

# Advanced Programming in the UNIX<sup>®</sup> Environment

**Third Edition**

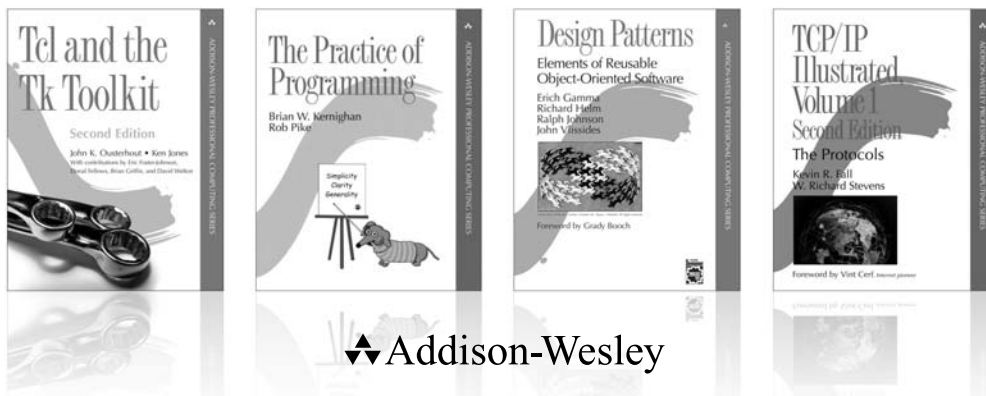
W. Richard Stevens  
Stephen A. Rago

FREE SAMPLE CHAPTER



SHARE WITH OTHERS

# The Addison-Wesley Professional Computing Series



Visit [informit.com/series/professionalcomputing](http://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](http://informit.com/socialconnect)



**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.”*

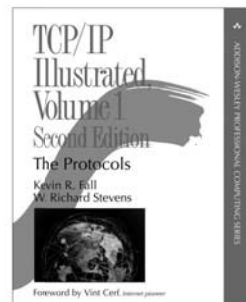
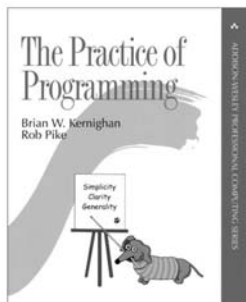
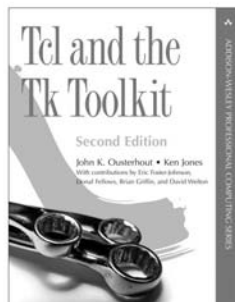
—*Open Systems Today*

*“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<sup>®</sup> Environment**  
Third Edition

# The Addison-Wesley Professional Computing Series



◆◆ Addison-Wesley

Visit [informit.com/series/professionalcomputing](http://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](http://informit.com/socialconnect)



**informIT.com**  
THE TRUSTED TECHNOLOGY LEARNING SOURCE

**Safari**  
Books Online

# **Advanced Programming in the UNIX<sup>®</sup> 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](http://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***

