `unlink` function returns, the directory entry for `tempfile` is gone. The `du .` command just shown would not account for the space still taken by

`tempfile`. We have to use the `df` command in this example to see the actual amount of free space on the file system.

4.9 If the link being removed is not the last link to the file, the file is not removed. In this case, the changed-status time of the file is updated. But if the link being removed is the last link to the file, it makes no sense to update this time, because all the information about the file (the i-node) is removed with the file.

4.10 We recursively call our function `dopath` after opening a directory with `opendir`. Assuming that `opendir` uses a single file descriptor, this means that each time we descend one level, we use another descriptor. (We assume that the descriptor isn't closed until we're finished with a directory and call `closedir`.) This limits the depth of the file system tree that we can traverse to the maximum number of open descriptors for the process. Note that the `nftw` function as specified in the XSI option of the Single UNIX Specification allows the caller to specify the number of descriptors to use, implying that it can close and reuse descriptors.

4.12 The `chroot` function is used by the Internet File Transfer Protocol (FTP) program to aid in security. Users without accounts on a system (termed *anonymous FTP*) are placed in a separate directory, and a `chroot` is done to that directory. This prevents the user from accessing any file outside this new root directory.

In addition, `chroot` can be used to build a copy of a file system hierarchy at a new location and then modify this new copy without changing the original file system. This could be used, for example, to test the installation of new software packages.

Only the superuser can execute `chroot`, and once you change the root of a process, it (and all its descendants) can never get back to the original root.

4.13 First, call `stat` to fetch the three times for the file; then call `utime` to set the desired value. The value that we don't want to change in the call to `utime` should be the corresponding value from `stat`.

4.14 The `finger`(1) command calls `stat` on the mailbox. The last-modification time is the time that mail was last received, and the last-access time is when the mail was last read.

4.15 Both `cpio` and `tar` store only the modification time (`st_mtime`) in the archive. The access time isn't stored, because its value corresponds to the time the archive was created, since the file has to be read to be archived. The `-a` option to `cpio` has it reset the access time of each input file after the file has been read. This way, the creation of the archive doesn't change the access time. (Resetting the access time, however, does modify the changed-status time.) The changed-status time isn't stored in the archive, because we can't set this value on extraction even if it was archived. (The `utimes` function and its related functions, `futimens` and `utimensat`, can change only the access time and the modification time.)

When the archive is read back (extracted), `tar`, by default, restores the modification time to the value in the archive. The `m` option to `tar` tells it to not

restore the modification time from the archive; instead, the modification time is set to the time of extraction. In all cases with `tar`, the access time after extraction will be the time of extraction.

In contrast, `cpio` sets the access time and the modification time to the time of extraction. By default, it doesn't try to set the modification time to the value on the archive. The `-m` option to `cpio` has it set both the access time and the modification time to the value that was archived.

4.16 The kernel has no inherent limit on the depth of a directory tree. Nevertheless, many commands will fail on pathnames that exceed `PATH_MAX`. The program shown in Figure C.3 creates a directory tree that is 1,000 levels deep, with each level being a 45-character name. We are able to create this structure on all platforms; however, we cannot obtain the absolute pathname of the directory at the 1,000th level using `getcwd` on all platforms. On Mac OS X 10.6.8, we can never get `getcwd` to succeed while in the directory at the end of this long path. The program is able to retrieve the pathname on FreeBSD 8.0, Linux 3.2.0, and Solaris 10, but we have to call `realloc` numerous times to obtain a buffer that is large enough. Running this program on Linux 3.2.0 gives us

```
$ ./a.out
getcwd failed, size = 4096: Numerical result out of range
getcwd failed, size = 4196: Numerical result out of range
...                          418 more lines
getcwd failed, size = 45896: Numerical result out of range
getcwd failed, size = 45996: Numerical result out of range
length = 46004
                                the 46,004-byte pathname is printed here
```

We are not able to archive this directory, however, using `cpio`. It complains that many of the filenames are too long. In fact, `cpio` is unable to archive this directory on all four platforms. In contrast, we can archive this directory using `tar` on FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8. However, we are unable to extract the directory hierarchy from the archive on Linux 3.2.0.

4.17 The `/dev` directory has write permissions turned off to prevent a normal user from removing the filenames in the directory. This means that the `unlink` attempt fails.

Chapter 5

5.2 The `fgets` function reads up through and including the next newline *or* until the buffer is full (leaving room, of course, for the terminating null). Also, `fputs` writes everything in the buffer until it hits a null byte; it doesn't care whether a newline is in the buffer. So, if `MAXLINE` is too small, both functions still work; they're just called more often than they would be if the buffer were larger.

If either of these functions removed or added the newline (as `gets` and `puts` do), we would have to ensure that our buffer was big enough for the largest line.

```
#include "apue.h"
#include <fcntl.h>

#define DEPTH    1000           /* directory depth */
#define STARTDIR "/tmp"
#define NAME     "alonglonglonglonglonglonglonglonglongname"
#define MAXSZ    (10*8192)

int
main(void)
{
    int     i;
    size_t  size;
    char    *path;

    if (chdir(STARTDIR) < 0)
        err_sys("chdir error");

    for (i = 0; i < DEPTH; i++) {
        if (mkdir(NAME, DIR_MODE) < 0)
            err_sys("mkdir failed, i = %d", i);
        if (chdir(NAME) < 0)
            err_sys("chdir failed, i = %d", i);
    }

    if (creat("afile", FILE_MODE) < 0)
        err_sys("creat error");

    /*
     * The deep directory is created, with a file at the leaf.
     * Now let's try to obtain its pathname.
     */
    path = path_alloc(&size);
    for ( ; ; ) {
        if (getcwd(path, size) != NULL) {
            break;
        } else {
            err_ret("getcwd failed, size = %ld", (long)size);
            size += 100;
            if (size > MAXSZ)
                err_quit("giving up");
            if ((path = realloc(path, size)) == NULL)
                err_sys("realloc error");
        }
    }
    printf("length = %ld\n%s\n", (long)strlen(path), path);

    exit(0);
}
```

Figure C.3 Create a deep directory tree

5.3 The function call

```
printf("");
```

returns 0, since no characters are output.

5.4 This is a common error. The return value from `getc` and `getchar` is an `int`, not a `char`. `EOF` is often defined to be –1, so if the system uses signed characters, the code normally works. But if the system uses unsigned characters, after the `EOF` returned by `getchar` is stored as an unsigned character, the character's value no longer equals –1, so the loop never terminates. The four platforms described in this book all use signed characters, so the example code works on these platforms.

5.5 Call `fsync` after each call to `fflush`. The argument to `fsync` is obtained with the `fileno` function. Calling `fsync` without calling `fflush` might do nothing if all the data were still in memory buffers.

5.6 Standard input and standard output are both line buffered when a program is run interactively. When `fgets` is called, standard output is flushed automatically.

5.7 An implementation of `fmemopen` for BSD-based systems is shown in Figure C.4.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

/*
 * Our internal structure tracking a memory stream
 */
struct memstream
{
    char    *buf;   /* in-memory buffer */
    size_t  rsize;  /* real size of buffer */
    size_t  vsize;  /* virtual size of buffer */
    size_t  curpos; /* current position in buffer */
    int     flags;  /* see below */
};

/* flags */
#define MS_READ     0x01    /* open for reading */
#define MS_WRITE    0x02    /* open for writing */
#define MS_APPEND   0x04    /* append to stream */
#define MS_TRUNCATE 0x08    /* truncate the stream on open */
#define MS_MYBUF    0x10    /* free buffer on close */

#ifndef MIN
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#endif

static int mstream_read(void *, char *, int);
static int mstream_write(void *, const char *, int);
static fpos_t mstream_seek(void *, fpos_t, int);
static int mstream_close(void *);
```

```
static int type_to_flags(const char *__restrict type);
static off_t find_end(char *buf, size_t len);

FILE *
fmemopen(void *__restrict buf, size_t size,
    const char *__restrict type)
{
    struct memstream *ms;
    FILE *fp;

    if (size == 0) {
        errno = EINVAL;
        return(NULL);
    }
    if ((ms = malloc(sizeof(struct memstream))) == NULL) {
        errno = ENOMEM;
        return(NULL);
    }
    if ((ms->flags = type_to_flags(type)) == 0) {
        errno = EINVAL;
        free(ms);
        return(NULL);
    }
    if (buf == NULL) {
        if ((ms->flags & (MS_READ|MS_WRITE)) !=
          (MS_READ|MS_WRITE)) {
            errno = EINVAL;
            free(ms);
            return(NULL);
        }
        if ((ms->buf = malloc(size)) == NULL) {
            errno = ENOMEM;
            free(ms);
            return(NULL);
        }
        ms->rsize = size;
        ms->flags |= MS_MYBUF;
        ms->curpos = 0;
    } else {
        ms->buf = buf;
        ms->rsize = size;
        if (ms->flags & MS_APPEND)
            ms->curpos = find_end(ms->buf, ms->rsize);
        else
            ms->curpos = 0;
    }
    if (ms->flags & MS_APPEND) {            /* "a" mode */
        ms->vsize = ms->curpos;
    } else if (ms->flags & MS_TRUNCATE) {   /* "w" mode */
        ms->vsize = 0;
    } else {                                /* "r" mode */
```

```
        ms->vsize = size;
    }
    fp = funopen(ms, mstream_read, mstream_write,
      mstream_seek, mstream_close);
    if (fp == NULL) {
        if (ms->flags & MS_MYBUF)
            free(ms->buf);
        free(ms);
    }
    return(fp);
}

static int
type_to_flags(const char *__restrict type)
{
    const char *cp;
    int flags = 0;

    for (cp = type; *cp != 0; cp++) {
        switch (*cp) {
        case 'r':
            if (flags != 0)
                return(0);  /* error */
            flags |= MS_READ;
            break;

        case 'w':
            if (flags != 0)
                return(0);  /* error */
            flags |= MS_WRITE|MS_TRUNCATE;
            break;

        case 'a':
            if (flags != 0)
                return(0);  /* error */
            flags |= MS_APPEND;
            break;

        case '+':
            if (flags == 0)
                return(0);  /* error */
            flags |= MS_READ|MS_WRITE;
            break;

        case 'b':
            if (flags == 0)
                return(0);  /* error */
            break;

        default:
            return(0);      /* error */
        }
    }
```

```
    return(flags);
}

static off_t
find_end(char *buf, size_t len)
{
    off_t off = 0;

    while (off < len) {
        if (buf[off] == 0)
            break;
        off++;
    }
    return(off);
}

static int
mstream_read(void *cookie, char *buf, int len)
{
    int nr;
    struct memstream *ms = cookie;

    if (!(ms->flags & MS_READ)) {
        errno = EBADF;
        return(-1);
    }
    if (ms->curpos >= ms->vsize)
        return(0);

    /* can only read from curpos to vsize */
    nr = MIN(len, ms->vsize - ms->curpos);
    memcpy(buf, ms->buf + ms->curpos, nr);
    ms->curpos += nr;
    return(nr);
}

static int
mstream_write(void *cookie, const char *buf, int len)
{
    int nw, off;
    struct memstream *ms = cookie;

    if (!(ms->flags & (MS_APPEND|MS_WRITE))) {
        errno = EBADF;
        return(-1);
    }
    if (ms->flags & MS_APPEND)
        off = ms->vsize;
    else
        off = ms->curpos;
    nw = MIN(len, ms->rsize - off);
    memcpy(ms->buf + off, buf, nw);
    ms->curpos = off + nw;
```

```
    if (ms->curpos > ms->vsize) {
        ms->vsize = ms->curpos;
        if (((ms->flags & (MS_READ|MS_WRITE)) ==
          (MS_READ|MS_WRITE)) && (ms->vsize < ms->rsize))
            *(ms->buf + ms->vsize) = 0;
    }
    if ((ms->flags & (MS_WRITE|MS_APPEND)) &&
      !(ms->flags & MS_READ)) {
        if (ms->curpos < ms->rsize)
            *(ms->buf + ms->curpos) = 0;
        else
            *(ms->buf + ms->rsize - 1) = 0;
    }
    return(nw);
}

static fpos_t
mstream_seek(void *cookie, fpos_t pos, int whence)
{
    int off;
    struct memstream *ms = cookie;

    switch (whence) {
    case SEEK_SET:
        off = pos;
        break;
    case SEEK_END:
        off = ms->vsize + pos;
        break;
    case SEEK_CUR:
        off = ms->curpos + pos;
        break;
    }
    if (off < 0 || off > ms->vsize) {
        errno = EINVAL;
        return -1;
    }
    ms->curpos = off;
    return(off);
}

static int
mstream_close(void *cookie)
{
    struct memstream *ms = cookie;

    if (ms->flags & MS_MYBUF)
        free(ms->buf);
    free(ms);
    return(0);
}
```

Figure C.4 Implementation of `fmemopen` for BSD systems

Chapter 6

6.1 The functions to access the shadow password file on Linux and Solaris are discussed in Section 6.3. We can't use the value returned in the `pw_passwd` field by the functions described in Section 6.2 to compare an encrypted password, since that field is not the encrypted password. Instead, we need to find the user's entry in the shadow file and use its encrypted password field.

On FreeBSD and Mac OS X, the password file is shadowed automatically. In the `passwd` structure returned by `getpwnam` and `getpwuid` on FreeBSD 8.0, the `pw_passwd` field contains the encrypted password, but only if the caller's effective user ID is 0. On Mac OS X 10.6.8, the encrypted password is not accessible using these interfaces.

6.2 The program in Figure C.5 prints the encrypted password on Linux 3.2.0 and Solaris 10. Unless this program is run with superuser permissions, the call to `getspnam` fails with an error of `EACCES`.

```
#include "apue.h"
#include <shadow.h>

int
main(void)      /* Linux/Solaris version */
{
    struct spwd *ptr;

    if ((ptr = getspnam("sar")) == NULL)
        err_sys("getspnam error");
    printf("sp_pwdp = %s\n", ptr->sp_pwdp == NULL ||
      ptr->sp_pwdp[0] == 0 ?  "(null)" : ptr->sp_pwdp);
    exit(0);
}
```

Figure C.5 Print encrypted password under Linux and Solaris

Under FreeBSD 8.0, the program in Figure C.6 prints the encrypted password if the program is run with superuser permissions. Otherwise, the value returned in `pw_passwd` is an asterisk. On Mac OS X 10.6.8, asterisks are printed regardless of the permissions with which it is run.

```
#include "apue.h"
#include <pwd.h>

int
main(void)      /* FreeBSD/Mac OS X version */
{
    struct passwd   *ptr;

    if ((ptr = getpwnam("sar")) == NULL)
        err_sys("getpwnam error");
```

```
    printf("pw_passwd = %s\n", ptr->pw_passwd == NULL ||
      ptr->pw_passwd[0] == 0 ?  "(null)" : ptr->pw_passwd);
    exit(0);
}
```

Figure C.6 Print encrypted password under FreeBSD and Mac OS X

6.5 The program shown in Figure C.7 prints the date in a format similar to the `date` command.

```
#include "apue.h"
#include <time.h>

int
main(void)
{
    time_t      caltime;
    struct tm   *tm;
    char        line[MAXLINE];

    if ((caltime = time(NULL)) == -1)
        err_sys("time error");
    if ((tm = localtime(&caltime)) == NULL)
        err_sys("localtime error");
    if (strftime(line, MAXLINE, "%a %b %d %X %Z %Y\n", tm) == 0)
        err_sys("strftime error");
    fputs(line, stdout);
    exit(0);
}
```

Figure C.7 Print the time and date in a format similar to `date`(1)

Running this program gives us

```
$ ./a.out                              author's default is US/Eastern
Wed Jul 25 22:58:32 EDT 2012
$ TZ=US/Mountain ./a.out               U.S. Mountain time zone
Wed Jul 25 20:58:32 MDT 2012
$ TZ=Japan ./a.out                     Japan
Thu Jul 26 11:58:32 JST 2012
```

Chapter 7

7.1 It appears that the return value from `printf` (the number of characters output) becomes the return value of `main`. To verify this theory, change the length of the string printed and see if the new length matches the return value. Note that not all systems exhibit this property. Also note that if you enable the ISO C extensions in `gcc`, then the return value is always 0, as required by the standard.

7.2 When the program is run interactively, standard output is usually line buffered, so the actual output occurs when each newline is output. If standard output were directed to a file, however, it would probably be fully buffered, and the actual output wouldn't occur until the standard I/O cleanup is performed.

7.3 On most UNIX systems, there is no way to do this. Copies of `argc` and `argv` are not kept in global variables like `environ` is.

7.4 This provides a way to terminate the process when it tries to dereference a null pointer, a common C programming error.

7.5 The definitions are

```
typedef void    Exitfunc(void);

int atexit(Exitfunc *func);
```

7.6 `calloc` initializes the memory that it allocates to all zero bits. ISO C does not guarantee that this is the same as either a floating-point 0 or a null pointer.

7.7 The heap and the stack aren't allocated until a program is executed by one of the `exec` functions (described in Section 8.10).

7.8 The executable file (`a.out`) contains symbol table information that can be helpful in debugging a `core` file. To remove this information, use the `strip`(1) command. Stripping the two `a.out` files reduces their size to 798,760 and 6,200 bytes.

7.9 When shared libraries are not used, a large portion of the executable file is occupied by the standard I/O library.

7.10 The code is incorrect, since it references the automatic integer `val` through a pointer after the automatic variable is no longer in existence. Automatic variables declared after the left brace that starts a compound statement disappear after the matching right brace.

Chapter 8

8.1 To simulate the behavior of the child closing the standard output when it exits, add the following line before calling `exit` in the child:

```
fclose(stdout);
```

To see the effects of doing this, replace the call to `printf` with the lines

```
i = printf("pid = %ld, glob = %d, var = %d\n",
    (long)getpid(), glob, var);
sprintf(buf, "%d\n", i);
write(STDOUT_FILENO, buf, strlen(buf));
```

You need to define the variables `i` and `buf` also.

This assumes that the standard I/O stream `stdout` is closed when the child calls `exit`, not the file descriptor `STDOUT_FILENO`. Some versions of the standard I/O library close the file descriptor associated with standard output, which would cause the `write` to standard output to also fail. In this case, `dup` standard output to another descriptor, and use this new descriptor for the `write`.

8.2 Consider Figure C.8.

```
#include "apue.h"

static void f1(void), f2(void);

int
main(void)
{
    f1();
    f2();
    _exit(0);
}

static void
f1(void)
{
    pid_t   pid;

    if ((pid = vfork()) < 0)
        err_sys("vfork error");
    /* child and parent both return */
}

static void
f2(void)
{
    char    buf[1000];      /* automatic variables */
    int     i;

    for (i = 0; i < sizeof(buf); i++)
        buf[i] = 0;
}
```

Figure C.8 Incorrect use of `vfork`

When `vfork` is called, the parent's stack pointer points to the stack frame for the `f1` function that calls `vfork`. Figure C.9 shows this. `vfork` causes the child to execute first, and the child returns from `f1`. The child then calls `f2`, and its stack frame overwrites the previous stack frame for `f1`. The child then zeros out the automatic variable `buf`, setting 1,000 bytes of the stack frame to 0. The child returns from `f2` and then calls `_exit`, but the contents of the stack beneath the stack frame for `main` have been changed. The parent then resumes after the call

to `vfork` and does a return from `f1`. The return information is often stored in the stack frame, and that information has probably been modified by the child. After the parent resumes, what happens with this example depends on many implementation features of your UNIX system (where in the stack frame the return information is stored, what information in the stack frame is wiped out when the automatic variables are modified, and so on). The normal result is a `core` file, but your results may differ.

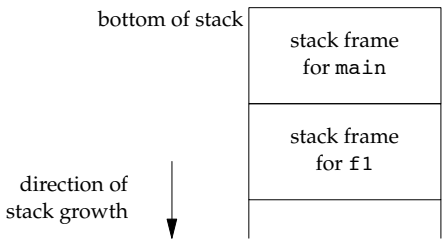


Figure C.9 Stack frames when `vfork` is called

8.4 In Figure 8.13, we have the parent write its output first. When the parent is done, the child writes its output, but we let the parent terminate. Whether the parent terminates or whether the child finishes its output first depends on the kernel's scheduling of the two processes (another race condition). When the parent terminates, the shell starts up the next program, and this next program can interfere with the output from the previous child.

We can prevent this from happening by not letting the parent terminate until the child has also finished its output. Replace the code following the `fork` with the following:

```
else if (pid == 0) {
    WAIT_PARENT();          /* parent goes first */
    charatatime("output from child\n");
    TELL_PARENT(getppid()); /* tell parent we're done */
} else {
    charatatime("output from parent\n");
    TELL_CHILD(pid);        /* tell child we're done */
    WAIT_CHILD();           /* wait for child to finish */
}
```

We won't see this happen if we let the child go first, since the shell doesn't start the next program until the parent terminates.

8.5 The same value (`/home/sar/bin/testinterp`) is printed for `argv[2]`. The reason is that `execlp` ends up calling `execve` with the same *pathname* as when we call `execl` directly. Recall Figure 8.15.

8.6 The program in Figure C.10 creates a zombie.

```
#include "apue.h"

#ifdef SOLARIS
#define PSCMD   "ps -a -o pid,ppid,s,tty,comm"
#else
#define PSCMD   "ps -o pid,ppid,state,tty,command"
#endif

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)      /* child */
        exit(0);

    /* parent */
    sleep(4);
    system(PSCMD);

    exit(0);
}
```

Figure C.10 Create a zombie and look at its status with `ps`

Zombies are usually designated by `ps`(1) with a status of `Z`:

```
$ ./a.out
 PID  PPID S TT       COMMAND
2369  2208 S pts/2    -bash
7230  2369 S pts/2    ./a.out
7231  7230 Z pts/2    [a.out] <defunct>
7232  7230 S pts/2    sh -c ps -o pid,ppid,state,tty,command
7233  7232 R pts/2    ps -o pid,ppid,state,tty,command
```

Chapter 9

9.1 The `init` process learns when a terminal user logs out, because `init` is the parent of the login shell and receives the `SIGCHLD` signal when the login shell terminates.

For a network login, however, `init` is not involved. Instead, the login entries in the `utmp` and `wtmp` files, and their corresponding logout entries, are usually written by the process that handles the login and detects the logout (`telnetd` in our example).

Chapter 10

10.1 The program terminates the first time we send it a signal. The reason is that the `pause` function returns whenever a signal is caught.

10.3 Figure C.11 shows the stack frames.

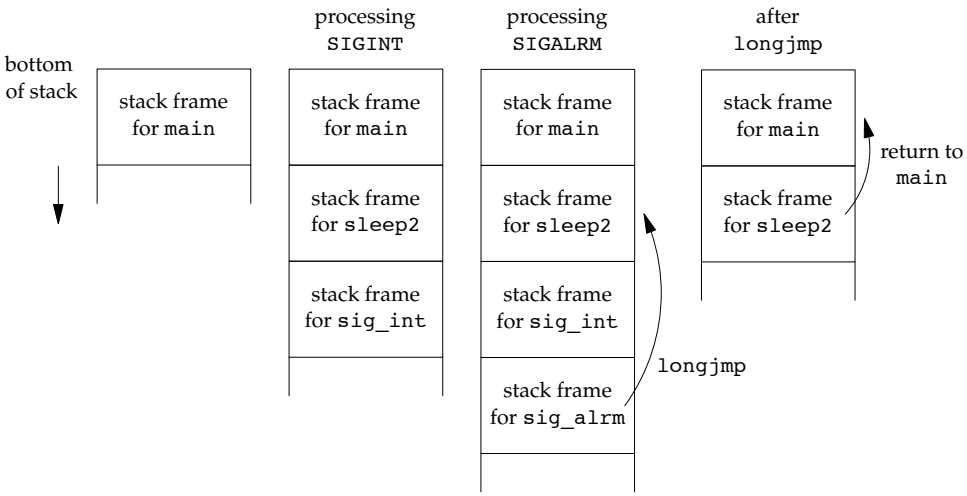


Figure C.11 Stack frames before and after `longjmp`

The `longjmp` from `sig_alrm` back to `sleep2` effectively aborts the call to `sig_int`. From this point, `sleep2` returns to main (recall Figure 10.8).

10.4 We again have a race condition, this time between the first call to `alarm` and the call to `setjmp`. If the process is blocked by the kernel between these two function calls, the alarm goes off, the signal handler is called, and `longjmp` is called. But since `setjmp` was never called, the buffer `env_alrm` is not set. The operation of `longjmp` is undefined if its jump buffer has not been initialized by `setjmp`.

10.5 See "Implementing Software Timers" by Don Libes (*C Users Journal*, vol. 8, no. 11, Nov. 1990) for an example. A copy of this paper is available online at `http://www.kohala.com/start/libes.timers.txt`.

10.7 If we simply called `_exit`, the termination status of the process would not show that it was terminated by the `SIGABRT` signal.

10.8 If the signal was sent by a process owned by some other user, the process has to be set-user-ID to either root or to the owner of the receiving process, or the `kill` attempt won't work. Therefore, the real user ID provides more information to the receiver of the signal.

10.10 On one system used by the author, the value for the number of seconds increased by 1 about every 60−90 minutes. This skew occurs because each call to `sleep` schedules an event for a time in the future, but is not awakened exactly when that event occurs (because of CPU scheduling). In addition, a finite amount of time is required for our process to start running and call `sleep` again.

A program such as the `cron` daemon has to fetch the current time every minute, as well as to set its first sleep period so that it wakes up at the beginning of the next minute. (Convert the current time to the local time and look at the `tm_sec` value.) Every minute, it sets the next sleep period so that it'll wake up at the next minute. Most of the calls will probably be `sleep(60)`, with an occasional `sleep(59)` to resynchronize with the next minute. But if at some point the process takes a long time executing commands or if the system gets heavily loaded and scheduling delays hold up the process, the sleep value can be much less than 60.

10.11 Under Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10, the signal handler for `SIGXFSZ` is never called. But `write` returns a count of 24 as soon as the file's size reaches 1,024 bytes.

When the file's size has reached 1,000 bytes under FreeBSD 8.0, the signal handler is called on the next attempt to write 100 bytes, and the `write` call returns –1 with `errno` set to `EFBIG` ("File too big").

On all four platforms, if we attempt an additional `write` at the current file offset (the end of the file), we will receive `SIGXFSZ` and `write` will fail, returning –1 with `errno` set to `EFBIG`.

10.12 The results depend on the implementation of the standard I/O library: how the `fwrite` function handles an interrupted `write`.

On Linux 3.2.0, for example, when we use the `fwrite` function to write a large buffer, the `fwrite` function calls `write` directly for the same number of bytes. While in the middle of the `write` system call, the alarm fires, but we don't see the signal until the write completes. It appears as if the kernel is blocking the signal while we are in the middle of the `write` system call.

In contrast, on Solaris 10, the `fwrite` function calls `write` in 8 KB increments until the entire amount is written. When the alarm fires, it is caught, interrupting the call to `fwrite`. After we return from the signal handler, we return to the loop inside the `fwrite` function and continue writing in 8 KB increments.

Chapter 11

11.1 A version of the program that allocates the memory dynamically instead of using an automatic variable is shown in Figure C.12.

