

THE **LINUX** PROGRAMMING INTERFACE

A Linux and UNIX® System Programming Handbook

MICHAEL KERRISK



PRAISE FOR *THE LINUX PROGRAMMING INTERFACE*

“If I had to choose a single book to sit next to my machine when writing software for Linux, this would be it.”

—MARTIN LANDERS, SOFTWARE ENGINEER, GOOGLE

“This book, with its detailed descriptions and examples, contains everything you need to understand the details and nuances of the low-level programming APIs in Linux . . . no matter what the level of reader, there will be something to be learnt from this book.”

—MEL GORMAN, AUTHOR OF *Understanding the Linux Virtual Memory Manager*

“Michael Kerrisk has not only written a great book about Linux programming and how it relates to various standards, but has also taken care that bugs he noticed got fixed and the man pages were (greatly) improved. In all three ways, he has made Linux programming easier. The in-depth treatment of topics in *The Linux Programming Interface* . . . makes it a must-have reference for both new and experienced Linux programmers.”

—ANDREAS JAEGER, PROGRAM MANAGER, OPENSUSE, NOVELL

“Michael’s inexhaustible determination to get his information right, and to express it clearly and concisely, has resulted in a strong reference source for programmers. While this work is targeted at Linux programmers, it will be of value to any programmer working in the UNIX/POSIX ecosystem.”

—DAVID BUTENHOF, AUTHOR OF *Programming with POSIX Threads* AND CONTRIBUTOR TO THE POSIX AND UNIX STANDARDS

“ . . . a very thorough—yet easy to read—explanation of UNIX system and network programming, with an emphasis on Linux systems. It’s certainly a book I’d recommend to anybody wanting to get into UNIX programming (in general) or to experienced UNIX programmers wanting to know ‘what’s new’ in the popular GNU/Linux system.”

—FERNANDO GONT, NETWORK SECURITY RESEARCHER, IETF PARTICIPANT, AND RFC AUTHOR

“ . . . encyclopedic in the breadth and depth of its coverage, and textbook-like in its wealth of worked examples and exercises. Each topic is clearly and comprehensively covered, from theory to hands-on working code. Professionals, students, educators, this is the Linux/UNIX reference that you have been waiting for.”

—ANTHONY ROBINS, ASSOCIATE PROFESSOR OF COMPUTER SCIENCE, THE UNIVERSITY OF OTAGO

“I’ve been very impressed by the precision, the quality and the level of detail Michael Kerrisk put in his book. He is a great expert of Linux system calls and lets us share his knowledge and understanding of the Linux APIs.”

—CHRISTOPHE BLAESS, AUTHOR OF *Programmation système en C sous Linux*

“ . . . an essential resource for the serious or professional Linux and UNIX systems programmer. Michael Kerrisk covers the use of all the key APIs across both the Linux and UNIX system interfaces with clear descriptions and tutorial examples and stresses the importance and benefits of following standards such as the Single UNIX Specification and POSIX 1003.1.”

—ANDREW JOSEY, DIRECTOR, STANDARDS, THE OPEN GROUP, AND CHAIR OF THE POSIX 1003.1 WORKING GROUP

“What could be better than an encyclopedic reference to the Linux system, from the standpoint of the system programmer, written by none other than the maintainer of the man pages himself? *The Linux Programming Interface* is comprehensive and detailed. I firmly expect it to become an indispensable addition to my programming bookshelf.”

—BILL GALLMEISTER, AUTHOR OF *POSIX.4 Programmer’s Guide: Programming for the Real World*

“ . . . the most complete and up-to-date book about Linux and UNIX system programming. If you’re new to Linux system programming, if you’re a UNIX veteran focused on portability while interested in learning the Linux way, or if you’re simply looking for an excellent reference about the Linux programming interface, then Michael Kerrisk’s book is definitely the companion you want on your bookshelf.”

—LOÏC DOMAIGNÉ, CHIEF SOFTWARE ARCHITECT (EMBEDDED), CORPULS.COM

THE LINUX PROGRAMMING INTERFACE

A Linux and UNIX® System Programming Handbook

MICHAEL KERRISK



**no starch
press**

San Francisco

THE LINUX PROGRAMMING INTERFACE. Copyright © 2010 by Michael Kerrisk.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

14 13 12 11 10 1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-220-0

ISBN-13: 978-1-59327-220-3

Publisher: William Pollock

Production Editor: Riley Hoffman

Cover Design: Octopod Studios

Front Cover Photo: Rob Suisted

Back Cover Photo: Lynley Cook

Copyeditor: Marilyn Smith

Compositor: Susan Glinert Stevens

Proofreader: Linda Seifert

For technical reviewers, please refer to the author's acknowledgements

For information on book distributors or translations, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.

38 Ringold Street, San Francisco, CA 94103

phone: 415.863.9900; fax: 415.863.9950; info@nostarch.com; www.nostarch.com

Library of Congress Cataloging-in-Publication Data

Kerrisk, Michael, 1961-

The Linux programming interface : a Linux and UNIX system programming handbook / by Michael Kerrisk.
p. cm.

Includes bibliographical references and index.

ISBN-13: 978-1-59327-220-3 (hardcover)

ISBN-10: 1-59327-220-0 (hardcover)

1. Linux. 2. UNIX (Computer file) 3. Operating systems (Computers) I. Title.

QA76.76.063K496 2010

005.4'32--dc22

2010001947

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

UNIX® is a registered trademark of The Open Group.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

This logo applies only to the text stock.

For Cecilia, who lights up my world.

BRIEF CONTENTS

Preface	xxxi
Chapter 1: History and Standards	1
Chapter 2: Fundamental Concepts	21
Chapter 3: System Programming Concepts	43
Chapter 4: File I/O: The Universal I/O Model	69
Chapter 5: File I/O: Further Details	89
Chapter 6: Processes	113
Chapter 7: Memory Allocation	139
Chapter 8: Users and Groups	153
Chapter 9: Process Credentials	167
Chapter 10: Time	185
Chapter 11: System Limits and Options	211
Chapter 12: System and Process Information	223
Chapter 13: File I/O Buffering	233
Chapter 14: File Systems	251
Chapter 15: File Attributes	279
Chapter 16: Extended Attributes	311
Chapter 17: Access Control Lists	319
Chapter 18: Directories and Links	339
Chapter 19: Monitoring File Events	375

Chapter 20: Signals: Fundamental Concepts	387
Chapter 21: Signals: Signal Handlers.....	421
Chapter 22: Signals: Advanced Features	447
Chapter 23: Timers and Sleeping.....	479
Chapter 24: Process Creation.....	513
Chapter 25: Process Termination.....	531
Chapter 26: Monitoring Child Processes	541
Chapter 27: Program Execution	563
Chapter 28: Process Creation and Program Execution in More Detail.....	591
Chapter 29: Threads: Introduction.....	617
Chapter 30: Threads: Thread Synchronization	631
Chapter 31: Threads: Thread Safety and Per-Thread Storage.....	655
Chapter 32: Threads: Thread Cancellation	671
Chapter 33: Threads: Further Details	681
Chapter 34: Process Groups, Sessions, and Job Control	699
Chapter 35: Process Priorities and Scheduling	733
Chapter 36: Process Resources	753
Chapter 37: Daemons	767
Chapter 38: Writing Secure Privileged Programs	783
Chapter 39: Capabilities	797
Chapter 40: Login Accounting	817
Chapter 41: Fundamentals of Shared Libraries	833
Chapter 42: Advanced Features of Shared Libraries.....	859
Chapter 43: Interprocess Communication Overview	877
Chapter 44: Pipes and FIFOs.....	889
Chapter 45: Introduction to System V IPC.....	921
Chapter 46: System V Message Queues	937

Chapter 47: System V Semaphores	965
Chapter 48: System V Shared Memory	997
Chapter 49: Memory Mappings	1017
Chapter 50: Virtual Memory Operations	1045
Chapter 51: Introduction to POSIX IPC	1057
Chapter 52: POSIX Message Queues	1063
Chapter 53: POSIX Semaphores	1089
Chapter 54: POSIX Shared Memory	1107
Chapter 55: File Locking	1117
Chapter 56: Sockets: Introduction	1149
Chapter 57: Sockets: UNIX Domain	1165
Chapter 58: Sockets: Fundamentals of TCP/IP Networks	1179
Chapter 59: Sockets: Internet Domains	1197
Chapter 60: Sockets: Server Design	1239
Chapter 61: Sockets: Advanced Topics	1253
Chapter 62: Terminals	1289
Chapter 63: Alternative I/O Models	1325
Chapter 64: Pseudoterminals	1375
Appendix A: Tracing System Calls	1401
Appendix B: Parsing Command-Line Options	1405
Appendix C: Casting the NULL Pointer	1413
Appendix D: Kernel Configuration	1417
Appendix E: Further Sources of Information	1419
Appendix F: Solutions to Selected Exercises	1425
Bibliography	1437
Index	1447

CONTENTS IN DETAIL

PREFACE

xxxi

1	HISTORY AND STANDARDS	1
1.1	A Brief History of UNIX and C	2
1.2	A Brief History of Linux	5
1.2.1	The GNU Project	5
1.2.2	The Linux Kernel	6
1.3	Standardization	10
1.3.1	The C Programming Language	10
1.3.2	The First POSIX Standards	11
1.3.3	X/Open Company and The Open Group	13
1.3.4	SUSv3 and POSIX.1-2001	13
1.3.5	SUSv4 and POSIX.1-2008	15
1.3.6	UNIX Standards Timeline	16
1.3.7	Implementation Standards	17
1.3.8	Linux, Standards, and the Linux Standard Base	18
1.4	Summary	19
2	FUNDAMENTAL CONCEPTS	21
2.1	The Core Operating System: The Kernel	21
2.2	The Shell	24
2.3	Users and Groups	26
2.4	Single Directory Hierarchy, Directories, Links, and Files	27
2.5	File I/O Model	29
2.6	Programs	30
2.7	Processes	31
2.8	Memory Mappings	35
2.9	Static and Shared Libraries	35
2.10	Interprocess Communication and Synchronization	36
2.11	Signals	37
2.12	Threads	38
2.13	Process Groups and Shell Job Control	38
2.14	Sessions, Controlling Terminals, and Controlling Processes	39
2.15	Pseudoterminals	39
2.16	Date and Time	40
2.17	Client-Server Architecture	40
2.18	Realtime	41
2.19	The /proc File System	42
2.20	Summary	42
3	SYSTEM PROGRAMMING CONCEPTS	43
3.1	System Calls	43
3.2	Library Functions	46
3.3	The Standard C Library; The GNU C Library (<i>glibc</i>)	47
3.4	Handling Errors from System Calls and Library Functions	48
3.5	Notes on the Example Programs in This Book	50
3.5.1	Command-Line Options and Arguments	50
3.5.2	Common Functions and Header Files	51

3.6	Portability Issues	61
3.6.1	Feature Test Macros	61
3.6.2	System Data Types	63
3.6.3	Miscellaneous Portability Issues	66
3.7	Summary	68
3.8	Exercise	68
4	FILE I/O: THE UNIVERSAL I/O MODEL	69
4.1	Overview	69
4.2	Universality of I/O	72
4.3	Opening a File: <i>open()</i>	72
4.3.1	The <i>open()</i> <i>flags</i> Argument	74
4.3.2	Errors from <i>open()</i>	77
4.3.3	The <i>creat()</i> System Call	78
4.4	Reading from a File: <i>read()</i>	79
4.5	Writing to a File: <i>write()</i>	80
4.6	Closing a File: <i>close()</i>	80
4.7	Changing the File Offset: <i>lseek()</i>	81
4.8	Operations Outside the Universal I/O Model: <i>ioctl()</i>	86
4.9	Summary	86
4.10	Exercises	87
5	FILE I/O: FURTHER DETAILS	89
5.1	Atomicity and Race Conditions	90
5.2	File Control Operations: <i>fcntl()</i>	92
5.3	Open File Status Flags	93
5.4	Relationship Between File Descriptors and Open Files	94
5.5	Duplicating File Descriptors	96
5.6	File I/O at a Specified Offset: <i>pread()</i> and <i>pwrite()</i>	98
5.7	Scatter-Gather I/O: <i>readv()</i> and <i>writev()</i>	99
5.8	Truncating a File: <i>truncate()</i> and <i>ftruncate()</i>	103
5.9	Nonblocking I/O	103
5.10	I/O on Large Files	104
5.11	The <i>/dev/fd</i> Directory	107
5.12	Creating Temporary Files	108
5.13	Summary	109
5.14	Exercises	110
6	PROCESSES	113
6.1	Processes and Programs	113
6.2	Process ID and Parent Process ID	114
6.3	Memory Layout of a Process	115
6.4	Virtual Memory Management	118
6.5	The Stack and Stack Frames	121
6.6	Command-Line Arguments (<i>argc</i> , <i>argv</i>)	122
6.7	Environment List	125
6.8	Performing a Nonlocal Goto: <i>setjmp()</i> and <i>longjmp()</i>	131
6.9	Summary	138
6.10	Exercises	138

7	MEMORY ALLOCATION	139
7.1	Allocating Memory on the Heap	139
7.1.1	Adjusting the Program Break: <i>brk()</i> and <i>sbrk()</i>	139
7.1.2	Allocating Memory on the Heap: <i>malloc()</i> and <i>free()</i>	140
7.1.3	Implementation of <i>malloc()</i> and <i>free()</i>	144
7.1.4	Other Methods of Allocating Memory on the Heap	147
7.2	Allocating Memory on the Stack: <i>alloca()</i>	150
7.3	Summary	151
7.4	Exercises	152
8	USERS AND GROUPS	153
8.1	The Password File: <i>/etc/passwd</i>	153
8.2	The Shadow Password File: <i>/etc/shadow</i>	155
8.3	The Group File: <i>/etc/group</i>	155
8.4	Retrieving User and Group Information	157
8.5	Password Encryption and User Authentication	162
8.6	Summary	166
8.7	Exercises	166
9	PROCESS CREDENTIALS	167
9.1	Real User ID and Real Group ID	167
9.2	Effective User ID and Effective Group ID	168
9.3	Set-User-ID and Set-Group-ID Programs	168
9.4	Saved Set-User-ID and Saved Set-Group-ID	170
9.5	File-System User ID and File-System Group ID	171
9.6	Supplementary Group IDs	172
9.7	Retrieving and Modifying Process Credentials	172
9.7.1	Retrieving and Modifying Real, Effective, and Saved Set IDs	172
9.7.2	Retrieving and Modifying File-System IDs	178
9.7.3	Retrieving and Modifying Supplementary Group IDs	178
9.7.4	Summary of Calls for Modifying Process Credentials	180
9.7.5	Example: Displaying Process Credentials	182
9.8	Summary	183
9.9	Exercises	184
10	TIME	185
10.1	Calendar Time	186
10.2	Time-Conversion Functions	187
10.2.1	Converting <i>time_t</i> to Printable Form	188
10.2.2	Converting Between <i>time_t</i> and Broken-Down Time	189
10.2.3	Converting Between Broken-Down Time and Printable Form	191
10.3	Timezones	197
10.4	Locales	200
10.5	Updating the System Clock	204
10.6	The Software Clock (Jiffies)	205
10.7	Process Time	206
10.8	Summary	209
10.9	Exercise	210

11	SYSTEM LIMITS AND OPTIONS	211
11.1	System Limits.....	212
11.2	Retrieving System Limits (and Options) at Run Time.....	215
11.3	Retrieving File-Related Limits (and Options) at Run Time.....	217
11.4	Indeterminate Limits	219
11.5	System Options.....	219
11.6	Summary.....	221
11.7	Exercises	222
12	SYSTEM AND PROCESS INFORMATION	223
12.1	The /proc File System.....	223
12.1.1	Obtaining Information About a Process: <i>/proc/PID</i>	224
12.1.2	System Information Under /proc.....	226
12.1.3	Accessing /proc Files	226
12.2	System Identification: <i>uname()</i>	229
12.3	Summary.....	231
12.4	Exercises	231
13	FILE I/O BUFFERING	233
13.1	Kernel Buffering of File I/O: The Buffer Cache	233
13.2	Buffering in the <i>stdio</i> Library	237
13.3	Controlling Kernel Buffering of File I/O.....	239
13.4	Summary of I/O Buffering	243
13.5	Advising the Kernel About I/O Patterns.....	244
13.6	Bypassing the Buffer Cache: Direct I/O.....	246
13.7	Mixing Library Functions and System Calls for File I/O	248
13.8	Summary.....	249
13.9	Exercises	250
14	FILE SYSTEMS	251
14.1	Device Special Files (Devices)	252
14.2	Disks and Partitions	253
14.3	File Systems.....	254
14.4	Inodes.....	256
14.5	The Virtual File System (VFS).....	259
14.6	Journaling File Systems.....	260
14.7	Single Directory Hierarchy and Mount Points	261
14.8	Mounting and Unmounting File Systems	262
14.8.1	Mounting a File System: <i>mount()</i>	264
14.8.2	Unmounting a File System: <i>umount()</i> and <i>umount2()</i>	269
14.9	Advanced Mount Features	271
14.9.1	Mounting a File System at Multiple Mount Points.....	271
14.9.2	Stacking Multiple Mounts on the Same Mount Point.....	271
14.9.3	Mount Flags That Are Per-Mount Options	272
14.9.4	Bind Mounts.....	272
14.9.5	Recursive Bind Mounts.....	273
14.10	A Virtual Memory File System: <i>tmpfs</i>	274
14.11	Obtaining Information About a File System: <i>statvfs()</i>	276
14.12	Summary.....	277
14.13	Exercise	278

15	FILE ATTRIBUTES	279
15.1	Retrieving File Information: <i>stat()</i>	279
15.2	File Timestamps	285
15.2.1	Changing File Timestamps with <i>utime()</i> and <i>utimes()</i>	287
15.2.2	Changing File Timestamps with <i>utimensat()</i> and <i>futimens()</i>	289
15.3	File Ownership	291
15.3.1	Ownership of New Files	291
15.3.2	Changing File Ownership: <i>chown()</i> , <i>fchown()</i> , and <i>lchown()</i>	291
15.4	File Permissions	294
15.4.1	Permissions on Regular Files	294
15.4.2	Permissions on Directories	297
15.4.3	Permission-Checking Algorithm	297
15.4.4	Checking File Accessibility: <i>access()</i>	299
15.4.5	Set-User-ID, Set-Group-ID, and Sticky Bits	300
15.4.6	The Process File Mode Creation Mask: <i>umask()</i>	301
15.4.7	Changing File Permissions: <i>chmod()</i> and <i>fchmod()</i>	303
15.5	I-node Flags (<i>ext2</i> Extended File Attributes)	304
15.6	Summary	308
15.7	Exercises	309
16	EXTENDED ATTRIBUTES	311
16.1	Overview	311
16.2	Extended Attribute Implementation Details	313
16.3	System Calls for Manipulating Extended Attributes	314
16.4	Summary	318
16.5	Exercise	318
17	ACCESS CONTROL LISTS	319
17.1	Overview	320
17.2	ACL Permission-Checking Algorithm	321
17.3	Long and Short Text Forms for ACLs	323
17.4	The <i>ACL_MASK</i> Entry and the ACL Group Class	324
17.5	The <i>getfacl</i> and <i>setfacl</i> Commands	325
17.6	Default ACLs and File Creation	327
17.7	ACL Implementation Limits	328
17.8	The ACL API	329
17.9	Summary	337
17.10	Exercise	337
18	DIRECTORIES AND LINKS	339
18.1	Directories and (Hard) Links	339
18.2	Symbolic (Soft) Links	342
18.3	Creating and Removing (Hard) Links: <i>link()</i> and <i>unlink()</i>	344
18.4	Changing the Name of a File: <i>rename()</i>	348
18.5	Working with Symbolic Links: <i>symlink()</i> and <i>readlink()</i>	349
18.6	Creating and Removing Directories: <i>mkdir()</i> and <i>rmdir()</i>	350
18.7	Removing a File or Directory: <i>remove()</i>	352
18.8	Reading Directories: <i>opendir()</i> and <i>readdir()</i>	352
18.9	File Tree Walking: <i>nftw()</i>	358
18.10	The Current Working Directory of a Process	363
18.11	Operating Relative to a Directory File Descriptor	365
18.12	Changing the Root Directory of a Process: <i>chroot()</i>	367
18.13	Resolving a Pathname: <i>realpath()</i>	369

18.14	Parsing Pathname Strings: <i>dirname()</i> and <i>basename()</i>	370
18.15	Summary.....	372
18.16	Exercises	373

19 MONITORING FILE EVENTS 375

19.1	Overview	376
19.2	The <i>inotify</i> API	376
19.3	<i>inotify</i> Events	378
19.4	Reading <i>inotify</i> Events.....	379
19.5	Queue Limits and <i>/proc</i> Files.....	385
19.6	An Older System for Monitoring File Events: <i>dnotify</i>	386
19.7	Summary.....	386
19.8	Exercise	386

20 SIGNALS: FUNDAMENTAL CONCEPTS 387

20.1	Concepts and Overview	388
20.2	Signal Types and Default Actions	390
20.3	Changing Signal Dispositions: <i>signal()</i>	397
20.4	Introduction to Signal Handlers	398
20.5	Sending Signals: <i>kill()</i>	401
20.6	Checking for the Existence of a Process.....	403
20.7	Other Ways of Sending Signals: <i>raise()</i> and <i>killpg()</i>	404
20.8	Displaying Signal Descriptions.....	406
20.9	Signal Sets	406
20.10	The Signal Mask (Blocking Signal Delivery)	410
20.11	Pending Signals	411
20.12	Signals Are Not Queued.....	412
20.13	Changing Signal Dispositions: <i>sigaction()</i>	416
20.14	Waiting for a Signal: <i>pause()</i>	418
20.15	Summary.....	418
20.16	Exercises	419

21 SIGNALS: SIGNAL HANDLERS 421

21.1	Designing Signal Handlers	422
21.1.1	Signals Are Not Queued (Revisited)	422
21.1.2	Reentrant and Async-Signal-Safe Functions	422
21.1.3	Global Variables and the <i>sig_atomic_t</i> Data Type	428
21.2	Other Methods of Terminating a Signal Handler	428
21.2.1	Performing a Nonlocal Goto from a Signal Handler	429
21.2.2	Terminating a Process Abnormally: <i>abort()</i>	433
21.3	Handling a Signal on an Alternate Stack: <i>sigaltstack()</i>	434
21.4	The <i>SA_SIGINFO</i> Flag.....	437
21.5	Interruption and Restarting of System Calls	442
21.6	Summary.....	445
21.7	Exercise	446

22 SIGNALS: ADVANCED FEATURES 447

22.1	Core Dump Files	448
22.2	Special Cases for Delivery, Disposition, and Handling	450
22.3	Interruptible and Uninterruptible Process Sleep States.....	451
22.4	Hardware-Generated Signals.....	452
22.5	Synchronous and Asynchronous Signal Generation	452

22.6	Timing and Order of Signal Delivery	453
22.7	Implementation and Portability of <i>signal()</i>	454
22.8	Realtime Signals	456
22.8.1	Sending Realtime Signals	458
22.8.2	Handling Realtime Signals	460
22.9	Waiting for a Signal Using a Mask: <i>sigsuspend()</i>	464
22.10	Synchronously Waiting for a Signal	468
22.11	Fetching Signals via a File Descriptor	471
22.12	Interprocess Communication with Signals	474
22.13	Earlier Signal APIs (System V and BSD)	475
22.14	Summary	477
22.15	Exercises	478

23 TIMERS AND SLEEPING 479

23.1	Interval Timers	479
23.2	Scheduling and Accuracy of Timers	485
23.3	Setting Timeouts on Blocking Operations	486
23.4	Suspending Execution for a Fixed Interval (Sleeping)	487
23.4.1	Low-Resolution Sleeping: <i>sleep()</i>	487
23.4.2	High-Resolution Sleeping: <i>nanosleep()</i>	488
23.5	POSIX Clocks	491
23.5.1	Retrieving the Value of a Clock: <i>clock_gettime()</i>	491
23.5.2	Setting the Value of a Clock: <i>clock_settime()</i>	492
23.5.3	Obtaining the Clock ID of a Specific Process or Thread	493
23.5.4	Improved High-Resolution Sleeping: <i>clock_nanosleep()</i>	493
23.6	POSIX Interval Timers	495
23.6.1	Creating a Timer: <i>timer_create()</i>	495
23.6.2	Arming and Disarming a Timer: <i>timer_settime()</i>	498
23.6.3	Retrieving the Current Value of a Timer: <i>timer_gettime()</i>	499
23.6.4	Deleting a Timer: <i>timer_delete()</i>	499
23.6.5	Notification via a Signal	499
23.6.6	Timer Overruns	503
23.6.7	Notification via a Thread	504
23.7	Timers That Notify via File Descriptors: the <i>timerfd</i> API	507
23.8	Summary	511
23.9	Exercises	512

24 PROCESS CREATION 513

24.1	Overview of <i>fork()</i> , <i>exit()</i> , <i>wait()</i> , and <i>execve()</i>	513
24.2	Creating a New Process: <i>fork()</i>	515
24.2.1	File Sharing Between Parent and Child	517
24.2.2	Memory Semantics of <i>fork()</i>	520
24.3	The <i>vfork()</i> System Call	522
24.4	Race Conditions After <i>fork()</i>	525
24.5	Avoiding Race Conditions by Synchronizing with Signals	527
24.6	Summary	529
24.7	Exercises	530

25 PROCESS TERMINATION 531

25.1	Terminating a Process: <i>_exit()</i> and <i>exit()</i>	531
25.2	Details of Process Termination	533
25.3	Exit Handlers	533
25.4	Interactions Between <i>fork()</i> , <i>stdio</i> Buffers, and <i>_exit()</i>	537

25.5	Summary.....	538
25.6	Exercise	539
26	MONITORING CHILD PROCESSES	541
26.1	Waiting on a Child Process	541
26.1.1	The <i>wait()</i> System Call.....	541
26.1.2	The <i>waitpid()</i> System Call	544
26.1.3	The Wait Status Value	545
26.1.4	Process Termination from a Signal Handler	549
26.1.5	The <i>waitid()</i> System Call.....	550
26.1.6	The <i>wait3()</i> and <i>wait4()</i> System Calls	552
26.2	Orphans and Zombies	553
26.3	The SIGCHLD Signal	555
26.3.1	Establishing a Handler for SIGCHLD	555
26.3.2	Delivery of SIGCHLD for Stopped Children	559
26.3.3	Ignoring Dead Child Processes	559
26.4	Summary.....	561
26.5	Exercises	562
27	PROGRAM EXECUTION	563
27.1	Executing a New Program: <i>execve()</i>	563
27.2	The <i>exec()</i> Library Functions.....	567
27.2.1	The PATH Environment Variable	568
27.2.2	Specifying Program Arguments as a List.....	570
27.2.3	Passing the Caller's Environment to the New Program	570
27.2.4	Executing a File Referred to by a Descriptor: <i>fexecve()</i>	571
27.3	Interpreter Scripts	572
27.4	File Descriptors and <i>exec()</i>	575
27.5	Signals and <i>exec()</i>	578
27.6	Executing a Shell Command: <i>system()</i>	579
27.7	Implementing <i>system()</i>	582
27.8	Summary.....	588
27.9	Exercises	589
28	PROCESS CREATION AND PROGRAM EXECUTION IN MORE DETAIL	591
28.1	Process Accounting.....	591
28.2	The <i>clone()</i> System Call	598
28.2.1	The <i>clone()</i> <i>flags</i> Argument	603
28.2.2	Extensions to <i>waitpid()</i> for Cloned Children	609
28.3	Speed of Process Creation.....	610
28.4	Effect of <i>exec()</i> and <i>fork()</i> on Process Attributes.....	612
28.5	Summary.....	616
28.6	Exercise	616
29	THREADS: INTRODUCTION	617
29.1	Overview	617
29.2	Background Details of the Pthreads API.....	620
29.3	Thread Creation.....	622
29.4	Thread Termination.....	623
29.5	Thread IDs.....	624
29.6	Joining with a Terminated Thread.....	625
29.7	Detaching a Thread.....	627

29.8	Thread Attributes	628
29.9	Threads Versus Processes	629
29.10	Summary	629
29.11	Exercises	630

30 THREADS: THREAD SYNCHRONIZATION 631

30.1	Protecting Accesses to Shared Variables: Mutexes	631
30.1.1	Statically Allocated Mutexes	635
30.1.2	Locking and Unlocking a Mutex	635
30.1.3	Performance of Mutexes	638
30.1.4	Mutex Deadlocks	639
30.1.5	Dynamically Initializing a Mutex	639
30.1.6	Mutex Attributes	640
30.1.7	Mutex Types	640
30.2	Signaling Changes of State: Condition Variables	642
30.2.1	Statically Allocated Condition Variables	643
30.2.2	Signaling and Waiting on Condition Variables	643
30.2.3	Testing a Condition Variable's Predicate	647
30.2.4	Example Program: Joining Any Terminated Thread	648
30.2.5	Dynamically Allocated Condition Variables	651
30.3	Summary	652
30.4	Exercises	652

31 THREADS: THREAD SAFETY AND PER-THREAD STORAGE 655

31.1	Thread Safety (and Reentrancy Revisited)	655
31.2	One-Time Initialization	658
31.3	Thread-Specific Data	659
31.3.1	Thread-Specific Data from the Library Function's Perspective	660
31.3.2	Overview of the Thread-Specific Data API	660
31.3.3	Details of the Thread-Specific Data API	661
31.3.4	Employing the Thread-Specific Data API	663
31.3.5	Thread-Specific Data Implementation Limits	668
31.4	Thread-Local Storage	668
31.5	Summary	669
31.6	Exercises	670

32 THREADS: THREAD CANCELLATION 671

32.1	Canceling a Thread	671
32.2	Cancellation State and Type	672
32.3	Cancellation Points	673
32.4	Testing for Thread Cancellation	675
32.5	Cleanup Handlers	676
32.6	Asynchronous Cancelability	680
32.7	Summary	680