<b>Foreword to the Second Edition</b>	<b>xix</b>
<b>Preface</b>	<b>xxi</b>
<b>Preface to the Second Edition</b>	<b>xxv</b>
<b>Preface to the First Edition</b>	<b>xxix</b>
<b>Chapter 1. UNIX System Overview</b>	<b>1</b>
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
<b>Chapter 2. UNIX Standardization and Implementations</b>	<b>25</b>
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	
<b>Chapter 3.</b>	<b>File I/O</b>		<b>61</b>
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	<code>ioctl</code> Function	87
3.16	<code>/dev/fd</code>	88
3.17	Summary	90
<b>Chapter 4.</b>	<b>Files and Directories</b>	<b>93</b>
4.1	Introduction	93
4.2	<code>stat</code> , <code>fstat</code> , <code>fstatat</code> , and <code>lstat</code> 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	<code>access</code> and <code>faccessat</code> Functions	102
4.8	<code>umask</code> Function	104
4.9	<code>chmod</code> , <code>fchmod</code> , and <code>fchmodat</code> Functions	106
4.10	Sticky Bit	108
4.11	<code>chown</code> , <code>fchown</code> , <code>fchownat</code> , and <code>lchown</code> Functions	109
4.12	File Size	111
4.13	File Truncation	112
4.14	File Systems	113
4.15	<code>link</code> , <code>linkat</code> , <code>unlink</code> , <code>unlinkat</code> , and <code>remove</code> Functions	116
4.16	<code>rename</code> and <code>renameat</code> Functions	119
4.17	Symbolic Links	120
4.18	Creating and Reading Symbolic Links	123
4.19	File Times	124
4.20	<code>futimens</code> , <code>utimensat</code> , and <code>utimes</code> Functions	126
4.21	<code>mkdir</code> , <code>mkdirat</code> , and <code>rmdir</code> Functions	129
4.22	Reading Directories	130
4.23	<code>chdir</code> , <code>fchdir</code> , and <code>getcwd</code> Functions	135
4.24	Device Special Files	137
4.25	Summary of File Access Permission Bits	140
4.26	Summary	140
<b>Chapter 5.</b>	<b>Standard I/O Library</b>	<b>143</b>
5.1	Introduction	143
5.2	Streams and <code>FILE</code> 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
<b>Chapter 6.</b>	<b>System Data Files and Information</b>	<b>177</b>
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
<b>Chapter 7.</b>	<b>Process Environment</b>	<b>197</b>
7.1	Introduction	197
7.2	<code>main</code> 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	<code>setjmp</code> and <code>longjmp</code> Functions	213
7.11	<code>getrlimit</code> and <code>setrlimit</code> Functions	220
7.12	Summary	225
<b>Chapter 8.</b>	<b>Process Control</b>	<b>227</b>
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	
<b>Chapter 9.</b>	<b>Process Relationships</b>		<b>285</b>
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	
<b>Chapter 10.</b>	<b>Signals</b>		<b>313</b>
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
<b>Chapter 11.</b>	<b>Threads</b>	<b>383</b>
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
<b>Chapter 12.</b>	<b>Thread Control</b>	<b>425</b>
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 <code>fork</code>	457
12.10	Threads and I/O	461
12.11	Summary	462
<b>Chapter 13.</b>	<b>Daemon Processes</b>	<b>463</b>
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
<b>Chapter 14.</b>	<b>Advanced I/O</b>	<b>481</b>
14.1	Introduction	481
14.2	Nonblocking I/O	481
14.3	Record Locking	485
14.4	I/O Multiplexing	500
14.4.1	<code>select</code> and <code>pselect</code> Functions	502
14.4.2	<code>poll</code> 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	<code>readv</code> and <code>writew</code> Functions	521
14.7	<code>readn</code> and <code>writen</code> Functions	523
14.8	Memory-Mapped I/O	525
14.9	Summary	531
<b>Chapter 15.</b>	<b>Interprocess Communication</b>	<b>533</b>
15.1	Introduction	533
15.2	Pipes	534
15.3	<code>popen</code> and <code>pclose</code> 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	<code>stty</code> 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	<code>termcap</code> , <code>terminfo</code> , and <code>curses</code>	712	
18.14	Summary	713	
<b>Chapter 19.</b>	<b>Pseudo Terminals</b>		<b>715</b>
19.1	Introduction	715	
19.2	Overview	715	
19.3	Opening Pseudo-Terminal Devices	722	
19.4	<code>pty_fork</code> Function	726	
19.5	<code>pty</code> Program	729	
19.6	Using the <code>pty</code> Program	733	
19.7	Advanced Features	740	
19.8	Summary	741	
<b>Chapter 20.</b>	<b>A Database Library</b>		<b>743</b>
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	
<b>Chapter 21.</b>	<b>Communicating with a Network Printer</b>		<b>789</b>
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	
<b>Appendix A. Function Prototypes</b>			<b>845</b>
<b>Appendix B. Miscellaneous Source Code</b>			<b>895</b>
B.1	Our Header File	895	
B.2	Standard Error Routines	898	
<b>Appendix C. Solutions to Selected Exercises</b>			<b>905</b>
<b>Bibliography</b>			<b>947</b>
<b>Index</b>			<b>955</b>

# ***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*  
*March 2005*

Dennis Ritchie

# 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 Domaig  , 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](http://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](mailto: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](http://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](mailto: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.