```
#include "apue.h"
#include <pthread.h>

struct foo {
    int a, b, c, d;
};

void
printfoo(const char *s, const struct foo *fp)
{
    fputs(s, stdout);
    printf("  structure at 0x%lx\n", (unsigned long)fp);
    printf("  foo.a = %d\n", fp->a);
    printf("  foo.b = %d\n", fp->b);
    printf("  foo.c = %d\n", fp->c);
    printf("  foo.d = %d\n", fp->d);
}

void *
thr_fn1(void *arg)
{
    struct foo *fp;

    if ((fp = malloc(sizeof(struct foo))) == NULL)
        err_sys("can't allocate memory");
    fp->a = 1;
    fp->b = 2;
    fp->c = 3;
    fp->d = 4;
    printfoo("thread:\n", fp);
    return((void *)fp);
}

int
main(void)
{
    int err;
    pthread_t tid1;
    struct foo *fp;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_exit(err, "can't create thread 1");
    err = pthread_join(tid1, (void *)&fp);
    if (err != 0)
        err_exit(err, "can't join with thread 1");
    printfoo("parent:\n", fp);
    exit(0);
}
```

Figure C.12 Correct use of thread return value

11.2 To change the thread ID of a pending job, the reader–writer lock must be held in write mode to prevent anyone from searching the list while the ID is being changed. The problem with the way the interfaces are currently defined is that the ID of a job can change between the time that the job is found with `job_find` and the job is removed from the list by calling `job_remove`. This problem can be solved by embedding a reference count and a mutex inside the `job` structure and having `job_find` increment the reference count. The code that changes the ID can then avoid any job in the list that has a nonzero reference count.

11.3 First of all, the list is protected by a reader–writer lock, but the condition variable needs a mutex to protect the condition. Second, the condition each thread should wait to be satisfied is that there is a job for it to process, so we need to create a per-thread data structure to represent this condition. Alternatively, we can embed the mutex and condition variable in the `queue` structure, but this means that all worker threads will wait on the same condition. If there are many worker threads, we can run into a *thundering herd* problem, whereby many threads are awakened without work to do, resulting in a waste of CPU resources and increased lock contention.

11.4 It depends on the circumstances. In general, both can be correct, but each alternative has drawbacks. In the first sequence, the waiting threads will be scheduled to run after we call `pthread_cond_broadcast`. If the program is running on a multiprocessor, some threads will run and immediately block because we are still holding the mutex (recall that `pthread_cond_wait` returns with the mutex held). In the second sequence, a running thread can acquire the mutex between steps 3 and 4, invalidate the condition, and release the mutex. Then, when we call `pthread_cond_broadcast`, the condition will no longer be true, and the threads will run needlessly. This is why the awakened threads must recheck the condition and not assume that it is true merely because `pthread_cond_wait` returned.

Chapter 12

12.1 This is not a multithreading problem, as one might first guess. The standard I/O routines are indeed thread-safe. When we call `fork`, each process gets a copy of the standard I/O data structures. When we run the program with standard output attached to a terminal, the output is line buffered, so every time we print a line, the standard I/O library writes it to our terminal. However, if we redirect the standard output to a file, then the standard output is fully buffered. The output is written when the buffer fills or the process closes the stream. When we `fork` in this example, the buffer contains several printed lines not yet written, so when the parent and the child finally flush their copies of the buffer, the initial duplicate contents are written to the file.

12.3 Theoretically, if we arrange for all signals to be blocked when the signal handler runs, we should be able to make a function async-signal safe. The problem is

that we don't know whether any of the functions we call might unmask a signal that we've blocked, thereby making it possible for the function to be reentered through another signal handler.

12.4 On FreeBSD 8.0, the program drops core. With `gdb`, we are able to see that the program initialization calls pthread functions, which call `getenv` to find the value of the `LIBPTHREAD_SPINLOOPS` and `LIBPTHREAD_YIELDLOOPS` environment variables. However, our thread-safe version of `getenv` calls back into the pthread library while it is in an intermediate, inconsistent state. In addition, the thread initialization functions call `malloc`, which, in turn, call `getenv` to find the value of the `MALLOC_OPTIONS` environment variable.

To get around this problem, we could make the reasonable assumption that program start-up is single threaded, and use a flag to indicate whether the thread initialization had been completed by our version of `getenv`. While this flag is false, our version of `getenv` can operate as the non-reentrant version does (and avoid all calls to pthread functions and `malloc`). Then we could provide a separate initialization function to call `pthread_once`, instead of calling it from inside `getenv`. This requires that the program call our initialization function before calling `getenv`. This solves our problem, because this can't be done until the program start-up initialization completes. After the program calls our initialization function, our version of `getenv` operates in a thread-safe manner.

12.5 We still need `fork` if we want to run a program from within another program (i.e., before calling `exec`).

12.6 Figure C.13 shows a thread-safe `sleep` implementation that uses `select` to delay for the specified amount of time. It is thread-safe because it doesn't use any unprotected global or static data and calls only other thread-safe functions.

12.7 The implementation of a condition variable most likely uses a mutex to protect its internal structure. Because this is an implementation detail and therefore hidden, there is no portable way for us to acquire and release the lock in the fork handlers. Since we can't determine the state of the internal lock in a condition variable after calling `fork`, it is unsafe for us to use the condition variable in the child process.

Chapter 13

13.1 If it calls `chroot`, the process will not be able to open `/dev/log`. The solution is for the daemon to call `openlog` with an *option* of `LOG_NDELAY`, before calling `chroot`. This opens the special device file (the UNIX domain datagram socket), yielding a descriptor that is still valid, even after a call to `chroot`. This scenario is encountered in daemons, such as `ftpd` (the File Transfer Protocol daemon), that specifically call `chroot` for security reasons but still need to call `syslog` to log error conditions.

```
#include <unistd.h>
#include <time.h>
#include <sys/select.h>

unsigned
sleep(unsigned seconds)
{
    int n;
    unsigned slept;
    time_t start, end;
    struct timeval tv;

    tv.tv_sec = seconds;
    tv.tv_usec = 0;
    time(&start);
    n = select(0, NULL, NULL, NULL, &tv);
    if (n == 0)
        return(0);
    time(&end);
    slept = end - start;
    if (slept >= seconds)
        return(0);
    return(seconds - slept);
}
```

Figure C.13 A thread-safe implementation of `sleep`

13.3 Figure C.14 shows a solution.

```
#include "apue.h"

int
main(void)
{
    FILE *fp;
    char *p;

    daemonize("getlog");
    p = getlogin();
    fp = fopen("/tmp/getlog.out", "w");
    if (fp != NULL) {
        if (p == NULL)
            fprintf(fp, "no login name\n");
        else
            fprintf(fp, "login name: %s\n", p);
    }
    exit(0);
}
```

Figure C.14 Call `daemonize` and then obtain login name

The results depend on the platform. Recall that `daemonize` closes all open file descriptors and then reopens the first three to `/dev/null`. This means that the process won't have a controlling terminal, so `getlogin` won't be able to look in the `utmp` file for the process's login entry. Thus, on Linux 3.2.0 and Solaris 10, we find that a daemon has no login name.

Under FreeBSD 8.0 and Mac OS X 10.6.8, however, the login name is maintained in the process table and copied across a `fork`. This means that the process can always get the login name, unless the parent didn't have one to start out (such as `init` when the system is bootstrapped).

Chapter 14

14.1 The test program is shown in Figure C.15.

```
#include "apue.h"
#include <fcntl.h>
#include <errno.h>

void
sigint(int signo)
{
}

int
main(void)
{
    pid_t pid1, pid2, pid3;
    int fd;

    setbuf(stdout, NULL);
    signal_intr(SIGINT, sigint);

    /*
     * Create a file.
     */
    if ((fd = open("lockfile", O_RDWR|O_CREAT, 0666)) < 0)
        err_sys("can't open/create lockfile");

    /*
     * Read-lock the file.
     */
    if ((pid1 = fork()) < 0) {
        err_sys("fork failed");
    } else if (pid1 == 0) { /* child */
        if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
            err_sys("child 1: can't read-lock file");
        printf("child 1: obtained read lock on file\n");
        pause();
        printf("child 1: exit after pause\n");
```

```
    exit(0);
} else {        /* parent */
    sleep(2);
}

/*
 * Parent continues ... read-lock the file again.
 */
if ((pid2 = fork()) < 0) {
    err_sys("fork failed");
} else if (pid2 == 0) { /* child */
    if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
        err_sys("child 2: can't read-lock file");
    printf("child 2: obtained read lock on file\n");
    pause();
    printf("child 2: exit after pause\n");
    exit(0);
} else {        /* parent */
    sleep(2);
}

/*
 * Parent continues ... block while trying to write-lock
 * the file.
 */
if ((pid3 = fork()) < 0) {
    err_sys("fork failed");
} else if (pid3 == 0) { /* child */
    if (lock_reg(fd, F_SETLK, F_WRLCK, 0, SEEK_SET, 0) < 0)
        printf("child 3: can't set write lock: %s\n",
          strerror(errno));
    printf("child 3 about to block in write-lock...\n");
    if (lock_reg(fd, F_SETLKW, F_WRLCK, 0, SEEK_SET, 0) < 0)
        err_sys("child 3: can't write-lock file");
    printf("child 3 returned and got write lock????\n");
    pause();
    printf("child 3: exit after pause\n");
    exit(0);
} else {        /* parent */
    sleep(2);
}

/*
 * See if a pending write lock will block the next
 * read-lock attempt.
 */
if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
    printf("parent: can't set read lock: %s\n",
      strerror(errno));
else
    printf("parent: obtained additional read lock while"
```

```
            " write lock is pending\n");
    printf("killing child 1...\n");
    kill(pid1, SIGINT);
    printf("killing child 2...\n");
    kill(pid2, SIGINT);
    printf("killing child 3...\n");
    kill(pid3, SIGINT);
    exit(0);
}
```

Figure C.15 Determine record-locking behavior

On FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8, the behavior is the same: additional readers can starve pending writers. Running the program gives us

```
child 1: obtained read lock on file
child 2: obtained read lock on file
child 3: can't set write lock: Resource temporarily unavailable
child 3 about to block in write-lock...
parent: obtained additional read lock while write lock is pending
killing child 1...
child 1: exit after pause
killing child 2...
child 2: exit after pause
killing child 3...
child 3: can't write-lock file: Interrupted system call
```

On Solaris 10, readers don't starve waiting writers. In this case, the parent is unable to obtain a read lock because there is a process waiting for a write lock.

14.2 Most systems define the `fd_set` data type to be a structure that contains a single member: an array of long integers. One bit in this array corresponds to each descriptor. The four `FD_` macros then manipulate this array of longs, turning specific bits on and off and testing specific bits.

One reason that the data type is defined to be a structure containing an array and not simply an array is to allow variables of type `fd_set` to be assigned to one another with the C assignment statement.

14.3 In the good ol' days, most systems allowed us to define the constant `FD_SETSIZE` before including the header `<sys/select.h>`. For example, we could write

```
#define FD_SETSIZE  2048
#include <sys/select.h>
```

to define the `fd_set` data type to accommodate 2,048 descriptors. Unfortunately, things aren't that simple anymore. To use this technique with contemporary systems, we need to do several things:

1. Before we include any header files, we need to define whatever symbol prevents us from including `<sys/select.h>`. Some systems might protect

the definition of the fd_set type with a separate symbol. We need to define this, too.

For example, on FreeBSD 8.0, we need to define _SYS_SELECT_H_ to prevent the inclusion of <sys/select.h> and we need to define _FD_SET to prevent the inclusion of the definition for the fd_set data type.

2. Sometimes, for compatibility with older applications, <sys/types.h> defines the size of the fd_set, so we need to include it first, then undefine FD_SETSIZE. Note that some systems use __FD_SETSIZE instead.
3. We need to redefine FD_SETSIZE (or__FD_SETSIZE) to the maximum file descriptor number we want to be able to use with select.
4. We need to undefine the symbols we defined in step 1.
5. Finally, we can include <sys/select.h>.

Before we run the program, we need to configure the system to allow us to open as many file descriptors as we might need so that we can actually make use of FD_SETSIZE file descriptors.

14.4 The following table lists the functions that do similar things.

FD_ZERO	sigemptyset
FD_SET	sigaddset
FD_CLR	sigdelset
FD_ISSET	sigismember

There is not an FD_xxx function that corresponds to sigfillset. With signal sets, the pointer to the set is always the first argument, and the signal number is the second argument. With descriptor sets, the descriptor number is the first argument, and the pointer to the set is the next argument.

14.5 Figure C.16 shows an implementation using select.

```
#include "apue.h"
#include <sys/select.h>

void
sleep_us(unsigned int nusecs)
{
    struct timeval  tval;

    tval.tv_sec = nusecs / 1000000;
    tval.tv_usec = nusecs % 1000000;
    select(0, NULL, NULL, NULL, &tval);
}
```

Figure C.16 Implementation of sleep_us using select

Figure C.17 shows an implementation using poll.

```
#include <poll.h>

void
sleep_us(unsigned int nusecs)
{
    struct pollfd   dummy;
    int             timeout;

    if ((timeout = nusecs / 1000) <= 0)
        timeout = 1;
    poll(&dummy, 0, timeout);
}
```

Figure C.17 Implementation of `sleep_us` using `poll`

As the BSD `usleep`(3) manual page states, `usleep` uses the `nanosleep` function, which doesn't interfere with timers set by the calling process.

14.6 No. What we would like to do is have `TELL_WAIT` create a temporary file and use 1 byte for the parent's lock and 1 byte for the child's lock. `WAIT_CHILD` would have the parent wait to obtain a lock on the child's byte, and `TELL_PARENT` would have the child release the lock on the child's byte. The problem, however, is that calling `fork` releases all the locks in the child, so the child can't start off with any locks of its own.

14.7 A solution is shown in Figure C.18.

```
#include "apue.h"
#include <fcntl.h>

int
main(void)
{
    int i, n;
    int fd[2];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    set_fl(fd[1], O_NONBLOCK);

    /* write 1 byte at a time until pipe is full */
    for (n = 0; ; n++) {
        if ((i = write(fd[1], "a", 1)) != 1) {
            printf("write ret %d, ", i);
            break;
        }
    }
    printf("pipe capacity = %d\n", n);
    exit(0);
}
```

Figure C.18 Calculation of pipe capacity using nonblocking writes

The following table shows the values calculated for our four platforms.

Platform	Pipe Capacity (bytes)
FreeBSD 8.0	65,536
Linux 3.2.0	65,536
Mac OS X 10.6.8	16,384
Solaris 10	16,384

These values can differ from the corresponding `PIPE_BUF` values, because `PIPE_BUF` is defined to be the maximum amount of data that can be written to a pipe *atomically*. Here, we calculate the amount of data that a pipe can hold independent of any atomicity constraints.

14.10 Whether the program in Figure 14.27 updates the last-access time for the input file depends on the operating system and the type of file system in which the file resides. On all four platforms, the last-access time is updated when the file resides in the default file system type for the given operating system.

Chapter 15

15.1 If the write end of the pipe is never closed, the reader never sees an end of file. The pager program blocks forever reading from its standard input.

15.2 The parent terminates right after writing the last line to the pipe. The read end of the pipe is automatically closed when the parent terminates. But the parent is probably running ahead of the child by one pipe buffer, since the child (the pager program) is waiting for us to look at a page of output. If we're running a shell, such as the Korn shell, with interactive command-line editing enabled, the shell probably changes the terminal mode when our parent terminates and the shell prints a prompt. This undoubtedly interferes with the pager program, which has also modified the terminal mode. (Most pager programs set the terminal to noncanonical mode when awaiting input to proceed to the next page.)

15.3 The `popen` function returns a file pointer because the shell is executed. But the shell can't execute the nonexistent command, so it prints

```
sh: line 1: ./a.out: No such file or directory
```

on the standard error and terminates with an exit status of 127 (although the value depends on the type of shell). `pclose` returns the termination status of the command as it is returned by `waitpid`.

15.4 When the parent terminates, look at its termination status with the shell. For the Bourne shell, Bourne-again shell, and Korn shell, the command is `echo $?`. The number printed is 128 plus the signal number.

15.5 First add the declaration

```
FILE    *fpin, *fpout;
```

Then use fdopen to associate the pipe descriptors with a standard I/O stream, and set the streams to be line buffered. Do this before the while loop that reads from standard input:

```
if ((fpin = fdopen(fd2[0], "r")) == NULL)
    err_sys("fdopen error");
if ((fpout = fdopen(fd1[1], "w")) == NULL)
    err_sys("fdopen error");
if (setvbuf(fpin, NULL, _IOLBF, 0) < 0)
    err_sys("setvbuf error");
if (setvbuf(fpout, NULL, _IOLBF, 0) < 0)
    err_sys("setvbuf error");
```

The write and read in the while loop are replaced with

```
if (fputs(line, fpout) == EOF)
    err_sys("fputs error to pipe");
if (fgets(line, MAXLINE, fpin) == NULL) {
    err_msg("child closed pipe");
    break;
}
```

15.6 The system function calls wait, and the first child to terminate is the child generated by popen. Since that's not the child that system created, it calls wait again and blocks until the sleep is done. Then system returns. When pclose calls wait, an error is returned, since there are no more children to wait for. Then pclose returns an error.

15.7 Although the details vary by platform (see Figure C.19), select indicates that the descriptor is readable. After all the data has been read, read returns 0 to indicate the end of file. But with poll, the POLLHUP event is returned, and this can happen while there is still data to be read. Once we have read all the data, however, read returns 0 to indicate the end of file. After all the data has been read, the POLLIN event is not returned, even though we need to issue a read to receive the end-of-file notification (the return value of 0).

Operation	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
select on read end of pipe with write end closed	R/W/E	R	R/W	R/W/E
poll on read end of pipe with write end closed	R/HUP	HUP	INV	HUP
select on write end of pipe with read end closed	R/W/E	R/W	R/W	R/W
poll on write end of pipe with read end closed	R/HUP	W/ERR	INV	HUP

Figure C.19 Pipe behavior with select and poll

The conditions shown in Figure C.19 include R (readable), W (writable), E (exception), HUP (hangup), ERR (error), and INV (invalid file descriptor). With an output descriptor that refers to a pipe that has been closed by the reader, select indicates that the descriptor is writable. But when we call write, the SIGPIPE signal is generated. If we either ignore this signal or return from its

signal handler, `write` fails with `errno` set to `EPIPE`. With `poll`, however, the behavior varies by platform.

15.8 Anything written by the child to standard error appears wherever the parent's standard error would appear. To send standard error back to the parent, include the shell redirection `2>&1` in the *cmdstring*.

15.9 The `popen` function `fork`s a child, and the child executes the shell. The shell in turn calls `fork`, and the child of the shell executes the command string. When *cmdstring* terminates, the shell is waiting for this to happen. The shell then exits, which is what the `waitpid` in `pclose` is waiting for.

15.10 The trick is to `open` the FIFO twice: once for reading and once for writing. We never use the descriptor that is opened for writing, but leaving that descriptor open prevents an end of file from being generated when the number of clients goes from 1 to 0. Opening the FIFO twice requires some care, as a nonblocking `open` is required. We have to do a nonblocking, read-only `open` first, followed by a blocking `open` for write-only. (If we tried a nonblocking `open` for write-only first, it would return an error.) We then turn off nonblocking for the read descriptor. Figure C.20 shows the code for this.

```
#include "apue.h"
#include <fcntl.h>

#define FIFO    "temp.fifo"

int
main(void)
{
    int     fdread, fdwrite;

    unlink(FIFO);
    if (mkfifo(FIFO, FILE_MODE) < 0)
        err_sys("mkfifo error");
    if ((fdread = open(FIFO, O_RDONLY | O_NONBLOCK)) < 0)
        err_sys("open error for reading");
    if ((fdwrite = open(FIFO, O_WRONLY)) < 0)
        err_sys("open error for writing");
    clr_fl(fdread, O_NONBLOCK);
    exit(0);
}
```

Figure C.20 Opening a FIFO for reading and writing, without blocking

15.11 Randomly reading a message from an active queue would interfere with the client–server protocol, as either a client request or a server's response would be lost. To read the queue, all that is needed is for the process to know the identifier for the queue and for the queue to allow world-read access.

15.13 We never store actual addresses in a shared memory segment, since it's possible for the server and all the clients to attach the segment at different addresses.

Instead, when a linked list is built in a shared memory segment, the list pointers should be stored as offsets to other objects in the shared memory segment. These offsets are formed by subtracting the start of the shared memory segment from the actual address of the object.

15.14 Figure C.21 shows the relevant events.

Parent `i` set to	Child `i` set to	Shared value set to	`update` returns	Comment
		0		initialized by `mmap`
	1			child runs first, then is blocked
0				parent runs
		1		
			0	then parent is blocked
		2		child resumes
			1	
	3			then child is blocked
2				parent resumes
		3		
			2	then parent is blocked
		4		
			3	
	5			then child is blocked
4				parent resumes

Figure C.21 Alternation between parent and child in Figure 15.33

Chapter 16

16.1 Figure C.22 shows a program that prints the system's byte order.

```
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

int
main(void)
{
    uint32_t        i = 0x04030201;
    unsigned char   *cp = (unsigned char *)&i;

    if (*cp == 1)
        printf("little-endian\n");
    else if (*cp == 4)
        printf("big-endian\n");
    else
        printf("who knows?\n");
    exit(0);
}
```

Figure C.22 Determine byte order on system

16.3 For each endpoint we will be listening on, we need to bind the proper address and record an entry in an `fd_set` structure corresponding to each file descriptor. We will use `select` to wait for connect requests to arrive on multiple endpoints. Recall from Section 16.4 that a passive endpoint will appear to be readable when a connect request arrives on it. When a connect request does arrive, we will accept the request and process it as before.

16.5 In the `main` procedure, we need to arrange to catch `SIGCHLD` by calling our `signal` function (Figure 10.18), which will use `sigaction` to install the handler specifying the restartable system call option. Next, we need to remove the call to `waitpid` from our `serve` function. After `fork`ing the child to service the request, the parent closes the new file descriptor and resumes listening for additional connect requests. Finally, we need a signal handler for `SIGCHLD`, as follows:

```
void
sigchld(int signo)
{
    while (waitpid((pid_t)-1, NULL, WNOHANG) > 0)
        ;
}
```

16.6 To enable asynchronous socket I/O, we need to establish socket ownership using the `F_SETOWN fcntl` command, and then enable asynchronous signaling using the `FIOASYNC ioctl` command. To disable asynchronous socket I/O, we simply need to disable asynchronous signaling. The reason we mix `fcntl` and `ioctl` commands is to find the methods that are most portable. The code is shown in Figure C.23.

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#if defined(BSD) || defined(MACOS) || defined(SOLARIS)
#include <sys/filio.h>
#endif

int
setasync(int sockfd)
{
    int n;

    if (fcntl(sockfd, F_SETOWN, getpid()) < 0)
        return(-1);
    n = 1;
    if (ioctl(sockfd, FIOASYNC, &n) < 0)
        return(-1);
    return(0);
}
```

```
int
clrasync(int sockfd)
{
    int n;

    n = 0;
    if (ioctl(sockfd, FIOASYNC, &n) < 0)
        return(-1);
    return(0);
}
```

Figure C.23 Enable and disable asynchronous socket I/O

Chapter 17

17.1 Regular pipes provide a byte stream interface. To detect message boundaries, we'd have to add a header to each message to indicate the length. But this still involves two extra copy operations: one to write to the pipe and one to read from the pipe. A more efficient approach is to use the pipe only to signal the main thread that a new message is available. We can use a single byte for this purpose. With this approach, we need to move the `mymesg` structure to the `threadinfo` structure and use a mutex and a condition variable to prevent the helper thread from reusing the `mymesg` structure until the main thread is done with it. The solution is shown in Figure C.24.

```
#include "apue.h"
#include <poll.h>
#include <pthread.h>
#include <sys/msg.h>
#include <sys/socket.h>

#define NQ      3       /* number of queues */
#define MAXMSZ  512     /* maximum message size */
#define KEY     0x123   /* key for first message queue */

struct mymesg {
    long        mtype;
    char        mtext[MAXMSZ+1];
};

struct threadinfo {
    int             qid;
    int             fd;
    int             len;
    pthread_mutex_t mutex;
    pthread_cond_t  ready;
    struct mymesg   m;
};
```

```
void *
helper(void *arg)
{
    int                 n;
    struct threadinfo   *tip = arg;

    for(;;) {
        memset(&tip->m, 0, sizeof(struct mymsg));
        if ((n = msgrcv(tip->qid, &tip->m, MAXMSZ, 0,
          MSG_NOERROR)) < 0)
            err_sys("msgrcv error");
        tip->len = n;
        pthread_mutex_lock(&tip->mutex);
        if (write(tip->fd, "a", sizeof(char)) < 0)
            err_sys("write error");
        pthread_cond_wait(&tip->ready, &tip->mutex);
        pthread_mutex_unlock(&tip->mutex);
    }
}

int
main()
{
    char                c;
    int                 i, n, err;
    int                 fd[2];
    int                 qid[NQ];
    struct pollfd       pfd[NQ];
    struct threadinfo   ti[NQ];
    pthread_t           tid[NQ];

    for (i = 0; i < NQ; i++) {
        if ((qid[i] = msgget((KEY+i), IPC_CREAT|0666)) < 0)
            err_sys("msgget error");

        printf("queue ID %d is %d\n", i, qid[i]);

        if (socketpair(AF_UNIX, SOCK_DGRAM, 0, fd) < 0)
            err_sys("socketpair error");
        pfd[i].fd = fd[0];
        pfd[i].events = POLLIN;
        ti[i].qid = qid[i];
        ti[i].fd = fd[1];
        if (pthread_cond_init(&ti[i].ready, NULL) != 0)
            err_sys("pthread_cond_init error");
        if (pthread_mutex_init(&ti[i].mutex, NULL) != 0)
            err_sys("pthread_mutex_init error");
        if ((err = pthread_create(&tid[i], NULL, helper,
          &ti[i])) != 0)
            err_exit(err, "pthread_create error");
    }
```

```
    for (;;) {
        if (poll(pfd, NQ, -1) < 0)
            err_sys("poll error");
        for (i = 0; i < NQ; i++) {
            if (pfd[i].revents & POLLIN) {
                if ((n = read(pfd[i].fd, &c, sizeof(char))) < 0)
                    err_sys("read error");
                ti[i].m.mtext[ti[i].len] = 0;
                printf("queue id %d, message %s\n", qid[i],
                  ti[i].m.mtext);
                pthread_mutex_lock(&ti[i].mutex);
                pthread_cond_signal(&ti[i].ready);
                pthread_mutex_unlock(&ti[i].mutex);
            }
        }
    }

    exit(0);
}
```

Figure C.24 Poll for XSI messages using pipes

17.3 A *declaration* specifies the attributes (such as the data type) of a set of identifiers. If the declaration also causes storage to be allocated, it is called a *definition*.

In the `opend.h` header, we declare the three global variables with the `extern` storage class. These declarations do not cause storage to be allocated for the variables. In the `main.c` file, we define the three global variables. Sometimes, we'll also initialize a global variable when we define it, but we typically let the C default apply.

17.5 Both `select` and `poll` return the number of ready descriptors as the value of the function. The loop that goes through the `client` array can terminate when the number of ready descriptors has been processed.

17.6 The first problem with the proposed solution is that there is a race between the call to `stat` and the call to `unlink` where the file can change. The second problem is that if the name is a symbolic link pointing to the UNIX domain socket file, then `stat` will report that the name is a socket (recall that the `stat` function follows symbolic links), but when we call `unlink`, we will actually remove the symbolic link instead of the socket file. To solve this problem, we should use `lstat` instead of `stat`, but this doesn't solve the first problem.

17.7 The first option is to send both file descriptors in one control message. Each file descriptor is stored in adjacent memory locations. The following code shows this:

```
struct msghdr msg;
struct cmsghdr *cmptr;
int *ip;
```

```
if ((cmptr = calloc(1, CMSG_LEN(2*sizeof(int)))) == NULL)
      err_sys("calloc error");
msg.msg_control = cmptr;
msg.msg_controllen = CMSG_LEN(2*sizeof(int));
/* continue initializing msghdr... */
cmptr->cmsg_len = CMSG_LEN(2*sizeof(int));
cmptr->cmsg_level = SOL_SOCKET;
cmptr->cmsg_type = SCM_RIGHTS;
ip = (int *)CMSG_DATA(cmptr);
*ip++ = fd1;
*ip = fd2;
```

This approach works on all four platforms covered in this book. The second option is to pack two separate `cmsghdr` structures into a single message:

```
struct msghdr msg;
struct cmsghdr *cmptr;

if ((cmptr = calloc(1, 2*CMSG_LEN(sizeof(int)))) == NULL)
      err_sys("calloc error");
msg.msg_control = cmptr;
msg.msg_controllen = 2*CMSG_LEN(sizeof(int));
/* continue initializing msghdr... */
cmptr->cmsg_len = CMSG_LEN(sizeof(int));
cmptr->cmsg_level = SOL_SOCKET;
cmptr->cmsg_type = SCM_RIGHTS;
*(int *)CMSG_DATA(cmptr) = fd1;
cmptr = CMPTR_NXTHDR(&msg, cmptr);
cmptr->cmsg_len = CMSG_LEN(sizeof(int));
cmptr->cmsg_level = SOL_SOCKET;
cmptr->cmsg_type = SCM_RIGHTS;
*(int *)CMSG_DATA(cmptr) = fd2;
```

Unlike the first approach, this method works only on FreeBSD 8.0.

Chapter 18

18.1 Note that you have to terminate the `reset` command with a line feed character, not a return, since the terminal is in noncanonical mode.

18.2 It builds a table for each of the 128 characters and sets the high-order bit (the parity bit) according to the user's specification. It then uses 8-bit I/O, handling the parity generation itself.

18.3 If you happen to be on a windowing terminal, you don't need to log in twice. You can do this experiment between two separate windows. Under Solaris, execute `stty -a` with standard input redirected from the terminal window running `vi`. This shows that `vi` sets MIN to 1 and TIME to 1. A call to `read` will wait for at least one character to be typed, but after that character is entered, `read` waits only one-tenth of a second for additional characters before returning.

Chapter 19

19.1 Both servers, `telnetd` and `rlogind`, run with superuser privileges, so their calls to `chown` and `chmod` succeed.

19.2 Execute `pty -n stty -a` to prevent the slave's `termios` structure and `winsize` structure from being initialized.

19.4 Unfortunately, the `F_SETFL` command of `fcntl` doesn't allow the read–write status to be changed.

19.5 There are three process groups: (1) the login shell, (2) the `pty` parent and child, and (3) the `cat` process. The first two process groups constitute a session with the login shell as the session leader. The second session contains only the `cat` process. The first process group (the login shell) is a background process group, and the other two are foreground process groups.

19.6 First, `cat` terminates when it receives the end of file from its line discipline. This causes the PTY slave to terminate, which causes the PTY master to terminate. This in turn generates an end of file for the `pty` parent that's reading from the PTY master. The parent sends `SIGTERM` to the child, so the child terminates next. (The child doesn't catch this signal.) Finally, the parent calls `exit(0)` at the end of the `main` function.