33 THREADS: FURTHER DETAILS 681

33.1	Thread Stacks	681
33.2	Threads and Signals	682
33.2.1	How the UNIX Signal Model Maps to Threads	682
33.2.2	Manipulating the Thread Signal Mask	684
33.2.3	Sending a Signal to a Thread	684
33.2.4	Dealing with Asynchronous Signals Sanely	685

33.3	Threads and Process Control.....	686
33.4	Thread Implementation Models	687
33.5	Linux Implementations of POSIX Threads	689
33.5.1	LinuxThreads	689
33.5.2	NPTL	692
33.5.3	Which Threading Implementation?.....	694
33.6	Advanced Features of the Pthreads API	696
33.7	Summary.....	696
33.8	Exercises	697
34	PROCESS GROUPS, SESSIONS, AND JOB CONTROL	699
34.1	Overview	699
34.2	Process Groups	701
34.3	Sessions	704
34.4	Controlling Terminals and Controlling Processes.....	706
34.5	Foreground and Background Process Groups	708
34.6	The SIGHUP Signal.....	709
34.6.1	Handling of SIGHUP by the Shell	710
34.6.2	SIGHUP and Termination of the Controlling Process.....	712
34.7	Job Control.....	714
34.7.1	Using Job Control Within the Shell	714
34.7.2	Implementing Job Control.....	717
34.7.3	Handling Job-Control Signals	722
34.7.4	Orphaned Process Groups (and SIGHUP Revisited)	725
34.8	Summary.....	730
34.9	Exercises	731
35	PROCESS PRIORITIES AND SCHEDULING	733
35.1	Process Priorities (Nice Values)	733
35.2	Overview of Realtime Process Scheduling.....	737
35.2.1	The SCHED_RR Policy	739
35.2.2	The SCHED_FIFO Policy	740
35.2.3	The SCHED_BATCH and SCHED_IDLE Policies.....	740
35.3	Realtime Process Scheduling API	740
35.3.1	Realtime Priority Ranges	740
35.3.2	Modifying and Retrieving Policies and Priorities.....	741
35.3.3	Relinquishing the CPU	747
35.3.4	The SCHED_RR Time Slice	747
35.4	CPU Affinity.....	748
35.5	Summary.....	751
35.6	Exercises	751
36	PROCESS RESOURCES	753
36.1	Process Resource Usage	753
36.2	Process Resource Limits	755
36.3	Details of Specific Resource Limits	760
36.4	Summary.....	765
36.5	Exercises	765
37	DAEMONS	767
37.1	Overview	767
37.2	Creating a Daemon	768

37.3	Guidelines for Writing Daemons	771
37.4	Using SIGHUP to Reinitialize a Daemon	772
37.5	Logging Messages and Errors Using <i>syslog</i>	775
37.5.1	Overview	775
37.5.2	The <i>syslog</i> API	777
37.5.3	The <i>/etc/syslog.conf</i> File	781
37.6	Summary	782
37.7	Exercise	782

38 WRITING SECURE PRIVILEGED PROGRAMS 783

38.1	Is a Set-User-ID or Set-Group-ID Program Required?	784
38.2	Operate with Least Privilege	784
38.3	Be Careful When Executing a Program	787
38.4	Avoid Exposing Sensitive Information	788
38.5	Confine the Process	789
38.6	Beware of Signals and Race Conditions	790
38.7	Pitfalls When Performing File Operations and File I/O	790
38.8	Don't Trust Inputs or the Environment	791
38.9	Beware of Buffer Overruns	792
38.10	Beware of Denial-of-Service Attacks	793
38.11	Check Return Statuses and Fail Safely	794
38.12	Summary	795
38.13	Exercises	796

39 CAPABILITIES 797

39.1	Rationale for Capabilities	797
39.2	The Linux Capabilities	798
39.3	Process and File Capabilities	798
39.3.1	Process Capabilities	798
39.3.2	File Capabilities	799
39.3.3	Purpose of the Process Permitted and Effective Capability Sets	802
39.3.4	Purpose of the File Permitted and Effective Capability Sets	802
39.3.5	Purpose of the Process and File Inheritable Sets	802
39.3.6	Assigning and Viewing File Capabilities from the Shell	803
39.4	The Modern Capabilities Implementation	804
39.5	Transformation of Process Capabilities During <i>exec()</i>	805
39.5.1	Capability Bounding Set	805
39.5.2	Preserving <i>root</i> Semantics	806
39.6	Effect on Process Capabilities of Changing User IDs	806
39.7	Changing Process Capabilities Programmatically	807
39.8	Creating Capabilities-Only Environments	811
39.9	Discovering the Capabilities Required by a Program	813
39.10	Older Kernels and Systems Without File Capabilities	814
39.11	Summary	816
39.12	Exercise	816

40 LOGIN ACCOUNTING 817

40.1	Overview of the <i>utmp</i> and <i>wtmp</i> Files	817
40.2	The <i>utmpx</i> API	818
40.3	The <i>utmpx</i> Structure	818
40.4	Retrieving Information from the <i>utmp</i> and <i>wtmp</i> Files	821
40.5	Retrieving the Login Name: <i>getlogin()</i>	825
40.6	Updating the <i>utmp</i> and <i>wtmp</i> Files for a Login Session	825

40.7	The lastlog File	830
40.8	Summary.....	832
40.9	Exercises	832

41 FUNDAMENTALS OF SHARED LIBRARIES 833

41.1	Object Libraries	833
41.2	Static Libraries	834
41.3	Overview of Shared Libraries.....	836
41.4	Creating and Using Shared Libraries—A First Pass	837
41.4.1	Creating a Shared Library.....	837
41.4.2	Position-Independent Code.....	838
41.4.3	Using a Shared Library.....	839
41.4.4	The Shared Library Soname	840
41.5	Useful Tools for Working with Shared Libraries	843
41.6	Shared Library Versions and Naming Conventions	844
41.7	Installing Shared Libraries	847
41.8	Compatible Versus Incompatible Libraries.....	850
41.9	Upgrading Shared Libraries.....	850
41.10	Specifying Library Search Directories in an Object File	851
41.11	Finding Shared Libraries at Run Time	854
41.12	Run-Time Symbol Resolution	854
41.13	Using a Static Library Instead of a Shared Library	856
41.14	Summary.....	856
41.15	Exercise	857

42 ADVANCED FEATURES OF SHARED LIBRARIES 859

42.1	Dynamically Loaded Libraries	859
42.1.1	Opening a Shared Library: <i>dlopen()</i>	860
42.1.2	Diagnosing Errors: <i>dlderror()</i>	862
42.1.3	Obtaining the Address of a Symbol: <i>dlsym()</i>	862
42.1.4	Closing a Shared Library: <i>dlclose()</i>	866
42.1.5	Obtaining Information About Loaded Symbols: <i>dladdr()</i>	866
42.1.6	Accessing Symbols in the Main Program.....	867
42.2	Controlling Symbol Visibility	867
42.3	Linker Version Scripts.....	868
42.3.1	Controlling Symbol Visibility with Version Scripts	868
42.3.2	Symbol Versioning	870
42.4	Initialization and Finalization Functions.....	872
42.5	Preloading Shared Libraries.....	873
42.6	Monitoring the Dynamic Linker: LD_DEBUG.....	874
42.7	Summary.....	875
42.8	Exercises	876

43 INTERPROCESS COMMUNICATION OVERVIEW 877

43.1	A Taxonomy of IPC Facilities	877
43.2	Communication Facilities.....	879
43.3	Synchronization Facilities	880
43.4	Comparing IPC Facilities	882
43.5	Summary.....	887
43.6	Exercises	887

44	PIPES AND FIFOs	889
44.1	Overview	889
44.2	Creating and Using Pipes	892
44.3	Pipes as a Method of Process Synchronization	897
44.4	Using Pipes to Connect Filters	899
44.5	Talking to a Shell Command via a Pipe: <i>popen()</i>	902
44.6	Pipes and <i>stdio</i> Buffering	906
44.7	FIFOs	906
44.8	A Client-Server Application Using FIFOs	909
44.9	Nonblocking I/O	915
44.10	Semantics of <i>read()</i> and <i>write()</i> on Pipes and FIFOs	917
44.11	Summary	918
44.12	Exercises	919
45	INTRODUCTION TO SYSTEM V IPC	921
45.1	API Overview	922
45.2	IPC Keys	925
45.3	Associated Data Structure and Object Permissions	927
45.4	IPC Identifiers and Client-Server Applications	929
45.5	Algorithm Employed by System V IPC <i>get</i> Calls	931
45.6	The <i>ipcs</i> and <i>ipcrm</i> Commands	934
45.7	Obtaining a List of All IPC Objects	935
45.8	IPC Limits	935
45.9	Summary	936
45.10	Exercises	936
46	SYSTEM V MESSAGE QUEUES	937
46.1	Creating or Opening a Message Queue	938
46.2	Exchanging Messages	940
46.2.1	Sending Messages	940
46.2.2	Receiving Messages	943
46.3	Message Queue Control Operations	947
46.4	Message Queue Associated Data Structure	948
46.5	Message Queue Limits	950
46.6	Displaying All Message Queues on the System	951
46.7	Client-Server Programming with Message Queues	953
46.8	A File-Server Application Using Message Queues	955
46.9	Disadvantages of System V Message Queues	961
46.10	Summary	962
46.11	Exercises	963
47	SYSTEM V SEMAPHORES	965
47.1	Overview	966
47.2	Creating or Opening a Semaphore Set	969
47.3	Semaphore Control Operations	969
47.4	Semaphore Associated Data Structure	972
47.5	Semaphore Initialization	975
47.6	Semaphore Operations	978
47.7	Handling of Multiple Blocked Semaphore Operations	986
47.8	Semaphore Undo Values	986
47.9	Implementing a Binary Semaphores Protocol	988

47.10	Semaphore Limits	991
47.11	Disadvantages of System V Semaphores	993
47.12	Summary.....	993
47.13	Exercises	994

48 SYSTEM V SHARED MEMORY 997

48.1	Overview	998
48.2	Creating or Opening a Shared Memory Segment	998
48.3	Using Shared Memory	999
48.4	Example: Transferring Data via Shared Memory	1001
48.5	Location of Shared Memory in Virtual Memory.....	1006
48.6	Storing Pointers in Shared Memory.....	1010
48.7	Shared Memory Control Operations	1011
48.8	Shared Memory Associated Data Structure.....	1012
48.9	Shared Memory Limits.....	1014
48.10	Summary.....	1015
48.11	Exercises	1016

49 MEMORY MAPPINGS 1017

49.1	Overview	1017
49.2	Creating a Mapping: <i>mmap()</i>	1020
49.3	Unmapping a Mapped Region: <i>munmap()</i>	1023
49.4	File Mappings.....	1024
49.4.1	Private File Mappings.....	1024
49.4.2	Shared File Mappings	1025
49.4.3	Boundary Cases	1029
49.4.4	Memory Protection and File Access Mode Interactions.....	1030
49.5	Synchronizing a Mapped Region: <i>msync()</i>	1031
49.6	Additional <i>mmap()</i> Flags.....	1033
49.7	Anonymous Mappings	1034
49.8	Remapping a Mapped Region: <i>mremap()</i>	1037
49.9	MAP_NORESERVE and Swap Space Overcommitting	1038
49.10	The MAP_FIXED Flag	1040
49.11	Nonlinear Mappings: <i>remap_file_pages()</i>	1041
49.12	Summary.....	1043
49.13	Exercises	1044

50 VIRTUAL MEMORY OPERATIONS 1045

50.1	Changing Memory Protection: <i>mprotect()</i>	1045
50.2	Memory Locking: <i>mlock()</i> and <i>mlockall()</i>	1047
50.3	Determining Memory Residence: <i>mincore()</i>	1051
50.4	Advising Future Memory Usage Patterns: <i>madvise()</i>	1054
50.5	Summary.....	1056
50.6	Exercises	1056

51 INTRODUCTION TO POSIX IPC 1057

51.1	API Overview	1058
51.2	Comparison of System V IPC and POSIX IPC	1061
51.3	Summary.....	1062

52	POSIX MESSAGE QUEUES	1063
52.1	Overview	1064
52.2	Opening, Closing, and Unlinking a Message Queue	1064
52.3	Relationship Between Descriptors and Message Queues	1067
52.4	Message Queue Attributes	1068
52.5	Exchanging Messages	1073
52.5.1	Sending Messages	1073
52.5.2	Receiving Messages	1074
52.5.3	Sending and Receiving Messages with a Timeout	1077
52.6	Message Notification	1077
52.6.1	Receiving Notification via a Signal	1079
52.6.2	Receiving Notification via a Thread	1082
52.7	Linux-Specific Features	1083
52.8	Message Queue Limits	1085
52.9	Comparison of POSIX and System V Message Queues	1086
52.10	Summary	1087
52.11	Exercises	1087
53	POSIX SEMAPHORES	1089
53.1	Overview	1089
53.2	Named Semaphores	1090
53.2.1	Opening a Named Semaphore	1090
53.2.2	Closing a Semaphore	1093
53.2.3	Removing a Named Semaphore	1093
53.3	Semaphore Operations	1094
53.3.1	Waiting on a Semaphore	1094
53.3.2	Posting a Semaphore	1096
53.3.3	Retrieving the Current Value of a Semaphore	1097
53.4	Unnamed Semaphores	1099
53.4.1	Initializing an Unnamed Semaphore	1100
53.4.2	Destroying an Unnamed Semaphore	1102
53.5	Comparisons with Other Synchronization Techniques	1103
53.6	Semaphore Limits	1104
53.7	Summary	1105
53.8	Exercises	1105
54	POSIX SHARED MEMORY	1107
54.1	Overview	1108
54.2	Creating Shared Memory Objects	1109
54.3	Using Shared Memory Objects	1112
54.4	Removing Shared Memory Objects	1114
54.5	Comparisons Between Shared Memory APIs	1115
54.6	Summary	1116
54.7	Exercise	1116
55	FILE LOCKING	1117
55.1	Overview	1117
55.2	File Locking with <i>flock()</i>	1119
55.2.1	Semantics of Lock Inheritance and Release	1122
55.2.2	Limitations of <i>flock()</i>	1123

55.3	Record Locking with <i>fcntl()</i>	1124
55.3.1	Deadlock	1128
55.3.2	Example: An Interactive Locking Program.....	1129
55.3.3	Example: A Library of Locking Functions	1133
55.3.4	Lock Limits and Performance.....	1135
55.3.5	Semantics of Lock Inheritance and Release	1136
55.3.6	Lock Starvation and Priority of Queued Lock Requests.....	1137
55.4	Mandatory Locking.....	1137
55.5	The <i>/proc/locks</i> File	1140
55.6	Running Just One Instance of a Program.....	1142
55.7	Older Locking Techniques	1144
55.8	Summary.....	1146
55.9	Exercises	1147

56 SOCKETS: INTRODUCTION 1149

56.1	Overview	1150
56.2	Creating a Socket: <i>socket()</i>	1153
56.3	Binding a Socket to an Address: <i>bind()</i>	1153
56.4	Generic Socket Address Structures: <i>struct sockaddr</i>	1154
56.5	Stream Sockets.....	1155
56.5.1	Listening for Incoming Connections: <i>listen()</i>	1156
56.5.2	Accepting a Connection: <i>accept()</i>	1157
56.5.3	Connecting to a Peer Socket: <i>connect()</i>	1158
56.5.4	I/O on Stream Sockets	1159
56.5.5	Connection Termination: <i>close()</i>	1159
56.6	Datagram Sockets	1159
56.6.1	Exchanging Datagrams: <i>recvfrom()</i> and <i>sendto()</i>	1160
56.6.2	Using <i>connect()</i> with Datagram Sockets	1162
56.7	Summary.....	1162

57 SOCKETS: UNIX DOMAIN 1165

57.1	UNIX Domain Socket Addresses: <i>struct sockaddr_un</i>	1165
57.2	Stream Sockets in the UNIX Domain	1167
57.3	Datagram Sockets in the UNIX Domain	1171
57.4	UNIX Domain Socket Permissions	1174
57.5	Creating a Connected Socket Pair: <i>socketpair()</i>	1174
57.6	The Linux Abstract Socket Namespace	1175
57.7	Summary.....	1176
57.8	Exercises	1177

58 SOCKETS: FUNDAMENTALS OF TCP/IP NETWORKS 1179

58.1	Internets	1179
58.2	Networking Protocols and Layers	1180
58.3	The Data-Link Layer.....	1182
58.4	The Network Layer: IP	1184
58.5	IP Addresses	1186
58.6	The Transport Layer	1188
58.6.1	Port Numbers	1188
58.6.2	User Datagram Protocol (UDP)	1189
58.6.3	Transmission Control Protocol (TCP).....	1190
58.7	Requests for Comments (RFCs)	1193
58.8	Summary.....	1195

59	SOCKETS: INTERNET DOMAINS	1197
59.1	Internet Domain Sockets	1197
59.2	Network Byte Order	1198
59.3	Data Representation	1199
59.4	Internet Socket Addresses	1202
59.5	Overview of Host and Service Conversion Functions	1204
59.6	The <i>inet_pton()</i> and <i>inet_ntop()</i> Functions	1206
59.7	Client-Server Example (Datagram Sockets)	1207
59.8	Domain Name System (DNS)	1209
59.9	The <i>/etc/services</i> File	1212
59.10	Protocol-Independent Host and Service Conversion	1213
59.10.1	The <i>getaddrinfo()</i> Function	1213
59.10.2	Freeing <i>addrinfo</i> Lists: <i>freeaddrinfo()</i>	1217
59.10.3	Diagnosing Errors: <i>gai_strerror()</i>	1217
59.10.4	The <i>getnameinfo()</i> Function	1218
59.11	Client-Server Example (Stream Sockets)	1219
59.12	An Internet Domain Sockets Library	1225
59.13	Obsolete APIs for Host and Service Conversions	1230
59.13.1	The <i>inet_aton()</i> and <i>inet_ntoa()</i> Functions	1230
59.13.2	The <i>gethostbyname()</i> and <i>gethostbyaddr()</i> Functions	1231
59.13.3	The <i>getservbyname()</i> and <i>getservbyport()</i> Functions	1234
59.14	UNIX Versus Internet Domain Sockets	1235
59.15	Further Information	1235
59.16	Summary	1236
59.17	Exercises	1236
60	SOCKETS: SERVER DESIGN	1239
60.1	Iterative and Concurrent Servers	1239
60.2	An Iterative UDP <i>echo</i> Server	1240
60.3	A Concurrent TCP <i>echo</i> Server	1243
60.4	Other Concurrent Server Designs	1245
60.5	The <i>inetd</i> (Internet Superserver) Daemon	1247
60.6	Summary	1252
60.7	Exercises	1252
61	SOCKETS: ADVANCED TOPICS	1253
61.1	Partial Reads and Writes on Stream Sockets	1254
61.2	The <i>shutdown()</i> System Call	1256
61.3	Socket-Specific I/O System Calls: <i>recv()</i> and <i>send()</i>	1259
61.4	The <i>sendfile()</i> System Call	1260
61.5	Retrieving Socket Addresses	1263
61.6	A Closer Look at TCP	1266
61.6.1	Format of a TCP Segment	1266
61.6.2	TCP Sequence Numbers and Acknowledgements	1268
61.6.3	TCP State Machine and State Transition Diagram	1269
61.6.4	TCP Connection Establishment	1270
61.6.5	TCP Connection Termination	1272
61.6.6	Calling <i>shutdown()</i> on a TCP Socket	1273
61.6.7	The <i>TIME_WAIT</i> State	1274
61.7	Monitoring Sockets: <i>netstat</i>	1275
61.8	Using <i>tcpdump</i> to Monitor TCP Traffic	1276
61.9	Socket Options	1278
61.10	The <i>SO_REUSEADDR</i> Socket Option	1279
61.11	Inheritance of Flags and Options Across <i>accept()</i>	1281

61.12	TCP Versus UDP	1282
61.13	Advanced Features	1283
61.13.1	Out-of-Band Data	1283
61.13.2	The <i>sendmsg()</i> and <i>recvmsg()</i> System Calls	1284
61.13.3	Passing File Descriptors	1284
61.13.4	Receiving Sender Credentials	1284
61.13.5	Sequenced-Packet Sockets	1285
61.13.6	SCTP and DCCP Transport-Layer Protocols	1285
61.14	Summary	1286
61.15	Exercises	1287

62 TERMINALS 1289

62.1	Overview	1290
62.2	Retrieving and Modifying Terminal Attributes	1291
62.3	The <i>stty</i> Command	1294
62.4	Terminal Special Characters	1296
62.5	Terminal Flags	1301
62.6	Terminal I/O Modes	1307
62.6.1	Canonical Mode	1307
62.6.2	Noncanonical Mode	1307
62.6.3	Cooked, Cbreak, and Raw Modes	1309
62.7	Terminal Line Speed (Bit Rate)	1316
62.8	Terminal Line Control	1317
62.9	Terminal Window Size	1319
62.10	Terminal Identification	1321
62.11	Summary	1322
62.12	Exercises	1323

63 ALTERNATIVE I/O MODELS 1325

63.1	Overview	1325
63.1.1	Level-Triggered and Edge-Triggered Notification	1329
63.1.2	Employing Nonblocking I/O with Alternative I/O Models	1330
63.2	I/O Multiplexing	1330
63.2.1	The <i>select()</i> System Call	1331
63.2.2	The <i>poll()</i> System Call	1337
63.2.3	When Is a File Descriptor Ready?	1341
63.2.4	Comparison of <i>select()</i> and <i>poll()</i>	1344
63.2.5	Problems with <i>select()</i> and <i>poll()</i>	1346
63.3	Signal-Driven I/O	1346
63.3.1	When Is “I/O Possible” Signaled?	1351
63.3.2	Refining the Use of Signal-Driven I/O	1352
63.4	The <i>epoll</i> API	1355
63.4.1	Creating an <i>epoll</i> Instance: <i>epoll_create()</i>	1356
63.4.2	Modifying the <i>epoll</i> Interest List: <i>epoll_ctl()</i>	1356
63.4.3	Waiting for Events: <i>epoll_wait()</i>	1358
63.4.4	A Closer Look at <i>epoll</i> Semantics	1363
63.4.5	Performance of <i>epoll</i> Versus I/O Multiplexing	1365
63.4.6	Edge-Triggered Notification	1366
63.5	Waiting on Signals and File Descriptors	1368
63.5.1	The <i>pselect()</i> System Call	1369
63.5.2	The Self-Pipe Trick	1370
63.6	Summary	1373
63.7	Exercises	1374

64	PSEUDOTERMINALS	1375
64.1	Overview	1375
64.2	UNIX 98 Pseudoterminals.....	1380
64.2.1	Opening an Unused Master: <i>posix_openpt()</i>	1380
64.2.2	Changing Slave Ownership and Permissions: <i>grantpt()</i>	1381
64.2.3	Unlocking the Slave: <i>unlockpt()</i>	1382
64.2.4	Obtaining the Name of the Slave: <i>ptsname()</i>	1382
64.3	Opening a Master: <i>ptyMasterOpen()</i>	1383
64.4	Connecting Processes with a Pseudoterminal: <i>ptyFork()</i>	1385
64.5	Pseudoterminal I/O	1388
64.6	Implementing <i>script(1)</i>	1390
64.7	Terminal Attributes and Window Size	1394
64.8	BSD Pseudoterminals	1395
64.9	Summary.....	1397
64.10	Exercises	1398
A	TRACING SYSTEM CALLS	1401
B	PARSING COMMAND-LINE OPTIONS	1405
C	CASTING THE NULL POINTER	1413
D	KERNEL CONFIGURATION	1417
E	FURTHER SOURCES OF INFORMATION	1419
F	SOLUTIONS TO SELECTED EXERCISES	1425
	BIBLIOGRAPHY	1437
	INDEX	1447

PREFACE

Subject

In this book, I describe the Linux programming interface—the system calls, library functions, and other low-level interfaces provided by Linux, a free implementation of the UNIX operating system. These interfaces are used, directly or indirectly, by every program that runs on Linux. They allow applications to perform tasks such as file I/O, creating and deleting files and directories, creating new processes, executing programs, setting timers, communicating between processes and threads on the same computer, and communicating between processes residing on different computers connected via a network. This set of low-level interfaces is sometimes also known as the *system programming* interface.

Although I focus on Linux, I give careful attention to standards and portability issues, and clearly distinguish the discussion of Linux-specific details from the discussion of features that are common to most UNIX implementations and standardized by POSIX and the Single UNIX Specification. Thus, this book also provides a comprehensive description of the UNIX/POSIX programming interface and can be used by programmers writing applications targeted at other UNIX systems or intended to be portable across multiple systems.

Intended audience

This book is aimed primarily at the following audience:

- programmers and software designers building applications for Linux, other UNIX systems, or other POSIX-conformant systems;
- programmers porting applications between Linux and other UNIX implementations or between Linux and other operating systems;
- instructors and advanced students teaching or learning Linux or UNIX system programming; and
- system managers and “power users” wishing to gain a greater understanding of the Linux/UNIX programming interface and of how various pieces of system software are implemented.

I assume you have some prior programming experience, but no previous system programming experience is required. I also assume you have a reading knowledge of the C programming language, and know how to use the shell and common Linux or UNIX commands. If you are new to Linux or UNIX, you will find it helpful to read the programmer-oriented review of fundamental concepts of Linux and UNIX systems in Chapter 2.

The standard tutorial reference for C is [Kernighan & Ritchie, 1988]. [Harbison & Steele, 2002] goes into even more detail on C, and includes coverage of changes introduced with the C99 standard. [van der Linden, 1994] is an alternative look at C that is both highly amusing and instructive. [Peek et al., 2001] provides a good, brief introduction to using a UNIX system.

Throughout this book, indented small-font paragraphs like these are used for asides containing rationale, implementation details, background information, historical notes, and other topics that are ancillary to the main text.

Linux and UNIX

This book could have been purely about standard UNIX (that is, POSIX) system programming because most features found on other UNIX implementations are also present on Linux and vice versa. However, while writing portable applications is a worthy goal, it is also important to describe Linux extensions to the standard UNIX programming interface. One reason for this is the popularity of Linux. Another is that the use of nonstandard extensions is sometimes essential, either for performance reasons or to access functionality that is unavailable in the standard UNIX programming interface. (All UNIX implementations provide nonstandard extensions for these reasons.)

Therefore, while I’ve designed this book to be useful to programmers working with all UNIX implementations, I also provide full coverage of programming features that are specific to Linux. These features include:

- *epoll*, a mechanism for obtaining notification of file I/O events;
- *inotify*, a mechanism for monitoring changes in files and directories;
- capabilities, a mechanism for granting a process a subset of the powers of the superuser;

- extended attributes;
- i-node flags;
- the *clone()* system call;
- the */proc* file system; and
- Linux-specific details of the implementation of file I/O, signals, timers, threads, shared libraries, interprocess communication, and sockets.

Usage and organization

You can use this book in at least two ways:

- As a tutorial introduction to the Linux/UNIX programming interface. You can read the book linearly. Later chapters build on material presented in earlier chapters, with forward references minimized as far as possible.
- As a comprehensive reference to the Linux/UNIX programming interface. An extensive index and frequent cross-references allow topics to be read in random order.

I've grouped the chapters of this book into the following parts:

1. *Background and concepts*: history of UNIX, C, and Linux and overview of UNIX standards (Chapter 1); a programmer-oriented introduction to Linux and UNIX concepts (Chapter 2); and fundamental concepts for system programming on Linux and UNIX (Chapter 3).
2. *Fundamental features of the system programming interface*: file I/O (Chapter 4 and Chapter 5); processes (Chapter 6); memory allocation (Chapter 7); users and groups (Chapter 8); process credentials (Chapter 9); time (Chapter 10); system limits and options (Chapter 11); and retrieving system and process information (Chapter 12).
3. *More advanced features of the system programming interface*: file I/O buffering (Chapter 13); file systems (Chapter 14); file attributes (Chapter 15); extended attributes (Chapter 16); access control lists (Chapter 17); directories and links (Chapter 18); monitoring file events (Chapter 19); signals (Chapter 20 to Chapter 22); and timers (Chapter 23).
4. *Processes, programs, and threads*: process creation, process termination, monitoring child processes, and executing programs (Chapter 24 to Chapter 28); and POSIX threads (Chapter 29 to Chapter 33).
5. *Advanced process and program topics*: process groups, sessions, and job control (Chapter 34); process priorities and scheduling (Chapter 35); process resources (Chapter 36); daemons (Chapter 37); writing secure privileged programs (Chapter 38); capabilities (Chapter 39); login accounting (Chapter 40); and shared libraries (Chapter 41 and Chapter 42).
6. *Interprocess communication (IPC)*: IPC overview (Chapter 43); pipes and FIFOs (Chapter 44); System V IPC—message queues, semaphores, and shared memory (Chapter 45 to Chapter 48); memory mappings (Chapter 49); virtual memory operations (Chapter 50); POSIX IPC—message queues, semaphores, and shared memory (Chapter 51 to Chapter 54); and file locking (Chapter 55).

7. *Sockets and network programming*: IPC and network programming with sockets (Chapter 56 to Chapter 61).
8. *Advanced I/O topics*: terminals (Chapter 62); alternative I/O models (Chapter 63); and pseudoterminals (Chapter 64).

Example programs

I illustrate the use of most of the interfaces described in this book with short, complete programs, many of which are designed to allow you to easily experiment from the command line to see just how various system calls and library functions work. Consequently, this book contains a lot of example code—around 15,000 lines of C source code and shell session logs.

Although reading and experimenting with the example programs is a useful starting point, the most effective way to consolidate the concepts discussed in this book is to write code, either modifying the example programs to try out your own ideas or writing new programs.

All of the source code in this book is available for download from the book's web site. The source code distribution also includes many additional programs that don't appear in the book. The purpose and details of these programs are described in comments in the source code. *Makefiles* are provided for building the programs, and an accompanying README file gives further details about the programs.