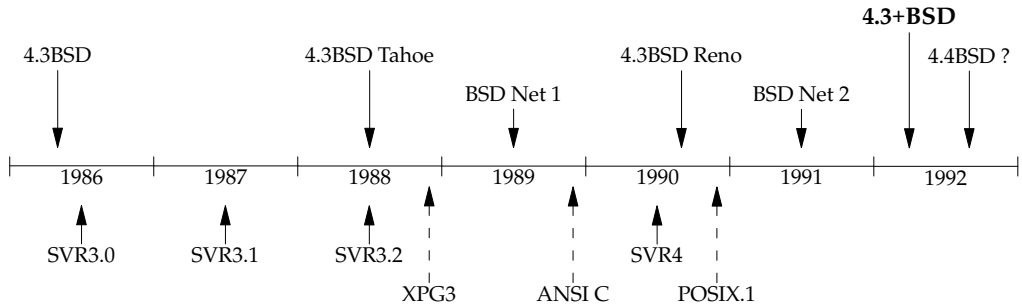
Almost every function and system call is demonstrated with a small, complete program. This lets us see the arguments and return values and is often easier to comprehend than the use of the function in a much larger program. But since some of the small programs are contrived examples, a few bigger examples 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. SVR $x$  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 Saksen 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>

# 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

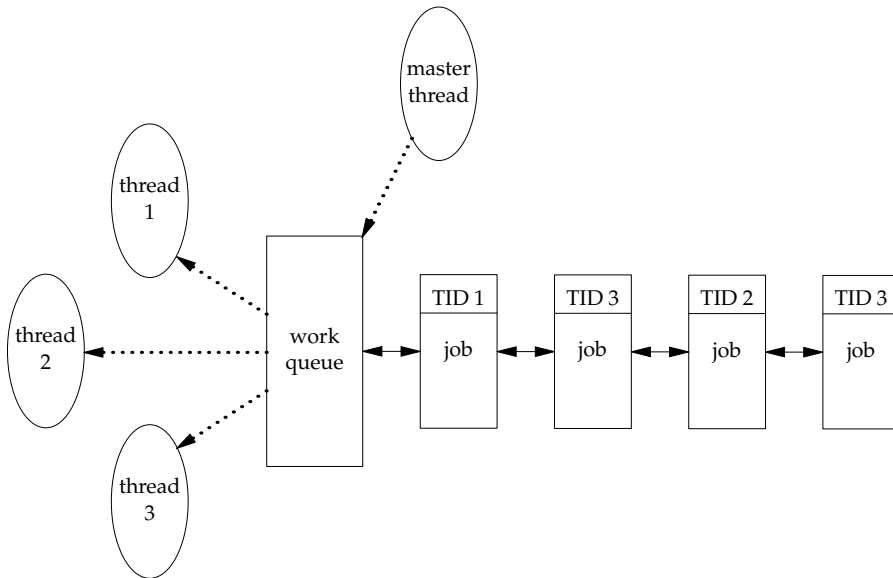


Figure 11.1 Work queue example

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

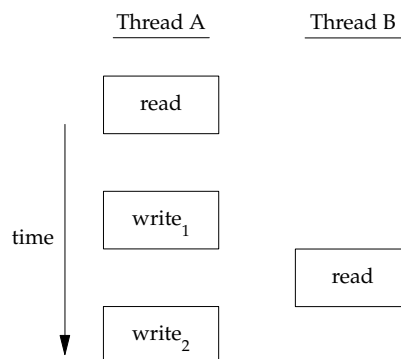
As we will see in the next chapter, we can create a thread that is already in the detached state by modifying the thread attributes we pass to `pthread_create`.