The relevant output from the program shown in Figure 8.29 is

```
cat       e =    270, chars =     274, stat =   0:
pty       e =    262, chars =      40, stat =  15: F     X
pty       e =    288, chars =     188, stat =   0:
```

19.7 This can be done with the shell's `echo` command and the `date`(1) command, all in a subshell:

```
#!/bin/sh
( echo "Script started on " `date`;
  pty "${SHELL:-/bin/sh}";
  echo "Script done on " `date` ) | tee typescript
```

19.8 The line discipline above the PTY slave has echo enabled, so whatever `pty` reads on its standard input and writes to the PTY master gets echoed by default. This echoing is done by the line discipline module above the slave even though the program (`ttyname`) never reads the data.

Chapter 20

20.1 Our conservative locking in `_db_dodelete` is meant to avoid race conditions with `db_nextrec`. If the call to `_db_writedat` were not protected with a write lock, it would be possible to erase the data record while `db_nextrec` was reading that data record: `db_nextrec` would read an index record, determine

that it was not blank, and then read the data record, which could be erased by `_db_dodelete` between the calls to `_db_readidx` and `_db_readdat` in `db_nextrec`.

20.2 Assume that `db_nextrec` calls `_db_readidx`, which reads the key into the index buffer for the process. This process is then stopped by the kernel, and another process runs. This other process calls `db_delete`, and the record being read by the other process is deleted. Both its key and its data are rewritten in the two files as all blanks. The first process resumes and calls `_db_readdat` (from `db_nextrec`) and reads the all-blank data record. The read lock by `db_nextrec` allows it to do the read of the index record, followed by the read of the data record, as an atomic operation (with regard to other cooperating processes using the same database).

20.3 With mandatory locking, other readers and writers are affected. Other reads and writes are blocked by the kernel until the locks placed by `_db_writeidx` and `_db_writedat` are removed.

20.5 By writing the data record before the index record, we protect ourselves from generating a corrupt record if the process should be killed in between the two writes. If the process were to write the index record first, but be killed before writing the data record, then we'd have a valid index record that pointed to invalid data.

Chapter 21

21.5 Here are some hints. There are two places to check for queued jobs: the printer spooling daemon's queue and the network printer's internal queue. Take care to prevent one user from being able to cancel someone else's print job. Of course, the superuser should be able to cancel any job.

21.7 We don't need to prod the daemon, because we don't need to reread the configuration file until we need to print a file. The `printer_thread` function checks whether it needs to reread the configuration file before each attempt to send a job to the printer.

21.9 We need to null-terminate the string we write to the job file (recall that `strlen` doesn't include the terminating null byte when it calculates the length of a string). There are two simple approaches: either we can add 1 to the number of bytes we write, or we can use the `dprintf` function instead of calling `sprintf` and `write`.

This page intentionally left blank

Bibliography

Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. 1986. "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the 1986 Summer USENIX Conference*, pp. 93−113, Atlanta, GA.

A paper introducing the Mach operating system.

Adams, J., Bustos, D., Hahn, S., Powell, D., and Praza, L. 2005. "Solaris Service Management Facility: Modern System Startup and Administration," *Proceedings of the 19th Large Installation System Administration Conference (LISA'05)*, pp. 225−236, San Diego, CA.

A paper describing the Service Management Facility (SMF) in Solaris, which provides a framework for starting and monitoring administrative processes, and recovering from failures affecting the services they provide.

Adobe Systems Inc. 1999. *PostScript Language Reference Manual, Third Edition*. Addison-Wesley, Reading, MA.

The language reference manual for PostScript.

Aho, A. V., Kernighan, B. W., and Weinberger, P. J. 1988. *The AWK Programming Language*. Addison-Wesley, Reading, MA.

A complete book on the awk programming language. The version of awk described in this book is sometimes called "nawk" (for new awk).

Andrade, J. M., Carges, M. T., and Kovach, K. R. 1989. "Building a Transaction Processing System on UNIX Systems," *Proceedings of the 1989 USENIX Transaction Processing Workshop*, pp. 13−22, Pittsburgh, PA.

A description of the AT&T Tuxedo Transaction Processing System.

Arnold, J. Q. 1986. "Shared Libraries on UNIX System V," *Proceedings of the 1986 Summer USENIX Conference*, pp. 395−404, Atlanta, GA.

Describes the implementation of shared libraries in SVR3.

AT&T. 1989. *System V Interface Definition, Third Edition*. Addison-Wesley, Reading, MA.

This four-volume set specifies the source code interface and runtime behavior of System V. The third edition corresponds to SVR4. A fifth volume, containing updated versions of commands and functions from volumes 1−4, was published in 1991. Currently out of print.

AT&T. 1990a. *UNIX Research System Programmer's Manual, Tenth Edition, Volume I*. Saunders College Publishing, Fort Worth, TX.

The version of the *UNIX Programmer's Manual* for the 10th Edition of the Research UNIX System (V10). This volume contains the traditional UNIX System manual pages (Sections 1−9).

AT&T. 1990b. *UNIX Research System Papers, Tenth Edition, Volume II*. Saunders College Publishing, Fort Worth, TX.

Volume II for the 10th Edition of the Research UNIX System (V10) contains 40 papers describing various aspects of the system.

AT&T. 1990c. *UNIX System V Release 4 BSD/XENIX Compatibility Guide*. Prentice Hall, Englewood Cliffs, NJ.

Contains manual pages describing the compatibility library.

AT&T. 1990d. *UNIX System V Release 4 Programmer's Guide: STREAMS*. Prentice Hall, Englewood Cliffs, NJ.

Describes the STREAMS system in SVR4.

AT&T. 1990e. *UNIX System V Release 4 Programmer's Reference Manual*. Prentice Hall, Englewood Cliffs, NJ.

This is the programmer's reference manual for the SVR4 implementation for the Intel 80386 processor. It contains Sections 1 (commands), 2 (system calls), 3 (subroutines), 4 (file formats), and 5 (miscellaneous facilities).

AT&T. 1991. *UNIX System V Release 4 System Administrator's Reference Manual*. Prentice Hall, Englewood Cliffs, NJ.

This is the system administrator's reference manual for the SVR4 implementation for the Intel 80386 processor. It contains Sections 1 (commands), 4 (file formats), 5 (miscellaneous facilities), and 7 (special files).

Bach, M. J. 1986. *The Design of the UNIX Operating System*. Prentice Hall, Englewood Cliffs, NJ.

A book on the details of the design and implementation of the UNIX operating system. Although actual UNIX System source code is not provided in this text (since it was proprietary to AT&T at the time), many of the algorithms and data structures used by the UNIX kernel are presented and discussed. This book describes SVR2.

Bolsky, M. I., and Korn, D. G. 1995. *The New KornShell Command and Programming Language, Second Edition*. Prentice Hall, Englewood Cliffs, NJ.

A book describing how to use the Korn shell, both as a command interpreter and as a programming language.

Chen, D., Barkley, R. E., and Lee, T. P. 1990. "Insuring Improved VM Performance: Some No-Fault Policies," *Proceedings of the 1990 Winter USENIX Conference*, pp. 11−22, Washington, DC.

Describes changes made to the virtual memory implementation of SVR4 to improve its performance, especially for `fork` and `exec`.

Comer, D. E. 1979. "The Ubiquitous B-Tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121−137 (June).

A good, comprehensive paper on B-trees.

Date, C. J. 2004. *An Introduction to Database Systems, Eighth Edition*. Addison-Wesley, Boston, MA.

A comprehensive overview of database systems.

Evans, J. 2006. "A Scalable Concurrent `malloc` Implementation for FreeBSD," *Proceedings of BSDCan*.

A paper describing the `jemalloc` implementation of the dynamic memory allocation library used in FreeBSD.

Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R. 1979. "Extendible Hashing—A Fast Access Method for Dynamic Files," *ACM Transactions on Databases*, vol. 4, no. 3, pp. 315−344 (September).

A paper describing the extendible hashing technique.

Fowler, G. S., Korn, D. G., and Vo, K. P. 1989. "An Efficient File Hierarchy Walker," *Proceeding of the 1989 Summer USENIX Conference*, pp. 173−188, Baltimore, MD.

Describes an alternative library function to traverse a file system hierarchy.

Gallmeister, B. O. 1995. *POSIX.4: Programming for the Real World*. O'Reilly & Associates, Sebastopol, CA.

Describes the real-time interfaces in the POSIX standard.

Garfinkel, S., Spafford, G., and Schwartz, A. 2003. *Practical UNIX & Internet Security, Third Edition*. O'Reilly & Associates, Sebastopol, CA.

A detailed book on UNIX System security.

Ghemawat, S., and Menage, P. 2005. "TCMalloc: Thread-Caching Malloc."

A brief description of Google's `TCMalloc` memory allocator. The description is available at `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

Gingell, R. A., Lee, M., Dang, X. T., and Weeks, M. S. 1987. "Shared Libraries in SunOS," *Proceedings of the 1987 Summer USENIX Conference*, pp. 131−145, Phoenix, AZ.

Describes the implementation of shared libraries in SunOS.

Gingell, R. A., Moran, J. P., and Shannon, W. A. 1987. "Virtual Memory Architecture in SunOS," *Proceedings of the 1987 Summer USENIX Conference*, pp. 81−94, Phoenix, AZ.

Describes the initial implementation of the `mmap` function and related issues in the virtual memory design.

Goodheart, B. 1991. *UNIX Curses Explained*. Prentice Hall, Englewood Cliffs, NJ.

A complete reference on terminfo and the `curses` library. Currently out of print.

Hume, A. G. 1988. "A Tale of Two Greps," *Software Practice and Experience*, vol. 18, no. 11, pp. 1063−1072.

An interesting paper that discusses performance improvements in `grep`.

IEEE. 1990. *Information Technology—Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language].* IEEE (Dec.).

This was the first of the POSIX standards, and it defined the C language systems interface standard, based on the UNIX operating system. It is often called POSIX.1. It is now part of the Single UNIX Specification published by The Open Group [2008].

ISO. 1999. *International Standard ISO/IEC 9899—Programming Language C.* ISO/IEC.

The official standard for the C language and the standard libraries. Although this was replaced by a new version of the standard in 2011, the systems described in this book still conform to the 1999 version of the standard.

PDF versions of this standard can be purchased online at either `http://www.ansi.org` or `http://www.iso.org`.

ISO. 2011. *International Standard ISO/IEC 9899, Information Technology—Programming Languages—C.* ISO/IEC.

The latest version of the official standard for the C language and the standard libraries, which replaces the 1999 version.

PDF versions of this standard can be purchased online at either `http://www.ansi.org` or `http://www.iso.org`.

Kernighan, B. W., and Pike, R. 1984. *The UNIX Programming Environment*. Prentice Hall, Englewood Cliffs, NJ.

A general reference for additional details on UNIX programming. This book covers numerous UNIX commands and utilities, such as `grep`, `sed`, `awk`, and the Bourne shell.

Kernighan, B. W., and Ritchie, D. M. 1988. *The C Programming Language, Second Edition*. Prentice Hall, Englewood Cliffs, NJ.

A book on the ANSI standard version of the C programming language. Appendix B contains a description of the libraries defined by the ANSI standard.

Kerrisk, M. 2010. *The Linux Programming Interface*. No Startch Press, San Francisco, CA.

If you thought this book was long, here is one that is half again as big, but focuses only on the Linux programming interface.

Kleiman, S. R. 1986. "Vnodes: An Architecture for Multiple File System Types in Sun Unix," *Proceedings of the 1986 Summer USENIX Conference*, pp. 238−247, Atlanta, GA.

A description of the original v-node implementation.

Knuth, D. E. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition*. Addison-Wesley, Boston, MA.

Describes sorting and searching algorithms.

Korn, D. G., and Vo, K. P. 1991. "SFIO: Safe/Fast String/File IO," *Proceedings of the 1991 Summer USENIX Conference*, pp. 235−255, Nashville, TN.

A description of an alternative to the standard I/O library. The library is available at `http://www.research.att.com/sw/tools/sfio`.

Krieger, O., Stumm, M., and Unrau, R. 1992. "Exploiting the Advantages of Mapped Files for Stream I/O," *Proceedings of the 1992 Winter USENIX Conference*, pp. 27−42, San Francisco, CA.

An alternative to the standard I/O library based on mapped files.

Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. 1989. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, MA.

An entire book on the 4.3BSD operating system. This book describes the Tahoe release of 4.3BSD. Currently out of print.

Lennert, D. 1987. "How to Write a UNIX Daemon," *;login:*, vol. 12, no. 4, pp. 17–23 (July/August).

Describes how to write a daemon in the UNIX System.

Libes, D. 1990. "expect: Curing Those Uncontrollable Fits of Interaction," *Proceedings of the 1990 Summer USENIX Conference*, pp. 183–192, Anaheim, CA.

A description of the `expect` program and its implementation.

Libes, D. 1991. "expect: Scripts for Controlling Interactive Processes," *Computing Systems*, vol. 4, no. 2, pp. 99–125 (Spring).

This paper presents numerous `expect` scripts.

Libes, D. 1994. *Exploring Expect*. O'Reilly & Associates, Sebastopol, CA.

A complete book on using the `expect` program.

Lions, J. 1977. *A Commentary on the UNIX Operating System*. AT&T Bell Laboratories, Murray Hill, NJ.

Describes the source code of the 6th Edition UNIX System. Available only to AT&T employees, contractors, and interns, although copies leaked outside of AT&T.

Lions, J. 1996. *Lions' Commentary on UNIX 6th Edition*. Peer-to-Peer Communications, San Jose, CA.

Describes the 6th Edition UNIX System in a publicly available version of the 1977 classic.

Litwin, W. 1980. "Linear Hashing: A New Tool for File and Table Addressing," *Proceedings of the 6th International Conference on Very Large Databases*, pp. 212–223, Montreal, Canada.

A paper describing the linear hashing technique.

McKusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S. 1996. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, Reading, MA.

An entire book on the 4.4BSD operating system.

McKusick, M. K., and Neville-Neil, G. V. 2005. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, Boston, MA.

An entire book on the FreeBSD operating system, version 5.2.

McDougall, R., and Mauro, J. 2007. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture, Second Edition*. Prentice Hall, Upper Saddle River, NJ.

A book on the internals of the Solaris 10 operating system. Also covers OpenSolaris.

Morris, R., and Thompson, K. 1979. "UNIX Password Security," *Communications of the ACM*, vol. 22, no. 11, pp. 594–597 (Nov.).

A description of the history of the design of the password scheme used in UNIX Systems.

Nemeth, E., Snyder, G., Seebass, S., and Hein, T. R. 2001. *UNIX System Administration Handbook, Third Edition*. Prentice Hall, Upper Saddle River, NJ.

A book with many details on administering a UNIX system.

The Open Group. 2008. *The Single UNIX Specification, Version 4*. The Open Group, Berkshire, UK.

The POSIX and X/Open standards combined into a single reference.

The HTML version can be viewed for free online at `http://www.opengroup.org`.

Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. 1995. "Plan 9 from Bell Labs," *Plan 9 Programmer's Manual Volume 2*. AT&T, Reading, MA.

A description of the Plan 9 operating system, developed in the same department where the UNIX System was invented.

Plauger, P. J. 1992. *The Standard C Library*. Prentice Hall, Englewood Cliffs, NJ.

A complete book on the ANSI C library. It contains a complete C implementation of the library.

Presotto, D. L., and Ritchie, D. M. 1990. "Interprocess Communication in the Ninth Edition UNIX System," *Software Practice and Experience*, vol. 20, no. S1, pp. S1/3–S1/17 (June).

Describes the IPC facilities provided by the Ninth Edition Research UNIX System, developed at AT&T Bell Laboratories. The features are built on the stream input–output system and include full-duplex pipes, the ability to pass file descriptors between processes, and unique client connections to servers. A copy of this paper also appears in AT&T [1990b].

Rago, S. A. 1993. *UNIX System V Network Programming*. Addison-Wesley, Reading, MA.

A book that describes the networking programming environment of UNIX System V Release 4, which is based on STREAMS.

Raymond, E. S., ed. 1996. *The New Hacker's Dictionary, Third Edition.* MIT Press, Cambridge, MA.

Lots of computer hacker terms defined.

Salus, P. H. 1994. *A Quarter Century of UNIX*. Addison-Wesley, Reading, MA.

A history of the UNIX System from 1969 to 1994.

Seltzer, M., and Olson, M. 1992. "LIBTP: Portable Modular Transactions for UNIX," *Proceedings of the 1992 Winter USENIX Conference*, pp. 9−25, San Francisco, CA.

A modification of the `db`(3) library from 4.4BSD that implements transactions.

Seltzer, M., and Yigit, O. 1991. "A New Hashing Package for UNIX," *Proceedings of the 1991 Winter USENIX Conference*, pp. 173−184, Dallas, TX.

A description of the `dbm`(3) library and its implementations, and a newer hashing package.

Singh, A. 2006. *Mac OS X Internals: A Systems Approach*. Addison-Wesley, Upper Saddle River, NJ.

Roughly 1,600 pages on the design of the Mac OS X operating system.

Stevens, W. R. 1990. *UNIX Network Programming*. Prentice Hall, Englewood Cliffs, NJ.

A detailed book on network programming under the UNIX System. The contents of the first edition of this book differ greatly from later editions.

Stevens, W. R., Fenner, B., and Rudoff, A. M. 2004. *UNIX Network Programming, Volume 1, Third Edition*. Addison-Wesley, Boston, MA.

A detailed book on network programming under UNIX System. Redesigned and split into two volumes in the second edition and updated in the third edition.

Stonebraker, M. R. 1981. "Operating System Support for Database Management," *Communications of the ACM*, vol. 24, no. 7, pp. 412−418 (July).

Describes operating system services and how they affect database operation.

Strang, J. 1986. *Programming with curses*. O'Reilly & Associates, Sebastopol, CA.

A book on the Berkeley version of `curses`.

Strang, J., Mui, L., and O'Reilly, T. 1988. *termcap & terminfo, Third Edition*. O'Reilly & Associates, Sebastopol, CA.

A book on `termcap` and `terminfo`.

Sun Microsystems. 2005. *STREAMS Programming Guide*. Sun Microsystems, Santa Clara, CA.

Describes STREAMS programming on the Solaris platform.

Thompson, K. 1978. "UNIX Implementation," *The Bell System Technical Journal*, vol. 57, no. 6, pp. 1931−1946 (July–Aug.).

Describes some of the implementation details of Version 7.

Vo, Kiem-Phong. 1996. "Vmalloc: A General and Efficient Memory Allocator," *Software Practice and Experience*, vol. 26, no. 3, pp. 357−374.

Describes a flexible memory allocator.

Wei, J., and Pu, C. 2005. "TOCTTOU Vulnerabilities in UNIX_Style File Systems: An Anatomical Study," *Proceedings of the 4th USENIX Conference on File and Storage Technologoes (FAST'05)*, pp. 155−167, San Francisco, CA.

Describes TOCTTOU weaknesses in the UNIX file system interface.

Weinberger, P. J. 1982. "Making UNIX Operating Systems Safe for Databases," *The Bell System Technical Journal*, vol. 61, no. 9, pp. 2407−2422 (Nov.).

Describes some problems in implementing databases in early UNIX systems.

Weinstock, C. B., and Wulf, W. A. 1988. "Quick Fit: An Efficient Algorithm for Heap Storage Allocation," *SIGPLAN Notices*, vol. 23, no. 10, pp. 141−148.

Describes a memory allocation algorithm suitable for a wide variety of applications.

Williams, T. 1989. "Session Management in System V Release 4," *Proceedings of the 1989 Winter USENIX Conference*, pp. 365−375, San Diego, CA.

Describes the session architecture in SVR4, on which the POSIX.1 interfaces were based. This includes process groups, job control, and controlling terminals. Also describes the security concerns of existing approaches.

X/Open. 1989. *X/Open Portability Guide*. Prentice Hall, Englewood Cliffs, NJ.

A set of seven volumes covering commands and utilities (Vol. 1), system interfaces and headers (Vol. 2), supplementary definitions (Vol. 3), programming languages (Vol. 4), data management (Vol. 5), window management (Vol. 6), networking services (Vol. 7). Although out of print, this has been replaced by the Single UNIX Specification [Open Group 2008].

This page intentionally left blank

Index

The function subentries labeled "definition of" point to where the function prototype appears and, when applicable, to the source code for the function. Functions defined in the text that are used in later examples, such as the `set_fl` function in Figure 3.12, are included in this index. The definitions of functions that are part of the larger examples (Chapters 17, 19, 20, and 21) are also included to help in going through these examples. Also, significant functions and constants that occur in any of the examples in the text, such as `select` and `poll`, are also included in this index. Trivial functions that occur frequently, such as `printf`, are sometimes not referenced when they occur in examples.

`#!`, *see* interpreter file
`.`, *see* current directory
`..`, *see* parent directory
2.9BSD, 234
386BSD, xxxi, 34
4.1BSD, 525
4.2BSD, 18, 34, 81, 121, 129–130, 183, 277, 326, 329, 469, 502, 508, 521, 525, 589
4.3BSD, xxxi, 33–34, 36, 49, 201, 257, 267, 289, 313, 318, 329, 366, 482, 535, 735, 898, 951
  Reno, xxxi, 34, 76
  Tahoe, xxxi, 34, 951
4.4BSD, xxvi, xxxi, 21, 34, 74, 112, 121, 129, 149, 234, 329, 535, 589, 735, 744, 951

`a2ps` program, 842
`abort` function, 198, 236, 241, 272, 275, 313, 317–319, 331, 365–367, 381, 447, 900
  definition of, 365–366
absolute pathname, 5, 8, 43, 50, 64, 136, 141–142, 260, 553, 911
`accept` function, 148, 331, 451, 608–609, 615, 617, 635, 639–640, 648, 817
  definition of, 608
`access` function, 102–104, 121, 124, 331, 452
  definition of, 102
Accetta, M., 35
accounting
  login, 186–187
  process, 269–275
`acct` function, 269
`acct` structure, 270, 273
`acctcom` program, 269
`accton` program, 269, 274
`ACORE` constant, 271, 273–274
Adams, J., 293
`add_job` function, 814, 820, 823, 827
  definition of, 820
`add_option` function, 831, 834
  definition of, 831
addressing, socket, 593–605
`addrinfo` structure, 599–603, 614, 616, 618, 620, 622, 800, 802, 804, 807, 813–814, 816, 819, 833

add_worker function, 814, 824, 828
  definition of, 828
adjustment on exit, semaphore, 570–571
Adobe Systems, 825, 947
advisory record locking, 495
AES (Application Environment Specification), 32
AEXPND constant, 271
AF_INET constant, 590–591, 595–596, 598, 601, 603–604, 802, 808
AF_INET6 constant, 590, 595–596, 601
AF_IPX constant, 590
AF_LOCAL constant, 590
AFORK constant, 270–271, 273
AF_UNIX constant, 590, 601, 630, 632, 635, 637, 640–641, 941
AF_UNSPEC constant, 590, 601
agetty program, 290
Aho, A. V., 262, 947
AI_ALL constant, 603
AI_CANONNAME constant, 603, 616, 618, 623, 802
AI_NUMERICHOST constant, 603
AI_NUMERICSERV constant, 603
aio_cancel function, 514–515
  definition of, 514
aiocb structure, 511, 517–518
aio_error function, 331, 513, 515, 519–520
  definition of, 513
aio_fsync function, 512–513, 520
  definition of, 513
<aio.h> header, 29
AIO_LISTIO_MAX constant, 515–516
AIO_MAX constant, 515–516
AIO_PRIO_DELTA_MAX constant, 515–516
aio_read function, 512–513, 515, 518
  definition of, 512
aio_return function, 331, 513, 519–520
  definition of, 513
aio_suspend function, 331, 451, 514, 520
  definition of, 514
aio_write function, 512–513, 515, 519
  definition of, 512
AI_PASSIVE constant, 603
AI_V4MAPPED constant, 600, 603
AIX, 35, 334
alarm function, 313, 317, 331–332, 335, 338–343, 357, 373–374, 381–382, 620–621, 924
  definition of, 338
alloca function, 210
Almquist, K., 4
already_running function, 475–478
  definition of, 474
ALTWERASE constant, 676, 682, 685
American National Standards Institute, *see* ANSI
Andrade, J. M., 560, 947
ANSI (American National Standards Institute), 25
ANSI C, xxx–xxxi
Apple Computer, xxi, xxvi
Application Environment Specification, *see* AES
apue_db.h header, 745, 753, 757, 761
apue.h header, 7, 9–10, 247, 324, 489–490, 635, 755, 895–898
Architecture, UNIX, 1–2
argc variable, 815
ARG_MAX constant, 40, 43, 47, 49, 251
arguments, command-line, 203
argv variable, 663
Arnold J. Q., 206, 947
<arpa/inet.h> header, 29, 594
asctime function, 192
<assert.h> header, 27
assignment-allocation character, 162
ASU constant, 271, 273
asynchronous I/O, 501, 509–520
asynchronous socket I/O, 627
async-signal safe, 330, 446, 450, 457, 461–462, 927
at program, 259, 472
atd program, 259, 465
AT_EACCESS constant, 103
atexit function, 40–41, 43, 200, 202, 226, 236, 394, 731, 920
  definition of, 200
ATEXIT_MAX constant, 40–41, 43, 49, 52
AT_FDCWD constant, 65, 94, 102, 106, 110, 116–117, 120, 123–124, 127, 129, 553
atoi function, 766, 839–840
atol function, 765–767, 818, 823
atomic operation, 39, 44, 59, 63, 77–79, 81, 116, 149, 359, 365, 488, 553, 566, 568, 570, 945
AT_REMOVEDIR constant, 117
AT_SYMLINK_FOLLOW constant, 116
AT_SYMLINK_NOFOLLOW constant, 94, 106, 110, 127
AT&T, xix, 6, 33, 174, 336, 507, 948
automatic variables, 205, 215, 217, 219, 226
avoidance, deadlock, 402–407
awk program, 44, 46, 262–264, 552, 950
AXSIG constant, 271, 273–274

B0 constant, 692
B110 constant, 692
B115200 constant, 692
B1200 constant, 692
B134 constant, 692
B150 constant, 692

B1800 constant, 692
B19200 constant, 692
B200 constant, 692
B2400 constant, 692
B300 constant, 692
B38400 constant, 692
B4800 constant, 692
B50 constant, 692
B57600 constant, 692
B600 constant, 692
B75 constant, 692
B9600 constant, 692
Bach, M. J., xix, xxxii, 74, 81, 112, 116, 229, 907, 948
background process group, 296, 300, 302, 304, 306–307, 309, 321, 369, 377, 944
backoff, exponential, 606
Barkley, R. E., 948
barrier attributes, 441–442
barriers, 418–422
basename function, 442
bash program, 85, 372
.bash_login file, 289
.bash_profile file, 289
Bass, J., 485
baud rate, terminal I/O, 692–693
Berkeley Software Distribution, *see* BSD
bibliography, alphabetical, 947–953
big-endian byte order, 593, 791
bind function, 331, 604, 609, 624–625, 634–635, 637–638, 641
  definition of, 604
/bin/false program, 179
/bin/true program, 179
<bits/signum.h> header, 314
block special file, 95, 138–139
Bolsky, M. I., 548, 948
Bostic, K., xxxii, 33, 74, 112, 116, 525, 951
  Keith, 229, 236
Bourne, S. R., 3
Bourne shell, 3, 53, 90, 210, 222, 289, 299, 303, 372, 497, 542, 548, 702, 935, 950
Bourne-again shell, 3–4, 53, 85, 90, 210, 222, 289, 300, 548
Bovet, D. P., 74
BREAK character, 677, 682, 685, 688, 690, 694, 708
BRKINT constant, 676, 685, 688, 706–708
BS0 constant, 685
BS1 constant, 685
BSD (Berkeley Software Distribution), 34, 65, 111, 175, 286, 289, 291, 293, 296–297, 299, 482, 501, 509–511, 532, 596–597, 630, 726–727, 734, 742
BSD Networking Release 1.0, xxxi, 34
BSD Networking Release 2.0, xxxi, 34
BSD/386, xxxi
BSDLY constant, 676, 684–685, 689
__BSD_VISIBLE constant, 473
bss segment, 205
buf_args function, 656–658, 668–670, 897
  definition of, 657
buffer cache, 81
buffering, standard I/O, 145–147, 231, 235, 265, 367, 552, 721, 752
BUFSIZ constant, 49, 147, 166, 220
build_qonstart function, 814, 817, 822
  definition of, 822
BUS_ADRALN constant, 353
BUS_ADRERR constant, 353
BUS_OBJERR constant, 353
byte order, xxii, 593–594, 792, 810, 825, 831, 834, 842, 861, 865
  big-endian, 593, 791
  little-endian, 593