The source code is freely redistributable and modifiable under the terms of the GNU Affero General Public License (Affero GPL) version 3, a copy of which is provided in the source code distribution.

Exercises

Most chapters conclude with a set of exercises, some of which are suggestions for various experiments using the provided example programs. Other exercises are questions relating to concepts discussed in the chapter, and still others are suggestions for programs you might write in order to consolidate your understanding of the material. You'll find solutions to selected exercises in Appendix F.

Standards and portability

Throughout this book, I've taken special care to consider portability issues. You'll find frequent references to relevant standards, especially the combined POSIX.1-2001 and Single UNIX Specification version 3 (SUSv3) standard. You'll also find details about changes in the recent revision of that standard, the combined POSIX.1-2008 and SUSv4 standard. (Because SUSv3 was a much larger revision, and it is the UNIX standard that is in most widespread effect at the time of writing, discussions of standards in the book are generally framed in terms of SUSv3, with notes on the differences in SUSv4. However, you can assume that, except where noted, statements about specifications in SUSv3 also hold true in SUSv4.)

For features that are not standardized, I indicate the range of differences on other UNIX implementations. I also highlight those major features of Linux that are implementation-specific, as well as minor differences between the implementation of system calls and library functions on Linux and other UNIX implementations. Where a feature is not indicated as being Linux-specific, you can normally assume that it is a standard feature that appears on most or all UNIX implementations.

I've tested most of the example programs presented in this book (other than those that exploit features that are noted as being Linux-specific) on some or all of Solaris, FreeBSD, Mac OS X, Tru64 UNIX, and HP-UX. To improve portability to some of these systems, the web site for this book provides alternative versions of certain example programs with extra code that doesn't appear in the book.

Linux kernel and C library versions

The primary focus of this book is on Linux 2.6.x, the kernel version in widest use at the time of writing. Details for Linux 2.4 are also covered, and I've indicated where features differ between Linux 2.4 and 2.6. Where new features appear in the Linux 2.6.x series, the exact kernel version number of their appearance (e.g., 2.6.34) is noted.

With respect to the C library, the main focus is on the GNU C library (*glibc*) version 2. Where relevant, differences across *glibc* 2.x versions are noted.

As this book was heading to press, Linux kernel version 2.6.35 had just been released, and *glibc* version 2.12 had been recently released. This book is current with respect to both of these software versions. Changes that occur in the Linux and *glibc* interfaces after publication of this book will be noted on the book's web site.

Using the programming interface from other languages

Although the example programs are written in C, you can use the interfaces described in this book from other programming languages—for example, compiled languages such as C++, Pascal, Modula, Ada, FORTRAN, D, and scripting languages such as Perl, Python, and Ruby. (Java requires a different approach; see, for example, [Rochkind, 2004].) Different techniques will be required to obtain the necessary constant definitions and function declarations (except in the case of C++), and some extra work may be needed to pass function arguments in the manner required by C linkage conventions. Notwithstanding these differences, the essential concepts are the same, and you'll find the information in this book is applicable even if you are working in another programming language.

About the author

I started using UNIX and C in 1987, when I spent several weeks sitting in front of an HP Bobcat workstation with a copy of the first edition of Marc Rochkind's *Advanced UNIX Programming* and what ultimately became a very dog-eared printed copy of the C shell manual page. My approach then was one that I still try to follow today, and that I recommend to anyone approaching a new software technology: take the time to read the documentation (if it exists) and write small (but increasingly large) test programs until you become confident of your understanding of the software. I've found that, in the long run, this kind of self-training more than pays for itself in terms of saved time. Many of the programming examples in this book are constructed in ways that encourage this learning approach.

I've primarily been a software engineer and designer. However, I'm also a passionate teacher, and have spent several years teaching in both academic and commercial environments. I've run many week-long courses teaching UNIX system programming, and that experience informs the writing of this book.

I've been using Linux for about half as long as I've been using UNIX, and, over that time, my interest has increasingly centered on the boundary between the kernel and user space: the Linux programming interface. This interest has drawn me into a number of interrelated activities. I intermittently provide input and bug reports for the POSIX/SUS standard; I carry out tests and design reviews of new user-space interfaces added to the Linux kernel (and have helped find and fix many code and design bugs in those interfaces); I've been a regular speaker at conferences on topics related to interfaces and their documentation; and I've been invited on a number of occasions to the annual Linux Kernel Developers Summit. The common thread tying all of these activities together is my most visible contribution in the Linux world: my work on the *man-pages* project (<http://www.kernel.org/doc/man-pages/>).

The *man-pages* project provides pages in sections 2, 3, 4, 5, and 7 of the Linux manual pages. These are the manual pages describing the programming interfaces provided by the Linux kernel and the GNU C library—the same topic area as this book. I've been involved with *man-pages* for more than a decade. Since 2004, I've been the project maintainer, a task that involves, in roughly equal measure, writing documentation, reading kernel and library source code, and writing programs to verify the details for documentation. (Documenting an interface is a great way to find bugs in that interface.) I've also been the biggest contributor to *man-pages*—of the approximately 900 pages in *man-pages*, I am the author of 140, and the coauthor of another 125. So, even before you picked up this book, it's quite likely you've read some of my published work. I hope that you've found that work useful, and that you'll find this book even more so.

Acknowledgements

Without the support of a good many people, this book would have been far less than it is. It is a great pleasure to thank them.

A large team of technical reviewers around the world read drafts, found errors, pointed out confusing explanations, suggested rewordings and diagrams, tested programs, proposed exercises, identified aspects of the behavior of Linux and other UNIX implementations that I was not aware of, and offered support and encouragement. Many reviewers generously supplied insights and comments that I was able to incorporate into the book, at times making me look more knowledgeable than I am. Any mistakes that remain are, of course, my own.

Thanks especially to the following reviewers (listed alphabetically by surname), who either commented on large sections of the manuscript, commented extensively on smaller sections of the manuscript, or (magnificently) commented extensively on large sections of the manuscript:

- Christophe Blaess is a consulting software engineer and professional trainer who specializes in industrial (realtime and embedded) applications of Linux. Christophe is the author of *Programmation système en C sous Linux*, a fine French book covering many of the same topics as this book. He generously read and commented on many chapters of my book.
- David Butenhof (Hewlett-Packard) was a member of the original working group for POSIX threads and for the Single UNIX Specification threads extensions, and is the author of *Programming with POSIX Threads*. He wrote the original DCE Threads reference implementation for the Open Software Foundation,

and was lead architect of the threads implementation for OpenVMS and Digital UNIX. David reviewed the threads chapters, suggested many improvements, and patiently corrected several details of my understanding of the POSIX threads API.

- Geoff Clare works at The Open Group on their UNIX conformance test suites, has been involved with UNIX standardization for more than 20 years, and is one of half a dozen key participants in the Austin Group, which develops the joint standard that forms POSIX.1 and the base volumes of the Single UNIX Specification. Geoff provided detailed review of parts of the manuscript related to standard UNIX interfaces, patiently and politely suggested numerous fixes and improvements, spotted many obscure bugs, and provided much assistance in focusing on the importance of standards for portable programming.
- Loïc Domagné (then at German Air Traffic Control) is a software systems engineer working on the design and development of distributed, concurrent, and fault-tolerant embedded systems with hard realtime requirements. He provided review input for the threads specification in SUSv3, and is an enthusiastic educator and knowledgeable contributor in various online technical forums. Loïc carried out a detailed review of the threads chapters, as well as many other parts of the book. He also implemented a number of clever programs to verify details of the Linux threads implementation, provided a great deal of enthusiasm and encouragement, and proposed numerous ideas to improve the overall presentation of the material.
- Gert Döring programmed *mgetty* and *sendfax*, a pair of programs that together are one of the most widely used open source fax packages for UNIX and Linux. These days, he works mainly on building and operating large IPv4-based and IPv6-based networks, a task that includes working with colleagues across Europe to define the operational policies that ensure the smooth operation of the infrastructure of the Internet. Gert provided extensive and useful feedback on the chapters covering terminals, login accounting, process groups, sessions, and job control.
- Wolfram Gloger is an IT consultant who has worked on a range of Free and Open Source Software (FOSS) projects in the past decade and a half. Among other things, Wolfram is the implementer of the *malloc* package used in the GNU C library. Currently, he works on web services development, with a particular focus on E-learning, although he still does occasional work on the kernel and system libraries. Wolfram reviewed a number of chapters, especially helping with my discussion of memory-related topics.
- Fernando Gont is a member of the Centro de Estudios de Informática (CEDI) at the Universidad Tecnológica Nacional, Argentina. He focuses on Internet engineering, with active participation in the Internet Engineering Task Force (IETF), where he has authored a number of *Request for Comments* (RFC) documents. Fernando also works on security assessment of communications protocols for the UK Centre for the Protection of National Infrastructure (CPNI), and has produced the first thorough security assessment of the TCP and IP protocols. Fernando provided a very thorough review of the network programming chapters, explained many details of TCP/IP, and suggested a multitude of improvements to the material.

- Andreas Grünbacher (SUSE Labs) is a kernel hacker and author of the Linux implementation of extended attributes and POSIX access control lists. Andreas provided thorough review of many chapters, much encouragement, and the single comment that probably most changed the structure of the book.
- Christoph Hellwig is a Linux storage and file-systems consultant and a well-known kernel hacker who has worked on many parts of the Linux kernel. Christoph kindly took time out from writing and reviewing Linux kernel patches to review several chapters of this book, suggesting many useful corrections and improvements.
- Andreas Jaeger led the development of the Linux port to the x86-64 architecture. As a GNU C Library developer, he ported the library to x86-64, and helped make the library standards-conformant in several areas, especially in the math library. He is currently Program Manager for openSUSE at Novell. Andreas reviewed far more chapters than I could possibly have hoped, suggested a multitude of improvements, and warmly encouraged the ongoing work on the book.
- Rick Jones, also known as “Mr. Netperf” (Networked Systems Performance Curmudgeon at Hewlett-Packard), provided valuable review of the network programming chapters.
- Andi Kleen (then at SUSE Labs) is a well-known and long-term kernel hacker who has worked on many and diverse areas of the Linux kernel, including networking, error handling, scalability, and low-level architecture code. Andi did an extensive review of the material on network programming, expanded my knowledge of many details of the Linux TCP/IP implementation, and suggested many ways to improve my presentation of the subject.
- Martin Landers (Google) was still a student when I had the good fortune to meet him as a colleague. Since then, he has managed to pack rather a lot into a short time, having worked variously as software architect, IT trainer, and professional hacker. I was fortunate indeed to have Martin as a reviewer. He contributed numerous incisive comments and corrections that greatly improved many chapters of the book.
- Jamie Lokier is a well-known kernel hacker who has been contributing to Linux development for 15 years. He nowadays describes himself as “a consultant in solving difficult problems that often have embedded Linux somewhere.” Jamie provided an extraordinarily thorough review of the chapters on memory mappings, POSIX shared memory, and virtual memory operations. His comments corrected many details of my understanding of these topics and greatly improved the structure of the chapters.
- Barry Margolin has been a system programmer, system administrator, and support engineer throughout his 25-year career. He is currently a Senior Performance Engineer at Akamai Technologies. He is a frequent, well-respected contributor in various online forums discussing UNIX and Internet topics, and has reviewed a number of books on these topics. Barry reviewed a number of chapters of this book, suggesting many improvements.

- Paul Pluzhnikov (Google) was formerly the technical lead and a key developer of the *Insure++* memory-debugging tool. He is also a sometime *gdb* hacker, and a frequent responder in online forums answering questions on debugging, memory allocation, shared libraries, and run-time environments. Paul reviewed a wide range of chapters, suggesting many valuable improvements.
- John Reiser (with Tom London) carried out one of the earliest ports of UNIX to a 32-bit architecture: the VAX-11/780. He is also the creator of the *mmap()* system call. John reviewed many chapters (including, obviously, the chapter on *mmap()*), providing a multitude of historical insights and crystal-clear technical explanations that greatly improved the chapters.
- Anthony Robins (Associate Professor of Computer Science, University of Otago, New Zealand), a close friend of more than three decades, was the first reader of the drafts of several chapters, and offered valuable early comments and ongoing encouragement as the project evolved.
- Michael Schröder (Novell) is one of the main authors of the GNU *screen* program, a task that has imbued him with a thorough knowledge of the subtleties and differences in terminal-driver implementations. Michael reviewed the chapters covering terminals and pseudoterminals, and the chapter on process groups, sessions, and job control, providing much useful feedback.
- Manfred Spraul, who worked on the IPC code (among other things) in the Linux kernel, generously reviewed several of the chapters on IPC and suggested many improvements.
- Tom Swigg, a former UNIX training colleague at Digital, was an early reviewer who supplied important feedback on several chapters. A software engineer and IT trainer for more than 25 years, Tom currently works at London South Bank University, programming and supporting Linux in a VMware environment.
- Jens Thoms Törning is part of a fine tradition of physicists turned programmers, and has produced a variety of open source device drivers and other software. Jens read a surprisingly diverse collection of chapters, providing unique and valuable insight on how each could be improved.

Many other technical reviewers also read various parts of the book and made valuable comments. In alphabetical order by surname, thank you to George Anzinger (MontaVista Software), Stefan Becher, Krzysztof Benedyczak, Daniel Brahneborg, Andries Brouwer, Annabel Church, Dragan Cvetkovic, Floyd L. Davidson, Stuart Davidson (Hewlett-Packard Consulting), Kasper Dupont, Peter Fellingner (jambit GmbH), Mel Gorman (IBM), Niels Göllesch, Claus Gratzl, Serge Hallyn (IBM), Markus Hartinger (jambit GmbH), Richard Henderson (Red Hat), Andrew Josey (The Open Group), Dan Kegel (Google), Davide Libenzi, Robert Love (Google), H.J. Lu (Intel Corporation), Paul Marshall, Chris Mason, Michael Matz (SUSE), Trond Myklebust, James Peach, Mark Phillips (Automated Test Systems), Nick Piggin (SUSE Labs, Novell), Kay Johannes Potthoff, Florian Rammpp, Stephen Rothwell (Linux Technology Centre, IBM), Markus Schwaiger, Stephen Tweedie (Red Hat), Britta Vargas, Chris Wright, Michal Wronski, and Umberto Zamuner.

Aside from technical review, I received many other kinds of help from various people and organizations.

Thanks to the following people for answering technical questions: Jan Kara, Dave Kleikamp, and Jon Snader. Thanks to Claus Gratzl and Paul Marshall for system management assistance.

Thanks to the Linux Foundation (LF), which, during 2008, funded me as a Fellow to work full time on the *man-pages* project and on testing and design review of the Linux programming interface. Although the Fellowship provided no direct financial support for working on this book, it did keep me and my family fed, and the ability to focus full time on documenting and testing the Linux programming interface was a boon to my “private” project. At a more individual level, thanks to Jim Zemlin for being my “interface” while working at the LF, and to the members of the LF Technical Advisory Board, who supported my application for the Fellowship.

Thanks to Alejandro Forero Cuervo for suggesting the title of the book!

More than 25 years ago, Robert Biddle intrigued me during my first degree with tales of UNIX, C, and Ratfor; thank you. Thanks to the following people, who, although not directly connected with this project, encouraged me on the path of writing during my second degree at the University of Canterbury, New Zealand: Michael Howard, Jonathan Mane-Wheoki, Ken Strongman, Garth Fletcher, Jim Pollard, and Brian Haig.

The late Richard Stevens wrote several superb books on UNIX programming and TCP/IP, which I, like a multitude of programmers, have found to be a wonderful source of technical information over the years. Readers of those books will note several visual aspects that are similar between my book and those of Richard Stevens. This is no accident. As I considered how to design my book, and looked around more generally at book designs, time and again, the approach employed by Richard Stevens seemed the best solution, and where this was so, I have employed the same visual approach.

Thanks to the following people and organizations for providing UNIX systems that enabled me to run test programs and verify details on other UNIX implementations: Anthony Robins and Cathy Chandra, for test systems at the University of Otago, New Zealand; Martin Landers, Ralf Ebner, and Klaus Tilk, for test systems at the Technische Universität in Munich, Germany; Hewlett-Packard, for making their *testdrive* systems freely available on the Internet; and Paul de Weerd for providing OpenBSD access.

Heartfelt thanks to two Munich companies, and their owners, who, in addition to providing me with flexible employment and enjoyable colleagues, were extraordinarily generous in allowing me to use their offices while writing this book. Thanks to Thomas Kahabka and Thomas Gmelch of exolution GmbH, and, especially, to Peter Fellingner and Markus Hartinger of jambit GmbH.

Thanks for various kinds of help to the following people: Dan Randow, Karen Korrel, Claudio Scalmazzi, Michael Schüpbach, and Liz Wright. Thanks to Rob Suisted and Lynley Cook for the photographs used on the front and back covers.

Thanks to the following people who encouraged and supported me in various ways on this project: Deborah Church, Doris Church, and Annie Currie.

Thanks to the team at No Starch Press for all sorts of help on an enormous project. Thanks to Bill Pollock for being straight-talking from the start, having rock-solid faith in the project, and patiently keeping an eye on the project. Thanks to my initial production editor, Megan Dunchak. Thanks to my copyeditor, Marilyn Smith, who, despite my best efforts at clarity and consistency, still found many things to fix. Riley Hoffman had overall responsibility for layout and design of the book, and also took up the reins as production editor as we came into the home straight. Riley graciously bore with my many requests to achieve the right layout and produced a superb final result. Thank you.

I now know the truth of the cliché that a writer's family also pays the price of the writer's work. Thanks to Britta and Cecilia for their support, and for putting up with the many hours that I had to be away from family as I finished the book.

Permissions

The Institute of Electrical and Electronics Engineers and The Open Group have kindly given permission to quote portions of text from IEEE Std 1003.1, 2004 Edition, Standard for Information Technology—Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 6. The complete standard can be consulted online at <http://www.unix.org/version3/online.html>.

Web site and source code of example programs

You can find further information about this book, including errata and source code for the example programs, at <http://www.man7.org/tlpi>.

Feedback

I welcome bug reports, suggestions for code improvements, and fixes to further improve code portability. Book bugs and general suggestions about how the explanations in the book can be improved are also welcome. Since changes in the Linux programming interface are varied and sometimes too frequent for one person to keep up with, I would be happy to receive suggestions about new and changed features that should be covered in a future edition of this book.

Michael Timothy Kerrisk
Munich, Germany and Christchurch, New Zealand
August 2010

mtk@man7.org

1

HISTORY AND STANDARDS

Linux is a member of the UNIX family of operating systems. In computing terms, UNIX has a long history. The first part of this chapter provides a brief outline of that history. We begin with a description of the origins of the UNIX system and the C programming language, and then consider the two key currents that led to the Linux system as it exists today: the GNU project and the development of the Linux kernel.

One of the notable features of the UNIX system is that its development was not controlled by a single vendor or organization. Rather, many groups, both commercial and noncommercial, contributed to its evolution. This history resulted in many innovative features being added to UNIX, but also had the negative consequence that UNIX implementations diverged over time, so that writing applications that worked on all UNIX implementations became increasingly difficult. This led to a drive for standardization of UNIX implementations, which we discuss in the second part of this chapter.

Two definitions of the term UNIX are in common use. One of these denotes operating systems that have passed the official conformance tests for the Single UNIX Specification and thus are officially granted the right to be branded as “UNIX” by The Open Group (the holders of the UNIX trademark). At the time of writing, none of the free UNIX implementations (e.g., Linux and FreeBSD) has obtained this branding.

The other common meaning attached to the term UNIX denotes those systems that look and behave like classical UNIX systems (i.e., the original Bell Laboratories UNIX and its later principal offshoots, System V and BSD). By this definition, Linux is generally considered to be a UNIX system (as are the modern BSDs). Although we give close attention to the Single UNIX Specification in this book, we'll follow this second definition of UNIX, so that we'll often say things such as "Linux, like other UNIX implementations. . . ."

1.1 A Brief History of UNIX and C

The first UNIX implementation was developed in 1969 (the same year that Linus Torvalds was born) by Ken Thompson at Bell Laboratories, a division of the telephone corporation, AT&T. It was written in assembler for a Digital PDP-7 minicomputer. The name UNIX was a pun on MULTICS (*Multiplexed Information and Computing Service*), the name of an earlier operating system project in which AT&T collaborated with Massachusetts Institute of Technology (MIT) and General Electric. (AT&T had by this time withdrawn from the project in frustration at its initial failure to develop an economically useful system.) Thompson drew several ideas for his new operating system from MULTICS, including a tree-structured file system, a separate program for interpreting commands (the shell), and the notion of files as unstructured streams of bytes.

In 1970, UNIX was rewritten in assembly language for a newly acquired Digital PDP-11 minicomputer, then a new and powerful machine. Vestiges of this PDP-11 heritage can be found in various names still used on most UNIX implementations, including Linux.

A short time later, Dennis Ritchie, one of Thompson's colleagues at Bell Laboratories and an early collaborator on UNIX, designed and implemented the C programming language. This was an evolutionary process; C followed an earlier interpreted language, B. B was initially implemented by Thompson and drew many of its ideas from a still earlier programming language named BCPL. By 1973, C had matured to a point where the UNIX kernel could be almost entirely rewritten in the new language. UNIX thus became one of the earliest operating systems to be written in a high-level language, a fact that made subsequent porting to other hardware architectures possible.

The genesis of C explains why it, and its descendant C++, have come to be used so widely as system programming languages today. Previous widely used languages were designed with other purposes in mind: FORTRAN for mathematical tasks performed by engineers and scientists; COBOL for commercial systems processing streams of record-oriented data. C filled a hitherto empty niche, and unlike FORTRAN and COBOL (which were designed by large committees), the design of C arose from the ideas and needs of a few individuals working toward a single goal: developing a high-level language for implementing the UNIX kernel and associated software. Like the UNIX operating system itself, C was designed by professional programmers for their own use. The resulting language was small, efficient, powerful, terse, modular, pragmatic, and coherent in its design.

UNIX First through Sixth editions

Between 1969 and 1979, UNIX went through a number of releases, known as *editions*. Essentially, these releases were snapshots of the evolving development version at AT&T. [Salus, 1994] notes the following dates for the first six editions of UNIX:

- First Edition, November 1971: By this time, UNIX was running on the PDP-11 and already had a FORTRAN compiler and versions of many programs still used today, including *ar*, *cat*, *chmod*, *chown*, *cp*, *dc*, *ed*, *find*, *ln*, *ls*, *mail*, *mkdir*, *mv*, *rm*, *sh*, *su*, and *who*.
- Second Edition, June 1972: By this time, UNIX was installed on ten machines within AT&T.
- Third Edition, February 1973: This edition included a C compiler and the first implementation of pipes.
- Fourth Edition, November 1973: This was the first version to be almost totally written in C.
- Fifth Edition, June 1974: By this time, UNIX was installed on more than 50 systems.
- Sixth Edition, May 1975: This was the first edition to be widely used outside AT&T.

Over the period of these releases, the use and reputation of UNIX began to spread, first within AT&T, and then beyond. An important contribution to this growing awareness was the publication of a paper on UNIX in the widely read journal *Communications of the ACM* ([Ritchie & Thompson, 1974]).

At this time, AT&T held a government-sanctioned monopoly on the US telephone system. The terms of AT&T's agreement with the US government prevented it from selling software, which meant that it could not sell UNIX as a product. Instead, beginning in 1974 with Fifth Edition, and especially with Sixth Edition, AT&T licensed UNIX for use in universities for a nominal distribution fee. The university distributions included documentation and the kernel source code (about 10,000 lines at the time).

AT&T's release of UNIX into universities greatly contributed to the popularity and use of the operating system, and by 1977, UNIX was running at some 500 sites, including 125 universities in the United States and several other countries. UNIX offered universities an interactive multiuser operating system that was cheap yet powerful, at a time when commercial operating systems were very expensive. It also gave university computer science departments the source code of a real operating system, which they could modify and offer to their students to learn from and experiment with. Some of these students, armed with UNIX knowledge, became UNIX evangelists. Others went on to found or join the multitude of startup companies selling inexpensive computer workstations running the easily ported UNIX operating system.

The birth of BSD and System V

January 1979 saw the release of Seventh Edition UNIX, which improved the reliability of the system and provided an enhanced file system. This release also contained a number of new tools, including *awk*, *make*, *sed*, *tar*, *uucp*, the Bourne shell,

and a FORTRAN 77 compiler. The release of Seventh Edition is also significant because, from this point, UNIX diverged into two important variants: BSD and System V, whose origins we now briefly describe.

Thompson spent the 1975/1976 academic year as a visiting professor at the University of California at Berkeley, the university from which he had graduated. There, he worked with several graduate students, adding many new features to UNIX. (One of these students, Bill Joy, subsequently went on to cofound Sun Microsystems, an early entry in the UNIX workstation market.) Over time, many new tools and features were developed at Berkeley, including the *C shell*, the *vi* editor, an improved file system (the *Berkeley Fast File System*), *sendmail*, a Pascal compiler, and virtual memory management on the new Digital VAX architecture.

Under the name Berkeley Software Distribution (BSD), this version of UNIX, including its source code, came to be widely distributed. The first full distribution was 3BSD in December 1979. (Earlier releases from Berkeley—BSD and 2BSD—were distributions of new tools produced at Berkeley, rather than complete UNIX distributions.)

In 1983, the *Computer Systems Research Group* at the University of California at Berkeley released 4.2BSD. This release was significant because it contained a complete TCP/IP implementation, including the sockets application programming interface (API) and a variety of networking tools. 4.2BSD and its predecessor 4.1BSD became widely distributed within universities around the world. They also formed the basis for SunOS (first released in 1983), the UNIX variant sold by Sun. Other significant BSD releases were 4.3BSD, in 1986, and the final release, 4.4BSD, in 1993.

The very first ports of the UNIX system to hardware other than the PDP-11 occurred during 1977 and 1978, when Dennis Ritchie and Steve Johnson ported it to the Interdata 8/32 and Richard Miller at the University of Wollongong in Australia simultaneously ported it to the Interdata 7/32. The Berkeley Digital VAX port was based on an earlier (1978) port by John Reiser and Tom London. Known as 32V, this port was essentially the same as Seventh Edition for the PDP-11, except for the larger address space and wider data types.

In the meantime, US antitrust legislation forced the breakup of AT&T (legal maneuvers began in the mid-1970s, and the breakup became effective in 1982), with the consequence that, since it no longer held a monopoly on the telephone system, the company was permitted to market UNIX. This resulted in the release of System III (three) in 1981. System III was produced by AT&T's UNIX Support Group (USG), which employed many hundreds of developers to enhance UNIX and develop UNIX applications (notably, document preparation packages and software development tools). The first release of System V (five) followed in 1983, and a series of releases led to the definitive System V Release 4 (SVR4) in 1989, by which time System V had incorporated many features from BSD, including networking facilities. System V was licensed to a variety of commercial vendors, who used it as the basis of their UNIX implementations.

Thus, in addition to the various BSD distributions spreading through academia, by the late 1980s, UNIX was available in a range of commercial implementations on various hardware. These implementations included Sun's SunOS and later Solaris, Digital's Ultrix and OSF/1 (nowadays, after a series of renamings and

acquisitions, HP Tru64 UNIX), IBM's AIX, Hewlett-Packard's (HP's) HP-UX, NeXT's NeXTStep, A/UX for the Apple Macintosh, and Microsoft and SCO's XENIX for the Intel x86-32 architecture. (Throughout this book, the Linux implementation for x86-32 is referred to as Linux/x86-32.) This situation was in sharp contrast to the typical proprietary hardware/operating system scenarios of the time, where each vendor produced one, or at most a few, proprietary computer chip architectures, on which they sold their own proprietary operating system(s). The proprietary nature of most vendor systems meant that purchasers were locked into one vendor. Switching to another proprietary operating system and hardware platform could become very expensive because of the need to port existing applications and retrain staff. This factor, coupled with the appearance of cheap single-user UNIX workstations from a variety of vendors, made the portable UNIX system increasingly attractive from a commercial perspective.

1.2 A Brief History of Linux

The term *Linux* is commonly used to refer to the entire UNIX-like operating system of which the Linux kernel forms a part. However, this is something of a misnomer, since many of the key components contained within a typical commercial Linux distribution actually originate from a project that predates the inception of Linux by several years.

1.2.1 The GNU Project

In 1984, Richard Stallman, an exceptionally talented programmer who had been working at MIT, set to work on creating a “free” UNIX implementation. Stallman's outlook was a moral one, and *free* was defined in a legal sense, rather than a financial sense (see <http://www.gnu.org/philosophy/free-sw.html>). Nevertheless, the legal freedom that Stallman described carried with it the implicit consequence that software such as operating systems would be available at no or very low cost.

Stallman militated against the legal restrictions placed on proprietary operating systems by computer vendors. These restrictions meant that purchasers of computer software in general could not see the source code of the software they were buying, and they certainly could not copy, change, or redistribute it. He pointed out that such a framework encouraged programmers to compete with each other and hoard their work, rather than to cooperate and share it.

In response, Stallman started the GNU project (a recursively defined acronym for “GNU's not UNIX”) to develop an entire, freely available, UNIX-like system, consisting of a kernel and all associated software packages, and encouraged others to join him. In 1985, Stallman founded the Free Software Foundation (FSF), a non-profit organization to support the GNU project as well as the development of free software in general.

When the GNU project was started, BSD was not free in the sense that Stallman meant. Use of BSD still required a license from AT&T, and users could not freely modify and redistribute the AT&T code that formed part of BSD.

One of the important results of the GNU project was the development of the GNU *General Public License* (GPL), the legal embodiment of Stallman's notion of free

software. Much of the software in a Linux distribution, including the kernel, is licensed under the GPL or one of a number of similar licenses. Software licensed under the GPL must be made available in source code form, and must be freely redistributable under the terms of the GPL. Modifications to GPL-licensed software are freely permitted, but any distribution of such modified software must also be under the terms of the GPL. If the modified software is distributed in executable form, the author must also allow any recipients the option of obtaining the modified source for no more than the cost of distribution. The first version of the GPL was released in 1989. The current version of the license, version 3, was released in 2007. Version 2 of the license, released in 1991, remains in wide use, and is the license used for the Linux kernel. (Discussions of various free software licenses can be found in [St. Laurent, 2004] and [Rosen, 2005].)