## 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.

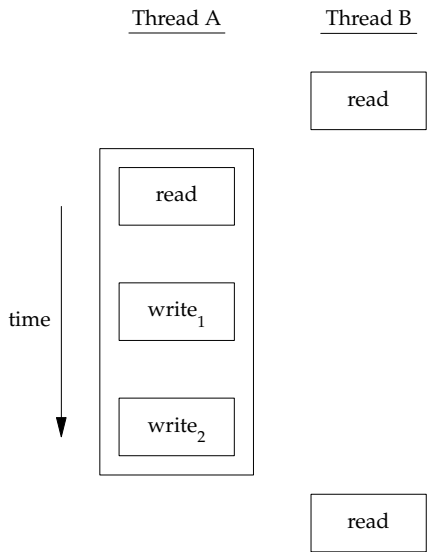
Figure 11.7 shows a hypothetical example of two threads reading and writing the same variable. In this example, thread A reads the variable and then writes a new value to it, but the write operation takes two memory cycles. If thread B reads the same variable between the two write cycles, it will see 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.

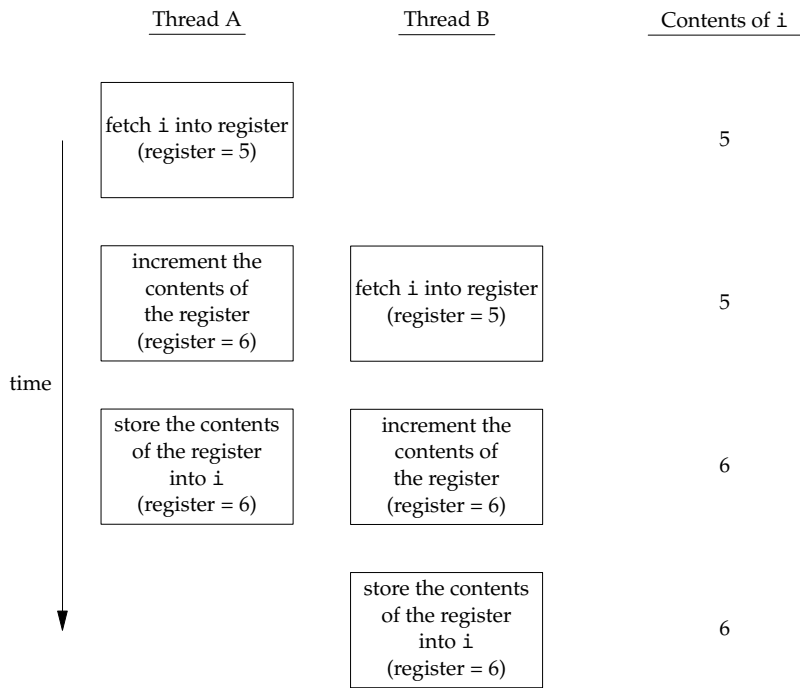


Figure 11.9 Two unsynchronized threads incrementing the same variable

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 tspr);

```

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 rlock,
                               const struct timespec *restrict tspr);

int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rlock,
                               const struct timespec *restrict tspr);
```

Both return: 0 if OK, error number on failure

These functions behave like their “untimed” counterparts. The *tspr* 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*

# 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\_ADRLN constant, 353
- BUS\_ADREERR 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
- 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
- CIBAUEXT 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
- cmmsgcred structure, 648–651
- MSG\_DATA function, 645–646, 648, 650, 652
  - definition of, 645
- MSG\_FIRSTHDR function, 645, 652
  - definition of, 645
- msgshdr structure, 645–647, 649, 651
- MSG\_LEN function, 645–647, 649, 651
  - definition of, 645
- MSG\_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
- ENOBUFFS 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
  - DATMSK, 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
- EWOLDBLOCK 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
- `exec1` 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
- `exec1p` 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
- `forkexecve` 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
  - definition of, 597
- `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
- getservernt 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
- IDXLLEN\_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

- 164a 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., xxii
- 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
- rand48 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
  - nfdst\_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
- perorr 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
- POLLRBAND 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_spawnnp` 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_SYMLINK_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
- prntd 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
- threads, 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
- 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
- Quarterman, J. S., 33–34, 74, 112, 116, 229, 236, 525, 951
- QUIT terminal character, 678, 681, 688, 702

- 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
- regexexec 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
- rlim\_t 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
- Saksen, 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\_RT\_SIG\_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\_SYMLINK\_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\_MSGQ 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
- SIOCSGRP 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
- socketmark 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
- SYMLINK\_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
- `tellldir` 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
- writew 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
- 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
- vfprintf 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
- WERASE 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
- wctomb 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
- written 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