C, ANSI, xxx–xxxi
  ISO, 25–26, 153, 950
C shell, 3, 53, 222, 289, 299, 548
c99 program, 58, 70
cache
  buffer, 81
  page, 81
CAE (Common Application Environment), 32
calendar time, 20, 24, 59, 126, 189, 191–192, 264, 270
calloc function, 207–208, 226, 544, 760, 920
  definition of, 207
cancellation point, 451
canonical mode, terminal I/O, 700–703
Carges, M. T., 560, 947
cat constant, 301
cat program, 89, 112, 123, 301, 304, 734–735, 748, 944
catclose function, 452
catgets function, 442, 452
catopen function, 452
CBAUDEXT constant, 675, 685
cbreak terminal mode, 672, 704, 708, 713
cc program, 6, 57, 206
CCAR_OFLOW constant, 675, 685, 689
cc_t data type, 674
CCTS_OFLOW constant, 675, 685
cd program, 136
CDSR_OFLOW constant, 675, 685
CDTR_IFLOW constant, 675, 685
Cesati, M., 74

cfgetispeed function, 331, 677, 692
  definition of, 692
cfgetospeed function, 331, 677, 692
  definition of, 692
cfsetispeed function, 331, 677, 692
  definition of, 692
cfsetospeed function, 331, 677, 692
  definition of, 692
character special file, 95, 138–139, 699
CHAR_BIT constant, 37–38
CHARCLASS_NAME_MAX constant, 39, 49
CHAR_MAX constant, 37–38
CHAR_MIN constant, 37–38
chdir function, 8, 121, 135–137, 141, 222, 288, 331, 468, 912
  definition of, 135
Chen, D., 948
CHILD_MAX constant, 40, 43, 49, 233
chmod function, 106–108, 121, 125, 331, 452, 558, 641, 944
  definition of, 106
chmod program, 99–100, 559
chown function, 55, 109–110, 120–121, 125, 288, 331, 452, 558, 944
  definition of, 109
chroot function, 141, 480, 910, 928
CIBAUDEXT constant, 675, 685
CIGNORE constant, 675, 685
Clark, J. J., xxxii
CLD_CONTINUED constant, 353
CLD_DUMPED constant, 353
CLD_EXITED constant, 353
CLD_KILLED constant, 353
CLD_STOPPED constant, 353
CLD_TRAPPED constant, 353
clearenv function, 212
clearerr function, 151
  definition of, 151
cli_args function, 656–658, 668–669
  definition of, 658
cli_conn function, 636–637, 640, 659, 665, 897
  definition of, 636, 640
client_add function, 662, 665, 667
  definition of, 661
client_alloc function, 661–662, 668
  definition of, 660
client_cleanup function, 814, 824, 829
  definition of, 829
client_del function, 665, 667
  definition of, 661
client–server model, 479–480, 585–587
client_thread function, 814, 817, 824
  definition of, 824
CLOCAL constant, 318, 675, 685
clock function, 58–59
clock tick, 20, 42–43, 49, 59, 270, 280
clock_getres function, 190
  definition of, 190
clock_gettime function, 189–190, 331, 408, 414, 437, 439
  definition of, 189
clockid_t data type, 189
CLOCK_MONOTONIC constant, 189
clock_nanosleep function, 373–375, 437, 439, 451, 462
  definition of, 375
CLOCK_PROCESS_CPUTIME_ID constant, 189
CLOCK_REALTIME constant, 189–190, 408, 437, 439, 581
clock_settime function, 190, 439
  definition of, 190
CLOCKS_PER_SEC constant, 59
clock_t data type, 20, 58–59, 280
CLOCK_THREAD_CPUTIME_ID constant, 189
clone function, 229
close function, 8, 52, 61, 66, 80–81, 124, 128, 331, 451, 468, 474, 492, 532, 537–539, 544, 550, 553, 560, 577–578, 587, 592–593, 609, 616, 618, 625, 638–639, 641, 654–655, 657, 665, 667–669, 725–726, 728–729, 739–740, 761, 823, 826–827, 829, 833, 837
  definition of, 66
closedir function, 5, 7, 130–135, 452, 698, 823, 910
  definition of, 130
closelog function, 452, 470
  definition of, 470
close-on-exec flag, 80, 83, 252–253, 479–480, 492
clrasync function, definition of, 940
clr_fl function, 85, 482–483, 896, 937
clri program, 122
cmsgcred structure, 648–651
CMSG_DATA function, 645–646, 648, 650, 652
  definition of, 645
CMSG_FIRSTHDR function, 645, 652
  definition of, 645
cmsghdr structure, 645–647, 649, 651
CMSG_LEN function, 645–647, 649, 651
  definition of, 645
CMSG_NXTHDR function, 645, 650, 652
  definition of, 645
CMSPAR constant, 675, 685, 690
codes, option, 31
COLL_WEIGHTS_MAX constant, 39, 43, 49
COLUMNS environment variable, 211
Comer, D. E., 744, 949

command-line arguments, 203
Common Application Environment, *see* CAE
Common Open Software Environment, *see* COSE
communication, network printer, 789–843
<complex.h> header, 27
comp_t data type, 59
Computing Science Research Group, *see* CSRG
condition variable attributes, 440–441
condition variables, 413–416
cond_signal function, 416
connect function, 331, 451, 605–608, 610–611, 621, 635, 641–642
  definition of, 605
connection establishment, 605–609
connect_retry function, 607, 614, 800, 808, 834
  definition of, 606–607
controlling
  process, 296–297, 318
  terminal, 63, 233, 252, 270, 292, 295–298, 301, 303–304, 306, 309, 311–312, 318, 321, 377, 463, 465–466, 469, 480, 680, 685, 691, 694, 700, 702, 716, 724, 726–727, 898, 953
cooked terminal mode, 672
cooperating processes, 495, 752, 945
Coordinated Universal Time, *see* UTC
coprocesses, 548–552, 721, 737
copy-on-write, 229, 458
core dump, 74, 928
core file, 111, 124, 275, 315, 317, 320, 332, 366, 681, 703, 909, 920, 922
COSE (Common Open Software Environment), 32
count, link, 44, 59, 114–117, 130
cp program, 141, 528
cpio program, 127, 142, 910–911
<cpio.h> header, 29
CR terminal character, 678, 680, 703
CR0 constant, 685
CR1 constant, 685
CR2 constant, 685
CR3 constant, 685
CRDLY constant, 676, 684–685, 689
CREAD constant, 675, 686
creat function, 61, 66, 68, 79, 89, 101, 104, 118, 121, 125, 149, 331, 451, 491, 825–826, 909, 912
  definition of, 66
creation mask, file mode, 104–105, 129, 141, 169, 233, 252, 466
cron program, 259, 382, 465, 470, 472–474, 925
CRTSCTS constant, 675, 686
CRTS_IFLOW constant, 675, 686
CRTSXOFF constant, 675, 686
crypt function, 287, 298, 304, 442
crypt program, 298, 700
CS5 constant, 684, 686
CS6 constant, 684, 686
CS7 constant, 684, 686
CS8 constant, 684, 686, 706–708
.cshrc file, 289
CSIZE constant, 675, 684, 686, 706–707
csopen function, 653–654
  definition of, 654, 659
CSRG (Computing Science Research Group), xx, xxvi, 34
CSTOPB constant, 675, 686
ctermid function, 442, 452, 694, 700–701
  definition of, 694
ctime function, 192
<ctype.h> header, 27
cu program, 500
cupsd program, 465, 793
current directory, 4–5, 8, 13, 43, 50, 65, 94, 100, 115–117, 120, 127, 130, 135–137, 178, 211, 233, 252, 315, 317, 466
Curses, 32
curses library, 712–713, 949, 953
cuserid function, 276

daemon, 463–480
  coding, 466–469
  conventions, 474–479
  error logging, 469–473
daemonize function, 466, 468, 480, 616, 618, 623, 664, 815, 896, 929–930
  definition of, 467
Dang, X. T., 206, 949
Darwin, xxii, xxvii, 35
dash program, 372
data, out-of-band, 626
data segment
  initialized, 205
  uninitialized, 205
data transfer, 610–623
data types, primitive system, 58
database library, 743–787
  coarse-grained locking, 752
  concurrency, 752–753
  fine-grained locking, 752
  implementation, 746–750
  performance, 781–786
  source code, 753–781
database transactions, 952
Date, C. J., 753, 949
date functions, time and, 189–196
date program, 192, 196, 371, 919, 944

DATEMSK environment variable, 211
db library, 744, 952
DB structure, 756–758, 760–762, 765–768, 773, 776, 782
_db_alloc function, 757, 760–761
  definition of, 760
db_close function, 745, 749, 754, 761
  definition of, 745, 761
db_delete function, 746, 752, 754, 768–769, 771, 945
  definition of, 746, 768
_db_dodelete function, 757, 768–769, 772, 776, 780–781, 787, 944–945
  definition of, 769
db_fetch function, 745, 748–749, 752, 754, 762, 767
  definition of, 745, 762
_db_find_and_lock function, 757, 762–763, 767–768, 774–775, 777, 786
  definition of, 763
_db_findfree function, 757, 775, 777–778, 781
  definition of, 777
_db_free function, 757–758, 761
  definition of, 761
DBHANDLE data type, 749, 754, 757, 761–762, 768, 774, 779
_db_hash function, 757, 764, 787
  definition of, 764
DB_INSERT constant, 745, 749, 754, 774, 776
dbm library, 743–744, 952
dbm_clearerr function, 442
dbm_close function, 442, 452
dbm_delete function, 442, 452
dbm_error function, 442
dbm_fetch function, 442, 452
dbm_firstkey function, 442
dbm_nextkey function, 442, 452
dbm_open function, 442, 452
dbm_store function, 442, 452
db_nextrec function, 746, 750, 752, 754, 769, 779, 781, 787, 944–945
  definition of, 746, 779
db_open function, 745–746, 749, 752, 754–757, 759–761, 781
  definition of, 745, 757
_db_readdat function, 757, 762, 768, 780, 945
  definition of, 768
_db_readidx function, 757, 764–765, 778, 780, 945
  definition of, 765
_db_readptr function, 757, 763, 765, 770, 775–777, 787
  definition of, 765
DB_REPLACE constant, 745, 754, 774
db_rewind function, 746, 754, 760, 779, 781
  definition of, 746, 779
DB_STORE constant, 745, 754, 774
db_store function, 745, 747, 749, 752, 754, 769, 771, 774, 781, 787
  definition of, 745, 774
_db_writedat function, 757, 769, 771–772, 775–777, 781, 787, 944–945
  definition of, 771
_db_writeidx function, 522, 757, 759, 770, 772, 775–776, 781, 787, 945
  definition of, 772
_db_writeptr function, 757, 759, 770, 773, 775–776, 778
  definition of, 773
dcheck program, 122
dd program, 275
deadlock, 234, 402, 490, 552, 721
  avoidance, 402–407
  record locking, 490
Debian Almquist shell, 4, 53
Debian Linux distribution, 4
delayed write, 81
DELAYTIMER_MAX constant, 40, 43
descriptor set, 503, 505, 532, 933
detachstate attribute, 427–428
/dev/fd device, 88–89, 142, 696
/dev/fd/0 device, 89
/dev/fd/1 device, 89, 142
/dev/fd/2 device, 89
device number
  major, 58–59, 137, 139, 465, 699
  minor, 58–59, 137, 139, 465, 699
device special file, 137–139
/dev/klog device, 470
/dev/kmem device, 68
/dev/log device, 470, 480, 928
/dev/null device, 73, 86, 304
/dev/stderr device, 89, 697
/dev/stdin device, 89, 697
/dev/stdout device, 89, 697
dev_t data type, 59, 137–138
devtmpfs file system, 139
/dev/tty device, 298, 304, 312, 694, 700, 740
/dev/tty1 file, 290
/dev/zero device, 576–578
df program, 141, 910
DIR structure, 7, 131, 283, 697, 822
directories
  files and, 4–8
  hard links and, 117, 120
  reading, 130–135

directory, 4
  current, 4–5, 8, 13, 43, 50, 65, 94, 100, 115–117, 120, 127, 130, 135–137, 178, 211, 233, 252, 315, 317, 466
  file, 95
  home, 2, 8, 135, 211, 288, 292
  ownership, 101–102
  parent, 4, 108, 125, 129
  root, 4, 8, 24, 139, 141, 233, 252, 283, 910
Directory Services daemon, 185
dirent structure, 5, 7, 131, 133, 697, 822
<dirent.h> header, 7, 29, 131
dirname function, 442
DISCARD terminal character, 678, 680, 687
dlclose function, 452
dlerror function, 442
<dlfcn.h> header, 29
dlopen function, 452
do_driver function, 732, 739
  definition of, 739
Dorward, S., 229, 952
DOS, 57, 65
dot, *see* current directory
dot-dot, *see* parent directory
dprintf function, 159, 452, 945
  definition of, 159
drand48 function, 442
DSUSP terminal character, 678, 680, 688
dtruss program, 497
du program, 111, 141, 909
Duff, T., 88
dup function, 52, 61, 74, 77, 79–81, 148, 164, 231, 331, 468, 492–493, 592–593, 907–908, 921
  definition of, 79
dup2 function, 64, 79–81, 90, 148, 331, 539, 544, 550–551, 592, 618–619, 655, 728–729, 739–740, 907–908
  definition of, 79

E2BIG error, 564
EACCES error, 14–15, 474, 487, 499, 918
EAGAIN error, 16, 376, 474, 482, 484, 487, 496–497, 499, 514, 563, 569–570, 581, 609, 627
EBADF error, 52, 916
EBUSY error, 16, 400, 410, 418
ECANCELED error, 515
ECHILD error, 333, 351, 371, 546
ECHO constant, 676, 686–687, 701, 705–707, 731
echo program, 203
ECHOCTL constant, 676, 686
ECHOE constant, 676, 686–687, 701, 731
ECHOK constant, 676, 687, 701, 731
ECHOKE constant, 676, 687
ECHONL constant, 676, 687, 701, 731
ECHOPRT constant, 676, 686–687
ed program, 367, 369–370, 496–497
EDEADLK error, 418
EEXIST error, 121, 558, 584
EFBIG error, 925
effective
  group ID, 98–99, 101–102, 108, 110, 140, 183, 228, 233, 256, 258, 558, 587
  user ID, 98–99, 101–102, 106, 110, 126, 140, 228, 233, 253, 256–260, 276, 286, 288, 337, 381, 558, 562, 568, 573, 586–587, 637, 640, 809, 918
efficiency
  I/O, 72–74
  standard I/O, 153–156
EIDRM error, 562–564, 568–570, 579
EINPROGRESS error, 519–520, 608
EINTR error, 16, 265–266, 301, 327–329, 339, 359, 370, 502, 508, 514, 545–546, 563–564, 569–570, 620
EINVAL error, 42, 47–48, 345, 389, 543, 545–546, 705–707, 774, 914
EIO error, 309, 321, 823–824, 826–827
Ellis, M., xxxii
ELOOP error, 121–122
EMFILE error, 544, 546
EMSGSIZE error, 610
ENAMETOOLONG error, 65, 637, 640
encrypt function, 442
endgrent function, 183–184, 442, 452
  definition of, 183
endhostent function, 452, 597
  definition of, 597
endnetent function, 452, 598
  definition of, 598
endprotoent function, 452, 598
  definition of, 598
endpwent function, 180–181, 442, 452
  definition of, 180
endservent function, 452, 599
  definition of, 599
endspent function, 182
  definition of, 182
endutxent function, 442, 452
ENFILE error, 16
ENOBUFS error, 16
ENOENT error, 15, 170, 445, 745, 774
ENOLCK error, 16
ENOMEM error, 16, 914
ENOMSG error, 564
ENOSPC error, 16, 445

ENOTDIR error, 592
ENOTRECOVERABLE error, 433
ENOTTY error, 683, 693
environ variable, 203–204, 211, 213, 251, 255, 444–445, 450, 920
environment list, 203–204, 233, 251, 286–288
environment variable, 210–213
  COLUMNS, 211
  DATEMSK, 211
  HOME, 210–211, 288
  IFS, 269
  LANG, 41, 211
  LC_ALL, 211
  LC_COLLATE, 43, 211
  LC_CTYPE, 211
  LC_MESSAGES, 211
  LC_MONETARY, 211
  LC_NUMERIC, 211
  LC_TIME, 211
  LD_LIBRARY_PATH, 753
  LINES, 211
  LOGNAME, 211, 276, 288
  MAILPATH, 210
  MALLOC_OPTIONS, 928
  MSGVERB, 211
  NLSPATH, 211
  PAGER, 539, 542–543
  PATH, 100, 211, 250–251, 253, 260, 263, 265, 288–289
  POSIXLY_CORRECT, 111
  PWD, 211
  SHELL, 211, 288, 737
  TERM, 211, 287, 289
  TMPDIR, 211
  TZ, 190, 192, 195–196, 211, 919
  USER, 210, 288
ENXIO error, 553
EOF constant, 10, 151–152, 154, 164, 175, 545, 547–548, 550–551, 664, 730, 913
EOF terminal character, 678, 680, 686–687, 700, 703
EOL terminal character, 678, 680, 687, 700, 703
EOL2 terminal character, 678, 680, 687, 700, 703
EOWNERDEAD error, 432
EPERM error, 256
EPIPE error, 537, 937
Epoch, 20, 22, 126, 187, 189–190, 640
ERANGE error, 50
ERASE terminal character, 678, 680, 686–687, 702–703
ERASE2 terminal character, 678, 681
err_cont function, 897, 899
  definition of, 900
err_dump function, 366, 767, 897, 899
  definition of, 900
err_exit function, 809, 897, 899
  definition of, 900
err_msg function, 897, 899
  definition of, 901
errno variable, 14–15, 42, 50, 55, 65, 67, 81, 121, 144, 256, 265, 277, 301, 309, 314, 321, 327–328, 330–331, 333, 337, 339, 345, 351, 359, 371, 376, 380, 384, 386, 446–447, 454, 471, 474, 482, 484, 487, 499, 502, 508, 513–514, 537, 546, 553, 564, 568, 579, 581, 584, 592, 608–610, 627, 637–638, 640, 683, 693, 745, 805, 899, 925, 937
<errno.h> header, 14–16, 27
error
  handling, 14–16
  logging, daemon, 469–473
  recovery, 16
  routines, standard, 898–904
  TOCTTOU, 65, 250, 953
err_quit function, 7, 815, 897, 899, 912
  definition of, 901
err_ret function, 897, 899, 912
  definition of, 899
err_sys function, 7, 897, 899
  definition of, 899
ESPIPE error, 67, 592
ESRCH error, 337
/etc/gettydefs file, 290
/etc/group file, 17–18, 177, 185–186
/etc/hosts file, 186, 795
/etc/init directory, 290
/etc/inittab file, 290
/etc/master.passwd file, 185
/etc/networks file, 185–186
/etc/passwd file, 2, 99, 135, 177–178, 180, 182, 185–186
/etc/printer.conf file, 794–795, 799
/etc/protocols file, 185–186
/etc/pwd.db file, 185
/etc/rc file, 189, 291
/etc/services file, 185–186
/etc/shadow file, 99, 185–186
/etc/spwd.db file, 185
/etc/syslog.conf file, 470
/etc/termcap file, 712
/etc/ttys file, 286
ETIME error, 800, 805
ETIMEDOUT error, 407, 413, 415, 581, 800
Evans, J., 949
EWOULDBLOCK error, 16, 482, 609, 627
exec function, 10–11, 13, 23, 39–40, 43, 79, 82, 100, 121, 125, 197, 201, 203, 225, 229, 233–234, 249–257, 260–261, 264–266, 269–271, 275, 277, 282–283, 286–288, 290–292, 294, 305,

325, 372, 457, 479, 492, 527, 533, 538, 541, 557, 585, 653–654, 658–659, 669, 716–717, 721, 723, 727, 739, 742, 920, 928, 948
execl function, 249–251, 261, 265–266, 272, 274–275, 283, 288, 331, 370–371, 539, 544, 550–551, 618, 655, 737, 922
definition of, 249
execle function, 249–251, 254, 287, 331
definition of, 249
execlp function, 12–13, 19, 249–251, 253–254, 264–265, 283, 740, 922
definition of, 249
execv function, 249–251, 331
definition of, 249
execve function, 249–251, 253, 331, 922
definition of, 249
execvp function, 249–251, 253, 731–732
definition of, 249
exercises, solutions to, 905–945
_Exit function, 198, 201, 236–237, 239, 331, 365, 367, 388, 447
definition of, 198
_exit function, 198, 201, 235–239, 265–266, 282–283, 331, 365, 367, 370, 381, 388, 447, 921, 924
definition of, 198
exit function, 7, 150, 154, 198–202, 226, 231, 234–239, 246, 249, 265, 271–272, 274–275, 283, 288, 330, 365–366, 388, 447, 466, 542, 705, 732, 742, 817, 830, 895, 920–921, 944
definition of, 198
exit handler, 200
expect program, 720, 739–740, 951
exponential backoff, 606
ext2 file system, 129
ext3 file system, 129
ext4 file system, 73, 86, 129, 465
EXTPROC constant, 676, 687

faccessat function, 102–104, 331, 452
definition of, 102
Fagin, R., 744, 750, 949
Fast-STREAMS, Linux, 534
fatal error, 16
fchdir function, 135–137, 592
definition of, 135
fchmod function, 106–108, 120, 125, 331, 452, 498, 592
definition of, 106
fchmodat function, 106–108, 331, 452
definition of, 106
fchown function, 109–110, 125, 331, 452, 592
definition of, 109
fchownat function, 109–110, 331, 452
definition of, 109
fclose function, 148–150, 172–174, 199, 201, 365, 367, 452, 545, 701, 803
definition of, 150
fcntl function, 61, 77, 80–87, 90, 112, 148, 164, 252–253, 331, 451–452, 480, 482, 485–490, 492, 494–495, 510–511, 592, 626–627, 783, 785, 939, 944
definition of, 82
<fcntl.h> header, 29, 62
fdatasync function, 81, 86–87, 331, 451, 513, 592
definition of, 81
FD_CLOEXEC constant, 63, 79, 82–83, 252, 480
FD_CLR function, 503–504, 665, 933
definition of, 503
FD_ISSET function, 503–504, 665, 817, 933
definition of, 503
fdopen function, 148–150, 159, 544, 936
definition of, 148
fdopendir function, 130–135
definition of, 130
fd-pipe, 653–654, 656, 658
fd_pipe function, 630, 655, 739, 896
definition of, 630
fd_set data type, 59, 503–504, 532, 664, 805, 814, 816–817, 932–933, 939
FD_SET function, 503–504, 664–665, 805, 816, 933
definition of, 503
__FD_SETSIZE constant, 933
FD_SETSIZE constant, 504, 932–933
F_DUPFD constant, 81–83, 592
F_DUPFD_CLOEXEC constant, 82, 592
FD_ZERO function, 503–504, 664, 805, 933
definition of, 503
feature test macro, 57–58, 84
Fenner, B., 157, 291, 470, 589, 952
<fenv.h> header, 27
feof function, 151, 157
definition of, 151
ferror function, 10, 151, 154, 157, 273, 538, 543, 550
definition of, 151
fexecve function, 249–250, 253, 331
definition of, 249
FF0 constant, 687
FF1 constant, 687
FFDLY constant, 676, 684, 687, 689
fflush function, 145, 147, 149, 172, 174–175, 366, 452, 547–548, 552, 702, 721, 901, 904, 913
definition of, 147

F_FREESP constant, 112
fgetc function, 150–151, 154–155, 452
  definition of, 150
F_GETFD constant, 82–83, 480, 592
F_GETFL constant, 82–85, 592
F_GETLK constant, 82, 486–490
F_GETOWN constant, 82–83, 592, 626
fgetpos function, 157–159, 452
  definition of, 158
fgets function, 10, 12, 19, 150, 152–156, 168, 174–175, 214, 216, 452, 538, 543, 548, 550–552, 616, 622, 654, 738, 753, 803, 845, 911, 913, 936
  definition of, 152
fgetwc function, 452
fgetws function, 452
FIFOs, 95, 534, 552–556
file
  access permissions, 99–101, 140
  block special, 95, 138–139
  character special, 95, 138–139, 699
  descriptor passing, 587, 642–652
  descriptors, 8–10, 61–62
  device special, 137–139
  directory, 95
  group, 182–183
  holes, 68–69, 111–112
  mode creation mask, 104–105, 129, 141, 169, 233, 252, 466
  offset, 66–68, 74, 77–78, 80, 231–232, 494, 522, 747–748, 908
  ownership, 101–102
  pointer, 144
  regular, 95
  sharing, 74–77, 231
  size, 111–112
  times, 124–125, 532
  truncation, 112
  types, 95–98
FILE structure, 131, 143–144, 151, 164, 168, 171–172, 220, 235, 273, 443–444, 538, 542–543, 545, 547, 622, 701, 754, 803, 914, 929
file system, 4, 113–116
  devtmpfs, 139
  ext2, 129
  ext3, 129
  ext4, 73, 86, 129, 465
  HFS, 87, 113, 116
  HSFS, 113
  PCFS, 49, 57, 113
  S5, 65
  UFS, 49, 57, 65, 113, 116, 129
filename, 4
  truncation, 65–66
FILENAME_MAX constant, 38
fileno function, 164, 545, 701, 913
  definition of, 164
_FILE_OFFSET_BITS constant, 70
FILEPERM constant, 800, 825
files and directories, 4–8
FILESIZEBITS constant, 39, 44, 49
find program, 124, 135, 252
finger program, 141, 179, 910
FIOASYNC constant, 627, 939–940
FIOSETOWN constant, 627
FIPS, 32–33
Flandrena, B., 229, 952
<float.h> header, 27, 38
flock function, 485
flock structure, 486, 489–490, 494
flockfile function, 443–444
  definition of, 443
FLUSHO constant, 676, 680, 687
fmemopen function, 171–175, 913
  definition of, 171
fmtmsg function, 211, 452
<fmtmsg.h> header, 30
FNDELAY constant, 482
<fnmatch.h> header, 29
F_OK constant, 102
follow_link function, 48
fopen function, 6, 144, 148–150, 165, 220, 273, 452, 538–539, 542, 701, 803, 929
  definition of, 148
FOPEN_MAX constant, 38, 43
foreground process group, 296, 298, 300–303, 306, 311, 318–322, 369, 377, 680–682, 685, 689, 710, 719, 741, 944
foreground process group ID, 298, 303, 677
fork function, 11–13, 19, 23, 77, 228–237, 241–243, 245–249, 254, 260–261, 264–266, 269–272, 274–275, 277, 282, 286, 288, 290–292, 294, 296, 304, 307–308, 312, 326, 331, 334, 370–372, 381, 457–462, 466–469, 471, 479, 491–493, 498–500, 527, 533–539, 541, 544, 546, 550, 557, 565, 577, 585, 588, 618–619, 642, 653–655, 658–659, 669–670, 716, 721, 723–724, 726–728, 732, 739, 781, 922–923, 927–928, 930–931, 934, 937, 939, 948
  definition of, 229
fork1 function, 229
forkall function, 229
Fowler, G. S., 135, 949, 953
fpathconf function, 37, 39, 41–48, 53–55, 65, 110, 452, 537, 679
  definition of, 42