The GNU project did not initially produce a working UNIX kernel, but did produce a wide range of other programs. Since these programs were designed to run on a UNIX-like operating system, they could be, and were, used on existing UNIX implementations and, in some cases, even ported to other operating systems. Among the more well-known programs produced by the GNU project are the *Emacs* text editor, *GCC* (originally the GNU C compiler, but now renamed the GNU compiler collection, comprising compilers for C, C++, and other languages), the *bash* shell, and *glibc* (the GNU C library).

By the early 1990s, the GNU project had produced a system that was virtually complete, except for one important component: a working UNIX kernel. The GNU project had started work on an ambitious kernel design, known as the GNU/HURD, based on the Mach microkernel. However, the HURD was far from being in a form that could be released. (At the time of writing, work continues on the HURD, which currently runs only on the x86-32 architecture.)

Because a significant part of the program code that constitutes what is commonly known as the Linux system actually derives from the GNU project, Stallman prefers to use the term *GNU/Linux* to refer to the entire system. The question of naming (Linux versus GNU/Linux) is the source of some debate in the free software community. Since this book is primarily concerned with the API of the Linux kernel, we'll generally use the term *Linux*.

The stage was set. All that was required was a working kernel to go with the otherwise complete UNIX system already produced by the GNU project.

1.2.2 The Linux Kernel

In 1991, Linus Torvalds, a Finnish student at the University of Helsinki, was inspired to write an operating system for his Intel 80386 PC. In the course of his studies, Torvalds had come into contact with Minix, a small UNIX-like operating system kernel developed in the mid-1980s by Andrew Tanenbaum, a university professor in Holland. Tanenbaum made Minix, complete with source code, available as a tool for teaching operating system design in university courses. The Minix kernel could be built and run on a 386 system. However, since its primary purpose was as a teaching tool, it was designed to be largely independent of the hardware architecture, and it did not take full advantage of the 386 processor's capabilities.

Torvalds therefore started on a project to create an efficient, full-featured UNIX kernel to run on the 386. Over a few months, Torvalds developed a basic kernel that allowed him to compile and run various GNU programs. Then, on October 5, 1991, Torvalds requested the help of other programmers, making the following now much-quoted announcement of version 0.02 of his kernel in the *comp.os.minix* Usenet newsgroup:

Do you pine for the nice days of Minix-1.1, when men were men and wrote their own device drivers? Are you without a nice project and just dying to cut your teeth on a OS you can try to modify for your needs? Are you finding it frustrating when everything works on Minix? No more all-nighters to get a nifty program working? Then this post might be just for you. As I mentioned a month ago, I'm working on a free version of a Minix-look-alike for AT-386 computers. It has finally reached the stage where it's even usable (though may not be depending on what you want), and I am willing to put out the sources for wider distribution. It is just version 0.02 . . . but I've successfully run bash, gcc, gnu-make, gnu-sed, compress, etc. under it.

Following a time-honored tradition of giving UNIX clones names ending with the letter X, the kernel was (eventually) baptized Linux. Initially, Linux was placed under a more restrictive license, but Torvalds soon made it available under the GNU GPL.

The call for support proved effective. Other programmers joined Torvalds in the development of Linux, adding various features, such as an improved file system, networking support, device drivers, and multiprocessor support. By March 1994, the developers were able to release version 1.0. Linux 1.2 appeared in March 1995, Linux 2.0 in June 1996, Linux 2.2 in January 1999, and Linux 2.4 in January 2001. Work on the 2.5 development kernel began in November 2001, and led to the release of Linux 2.6 in December 2003.

An aside: the BSDs

It is worth noting that another free UNIX was already available for the x86-32 during the early 1990s. Bill and Lynne Jolitz had developed a port of the already mature BSD system for the x86-32, known as 386/BSD. This port was based on the BSD Net/2 release (June 1991), a version of the 4.3BSD source code in which all remaining proprietary AT&T source code had either been replaced or, in the case of six source code files that could not be trivially rewritten, removed. The Jolitizes ported the Net/2 code to x86-32, rewrote the missing source files, and made the first release (version 0.0) of 386/BSD in February 1992.

After an initial wave of success and popularity, work on 386/BSD lagged for various reasons. In the face of an increasingly large backlog of patches, two alternative development groups soon appeared, creating their own releases based on 386/BSD: NetBSD, which emphasizes portability to a wide range of hardware platforms, and FreeBSD, which emphasizes performance and is the most widespread of the modern BSDs. The first NetBSD release was 0.8, in April 1993. The first FreeBSD CD-ROM (version 1.0) appeared in December 1993. Another BSD, OpenBSD, appeared in 1996 (as an initial version numbered 2.0) after forking from the NetBSD project. OpenBSD emphasizes security. In mid-2003, a new BSD,

DragonFly BSD, appeared after a split from FreeBSD 4.x. DragonFly BSD takes a different approach from FreeBSD 5.x with respect to design for symmetric multiprocessing (SMP) architectures.

Probably no discussion of the BSDs in the early 1990s is complete without mention of the lawsuits between UNIX System Laboratories (USL, the AT&T subsidiary spun off to develop and market UNIX) and Berkeley. In early 1992, the company Berkeley Software Design, Incorporated (BSDi, nowadays part of Wind River) began distributing a commercially supported BSD UNIX, BSD/OS, based on the Net/2 release and the Jolitzes' 386/BSD additions. BSDi distributed binaries and source code for \$995 (US dollars), and advised potential customers to use their telephone number 1-800-ITS-UNIX.

In April 1992, USL filed suit against BSDi in an attempt to prevent BSDi from selling a product that USL claimed was still encumbered by proprietary USL source code and trade secrets. USL also demanded that BSDi cease using the deceptive telephone number. The suit was eventually widened to include a claim against the University of California. The court ultimately dismissed all but two of USL's claims, and a countersuit by the University of California against USL ensued, in which the university claimed that USL had not given due credit for the use of BSD code in System V.

While these suits were pending, USL was acquired by Novell, whose CEO, the late Ray Noorda, stated publicly that he would prefer to compete in the marketplace rather than in the court. Settlement was finally reached in January 1994, with the University of California being required to remove 3 of the 18,000 files in the Net/2 release, make some minor changes to a few other files, and add USL copyright notices to around 70 other files, which the university nevertheless could continue to distribute freely. This modified system was released as 4.4BSD-Lite in June 1994. (The last release from the university was 4.4BSD-Lite, Release 2 in June 1995.) At this point, the terms of the legal settlement required BSDi, FreeBSD, and NetBSD to replace their Net/2 base with the modified 4.4BSD-Lite source code. As [McKusick et al., 1996] notes, although this caused some delay in the development of the BSD derivatives, it also had the positive effect that these systems resynchronized with the three years of development work done by the university's Computer Systems Research Group since the release of Net/2.

Linux kernel version numbers

Like most free software projects, Linux follows a release-early, release-often model, so that new kernel revisions appear frequently (sometimes even daily). As the Linux user base increased, the release model was adapted to decrease disruption to existing users. Specifically, following the release of Linux 1.0, the kernel developers adopted a kernel version numbering scheme with each release numbered *x.y.z*: *x* representing a major version, *y* a minor version within that major version, and *z* a revision of the minor version (minor improvements and bug fixes).

Under this model, two kernel versions were always under development: a *stable* branch for use on production systems, which had an even minor version number, and a more volatile *development* branch, which carried the next higher odd minor version number. The theory—not always followed strictly in practice—was that all new features should be added in the current development kernel series, while new revisions in the stable kernel series should be restricted to minor improvements

and bug fixes. When the current development branch was deemed suitable for release, it became the new stable branch and was assigned an even minor version number. For example, the 2.3.z development kernel branch resulted in the 2.4 stable kernel branch.

Following the 2.6 kernel release, the development model was changed. The main motivation for this change arose from problems and frustrations caused by the long intervals between stable kernel releases. (Nearly three years passed between the release of Linux 2.4.0 and 2.6.0.) There have periodically been discussions about fine-tuning this model, but the essential details have remained as follows:

- There is no longer a separation between stable and development kernels. Each new 2.6.z release can contain new features, and goes through a life cycle that begins with the addition of new features, which are then stabilized over the course of a number of candidate release versions. When a candidate version is deemed sufficiently stable, it is released as kernel 2.6.z. Release cycles are typically about three months long.
- Sometimes, a stable 2.6.z release may require minor patches to fix bugs or security problems. If these fixes have a sufficiently high priority, and the patches are deemed simple enough to be “obviously” correct, then, rather than waiting for the next 2.6.z release, they are applied to create a release with a number of the form 2.6.z.r, where *r* is a sequential number for a minor revision of this 2.6.z kernel.
- Additional responsibility is shifted onto distribution vendors to ensure the stability of the kernel provided with a distribution.

Later chapters will sometimes note the kernel version in which a particular API change (i.e., new or modified system call) occurred. Although, prior to the 2.6.z series, most kernel changes occurred in the odd-numbered development branches, we’ll generally refer to the following stable kernel version in which the change appeared, since most application developers would normally be using a stable kernel, rather than one of the development kernels. In many cases, the manual pages note the precise development kernel in which a particular feature appeared or changed.

For changes that appear in the 2.6.z kernel series, we note the precise kernel version. When we say that a feature is new in kernel 2.6, without a *z* revision number, we mean a feature that was implemented in the 2.5 development kernel series and first appeared in the stable kernel at version 2.6.0.

At the time of writing, the 2.4 stable Linux kernel is still supported by maintainers who incorporate essential patches and bug fixes, and periodically release new revisions. This allows installed systems to continue to use 2.4 kernels, rather than being forced to upgrade to a new kernel series (which may entail significant work in some cases).

Ports to other hardware architectures

During the initial development of Linux, efficient implementation on the Intel 80386 was the primary goal, rather than portability to other processor architectures. However, with the increasing popularity of Linux, ports to other processor

architectures began to appear, starting with an early port to the Digital Alpha chip. The list of hardware architectures to which Linux has been ported continues to grow and includes x86-64, Motorola/IBM PowerPC and PowerPC64, Sun SPARC and SPARC64 (UltraSPARC), MIPS, ARM (Acorn), IBM zSeries (formerly System/390), Intel IA-64 (Itanium; see [Mosberger & Eranian, 2002]), Hitachi SuperH, HP PA-RISC, and Motorola 68000.

Linux distributions

Precisely speaking, the term *Linux* refers just to the kernel developed by Linus Torvalds and others. However, the term *Linux* is commonly used to mean the kernel, plus a wide range of other software (tools and libraries) that together make a complete operating system. In the very early days of Linux, the user was required to assemble all of this software, create a file system, and correctly place and configure all of the software on that file system. This demanded considerable time and expertise. As a result, a market opened for Linux distributors, who created packages (*distributions*) to automate most of the installation process, creating a file system and installing the kernel and other required software.

The earliest distributions appeared in 1992, and included MCC Interim Linux (Manchester Computing Centre, UK), TAMU (Texas A&M University), and SLS (SoftLanding Linux System). The oldest surviving commercial distribution, Slackware, appeared in 1993. The noncommercial Debian distribution appeared at around the same time, and SUSE and Red Hat soon followed. The currently very popular Ubuntu distribution first appeared in 2004. Nowadays, many distribution companies also employ programmers who actively contribute to existing free software projects or initiate new projects.

1.3 Standardization

By the late 1980s, the wide variety of available UNIX implementations also had its drawbacks. Some UNIX implementations were based on BSD, others were based on System V, and some drew features from both variants. Furthermore, each commercial vendor had added extra features to its own implementation. The consequence was that moving software and people from one UNIX implementation to another became steadily more difficult. This situation created strong pressure for standardization of the C programming language and the UNIX system, so that applications could more easily be ported from one system to another. We now look at the resulting standards.

1.3.1 The C Programming Language

By the early 1980s, C had been in existence for ten years, and was implemented on a wide variety of UNIX systems and on other operating systems. Minor differences had arisen between the various implementations, in part because certain aspects of how the language should function were not detailed in the existing de facto standard for C, Kernighan and Ritchie's 1978 book, *The C Programming Language*. (The older C syntax described in that book is sometimes called *traditional C* or *K&R C*.) Furthermore, the appearance of C++ in 1985 highlighted certain improvements and additions that could be made to C without breaking existing programs, notably

function prototypes, structure assignment, type qualifiers (*const* and *volatile*), enumeration types, and the *void* keyword.

These factors created a drive for C standardization that culminated in 1989 with the approval of the American National Standards Institute (ANSI) C standard (X3.159-1989), which was subsequently adopted in 1990 as an International Standards Organization (ISO) standard (ISO/IEC 9899:1990). As well as defining the syntax and semantics of C, this standard described the operation of the standard C library, which includes the *stdio* functions, string-handling functions, math functions, various header files, and so on. This version of C is usually known as *C89* or (less commonly) *ISO C90*, and is fully described in the second (1988) edition of Kernighan and Ritchie's *The C Programming Language*.

A revision of the C standard was adopted by ISO in 1999 (ISO/IEC 9899:1999; see <http://www.open-std.org/jtc1/sc22/wg14/www/standards>). This standard is usually referred to as C99, and includes a range of changes to the language and its standard library. These changes include the addition of *long long* and Boolean data types, C++-style (*//*) comments, restricted pointers, and variable-length arrays. (At the time of writing, work is in progress on a further revision of the C standard, informally named C1X. The new standard is expected to be ratified in 2011.)

The C standards are independent of the details of any operating system; that is, they are not tied to the UNIX system. This means that C programs written using only the standard C library should be portable to any computer and operating system providing a C implementation.

Historically, C89 was often called *ANSI C*, and this term is sometimes still used with that meaning. For example, *gcc* employs that meaning; its *-ansi* qualifier means “support all ISO C90 programs.” However, we avoid this term because it is now somewhat ambiguous. Since the ANSI committee adopted the C99 revision, properly speaking, ANSI C is now C99.

1.3.2 The First POSIX Standards

The term POSIX (an abbreviation of *Portable Operating System Interface*) refers to a group of standards developed under the auspices of the Institute of Electrical and Electronic Engineers (IEEE), specifically its Portable Application Standards Committee (PASC, <http://www.pasc.org/>). The goal of the PASC standards is to promote application portability at the source code level.

The name POSIX was suggested by Richard Stallman. The final X appears because the names of most UNIX variants end in X. The standard notes that the name should be pronounced “pahz-icks,” like “positive.”

The most interesting of the POSIX standards for our purposes are the first POSIX standard, referred to as POSIX.1 (or, more fully, POSIX 1003.1), and the subsequent POSIX.2 standard.

POSIX.1 and POSIX.2

POSIX.1 became an IEEE standard in 1988 and, with minor revisions, was adopted as an ISO standard in 1990 (ISO/IEC 9945-1:1990). (The original POSIX standards are not available online, but can be purchased from the IEEE at <http://www.ieee.org/>.)

POSIX.1 was initially based on an earlier (1984) unofficial standard produced by an association of UNIX vendors called */usr/group*.

POSIX.1 documents an API for a set of services that should be made available to a program by a conforming operating system. An operating system that does this can be certified as *POSIX.1 conformant*.

POSIX.1 is based on the UNIX system call and the C library function API, but it doesn't require any particular implementation to be associated with this interface. This means that the interface can be implemented by any operating system, not specifically a UNIX operating system. In fact, some vendors have added APIs to their proprietary operating systems that make them POSIX.1 conformant, while at the same time leaving the underlying operating system largely unchanged.

A number of extensions to the original POSIX.1 standard were also important. IEEE POSIX 1003.1b (POSIX.1b, formerly called POSIX.4 or POSIX 1003.4), ratified in 1993, contains a range of realtime extensions to the base POSIX standard. IEEE POSIX 1003.1c (POSIX.1c), ratified in 1995, is the definition of POSIX threads. In 1996, a revised version of the POSIX.1 standard (ISO/IEC 9945-1:1996) was produced, leaving the core text unchanged, but incorporating the realtime and threads extensions. IEEE POSIX 1003.1g (POSIX.1g) defined the networking APIs, including sockets. IEEE POSIX 1003.1d (POSIX.1d), ratified in 1999, and POSIX.1j, ratified in 2000, defined additional realtime extensions to the POSIX base standard.

The POSIX.1b realtime extensions include file synchronization; asynchronous I/O; process scheduling; high-precision clocks and timers; and interprocess communication using semaphores, shared memory, and message queues. The prefix *POSIX* is often applied to the three interprocess communication methods to distinguish them from the similar, but older, System V semaphores, shared memory, and message queues.

A related standard, POSIX.2 (1992, ISO/IEC 9945-2:1993), standardized the shell and various UNIX utilities, including the command-line interface of the C compiler.

FIPS 151-1 and FIPS 151-2

FIPS is an abbreviation for Federal Information Processing Standard, the name of a set of standards specified by the US government for the purchase of its computer systems. In 1989, FIPS 151-1 was published. This standard was based on the 1988 IEEE POSIX.1 standard and the draft ANSI C standard. The main difference between FIPS 151-1 and POSIX.1 (1988) was that the FIPS standard required some features that POSIX.1 left as optional. Because the US government is a major purchaser of computer systems, most computer vendors ensured that their UNIX systems conformed to the FIPS 151-1 version of POSIX.1.

FIPS 151-2 aligned with the 1990 ISO edition of POSIX.1, but was otherwise unchanged. The now outdated FIPS 151-2 was withdrawn as a standard in February 2000.

1.3.3 X/Open Company and The Open Group

X/Open Company was a consortium formed by an international group of computer vendors to adopt and adapt existing standards in order to produce a comprehensive, consistent set of open systems standards. It produced the *X/Open Portability Guide*, a series of portability guides based on the POSIX standards. The first important release of this guide was Issue 3 (XPG3) in 1989, followed by XPG4 in 1992. XPG4 was revised in 1994, which resulted in XPG4 version 2, a standard that also incorporated important parts of AT&T's System V Interface Definition Issue 3, described in Section 1.3.7. This revision was also known as *Spec 1170*, with 1170 referring to the number of *interfaces*—functions, header files, and commands—defined by the standard.

When Novell, which acquired the UNIX systems business from AT&T in early 1993, later divested itself of that business, it transferred the rights to the UNIX trademark to X/Open. (The plan to make this transfer was announced in 1993, but legal requirements delayed the transfer until early 1994.) XPG4 version 2 was subsequently repackaged as the *Single UNIX Specification* (SUS, or sometimes SUSv1), and is also known as *UNIX 95*. This repackaging included XPG4 version 2, the X/Open Curses Issue 4 version 2 specification, and the X/Open Networking Services (XNS) Issue 4 specification. Version 2 of the Single UNIX Specification (SUSv2, <http://www.unix.org/version2/online.html>) appeared in 1997, and UNIX implementations certified against this specification can call themselves *UNIX 98*. (This standard is occasionally also referred to as XPG5.)

In 1996, X/Open merged with the *Open Software Foundation* (OSF) to form *The Open Group*. Nearly every company or organization involved with the UNIX system is now a member of The Open Group, which continues to develop API standards.

OSF was one of two vendor consortia formed during the UNIX wars of the late 1980s. Among others, OSF included Digital, IBM, HP, Apollo, Bull, Nixdorf, and Siemens. OSF was formed primarily in response to the threat created by a business alliance between AT&T (the originators of UNIX) and Sun (the most powerful player in the UNIX workstation market). Consequently, AT&T, Sun, and other companies formed the rival *UNIX International* consortium.

1.3.4 SUSv3 and POSIX.1-2001

Beginning in 1999, the IEEE, The Open Group, and the ISO/IEC Joint Technical Committee 1 collaborated in the *Austin Common Standards Revision Group* (CSRG, <http://www.opengroup.org/austin/>) with the aim of revising and consolidating the POSIX standards and the Single UNIX Specification. (The Austin Group is so named because its inaugural meeting was in Austin, Texas in September 1998.) This resulted in the ratification of POSIX 1003.1-2001, sometimes just called POSIX.1-2001, in December 2001 (subsequently approved as an ISO standard, ISO/IEC 9945:2002).

POSIX 1003.1-2001 replaces SUSv2, POSIX.1, POSIX.2, and a raft of other earlier POSIX standards. This standard is also known as the Single UNIX Specification Version 3, and we'll generally refer to it in the remainder of this book as *SUSv3*.

The SUSv3 base specifications consists of around 3700 pages, divided into the following four parts:

- *Base Definitions* (XBD): This part contains definitions, terms, concepts, and specifications of the contents of header files. A total of 84 header file specifications are provided.
- *System Interfaces* (XSH): This part begins with various useful background information. Its bulk consists of the specification of various functions (which are implemented as either system calls or library functions on specific UNIX implementations). A total of 1123 system interfaces are included in this part.
- *Shell and Utilities* (XCU): This specifies the operation of the shell and various UNIX commands. A total of 160 utilities are specified in this part.
- *Rationale* (XRAT): This part includes informative text and justifications relating to the earlier parts.

In addition, SUSv3 includes the *X/Open CURSES Issue 4 Version 2* (XCURSES) specification, which specifies 372 functions and 3 header files for the *curses* screen-handling API.

In all, 1742 interfaces are specified in SUSv3. By contrast, POSIX.1-1990 (with FIPS 151-2) specified 199 interfaces, and POSIX.2-1992 specified 130 utilities.

SUSv3 is available online at <http://www.unix.org/version3/online.html>. UNIX implementations certified against SUSv3 can call themselves *UNIX 03*.

There have been various minor fixes and improvements for problems discovered since the ratification of the original SUSv3 text. These have resulted in the appearance of *Technical Corrigendum Number 1*, whose improvements were incorporated in a 2003 revision of SUSv3, and *Technical Corrigendum Number 2*, whose improvements were incorporated in a 2004 revision.

POSIX conformance, XSI conformance, and the XSI extension

Historically, the SUS (and XPG) standards deferred to the corresponding POSIX standards and were structured as functional supersets of POSIX. As well as specifying additional interfaces, the SUS standards made mandatory many of the interfaces and behaviors that were deemed optional in POSIX.

This distinction survives somewhat more subtly in POSIX 1003.1-2001, which is both an IEEE standard and an Open Group Technical Standard (i.e., as noted already, it is a consolidation of earlier POSIX and SUS standards). This document defines two levels of conformance:

- *POSIX conformance*: This defines a baseline of interfaces that a conforming implementation must provide. It permits the implementation to provide other optional interfaces.
- *X/Open System Interface (XSI) conformance*: To be XSI conformant, an implementation must meet all of the requirements of POSIX conformance and also must provide a number of interfaces and behaviors that are only optionally required for POSIX conformance. An implementation must reach this level of conformance in order to obtain the *UNIX 03* branding from The Open Group.

The additional interfaces and behaviors required for XSI conformance are collectively known as the *XSI extension*. They include support for features such as threads, *mmap()* and *munmap()*, the *dlopen* API, resource limits, pseudoterminals, System V IPC, the *syslog* API, *poll()*, and login accounting.

In later chapters, when we talk about SUSv3 conformance, we mean XSI conformance.

Because POSIX and SUSv3 are now part of the same document, the additional interfaces and the selection of mandatory options required for SUSv3 are indicated via the use of shading and margin markings within the document text.

Unspecified and weakly specified

Occasionally, we refer to an interface as being “unspecified” or “weakly specified” within SUSv3.

By an *unspecified interface*, we mean one that is not defined at all in the formal standard, although in a few cases there are background notes or rationale text that mention the interface.

Saying that an interface is *weakly specified* is shorthand for saying that, while the interface is included in the standard, important details are left unspecified (commonly because the committee members could not reach an agreement due to differences in existing implementations).

When using interfaces that are unspecified or weakly specified, we have few guarantees when porting applications to other UNIX implementations. Nevertheless, in a few cases, such an interface is quite consistent across implementations, and where this is so, we generally note it as such.

LEGACY features

Sometimes, we note that SUSv3 marks a specified feature as *LEGACY*. This term denotes a feature that is retained for compatibility with older applications, but whose limitations mean that its use should be avoided in new applications. In many cases, some other API exists that provides equivalent functionality.

1.3.5 SUSv4 and POSIX.1-2008

In 2008, the Austin group completed a revision of the combined POSIX.1 and Single UNIX Specification. As with the preceding version of the standard, it consists of a base specification coupled with an XSI extension. We'll refer to this revision as SUSv4.

The changes in SUSv4 are less wide-ranging than those that occurred for SUSv3. The most significant changes are as follows:

- SUSv4 adds new specifications for a range of functions. Among the newly specified functions that we mention in this book are *dirfd()*, *fdopendir()*, *fexecve()*, *futimens()*, *mkdtemp()*, *psignal()*, *strsignal()*, and *utimensat()*. Another range of new file-related functions (e.g., *openat()*, described in Section 18.11) are analogs of existing functions (e.g., *open()*), but differ in that they interpret relative pathnames with respect to the directory referred to by an open file descriptor, rather than relative to the process's current working directory.

- Some functions specified as options in SUSv3 become a mandatory part of the base standard in SUSv4. For example, a number of functions that were part of the XSI extension in SUSv3 become part of the base standard in SUSv4. Among the functions that become mandatory in SUSv4 are those in the *dlopen* API (Section 42.1), the realtime signals API (Section 22.8), the POSIX semaphore API (Chapter 53), and the POSIX timers API (Section 23.6).
- Some functions in SUSv3 are marked as obsolete in SUSv4. These include *asctime()*, *ctime()*, *ftw()*, *gettimeofday()*, *getitimer()*, *setitimer()*, and *siginterrupt()*.
- Specifications of some functions that were marked as obsolete in SUSv3 are removed in SUSv4. These functions include *gethostbyname()*, *gethostbyaddr()*, and *vfork()*.
- Various details of existing specifications in SUSv3 are changed in SUSv4. For example, various functions are added to the list of functions that are required to be async-signal-safe (Table 21-1 on page 426).

In the remainder of this book, we note changes in SUSv4 where they are relevant to the topic being discussed.

1.3.6 UNIX Standards Timeline

Figure 1-1 summarizes the relationships between the various standards described in the preceding sections, and places the standards in chronological order. In this diagram, the solid lines indicate direct descent between standards, and the dashed arrows indicate cases where one standard influenced another standard, was incorporated as part of another standard, or simply deferred to another standard.

The situation with networking standards is somewhat complex. Standardization efforts in this area began in the late 1980s with the formation of the POSIX 1003.12 committee to standardize the sockets API, the X/Open Transport Interface (XTI) API (an alternative network programming API based on System V's Transport Layer Interface), and various associated APIs. The gestation of this standard occurred over several years, during which time POSIX 1003.12 was renamed POSIX 1003.1g. It was ratified in 2000.

In parallel with the development of POSIX 1003.1g, X/Open was also developing its X/Open Networking Specification (XNS). The first version of this specification, XNS Issue 4, was part of the first version of the Single UNIX Specification. It was succeeded by XNS Issue 5, which formed part of SUSv2. XNS Issue 5 was essentially the same as the then current (6.6) draft of POSIX.1g. This was followed by XNS Issue 5.2, which differed from XNS Issue 5 and the ratified POSIX.1g standard in marking the XTI API as obsolete and in including coverage of Internet Protocol version 6 (IPv6), which was being designed in the mid-1990s). XNS Issue 5.2 formed the basis for the networking material included in SUSv3, and is thus now superseded. For similar reasons, POSIX.1g was withdrawn as a standard soon after it was ratified.

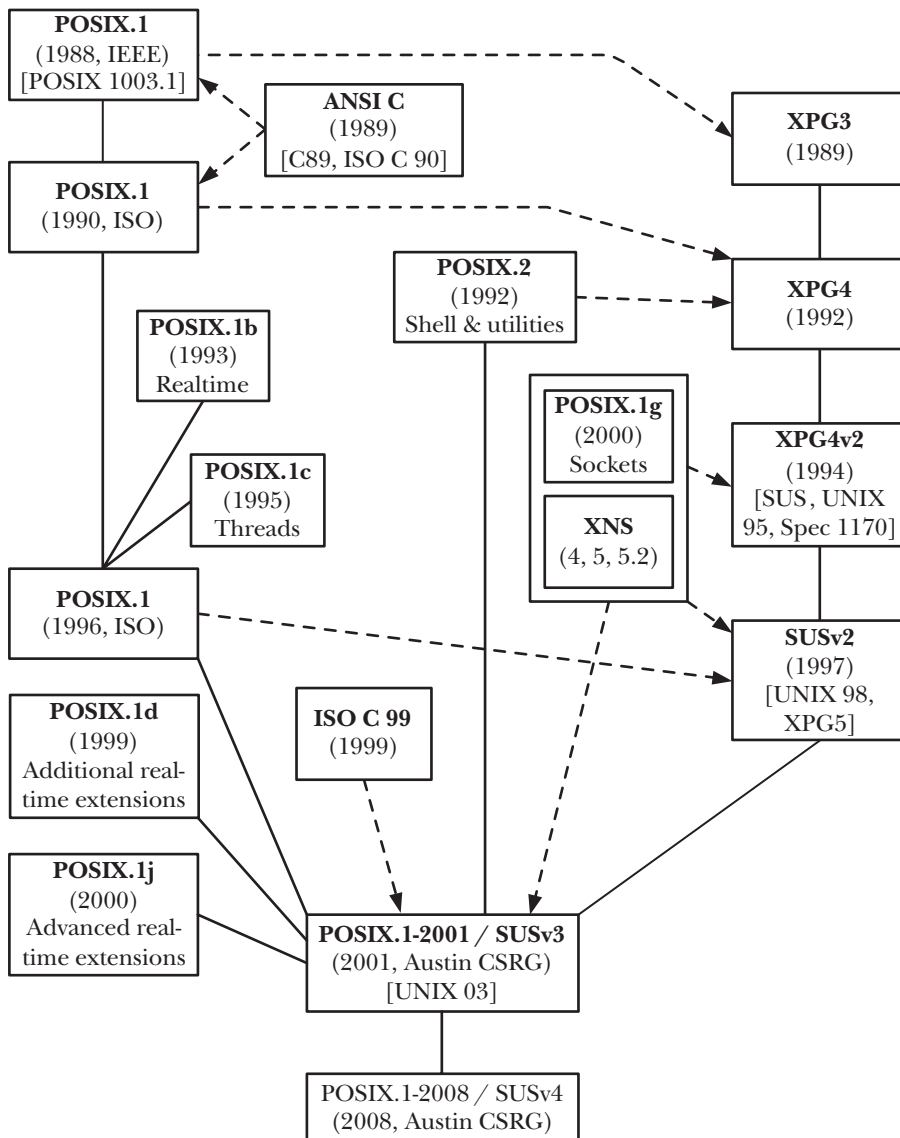


Figure 1-1: Relationships between various UNIX and C standards

1.3.7 Implementation Standards

In addition to the standards produced by independent or multiparty groups, reference is sometimes made to the two implementation standards defined by the final BSD release (4.4BSD) and AT&T's System V Release 4 (SVR4). The latter implementation standard was formalized by AT&T's publication of the System V Interface Definition (SVID). In 1989, AT&T published Issue 3 of the SVID, which

defined the interface that a UNIX implementation must provide in order to be able to call itself System V Release 4. (The SVID is available online at <http://www.sco.com/developers/devspecs/>.)