FPE_FLTDIV constant, 353
FPE_FLTINV constant, 353
FPE_FLTOVF constant, 353
FPE_FLTRES constant, 353
FPE_FLTSUB constant, 353
FPE_FLTUND constant, 353
FPE_INTDIV constant, 353
FPE_INTOVF constant, 353
fpos_t data type, 59, 157
fprintf function, 159, 452
  definition of, 159
fputc function, 145, 152, 154–155, 452
  definition of, 152
fputs function, 146, 150, 152–156, 164, 168, 174–175, 452, 543, 548, 550, 701, 901, 904, 911, 919, 926, 936
  definition of, 153
fputwc function, 452
fputws function, 452
F_RDLCK constant, 486–487, 489–490, 897, 930–931
fread function, 150, 156–157, 269, 273, 452
  definition of, 156
free function, 163, 174, 207–209, 330, 332, 401, 403–405, 407, 437–438, 450, 697, 762, 829, 833, 837, 842, 917
  definition of, 207
freeaddrinfo function, 599, 833
  definition of, 599
FreeBSD, xxi–xxii, xxvi–xxvii, 3–4, 21, 26–27, 29–30, 34–36, 38, 49, 57, 60, 62, 64, 68, 70, 81, 83, 88, 95, 102, 108–111, 121, 129, 132, 138, 175, 178, 182, 184–185, 187–188, 209–212, 222, 225, 229, 240, 245, 253, 257, 260, 262, 269, 271, 276–277, 288–289, 292, 298, 303, 310, 314–316, 319, 322, 329, 334, 351, 355, 358, 371, 373, 377, 379–380, 385, 388, 393, 396, 409, 426–427, 433, 439, 473, 485, 492–493, 497, 499, 503, 527, 534, 559, 561, 567, 572, 576, 594–595, 607, 611–613, 627, 634, 648–649, 652, 675–678, 685–691, 716, 724, 726–727, 740–741, 744, 799, 911, 918, 930, 932–933, 935–936, 949, 951
freopen function, 144, 148–150, 452
  definition of, 148
frequency scaling, 785
fscanf function, 162, 452
  definition of, 162
fsck program, 122
fseek function, 149, 157–159, 172, 452
  definition of, 158
fseeko function, 157–159, 172, 452
  definition of, 158
F_SETFD constant, 82, 85, 90, 480, 592, 907
F_SETFL constant, 82–83, 85, 90, 511, 592, 627, 907, 944
F_SETLK constant, 82, 486–488, 490, 494, 897, 930–931
F_SETLKW constant, 82, 486, 488, 490, 897, 931
F_SETOWN constant, 82–83, 510, 592, 626–627, 939
fsetpos function, 149, 157–159, 172, 452
  definition of, 158
fstat function, 4, 93–95, 120, 331, 452, 494, 498, 518, 529–530, 535, 586, 592, 698, 759, 808, 833
  definition of, 93
fstatat function, 93–95, 331, 452
  definition of, 93
fsync function, 61, 81, 86–87, 175, 331, 451, 513, 517, 528, 592, 787, 913
  definition of, 81
ftell function, 157–159, 452
  definition of, 158
ftello function, 157–159, 452
  definition of, 158
ftok function, 557–558
  definition of, 557
ftpd program, 472, 928
ftruncate function, 112, 125, 331, 529–530, 592
  definition of, 112
ftrylockfile function, 443–444
  definition of, 443
fts function, 132
ftw function, 122, 130–135, 141
<ftw.h> header, 30
full-duplex pipes, 534
  named, 534
  timing, 565
function prototypes, 845–893
functions, system calls versus, 21–23
F_UNLCK constant, 486–487, 489–490, 897
funlockfile function, 443–444
  definition of, 443
funopen function, 175, 915
futimens function, 125–128, 331, 452, 910
  definition of, 126
fwide function, 144
  definition of, 144
fwprintf function, 452
fwrite function, 150, 156–157, 382, 452, 925
  definition of, 156
F_WRLCK constant, 486–487, 489–490, 494, 897, 931
fwscanf function, 452

gai_strerror function, 600, 616, 619, 621, 623
  definition of, 600
Gallmeister, B. O., 949
Garfinkel, S., 181, 250, 298, 949
gather write, 521, 644
gawk program, 262
gcc program, 6, 26, 58, 919
gdb program, 928
gdbm library, 744
generic pointer, 71, 208
getaddrinfo function, 452, 599–601, 603–604, 614–616, 619, 621, 623, 802, 808
  definition of, 599
getaddrlist function, 800, 802, 804, 808, 815
  definition of, 802
GETALL constant, 568
getc function, 10, 150–156, 164–165, 452, 701–702, 913
  definition of, 150
getchar function, 150–151, 164, 175, 452, 547, 913
  definition of, 150
getchar_unlocked function, 442, 444, 452
  definition of, 444
getconf program, 70
getc_unlocked function, 442, 444, 452
  definition of, 444
getcwd function, 50, 135–137, 142, 208, 452, 911–912
  definition of, 136
getdate function, 211, 442, 452
getdelim function, 452
getegid function, 228, 331
  definition of, 228
getenv function, 204, 210–212, 442, 444–446, 449–450, 462, 539, 928
  definition of, 210
getenv_r function, 445–446
geteuid function, 228, 257, 268, 331, 650, 809
  definition of, 228
getgid function, 17, 228, 331
  definition of, 228
getgrent function, 183–184, 442, 452
  definition of, 183
getgrgid function, 182, 442, 452
  definition of, 182
getgrgid_r function, 443, 452
getgrnam function, 182, 442, 452
  definition of, 182
getgrnam_r function, 443, 452
getgroups function, 184, 331
  definition of, 184
gethostbyaddr function, 597, 599
gethostbyname function, 597, 599
gethostent function, 442, 452, 597
  definition of, 597
gethostid function, 452
gethostname function, 39–40, 43, 188, 452, 616–618, 623, 815
  definition of, 188
getline function, 452
getlogin function, 275–276, 442, 452, 480, 929–930
  definition of, 275
getlogin_r function, 443, 452
getmsg function, 740
getnameinfo function, 452, 600
  definition of, 600
GETNCNT constant, 568
getnetbyaddr function, 442, 452, 598
  definition of, 598
getnetbyname function, 442, 452, 598
  definition of, 598
getnetent function, 442, 452, 598
  definition of, 598
get_newjobno function, 814, 820, 825, 843
  definition of, 820
getopt function, 442, 452, 662–664, 669, 730–731, 807–808
  definition of, 662
getpass function, 287, 298, 700, 702–703
  definition of, 701
getpeername function, 331, 605
  definition of, 605
getpgid function, 293–294
  definition of, 294
getpgrp function, 293, 331
  definition of, 293
GETPID constant, 568
getpid function, 11, 228, 230, 235, 272, 308, 331, 366, 378, 387, 474, 650, 939
  definition of, 228
getppid function, 228–229, 331, 491, 732
  definition of, 228
get_printaddr function, 800, 804, 819
  definition of, 804
get_printserver function, 800, 804, 808
  definition of, 804
getpriority function, 277
  definition of, 277
getprotobyname function, 442, 452, 598
  definition of, 598
getprotobynumber function, 442, 452, 598
  definition of, 598
getprotoent function, 442, 452, 598
  definition of, 598
getpwent function, 180–181, 442, 452

definition of, 180
getpwnam function, 177–181, 186, 276, 287, 330–332, 442, 452, 816, 918
definition of, 179–180
getpwnam_r function, 443, 452
getpwuid function, 177–181, 186, 275–276, 442, 452, 809, 918
definition of, 179
getpwuid_r function, 443, 452
getresgid function, 257
getresuid function, 257
getrlimit function, 53, 220, 224, 466–467, 906–907
definition of, 220
getrusage function, 245, 280
gets function, 152–153, 911
definition of, 152
getservbyname function, 442, 452, 599
definition of, 599
getservbyport function, 442, 452, 599
definition of, 599
getservent function, 442, 452, 599
definition of, 599
getsid function, 296
definition of, 296
getsockname function, 331, 605
definition of, 605
getsockopt function, 331, 624–625
definition of, 624
getspent function, 182
definition of, 182
getspnam function, 182, 918
definition of, 182
gettimeofday function, 190, 414, 421, 437, 439
definition of, 190
getty program, 238, 286–288, 290, 472
gettytab file, 287
getuid function, 17, 228, 257, 268, 275–276, 331
definition of, 228
getutxent function, 442, 452
getutxid function, 442, 452
getutxline function, 442, 452
GETVAL constant, 568
getwc function, 452
getwchar function, 452
GETZCNT constant, 568
Ghemawat, S., 949
GID, *see* group ID
gid_t data type, 59
Gingell, R. A., 206, 525, 949
Gitlin, J. E., xxxii
glob function, 452
global variables, 219
<glob.h> header, 29
gmtime function, 191–192, 442
definition of, 192
gmtime_r function, 443
GNU, 2, 289, 753
GNU Public License, 35
Godsil, J. M., xxxii
Goodheart, B., 712, 949
Google, 210
goto, nonlocal, 213–220, 355–358
Grandi, S., xxxii
grantpt function, 723–725
definition of, 723
grep program, 20, 174, 200, 252, 949–950
group file, 182–183
group ID, 17, 255–260
effective, 98–99, 101–102, 108, 110, 140, 183, 228, 233, 256, 258, 558, 587
real, 98, 102, 183, 228, 233, 252–253, 256, 270, 585
supplementary, 18, 39, 98, 101, 108, 110, 183–184, 233, 252, 258
group structure, 182
<grp.h> header, 29, 182, 186
guardsize attribute, 427, 430

hack, 303, 842
half-duplex pipes, 534
handle_request function, 656, 665–666, 668
definition of, 657, 668
hard link, 4, 114, 117, 120, 122
hard links and directories, 117, 120
hcreate function, 442
hdestroy function, 442
headers
optional, 30
POSIX required, 29
standard, 27
XSI option, 30
heap, 205
Hein, T. R., xxxii, 951
Hewlett-Packard, 35, 835
HFS file system, 87, 113, 116
Hogue, J. E., xxxii
holes, file, 68–69, 111–112
home directory, 2, 8, 135, 211, 288, 292
HOME environment variable, 210–211, 288
Honeyman, P., xxxii
hostent structure, 597
hostname program, 189
HOST_NAME_MAX constant, 40, 43, 49, 188, 615–618, 622–623, 800, 815

HP-UX, 35
hsearch function, 442
HSFS file system, 113
htonl function, 594, 810, 824–827, 834
  definition of, 594
htons function, 594, 831, 834
  definition of, 594
HTTP (Hypertext Transfer Protocol), 792–793
Hume, A. G., 174, 949
HUPCL constant, 675, 687
Hypertext Transfer Protocol, *see* HTTP

IBM (International Business Machines), 35
ICANON constant, 676, 678, 680–682, 686–687, 691, 703, 705–707
iconv_close function, 452
<iconv.h> header, 29
iconv_open function, 452
ICRNL constant, 676, 680, 688, 700, 706–708
identifiers
  IPC, 556–558
  process, 227–228
IDXLEN_MAX constant, 779
IEC (International Electrotechnical Commission), 25
IEEE (Institute for Electrical and Electronic Engineers), xx, 26–27, 950
IEXTEN constant, 676, 678, 680–682, 688, 706–708
I_FIND constant, 725–726
IFS environment variable, 269
IGNBRK constant, 676, 685, 688
IGNCR constant, 676, 680, 688, 700
IGNPAR constant, 676, 688, 690
ILL_BADSTK constant, 353
ILL_COPROC constant, 353
ILL_ILLADR constant, 353
ILL_ILLOPC constant, 353
ILL_ILLOPN constant, 353
ILL_ILLTRP constant, 353
ILL_PRVOPC constant, 353
ILL_PRVREG constant, 353
Illumos, xxi
IMAXBEL constant, 676, 688
implementation differences, password, 184–185
implementations, UNIX System, 33
INADDR_ANY constant, 605
in_addr_t data type, 595
incore, 74, 152
INET6_ADDRSTRLEN constant, 596
inet_addr function, 596
INET_ADDRSTRLEN constant, 596, 603–604
inetd program, 291, 293, 465, 470, 472
inet_ntoa function, 442, 596
inet_ntop function, 596, 604
  definition of, 596
inet_pton function, 596
  definition of, 596
INFTIM constant, 508
init program, 187, 189, 228, 237–238, 246, 270, 286–291, 293, 307, 309, 312, 320, 337, 379, 464–465, 475, 923, 930
initgroups function, 184, 288
  definition of, 184
initialized data segment, 205
init_printer function, 814, 816, 819, 833
  definition of, 819
init_request function, 814, 816, 818
  definition of, 818
initserver function, 615–617, 619, 622–623, 800, 816
  definition of, 609, 625
inittab file, 320
INLCR constant, 676, 688
i-node, 59, 75–77, 94, 108, 113–116, 120, 124, 127, 130–131, 138–139, 179, 253, 493, 698, 905, 910
ino_t data type, 59, 114
INPCK constant, 676, 688, 690, 706–708
in_port_t data type, 595
Institute for Electrical and Electronic Engineers, *see* IEEE
int16_t data type, 831
Intel, xxii
International Business Machines, *see* IBM
International Electrotechnical Commission, *see* IEC
International Standards Organization, *see* ISO
Internet Printing Protocol, *see* IPP
Internet worm, 153
interpreter file, 260–264, 283
interprocess communication, *see* IPC
interrupted system calls, 327–330, 343, 351, 354–355, 365, 508
INT_MAX constant, 37–38
INT_MIN constant, 37–38
INTR terminal character, 678, 681, 688, 701
<inttypes.h> header, 27
I/O
  asynchronous, 501, 509–520
  asynchronous socket, 627
  efficiency, 72–74
  library, standard, 10, 143–175
  memory-mapped, 525–531
  multiplexing, 500–509
  nonblocking, 481–484
  nonblocking socket, 608–609, 627

terminal, 671–713
unbuffered, 8, 61–91
IOBUFSZ constant, 836
ioctl function, 61, 87–88, 90, 297–298, 322, 328–329, 452, 482, 510, 562, 592, 627, 674, 710–711, 718–719, 725–728, 730, 740–742, 939–940
definition of, 87
_IOFBF constant, 147
_IOLBF constant, 147, 166, 220
_IO_LINE_BUF constant, 165
_IONBF constant, 147, 166
_IO_UNBUFFERED constant, 165
iovec structure, 41, 43, 521, 611, 646–647, 649, 651, 655, 659, 765, 771–772, 832, 836
IOV_MAX constant, 41, 43, 49, 521
IPC (interprocess communication), 533–588, 629–670
identifiers, 556–558
key, 556–558, 562, 567, 572
XSI, 556–560
IPC_CREAT constant, 558, 632, 941
IPC_EXCL constant, 558
IPC_NOWAIT constant, 563–564, 569–570
ipc_perm structure, 558, 562, 567, 572, 587
IPC_PRIVATE constant, 557–558, 575, 586, 588
ipcrm program, 559
IPC_RMID constant, 562–563, 568, 573–575
ipcs program, 559, 588
IPC_SET constant, 562–563, 568, 573
IPC_STAT constant, 562–563, 568, 573
IPP (Internet Printing Protocol), 789–792
ipp.h header, 843
ipp_hdr structure, 798, 832, 834, 838, 842
IPPROTO_ICMP constant, 591
IPPROTO_IP constant, 591, 624
IPPROTO_IPV6 constant, 591
IPPROTO_RAW constant, 591, 602
IPPROTO_TCP constant, 591, 602, 624
IPPROTO_UDP constant, 591, 602
I_PUSH constant, 725–726
IRIX, 35
isalpha function, 516
isatty function, 679, 695, 698–699, 711, 730, 738
definition of, 695
isdigit function, 839–840
I_SETSIG constant, 510
ISIG constant, 676, 678, 680–682, 688, 706–708
ISO (International Standards Organization), xx, xxxi, 25–27, 950
ISO C, 25–26, 153, 950
<iso646.h> header, 27
is_read_lockable function, 490, 897
isspace function, 839–840
ISTRIP constant, 676, 688, 690, 706–708
is_write_lockable function, 490, 897
IUCLC constant, 676, 688
IUTF8 constant, 676, 689
IXANY constant, 676, 689
IXOFF constant, 676, 681–682, 689
IXON constant, 676, 681–682, 689, 706–708

jemalloc, 210
jmp_buf data type, 216, 218, 340, 343
job control, 299–303
shell, 294, 299, 306–307, 325, 358, 377, 379, 734–735
signals, 377–379
job structure, 812–813, 820–821, 832
job_append function, definition of, 411
job_find function, 927
definition of, 412
job_insert function, definition of, 411
job_remove function, 927
definition of, 412
Jolitz, W. F., 34
Joy, W. N., 3, 76
jsh program, 299

Karels, M. J., 33–34, 74, 112, 116, 229, 236, 525, 951
kernel, 1
Kernighan, B. W., xx, xxxii, 26, 149, 155, 162, 164, 208, 262, 898, 906, 947, 950
Kerrisk, M., 950
key, IPC, 556–558, 562, 567, 572
key_t data type, 557, 633
kill function, 18, 272, 308, 314, 325, 331, 335–338, 353, 363, 366–367, 376, 378–379, 381, 455, 679, 681, 702, 732–733, 924, 932
definition of, 337
kill program, 314–315, 321, 325, 551
KILL terminal character, 678, 681, 687, 702–703
kill_workers function, 814, 828–830
definition of, 828
Kleiman, S. R., 76, 950
Knuth, D. E., 422, 764, 950
Korn, D. G., 3, 135, 174, 548, 948–950, 953
Korn shell, 3, 53, 90, 210, 222, 289, 299, 497, 548, 702, 733–734, 737, 935, 948
Kovach, K. R., 560, 947
Krieger, O., 174, 531, 950

l64a function, 442
LANG environment variable, 41, 211
<langinfo.h> header, 29
last program, 187
launchctl program, 293
launchd program, 228, 259, 289, 292, 465
layers, shell, 299
LC_ALL environment variable, 211
LC_COLLATE environment variable, 43, 211
LC_CTYPE environment variable, 211
lchown function, 109–110, 121, 125
  definition of, 109
LC_MESSAGES environment variable, 211
LC_MONETARY environment variable, 211
LC_NUMERIC environment variable, 211
L_ctermid constant, 694
LC_TIME environment variable, 211
ld program, 206
LDAP (Lightweight Directory Access Protocol), 185
LD_LIBRARY_PATH environment variable, 753
ldterm STREAMS module, 716, 726
leakage, memory, 209
least privilege, 256, 795, 816
Lee, M., 206, 949
Lee, T. P., 948
Leffler, S. J., 34, 951
Lennert, D., 951
Lesk, M. E., 143
lgamma function, 442
lgammaf function, 442
lgammal function, 442
Libes, D., 720, 924, 951
<libgen.h> header, 30
libraries, shared, 206–207, 226, 753, 920, 947
Lightweight Directory Access Protocol, *see* LDAP
limit program, 53, 222
limits, 36–53
  C, 37–38
  POSIX, 38–41
  resource, 220–225, 233, 252, 322, 382
  runtime indeterminate, 49–53
  XSI, 41
<limits.h> header, 27, 37, 39, 41, 49–50
Linderman, J. P., xxxii
line control, terminal I/O, 693–694
LINE_MAX constant, 39, 43, 49
LINES environment variable, 211
link
  count, 44, 59, 114–117, 130
  hard, 4, 114, 117, 120, 122
  symbolic, 55, 94–95, 110–111, 114, 118, 120–123, 131, 137, 141, 186, 908–909
link function, 79, 115–119, 121–122, 125, 331, 452
  definition of, 116
linkat function, 116–119, 331, 452
  definition of, 116
LINK_MAX constant, 39, 44, 49, 114
lint program, 200
Linux, xxi–xxii, xxv, xxvii, 2–4, 7, 14, 21, 26–27, 29–30, 35–38, 49, 52, 57, 60, 62, 64–65, 70, 73, 75–76, 86–89, 102, 108–111, 121–122, 129, 132, 138, 173, 178, 182, 184–185, 187–188, 205, 209, 211–212, 222, 226, 229, 240, 244–245, 253, 257, 259–260, 262, 269, 271, 274, 276–277, 288–290, 293, 298, 303, 306, 314–316, 318–320, 322, 329, 334–335, 351, 354–355, 358, 371, 373, 377, 379–380, 385, 388, 392, 396, 409, 426–427, 432–433, 439, 462, 464–465, 473–474, 485, 496–497, 503, 522, 530–531, 534, 559, 561, 567, 571–573, 575–576, 578, 583, 594–596, 607, 611–613, 627, 634, 648–650, 652, 675–678, 684–691, 693, 716, 724, 726–727, 740–741, 744, 753, 783, 793, 799, 911, 918, 925, 930, 932, 935–936
Linux Fast-STREAMS, 534
LinuxThreads, 388
lio_listio function, 452, 515
  definition of, 515
LIO_NOWAIT constant, 515
Lions, J., 951
LIO_WAIT constant, 515
listen function, 331, 605, 608–609, 625, 635, 638, 800
  definition of, 608
little-endian byte order, 593
Litwin, W., 744, 750, 951
LLONG_MAX constant, 37
LLONG_MIN constant, 37
ln program, 115
LNEXT terminal character, 678, 681
locale, 43
localeconv function, 442
<locale.h> header, 27
localtime function, 190–192, 194–195, 264, 408, 442, 452, 919
  definition of, 192
localtime_r function, 443, 452
lockf function, 451–452, 485
lockf structure, 493
lockfile function, 473–474
  definition of, 494
locking
  database library, coarse-grained, 752
  database library, fine-grained, 752
locking function, 485

lock_reg function, 489, 897, 930–931
  definition of, 489
locks
  reader–writer, 409–413
  spin, 417–418
lock_test function, 489–490, 897
  definition of, 489
log function, 470
LOG_ALERT constant, 472
LOG_AUTH constant, 472
LOG_AUTHPRIV constant, 472
LOG_CONS constant, 468, 471
LOG_CRIT constant, 472
LOG_CRON constant, 472
LOG_DAEMON constant, 468, 472
LOG_DEBUG constant, 472
LOG_EMERG constant, 472
LOG_ERR constant, 472, 474–476, 478–479, 615–619, 622–623, 902–903
log_exit function, 817, 898–899
  definition of, 903
LOG_FTP constant, 472
logger program, 471
login accounting, 186–187
.login file, 289
login name, 2, 17, 135, 179, 187, 211, 275–276, 290, 480, 930
  root, 16
login program, 179, 182, 184, 187, 251, 254, 256, 276, 287–290, 292, 472, 700, 717, 738
LOG_INFO constant, 472, 476, 478
LOGIN_NAME_MAX constant, 40, 43, 49
logins
  network, 290–293
  terminal, 285–290
LOG_KERN constant, 472
LOG_LOCAL0 constant, 472
LOG_LOCAL1 constant, 472
LOG_LOCAL2 constant, 472
LOG_LOCAL3 constant, 472
LOG_LOCAL4 constant, 472
LOG_LOCAL5 constant, 472
LOG_LOCAL6 constant, 472
LOG_LOCAL7 constant, 472
LOG_LPR constant, 472
LOG_MAIL constant, 472
log_msg function, 897, 899
  definition of, 903
LOGNAME environment variable, 211, 276, 288
LOG_NDELAY constant, 471, 928
LOG_NEWS constant, 472
LOG_NOTICE constant, 472
log_open function, 664, 898
  definition of, 902
LOG_PERROR constant, 471
LOG_PID constant, 471, 664
log_quit function, 830, 898–899
  definition of, 903
log_ret function, 898–899
  definition of, 902
log_sys function, 804, 898–899
  definition of, 902
LOG_SYSLOG constant, 472
log_to_stderr variable, 664, 807, 813, 902, 904
LOG_USER constant, 472, 664
LOG_WARNING constant, 472
LONG_BIT constant, 38
_longjmp function, 355, 358
longjmp function, 197, 213, 215–219, 225, 330–331, 340–341, 343, 355–358, 365, 381, 924
  definition of, 215
LONG_MAX constant, 37, 52–53, 60, 420, 906–907
LONG_MIN constant, 37
loop function, 663–664, 666, 670, 732, 742
  definition of, 666, 732
lp program, 585, 793
lpc program, 472
lpd program, 472, 793
lpsched program, 585, 793
lrand48 function, 442
ls program, 5–8, 13, 107–108, 112, 123, 125, 131, 135, 139, 141, 177, 179, 559, 905
lseek function, 8, 59, 61, 66–70, 77–79, 88, 91, 149, 158, 331, 452, 462, 486, 489, 498, 592, 670, 765–766, 768, 771, 773, 779, 819, 908
  definition of, 67
lstat function, 93–97, 121–122, 133, 141, 331, 452, 942
  definition of, 93
L_tmpnam constant, 168
Lucchina, P., xxxii

Mac OS X, xxi–xxii, xxvi–xxvii, 3–4, 17, 26–27, 29–30, 35–36, 38, 49, 57, 60, 62, 64, 70, 83, 87–88, 102, 108–111, 113, 121, 129, 132, 138, 175, 178, 182, 184–185, 187–188, 193, 209, 211–212, 222, 228, 240, 244–245, 260, 262, 269, 271, 276–277, 288–289, 292–293, 298, 303, 314–317, 319, 322, 329, 334, 351, 355, 371, 373, 377, 379–380, 385, 388, 393, 396, 409, 426–427, 464–465, 485, 497, 503, 522, 534, 559, 561, 567, 572, 576, 594, 607, 611–613, 627, 634, 648, 675–678, 685–691, 716, 724, 726–727, 740–741, 744, 793, 799, 911, 918, 925, 930, 932, 935–936

Mach, xxii, xxvi–xxvii, 35, 947
<machine/_types.h> header, 906
macro, feature test, 57–58, 84
MAILPATH environment variable, 210
main function, 7, 150, 155, 197–200, 202, 204, 215–217, 226, 236–237, 249, 283, 330–332, 357–358, 468, 654, 656, 663, 729, 739, 811, 814, 817, 824, 830, 833, 919, 921, 939, 944
major device number, 58–59, 137, 139, 465, 699
major function, 138–139
make program, 300
makethread function, 436, 438–439
mallinfo function, 209
malloc function, 21–23, 51, 136, 145, 174, 207–210, 213, 330, 332, 392, 400–401, 403, 405, 429, 437, 447, 450, 575, 616, 618, 623, 646–647, 650–651, 661–662, 666, 696, 760–761, 815, 820, 828, 839, 926, 928
  definition of, 207
MALLOC_OPTIONS environment variable, 928
mallopt function, 209
mandatory record locking, 495
Mandrake, xxvii
MAP_ANON constant, 578
MAP_ANONYMOUS constant, 578
MAP_FAILED constant, 529, 577
MAP_FIXED constant, 526–527
MAP_PRIVATE constant, 526, 528, 578
MAP_SHARED constant, 526–529, 576–578
<math.h> header, 27
Mauro, J., 74, 112, 116, 951
MAX_CANON constant, 39, 44, 47, 49, 673
MAX_INPUT constant, 39, 44, 49, 672
MAXPATHLEN constant, 49
MB_LEN_MAX constant, 37
mbstate_t structure, 442
McDougall, R., 74, 112, 116, 951
McIlroy, M. D., xxxii
McKusick, M. K., xxxii, 33–34, 74, 112, 116, 229, 236, 525, 951
MD5, 181
MDMBUF constant, 675, 685, 689
memccpy function, 155
memcpy function, 530–531, 916
memory
  allocation, 207–210
  layout, 204–206
  leakage, 209
  shared, 534, 571–578
memory-mapped I/O, 525–531
memset function, 172–173, 614, 616, 618, 621, 623
Menage, P., 949
message queues, 534, 561–565
  timing, 565
mgetty program, 290
MIN terminal value, 687, 703–704, 708, 713, 943
minor device number, 58–59, 137, 139, 465, 699
minor function, 138–139
mkdir function, 101–102, 120–122, 125, 129–130, 331, 452, 912
  definition of, 129
mkdir program, 129
mkdirat function, 129–130, 331, 452
  definition of, 129
mkdtemp function, 167–171, 452
  definition of, 169
mkfifo function, 120–121, 125, 331, 452, 553, 937
  definition of, 553
mkfifo program, 553
mkfifoat function, 331, 452, 553
  definition of, 553
mknod function, 120–121, 129, 331, 452, 553
mknodat function, 331, 452, 553
mkstemp function, 167–171, 452
  definition of, 169
mktime function, 190, 192, 195, 452
  definition of, 192
mlock function, 221
mmap function, 174, 221, 429, 481, 525, 527, 529–532, 576–578, 587, 592, 949
  definition of, 525
modem, xx, xxvii, 285, 287, 297, 318, 328, 481, 508, 671, 674–675, 685, 687, 689, 692
mode_t data type, 59
<monetary.h> header, 29
Moran, J. P., 525, 949
more program, 543, 748
Morris, R., 181, 951
mount program, 102, 129, 139, 496
mounted STREAMS-based pipes, 534
mprotect function, 527
  definition of, 527
mq_receive function, 451
mq_send function, 451
mq_timedreceive function, 451
mq_timedsend function, 451
<mqueue.h> header, 30
mrand48 function, 442
MS_ASYNC constant, 528
MSG_CONFIRM constant, 611
msgctl function, 558–559, 562
  definition of, 562
MSG_CTRUNC constant, 613
MSG_DONTROUTE constant, 611
MSG_DONTWAIT constant, 611
MSG_EOF constant, 611

MSG_EOR constant, 611, 613
msgget function, 557–562, 632–633, 941
  definition of, 562
msghdr structure, 611, 613, 644, 646–647, 649, 651
MSG_MORE constant, 611
MSG_NOERROR constant, 564, 631, 941
MSG_NOSIGNAL constant, 611
MSG_OOB constant, 611–613, 626
MSG_PEEK constant, 612
msgrcv function, 451, 558–559, 561, 564, 585, 631, 941
  definition of, 564
msgsnd function, 451, 558, 560–561, 563–565, 633
  definition of, 563
MSG_TRUNC constant, 612–613
MSGVERB environment variable, 211
MSG_WAITALL constant, 612
MS_INVALIDATE constant, 528
msqid_ds structure, 561–562, 564
MS_SYNC constant, 528, 530
msync function, 451, 528, 530
  definition of, 528
Mui, L., 712, 953
multiplexing, I/O, 500–509
munmap function, 528–529
  definition of, 528
mutex attributes, 430–439
mutex timing comparison, 571
mutexes, 399–409
mv program, 115
myftw function, 133, 141

named full-duplex pipes, 534
NAME_MAX constant, 38–39, 44, 49, 55, 65, 131
nanosleep function, 373–375, 437, 439, 451, 462, 837, 934
  definition of, 374
Nataros, S., xxxii
Native POSIX Threads Library, *see* NPTL
nawk program, 262
NCCS constant, 674
ndbm library, 744
<ndbm.h> header, 30
Nemeth, E., xxxii, 951
<netdb.h> header, 29, 186
netent structure, 598
<net/if.h> header, 29
<netinet/in.h> header, 29, 595, 605
<netinet/tcp.h> header, 29
Network File System, Sun Microsystems, *see* NFS
Network Information Service, *see* NIS
network logins, 290–293
network printer communication, 789–843
Neville-Neil, G. V., 74, 112, 116, 951
newgrp program, 183
nfds_t data type, 507
_NFILE constant, 51
NFS (Network File System, Sun Microsystems), 76, 787
nftw function, 122, 131–132, 135, 442, 452, 910
NGROUPS_MAX constant, 39, 43, 49, 183–184
nice function, 276–277
  definition of, 276
nice value, 252, 276–277, 279
Nievergelt, J., 744, 750, 949
NIS (Network Information Service), 185
NIS+, 185
NL terminal character, 678, 680–681, 687, 700, 703
NL0 constant, 689
NL1 constant, 689
NL_ARGMAX constant, 39
NLDLY constant, 676, 684, 689
nlink_t data type, 59, 114
nl_langinfo function, 442
NL_LANGMAX constant, 41
NL_MSGMAX constant, 39
NL_SETMAX constant, 39
NLSPATH environment variable, 211
NL_TEXTMAX constant, 39
<nl_types.h> header, 29
nobody login name, 178–179
NOFILE constant, 51
NOFLSH constant, 676, 689
NOKERNINFO constant, 676, 682, 689
nologin program, 179
nonblocking
  I/O, 481–484
  socket I/O, 608–609, 627
noncanonical mode, terminal I/O, 703–710
nonfatal error, 16
nonlocal goto, 213–220, 355–358
NPTL (Native POSIX Threads Library), xxiii, 388
ntohl function, 594, 811, 825, 842
  definition of, 594
ntohs function, 594, 604, 842
  definition of, 594
NULL constant, 823
null signal, 314, 337
NZERO constant, 41, 276–277

O_ACCMODE constant, 83–84
O_APPEND constant, 63, 66, 72, 77–78, 83–84, 149, 497, 511

O_ASYNC constant, 83, 511, 627
O_CLOEXEC constant, 63
O_CREAT constant, 63, 66, 79, 89, 121, 125, 474, 496–498, 517–518, 529, 558, 579–580, 584, 749, 758, 818, 930
OCRNL constant, 676, 689
od program, 69
O_DIRECT constant, 150
O_DIRECTORY constant, 63
O_DSYNC constant, 64, 83, 513
O_EXCL constant, 63, 79, 121, 558, 580, 584
O_EXEC constant, 83
OFDEL constant, 676, 684, 689
off_t data type, 59, 67–70, 157–158, 772
OFILL constant, 676, 684, 689
O_FSYNC constant, 64, 83–84
OLCUC constant, 676, 689
Olson, M., 952
O_NDELAY constant, 36, 63, 482
ONLCR constant, 676, 690, 731, 738
ONLRET constant, 676, 690
ONOCR constant, 676, 690
O_NOCTTY constant, 63, 297–298, 466, 723–724, 726
ONOEOT constant, 676, 690
O_NOFOLLOW constant, 63
O_NONBLOCK constant, 36, 63, 83–84, 482–483, 496, 498, 553, 611–612, 934, 937
open function, 8, 14, 61–66, 77, 79, 83, 89, 91, 100–101, 103–104, 112, 118, 120–125, 127–128, 137, 148–150, 283, 287, 297–298, 331, 451, 468, 470, 474, 482, 492–493, 495–498, 517–518, 525, 529, 553, 556, 558, 560, 577–578, 585, 588, 592, 653, 656–657, 669–670, 685, 723, 725–726, 745, 757–758, 808, 818, 823, 833, 907, 909, 930, 937
  definition of, 62
Open Group, The, xxi, xxvi, 31, 196, 950
Open Software Foundation, *see* OSF
openat function, 62–66, 331, 451
  definition of, 62
opend.h header, 656, 660, 942
opendir function, 5, 7, 121, 130–135, 252–253, 283, 452, 697, 822, 910
  definition of, 130
openlog function, 452, 468, 470–471, 480, 902, 928
  definition of, 470
OPEN_MAX constant, 40, 43, 49, 51–53, 60, 62, 906
open_max function, 466, 544, 546, 666, 896
  definition of, 52, 907
open_memstream function, 171–174
  definition of, 173
OpenServer, 485
OpenSolaris, xxi
OpenSS7, 534
open_wmemstream function, 171–174
  definition of, 173
OPOST constant, 676, 690, 706–708, 710
optarg variable, 663
opterr variable, 663
optind variable, 808
option codes, 31
options, 53–57
  socket, 623–625
optopt variable, 663
Oracle Corporation, xxi–xxii, 35
O_RDONLY constant, 62, 83–84, 100, 103, 517–518, 529, 654, 808, 833, 937
O_RDWR constant, 62, 83–84, 100, 128, 468, 474, 498, 517–518, 529, 577, 723, 725, 749, 818, 930
O'Reilly, T., 712, 953
orientation, stream, 144
orphaned process group, 307–309, 469, 735
O_RSYNC constant, 64, 83
O_SEARCH constant, 63, 83
OSF (Open Software Foundation), 31–32
O_SYNC constant, 63–64, 83–84, 86–87, 513, 520
O_TRUNC constant, 63, 66, 100, 112, 125, 127–128, 149, 496, 498, 517–518, 529, 749
O_TTY_INIT constant, 64, 683, 722
out-of-band data, 626
ownership
  directory, 101–102
  file, 101–102
O_WRONLY constant, 62, 83–84, 100, 937
OXTABS constant, 676, 690

packet mode, pseudo terminal, 740
page cache, 81
page size, 573
pagedaemon process, 228
PAGER environment variable, 539, 542–543
PAGESIZE constant, 40, 43, 49
PAGE_SIZE constant, 41, 43, 49
P_ALL constant, 244
PARENB constant, 675, 688, 690, 706–708
parent
  directory, 4, 108, 125, 129
  process ID, 228, 233, 237, 243, 246, 252, 287–288, 309, 464
PAREXT constant, 675, 690
parity, terminal I/O, 688
PARMRK constant, 676, 685, 688, 690
PARODD constant, 675, 685, 688, 690, 713

Partridge, C., xxxii
passing, file descriptor, 587, 642–652
passwd program, 99, 182, 720
passwd structure, 177, 180, 332, 809, 814, 918
password
  file, 177–181
  implementation differences, 184–185
  shadow, 181–182, 196, 918
PATH environment variable, 100, 211, 250–251, 253, 260, 263, 265, 288–289
path_alloc function, 133, 137, 896, 912
  definition of, 50
pathconf function, 37, 39, 41–48, 50–51, 53–55, 57, 65, 110, 121, 452, 537
  definition of, 42
PATH_MAX constant, 38–39, 44, 49–50, 142, 911
pathname, 5
  absolute, 5, 8, 43, 50, 64, 136, 141–142, 260, 553, 911
  relative, 5, 8, 43–44, 50, 64–65, 135, 553
  truncation, 65–66
pause function, 324, 327–328, 331, 334, 338–343, 356, 359, 365, 374, 451, 460, 711, 924, 930–931
  definition of, 338
_PC_2_SYMLINKS constant, 55
_PC_ASYNC_IO constant, 55
_PC_CHOWN_RESTRICTED constant, 55
_PC_FILESIZEBITS constant, 42, 44
PCFS file system, 49, 57, 113
pckt STREAMS module, 716, 740
_PC_LINK_MAX constant, 42, 44
pclose function, 267, 452, 541–548, 616, 622, 935–937
  definition of, 541, 545
_PC_MAX_CANON constant, 42, 44, 47
_PC_MAX_INPUT constant, 42, 44
_PC_NAME_MAX constant, 42, 44
_PC_NO_TRUNC constant, 55, 57
_PC_PATH_MAX constant, 43–44, 51
_PC_PIPE_BUF constant, 44
_PC_PRIO_IO constant, 55
_PC_SYMLINK_MAX constant, 44
_PC_SYNC_IO constant, 55
_PC_TIMESTAMP_RESOLUTION constant, 42, 44
_PC_VDISABLE constant, 54–55, 679
PENDIN constant, 676, 690
Pentium, xxii, xxvii
permissions, file access, 99–101, 140
perror function, 15–16, 24, 334, 379, 452, 600, 905
  definition of, 15
pgrp structure, 311–312
PID, *see* process ID
pid_t data type, 11, 59, 293, 384
Pike, R., 229, 950, 952
pipe function, 125, 148, 331, 535, 537–538, 540, 544, 546, 550, 565, 630, 934
  definition of, 535
PIPE_BUF constant, 39, 44, 49, 532, 537, 554–555, 935
pipes, 534–541
  full-duplex, 534
  half-duplex, 534
  mounted STREAMS-based, 534
  named full-duplex, 534
  timing full-duplex, 565
Pippenger, N., 744, 750, 949
Plan 9 operating system, 229, 952
Plauger, P. J., 26, 164, 323, 952
pointer, generic, 71, 208
poll function, 319, 330–331, 343, 451, 481, 501–502, 506–509, 531–532, 560, 586, 588, 592, 608–609, 627, 631–632, 659, 664, 666–668, 718, 732, 742, 933–934, 936–937, 942
  definition of, 506
POLLERR constant, 508
pollfd structure, 507, 632, 666, 668, 934, 941
<poll.h> header, 29, 507
POLLHUP constant, 508, 667–668, 936
POLLIN constant, 508, 632, 666–668, 936, 941–942
polling, 246, 484, 501
POLLNVAL constant, 508
POLLOUT constant, 508
POLLPRI constant, 508
POLLRDBAND constant, 508
POLLRDNORM constant, 508
POLLWRBAND constant, 508
POLLWRNORM constant, 508
popen function, 23, 242, 249, 267, 452, 541–548, 587–588, 615, 619, 622–623, 935–937
  definition of, 541, 543
port number, 593, 595–596, 598–601, 605
Portable Operating System Environment for Computer Environments, IEEE, *see* POSIX
POSIX (Portable Operating System Environment for Computer Environments, IEEE), xix, xxxi, 26–30, 33, 265, 561, 674
POSIX semaphores, 579–584
POSIX.1, xxvi, xxxi, 4, 9, 27, 38, 41, 50, 53, 57–58, 88, 257, 262, 329, 367–368, 384, 533, 546, 553, 589, 617, 744, 950
POSIX.2, 262
_POSIX2_SYMLINKS constant, 55
_POSIX_ADVISORY_INFO constant, 31
_POSIX_AIO_LISTIO_MAX constant, 515
_POSIX_AIO_MAX constant, 515
_POSIX_ARG_MAX constant, 39–40

_POSIX_ASYNCHRONOUS_IO constant, 54, 57
_POSIX_ASYNC_IO constant, 55
_POSIX_BARRIERS constant, 54, 57
_POSIX_CHILD_MAX constant, 39–40
_POSIX_CHOWN_RESTRICTED constant, 55, 57, 110
_POSIX_CLOCKRES_MIN constant, 38
_POSIX_CLOCK_SELECTION constant, 54, 57
_POSIX_CPUTIME constant, 31, 189
_POSIX_C_SOURCE constant, 57–58, 84, 240
_POSIX_DELAYTIMER_MAX constant, 39–40
posix_fadvise function, 452
posix_fallocate function, 452
_POSIX_FSYNC constant, 31
_POSIX_HOST_NAME_MAX constant, 39–40
_POSIX_IPV6 constant, 31
_POSIX_JOB_CONTROL constant, 57
_POSIX_LINK_MAX constant, 39
_POSIX_LOGIN_NAME_MAX constant, 39–40
POSIXLY_CORRECT environment variable, 111
posix_madvise function, 452
_POSIX_MAPPED_FILES constant, 54, 57
_POSIX_MAX_CANON constant, 39
_POSIX_MAX_INPUT constant, 39
_POSIX_MEMLOCK constant, 31
_POSIX_MEMLOCK_RANGE constant, 31
_POSIX_MEMORY_PROTECTION constant, 54, 57
_POSIX_MESSAGE_PASSING constant, 31
_POSIX_MONOTONIC_CLOCK constant, 31, 189
_POSIX_NAME_MAX constant, 39, 580
_POSIX_NGROUPS_MAX constant, 39
_POSIX_NO_TRUNC constant, 55, 57, 65
_POSIX_OPEN_MAX constant, 39–40
posix_openpt function, 452, 722–725
  definition of, 722
_POSIX_PATH_MAX constant, 39–40, 696–697
_POSIX_PIPE_BUF constant, 39
_POSIX_PRIO_IO constant, 55
_POSIX_PRIORITIZED_IO constant, 31
_POSIX_PRIORITY_SCHEDULING constant, 31
_POSIX_RAW_SOCKETS constant, 31
_POSIX_READER_WRITER_LOCKS constant, 55, 57
_POSIX_REALTIME_SIGNALS constant, 55, 57
_POSIX_RE_DUP_MAX constant, 39
_POSIX_RTSIG_MAX constant, 39–40
_POSIX_SAVED_IDS constant, 57, 98, 256, 337
_POSIX_SEMAPHORES constant, 55, 57
_POSIX_SEM_NSEMS_MAX constant, 39–40
_POSIX_SEM_VALUE_MAX constant, 39–40
_POSIX_SHARED_MEMORY_OBJECTS constant, 31
_POSIX_SHELL constant, 57
_POSIX_SIGQUEUE_MAX constant, 39–40
_POSIX_SOURCE constant, 57
_POSIX_SPAWN constant, 31
posix_spawn function, 452
posix_spawnp function, 452
_POSIX_SPIN_LOCKS constant, 55, 57
_POSIX_SPORADIC_SERVER constant, 31
_POSIX_SSIZE_MAX constant, 39
_POSIX_STREAM_MAX constant, 39–40
_POSIX_SYMLINK_MAX constant, 39
_POSIX_SYMLOOP_MAX constant, 39–40
_POSIX_SYNCHRONIZED_IO constant, 31
_POSIX_SYNC_IO constant, 55
_POSIX_THREAD_ATTR_STACKADDR constant, 31, 429
_POSIX_THREAD_ATTR_STACKSIZE constant, 31, 429
_POSIX_THREAD_CPUTIME constant, 31, 189
_POSIX_THREAD_PRIO_INHERIT constant, 31
_POSIX_THREAD_PRIO_PROTECT constant, 31
_POSIX_THREAD_PRIORITY_SCHEDULING constant, 31
_POSIX_THREAD_PROCESS_SHARED constant, 31, 431
_POSIX_THREAD_ROBUST_PRIO_INHERIT constant, 31
_POSIX_THREAD_ROBUST_PRIO_PROTECT constant, 31
_POSIX_THREADS constant, 55, 57, 384
_POSIX_THREAD_SAFE_FUNCTIONS constant, 55, 57, 442
_POSIX_THREAD_SPORADIC_SERVER constant, 31
_POSIX_TIMEOUTS constant, 55
_POSIX_TIMER_MAX constant, 39–40
_POSIX_TIMERS constant, 55, 57
_POSIX_TIMESTAMP_RESOLUTION constant, 44
posix_trace_event function, 331
_POSIX_TTY_NAME_MAX constant, 39–40
posix_typed_mem_open function, 452
_POSIX_TYPED_MEMORY_OBJECTS constant, 31
_POSIX_TZNAME_MAX constant, 39–40
_POSIX_V6_ILP32_OFF32 constant, 70
_POSIX_V6_ILP32_OFFBIG constant, 70
_POSIX_V6_LP64_OFF64 constant, 70
_POSIX_V6_LP64_OFFBIG constant, 70
_POSIX_V7_ILP32_OFF32 constant, 70
_POSIX_V7_ILP32_OFFBIG constant, 70
_POSIX_V7_LP64_OFF64 constant, 70
_POSIX_V7_LP64_OFFBIG constant, 70
_POSIX_VDISABLE constant, 55, 57, 678–679
_POSIX_VERSION constant, 57, 188
PowerPC, xxi–xxii, xxvii
P_PGID constant, 244

PPID, *see* parent process ID
P_PID constant, 244
pr program, 753
prctl program, 559
pread function, 78, 451, 461–462, 592
  definition of, 78
Presotto, D. L., xxxii, 229, 952
pr_exit function, 239–241, 266–268, 281, 283, 372, 896
  definition of, 240
primitive system data types, 58
print program, 794, 801, 820, 824–825, 834, 843
printd program, 794, 843
printer communication, network, 789–843
printer spooling, 793–795
  source code, 795–842
printer_status function, 814, 837–838, 843
  definition of, 838
printer_thread function, 814, 832, 945
  definition of, 832
printf function, 10–11, 21, 150, 159, 161–163, 175, 192, 194, 219, 226, 231, 235, 283, 309, 330, 349, 452, 552, 919–920
  definition of, 159
print.h header, 815, 820, 825
printreq structure, 801, 809–810, 812, 820, 822–824, 827
printresp structure, 801, 809, 811, 824–827
PRIO_PGRP constant, 277
PRIO_PROCESS constant, 277
PRIO_USER constant, 277
privilege, least, 256, 795, 816
pr_mask function, 356–357, 360–361, 896
  definition of, 347
/proc, 136, 253
proc structure, 311–312
process, 11
  accounting, 269–275
  control, 11, 227–283
  ID, 11, 228, 252
  ID, parent, 228, 233, 237, 243, 246, 252, 287–288, 309, 464
  identifiers, 227–228
  relationships, 285–312
  scheduling, 276–280
  system, 228, 337
  termination, 198–202
  time, 20, 24, 59, 280–282
process group, 293–294
  background, 296, 300, 302, 304, 306–307, 309, 321, 369, 377, 944
  foreground, 296, 298, 300–303, 306, 311, 318–322, 369, 377, 680–682, 685, 689, 710, 719, 741, 944
  ID, 233, 252
  ID, foreground, 298, 303, 677
  ID, session, 304
  ID, terminal, 303, 463
  leader, 294–296, 306, 312, 465–466, 727
  lifetime, 294
  orphaned, 307–309, 469, 735
processes, cooperating, 495, 752, 945
process-shared attribute, 431
.profile file, 289
program, 10
PROT_EXEC constant, 525
PROT_NONE constant, 525
protoent structure, 598
prototypes, function, 845–893
PROT_READ constant, 525, 529, 577
PROT_WRITE constant, 525, 529, 577
PR_TEXT constant, 801, 810, 825, 835–836
ps program, 237, 283, 303, 306–307, 463–465, 468–469, 480, 736, 923
pselect function, 331, 451, 501, 506
  definition of, 506
pseudo terminal, 715–742
  packet mode, 740
  remote mode, 741
  signal generation, 741
  window size, 741
psiginfo function, 379–380, 452
  definition of, 379
psignal function, 379–380, 452
  definition of, 379
ptem STREAMS module, 716, 726
pthread structure, 385
pthread_atfork function, 457–461
  definition of, 458
pthread_attr_destroy function, 427–429
  definition of, 427
pthread_attr_getdetachstate function, 428
  definition of, 428
pthread_attr_getguardsize function, 430
  definition of, 430
pthread_attr_getstack function, 429
  definition of, 429
pthread_attr_getstacksize function, 429–430
  definition of, 430
pthread_attr_init function, 427–429
  definition of, 427
pthread_attr_setdetachstate function, 428
  definition of, 428
pthread_attr_setguardsize function, 430
  definition of, 430
pthread_attr_setstack function, 429
  definition of, 429

pthread_attr_setstacksize function, 429–430
  definition of, 430
pthread_attr_t data type, 427–428, 430, 451
pthread_barrierattr_destroy function, 441
  definition of, 441
pthread_barrierattr_getpshared function, 441
  definition of, 441
pthread_barrierattr_init function, 441
  definition of, 441
pthread_barrierattr_setpshared function, 441
  definition of, 441
pthread_barrier_destroy function, 418–419
  definition of, 418
pthread_barrier_init function, 418–419, 421
  definition of, 418
PTHREAD_BARRIER_SERIAL_THREAD constant, 419, 422
pthread_barrier_t data type, 419
pthread_barrier_wait function, 419–423
  definition of, 419
pthread_cancel function, 393, 451, 453, 828
  definition of, 393
PTHREAD_CANCEL_ASYNCHRONOUS constant, 453
PTHREAD_CANCEL_DEFERRED constant, 453
PTHREAD_CANCEL_DISABLE constant, 451
PTHREAD_CANCELED constant, 389, 393
PTHREAD_CANCEL_ENABLE constant, 451
pthread_cleanup_pop function, 394–396, 827, 829
  definition of, 394
pthread_cleanup_push function, 394–396, 824
  definition of, 394
pthread_condattr_destroy function, 440
  definition of, 440
pthread_condattr_getclock function, 441
  definition of, 441
pthread_condattr_getpshared function, 440
  definition of, 440
pthread_condattr_init function, 440
  definition of, 440
pthread_condattr_setclock function, 441
  definition of, 441
pthread_condattr_setpshared function, 440
  definition of, 440
pthread_condattr_t data type, 441
pthread_cond_broadcast function, 415, 422–423, 927
  definition of, 415
pthread_cond_destroy function, 414, 462
  definition of, 414
pthread_cond_init function, 414, 462, 941
  definition of, 414
PTHREAD_COND_INITIALIZER constant, 413, 416, 455, 814
pthread_cond_signal function, 415–416, 456, 821, 942
  definition of, 415
pthread_cond_t data type, 413, 416, 455, 814, 940
pthread_cond_timedwait function, 414–415, 434, 440–441, 451
  definition of, 414
pthread_cond_wait function, 414–416, 434, 451, 456, 832, 927, 941
  definition of, 414
pthread_create function, 385–388, 390–392, 395, 397, 421, 427–428, 456, 460, 477, 632, 817, 926, 941
  definition of, 385
PTHREAD_CREATE_DETACHED constant, 428
PTHREAD_CREATE_JOINABLE constant, 428
PTHREAD_DESTRUCTOR_ITERATIONS constant, 426, 447
pthread_detach function, 396–397, 427
  definition of, 397
pthread_equal function, 385, 412
  definition of, 385
pthread_exit function, 198, 236, 389–391, 393–396, 447, 824–829
  definition of, 389
pthread_getspecific function, 449–450
  definition of, 449
<pthread.h> header, 29
pthread_join function, 389–391, 395–396, 418, 451, 926
  definition of, 389
pthread_key_create function, 447–448, 450
  definition of, 447
pthread_key_delete function, 447–448
  definition of, 448
PTHREAD_KEYS_MAX constant, 426, 447
pthread_key_t data type, 449
pthread_kill function, 455
  definition of, 455
pthread_mutexattr_destroy function, 431, 445
  definition of, 431
pthread_mutexattr_getpshared function, 431
  definition of, 431
pthread_mutexattr_getrobust function, 432
  definition of, 432
pthread_mutexattr_gettype function, 434

definition of, 434
pthread_mutexattr_init function, 431, 438, 445
definition of, 431
pthread_mutexattr_setpshared function, 431
definition of, 431
pthread_mutexattr_setrobust function, 432
definition of, 432
pthread_mutexattr_settype function, 434, 438, 445
definition of, 434
pthread_mutexattr_t data type, 430–431, 438, 445
pthread_mutex_consistent function, 432–433, 571
definition of, 433
PTHREAD_MUTEX_DEFAULT constant, 433–434
pthread_mutex_destroy function, 400–401, 404, 407
definition of, 400
PTHREAD_MUTEX_ERRORCHECK constant, 433–434
pthread_mutex_init function, 400–401, 403, 405, 431, 438, 445, 941
definition of, 400
PTHREAD_MUTEX_INITIALIZER constant, 400, 403, 405, 408, 416, 431, 449, 455, 459, 813–814
pthread_mutex_lock function, 400–401, 403–404, 406–408, 416, 422–423, 432, 438, 445, 450, 456, 459–460, 820–821, 828–830, 832–833, 941–942
definition of, 400
PTHREAD_MUTEX_NORMAL constant, 433–434
PTHREAD_MUTEX_RECURSIVE constant, 433–434, 438, 445
PTHREAD_MUTEX_ROBUST constant, 432
PTHREAD_MUTEX_STALLED constant, 432
pthread_mutex_t data type, 400–401, 403, 405, 408, 416, 438, 445, 449, 455, 459, 813–814, 940
pthread_mutex_timedlock function, 407–409, 413
definition of, 407
pthread_mutex_trylock function, 400, 402
definition of, 400
pthread_mutex_unlock function, 400–401, 403–404, 406–407, 416, 422–423, 438–439, 445, 450, 456, 460, 820–821, 828–830, 832–833, 941–942
definition of, 400
pthread_once function, 445, 448, 450, 928
definition of, 448
PTHREAD_ONCE_INIT constant, 445, 448–449
pthread_once_t data type, 445, 449
PTHREAD_PROCESS_PRIVATE constant, 417, 431, 442
PTHREAD_PROCESS_SHARED constant, 417, 431, 442, 571
pthread_rwlockattr_destroy function, 439
definition of, 439
pthread_rwlockattr_getpshared function, 440
definition of, 440
pthread_rwlockattr_init function, 439
definition of, 439
pthread_rwlockattr_setpshared function, 440
definition of, 440
pthread_rwlockattr_t data type, 439
pthread_rwlock_destroy function, 409–410
definition of, 409
pthread_rwlock_init function, 409, 411
definition of, 409
PTHREAD_RWLOCK_INITIALIZER constant, 409
pthread_rwlock_rdlock function, 410, 412, 452
definition of, 410
pthread_rwlock_t data type, 411
pthread_rwlock_timedrdlock function, 413, 452
definition of, 413
pthread_rwlock_timedwrlock function, 413, 452
definition of, 413
pthread_rwlock_tryrdlock function, 410
definition of, 410
pthread_rwlock_trywrlock function, 410
definition of, 410
pthread_rwlock_unlock function, 410–412
definition of, 410
pthread_rwlock_wrlock function, 410–412, 452
definition of, 410
pthreads, 27, 229, 384, 426
pthread_self function, 385, 387, 391, 824
definition of, 385
pthread_setcancelstate function, 451
definition of, 451
pthread_setcanceltype function, 453
definition of, 453
pthread_setspecific function, 449–450
definition of, 449
pthread_sigmask function, 453–454, 477, 815
definition of, 454
pthread_spin_destroy function, 417
definition of, 417

pthread_spin_init function, 417
definition of, 417
pthread_spin_lock function, 418
definition of, 418
pthread_spin_trylock function, 418
definition of, 418
pthread_spin_unlock function, 418
definition of, 418
PTHREAD_STACK_MIN constant, 426, 430
pthread_t data type, 59, 384–385, 387, 390–391, 395, 411, 421, 428, 456, 460, 476, 632, 812, 814, 824, 829, 926, 941
pthread_testcancel function, 451, 453
definition of, 453
PTHREAD_THREADS_MAX constant, 426
ptrdiff_t data type, 59
ptsname function, 442, 723–725
definition of, 723
pty program, 309, 715, 720–721, 727, 729–742, 944
pty_fork function, 721, 724, 726–730, 732, 739, 741–742
definition of, 727
ptym_open function, 724, 726–728, 897
definition of, 724–725
ptys_fork function, 897
ptys_open function, 724, 726–728, 897
definition of, 724–725
Pu, C., 65, 953
putc function, 10, 152–156, 247–248, 452, 701
definition of, 152
putchar function, 152, 175, 452, 547–548
definition of, 152
putchar_unlocked function, 442, 444, 452
definition of, 444
putc_unlocked function, 442, 444, 452
definition of, 444
putenv function, 204, 212, 251, 442, 446, 462
definition of, 212
putenv_r function, 462
puts function, 152–153, 452, 911
definition of, 153
pututxline function, 442, 452
putwc function, 452
putwchar function, 452
PWD environment variable, 211
<pwd.h> header, 29, 177, 186
pwrite function, 78–79, 451, 461–462, 592
definition of, 78