Because the behavior of some system calls and library functions varies between SVR4 and BSD, many UNIX implementations provide compatibility libraries and conditional-compilation facilities that emulate the behavior of whichever UNIX flavor is not used as the base for that particular implementation (see Section 3.6.1). This eases the burden of porting an application from another UNIX implementation.

1.3.8 Linux, Standards, and the Linux Standard Base

As a general goal, Linux (i.e., kernel, *glibc*, and tool) development aims to conform to the various UNIX standards, especially POSIX and the Single UNIX Specification. However, at the time of writing, no Linux distributions are branded as “UNIX” by The Open Group. The problems are time and expense. Each vendor distribution would need to undergo conformance testing to obtain this branding, and it would need to repeat this testing with each new distribution release. Nevertheless, it is the de facto near-conformance to various standards that has enabled Linux to be so successful in the UNIX market.

With most commercial UNIX implementations, the same company both develops and distributes the operating system. With Linux, things are different, in that implementation is separate from distribution, and multiple organizations—both commercial and noncommercial—handle Linux distribution.

Linus Torvalds doesn’t contribute to or endorse a particular Linux distribution. However, in terms of other individuals carrying out Linux development, the situation is more complex. Many developers working on the Linux kernel and on other free software projects are employed by various Linux distribution companies or work for companies (such as IBM and HP) with a strong interest in Linux. While these companies can influence the direction in which Linux moves by allocating programmer hours to certain projects, none of them controls Linux as such. And, of course, many of the other contributors to the Linux kernel and GNU projects work voluntarily.

At the time of writing, Torvalds is employed as a fellow at the Linux Foundation (<http://www.linux-foundation.org/>; formerly the Open Source Development Laboratory, OSDL), a nonprofit consortium of commercial and noncommercial organizations chartered to foster the growth of Linux.

Because there are multiple Linux distributors and because the kernel implementers don’t control the contents of distributions, there is no “standard” commercial Linux as such. Each Linux distributor’s kernel offering is typically based on a snapshot of the mainline (i.e., the Torvalds) kernel at a particular point in time, with a number of patches applied.

These patches typically provide features that, to a greater or lesser extent, are deemed commercially desirable, and thus able to provide competitive differentiation in the marketplace. In some cases, these patches are later accepted into the mainline kernel. In fact, some new kernel features were initially developed by a distribution company, and appeared in their distribution before eventually being

integrated into the mainline. For example, version 3 of the *Reiserfs* journaling file system was part of some Linux distributions long before it was accepted into the mainline 2.4 kernel.

The upshot of the preceding points is that there are (mostly minor) differences in the systems offered by the various Linux distribution companies. On a much smaller scale, this is reminiscent of the splits in implementations that occurred in the early years of UNIX. The Linux Standard Base (LSB) is an effort to ensure compatibility among the various Linux distributions. To do this, the LSB (<http://www.linux-foundation.org/en/LSB>) develops and promotes a set of standards for Linux systems with the aim of ensuring that binary applications (i.e., compiled programs) can run on any LSB-conformant system.

The binary portability promoted by the LSB contrasts with the source code portability promoted by POSIX. Source code portability means that we can write a C program and then successfully compile and run it on any POSIX-conformant system. Binary compatibility is much more demanding and is generally not feasible across different hardware platforms. It allows us to compile a program once for a given hardware platform, and then run that compiled program on any conformant implementation running on that hardware platform. Binary portability is an essential requirement for the commercial viability of independent software vendor (ISV) applications built for Linux.

1.4 Summary

The UNIX system was first implemented in 1969 on a Digital PDP-7 minicomputer by Ken Thompson at Bell Laboratories (part of AT&T). The operating system drew many ideas, as well as its punned name, from the earlier MULTICS system. By 1973, UNIX had been moved to the PDP-11 mini-computer and rewritten in C, a programming language designed and implemented at Bell Laboratories by Dennis Ritchie. Legally prevented from selling UNIX, AT&T instead distributed the complete system to universities for a nominal charge. This distribution included source code, and became very popular within universities, since it provided a cheap operating system whose code could be studied and modified by computer science academics and students.

The University of California at Berkeley played a key role in the development of the UNIX system. There, Ken Thompson and a number of graduate students extended the operating system. By 1979, the University was producing its own UNIX distribution, BSD. This distribution became widespread in academia and formed the basis for several commercial implementations.

Meanwhile, the breakup of the AT&T monopoly permitted the company to sell the UNIX system. This resulted in the other major variant of UNIX, System V, which also formed the basis for several commercial implementations.

Two different currents led to the development of (GNU/) Linux. One of these was the GNU project, founded by Richard Stallman. By the late 1980s, the GNU project had produced an almost complete, freely distributable UNIX implementation. The one part lacking was a working kernel. In 1991, inspired by the Minix kernel written by Andrew Tanenbaum, Linus Torvalds produced a working UNIX kernel for the Intel x86-32 architecture. Torvalds invited other programmers to

join him in improving the kernel. Many programmers did so, and, over time, Linux was extended and ported to a wide variety of hardware architectures.

The portability problems that arose from the variations in UNIX and C implementations that existed by the late 1980s created a strong pressure for standardization. The C language was standardized in 1989 (C89), and a revised standard was produced in 1999 (C99). The first attempt to standardize the operating system interface yielded POSIX.1, ratified as an IEEE standard in 1988, and as an ISO standard in 1990. During the 1990s, further standards were drafted, including various versions of the Single UNIX Specification. In 2001, the combined POSIX 1003.1-2001 and SUSv3 standard was ratified. This standard consolidates and extends various earlier POSIX standards and earlier versions of the Single UNIX Specification. In 2008, a less wide-ranging revision of the standard was completed, yielding the combined POSIX 1003.1-2008 and SUSv4 standard.

Unlike most commercial UNIX implementations, Linux separates implementation from distribution. Consequently, there is no single “official” Linux distribution. Each Linux distributor’s offering consists of a snapshot of the current stable kernel, with various patches applied. The LSB develops and promotes a set of standards for Linux systems with the aim of ensuring binary application compatibility across Linux distributions, so that compiled applications should be able to run on any LSB-conformant system running on the same hardware.

Further information

Further information about UNIX history and standards can be found in [Ritchie, 1984], [McKusick et al., 1996], [McKusick & Neville-Neil, 2005], [Libes & Ressler, 1989], [Garfinkel et al., 2003], [Stevens & Rago, 2005], [Stevens, 1999], [Quartermann & Wilhelm, 1993], [Goodheart & Cox, 1994], and [McKusick, 1999].

[Salus, 1994] is a detailed history of UNIX, from which much of the information at the beginning of this chapter was drawn. [Salus, 2008] provides a short history of Linux and other free software projects. Many details of the history of UNIX can also be found in the online book *History of UNIX*, written by Ronda Hauben. This book is available at <http://www.dei.isep.ipp.pt/~acc/docs/unix.html>. An extremely detailed timeline showing the releases of various UNIX implementations can be found at <http://www.levenez.com/unix/>.

[Josey, 2004] provides an overview of the history of the UNIX system and the development of SUSv3, guidance on how to use the specification, summary tables of the interfaces in SUSv3, and migration guides for the transitions from SUSv2 to SUSv3 and C89 to C99.

As well as providing software and documentation, the GNU web site (<http://www.gnu.org/>) contains a number of philosophical papers on the subject of free software. [Williams, 2002] is a biography of Richard Stallman.

Torvalds provides his own account of the development of Linux in [Torvalds & Diamond, 2001].

2

FUNDAMENTAL CONCEPTS

This chapter introduces a range of concepts related to Linux system programming. It is intended for readers who have worked primarily with other operating systems, or who have only limited experience with Linux or another UNIX implementation.

2.1 The Core Operating System: The Kernel

The term *operating system* is commonly used with two different meanings:

- To denote the entire package consisting of the central software managing a computer's resources and all of the accompanying standard software tools, such as command-line interpreters, graphical user interfaces, file utilities, and editors.
- More narrowly, to refer to the central software that manages and allocates computer resources (i.e., the CPU, RAM, and devices).

The term *kernel* is often used as a synonym for the second meaning, and it is with this meaning of the term *operating system* that we are concerned in this book.

Although it is possible to run programs on a computer without a kernel, the presence of a kernel greatly simplifies the writing and use of other programs, and increases the power and flexibility available to programmers. The kernel does this by providing a software layer to manage the limited resources of a computer.

The Linux kernel executable typically resides at the pathname `/boot/vmlinuz`, or something similar. The derivation of this filename is historical. On early UNIX implementations, the kernel was called `unix`. Later UNIX implementations, which implemented virtual memory, renamed the kernel as `vmunix`. On Linux, the filename mirrors the system name, with the `z` replacing the final `x` to signify that the kernel is a compressed executable.

Tasks performed by the kernel

Among other things, the kernel performs the following tasks:

- *Process scheduling*: A computer has one or more central processing units (CPUs), which execute the instructions of programs. Like other UNIX systems, Linux is a *preemptive multitasking* operating system. *Multitasking* means that multiple processes (i.e., running programs) can simultaneously reside in memory and each may receive use of the CPU(s). *Preemptive* means that the rules governing which processes receive use of the CPU and for how long are determined by the kernel process scheduler (rather than by the processes themselves).
- *Memory management*: While computer memories are enormous by the standards of a decade or two ago, the size of software has also correspondingly grown, so that physical memory (RAM) remains a limited resource that the kernel must share among processes in an equitable and efficient fashion. Like most modern operating systems, Linux employs virtual memory management (Section 6.4), a technique that confers two main advantages:
 - Processes are isolated from one another and from the kernel, so that one process can't read or modify the memory of another process or the kernel.
 - Only part of a process needs to be kept in memory, thereby lowering the memory requirements of each process and allowing more processes to be held in RAM simultaneously. This leads to better CPU utilization, since it increases the likelihood that, at any moment in time, there is at least one process that the CPU(s) can execute.
- *Provision of a file system*: The kernel provides a file system on disk, allowing files to be created, retrieved, updated, deleted, and so on.
- *Creation and termination of processes*: The kernel can load a new program into memory, providing it with the resources (e.g., CPU, memory, and access to files) that it needs in order to run. Such an instance of a running program is termed a *process*. Once a process has completed execution, the kernel ensures that the resources it uses are freed for subsequent reuse by later programs.
- *Access to devices*: The devices (mice, monitors, keyboards, disk and tape drives, and so on) attached to a computer allow communication of information between the computer and the outside world, permitting input, output, or both. The kernel provides programs with an interface that standardizes and simplifies access to devices, while at the same time arbitrating access by multiple processes to each device.

- *Networking*: The kernel transmits and receives network messages (packets) on behalf of user processes. This task includes routing of network packets to the target system.
- *Provision of a system call application programming interface (API)*: Processes can request the kernel to perform various tasks using kernel entry points known as *system calls*. The Linux system call API is the primary topic of this book. Section 3.1 details the steps that occur when a process performs a system call.

In addition to the above features, multiuser operating systems such as Linux generally provide users with the abstraction of a *virtual private computer*; that is, each user can log on to the system and operate largely independently of other users. For example, each user has their own disk storage space (home directory). In addition, users can run programs, each of which gets a share of the CPU and operates in its own virtual address space, and these programs can independently access devices and transfer information over the network. The kernel resolves potential conflicts in accessing hardware resources, so users and processes are generally unaware of the conflicts.

Kernel mode and user mode

Modern processor architectures typically allow the CPU to operate in at least two different modes: *user mode* and *kernel mode* (sometimes also referred to as *supervisor mode*). Hardware instructions allow switching from one mode to the other. Correspondingly, areas of virtual memory can be marked as being part of *user space* or *kernel space*. When running in user mode, the CPU can access only memory that is marked as being in user space; attempts to access memory in kernel space result in a hardware exception. When running in kernel mode, the CPU can access both user and kernel memory space.

Certain operations can be performed only while the processor is operating in kernel mode. Examples include executing the halt instruction to stop the system, accessing the memory-management hardware, and initiating device I/O operations. By taking advantage of this hardware design to place the operating system in kernel space, operating system implementers can ensure that user processes are not able to access the instructions and data structures of the kernel, or to perform operations that would adversely affect the operation of the system.

Process versus kernel views of the system

In many everyday programming tasks, we are accustomed to thinking about programming in a process-oriented way. However, when considering various topics covered later in this book, it can be useful to reorient our perspective to consider things from the kernel's point of view. To make the contrast clear, we now consider how things look first from a process viewpoint and then from a kernel viewpoint.

A running system typically has numerous processes. For a process, many things happen asynchronously. An executing process doesn't know when it will next time out, which other processes will then be scheduled for the CPU (and in what order), or when it will next be scheduled. The delivery of signals and the occurrence of interprocess communication events are mediated by the kernel, and can occur at any time for a process. Many things happen transparently for a process. A process

doesn't know where it is located in RAM or, in general, whether a particular part of its memory space is currently resident in memory or held in the swap area (a reserved area of disk space used to supplement the computer's RAM). Similarly, a process doesn't know where on the disk drive the files it accesses are being held; it simply refers to the files by name. A process operates in isolation; it can't directly communicate with another process. A process can't itself create a new process or even end its own existence. Finally, a process can't communicate directly with the input and output devices attached to the computer.

By contrast, a running system has one kernel that knows and controls everything. The kernel facilitates the running of all processes on the system. The kernel decides which process will next obtain access to the CPU, when it will do so, and for how long. The kernel maintains data structures containing information about all running processes and updates these structures as processes are created, change state, and terminate. The kernel maintains all of the low-level data structures that enable the filenames used by programs to be translated into physical locations on the disk. The kernel also maintains data structures that map the virtual memory of each process into the physical memory of the computer and the swap area(s) on disk. All communication between processes is done via mechanisms provided by the kernel. In response to requests from processes, the kernel creates new processes and terminates existing processes. Lastly, the kernel (in particular, device drivers) performs all direct communication with input and output devices, transferring information to and from user processes as required.

Later in this book we'll say things such as "a process can create another process," "a process can create a pipe," "a process can write data to a file," and "a process can terminate by calling *exit()*." Remember, however, that the kernel mediates all such actions, and these statements are just shorthand for "a process can *request that the kernel* create another process," and so on.

Further information

Modern texts covering operating systems concepts and design, with particular reference to UNIX systems, include [Tanenbaum, 2007], [Tanenbaum & Woodhull, 2006], and [Vahalia, 1996], the last of these containing much detail on virtual memory architectures. [Goodheart & Cox, 1994] provide details on System V Release 4. [Maxwell, 1999] provides an annotated listing of selected parts of the Linux 2.2.5 kernel. [Lions, 1996] is a detailed exposition of the Sixth Edition UNIX source code that remains a useful introduction to UNIX operating system internals. [Bovet & Cesati, 2005] describes the implementation of the Linux 2.6 kernel.

2.2 The Shell

A *shell* is a special-purpose program designed to read commands typed by a user and execute appropriate programs in response to those commands. Such a program is sometimes known as a *command interpreter*.

The term *login shell* is used to denote the process that is created to run a shell when the user first logs in.

Whereas on some operating systems the command interpreter is an integral part of the kernel, on UNIX systems, the shell is a user process. Many different shells exist, and different users (or, for that matter, a single user) on the same computer can simultaneously use different shells. A number of important shells have appeared over time:

- *Bourne shell (sh)*: This is the oldest of the widely used shells, and was written by Steve Bourne. It was the standard shell for Seventh Edition UNIX. The Bourne shell contains many of the features familiar in all shells: I/O redirection, pipelines, filename generation (globbing), variables, manipulation of environment variables, command substitution, background command execution, and functions. All later UNIX implementations include the Bourne shell in addition to any other shells they might provide.
- *C shell (csh)*: This shell was written by Bill Joy at the University of California at Berkeley. The name derives from the resemblance of many of the flow-control constructs of this shell to those of the C programming language. The C shell provided several useful interactive features unavailable in the Bourne shell, including command history, command-line editing, job control, and aliases. The C shell was not backward compatible with the Bourne shell. Although the standard interactive shell on BSD was the C shell, shell scripts (described in a moment) were usually written for the Bourne shell, so as to be portable across all UNIX implementations.
- *Korn shell (ksh)*: This shell was written as the successor to the Bourne shell by David Korn at AT&T Bell Laboratories. While maintaining backward compatibility with the Bourne shell, it also incorporated interactive features similar to those provided by the C shell.
- *Bourne again shell (bash)*: This shell is the GNU project's reimplementation of the Bourne shell. It supplies interactive features similar to those available in the C and Korn shells. The principal authors of *bash* are Brian Fox and Chet Ramey. Bash is probably the most widely used shell on Linux. (On Linux, the Bourne shell, *sh*, is actually provided by *bash* emulating *sh* as closely as possible.)

POSIX.2-1992 specified a standard for the shell that was based on the then current version of the Korn shell. Nowadays, the Korn shell and *bash* both conform to POSIX, but provide a number of extensions to the standard, and many of these extensions differ between the two shells.

The shells are designed not merely for interactive use, but also for the interpretation of *shell scripts*, which are text files containing shell commands. For this purpose, each of the shells has the facilities typically associated with programming languages: variables, loop and conditional statements, I/O commands, and functions.

Each of the shells performs similar tasks, albeit with variations in syntax. Unless referring to the operation of a specific shell, we typically refer to “the shell,” with the understanding that all shells operate in the manner described. Most of the examples in this book that require a shell use *bash*, but, unless otherwise noted, the reader can assume these examples work the same way in other Bourne-type shells.

2.3 Users and Groups

Each user on the system is uniquely identified, and users may belong to groups.

Users

Every user of the system has a unique *login name* (username) and a corresponding numeric *user ID* (UID). For each user, these are defined by a line in the system *password file*, `/etc/passwd`, which includes the following additional information:

- *Group ID*: the numeric group ID of the first of the groups of which the user is a member.
- *Home directory*: the initial directory into which the user is placed after logging in.
- *Login shell*: the name of the program to be executed to interpret user commands.

The password record may also include the user's password, in encrypted form. However, for security reasons, the password is often stored in the separate *shadow password file*, which is readable only by privileged users.

Groups

For administrative purposes—in particular, for controlling access to files and other system resources—it is useful to organize users into *groups*. For example, the people in a team working on a single project, and thus sharing a common set of files, might all be made members of the same group. In early UNIX implementations, a user could be a member of only one group. BSD allowed a user to simultaneously belong to multiple groups, an idea that was taken up by other UNIX implementations and the POSIX.1-1990 standard. Each group is identified by a single line in the system *group file*, `/etc/group`, which includes the following information:

- *Group name*: the (unique) name of the group.
- *Group ID* (GID): the numeric ID associated with this group.
- *User list*: a comma-separated list of login names of users who are members of this group (and who are not otherwise identified as members of the group by virtue of the group ID field of their password file record).

Superuser

One user, known as the *superuser*, has special privileges within the system. The superuser account has user ID 0, and normally has the login name *root*. On typical UNIX systems, the superuser bypasses all permission checks in the system. Thus, for example, the superuser can access any file in the system, regardless of the permissions on that file, and can send signals to any user process in the system. The system administrator uses the superuser account to perform various administrative tasks on the system.

2.4 Single Directory Hierarchy, Directories, Links, and Files

The kernel maintains a single hierarchical directory structure to organize all files in the system. (This contrasts with operating systems such as Microsoft Windows, where each disk device has its own directory hierarchy.) At the base of this hierarchy is the *root directory*, named / (slash). All files and directories are children or further removed descendants of the root directory. Figure 2-1 shows an example of this hierarchical file structure.

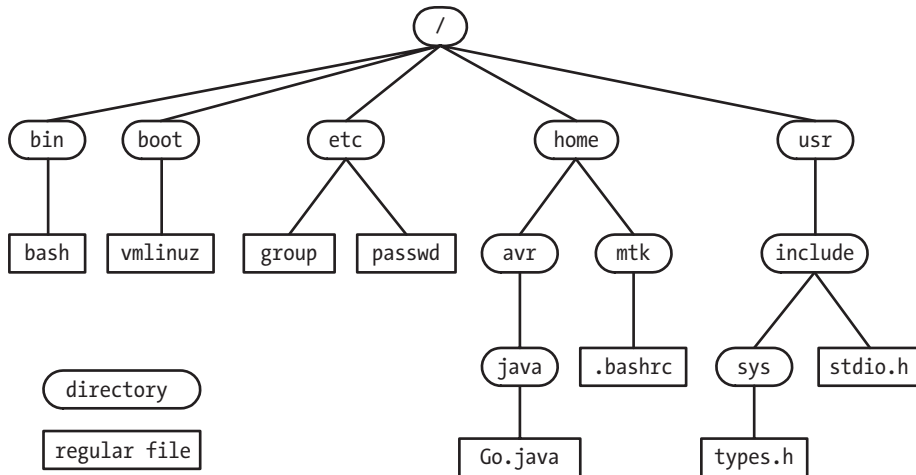


Figure 2-1: Subset of the Linux single directory hierarchy

File types

Within the file system, each file is marked with a *type*, indicating what kind of file it is. One of these file types denotes ordinary data files, which are usually called *regular* or *plain* files to distinguish them from other file types. These other file types include devices, pipes, sockets, directories, and symbolic links.

The term *file* is commonly used to denote a file of any type, not just a regular file.

Directories and links

A *directory* is a special file whose contents take the form of a table of filenames coupled with references to the corresponding files. This filename-plus-reference association is called a *link*, and files may have multiple links, and thus multiple names, in the same or in different directories.

Directories may contain links both to files and to other directories. The links between directories establish the directory hierarchy shown in Figure 2-1.

Every directory contains at least two entries: `.` (dot), which is a link to the directory itself, and `..` (dot-dot), which is a link to its *parent directory*, the directory above it in the hierarchy. Every directory, except the root directory, has a parent. For the root directory, the dot-dot entry is a link to the root directory itself (thus, `/..` equates to `/`).

Symbolic links

Like a normal link, a *symbolic link* provides an alternative name for a file. But whereas a normal link is a filename-plus-pointer entry in a directory list, a symbolic link is a specially marked file containing the name of another file. (In other words, a symbolic link has a filename-plus-pointer entry in a directory, and the file referred to by the pointer contains a string that names another file.) This latter file is often called the *target* of the symbolic link, and it is common to say that the symbolic link “points” or “refers” to the target file. When a pathname is specified in a system call, in most circumstances, the kernel automatically *dereferences* (or synonymously, *follows*) each symbolic link in the pathname, replacing it with the filename to which it points. This process may happen recursively if the target of a symbolic link is itself a symbolic link. (The kernel imposes limits on the number of dereferences to handle the possibility of circular chains of symbolic links.) If a symbolic link refers to a file that doesn’t exist, it is said to be a *dangling link*.

Often *hard link* and *soft link* are used as alternative terms for normal and symbolic links. The reasons for having two different types of links are explained in Chapter 18.

Filenames

On most Linux file systems, filenames can be up to 255 characters long. Filenames may contain any characters except slashes (/) and null characters (\0). However, it is advisable to employ only letters and digits, and the . (period), _ (underscore), and - (hyphen) characters. This 65-character set, [-._a-zA-Z0-9], is referred to in SUSv3 as the *portable filename character set*.

We should avoid the use of characters in filenames that are not in the portable filename character set because those characters may have special meanings within the shell, within regular expressions, or in other contexts. If a filename containing characters with special meanings appears in such contexts, then these characters must be *escaped*; that is, specially marked—typically with a preceding backslash (\)—to indicate that they should not be interpreted with those special meanings. In contexts where no escape mechanism is available, the filename is not usable.

We should also avoid filenames beginning with a hyphen (-), since such filenames may be mistaken for options when specified in a shell command.

Pathnames

A *pathname* is a string consisting of an optional initial slash (/) followed by a series of filenames separated by slashes. All but the last of these component filenames identifies a directory (or a symbolic link that resolves to a directory). The last component of a pathname may identify any type of file, including a directory. The series of component filenames preceding the final slash is sometimes referred to as the *directory* part of a pathname, while the name following the final slash is sometimes referred to as the file or *base* part of the pathname.

A pathname is read from left to right; each filename resides in the directory specified by the preceding part of the pathname. The string .. can be used anywhere in a pathname to refer to the parent of the location so far specified in the pathname.

A pathname describes the location of a file within the single directory hierarchy, and is either absolute or relative:

- An *absolute pathname* begins with a slash (/) and specifies the location of a file with respect to the root directory. Examples of absolute pathnames for files in Figure 2-1 are `/home/mtk/.bashrc`, `/usr/include`, and `/` (the pathname of the root directory).
- A *relative pathname* specifies the location of a file relative to a process's current working directory (see below), and is distinguished from an absolute pathname by the absence of an initial slash. In Figure 2-1, from the directory `usr`, the file `types.h` could be referenced using the relative pathname `include/sys/types.h`, while from the directory `avr`, the file `.bashrc` could be accessed using the relative pathname `../mtk/.bashrc`.

Current working directory

Each process has a *current working directory* (sometimes just referred to as the process's *working directory* or *current directory*). This is the process's "current location" within the single directory hierarchy, and it is from this directory that relative pathnames are interpreted for the process.

A process inherits its current working directory from its parent process. A login shell has its initial current working directory set to the location named in the home directory field of the user's password file entry. The shell's current working directory can be changed with the `cd` command.

File ownership and permissions

Each file has an associated user ID and group ID that define the owner of the file and the group to which it belongs. The ownership of a file is used to determine the access rights available to users of the file.

For the purpose of accessing a file, the system divides users into three categories: the *owner* of the file (sometimes termed the *user* of the file), users who are members of the group matching the file's group ID (*group*), and the rest of the world (*other*). Three permission bits may be set for each of these categories of user (making a total of nine permission bits): *read* permission allows the contents of the file to be read; *write* permission allows modification of the contents of the file; and *execute* permission allows execution of the file, which is either a program or a script to be processed by some interpreter (usually, but not always, one of the shells).

These permissions may also be set on directories, although their meanings are slightly different: *read* permission allows the contents of (i.e., the filenames in) the directory to be listed; *write* permission allows the contents of the directory to be changed (i.e., filenames can be added, removed, and changed); and *execute* (sometimes called *search*) permission allows access to files within the directory (subject to the permissions on the files themselves).

2.5 File I/O Model

One of the distinguishing features of the I/O model on UNIX systems is the concept of *universality of I/O*. This means that the same system calls (`open()`, `read()`,

write(), *close()*, and so on) are used to perform I/O on all types of files, including devices. (The kernel translates the application's I/O requests into appropriate file-system or device-driver operations that perform I/O on the target file or device.) Thus, a program employing these system calls will work on any type of file.

The kernel essentially provides one file type: a sequential stream of bytes, which, in the case of disk files, disks, and tape devices, can be randomly accessed using the *lseek()* system call.

Many applications and libraries interpret the *newline* character (ASCII code 10 decimal, sometimes also known as *linefeed*) as terminating one line of text and commencing another. UNIX systems have no *end-of-file* character; the end of a file is detected by a read that returns no data.

File descriptors

The I/O system calls refer to open files using a *file descriptor*, a (usually small) non-negative integer. A file descriptor is typically obtained by a call to *open()*, which takes a pathname argument specifying a file upon which I/O is to be performed.

Normally, a process inherits three open file descriptors when it is started by the shell: descriptor 0 is *standard input*, the file from which the process takes its input; descriptor 1 is *standard output*, the file to which the process writes its output; and descriptor 2 is *standard error*, the file to which the process writes error messages and notification of exceptional or abnormal conditions. In an interactive shell or program, these three descriptors are normally connected to the terminal. In the *stdio* library, these descriptors correspond to the file streams *stdin*, *stdout*, and *stderr*.

The *stdio* library

To perform file I/O, C programs typically employ I/O functions contained in the standard C library. This set of functions, referred to as the *stdio* library, includes *fopen()*, *fclose()*, *scanf()*, *printf()*, *fgets()*, *fputs()*, and so on. The *stdio* functions are layered on top of the I/O system calls (*open()*, *close()*, *read()*, *write()*, and so on).

We assume that the reader is already familiar with the C standard I/O (*stdio*) functions, and don't cover them in this book. Further information on the *stdio* library can be found in [Kernighan & Ritchie, 1988], [Harbison & Steele, 2002], [Plauger, 1992], and [Stevens & Rago, 2005].

2.6 Programs

Programs normally exist in two forms. The first form is *source code*, human-readable text consisting of a series of statements written in a programming language such as C. To be executed, source code must be converted to the second form: binary machine-language instructions that the computer can understand. (This contrasts with a *script*, which is a text file containing commands to be directly processed by a program such as a shell or other command interpreter.) The two meanings of the term *program* are normally considered synonymous, since the step of compiling and linking converts source code into semantically equivalent binary machine code.