Quarterman, J. S., 33–34, 74, 112, 116, 229, 236, 525, 951
QUIT terminal character, 678, 681, 688, 702

race conditions, 245–249, 339, 784, 922, 924
Rago, J. E., xxvii
Rago, S. A., xxxii, 88, 157, 290, 952
raise function, 331, 336–338, 365
definition of, 337
rand function, 442
raw terminal mode, 672, 704, 708, 713, 732, 734
Raymond, E. S., 952
read function, 8–10, 20, 59, 61, 64, 71–72, 78, 88, 90–91, 111, 124–125, 130, 145, 154–156, 174, 301, 308–309, 328–331, 342–343, 364–365, 378, 451, 462, 470, 482–483, 495–496, 498–502, 505–506, 508–509, 513, 517, 523–525, 530–531, 536–537, 540–541, 549–551, 553, 556, 587, 590, 592, 610, 612, 654, 656, 665–667, 672, 702–704, 708–709, 732–733, 738, 740, 748, 752, 765, 767–768, 805–806, 811, 818, 823, 836–838, 907–908, 936, 943
definition of, 71
read, scatter, 521, 644
readdir function, 5, 7, 130–135, 442, 452, 697, 823
definition of, 130
readdir_r function, 443, 452
reader–writer lock attributes, 439–440
reader–writer locks, 409–413
reading directories, 130–135
readlink function, 121, 123–124, 331, 452
definition of, 123
readlinkat function, 123–124, 331, 452
definition of, 123
read_lock function, 489, 493, 498, 897
readmore function, 814, 837, 840–841
definition of, 837
readn function, 523–524, 738, 806, 811, 896
definition of, 523–524
readv function, 41, 43, 329, 451, 481, 521–523, 531, 592, 613, 644, 752, 766
definition of, 521
readw_lock function, 489, 759, 763, 780, 897
real
group ID, 98, 102, 183, 228, 233, 252–253, 256, 270, 585
user ID, 39–40, 43, 98–99, 102, 221, 228, 233, 252–253, 256–260, 270, 276, 286, 288, 337, 381, 585, 924
realloc function, 50, 174, 207–208, 213, 661–662, 666, 761, 838, 840, 911–912
definition of, 207
record locking, 485–499
advisory, 495
deadlock, 490
mandatory, 495

timing comparison, 571
recv function, 331, 451, 592, 612–615, 626–627
definition of, 612
recv_fd function, 642–644, 650, 655, 660, 896
definition of, 642, 647
recvfrom function, 331, 451, 613, 620–623
definition of, 613
recvmsg function, 331, 451, 613, 644, 647–648, 651
definition of, 613
recv_ufd function, 650
definition of, 651
RE_DUP_MAX constant, 39, 43, 49
reentrant functions, 330–332
regcomp function, 39, 43
regexec function, 39, 43
<regex.h> header, 29
register variables, 217
regular file, 95
relative pathname, 5, 8, 43–44, 50, 64–65, 135, 553
reliable signals, 335–336
remote mode, pseudo terminal, 741
remove function, 116–119, 121, 125, 452
definition of, 119
remove_job function, 814, 822, 832
definition of, 822
rename function, 119–121, 125, 331, 452
definition of, 119
renameat function, 119–120, 331, 452
definition of, 119
replace_job function, 814, 821, 837
definition of, 821
REPRINT terminal character, 678, 681, 687, 690, 703
reset program, 713, 943
resource limits, 220–225, 233, 252, 322, 382
restarted system calls, 329–330, 342–343, 351, 354, 508, 700
restrict keyword, 26, 93, 123, 146, 148, 152–153, 156, 158–159, 161–163, 190, 192, 195, 346, 350, 385, 400, 409, 414, 428–432, 434, 440–441, 454, 502, 506, 596, 599–600, 605, 608, 613, 624
rewind function, 149, 158, 168, 452
definition of, 158
rewinddir function, 130–135, 452
definition of, 130
rfork function, 229
Ritchie, D. M., xx, 26, 143, 149, 155, 162, 164, 208, 898, 906, 950, 952
RLIM_INFINITY constant, 221, 468
rlimit structure, 220, 224, 467, 907
RLIMIT_AS constant, 221–223
RLIMIT_CORE constant, 221–223, 317
RLIMIT_CPU constant, 221–223
RLIMIT_DATA constant, 221–223
RLIMIT_FSIZE constant, 221–223, 382
RLIMIT_INFINITY constant, 224, 907
RLIMIT_MEMLOCK constant, 221–223
RLIMIT_MSGQUEUE constant, 221, 223
RLIMIT_NICE constant, 221, 223
RLIMIT_NOFILE constant, 221–223, 467, 907
RLIMIT_NPROC constant, 221–223
RLIMIT_NPTS constant, 221, 223
RLIMIT_RSS constant, 222–223
RLIMIT_SBSIZE constant, 222–223
RLIMIT_SIGPENDING constant, 222, 224
RLIMIT_STACK constant, 222, 224
RLIMIT_SWAP constant, 222, 224
RLIMIT_VMEM constant, 222, 224
rlim_t data type, 59, 223
rlogin program, 717, 741–742
rlogind program, 717, 734, 741, 944
rm program, 559, 663
rmdir function, 117, 119–120, 125, 129–130, 331
definition of, 130
robust attribute, 431, 571
R_OK constant, 102–103
root
directory, 4, 8, 24, 139, 141, 233, 252, 283, 910
login name, 16
routed program, 472
rpcbind program, 465
RS-232, 674, 685–686
rsyslogd program, 465, 480
RTSIG_MAX constant, 40, 43
Rudoff, A. M., 157, 291, 470, 589, 952
runacct program, 269

S5 file system, 65
sa program, 269
sac program, 290
Sacksen, J., xxxii
SAF (Service Access Facility), 290
safe, async-signal, 330, 446, 450, 457, 461–462, 927
sa_handler structure, 376
SA_INTERRUPT constant, 351, 354–355
s_alloc function, 584
Salus, P. H., xxxii, 952
SA_NOCLDSTOP constant, 351
SA_NOCLDWAIT constant, 333, 351
SA_NODEFER constant, 351, 354
Santa Cruz Operation, *see* SCO
SA_ONSTACK constant, 351

SA_RESETHAND constant, 351, 354
SA_RESTART constant, 329, 351, 354, 508–509
SA_SIGINFO constant, 336, 350–353, 376, 512
saved
  set-group-ID, 56, 98, 257
  set-user-ID, 56, 98, 256–260, 288, 337
S_BANDURG constant, 510
sbrk function, 21–23, 208, 221
_SC_AIO_MAX constant, 516
_SC_AIO_PRIO_DELTA_MAX constant, 516
scaling, frequency, 785
scan_configfile function, 803–804
  definition of, 803
scandir function, 452
scanf function, 150, 162–163, 452
  definition of, 162
_SC_ARG_MAX constant, 43, 47
_SC_ASYNCHRONOUS_IO constant, 57
_SC_ATEXIT_MAX constant, 43
scatter read, 521, 644
_SC_BARRIERS constant, 57
_SC_CHILD_MAX constant, 43, 221
_SC_CLK_TCK constant, 42–43, 280–281
_SC_CLOCK_SELECTION constant, 57
_SC_COLL_WEIGHTS_MAX constant, 43
_SC_DELAYTIMER_MAX constant, 43
SCHAR_MAX constant, 37–38
SCHAR_MIN constant, 37–38
<sched.h> header, 29
scheduling, process, 276–280
_SC_HOST_NAME_MAX constant, 43, 616, 618, 623, 815
Schwartz, A., 181, 250, 298, 949
_SC_IO_LISTIO_MAX constant, 516
_SC_IOV_MAX constant, 43
_SC_JOB_CONTROL constant, 54, 57
_SC_LINE_MAX constant, 43
_SC_LOGIN_NAME_MAX constant, 43
_SC_MAPPED_FILES constant, 57
SCM_CREDENTIALS constant, 649–652
SCM_CREDS constant, 649–650, 652
SCM_CREDTYPE constant, 650, 652
_SC_MEMORY_PROTECTION constant, 57
SCM_RIGHTS constant, 645–646, 650, 652
_SC_NGROUPS_MAX constant, 43
_SC_NZERO function, 276
SCO (Santa Cruz Operation), 35
_SC_OPEN_MAX constant, 43, 52, 221, 907
_SC_PAGESIZE constant, 43, 527
_SC_PAGE_SIZE constant, 43, 527
_SC_READER_WRITER_LOCKS constant, 57
_SC_REALTIME_SIGNALS constant, 57
_SC_RE_DUP_MAX constant, 43
script program, 715, 719–720, 734, 736–737, 741–742
_SC_RTSIG_MAX constant, 43
_SC_SAVED_IDS constant, 54, 57, 98, 256
_SC_SEMAPHORES constant, 57
_SC_SEM_NSEMS_MAX constant, 43
_SC_SEM_VALUE_MAX constant, 43
_SC_SHELL constant, 57
_SC_SIGQUEUE_MAX constant, 43
_SC_SPIN_LOCKS constant, 57
_SC_STREAM_MAX constant, 43
_SC_SYMLOOP_MAX constant, 43
_SC_THREAD_ATTR_STACKADDR constant, 429
_SC_THREAD_ATTR_STACKSIZE constant, 429
_SC_THREAD_DESTRUCTOR_ITERATIONS constant, 426
_SC_THREAD_KEYS_MAX constant, 426
_SC_THREAD_PROCESS_SHARED constant, 431
_SC_THREADS constant, 57, 384
_SC_THREAD_SAFE_FUNCTIONS constant, 57, 442
_SC_THREAD_STACK_MIN constant, 426
_SC_THREAD_THREADS_MAX constant, 426
_SC_TIMER_MAX constant, 43
_SC_TIMERS constant, 57
_SC_TTY_NAME_MAX constant, 43
_SC_TZNAME_MAX constant, 43
_SC_V7_ILP32_OFF32 constant, 70
_SC_V7_ILP32_OFFBIG constant, 70
_SC_V7_LP64_OFF64 constant, 70
_SC_V7_LP64_OFFBIG constant, 70
_SC_VERSION constant, 50, 54, 57
_SC_XOPEN_CRYPT constant, 57
_SC_XOPEN_REALTIME constant, 57
_SC_XOPEN_REALTIME_THREADS constant, 57
_SC_XOPEN_SHM constant, 57
_SC_XOPEN_VERSION constant, 50, 54, 57
<search.h> header, 30
sed program, 950
Seebass, S., 951
seek function, 67
SEEK_CUR constant, 67, 158, 486, 494–495, 766
seekdir function, 130–135, 452
  definition of, 130
SEEK_END constant, 67, 158, 486, 494–495, 771–773, 781
SEEK_SET constant, 67, 158, 172, 486, 494–495, 498, 759, 762–763, 765–766, 768–773, 775–780, 818–819, 930–931
SEGV_ACCERR constant, 353
SEGV_MAPERR constant, 353
select function, 330–331, 343, 451, 481, 501–509, 531–532, 560, 586, 588, 592, 608–609, 626–627, 631–632, 659, 664–666, 668, 718,

732, 742, 805–806, 816–817, 928–929, 933, 936, 939, 942
definition of, 502
Seltzer, M., 744, 952
semaphore, 57, 534, 565–571
adjustment on exit, 570–571
locking timing comparison, 571, 583
<semaphore.h> header, 29
sembuf structure, 568–569
sem_close function, 580, 584
definition of, 580
semctl function, 558, 562, 566–568, 570
definition of, 567
sem_destroy function, 582
definition of, 582
SEM_FAILED constant, 584
semget function, 557–558, 566–567
definition of, 567
sem_getvalue function, 582
definition of, 582
semid_ds structure, 566–568
sem_init function, 582
definition of, 582
SEM_NSEMS_MAX constant, 40, 43
semop function, 452, 559, 567–570
definition of, 568
sem_open function, 579–580, 582, 584
definition of, 579
sem_post function, 331, 581–582, 584
definition of, 582
sem_t structure, 582
sem_timedwait function, 451, 581–582
definition of, 581
sem_trywait function, 581, 584
semun union, 567–568
SEM_UNDO constant, 569–570, 580, 583
sem_unlink function, 580–581, 584
definition of, 580
SEM_VALUE_MAX constant, 40, 43, 580
sem_wait function, 451, 581–582, 584
definition of, 581
send function, 331, 451, 592, 610, 616, 626–627
definition of, 610
send_err function, 642–644, 653, 656–657, 668–669, 897
definition of, 642, 644
send_fd function, 642–645, 649, 653, 656–657, 669, 897
definition of, 642, 646, 649
sendmsg function, 331, 451, 611, 613, 644–646, 650, 670
definition of, 611
sendto function, 331, 451, 610–611, 620, 622–623
definition of, 610
S_ERROR constant, 510
serv_accept function, 636–638, 641, 648, 659, 665, 667–668, 897
definition of, 636, 638
servent structure, 599
Service Access Facility, *see* SAF
Service Management Facility, *see* SMF
serv_listen function, 636–637, 659, 664–665, 667, 670, 897
definition of, 636–637
session, 295–296
ID, 233, 252, 296, 311, 463–464
leader, 295–297, 311, 318, 464–466, 469, 726–727, 742, 944
process group ID, 304
session structure, 310–311, 318, 464
set
descriptor, 503, 505, 532, 933
signal, 336, 344–345, 532, 933
SETALL constant, 568, 570
setasync function, definition of, 939
setbuf function, 146–147, 150, 171, 175, 247–248, 701, 930
definition of, 146
set_cloexec function, 615, 617, 622, 896
definition of, 480
setegid function, 258
definition of, 258
setenv function, 212, 251, 442
definition of, 212
seteuid function, 258–260
definition of, 258
set_fl function, 86, 482–483, 498, 896, 934
definition of, 85
setgid function, 256, 258, 288, 331, 816
definition of, 256
setgrent function, 183–184, 442, 452
definition of, 183
set-group-ID, 98–99, 102, 107–108, 110, 129, 140, 233, 253, 317, 496, 546, 723
saved, 56, 98, 257
setgroups function, 184
definition of, 184
sethostent function, 452, 597
definition of, 597
sethostname function, 189
setitimer function, 317, 320, 322, 381
_setjmp function, 355, 358
setjmp function, 197, 213, 215–219, 225, 340, 343, 355–356, 358, 381, 924
definition of, 215
<setjmp.h> header, 27

setkey function, 442
setlogmask function, 470–471
  definition of, 470
setnetent function, 452, 598
  definition of, 598
setpgid function, 294, 331
  definition of, 294
setpriority function, 277
  definition of, 277
setprotoent function, 452, 598
  definition of, 598
setpwent function, 180–181, 442, 452
  definition of, 180
setregid function, 257–258
  definition of, 257
setreuid function, 257
  definition of, 257
setrlimit function, 53, 220, 382
  definition of, 220
setservent function, 452, 599
  definition of, 599
setsid function, 294–295, 297, 310–311, 331, 464–467, 724, 727–728
  definition of, 295
setsockopt function, 331, 624–625, 651
  definition of, 624
setspent function, 182
  definition of, 182
settimeofday function, 190
setuid function, 98, 256, 258, 260, 288, 331, 816
  definition of, 256
set-user-ID, 98–99, 102, 104, 107–108, 110, 129, 140, 182, 233, 253, 256–257, 259, 267, 317, 546, 585–586, 653, 924
  saved, 56, 98, 256–260, 288, 337
setutxent function, 442, 452
SETVAL constant, 568, 570
setvbuf function, 146–147, 150, 171, 175, 220, 552, 721, 936
  definition of, 146
SGI (Silicon Graphics, Inc.), 35
SGID, *see* set-group-ID
SHA-1, 181
shadow passwords, 181–182, 196, 918
<shadow.h> header, 186
S_HANGUP constant, 510
Shannon, W. A., 525, 949
shared
  libraries, 206–207, 226, 753, 920, 947
  memory, 534, 571–578
sharing, file, 74–77, 231
shell, *see* Bourne shell, Bourne-again shell, C shell, Debian Almquist shell, Korn shell, TENEX C shell
SHELL environment variable, 211, 288, 737
shell, job-control, 294, 299, 306–307, 325, 358, 377, 379, 734–735
shell layers, 299
shells, 3
S_HIPRI constant, 510
shmat function, 559, 573–576
  definition of, 574
shmatt_t data type, 572
shmctl function, 558, 562, 573–575
  definition of, 573
shmdt function, 574
  definition of, 574
shmget function, 557–558, 572, 575
  definition of, 572
shmid_ds structure, 572–574
SHMLBA constant, 574
SHM_LOCK constant, 573
SHM_RDONLY constant, 574
SHM_RND constant, 574
SHRT_MAX constant, 37
SHRT_MIN constant, 37
shutdown function, 331, 592–593, 612
  definition of, 592
SHUT_RD constant, 592
SHUT_RDWR constant, 592
SHUT_WR constant, 592
SI_ASYNCIO constant, 353
S_IFBLK constant, 134
S_IFCHR constant, 134
S_IFDIR constant, 134
S_IFIFO constant, 134
S_IFLNK constant, 114, 134
S_IFMT constant, 97
S_IFREG constant, 134
S_IFSOCK constant, 134, 634
sig2str function, 380–381
  definition of, 380
SIG2STR_MAX constant, 380
SIGABRT signal, 236, 240–241, 275, 313, 317–319, 365–367, 381, 924
sigaction function, 59, 323, 326, 329–331, 333, 335–336, 349–355, 366, 370, 374, 376, 455, 468, 476, 478–479, 510, 621, 815, 939
  definition of, 350
sigaction structure, 350, 354–355, 366, 369, 374, 376, 379, 467, 476, 478, 621, 814
sigaddset function, 331, 344–345, 348, 360, 362–363, 370, 374, 378, 456, 478–479, 701, 815, 933
  definition of, 344–345
SIGALRM signal, 313–314, 317, 330–332, 338–340, 342–343, 347, 354, 356–357, 364–365, 373–374, 621

sigaltstack function, 351
sig_atomic_t data type, 59, 356–357, 361–363, 732
SIG_BLOCK constant, 346, 348, 360, 362–363, 370, 374, 454, 456, 477, 701, 815
SIGBUS signal, 317, 352–353, 527, 530
SIGCANCEL signal, 317
SIGCHLD signal, 238, 288, 315, 317, 331–335, 351–353, 367–368, 370–371, 377, 471, 501, 546, 723, 923, 939
  semantics, 332–335
SIGCLD signal, 317, 332–336
SIGCONT signal, 301, 309, 317, 337, 377, 379
sigdelset function, 331, 344–345, 366, 374, 933
  definition of, 344–345
SIG_DFL constant, 323, 333, 350–351, 366, 378–379, 476
sigemptyset function, 331, 344, 348, 354–355, 360, 362–363, 369–370, 374, 378, 456, 467, 476, 478, 621, 701, 815, 933
  definition of, 344
SIGEMT signal, 317–318
SIG_ERR constant, 19, 324, 334, 340–343, 348, 354–356, 360–361, 363, 368, 550, 709, 711, 733
sigevent structure, 512
SIGEV_NONE constant, 518
sigfillset function, 331, 344, 366, 477, 933
  definition of, 344
SIGFPE signal, 18, 240–241, 317–318, 352–353
SIGFREEZE signal, 317–318
Sigfunc data type, 354–355, 896
SIGHUP signal, 308–309, 317–318, 468, 475–479, 546, 815, 830, 843
SIG_IGN constant, 323, 333, 350, 366, 369, 379, 467, 815
SIGILL signal, 317–318, 351–353, 366
SIGINFO signal, 317–318, 682, 689
siginfo structure, 244, 283, 351–352, 376, 379, 381, 512
SIGINT signal, 18–19, 300, 314, 317, 319–320, 340–341, 347, 359–361, 364–365, 367–370, 372, 455–457, 546, 679, 681, 685, 688–689, 701–702, 709, 930, 932
SIGIO signal, 83, 317, 319, 501, 509–510, 627
SIGIOT signal, 317, 319, 365
sigismember function, 331, 344–345, 347–348, 933
  definition of, 344–345
sigjmp_buf data type, 356
SIGJVM1 signal, 317
SIGJVM2 signal, 317
SIGKILL signal, 272, 275, 315, 317, 319, 321, 323, 346, 380, 735
siglongjmp function, 219, 331, 355–358, 365
  definition of, 356
SIGLOST signal, 317
SIGLWP signal, 317, 319, 321
signal function, 18–19, 59, 308, 323–326, 329–335, 339–343, 348–349, 354–356, 360–361, 363, 368, 378, 510, 550, 709, 711, 939
  definition of, 323, 354
signal mask, 336
signal set, 336, 344–345, 532, 933
<signal.h> header, 27, 240, 314, 324, 344–345, 380
signal_intr function, 330, 355, 364, 382, 508, 733, 896, 930
  definition of, 355
signals, 18–19, 313–382
  blocking, 335
  delivery, 335
  generation, 335
  generation, pseudo terminal, 741
  job-control, 377–379
  null, 314, 337
  pending, 335
  queueing, 336, 349, 376
  reliable, 335–336
  unreliable, 326–327
signal_thread function, 814, 830
  definition of, 830
sigpause function, 331
sigpending function, 331, 335, 347–349
  definition of, 347
SIGPIPE signal, 314, 317, 319, 537, 550–551, 553, 556, 587, 611, 815, 936
SIGPOLL signal, 317, 319, 501, 509–510
sigprocmask function, 331, 336, 340, 344, 346–349, 360, 362–364, 366, 370, 374, 378, 453–454, 456, 701
  definition of, 346
SIGPROF signal, 317, 320
SIGPWR signal, 317–318, 320
sigqueue function, 222, 331, 353, 376–377
  definition of, 376
SIGQUEUE_MAX constant, 40, 43, 376
SIGQUIT signal, 300, 317, 320, 347–349, 361–362, 367, 370, 372, 456–457, 546, 681, 689, 702, 709
SIGRTMAX constant, 376
SIGRTMIN constant, 376
SIGSEGV signal, 314, 317, 320, 332, 336, 352–353, 393, 527
sigset function, 331, 333
sigsetjmp function, 219, 331, 355–358
  definition of, 356
SIG_SETMASK constant, 346, 348–349, 360, 362–364, 366, 370, 374, 454, 456, 701

sigset_t data type, 59, 336, 344, 347–348, 360–361, 363, 366, 369, 374, 378, 454–456, 701, 813
SIGSTKFLT signal, 317, 320
SIGSTOP signal, 315, 317, 320, 323, 346, 377
SIGSUSP signal, 689
sigsuspend function, 331, 340, 359–365, 374, 451
  definition of, 359
SIGSYS signal, 317, 320
SIGTERM signal, 315, 317, 321, 325, 476–479, 709, 732–733, 742, 815, 830, 944
SIGTHAW signal, 317, 321
SIGTHR signal, 319
sigtimedwait function, 451
SIGTRAP signal, 317, 321, 351, 353
SIGTSTP signal, 300, 308, 317, 320–321, 377–379, 680, 682, 701, 735
SIGTTIN signal, 300–301, 304, 309, 317, 321, 377, 379
SIGTTOU signal, 301–302, 317, 321, 377, 379, 691
SIG_UNBLOCK constant, 346, 349, 378, 454
SIGURG signal, 83, 314, 317, 319, 322, 510–511, 626
SIGUSR1 signal, 317, 322, 324, 347, 356–358, 360–361, 363–364, 501
SIGUSR2 signal, 317, 322, 324, 363–364
sigval structure, 352
SIGVTALRM signal, 317, 322
sigwait function, 451, 454–455, 457, 475, 477, 830
  definition of, 454
sigwaitinfo function, 451
SIGWAITING signal, 317, 322
SIGWINCH signal, 311, 317, 322, 710–712, 718–719, 741–742
SIGXCPU signal, 221, 317, 322
SIGXFSZ signal, 221, 317, 322, 382, 925
SIGXRES signal, 317, 322
Silicon Graphics, Inc., *see* SGI
SI_MESGQ constant, 353
Singh, A., 112, 116, 952
Single UNIX Specification, *see* SUS
  Version 3, *see* SUSv3
  Version 4, *see* SUSv4
single-instance daemons, 473–474
S_INPUT constant, 510
SIOCSPGRP constant, 627
SI_QUEUE constant, 353
S_IRGRP constant, 99, 104, 107, 140, 149, 473, 896
S_IROTH constant, 99, 104, 107, 140, 149, 473, 896
S_IRUSR constant, 99, 104, 107, 140, 149, 169, 473, 818, 896
S_IRWXG constant, 107, 639
S_IRWXO constant, 107, 639
S_IRWXU constant, 107, 584, 639
S_ISBLK function, 96–97, 139
S_ISCHR function, 96–97, 139, 698
S_ISDIR function, 96–97, 133, 698
S_ISFIFO function, 96–97, 535, 552
S_ISGID constant, 99, 107, 140, 498
S_ISLNK function, 96–97
S_ISREG function, 96, 808
S_ISSOCK function, 96–97, 639
S_ISUID constant, 99, 107, 140
S_ISVTX constant, 107–109, 140
SI_TIMER constant, 353
SI_USER constant, 353
S_IWGRP constant, 99, 104, 107, 140, 149
S_IWOTH constant, 99, 104, 107, 140, 149
S_IWUSR constant, 99, 104, 107, 140, 149, 169, 473, 818, 896
S_IXGRP constant, 99, 107, 140, 498, 896
S_IXOTH constant, 99, 107, 140, 896
S_IXUSR constant, 99, 107, 140, 169, 896
size, file, 111–112
size program, 206–207, 226
sizeof operator, 231
size_t data type, 59–60, 71, 507, 772, 906
__SLBF constant, 166
sleep function, 230, 234, 243, 246, 272, 274, 308, 331, 334, 339–342, 348, 372–375, 381–382, 387, 391–392, 439, 451, 460, 504, 532, 606–607, 923, 925, 928, 931, 936
  definition of, 373–374, 929
sleep program, 372
sleep2 function, 924
sleep_us function, 532, 896
  definition of, 933–934
SMF (Service Management Facility), 293
S_MSG constant, 510
__SNBF constant, 165
Snow Leopard, xxi
snprintf function, 159, 901, 904
  definition of, 159
Snyder, G., 951
sockaddr structure, 595–597, 605–607, 609, 622, 625, 635, 637, 639, 641, 800
sockaddr_in structure, 595–596, 603
sockaddr_in6 structure, 595–596
sockaddr_un structure, 634–638, 640–642
sockatmark function, 331, 626
  definition of, 626
SOCK_DGRAM constant, 590–591, 602, 608, 612, 621, 623, 632, 941
socket
  addressing, 593–605
  descriptors, 590–593
  I/O, asynchronous, 627

I/O, nonblocking, 608–609, 627
mechanism, 95, 534, 587, 589–628
options, 623–625
socket function, 148, 331, 590, 592, 607, 609, 621, 625, 637–638, 640–641, 808
definition of, 590
socketpair function, 148, 331, 629–630, 632, 634, 941
definition of, 630
sockets, UNIX domain, 629–642
timing, 565
socklen_t data type, 606–607, 609, 622, 625, 800
SOCK_RAW constant, 590–591, 602
SOCK_SEQPACKET constant, 590–591, 602, 605, 609, 612, 625
SOCK_STREAM constant, 319, 590–591, 602, 605, 609, 612, 614–616, 618–619, 625, 630, 635, 637, 640, 802, 808, 816
Solaris, xxi–xxii, xxv, xxvii, 3–4, 26–27, 29–30, 35–36, 38, 41, 48–49, 57–60, 62, 64–65, 70, 76, 88, 102, 108–113, 121–122, 129, 131–132, 138, 178, 182, 184–188, 208–209, 211–212, 222, 225, 229, 240, 242, 244–245, 260, 277, 288, 290, 293, 296, 298, 303, 314, 316–323, 329, 334–335, 351, 355, 371, 373, 377, 379–380, 385, 388, 392, 396, 409, 426–427, 432, 439, 471, 485, 496–497, 499, 503, 530–531, 534, 559, 561, 563, 565, 567, 572–573, 576, 592, 594, 607–608, 611–613, 627, 634, 648, 675–678, 684–691, 693, 700, 704, 716–717, 723–724, 726–727, 740–741, 744, 799, 911, 918, 925, 930, 932, 935–936, 951
SOL_SOCKET constant, 624–625, 645–646, 650–652
solutions to exercises, 905–945
SOMAXCONN constant, 608
SO_OOBINLINE constant, 626
SO_PASSCRED constant, 651
SO_REUSEADDR constant, 625
source code, availability, xxx
S_OUTPUT constant, 510
Spafford, G., 181, 250, 298, 949
spawn function, 234
<spawn.h> header, 30
spin locks, 417–418
spooling, printer, 793–795
sprintf function, 159, 549, 616, 622, 640, 655, 657, 659, 668–669, 759, 772–773, 803, 818–819, 822–823, 825–827, 833–835, 837, 845, 945
definition of, 159
spwd structure, 918
squid login name, 178
S_RDBAND constant, 510
S_RDNORM constant, 510
sscanf function, 162, 549, 551, 802–803
definition of, 162
ssh program, 293
sshd program, 465
SSIZE_MAX constant, 38, 41, 71
ssize_t data type, 39, 59, 71
stack, 205, 215
stackaddr attribute, 427
stacksize attribute, 427
standard error, 8, 145, 617
standard error routines, 898–904
standard input, 8, 145
standard I/O
alternatives, 174–175
buffering, 145–147, 231, 235, 265, 367, 552, 721, 752
efficiency, 153–156
implementation, 164–167
library, 10, 143–175
streams, 143–144
versus unbuffered I/O, timing, 155
standard output, 8, 145, 617
standards, 25–33
differences, 58–59
START terminal character, 678, 680–682, 686, 689, 693
stat function, 4, 7, 65, 93–95, 97, 99, 107, 121–122, 124, 126–128, 131, 138, 140–141, 170, 331, 452, 586, 592, 628, 639–640, 670, 698, 908, 910, 942
definition of, 93
stat structure, 93–96, 98, 111, 114, 124, 140, 147, 167, 170, 498, 518, 529, 535, 552, 557, 586, 638, 697–698, 757, 807, 832
static variables, 219
STATUS terminal character, 678, 682, 687, 689, 703
<stdarg.h> header, 27, 162–163, 755, 758
<stdbool.h> header, 27
__STDC_IEC_559__ constant, 31
<stddef.h> header, 27, 635
stderr variable, 145, 483, 731, 901
STDERR_FILENO constant, 62, 145, 618–619, 643, 648, 652, 729
stdin variable, 10, 145, 154, 214, 216, 550–551, 654
STDIN_FILENO constant, 9, 62, 67, 72, 145, 308, 378, 483, 539, 544, 549–550, 619, 655–656, 679, 684, 709, 711, 728, 730–732, 739–740
<stdint.h> header, 27, 595
<stdio.h> header, 10, 27, 38, 51, 145, 147, 151, 164, 168, 694, 755, 895

<stdlib.h> header, 27, 208, 895
stdout variable, 10, 145, 154, 247–248, 275, 901, 921, 930
STDOUT_FILENO constant, 9, 62, 72, 145, 230, 235, 378, 483, 537, 544, 549–550, 614, 618–620, 654–656, 729, 733, 739–740, 921
Stevens, D. A., xxxii
Stevens, E. M., xxxii
Stevens, S. H., xxxii
Stevens, W. R., xx, xxv–xxvi, xxxii, 157, 291, 470, 505, 589, 717, 793, 952
sticky bit, 107–109, 117, 140
stime function, 190
Stonebraker, M. R., 743, 953
STOP terminal character, 678, 680–682, 686, 689, 693
str2sig function, 380
  definition of, 380
strace program, 497
Strang, J., 712, 953
strchr function, 767
stream orientation, 144
STREAM_MAX constant, 38, 40, 43, 49
STREAMS, xxii, 88, 143, 501–502, 506, 508, 510, 534, 560, 565, 648, 716–717, 722, 726, 740
streams, memory, 171–174
STREAMS module
  ldterm, 716, 726
  pckt, 716, 740
  ptem, 716, 726
  ttcompat, 716, 726
streams, standard I/O, 143–144
STREAMS-based pipes, mounted, 534
  timing, 565
strerror function, 15–16, 24, 380, 442, 452, 471, 474, 478–479, 600, 615–618, 621–622, 657, 669, 823–827, 830, 833–834, 842, 899, 901, 904, 906, 931
  definition of, 15
strerror_r function, 443, 452
strftime function, 190, 192–196, 264, 408, 452, 919
  definition of, 192
strftime_l function, 192
  definition of, 192
<string.h> header, 27, 895
<strings.h> header, 29
strip program, 920
strlen function, 12, 231, 945
strncasecmp function, 840
strncpy function, 809
Strong, H. R., 744, 750, 949
<stropts.h> header, 508, 510
strptime function, 195
  definition of, 195
strsignal function, 380, 830
  definition of, 380
strtok function, 442, 657–658
strtok_r function, 443
strtol function, 633
stty program, 301, 691–692, 702, 713, 943
Stumm, M., 174, 531, 950
S_TYPEISMQ function, 96
S_TYPEISSEM function, 96
S_TYPEISSHM function, 96
su program, 472
submit_file function, 807, 809, 811
  definition of, 809
SUID, *see* set-user-ID
Sun Microsystems, xxi–xxii, xxvii, 33, 35, 76, 740, 953
SunOS, xxxi, 33, 206, 330, 354
superuser, 16
supplementary group ID, 18, 39, 98, 101, 108, 110, 183–184, 233, 252, 258
SUS (Single UNIX Specification), xxi, xxvi, 28, 30–33, 36, 50, 53–54, 57–58, 60–61, 64, 69, 78, 88, 94, 105, 107, 109, 131, 136, 143, 157, 163, 168–169, 180, 183, 190–191, 196, 211–212, 220–221, 234, 239, 244–245, 262, 293, 296, 311, 315, 322, 330, 333, 352, 354, 410, 425, 429–431, 442, 469–472, 485, 496, 501, 507, 509, 521, 527–528, 533–534, 559, 561, 565–566, 572–573, 583, 596, 607, 610, 612, 623, 627, 645, 662, 674, 678, 683, 722–724, 744, 910, 950, 953
SUSP terminal character, 678, 680, 682, 688, 701
SUSv3 (Single UNIX Specification, Version 3), 32
SUSv4 (Single UNIX Specification, Version 4), 32, 88, 132, 143, 153, 168–169, 189, 314, 319–320, 336, 375–376, 384, 442, 501, 509–510, 525, 533, 571, 579
SVID (System V Interface Definition), xix, 32–33, 948
SVR2, 65, 187, 317, 329, 336, 340–341, 712, 948
SVR3, 76, 129, 201, 299, 313, 317, 319, 326, 329, 333, 336, 496, 502, 507, 898, 948
SVR3.0, xxxi
SVR3.1, xxxi
SVR3.2, xxxi, 36, 81, 267
SVR4, xxii, xxxi–xxxii, 3, 21, 33, 35–36, 48, 63, 65, 76, 121, 187, 209, 290, 296, 299, 310, 313, 317, 329, 333, 336, 469, 502, 507–508, 521, 712, 722, 744, 948, 953
swapper process, 227
S_WRBAND constant, 510
S_WRNORM constant, 510

symbolic link, 55, 94–95, 110–111, 114, 118, 120–123, 131, 137, 141, 186, 908–909
symlink function, 123–124, 331, 452
  definition of, 123
symlinkat function, 123–124, 331, 452
  definition of, 123
SYMLINK_MAX constant, 39, 44, 49
SYMLOOP_MAX constant, 40, 43, 48–49
sync function, 61, 81, 452
  definition of, 81
sync program, 81
synchronization mechanisms, 86–87
synchronous write, 63, 86–87
<sys/acct.h> header, 269
sysconf function, 20, 37, 39, 41–48, 50–54, 57, 59–60, 69, 98, 201, 221, 256, 276, 280–281, 384, 425–426, 429, 431, 442, 516, 527, 616, 618, 623, 800, 815, 907
  definition of, 42
sysctl program, 315, 559
sysdef program, 559
<sys/disklabel.h> header, 88
<sys/filio.h> header, 88
<sys/ipc.h> header, 30, 558
<sys/iso/signal_iso.h> header, 314
syslog function, 452, 465, 468–476, 478–480, 615–619, 622–623, 901, 904, 928
  definition of, 470
syslogd program, 470–471, 473, 475, 479–480
<syslog.h> header, 30
<sys/mkdev.h> header, 138
<sys/mman.h> header, 29
<sys/msg.h> header, 30
<sys/mtio.h> header, 88
<sys/param.h> header, 49, 51
<sys/resource.h> header, 30
<sys/select.h> header, 29, 501, 504, 932–933
<sys/sem.h> header, 30, 568
<sys/shm.h> header, 30
sys_siglist variable, 379
<sys/signal.h> header, 314
<sys/socket.h> header, 29, 608
<sys/sockio.h> header, 88
<sys/stat.h> header, 29, 97
<sys/statvfs.h> header, 29
<sys/sysmacros.h> header, 138
system calls, 1, 21
  interrupted, 327–330, 343, 351, 354–355, 365, 508
  restarted, 329–330, 342–343, 351, 354, 508, 700
  tracing, 497
  versus functions, 21–23
system function, 23, 129, 227, 249, 264–269, 281–283, 349, 367–372, 381, 451, 538, 542, 923, 936
  definition of, 265–266, 369
  return value, 371
system identification, 187–189
system process, 228, 337
System V, xxv, 87, 464, 466, 469, 475, 482, 485, 500–501, 506, 509–510, 722, 726
System V Interface Definition, *see* SVID
<sys/time.h> header, 30, 501
<sys/times.h> header, 29
<sys/ttycom.h> header, 88
<sys/types.h> header, 29, 58, 138, 501, 557, 933
<sys/uio.h> header, 30
<sys/un.h> header, 29, 634
<sys/utsname.h> header, 29
<sys/wait.h> header, 29, 239

TAB0 constant, 691
TAB1 constant, 691
TAB2 constant, 691
TAB3 constant, 690–691
TABDLY constant, 676, 684, 689–691
Tankus, E., xxxii
tar program, 127, 135, 142, 910–911
<tar.h> header, 29
tcdrain function, 322, 331, 451, 677, 693
  definition of, 693
tcflag_t data type, 674
tcflow function, 322, 331, 677, 693
  definition of, 693
tcflush function, 145, 322, 331, 673, 677, 693
  definition of, 693
tcgetattr function, 331, 674, 677, 679, 683–684, 691–692, 695, 701, 705–707, 722, 730–731
  definition of, 683
tcgetpgrp function, 298–299, 331, 674, 677
  definition of, 298
tcgetsid function, 298–299, 674, 677
  definition of, 299
TCIFLUSH constant, 693
TCIOFF constant, 693
TCIOFLUSH constant, 693
TCION constant, 693
TCMalloc, 210, 949
TCOFLUSH constant, 693
TCOOFF constant, 693
TCOON constant, 693
TCSADRAIN constant, 683
TCSAFLUSH constant, 679, 683, 701, 705–707
TCSANOW constant, 683–684, 728, 731

tcsendbreak function, 322, 331, 677, 682, 693–694
  definition of, 693
tcsetattr function, 322, 331, 673–674, 677, 679, 683–684, 691–692, 701, 705–707, 722, 728, 731, 738
  definition of, 683
tcsetpgrp function, 298–299, 301, 303, 322, 331, 674, 677
  definition of, 298
tee program, 554–555
tell function, 67
TELL_CHILD function, 247–248, 362, 491, 498, 532, 539, 541, 577, 898
  definition of, 363, 540
telldir function, 130–135
  definition of, 130
TELL_PARENT function, 247, 362, 491, 532, 539, 541, 577, 898, 934
  definition of, 363, 540
TELL_WAIT function, 247–248, 362, 491, 498, 532, 539, 577, 898, 934
  definition of, 363, 540
telnet program, 292–293, 500, 738–739, 742
telnetd program, 291–292, 500–501, 717, 734, 923, 944
tempnam function, 169
TENEX C shell, 3
TERM environment variable, 211, 287, 289
termcap, 712–713, 953
terminal
  baud rate, 692–693
  canonical mode, 700–703
  controlling, 63, 233, 252, 270, 292, 295–298, 301, 303–304, 306, 309, 311–312, 318, 321, 377, 463, 465–466, 469, 480, 680, 685, 691, 694, 700, 702, 716, 724, 726–727, 898, 953
  identification, 694–700
  I/O, 671–713
  line control, 693–694
  logins, 285–290
  mode, cbreak, 672, 704, 708, 713
  mode, cooked, 672
  mode, raw, 672, 704, 708, 713, 732, 734
  noncanonical mode, 703–710
  options, 683–691
  parity, 688
  process group ID, 303, 463
  special input characters, 678–682
  window size, 311, 322, 710–712, 718, 727, 741–742
termination, process, 198–202
terminfo, 712–713, 949, 953
termio structure, 674
<termio.h> header, 674
termios structure, 64, 311, 674, 677–679, 683–684, 692–693, 695, 701, 703–706, 708, 722, 727, 730–732, 738, 741–742, 897, 944
<termios.h> header, 29, 88, 674
text segment, 204
<tgmath.h> header, 27
Thompson, K., 75, 181, 229, 743, 951–953
thread–fork interactions, 457–461
thread_init function, 445
threads, 14, 27, 229, 383–423, 578
  cancellation options, 451–453
  concepts, 383–385
  control, 425–462
  creation, 385–388
  I/O, 461–462
  reentrancy, 442–446
  synchronization, 397–422
  termination, 388–397
thread–signal interactions, 453–457
thread-specific data, 446–451
thundering herd, 927
tick, clock, 20, 42–43, 49, 59, 270, 280
time
  and date functions, 189–196
  calendar, 20, 24, 59, 126, 189, 191–192, 264, 270
  process, 20, 24, 59, 280–282
  values, 20
time program, 20
TIME terminal value, 687, 703–704, 708, 713, 943
time function, 189–190, 194, 264, 331, 357, 639–640, 919, 929
  definition of, 189
<time.h> header, 27, 59
timeout function, 439, 462
TIMER_ABSTIME constant, 375
timer_getoverrun function, 331
timer_gettime function, 331
TIMER_MAX constant, 40, 43
timer_settime function, 331, 353
times, file, 124–125, 532
times function, 42, 59, 280–281, 331, 522
  definition of, 280
timespec structure, 94, 126, 128, 189–190, 375, 407–408, 413–414, 437–438, 506, 832
time_t data type, 20, 59, 94, 189, 192, 196, 906
timeval structure, 190, 414, 421, 437, 503, 506, 805–806, 929, 933
timing
  full-duplex pipes, 565
  message queues, 565
  read buffer sizes, 73

read/write versus mmap, 530
standard I/O versus unbuffered I/O, 155
STREAMS-based pipes, 565
synchronization mechanisms, 86–87
UNIX domain sockets, 565
writev versus other techniques, 522
timing comparison, mutex, 571
record locking, 571
semaphore locking, 571, 583
TIOCGWINSZ constant, 710–711, 719, 730, 897
TIOCPKT constant, 740
TIOCREMOTE constant, 741
TIOCSCTTY constant, 297–298, 727–728
TIOCSIG constant, 741
TIOCSIGNAL constant, 741
TIOCSWINSZ constant, 710, 718, 728, 741
tip program, 713
tm structure, 191, 194, 408, 919
TMPDIR environment variable, 211
tmpfile function, 167–171, 366, 452
definition of, 167
TMP_MAX constant, 38, 168
tmpnam function, 38, 167–171, 442
definition of, 167
tms structure, 280–281
TOCTTOU error, 65, 250, 953
Torvalds, L., 35
TOSTOP constant, 676, 691
touch program, 127
tracing system calls, 497
transactions, database, 952
TRAP_BRKPT constant, 353
TRAP_TRACE constant, 353
tread function, 800, 805–806, 825, 838–839
definition of, 805
treadn function, 800, 806, 824
definition of, 806
Trickey, H., 229, 952
truncate function, 112, 121, 125, 474
definition of, 112
truncation
file, 112
filename, 65–66
pathname, 65–66
truss program, 497
ttcompat STREAMS module, 716, 726
tty structure, 311
tty_atexit function, 705, 731, 897
definition of, 708
tty_cbreak function, 704, 709, 897
definition of, 705
ttymon program, 290
ttyname function, 137, 276, 442, 452, 695–696, 699
definition of, 695, 698
TTY_NAME_MAX constant, 40, 43, 49
ttyname_r function, 443, 452
tty_raw function, 704, 709, 713, 731, 897
definition of, 706
tty_reset function, 704, 709, 897
definition of, 707
tty_termios function, 705, 897
definition of, 708
type attribute, 431
typescript file, 719, 737
TZ environment variable, 190, 192, 195–196, 211, 919
TZNAME_MAX constant, 40, 43, 49
tzset function, 452

Ubuntu, xxii, 7, 26, 35, 290
UCHAR_MAX constant, 37–38
ucontext_t structure, 352
ucred structure, 649, 651
UFS file system, 49, 57, 65, 113, 116, 129
UID, *see* user ID
uid_t data type, 59
uint16_t data type, 595
uint32_t data type, 595
UINT_MAX constant, 37–38
ulimit program, 53, 222
ULLONG_MAX constant, 37
ULONG_MAX constant, 37
UltraSPARC, xxii, xxvii
umask function, 104–107, 222, 331, 466–467
definition of, 104
umask program, 105, 141
uname function, 187, 196, 331
definition of, 187
uname program, 188, 196
unbuffered I/O, 8, 61–91
unbuffered I/O timing, standard I/O versus, 155
ungetc function, 151–152, 452
definition of, 151
ungetwc function, 452
uninitialized data segment, 205
<unistd.h> header, 9, 29, 53, 62, 110, 442, 501, 755, 895
UNIX Architecture, 1–2
UNIX domain sockets, 629–642
timing, 565
UNIX System implementations, 33
Unix-to-Unix Copy, *see* UUCP
UnixWare, 35

unlink function, 114, 116–119, 121–122, 125, 141, 169–170, 331, 366, 452, 497, 553, 637, 639, 641, 823, 826–827, 837, 909, 911, 937, 942
  definition of, 117
unlinkat function, 116–119, 331, 452
  definition of, 117
un_lock function, 489, 759–760, 762, 768, 770–771, 773, 777–778, 780, 897
unlockpt function, 723–725
  definition of, 723
Unrau, R., 174, 531, 950
unreliable signals, 326–327
unsetenv function, 212, 442
  definition of, 212
update program, 81
update_jobno function, 814, 819, 832, 843
  definition of, 819
Upstart, 290
uptime program, 614–615, 617, 619–620, 622–623, 628
__USE_BSD constant, 473
USER environment variable, 210, 288
user ID, 16, 255–260
  effective, 98–99, 101–102, 106, 110, 126, 140, 228, 233, 253, 256–260, 276, 286, 288, 337, 381, 558, 562, 568, 573, 586–587, 637, 640, 809, 918
  real, 39–40, 43, 98–99, 102, 221, 228, 233, 252–253, 256–260, 270, 276, 286, 288, 337, 381, 585, 924
USHRT_MAX constant, 37
usleep function, 532, 934
UTC (Coordinated Universal Time), 20, 189, 192, 196
utime function, 127, 331, 910
UTIME_NOW constant, 126
utimensat function, 125–128, 331, 452, 910
  definition of, 126
UTIME_OMIT constant, 126–127
utimes function, 125–128, 141, 331, 452, 910
  definition of, 127
utmp file, 186–187, 276, 312, 734, 923, 930
utmp structure, 187
utmpx file, 187
<utmpx.h> header, 30
utsname structure, 187–188, 196
UUCP (Unix-to-Unix Copy), 188
uucp program, 500

V7, 329, 726
va_arg function, 758
va_end function, 758, 899–903
va_list data type, 758, 899–903
/var/account/acct file, 269
/var/adm/pacct file, 269
<varargs.h> header, 162
variables
  automatic, 205, 215, 217, 219, 226
  global, 219
  register, 217
  static, 219
  volatile, 217, 219, 340, 357
/var/log/account/pacct file, 269
/var/log/wtmp file, 187
/var/run/utmp file, 187
va_start function, 758, 899–903
VDISCARD constant, 678
vdprintf function, 161, 452
  definition of, 161
VDSUSP constant, 678
VEOF constant, 678–679, 704
VEOL constant, 678, 704
VEOL2 constant, 678
VERASE constant, 678
VERASE2 constant, 678
vfork function, 229, 234–236, 283, 921–922
vfprintf function, 161, 452
  definition of, 161
vfscanf function, 163
  definition of, 163
vfwprintf function, 452
vi program, 377, 497, 499, 672, 711–713, 943
VINTR constant, 678–679
vipw program, 179
VKILL constant, 678
VLNEXT constant, 678
VMIN constant, 703–705, 707
v-node, 74–76, 78, 136, 312, 642, 907, 950
vnode structure, 311–312
Vo, K. P., 135, 174, 949–950, 953
volatile variables, 217, 219, 340, 357
vprintf function, 161, 452
  definition of, 161
VQUIT constant, 678
vread function, 525
VREPRINT constant, 678
vscanf function, 163
  definition of, 163
vsnprintf function, 161, 901
  definition of, 161
vsprintf function, 161, 471
  definition of, 161
vsscanf function, 163
  definition of, 163
VSTART constant, 678

VSTATUS constant, 678
VSTOP constant, 678
VSUSP constant, 678
vsyslog function, 472
  definition of, 472
VT0 constant, 691
VT1 constant, 691
VTDLY constant, 676, 684, 689, 691
VTIME constant, 703–705, 707
VWERASE constant, 678
vwprintf function, 452
vwrite function, 525

wait function, 231–232, 237–246, 249, 255, 264, 267, 280, 282–283, 301, 317, 328–329, 331, 333–335, 351, 368, 371–372, 451, 471, 499, 546, 588, 936
  definition of, 238
Wait, J. W., xxxii
wait3 function, 245
  definition of, 245
wait4 function, 245
  definition of, 245
WAIT_CHILD function, 247, 362, 491, 532, 539, 577, 898, 934
  definition of, 363, 540
waitid function, 244–245, 283, 451
  definition of, 244
WAIT_PARENT function, 247–248, 362, 491, 498, 532, 539, 577, 898
  definition of, 363, 540
waitpid function, 11–13, 19, 23, 237–245, 254, 261, 265–267, 282, 285, 294, 301, 315, 329, 331, 370–371, 451, 498, 538, 545–546, 587–588, 618, 935, 937, 939
  definition of, 238
wall program, 723
wc program, 112
<wchar.h> header, 27, 144
wchar_t data type, 59
WCONTINUED constant, 242, 244
WCOREDUMP function, 239–240
wcrtomb function, 442
wcsftime function, 452
wcsrtombs function, 442
wcstombs function, 442
wctomb function, 442
<wctype.h> header, 27
Weeks, M. S., 206, 949
Wei, J., 65, 953
Weinberger, P. J., 76, 262, 743, 947, 953
Weinstock, C. B., 953
WERASE terminal character, 678, 682, 685–687, 703
WEXITED constant, 244
WEXITSTATUS function, 239–240
who program, 187, 734
WIFCONTINUED function, 239
WIFEXITED function, 239–240
WIFSIGNALED function, 239–240
WIFSTOPPED function, 239–240, 242
Williams, T., 310, 953
Wilson, G. A., xxxii
window size
  pseudo terminal, 741
  terminal, 311, 322, 710–712, 718, 727, 741–742
winsize structure, 311, 710–711, 727, 730, 732, 742, 897, 944
Winterbottom, P., 229, 952
WNOHANG constant, 242, 244
WNOWAIT constant, 242, 244
W_OK constant, 102
Wolff, R., xxxii
Wolff, S., xxxii
WORD_BIT constant, 38
wordexp function, 452
<wordexp.h> header, 29
worker_thread structure, 812–813, 828–829
working directory, *see* current directory
worm, Internet, 153
wprintf function, 452
Wright, G. R., xxxii
write
  delayed, 81
  gather, 521, 644
  synchronous, 63, 86–87
write program, 723
write function, 8–10, 20–21, 59, 61, 63–64, 68–69, 72, 77–79, 86–88, 90, 125, 145–146, 156, 167, 174, 230–231, 234, 247, 328–329, 331, 342–343, 378, 382, 451, 474, 482–484, 491, 495–498, 502, 505, 509, 513, 517, 522–526, 530–532, 537–538, 540, 549–551, 553, 555, 560, 565, 587, 590, 592, 610, 614, 620, 643, 654–655, 672, 752, 760, 773, 810, 819, 826, 836, 907–908, 921, 925, 934, 936–937, 945
  definition of, 72
write_lock function, 489, 493, 498, 818, 897
writen function, 523–524, 644, 732–733, 738, 810–811, 824–827, 836, 896
  definition of, 523–524
writev function, 41, 43, 329, 451, 481, 521–523, 531–532, 592, 611, 644, 655, 660, 752, 771, 773, 832, 836
  definition of, 521

writew_lock function, 489, 491, 759, 763, 769, 771–772, 777, 787, 897
wscanf function, 452
WSTOPPED constant, 244
WSTOPSIG function, 239–240
WTERMSIG function, 239–240
wtmp file, 186–187, 312, 923
Wulf, W. A., 953
WUNTRACED constant, 242

x86, xxi
xargs program, 252
XCASE constant, 691
Xenix, 33, 485, 726
xinetd program, 293
X_OK constant, 102
X/Open, xxvi, 31, 953
X/Open Curses, 32
X/Open Portability Guide, 31–32
  Issue 3, *see* XPG3
  Issue 4, *see* XPG4
_XOPEN_CRYPT constant, 31, 57
_XOPEN_IOV_MAX constant, 41
_XOPEN_NAME_MAX constant, 41
_XOPEN_PATH_MAX constant, 41
_XOPEN_REALTIME constant, 31, 57
_XOPEN_REALTIME_THREADS constant, 31, 57
_XOPEN_SHM constant, 57
_XOPEN_SOURCE constant, 57–58
_XOPEN_UNIX constant, 30–31, 57
_XOPEN_VERSION constant, 57
XPG3 (X/Open Portability Guide, Issue 3), xxxi, 33, 953
XPG4 (X/Open Portability Guide, Issue 4), 32, 54
XSI, 30–31, 53–54, 57, 94, 107, 109, 131–132, 143, 161, 163, 168–169, 180, 183, 211–212, 220, 222, 239, 242, 244–245, 252, 257, 276, 293, 315, 317, 322, 329, 333, 350–352, 377, 429, 431, 442, 469–472, 485, 521, 526, 528, 534, 553, 562–563, 566, 571, 576, 578, 587–588, 666, 676, 685, 687, 689–691, 722, 724, 744, 910
XSI IPC, 556–560
XTABS constant, 690–691

Yigit, O., 744, 952

zombie, 237–238, 242, 283, 333, 351, 923

This page intentionally left blank



Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **informit.com/register** to sign in or create an account. You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

informIT.com
THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram
IBM Press | Que | Prentice Hall | Sams

SAFARI BOOKS ONLINE

informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE

PEARSON

InformIT is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the top brands, authors, and contributors from the tech community.

Addison-Wesley Cisco Press EXAM/CRAM IBM Press. QUE PRENTICE HALL SAMS | Safari Books Online

LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials? Looking for expert opinions, advice, and tips? **InformIT has the solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of newsletters. Visit **informit.com/newsletters**.
- Access FREE podcasts from experts at **informit.com/podcasts**.
- Read the latest author articles and sample chapters at **informit.com/articles**.
- Access thousands of books and videos in the Safari Books Online digital library at **safari.informit.com**.
- Get tips from expert blogs at **informit.com/blogs**.

Visit **informit.com/learn** to discover all the ways you can access the hottest technology content.

Are You Part of the IT Crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube, and more! Visit **informit.com/socialconnect**.



informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE

PEARSON

Addison-Wesley Cisco Press EXAM/CRAM IBM Press. QUE PRENTICE HALL SAMS | Safari Books Online

Try Safari Books Online FREE for 15 days

Get online access to Thousands of Books and Videos



Safari Books Online

FREE 15-DAY TRIAL + 15% OFF*
informit.com/safaritrial

Feed your brain

Gain unlimited access to thousands of books and videos about technology, digital media and professional development from O'Reilly Media, Addison-Wesley, Microsoft Press, Cisco Press, McGraw Hill, Wiley, WROX, Prentice Hall, Que, Sams, Apress, Adobe Press and other top publishers.

See it, believe it

Watch hundreds of expert-led instructional videos on today's hottest topics.

WAIT, THERE'S MORE!

Gain a competitive edge

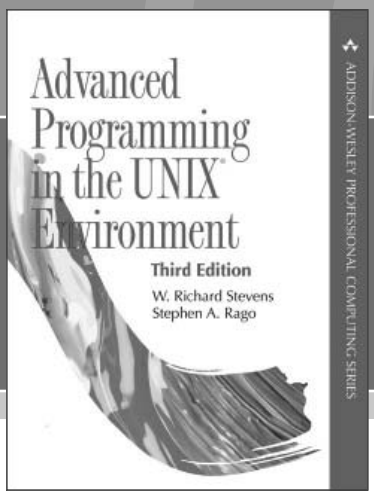
Be first to learn about the newest technologies and subjects with Rough Cuts pre-published manuscripts and new technology overviews in Short Cuts.

Accelerate your project

Copy and paste code, create smart searches that let you know when new books about your favorite topics are available, and customize your library with favorites, highlights, tags, notes, mash-ups and more.

* Available to new subscribers only. Discount applies to the Safari Library and is valid for first 12 consecutive monthly billing cycles. Safari Library is not available in all countries.

Addison Wesley · Adobe Press · Cisco Press · FT Press · IBM Press · Microsoft Press · New Riders · O'REILLY · Peachpit Press · PEARSON IT Certification · PRENTICE HALL · que · SAMS · vmware PRESS · WILEY · wrox



Safari Books Online

FREE Online Edition

Your purchase of ***Advanced Programming in the UNIX® Environment, Third Edition,*** includes access to a free online edition for 45 days through the **Safari Books Online** subscription service. Nearly every Addison-Wesley Professional book is available online through **Safari Books Online**, along with thousands of books and videos from publishers such as Cisco Press, Exam Cram, IBM Press, O'Reilly Media, Prentice Hall, Que, Sams, and VMware Press.

Safari Books Online is a digital library providing searchable, on-demand access to thousands of technology, digital media, and professional development books and videos from leading publishers. With one monthly or yearly subscription price, you get unlimited access to learning tools and information on topics including mobile app and software development, tips and tricks on using your favorite gadgets, networking, project management, graphic design, and much more.

Activate your FREE Online Edition at informit.com/safarifree

STEP 1: Enter the coupon code: LDMRWBI.

STEP 2: New Safari users, complete the brief registration form. Safari subscribers, just log in.

If you have difficulty registering on Safari or accessing the online edition, please e-mail customer-service@safaribooksonline.com

Addison Wesley · Adobe Press · Alpha · Cisco Press · FT Press · IBM Press · Microsoft Press · New Riders · O'Reilly · Peachpit Press · Prentice Hall · Que · Redbooks · Sams · SAS Publishing · VMware Press · Wiley · Wrox



JOIN THE INFORMIT AFFILIATE TEAM!

and you love to share them with your colleagues and friends...why not earn some $$ doing it!

If you have a website, blog, or even a Facebook page, you can start earning money by putting InformIT links on your page.

Whenever a visitor clicks on these links and makes a purchase on informit.com, you earn commissions* on all sales!

Every sale you bring to our site will earn you a commission. All you have to do is post the links to the titles you want, as many as you want, and we'll take care of the rest.

It's quick and easy to apply.
To learn more go to:
http://www.informit.com/affiliates/

*Valid for all books, eBooks and video sales at www.informit.com

Addison Wesley PRENTICE HALL SAMS IT