Filters

A *filter* is the name often applied to a program that reads its input from *stdin*, performs some transformation of that input, and writes the transformed data to *stdout*. Examples of filters include *cat*, *grep*, *tr*, *sort*, *wc*, *sed*, and *awk*.

Command-line arguments

In C, programs can access the *command-line arguments*, the words that are supplied on the command line when the program is run. To access the command-line arguments, the *main()* function of the program is declared as follows:

```
int main(int argc, char *argv[])
```

The *argc* variable contains the total number of command-line arguments, and the individual arguments are available as strings pointed to by members of the array *argv*. The first of these strings, *argv[0]*, identifies the name of the program itself.

2.7 Processes

Put most simply, a *process* is an instance of an executing program. When a program is executed, the kernel loads the code of the program into virtual memory, allocates space for program variables, and sets up kernel bookkeeping data structures to record various information (such as process ID, termination status, user IDs, and group IDs) about the process.

From a kernel point of view, processes are the entities among which the kernel must share the various resources of the computer. For resources that are limited, such as memory, the kernel initially allocates some amount of the resource to the process, and adjusts this allocation over the lifetime of the process in response to the demands of the process and the overall system demand for that resource. When the process terminates, all such resources are released for reuse by other processes. Other resources, such as the CPU and network bandwidth, are renewable, but must be shared equitably among all processes.

Process memory layout

A process is logically divided into the following parts, known as *segments*:

- *Text*: the instructions of the program.
- *Data*: the static variables used by the program.
- *Heap*: an area from which programs can dynamically allocate extra memory.
- *Stack*: a piece of memory that grows and shrinks as functions are called and return and that is used to allocate storage for local variables and function call linkage information.

Process creation and program execution

A process can create a new process using the *fork()* system call. The process that calls *fork()* is referred to as the *parent process*, and the new process is referred to as the *child process*. The kernel creates the child process by making a duplicate of the

parent process. The child inherits copies of the parent's data, stack, and heap segments, which it may then modify independently of the parent's copies. (The program text, which is placed in memory marked as read-only, is shared by the two processes.)

The child process goes on either to execute a different set of functions in the same code as the parent, or, frequently, to use the *execve()* system call to load and execute an entirely new program. An *execve()* call destroys the existing text, data, stack, and heap segments, replacing them with new segments based on the code of the new program.

Several related C library functions are layered on top of *execve()*, each providing a slightly different interface to the same functionality. All of these functions have names starting with the string *exec*, and where the differences don't matter, we'll use the notation *exec()* to refer generally to these functions. Be aware, however, that there is no actual function with the name *exec()*.

Commonly, we'll use the verb *to exec* to describe the operation performed *execve()* and the library functions layered on top of it.

Process ID and parent process ID

Each process has a unique integer *process identifier* (PID). Each process also has a *parent process identifier* (PPID) attribute, which identifies the process that requested the kernel to create this process.

Process termination and termination status

A process can terminate in one of two ways: by requesting its own termination using the *_exit()* system call (or the related *exit()* library function), or by being killed by the delivery of a signal. In either case, the process yields a *termination status*, a small nonnegative integer value that is available for inspection by the parent process using the *wait()* system call. In the case of a call to *_exit()*, the process explicitly specifies its own termination status. If a process is killed by a signal, the termination status is set according to the type of signal that caused the death of the process. (Sometimes, we'll refer to the argument passed to *_exit()* as the *exit status* of the process, as distinct from the termination status, which is either the value passed to *_exit()* or an indication of the signal that killed the process.)

By convention, a termination status of 0 indicates that the process succeeded, and a nonzero status indicates that some error occurred. Most shells make the termination status of the last executed program available via a shell variable named *\$?*.

Process user and group identifiers (credentials)

Each process has a number of associated user IDs (UIDs) and group IDs (GIDs). These include:

- *Real user ID* and *real group ID*: These identify the user and group to which the process belongs. A new process inherits these IDs from its parent. A login shell gets its real user ID and real group ID from the corresponding fields in the system password file.

- *Effective user ID* and *effective group ID*: These two IDs (in conjunction with the supplementary group IDs discussed in a moment) are used in determining the permissions that the process has when accessing protected resources such as files and interprocess communication objects. Typically, the process's effective IDs have the same values as the corresponding real IDs. Changing the effective IDs is a mechanism that allows a process to assume the privileges of another user or group, as described in a moment.
- *Supplementary group IDs*: These IDs identify additional groups to which a process belongs. A new process inherits its supplementary group IDs from its parent. A login shell gets its supplementary group IDs from the system group file.

Privileged processes

Traditionally, on UNIX systems, a *privileged process* is one whose *effective* user ID is 0 (superuser). Such a process bypasses the permission restrictions normally applied by the kernel. By contrast, the term *unprivileged* (or *nonprivileged*) is applied to processes run by other users. Such processes have a nonzero effective user ID and must abide by the permission rules enforced by the kernel.

A process may be privileged because it was created by another privileged process—for example, by a login shell started by *root* (superuser). Another way a process may become privileged is via the set-user-ID mechanism, which allows a process to assume an effective user ID that is the same as the user ID of the program file that it is executing.

Capabilities

Since kernel 2.2, Linux divides the privileges traditionally accorded to the superuser into a set of distinct units called *capabilities*. Each privileged operation is associated with a particular capability, and a process can perform an operation only if it has the corresponding capability. A traditional superuser process (effective user ID of 0) corresponds to a process with all capabilities enabled.

Granting a subset of capabilities to a process allows it to perform some of the operations normally permitted to the superuser, while preventing it from performing others.

Capabilities are described in detail in Chapter 39. In the remainder of the book, when noting that a particular operation can be performed only by a privileged process, we'll usually identify the specific capability in parentheses. Capability names begin with the prefix *CAP_*, as in *CAP_KILL*.

The *init* process

When booting the system, the kernel creates a special process called *init*, the “parent of all processes,” which is derived from the program file */sbin/init*. All processes on the system are created (using *fork()*) either by *init* or by one of its descendants. The *init* process always has the process ID 1 and runs with superuser privileges. The *init* process can't be killed (not even by the superuser), and it terminates only when the system is shut down. The main task of *init* is to create and monitor a range of processes required by a running system. (For details, see the *init(8)* manual page.)

Daemon processes

A *daemon* is a special-purpose process that is created and handled by the system in the same way as other processes, but which is distinguished by the following characteristics:

- It is long-lived. A daemon process is often started at system boot and remains in existence until the system is shut down.
- It runs in the background, and has no controlling terminal from which it can read input or to which it can write output.

Examples of daemon processes include *syslogd*, which records messages in the system log, and *httpd*, which serves web pages via the Hypertext Transfer Protocol (HTTP).

Environment list

Each process has an *environment list*, which is a set of *environment variables* that are maintained within the user-space memory of the process. Each element of this list consists of a name and an associated value. When a new process is created via *fork()*, it inherits a copy of its parent's environment. Thus, the environment provides a mechanism for a parent process to communicate information to a child process. When a process replaces the program that it is running using *exec()*, the new program either inherits the environment used by the old program or receives a new environment specified as part of the *exec()* call.

Environment variables are created with the *export* command in most shells (or the *setenv* command in the C shell), as in the following example:

```
$ export MYVAR='Hello world'
```

Whenever we present a shell session log showing interactive input and output, the input text is always boldfaced. Sometimes, we include commentary in the log in italic text, adding notes about the commands entered or the output produced.

C programs can access the environment using an external variable (*char **environ*), and various library functions allow a process to retrieve and modify values in its environment.

Environment variables are used for a variety of purposes. For example, the shell defines and uses a range of variables that can be accessed by scripts and programs executed from the shell. These include the variable *HOME*, which specifies the pathname of the user's login directory, and the variable *PATH*, which specifies a list of directories that the shell should search when looking for programs corresponding to commands entered by the user.

Resource limits

Each process consumes resources, such as open files, memory, and CPU time. Using the *setrlimit()* system call, a process can establish upper limits on its consumption of various resources. Each such *resource limit* has two associated values: a *soft limit*, which limits the amount of the resource that the process may consume; and a

hard limit, which is a ceiling on the value to which the soft limit may be adjusted. An unprivileged process may change its soft limit for a particular resource to any value in the range from zero up to the corresponding hard limit, but can only lower its hard limit.

When a new process is created with *fork()*, it inherits copies of its parent's resource limit settings.

The resource limits of the shell can be adjusted using the *ulimit* command (*limit* in the C shell). These limit settings are inherited by the child processes that the shell creates to execute commands.

2.8 Memory Mappings

The *mmap()* system call creates a new *memory mapping* in the calling process's virtual address space.

Mappings fall into two categories:

- A *file mapping* maps a region of a file into the calling process's virtual memory. Once mapped, the file's contents can be accessed by operations on the bytes in the corresponding memory region. The pages of the mapping are automatically loaded from the file as required.
- By contrast, an *anonymous mapping* doesn't have a corresponding file. Instead, the pages of the mapping are initialized to 0.

The memory in one process's mapping may be shared with mappings in other processes. This can occur either because two processes map the same region of a file or because a child process created by *fork()* inherits a mapping from its parent.

When two or more processes share the same pages, each process may see the changes made by other processes to the contents of the pages, depending on whether the mapping is created as private or shared. When a mapping is *private*, modifications to the contents of the mapping are not visible to other processes and are not carried through to the underlying file. When a mapping is *shared*, modifications to the contents of the mapping are visible to other processes sharing the same mapping and are carried through to the underlying file.

Memory mappings serve a variety of purposes, including initialization of a process's text segment from the corresponding segment of an executable file, allocation of new (zero-filled) memory, file I/O (memory-mapped I/O), and inter-process communication (via a shared mapping).

2.9 Static and Shared Libraries

An *object library* is a file containing the compiled object code for a (usually logically related) set of functions that may be called from application programs. Placing code for a set of functions in a single object library eases the tasks of program creation and maintenance. Modern UNIX systems provide two types of object libraries: *static libraries* and *shared libraries*.

Static libraries

Static libraries (sometimes also known as *archives*) were the only type of library on early UNIX systems. A static library is essentially a structured bundle of compiled object modules. To use functions from a static library, we specify that library in the link command used to build a program. After resolving the various function references from the main program to the modules in the static library, the linker extracts copies of the required object modules from the library and copies these into the resulting executable file. We say that such a program is *statically linked*.

The fact that each statically linked program includes its own copy of the object modules required from the library creates a number of disadvantages. One is that the duplication of object code in different executable files wastes disk space. A corresponding waste of memory occurs when statically linked programs using the same library function are executed at the same time; each program requires its own copy of the function to reside in memory. Additionally, if a library function requires modification, then, after recompiling that function and adding it to the static library, all applications that need to use the updated function must be relinked against the library.

Shared libraries

Shared libraries were designed to address the problems with static libraries.

If a program is linked against a shared library, then, instead of copying object modules from the library into the executable, the linker just writes a record into the executable to indicate that at run time the executable needs to use that shared library. When the executable is loaded into memory at run time, a program called the *dynamic linker* ensures that the shared libraries required by the executable are found and loaded into memory, and performs run-time linking to resolve the function calls in the executable to the corresponding definitions in the shared libraries. At run time, only a single copy of the code of the shared library needs to be resident in memory; all running programs can use that copy.

The fact that a shared library contains the sole compiled version of a function saves disk space. It also greatly eases the job of ensuring that programs employ the newest version of a function. Simply rebuilding the shared library with the new function definition causes existing programs to automatically use the new definition when they are next executed.

2.10 Interprocess Communication and Synchronization

A running Linux system consists of numerous processes, many of which operate independently of each other. Some processes, however, cooperate to achieve their intended purposes, and these processes need methods of communicating with one another and synchronizing their actions.

One way for processes to communicate is by reading and writing information in disk files. However, for many applications, this is too slow and inflexible.

Therefore, Linux, like all modern UNIX implementations, provides a rich set of mechanisms for *interprocess communication* (IPC), including the following:

- *signals*, which are used to indicate that an event has occurred;
- *pipes* (familiar to shell users as the `|` operator) and *FIFOs*, which can be used to transfer data between processes;
- *sockets*, which can be used to transfer data from one process to another, either on the same host computer or on different hosts connected by a network;
- *file locking*, which allows a process to lock regions of a file in order to prevent other processes from reading or updating the file contents;
- *message queues*, which are used to exchange messages (packets of data) between processes;
- *semaphores*, which are used to synchronize the actions of processes; and
- *shared memory*, which allows two or more processes to share a piece of memory. When one process changes the contents of the shared memory, all of the other processes can immediately see the changes.

The wide variety of IPC mechanisms on UNIX systems, with sometimes overlapping functionality, is in part due to their evolution under different variants of the UNIX system and the requirements of various standards. For example, FIFOs and UNIX domain sockets essentially perform the same function of allowing unrelated processes on the same system to exchange data. Both exist in modern UNIX systems because FIFOs came from System V, while sockets came from BSD.

2.11 Signals

Although we listed them as a method of IPC in the previous section, signals are more usually employed in a wide range of other contexts, and so deserve a longer discussion.

Signals are often described as “software interrupts.” The arrival of a signal informs a process that some event or exceptional condition has occurred. There are various types of signals, each of which identifies a different event or condition. Each signal type is identified by a different integer, defined with symbolic names of the form `SIGxxx`.

Signals are sent to a process by the kernel, by another process (with suitable permissions), or by the process itself. For example, the kernel may send a signal to a process when one of the following occurs:

- the user typed the *interrupt* character (usually *Control-C*) on the keyboard;
- one of the process’s children has terminated;
- a timer (alarm clock) set by the process has expired; or
- the process attempted to access an invalid memory address.

Within the shell, the *kill* command can be used to send a signal to a process. The *kill()* system call provides the same facility within programs.

When a process receives a signal, it takes one of the following actions, depending on the signal:

- it ignores the signal;
- it is killed by the signal; or
- it is suspended until later being resumed by receipt of a special-purpose signal.

For most signal types, instead of accepting the default signal action, a program can choose to ignore the signal (useful if the default action for the signal is something other than being ignored), or to establish a *signal handler*. A signal handler is a programmer-defined function that is automatically invoked when the signal is delivered to the process. This function performs some action appropriate to the condition that generated the signal.

In the interval between the time it is generated and the time it is delivered, a signal is said to be *pending* for a process. Normally, a pending signal is delivered as soon as the receiving process is next scheduled to run, or immediately if the process is already running. However, it is also possible to *block* a signal by adding it to the process's *signal mask*. If a signal is generated while it is blocked, it remains pending until it is later unblocked (i.e., removed from the signal mask).

2.12 Threads

In modern UNIX implementations, each process can have multiple *threads* of execution. One way of envisaging threads is as a set of processes that share the same virtual memory, as well as a range of other attributes. Each thread is executing the same program code and shares the same data area and heap. However, each thread has its own stack containing local variables and function call linkage information.

Threads can communicate with each other via the global variables that they share. The threading API provides *condition variables* and *mutexes*, which are primitives that enable the threads of a process to communicate and synchronize their actions, in particular, their use of shared variables. Threads can also communicate with one another using the IPC and synchronization mechanisms described in Section 2.10.

The primary advantages of using threads are that they make it easy to share data (via global variables) between cooperating threads and that some algorithms transpose more naturally to a multithreaded implementation than to a multiprocess implementation. Furthermore, a multithreaded application can transparently take advantage of the possibilities for parallel processing on multiprocessor hardware.

2.13 Process Groups and Shell Job Control

Each program executed by the shell is started in a new process. For example, the shell creates three processes to execute the following pipeline of commands (which displays a list of files in the current working directory sorted by file size):

```
$ ls -l | sort -k5n | less
```

All major shells, except the Bourne shell, provide an interactive feature called *job control*, which allows the user to simultaneously execute and manipulate multiple commands or pipelines. In job-control shells, all of the processes in a pipeline are placed in a new *process group* or *job*. (In the simple case of a shell command line containing a single command, a new process group containing just a single process is created.) Each process in a process group has the same integer *process group identifier*, which is the same as the process ID of one of the processes in the group, termed the *process group leader*.

The kernel allows for various actions, notably the delivery of signals, to be performed on all members of a process group. Job-control shells use this feature to allow the user to suspend or resume all of the processes in a pipeline, as described in the next section.

2.14 Sessions, Controlling Terminals, and Controlling Processes

A *session* is a collection of process groups (jobs). All of the processes in a session have the same *session identifier*. A *session leader* is the process that created the session, and its process ID becomes the session ID.

Sessions are used mainly by job-control shells. All of the process groups created by a job-control shell belong to the same session as the shell, which is the session leader.

Sessions usually have an associated *controlling terminal*. The controlling terminal is established when the session leader process first opens a terminal device. For a session created by an interactive shell, this is the terminal at which the user logged in. A terminal may be the controlling terminal of at most one session.

As a consequence of opening the controlling terminal, the session leader becomes the *controlling process* for the terminal. The controlling process receives a SIGHUP signal if a terminal disconnect occurs (e.g., if the terminal window is closed).

At any point in time, one process group in a session is the *foreground process group* (*foreground job*), which may read input from the terminal and send output to it. If the user types the *interrupt* character (usually *Control-C*) or the *suspend* character (usually *Control-Z*) on the controlling terminal, then the terminal driver sends a signal that kills or suspends (i.e., stops) the foreground process group. A session can have any number of *background process groups* (*background jobs*), which are created by terminating a command with the ampersand (&) character.

Job-control shells provide commands for listing all jobs, sending signals to jobs, and moving jobs between the foreground and background.

2.15 Pseudoterminals

A *pseudoterminal* is a pair of connected virtual devices, known as the *master* and *slave*. This device pair provides an IPC channel allowing data to be transferred in both directions between the two devices.

The key point about a pseudoterminal is that the slave device provides an interface that behaves like a terminal, which makes it possible to connect a terminal-oriented program to the slave device and then use another program connected to the master device to drive the terminal-oriented program. Output written by the driver

program undergoes the usual input processing performed by the terminal driver (for example, in the default mode, a carriage return is mapped to a newline) and is then passed as input to the terminal-oriented program connected to the slave. Anything that the terminal-oriented program writes to the slave is passed (after performing all of the usual terminal output processing) as input to the driver program. In other words, the driver program is performing the function normally performed by the user at a conventional terminal.

Pseudoterminals are used in a variety of applications, most notably in the implementation of terminal windows provided under an X Window System login and in applications providing network login services, such as *telnet* and *ssh*.

2.16 Date and Time

Two types of time are of interest to a process:

- *Real time* is measured either from some standard point (*calendar time*) or from some fixed point, typically the start, in the life of a process (*elapsed* or *wall clock time*). On UNIX systems, calendar time is measured in seconds since midnight on the morning of January 1, 1970, Universal Coordinated Time (usually abbreviated UTC), and coordinated on the base point for timezones defined by the longitudinal line passing through Greenwich, England. This date, which is close to the birth of the UNIX system, is referred to as the *Epoch*.
- *Process time*, also called *CPU time*, is the total amount of CPU time that a process has used since starting. CPU time is further divided into *system CPU time*, the time spent executing code in *kernel mode* (i.e., executing system calls and performing other kernel services on behalf of the process), and *user CPU time*, the time spent executing code in *user mode* (i.e., executing normal program code).

The *time* command displays the real time, the system CPU time, and user CPU time taken to execute the processes in a pipeline.

2.17 Client-Server Architecture

At various points in this book, we discuss the design and implementation of client-server applications.

A *client-server application* is one that is broken into two component processes:

- a *client*, which asks the server to carry out some *service* by sending it a request message; and
- a *server*, which examines the client's request, performs appropriate actions, and then sends a response message back to the client.

Sometimes, the client and server may engage in an extended dialogue of requests and responses.

Typically, the client application interacts with a user, while the server application provides access to some shared resource. Commonly, there are multiple instances of client processes communicating with one or a few instances of the server process.

The client and server may reside on the same host computer or on separate hosts connected via a network. To communicate with one another, the client and server use the IPC mechanisms discussed in Section 2.10.

Servers may implement a variety of services, such as:

- providing access to a database or other shared information resource;
- providing access to a remote file across a network;
- encapsulating some business logic;
- providing access to a shared hardware resource (e.g., a printer); or
- serving web pages.

Encapsulating a service within a single server is useful for a number of reasons, such as the following:

- *Efficiency*: It may be cheaper to provide one instance of a resource (e.g., a printer) that is managed by a server than to provide the same resource locally on every computer.
- *Control, coordination, and security*: By holding a resource (especially an information resource) at a single location, the server can coordinate access to the resource (e.g., so that two clients don't simultaneously update the same piece of information) or secure it so that it is made available to only selected clients.
- *Operation in a heterogeneous environment*: In a network, the various clients, and the server, can be running on different hardware and operating system platforms.

2.18 Realtime

Realtime applications are those that need to respond in a timely fashion to input. Frequently, such input comes from an external sensor or a specialized input device, and output takes the form of controlling some external hardware. Examples of applications with realtime response requirements include automated assembly lines, bank ATMs, and aircraft navigation systems.

Although many realtime applications require rapid responses to input, the defining factor is that the response is guaranteed to be delivered within a certain deadline time after the triggering event.

The provision of realtime responsiveness, especially where short response times are demanded, requires support from the underlying operating system. Most operating systems don't natively provide such support because the requirements of realtime responsiveness can conflict with the requirements of multiuser time-sharing operating systems. Traditional UNIX implementations are not realtime operating systems, although realtime variants have been devised. Realtime variants of Linux have also been created, and recent Linux kernels are moving toward full native support for realtime applications.

POSIX.1b defined a number of extensions to POSIX.1 for the support of realtime applications. These include asynchronous I/O, shared memory, memory-mapped files, memory locking, realtime clocks and timers, alternative scheduling policies, realtime signals, message queues, and semaphores. Even though they

don't strictly qualify as realtime, most UNIX implementations now support some or all of these extensions. (During the course of this book, we describe those features of POSIX.1b that are supported by Linux.)

In this book, we use the term *real time* to refer to the concept of calendar or elapsed time, and the term *realtime* to denote an operating system or application providing the type of responsiveness described in this section.

2.19 The /proc File System

Like several other UNIX implementations, Linux provides a /proc file system, which consists of a set of directories and files mounted under the /proc directory.

The /proc file system is a virtual file system that provides an interface to kernel data structures in a form that looks like files and directories on a file system. This provides an easy mechanism for viewing and changing various system attributes. In addition, a set of directories with names of the form /proc/*PID*, where *PID* is a process ID, allows us to view information about each process running on the system.

The contents of /proc files are generally in human-readable text form and can be parsed by shell scripts. A program can simply open and read from, or write to, the desired file. In most cases, a process must be privileged to modify the contents of files in the /proc directory.

As we describe various parts of the Linux programming interface, we'll also describe the relevant /proc files. Section 12.1 provides further general information on this file system. The /proc file system is not specified by any standards, and the details that we describe are Linux-specific.

2.20 Summary

In this chapter, we surveyed a range of fundamental concepts related to Linux system programming. An understanding of these concepts should provide readers with limited experience on Linux or UNIX with enough background to begin learning system programming.

3

SYSTEM PROGRAMMING CONCEPTS

This chapter covers various topics that are prerequisites for system programming. We begin by introducing system calls and detailing the steps that occur during their execution. We then consider library functions and how they differ from system calls, and couple this with a description of the (GNU) C library.

Whenever we make a system call or call a library function, we should always check the return status of the call in order to determine if it was successful. We describe how to perform such checks, and present a set of functions that are used in most of the example programs in this book to diagnose errors from system calls and library functions.

We conclude by looking at various issues related to portable programming, specifically the use of feature test macros and the standard system data types defined by SUSv3.

3.1 System Calls

A *system call* is a controlled entry point into the kernel, allowing a process to request that the kernel perform some action on the process's behalf. The kernel makes a range of services accessible to programs via the system call application programming interface (API). These services include, for example, creating a

new process, performing I/O, and creating a pipe for interprocess communication. (The *syscalls(2)* manual page lists the Linux system calls.)

Before going into the details of how a system call works, we note some general points:

- A system call changes the processor state from user mode to kernel mode, so that the CPU can access protected kernel memory.
- The set of system calls is fixed. Each system call is identified by a unique number. (This numbering scheme is not normally visible to programs, which identify system calls by name.)
- Each system call may have a set of arguments that specify information to be transferred from user space (i.e., the process's virtual address space) to kernel space and vice versa.

From a programming point of view, invoking a system call looks much like calling a C function. However, behind the scenes, many steps occur during the execution of a system call. To illustrate this, we consider the steps in the order that they occur on a specific hardware implementation, the x86-32. The steps are as follows:

1. The application program makes a system call by invoking a wrapper function in the C library.
2. The wrapper function must make all of the system call arguments available to the system call trap-handling routine (described shortly). These arguments are passed to the wrapper via the stack, but the kernel expects them in specific registers. The wrapper function copies the arguments to these registers.
3. Since all system calls enter the kernel in the same way, the kernel needs some method of identifying the system call. To permit this, the wrapper function copies the system call number into a specific CPU register (%eax).
4. The wrapper function executes a *trap* machine instruction (int 0x80), which causes the processor to switch from user mode to kernel mode and execute code pointed to by location 0x80 (128 decimal) of the system's trap vector.

More recent x86-32 architectures implement the *sysenter* instruction, which provides a faster method of entering kernel mode than the conventional int 0x80 trap instruction. The use of *sysenter* is supported in the 2.6 kernel and from *glibc* 2.3.2 onward.

5. In response to the trap to location 0x80, the kernel invokes its *system_call()* routine (located in the assembler file *arch/i386/entry.S*) to handle the trap. This handler:
 - a) Saves register values onto the kernel stack (Section 6.5).
 - b) Checks the validity of the system call number.
 - c) Invokes the appropriate system call service routine, which is found by using the system call number to index a table of all system call service routines (the kernel variable *sys_call_table*). If the system call service routine has any arguments, it first checks their validity; for example, it checks that addresses point to valid locations in user memory. Then the service

routine performs the required task, which may involve modifying values at addresses specified in the given arguments and transferring data between user memory and kernel memory (e.g., in I/O operations). Finally, the service routine returns a result status to the *system_call()* routine.

- d) Restores register values from the kernel stack and places the system call return value on the stack.
 - e) Returns to the wrapper function, simultaneously returning the processor to user mode.
6. If the return value of the system call service routine indicated an error, the wrapper function sets the global variable *errno* (see Section 3.4) using this value. The wrapper function then returns to the caller, providing an integer return value indicating the success or failure of the system call.

On Linux, system call service routines follow a convention of returning a nonnegative value to indicate success. In case of an error, the routine returns a negative number, which is the negated value of one of the *errno* constants. When a negative value is returned, the C library wrapper function negates it (to make it positive), copies the result into *errno*, and returns -1 as the function result of the wrapper to indicate an error to the calling program.

This convention relies on the assumption that system call service routines don't return negative values on success. However, for a few of these routines, this assumption doesn't hold. Normally, this is not a problem, since the range of negated *errno* values doesn't overlap with valid negative return values. However, this convention does cause a problem in one case: the *F_GETOWN* operation of the *fcntl()* system call, which we describe in Section 63.3.

Figure 3-1 illustrates the above sequence using the example of the *execve()* system call. On Linux/x86-32, *execve()* is system call number 11 (*_NR_execve*). Thus, in the *sys_call_table* vector, entry 11 contains the address of *sys_execve()*, the service routine for this system call. (On Linux, system call service routines typically have names of the form *sys_xyz()*, where *xyz()* is the system call in question.)

The information given in the preceding paragraphs is more than we'll usually need to know for the remainder of this book. However, it illustrates the important point that, even for a simple system call, quite a bit of work must be done, and thus system calls have a small but appreciable overhead.

As an example of the overhead of making a system call, consider the *getppid()* system call, which simply returns the process ID of the parent of the calling process. On one of the author's x86-32 systems running Linux 2.6.25, 10 million calls to *getppid()* required approximately 2.2 seconds to complete. This amounts to around 0.3 microseconds per call. By comparison, on the same system, 10 million calls to a C function that simply returns an integer required 0.11 seconds, or around one-twentieth of the time required for calls to *getppid()*. Of course, most system calls have significantly more overhead than *getppid()*.

Since, from the point of view of a C program, calling the C library wrapper function is synonymous with invoking the corresponding system call service routine, in the remainder of this book, we use wording such as "invoking the system call *xyz()*" to mean "calling the wrapper function that invokes the system call *xyz()*."

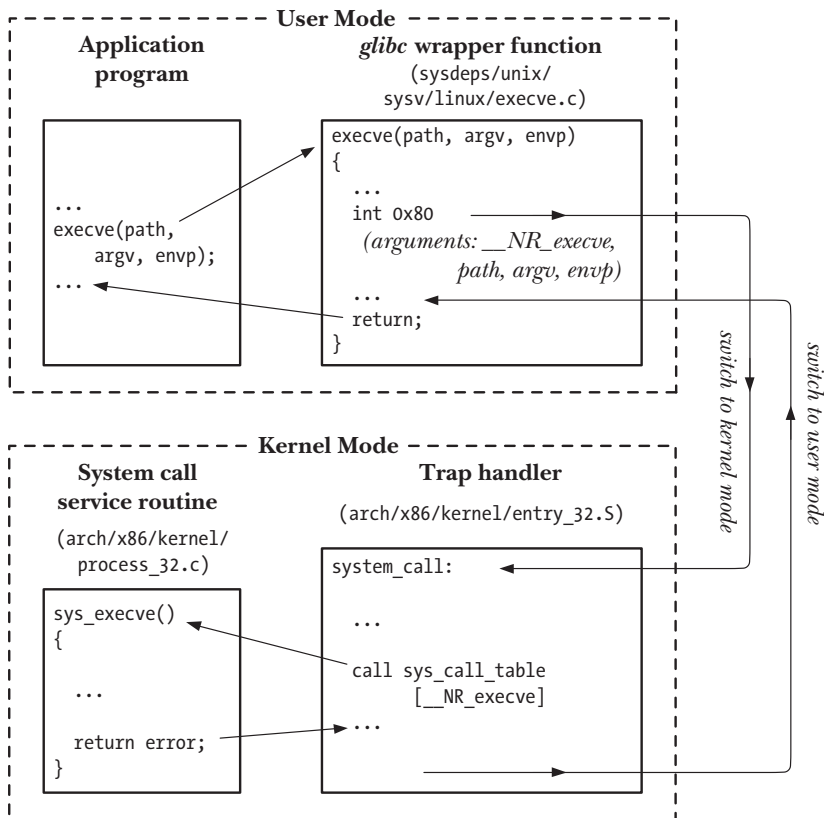


Figure 3-1: Steps in the execution of a system call

Appendix A describes the *strace* command, which can be used to trace the system calls made by a program, either for debugging purposes or simply to investigate what a program is doing.

More information about the Linux system call mechanism can be found in [Love, 2010], [Bovet & Cesati, 2005], and [Maxwell, 1999].

3.2 Library Functions

A *library function* is simply one of the multitude of functions that constitutes the standard C library. (For brevity, when talking about a specific function in the rest of the book we'll often just write *function* rather than *library function*.) The purposes of these functions are very diverse, including such tasks as opening a file, converting a time to a human-readable format, and comparing two character strings.

Many library functions don't make any use of system calls (e.g., the string-manipulation functions). On the other hand, some library functions are layered on top of system calls. For example, the *fopen()* library function uses the *open()* system call to actually open a file. Often, library functions are designed to provide a more caller-friendly interface than the underlying system call. For example, the *printf()* function provides output formatting and data buffering, whereas the *write()* system

call just outputs a block of bytes. Similarly, the *malloc()* and *free()* functions perform various bookkeeping tasks that make them a much easier way to allocate and free memory than the underlying *brk()* system call.

3.3 The Standard C Library; The GNU C Library (*glibc*)

There are different implementations of the standard C library on the various UNIX implementations. The most commonly used implementation on Linux is the GNU C library (*glibc*, <http://www.gnu.org/software/libc/>).

The principal developer and maintainer of the GNU C library was initially Roland McGrath. Nowadays, this task is carried out by Ulrich Drepper.

Various other C libraries are available for Linux, including libraries with smaller memory requirements for use in embedded device applications. Examples include *uClibc* (<http://www.uclibc.org/>) and *diet libc* (<http://www.fefe.de/dietlibc/>). In this book, we confine the discussion to *glibc*, since that is the C library used by most applications developed on Linux.

Determining the version of *glibc* on the system

Sometimes, we need to determine the version of *glibc* on a system. From the shell, we can do this by running the *glibc* shared library file as though it were an executable program. When we run the library as an executable, it displays various text, including its version number:

```
$ /lib/libc.so.6
GNU C Library stable release version 2.10.1, by Roland McGrath et al.
Copyright (C) 2009 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 4.4.0 20090506 (Red Hat 4.4.0-4).
Compiled on a Linux >>2.6.18-128.4.1.el5<< system on 2009-08-19.
Available extensions:
  The C stubs add-on version 2.1.2.
  crypt add-on version 2.1 by Michael Glad and others
  GNU Libidn by Simon Josefsson
  Native POSIX Threads Library by Ulrich Drepper et al
  BIND-8.2.3-T5B
  RT using linux kernel aio
For bug reporting instructions, please see:
<http://www.gnu.org/software/libc/bugs.html>.
```

In some Linux distributions, the GNU C library resides at a pathname other than */lib/libc.so.6*. One way of determining the location of the library is to run the *ldd* (list dynamic dependencies) program against an executable linked dynamically against *glibc* (most executables are linked in this manner). We can then inspect the resulting library dependency list to find the location of the *glibc* shared library:

```
$ ldd myprog | grep libc
libc.so.6 => /lib/tls/libc.so.6 (0x4004b000)
```

There are two means by which an application program can determine the version of the GNU C library present on the system: by testing constants or by calling a library function. From version 2.0 onward, *glibc* defines two constants, `__GLIBC__` and `__GLIBC_MINOR__`, that can be tested at compile time (in `#ifdef` statements). On a system with *glibc* 2.12 installed, these constants would have the values 2 and 12. However, these constants are of limited use in a program that is compiled on one system but run on another system with a different *glibc*. To handle this possibility, a program can call the `gnu_get_libc_version()` function to determine the version of *glibc* available at run time.

```
#include <gnu/libc-version.h>
```

```
const char *gnu_get_libc_version(void);
```

Returns pointer to null-terminated, statically allocated string
containing GNU C library version number

The `gnu_get_libc_version()` function returns a pointer to a string, such as *2.12*.

We can also obtain version information by using the `confstr()` function to retrieve the value of the (*glibc*-specific) `_CS_GNU_LIBC_VERSION` configuration variable. This call returns a string such as *glibc 2.12*.

3.4 Handling Errors from System Calls and Library Functions

Almost every system call and library function returns some type of status value indicating whether the call succeeded or failed. This status value should *always* be checked to see whether the call succeeded. If it did not, then appropriate action should be taken—at the very least, the program should display an error message warning that something unexpected occurred.

Although it is tempting to save typing time by excluding these checks (especially after seeing examples of UNIX and Linux programs where status values are not checked), it is a false economy. Many hours of debugging time can be wasted because a check was not made on the status return of a system call or library function that “couldn’t possibly fail.”

A few system calls never fail. For example, `getpid()` always successfully returns the ID of a process, and `_exit()` always terminates a process. It is not necessary to check the return values from such system calls.

Handling system call errors

The manual page for each system call documents the possible return values of the call, showing which value(s) indicate an error. Usually, an error is indicated by a return of `-1`. Thus, a system call can be checked with code such as the following:

```
fd = open(pathname, flags, mode);      /* system call to open a file */
if (fd == -1) {
    /* Code to handle the error */
}
...
```

```

if (close(fd) == -1) {
    /* Code to handle the error */
}

```

When a system call fails, it sets the global integer variable *errno* to a positive value that identifies the specific error. Including the `<errno.h>` header file provides a declaration of *errno*, as well as a set of constants for the various error numbers. All of these symbolic names commence with *E*. The section headed *ERRORS* in each manual page provides a list of possible *errno* values that can be returned by each system call. Here is a simple example of the use of *errno* to diagnose a system call error:

```

cnt = read(fd, buf, numbytes);
if (cnt == -1) {
    if (errno == EINTR)
        fprintf(stderr, "read was interrupted by a signal\n");
    else {
        /* Some other error occurred */
    }
}

```

Successful system calls and library functions never reset *errno* to 0, so this variable may have a nonzero value as a consequence of an error from a previous call. Furthermore, SUSv3 permits a successful function call to set *errno* to a nonzero value (although few functions do this). Therefore, when checking for an error, we should always first check if the function return value indicates an error, and only then examine *errno* to determine the cause of the error.

A few system calls (e.g., *getpriority()*) can legitimately return -1 on success. To determine whether an error occurs in such calls, we set *errno* to 0 before the call, and then check it afterward. If the call returns -1 and *errno* is nonzero, an error occurred. (A similar statement also applies to a few library functions.)

A common course of action after a failed system call is to print an error message based on the *errno* value. The *perror()* and *strerror()* library functions are provided for this purpose.

The *perror()* function prints the string pointed to by its *msg* argument, followed by a message corresponding to the current value of *errno*.

```

#include <stdio.h>

void perror(const char *msg);

```

A simple way of handling errors from system calls would be as follows:

```

fd = open(pathname, flags, mode);
if (fd == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}

```

The *strerror()* function returns the error string corresponding to the error number given in its *errnum* argument.

```
#include <string.h>
```

```
char *strerror(int errnum);
```

Returns pointer to error string corresponding to *errnum*

The string returned by *strerror()* may be statically allocated, which means that it could be overwritten by subsequent calls to *strerror()*.

If *errnum* specifies an unrecognized error number, *strerror()* returns a string of the form *Unknown error nnn*. On some other implementations, *strerror()* instead returns NULL in this case.

Because *perror()* and *strerror()* functions are locale-sensitive (Section 10.4), error descriptions are displayed in the local language.

Handling errors from library functions

The various library functions return different data types and different values to indicate failure. (Check the manual page for each function.) For our purposes, library functions can be divided into the following categories:

- Some library functions return error information in exactly the same way as system calls: a -1 return value, with *errno* indicating the specific error. An example of such a function is *remove()*, which removes a file (using the *unlink()* system call) or a directory (using the *rmdir()* system call). Errors from these functions can be diagnosed in the same way as errors from system calls.
- Some library functions return a value other than -1 on error, but nevertheless set *errno* to indicate the specific error condition. For example, *fopen()* returns a NULL pointer on error, and the setting of *errno* depends on which underlying system call failed. The *perror()* and *strerror()* functions can be used to diagnose these errors.
- Other library functions don't use *errno* at all. The method for determining the existence and cause of errors depends on the particular function and is documented in the function's manual page. For these functions, it is a mistake to use *errno*, *perror()*, or *strerror()* to diagnose errors.

3.5 Notes on the Example Programs in This Book

In this section, we describe various conventions and features commonly employed by the example programs presented in this book.

3.5.1 Command-Line Options and Arguments

Many of the example programs in this book rely on command-line options and arguments to determine their behavior.

Traditional UNIX command-line options consist of an initial hyphen, a letter that identifies the option, and an optional argument. (GNU utilities provide an extended option syntax consisting of two initial hyphens, followed by a string identifying the option and an optional argument.) To parse these options, we use the standard *getopt()* library function (described in Appendix B).

Each of our example programs that has a nontrivial command-line syntax provides a simple help facility for the user: if invoked with the `--help` option, the program displays a usage message that indicates the syntax for command-line options and arguments.

3.5.2 Common Functions and Header Files

Most of the example programs include a header file containing commonly required definitions, and they also use a set of common functions. We discuss the header file and functions in this section.

Common header file

Listing 3-1 is the header file used by nearly every program in this book. This header file includes various other header files used by many of the example programs, defines a *Boolean* data type, and defines macros for calculating the minimum and maximum of two numeric values. Using this header file allows us to make the example programs a bit shorter.

Listing 3-1: Header file used by most example programs

```
lib/tlpi_hdr.h

#ifndef TLPI_HDR_H
#define TLPI_HDR_H    /* Prevent accidental double inclusion */

#include <sys/types.h> /* Type definitions used by many programs */
#include <stdio.h>     /* Standard I/O functions */
#include <stdlib.h>    /* Prototypes of commonly used library functions,
                        plus EXIT_SUCCESS and EXIT_FAILURE constants */
#include <unistd.h>    /* Prototypes for many system calls */
#include <errno.h>     /* Declares errno and defines error constants */
#include <string.h>    /* Commonly used string-handling functions */

#include "get_num.h"   /* Declares our functions for handling numeric
                        arguments (getInt(), getLong()) */

#include "error_functions.h" /* Declares our error-handling functions */

typedef enum { FALSE, TRUE } Boolean;

#define min(m,n) ((m) < (n) ? (m) : (n))
#define max(m,n) ((m) > (n) ? (m) : (n))

#endif

lib/tlpi_hdr.h
```

Error-diagnostic functions

To simplify error handling in our example programs, we use the error-diagnostic functions whose declarations are shown in Listing 3-2.

Listing 3-2: Declarations for common error-handling functions

lib/error_functions.h

```
#ifndef ERROR_FUNCTIONS_H
#define ERROR_FUNCTIONS_H

void errMsg(const char *format, ...);

#ifdef __GNUC__

/* This macro stops 'gcc -Wall' complaining that "control reaches
   end of non-void function" if we use the following functions to
   terminate main() or some other non-void function. */

#define NORETURN __attribute__((__noreturn__))
#else
#define NORETURN
#endif

void errExit(const char *format, ...) NORETURN ;

void err_exit(const char *format, ...) NORETURN ;

void errExitEN(int errnum, const char *format, ...) NORETURN ;

void fatal(const char *format, ...) NORETURN ;

void usageErr(const char *format, ...) NORETURN ;

void cmdLineErr(const char *format, ...) NORETURN ;

#endif
```

lib/error_functions.h

To diagnose errors from system calls and library functions, we use *errMsg()*, *errExit()*, *err_exit()*, and *errExitEN()*.

```
#include "tspi_hdr.h"

void errMsg(const char *format, ...);
void errExit(const char *format, ...);
void err_exit(const char *format, ...);
void errExitEN(int errnum, const char *format, ...);
```

The *errMsg()* function prints a message on standard error. Its argument list is the same as for *printf()*, except that a terminating newline character is automatically appended to the output string. The *errMsg()* function prints the error text corresponding to the current value of *errno*—this consists of the error name, such as *EPERM*, plus the error description as returned by *strerror()*—followed by the formatted output specified in the argument list.

The *errExit()* function operates like *errMsg()*, but also terminates the program, either by calling *exit()* or, if the environment variable *EF_DUMPCORE* is defined with a

nonempty string value, by calling *abort()* to produce a core dump file for use with the debugger. (We explain core dump files in Section 22.1.)

The *err_exit()* function is similar to *errExit()*, but differs in two respects:

- It doesn't flush standard output before printing the error message.
- It terminates the process by calling *_exit()* instead of *exit()*. This causes the process to terminate without flushing *stdio* buffers or invoking exit handlers.

The details of these differences in the operation of *err_exit()* will become clearer in Chapter 25, where we describe the differences between *_exit()* and *exit()*, and consider the treatment of *stdio* buffers and exit handlers in a child created by *fork()*. For now, we simply note that *err_exit()* is especially useful if we write a library function that creates a child process that needs to terminate because of an error. This termination should occur without flushing the child's copy of the parent's (i.e., the calling process's) *stdio* buffers and without invoking exit handlers established by the parent.

The *errExitEN()* function is the same as *errExit()*, except that instead of printing the error text corresponding to the current value of *errno*, it prints the text corresponding to the error number (thus, the *EN* suffix) given in the argument *errnum*.

Mainly, we use *errExitEN()* in programs that employ the POSIX threads API. Unlike traditional UNIX system calls, which return -1 on error, the POSIX threads functions diagnose an error by returning an error number (i.e., a positive number of the type normally placed in *errno*) as their function result. (The POSIX threads functions return 0 on success.)

We could diagnose errors from the POSIX threads functions using code such as the following:

```
errno = pthread_create(&thread, NULL, func, &arg);
if (errno != 0)
    errExit("pthread_create");
```

However, this approach is inefficient because *errno* is defined in threaded programs as a macro that expands into a function call that returns a modifiable lvalue. Thus, each use of *errno* results in a function call. The *errExitEN()* function allows us to write a more efficient equivalent of the above code:

```
int s;

s = pthread_create(&thread, NULL, func, &arg);
if (s != 0)
    errExitEN(s, "pthread_create");
```

In C terminology, an *lvalue* is an expression referring to a region of storage. The most common example of an lvalue is an identifier for a variable. Some operators also yield lvalues. For example, if *p* is a pointer to a storage area, then **p* is an lvalue. Under the POSIX threads API, *errno* is redefined to be a function that returns a pointer to a thread-specific storage area (see Section 31.3).

To diagnose other types of errors, we use *fatal()*, *usageErr()*, and *cmdLineErr()*.

```

#include "tlpi_hdr.h"

void fatal(const char *format, ...);
void usageErr(const char *format, ...);
void cmdLineErr(const char *format, ...);

```

The *fatal()* function is used to diagnose general errors, including errors from library functions that don't set *errno*. Its argument list is the same as for *printf()*, except that a terminating newline character is automatically appended to the output string. It prints the formatted output on standard error and then terminates the program as with *errExit()*.

The *usageErr()* function is used to diagnose errors in command-line argument usage. It takes an argument list in the style of *printf()* and prints the string *Usage:* followed by the formatted output on standard error, and then terminates the program by calling *exit()*. (Some of the example programs in this book provide their own extended version of the *usageErr()* function, under the name *usageError()*.)

The *cmdLineErr()* function is similar to *usageErr()*, but is intended for diagnosing errors in the command-line arguments specified to a program.

The implementations of our error-diagnostic functions are shown in Listing 3-3.

Listing 3-3: Error-handling functions used by all programs

```

lib/error_functions.c

#include <stdarg.h>
#include "error_functions.h"
#include "tlpi_hdr.h"
#include "ename.c.inc"          /* Defines ename and MAX_ENAME */

#ifdef __GNUC__
__attribute__((__noreturn__))
#endif
static void
terminate(Boolean useExit3)
{
    char *s;

    /* Dump core if EF_DUMPCORE environment variable is defined and
       is a nonempty string; otherwise call exit(3) or _exit(2),
       depending on the value of 'useExit3'. */

    s = getenv("EF_DUMPCORE");

    if (s != NULL && *s != '\0')
        abort();
    else if (useExit3)
        exit(EXIT_FAILURE);
    else
        _exit(EXIT_FAILURE);
}

```

```

static void
outputError(Boolean useErr, int err, Boolean flushStdout,
             const char *format, va_list ap)
{
#define BUF_SIZE 500
    char buf[BUF_SIZE], userMsg[BUF_SIZE], errText[BUF_SIZE];

    vsnprintf(userMsg, BUF_SIZE, format, ap);

    if (useErr)
        snprintf(errText, BUF_SIZE, " [%s %s]",
                 (err > 0 && err <= MAX_ENAME) ?
                 ename[err] : "?UNKNOWN?", strerror(err));
    else
        snprintf(errText, BUF_SIZE, ":");

    snprintf(buf, BUF_SIZE, "ERROR%s %s\n", errText, userMsg);

    if (flushStdout)
        fflush(stdout);          /* Flush any pending stdout */
    fputs(buf, stderr);
    fflush(stderr);             /* In case stderr is not line-buffered */
}

void
errMsg(const char *format, ...)
{
    va_list argList;
    int savedErrno;

    savedErrno = errno;         /* In case we change it here */

    va_start(argList, format);
    outputError(TRUE, errno, TRUE, format, argList);
    va_end(argList);

    errno = savedErrno;
}

void
errExit(const char *format, ...)
{
    va_list argList;

    va_start(argList, format);
    outputError(TRUE, errno, TRUE, format, argList);
    va_end(argList);

    terminate(TRUE);
}

```

```

void
err_exit(const char *format, ...)
{
    va_list argList;

    va_start(argList, format);
    outputError(TRUE, errno, FALSE, format, argList);
    va_end(argList);

    terminate(FALSE);
}

void
errExitEN(int errnum, const char *format, ...)
{
    va_list argList;

    va_start(argList, format);
    outputError(TRUE, errnum, TRUE, format, argList);
    va_end(argList);

    terminate(TRUE);
}

void
fatal(const char *format, ...)
{
    va_list argList;

    va_start(argList, format);
    outputError(FALSE, 0, TRUE, format, argList);
    va_end(argList);

    terminate(TRUE);
}

void
usageErr(const char *format, ...)
{
    va_list argList;

    fflush(stdout);           /* Flush any pending stdout */

    fprintf(stderr, "Usage: ");
    va_start(argList, format);
    vfprintf(stderr, format, argList);
    va_end(argList);

    fflush(stderr);           /* In case stderr is not line-buffered */
    exit(EXIT_FAILURE);
}

```

```

void
cmdLineErr(const char *format, ...)
{
    va_list argList;

    fflush(stdout);           /* Flush any pending stdout */

    fprintf(stderr, "Command-line usage error: ");
    va_start(argList, format);
    vfprintf(stderr, format, argList);
    va_end(argList);

    fflush(stderr);           /* In case stderr is not line-buffered */
    exit(EXIT_FAILURE);
}

```

lib/error_functions.c

The file `ename.c.inc` included by Listing 3-3 is shown in Listing 3-4. This file defines an array of strings, *ename*, that are the symbolic names corresponding to each of the possible *errno* values. Our error-handling functions use this array to print out the symbolic name corresponding to a particular error number. This is a workaround to deal with the facts that, on the one hand, the string returned by *strerror()* doesn't identify the symbolic constant corresponding to its error message, while, on the other hand, the manual pages describe errors using their symbolic names. Printing out the symbolic name gives us an easy way of looking up the cause of an error in the manual pages.

The content of the `ename.c.inc` file is architecture-specific, because *errno* values vary somewhat from one Linux hardware architecture to another. The version shown in Listing 3-4 is for a Linux 2.6/x86-32 system. This file was built using a script (`lib/Build_ename.sh`) included in the source code distribution for this book. This script can be used to build a version of `ename.c.inc` that should be suitable for a specific hardware platform and kernel version.

Note that some of the strings in the *ename* array are empty. These correspond to unused error values. Furthermore, some of the strings in *ename* consist of two error names separate by a slash. These strings correspond to cases where two symbolic error names have the same numeric value.

From the `ename.c.inc` file, we can see that the `EAGAIN` and `EWOULDBLOCK` errors have the same value. (SUSv3 explicitly permits this, and the values of these constants are the same on most, but not all, other UNIX systems.) These errors are returned by a system call in circumstances in which it would normally block (i.e., be forced to wait before completing), but the caller requested that the system call return an error instead of blocking. `EAGAIN` originated on System V, and it was the error returned for system calls performing I/O, semaphore operations, message queue operations, and file locking (*fcntl()*). `EWOULDBLOCK` originated on BSD, and it was returned by file locking (*flock()*) and socket-related system calls.

Within SUSv3, `EWOULDBLOCK` is mentioned only in the specifications of various interfaces related to sockets. For these interfaces, SUSv3 permits either `EAGAIN` or `EWOULDBLOCK` to be returned by nonblocking calls. For all other nonblocking calls, only the error `EAGAIN` is specified in SUSv3.

Listing 3-4: Linux error names (x86-32 version)

`lib/ename.c.inc`

```
static char *ename[] = {
    /* 0 */ "",
    /* 1 */ "EPERM", "ENOENT", "ESRCH", "EINTR", "EIO", "ENXIO", "E2BIG",
    /* 8 */ "ENOEXEC", "EBADF", "ECHILD", "EAGAIN/EWOULDBLOCK", "ENOMEM",
    /* 13 */ "EACCES", "EFAULT", "ENOTBLK", "EBUSY", "EEXIST", "EXDEV",
    /* 19 */ "ENODEV", "ENOTDIR", "EISDIR", "EINVAL", "ENFILE", "EMFILE",
    /* 25 */ "ENOTTY", "ETXTBSY", "EFBIG", "ENOSPC", "ESPIPE", "EROFS",
    /* 31 */ "EMLINK", "EPIPE", "EDOM", "ERANGE", "EDEADLK/EDEADLOCK",
    /* 36 */ "ENAMETOOLONG", "ENOLCK", "ENOSYS", "ENOTEMPTY", "ELOOP", "",
    /* 42 */ "ENOMSG", "EIDRM", "ECHRNG", "EL2NSYNC", "EL3HLT", "EL3RST",
    /* 48 */ "ELNRNG", "EUNATCH", "ENOCST", "EL2HLT", "EBAD", "EBADR",
    /* 54 */ "EXFULL", "ENOANO", "EBADRQC", "EBADSLT", "", "EBFONT", "ENOSTR",
    /* 61 */ "ENODATA", "ETIME", "ENOSR", "ENONET", "ENOPKG", "EREMOTE",
    /* 67 */ "ENOLINK", "EADV", "ESRMNT", "ECOMM", "EPROTO", "EMULTIHOP",
    /* 73 */ "EDOTDOT", "EBADMSG", "EOVERFLOW", "ENOTUNIQ", "EBADFD",
    /* 78 */ "EREMCHG", "ELIBACC", "ELIBBAD", "ELIBSCN", "ELIBMAX",
    /* 83 */ "ELIBEXEC", "EILSEQ", "ERESTART", "ESTRPIPE", "EUSERS",
    /* 88 */ "ENOTSOCK", "EDESTADDRREQ", "EMSGSIZE", "EPROTOPTYPE",
    /* 92 */ "ENOPROTOPT", "EPROTONOSUPPORT", "ESOCKTNOSUPPORT",
    /* 95 */ "EOPNOTSUPP/ENOTSUP", "EPFNOSUPPORT", "EAFNOSUPPORT",
    /* 98 */ "EADDRINUSE", "EADDRNOTAVAIL", "ENETDOWN", "ENETUNREACH",
    /* 102 */ "ENETRESET", "ECONNABORTED", "ECONNRESET", "ENOBUFS", "EISCONN",
    /* 107 */ "ENOTCONN", "ESHUTDOWN", "ETOOMANYREFS", "ETIMEDOUT",
    /* 111 */ "ECONNREFUSED", "EHOSTDOWN", "EHOSTUNREACH", "EALREADY",
    /* 115 */ "EINPROGRESS", "ESTALE", "EUCLEAN", "ENOTNAM", "ENAVAIL",
    /* 120 */ "EISNAM", "EREMOTEIO", "EDQUOT", "ENOMEDIUM", "EMEDIUMTYPE",
    /* 125 */ "ECANCELED", "ENOKEY", "EKEYEXPIRED", "EKEYREVOKED",
    /* 129 */ "EKEYREJECTED", "EOWNERDEAD", "ENOTRECOVERABLE", "ERFKILL"
};

#define MAX_ENAME 132
```

`lib/ename.c.inc`**Functions for parsing numeric command-line arguments**

The header file in Listing 3-5 provides the declaration of two functions that we frequently use for parsing integer command-line arguments: *getInt()* and *getLong()*. The primary advantage of using these functions instead of *atoi()*, *atol()*, and *strtol()* is that they provide some basic validity checking of numeric arguments.

```
#include "tapi_hdr.h"
```

```
int getInt(const char *arg, int flags, const char *name);
long getLong(const char *arg, int flags, const char *name);
```

Both return *arg* converted to numeric form

The *getInt()* and *getLong()* functions convert the string pointed to by *arg* to an *int* or a *long*, respectively. If *arg* doesn't contain a valid integer string (i.e., only digits and the characters + and -), then these functions print an error message and terminate the program.

If the *name* argument is non-NULL, it should contain a string identifying the argument in *arg*. This string is included as part of any error message displayed by these functions.

The *flags* argument provides some control over the operation of the *getInt()* and *getLong()* functions. By default, these functions expect strings containing signed decimal integers. By ORing (!) one or more of the GN_* constants defined in Listing 3-5 into *flags*, we can select alternative bases for conversion and restrict the range of the number to being nonnegative or greater than 0.

The implementations of the *getInt()* and *getLong()* functions are provided in Listing 3-6.

Although the *flags* argument allows us to enforce the range checks described in the main text, in some cases, we don't request such checks in our example programs, even though it might seem logical to do so. For example, in Listing 47-1, we don't check the *init-value* argument. This means that the user could specify a negative number as the initial value for a semaphore, which would result in an error (ERANGE) in the subsequent *semctl()* system call, because a semaphore can't have a negative value. Omitting range checks in such cases allows us to experiment not just with the correct use of system calls and library functions, but also to see what happens when invalid arguments are supplied. Real-world applications would usually impose stronger checks on their command-line arguments.

Listing 3-5: Header file for *get_num.c*

```
#ifndef GET_NUM_H
#define GET_NUM_H

#define GN_NONNEG      01      /* Value must be >= 0 */
#define GN_GT_0        02      /* Value must be > 0 */

                                /* By default, integers are decimal */
#define GN_ANY_BASE    0100    /* Can use any base - like strtol(3) */
#define GN_BASE_8      0200    /* Value is expressed in octal */
#define GN_BASE_16     0400    /* Value is expressed in hexadecimal */

long getLong(const char *arg, int flags, const char *name);

int getInt(const char *arg, int flags, const char *name);

#endif
```

lib/get_num.h

Listing 3-6: Functions for parsing numeric command-line arguments

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <errno.h>
#include "get_num.h"
```

lib/get_num.c

```

static void
gnFail(const char *fname, const char *msg, const char *arg, const char *name)
{
    fprintf(stderr, "%s error", fname);
    if (name != NULL)
        fprintf(stderr, " (in %s)", name);
    fprintf(stderr, ": %s\n", msg);
    if (arg != NULL && *arg != '\0')
        fprintf(stderr, "      offending text: %s\n", arg);

    exit(EXIT_FAILURE);
}

static long
getNum(const char *fname, const char *arg, int flags, const char *name)
{
    long res;
    char *endptr;
    int base;

    if (arg == NULL || *arg == '\0')
        gnFail(fname, "null or empty string", arg, name);

    base = (flags & GN_ANY_BASE) ? 0 : (flags & GN_BASE_8) ? 8 :
          (flags & GN_BASE_16) ? 16 : 10;

    errno = 0;
    res = strtol(arg, &endptr, base);
    if (errno != 0)
        gnFail(fname, "strtol() failed", arg, name);

    if (*endptr != '\0')
        gnFail(fname, "nonnumeric characters", arg, name);

    if ((flags & GN_NONNEG) && res < 0)
        gnFail(fname, "negative value not allowed", arg, name);

    if ((flags & GN_GT_0) && res <= 0)
        gnFail(fname, "value must be > 0", arg, name);

    return res;
}

long
getLong(const char *arg, int flags, const char *name)
{
    return getNum("getLong", arg, flags, name);
}

int
getInt(const char *arg, int flags, const char *name)
{
    long res;

    res = getNum("getInt", arg, flags, name);
}

```

```

    if (res > INT_MAX || res < INT_MIN)
        gnFail("getInt", "integer out of range", arg, name);

    return (int) res;
}

```

lib/get_num.c

3.6 Portability Issues

In this section, we consider the topic of writing portable system programs. We introduce feature test macros and the standard system data types defined by SUSv3, and then look at some other portability issues.

3.6.1 Feature Test Macros

Various standards govern the behavior of the system call and library function APIs (see Section 1.3). Some of these standards are defined by standards bodies such as The Open Group (Single UNIX Specification), while others are defined by the two historically important UNIX implementations: BSD and System V Release 4 (and the associated System V Interface Definition).

Sometimes, when writing a portable application, we may want the various header files to expose only the definitions (constants, function prototypes, and so on) that follow a particular standard. To do this, we define one or more of the *feature test macros* listed below when compiling a program. One way that we can do this is by defining the macro in the program source code before including any header files:

```
#define _BSD_SOURCE 1
```

Alternatively, we can use the `-D` option to the C compiler:

```
$ cc -D_BSD_SOURCE prog.c
```

The term *feature test macro* may seem confusing, but it makes sense if we look at things from the perspective of the implementation. The implementation decides which of the *features* available in each header it should make visible, by *testing* (with `#if`) which values the application has defined for these *macros*.

The following feature test macros are specified by the relevant standards, and consequently their usage is portable to all systems that support these standards:

`_POSIX_SOURCE`

If defined (with any value), expose definitions conforming to POSIX.1-1990 and ISO C (1990). This macro is superseded by `_POSIX_C_SOURCE`.

`_POSIX_C_SOURCE`

If defined with the value 1, this has the same effect as `_POSIX_SOURCE`. If defined with a value greater than or equal to 199309, also expose definitions for POSIX.1b (realtime). If defined with a value greater than or equal to 199506, also expose definitions for POSIX.1c (threads). If defined with the value 200112, also expose definitions for the POSIX.1-2001 base specification (i.e., the XSI extension is excluded). (Prior to version 2.3.3, the

glibc headers don't interpret the value 200112 for `_POSIX_C_SOURCE`.) If defined with the value 200809, also expose definitions for the POSIX.1-2008 base specification. (Prior to version 2.10, the *glibc* headers don't interpret the value 200809 for `_POSIX_C_SOURCE`.)

`_XOPEN_SOURCE`

If defined (with any value), expose POSIX.1, POSIX.2, and X/Open (XPG4) definitions. If defined with the value 500 or greater, also expose SUSv2 (UNIX 98 and XPG5) extensions. Setting to 600 or greater additionally exposes SUSv3 XSI (UNIX 03) extensions and C99 extensions. (Prior to version 2.2, the *glibc* headers don't interpret the value 600 for `_XOPEN_SOURCE`.) Setting to 700 or greater also exposes SUSv4 XSI extensions. (Prior to version 2.10, the *glibc* headers don't interpret the value 700 for `_XOPEN_SOURCE`.) The values 500, 600, and 700 for `_XOPEN_SOURCE` were chosen because SUSv2, SUSv3, and SUSv4 are Issues 5, 6, and 7, respectively, of the X/Open specifications.

The following feature test macros listed are *glibc*-specific:

`_BSD_SOURCE`

If defined (with any value), expose BSD definitions. Defining this macro also defines `_POSIX_C_SOURCE` with the value 199506. Explicitly setting just this macro causes BSD definitions to be favored in a few cases where standards conflict.

`_SVID_SOURCE`

If defined (with any value), expose System V Interface Definition (SVID) definitions.

`_GNU_SOURCE`

If defined (with any value), expose all of the definitions provided by setting all of the preceding macros, as well as various GNU extensions.

When the GNU C compiler is invoked without special options, `_POSIX_SOURCE`, `_POSIX_C_SOURCE=200809` (200112 with *glibc* versions 2.5 to 2.9, or 199506 with *glibc* versions earlier than 2.4), `_BSD_SOURCE`, and `_SVID_SOURCE` are defined by default.

If individual macros are defined, or the compiler is invoked in one of its standard modes (e.g., `cc -ansi` or `cc -std=c99`), then only the requested definitions are supplied. There is one exception: if `_POSIX_C_SOURCE` is not otherwise defined, and the compiler is not invoked in one of its standard modes, then `_POSIX_C_SOURCE` is defined with the value 200809 (200112 with *glibc* versions 2.4 to 2.9, or 199506 with *glibc* versions earlier than 2.4).

Defining multiple macros is additive, so that we could, for example, use the following `cc` command to explicitly select the same macro settings as are provided by default:

```
$ cc -D_POSIX_SOURCE -D_POSIX_C_SOURCE=199506 \
      -D_BSD_SOURCE -D_SVID_SOURCE prog.c
```

The `<features.h>` header file and the *feature_test_macros*(7) manual page provide further information on precisely what values are assigned to each of the feature test macros.

`_POSIX_C_SOURCE`, `_XOPEN_SOURCE`, and `POSIX.1/SUS`

Only the `_POSIX_C_SOURCE` and `_XOPEN_SOURCE` feature test macros are specified in POSIX.1-2001/SUSv3, which requires that these macros be defined with the values 200112 and 600, respectively, in conforming applications. Defining `_POSIX_C_SOURCE` as 200112 provides conformance to the POSIX.1-2001 base specification (i.e., *POSIX conformance*, excluding the XSI extension). Defining `_XOPEN_SOURCE` as 600 provides conformance to SUSv3 (i.e., *XSI conformance*, the base specification plus the XSI extension). Analogous statements apply for POSIX.1-2008/SUSv4, which require that the two macros be defined with the values 200809 and 700.

SUSv3 specifies that setting `_XOPEN_SOURCE` to 600 should supply all of the features that are enabled if `_POSIX_C_SOURCE` is set to 200112. Thus, an application needs to define only `_XOPEN_SOURCE` for SUSv3 (i.e., XSI) conformance. SUSv4 makes an analogous specification that setting `_XOPEN_SOURCE` to 700 should supply all of the features that are enabled if `_POSIX_C_SOURCE` is set to 200809.

Feature test macros in function prototypes and source code examples

The manual pages describe which feature test macro(s) must be defined in order to make a particular constant definition or function declaration visible from a header file.

All of the source code examples in this book are written so that they will compile using either the default GNU C compiler options or the following options:

```
$ cc -std=c99 -D_XOPEN_SOURCE=600
```

The prototype of each function shown in this book indicates any feature test macro(s) that must be defined in order to employ that function in a program compiled with either the default compiler options or the options in the `cc` just shown. The manual pages provide more precise descriptions of the feature test macro(s) required to expose the declaration of each function.

3.6.2 System Data Types

Various implementation data types are represented using standard C types, for example, process IDs, user IDs, and file offsets. Although it would be possible to use the C fundamental types such as `int` and `long` to declare variables storing such information, this reduces portability across UNIX systems, for the following reasons:

- The sizes of these fundamental types vary across UNIX implementations (e.g., a `long` may be 4 bytes on one system and 8 bytes on another), or sometimes even in different compilation environments on the same implementation. Furthermore, different implementations may use different types to represent the same information. For example, a process ID might be an `int` on one system but a `long` on another.
- Even on a single UNIX implementation, the types used to represent information may differ between releases of the implementation. Notable examples on Linux are user and group IDs. On Linux 2.2 and earlier, these values were represented in 16 bits. On Linux 2.4 and later, they are 32-bit values.

To avoid such portability problems, SUSv3 specifies various standard system data types, and requires an implementation to define and use these types appropriately.

Each of these types is defined using the C typedef feature. For example, the *pid_t* data type is intended for representing process IDs, and on Linux/x86-32 this type is defined as follows:

```
typedef int pid_t;
```

Most of the standard system data types have names ending in *_t*. Many of them are declared in the header file `<sys/types.h>`, although a few are defined in other header files.

An application should employ these type definitions to portably declare the variables it uses. For example, the following declaration would allow an application to correctly represent process IDs on any SUSv3-conformant system:

```
pid_t mypid;
```

Table 3-1 lists some of the system data types we'll encounter in this book. For certain types in this table, SUSv3 requires that the type be implemented as an *arithmetic type*. This means that the implementation may choose the underlying type as either an integer or a floating-point (real or complex) type.

Table 3-1: Selected system data types

Data type	SUSv3 type requirement	Description
<i>blkcnt_t</i>	signed integer	File block count (Section 15.1)
<i>blksize_t</i>	signed integer	File block size (Section 15.1)
<i>cc_t</i>	unsigned integer	Terminal special character (Section 62.4)
<i>clock_t</i>	integer or real-floating	System time in clock ticks (Section 10.7)
<i>clockid_t</i>	an arithmetic type	Clock identifier for POSIX.1b clock and timer functions (Section 23.6)
<i>comp_t</i>	not in SUSv3	Compressed clock ticks (Section 28.1)
<i>dev_t</i>	an arithmetic type	Device number, consisting of major and minor numbers (Section 15.1)
<i>DIR</i>	no type requirement	Directory stream (Section 18.8)
<i>fd_set</i>	structure type	File descriptor set for <i>select()</i> (Section 63.2.1)
<i>fsblkcnt_t</i>	unsigned integer	File-system block count (Section 14.11)
<i>fsfilcnt_t</i>	unsigned integer	File count (Section 14.11)
<i>gid_t</i>	integer	Numeric group identifier (Section 8.3)
<i>id_t</i>	integer	A generic type for holding identifiers; large enough to hold at least <i>pid_t</i> , <i>uid_t</i> , and <i>gid_t</i>
<i>in_addr_t</i>	32-bit unsigned integer	IPv4 address (Section 59.4)
<i>in_port_t</i>	16-bit unsigned integer	IP port number (Section 59.4)
<i>ino_t</i>	unsigned integer	File i-node number (Section 15.1)
<i>key_t</i>	an arithmetic type	System V IPC key (Section 45.2)
<i>mode_t</i>	integer	File permissions and type (Section 15.1)
<i>mqd_t</i>	no type requirement, but shall not be an array type	POSIX message queue descriptor
<i>msglen_t</i>	unsigned integer	Number of bytes allowed in System V message queue (Section 46.4)

Table 3-1: Selected system data types (continued)

Data type	SUSv3 type requirement	Description
<i>msgqnum_t</i>	unsigned integer	Counts of messages in System V message queue (Section 46.4)
<i>nfds_t</i>	unsigned integer	Number of file descriptors for <i>poll()</i> (Section 63.2.2)
<i>nlink_t</i>	integer	Count of (hard) links to a file (Section 15.1)
<i>off_t</i>	signed integer	File offset or size (Sections 4.7 and 15.1)
<i>pid_t</i>	signed integer	Process ID, process group ID, or session ID (Sections 6.2, 34.2, and 34.3)
<i>ptrdiff_t</i>	signed integer	Difference between two pointer values, as a signed integer
<i>rlim_t</i>	unsigned integer	Resource limit (Section 36.2)
<i>sa_family_t</i>	unsigned integer	Socket address family (Section 56.4)
<i>shmatt_t</i>	unsigned integer	Count of attached processes for a System V shared memory segment (Section 48.8)
<i>sig_atomic_t</i>	integer	Data type that can be atomically accessed (Section 21.1.3)
<i>siginfo_t</i>	structure type	Information about the origin of a signal (Section 21.4)
<i>sigset_t</i>	integer or structure type	Signal set (Section 20.9)
<i>size_t</i>	unsigned integer	Size of an object in bytes
<i>socklen_t</i>	integer type of at least 32 bits	Size of a socket address structure in bytes (Section 56.3)
<i>speed_t</i>	unsigned integer	Terminal line speed (Section 62.7)
<i>ssize_t</i>	signed integer	Byte count or (negative) error indication
<i>stack_t</i>	structure type	Description of an alternate signal stack (Section 21.3)
<i>suseconds_t</i>	signed integer allowing range [-1, 1000000]	Microsecond time interval (Section 10.1)
<i>tcflag_t</i>	unsigned integer	Terminal mode flag bit mask (Section 62.2)
<i>time_t</i>	integer or real-floating	Calendar time in seconds since the Epoch (Section 10.1)
<i>timer_t</i>	an arithmetic type	Timer identifier for POSIX.1b interval timer functions (Section 23.6)
<i>uid_t</i>	integer	Numeric user identifier (Section 8.1)

When discussing the data types in Table 3-1 in later chapters, we'll often make statements that some type "is an integer type [specified by SUSv3]." This means that SUSv3 requires the type to be defined as an integer, but doesn't require that a particular native integer type (e.g., *short*, *int*, or *long*) be used. (Often, we won't say which particular native data type is actually used to represent each of the system data types in Linux, because a portable application should be written so that it doesn't care which data type is used.)

Printing system data type values

When printing values of one of the numeric system data types shown in Table 3-1 (e.g., *pid_t* and *uid_t*), we must be careful not to include a representation dependency in the *printf()* call. A representation dependency can occur because C's argument promotion rules convert values of type *short* to *int*, but leave values of type *int* and *long* unchanged. This means that, depending on the definition of the system data type, either an *int* or a *long* is passed in the *printf()* call. However, because *printf()* has no way to determine the types of its arguments at run time, the caller must explicitly provide this information using the *%d* or *%ld* format specifier. The problem is that simply coding one of these specifiers within the *printf()* call creates an implementation dependency. The usual solution is to use the *%ld* specifier and always cast the corresponding value to *long*, like so:

```
pid_t mypid;

mypid = getpid();           /* Returns process ID of calling process */
printf("My PID is %ld\n", (long) mypid);
```

We make one exception to the above technique. Because the *off_t* data type is the size of *long long* in some compilation environments, we cast *off_t* values to this type and use the *%lld* specifier, as described in Section 5.10.

The C99 standard defines the *z* length modifier for *printf()*, to indicate that the following integer conversion corresponds to a *size_t* or *ssize_t* type. Thus, we could write *%zd* instead of using *%ld* plus a cast for these types. Although this specifier is available in *glibc*, we avoid it because it is not available on all UNIX implementations.

The C99 standard also defines the *j* length modifier, which specifies that the corresponding argument is of type *intmax_t* (or *uintmax_t*), an integer type that is guaranteed to be large enough to be able to represent an integer of any type. Ultimately, the use of an (*intmax_t*) cast plus the *%jd* specifier should replace the (*long*) cast plus the *%ld* specifier as the best way of printing numeric system data type values, since the former approach also handles *long long* values and any extended integer types such as *int128_t*. However, (again) we avoid this technique since it is not possible on all UNIX implementations.

3.6.3 Miscellaneous Portability Issues

In this section, we consider a few other portability issues that we may encounter when writing system programs.

Initializing and using structures

Each UNIX implementation specifies a range of standard structures that are used in various system calls and library functions. As an example, consider the *sembuf* structure, which is used to represent a semaphore operation to be performed by the *semop()* system call:

```
struct sembuf {
    unsigned short sem_num;    /* Semaphore number */
    short          sem_op;     /* Operation to be performed */
    short          sem_flg;    /* Operation flags */
};
```

Although SUSv3 specifies structures such as *sembuf*, it is important to realize the following:

- In general, the order of field definitions within such structures is not specified.
- In some cases, extra implementation-specific fields may be included in such structures.

Consequently, it is not portable to use a structure initializer such as the following:

```
struct sembuf s = { 3, -1, SEM_UNDO };
```

Although this initializer will work on Linux, it won't work on another implementation where the fields in the *sembuf* structure are defined in a different order. To portably initialize such structures, we must use explicit assignment statements, as in the following:

```
struct sembuf s;  
  
s.sem_num = 3;  
s.sem_op = -1;  
s.sem_flg = SEM_UNDO;
```

If we are using C99, then we can employ that language's new syntax for structure initializers to write an equivalent initialization:

```
struct sembuf s = { .sem_num = 3, .sem_op = -1, .sem_flg = SEM_UNDO };
```

Considerations about the order of the members of standard structures also apply if we want to write the contents of a standard structure to a file. To do this portably, we can't simply do a binary write of the structure. Instead, the structure fields must be written individually (probably in text form) in a specified order.

Using macros that may not be present on all implementations

In some cases, a macro may be not be defined on all UNIX implementations. For example, the `WCOREDUMP()` macro (which checks whether a child process produced a core dump file) is widely available, but it is not specified in SUSv3. Therefore, this macro might not be present on some UNIX implementations. In order to portably handle such possibilities, we can use the C preprocessor `#ifdef` directive, as in the following example:

```
#ifdef WCOREDUMP  
/* Use WCOREDUMP() macro */  
#endif
```

Variation in required header files across implementations

In some cases, the header files required to prototype various system calls and library functions vary across UNIX implementations. In this book, we show the requirements on Linux and note any variations from SUSv3.

Some of the function synopses in this book show a particular header file with the accompanying comment */* For portability */*. This indicates that the header file

is not required on Linux or by SUSv3, but because some other (especially older) implementations may require it, we should include it in portable programs.

For many of the functions that it specified, POSIX.1-1990 required that the header `<sys/types.h>` be included before any other headers associated with the function. However, this requirement was redundant, because most contemporary UNIX implementations did not require applications to include this header for these functions. Consequently, SUSv1 removed this requirement. Nevertheless, when writing portable programs, it is wise to make this one of the first header files included. (However, we omit this header from our example programs because it is not required on Linux and omitting it allows us to make the example programs one line shorter.)

3.7 Summary

System calls allow processes to request services from the kernel. Even the simplest system calls have a significant overhead by comparison with a user-space function call, since the system must temporarily switch to kernel mode to execute the system call, and the kernel must verify system call arguments and transfer data between user memory and kernel memory.

The standard C library provides a multitude of library functions that perform a wide range of tasks. Some library functions employ system calls to do their work; others perform tasks entirely within user space. On Linux, the usual standard C library implementation that is used is *glibc*.

Most system calls and library functions return a status indicating whether a call has succeeded or failed. Such status returns should always be checked.

We introduced a number of functions that we have implemented for use in the example programs in this book. The tasks performed by these functions include diagnosing errors and parsing command-line arguments.

We discussed various guidelines and techniques that can help us write portable system programs that run on any standards-conformant system.

When compiling an application, we can define various feature test macros that control the definitions exposed by header files. This is useful if we want to ensure that a program conforms to some formal or implementation-defined standard(s).

We can improve the portability of system programs by using the system data types defined in various standards, rather than native C types. SUSv3 specifies a wide range of system data types that an implementation should support and that an application should employ.

3.8 Exercise

- 3-1. When using the Linux-specific *reboot()* system call to reboot the system, the second argument, *magic2*, must be specified as one of a set of magic numbers (e.g., `LINUX_REBOOT_MAGIC2`). What is the significance of these numbers? (Converting them to hexadecimal provides a clue.)

4

FILE I/O: THE UNIVERSAL I/O MODEL

We now start to look in earnest at the system call API. Files are a good place to start, since they are central to the UNIX philosophy. The focus of this chapter is the system calls used for performing file input and output.

We introduce the concept of a file descriptor, and then look at the system calls that constitute the so-called universal I/O model. These are the system calls that open and close a file, and read and write data.

We focus on I/O on disk files. However, much of the material covered here is relevant for later chapters, since the same system calls are used for performing I/O on all types of files, such as pipes and terminals.

Chapter 5 extends the discussion in this chapter with further details on file I/O. One other aspect of file I/O, buffering, is complex enough to deserve its own chapter. Chapter 13 covers I/O buffering in the kernel and in the *stdio* library.

4.1 Overview

All system calls for performing I/O refer to open files using a *file descriptor*, a (usually small) nonnegative integer. File descriptors are used to refer to all types of open files, including pipes, FIFOs, sockets, terminals, devices, and regular files. Each process has its own set of file descriptors.

By convention, most programs expect to be able to use the three standard file descriptors listed in Table 4-1. These three descriptors are opened on the program's

behalf by the shell, before the program is started. Or, more precisely, the program inherits copies of the shell's file descriptors, and the shell normally operates with these three file descriptors always open. (In an interactive shell, these three file descriptors normally refer to the terminal under which the shell is running.) If I/O redirections are specified on a command line, then the shell ensures that the file descriptors are suitably modified before starting the program.

Table 4-1: Standard file descriptors

File descriptor	Purpose	POSIX name	stdio stream
0	standard input	STDIN_FILENO	<i>stdin</i>
1	standard output	STDOUT_FILENO	<i>stdout</i>
2	standard error	STDERR_FILENO	<i>stderr</i>

When referring to these file descriptors in a program, we can use either the numbers (0, 1, or 2) or, preferably, the POSIX standard names defined in `<unistd.h>`.

Although the variables *stdin*, *stdout*, and *stderr* initially refer to the process's standard input, output, and error, they can be changed to refer to any file by using the *freopen()* library function. As part of its operation, *freopen()* may change the file descriptor underlying the reopened stream. In other words, after an *freopen()* on *stdout*, for example, it is no longer safe to assume that the underlying file descriptor is still 1.

The following are the four key system calls for performing file I/O (programming languages and software packages typically employ these calls only indirectly, via I/O libraries):

- *fd* = *open(pathname, flags, mode)* opens the file identified by *pathname*, returning a file descriptor used to refer to the open file in subsequent calls. If the file doesn't exist, *open()* may create it, depending on the settings of the *flags* bit-mask argument. The *flags* argument also specifies whether the file is to be opened for reading, writing, or both. The *mode* argument specifies the permissions to be placed on the file if it is created by this call. If the *open()* call is not being used to create a file, this argument is ignored and can be omitted.
- *numread* = *read(fd, buffer, count)* reads at most *count* bytes from the open file referred to by *fd* and stores them in *buffer*. The *read()* call returns the number of bytes actually read. If no further bytes could be read (i.e., end-of-file was encountered), *read()* returns 0.
- *numwritten* = *write(fd, buffer, count)* writes up to *count* bytes from *buffer* to the open file referred to by *fd*. The *write()* call returns the number of bytes actually written, which may be less than *count*.
- *status* = *close(fd)* is called after all I/O has been completed, in order to release the file descriptor *fd* and its associated kernel resources.

Before we launch into the details of these system calls, we provide a short demonstration of their use in Listing 4-1. This program is a simple version of the *cp(1)* command. It copies the contents of the existing file named in its first command-line argument to the new file named in its second command-line argument.

We can use the program in Listing 4-1 as follows:

```
$ ./copy oldfile newfile
```

Listing 4-1: Using I/O system calls

```
#include <sys/stat.h>
#include <fcntl.h>
#include "t1pi_hdr.h"

#ifndef BUF_SIZE          /* Allow "cc -D" to override definition */
#define BUF_SIZE 1024
#endif

int
main(int argc, char *argv[])
{
    int inputFd, outputFd, openFlags;
    mode_t filePerms;
    ssize_t numRead;
    char buf[BUF_SIZE];

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s old-file new-file\n", argv[0]);

    /* Open input and output files */

    inputFd = open(argv[1], O_RDONLY);
    if (inputFd == -1)
        errExit("opening file %s", argv[1]);

    openFlags = O_CREAT | O_WRONLY | O_TRUNC;
    filePerms = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
                S_IROTH | S_IWOTH; /* rw-rw-rw- */
    outputFd = open(argv[2], openFlags, filePerms);
    if (outputFd == -1)
        errExit("opening file %s", argv[2]);

    /* Transfer data until we encounter end of input or an error */

    while ((numRead = read(inputFd, buf, BUF_SIZE)) > 0)
        if (write(outputFd, buf, numRead) != numRead)
            fatal("couldn't write whole buffer");
    if (numRead == -1)
        errExit("read");

    if (close(inputFd) == -1)
        errExit("close input");
    if (close(outputFd) == -1)
        errExit("close output");

    exit(EXIT_SUCCESS);
}
```

fileio/copy.c

4.2 Universality of I/O

One of the distinguishing features of the UNIX I/O model is the concept of *universality of I/O*. This means that the same four system calls—*open()*, *read()*, *write()*, and *close()*—are used to perform I/O on all types of files, including devices such as terminals. Consequently, if we write a program using only these system calls, that program will work on any type of file. For example, the following are all valid uses of the program in Listing 4-1:

\$./copy test test.old	<i>Copy a regular file</i>
\$./copy a.txt /dev/tty	<i>Copy a regular file to this terminal</i>
\$./copy /dev/tty b.txt	<i>Copy input from this terminal to a regular file</i>
\$./copy /dev/pts/16 /dev/tty	<i>Copy input from another terminal</i>

Universality of I/O is achieved by ensuring that each file system and device driver implements the same set of I/O system calls. Because details specific to the file system or device are handled within the kernel, we can generally ignore device-specific factors when writing application programs. When access to specific features of a file system or device is required, a program can use the catchall *ioctl()* system call (Section 4.8), which provides an interface to features that fall outside the universal I/O model.

4.3 Opening a File: *open()*

The *open()* system call either opens an existing file or creates and opens a new file.

```
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags, ... /* mode_t mode */);

Returns file descriptor on success, or -1 on error
```

The file to be opened is identified by the *pathname* argument. If *pathname* is a symbolic link, it is dereferenced. On success, *open()* returns a file descriptor that is used to refer to the file in subsequent system calls. If an error occurs, *open()* returns -1 and *errno* is set accordingly.

The *flags* argument is a bit mask that specifies the *access mode* for the file, using one of the constants shown in Table 4-2.

Early UNIX implementations used the numbers 0, 1, and 2 instead of the names shown in Table 4-2. Most modern UNIX implementations define these constants to have those values. Thus, we can see that *O_RDWR* is not equivalent to *O_RDONLY* | *O_WRONLY*; the latter combination is a logical error.

When *open()* is used to create a new file, the *mode* bit-mask argument specifies the permissions to be placed on the file. (The *mode_t* data type used to type *mode* is an integer type specified in SUSv3.) If the *open()* call doesn't specify *O_CREAT*, *mode* can be omitted.

Table 4-2: File access modes

Access mode	Description
O_RDONLY	Open the file for reading only
O_WRONLY	Open the file for writing only
O_RDWR	Open the file for both reading and writing

We describe file permissions in detail in Section 15.4. Later, we'll see that the permissions actually placed on a new file depend not just on the *mode* argument, but also on the process *umask* (Section 15.4.6) and the (optionally present) default access control list (Section 17.6) of the parent directory. In the meantime, we'll just note that the *mode* argument can be specified as a number (typically in octal) or, preferably, by ORing (|) together zero or more of the bit-mask constants listed in Table 15-4, on page 295.

Listing 4-2 shows examples of the use of *open()*, some of which employ additional *flags* bits that we describe shortly.

Listing 4-2: Examples of the use of *open()*

```

/* Open existing file for reading */

fd = open("startup", O_RDONLY);
if (fd == -1)
    errExit("open");

/* Open new or existing file for reading and writing, truncating to zero
   bytes; file permissions read+write for owner, nothing for all others */

fd = open("myfile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
if (fd == -1)
    errExit("open");

/* Open new or existing file for writing; writes should always
   append to end of file */

fd = open("w.log", O_WRONLY | O_CREAT | O_TRUNC | O_APPEND,
          S_IRUSR | S_IWUSR);
if (fd == -1)
    errExit("open");

```

File descriptor number returned by *open()*

SUSv3 specifies that if *open()* succeeds, it is guaranteed to use the lowest-numbered unused file descriptor for the process. We can use this feature to ensure that a file is opened using a particular file descriptor. For example, the following sequence ensures that a file is opened using standard input (file descriptor 0).

```

if (close(STDIN_FILENO) == -1)      /* Close file descriptor 0 */
    errExit("close");

fd = open(pathname, O_RDONLY);
if (fd == -1)
    errExit("open");

```

Since file descriptor 0 is unused, *open()* is guaranteed to open the file using that descriptor. In Section 5.5, we look at the use of *dup2()* and *fcntl()* to achieve a similar result, but with more flexible control over the file descriptor used. In that section, we also show an example of why it can be useful to control the file descriptor on which a file is opened.

4.3.1 The *open()* flags Argument

In some of the example *open()* calls shown in Listing 4-2, we included other bits (*O_CREAT*, *O_TRUNC*, and *O_APPEND*) in *flags* in addition to the file access mode. We now consider the *flags* argument in more detail. Table 4-3 summarizes the full set of constants that can be bit-wise ORed (|) in *flags*. The final column indicates which of these constants are standardized in SUSv3 or SUSv4.

Table 4-3: Values for the *flags* argument of *open()*

Flag	Purpose	SUS?
<i>O_RDONLY</i>	Open for reading only	v3
<i>O_WRONLY</i>	Open for writing only	v3
<i>O_RDWR</i>	Open for reading and writing	v3
<i>O_CLOEXEC</i>	Set the close-on-exec flag (since Linux 2.6.23)	v4
<i>O_CREAT</i>	Create file if it doesn't already exist	v3
<i>O_DIRECT</i>	File I/O bypasses buffer cache	
<i>O_DIRECTORY</i>	Fail if <i>pathname</i> is not a directory	v4
<i>O_EXCL</i>	With <i>O_CREAT</i> : create file exclusively	v3
<i>O_LARGEFILE</i>	Used on 32-bit systems to open large files	
<i>O_NOATIME</i>	Don't update file last access time on <i>read()</i> (since Linux 2.6.8)	
<i>O_NOCTTY</i>	Don't let <i>pathname</i> become the controlling terminal	v3
<i>O_NOFOLLOW</i>	Don't dereference symbolic links	v4
<i>O_TRUNC</i>	Truncate existing file to zero length	v3
<i>O_APPEND</i>	Writes are always appended to end of file	v3
<i>O_ASYNC</i>	Generate a signal when I/O is possible	
<i>O_DSYNC</i>	Provide synchronized I/O data integrity (since Linux 2.6.33)	v3
<i>O_NONBLOCK</i>	Open in nonblocking mode	v3
<i>O_SYNC</i>	Make file writes synchronous	v3

The constants in Table 4-3 are divided into the following groups:

- *File access mode flags*: These are the `O_RDONLY`, `O_WRONLY`, and `O_RDWR` flags described earlier. They can be retrieved using the `fcntl()` `F_GETFL` operation (Section 5.3).
- *File creation flags*: These are the flags shown in the second part of Table 4-3. They control various aspects of the behavior of the `open()` call, as well as options for subsequent I/O operations. These flags can't be retrieved or changed.
- *Open file status flags*: These are the remaining flags in Table 4-3. They can be retrieved and modified using the `fcntl()` `F_GETFL` and `F_SETFL` operations (Section 5.3). These flags are sometimes simply called the *file status flags*.

Since kernel 2.6.22, the Linux-specific files in the directory `/proc/PID/fdinfo` can be read to obtain information about the file descriptors of any process on the system. There is one file in this directory for each of the process's open file descriptors, with a name that matches the number of the descriptor. The *pos* field in this file shows the current file offset (Section 4.7). The *flags* field is an octal number that shows the file access mode flags and open file status flags. (To decode this number, we need to look at the numeric values of these flags in the C library header files.)

Details for the *flags* constants are as follows:

`O_APPEND`

Writes are always appended to the end of the file. We discuss the significance of this flag in Section 5.1.

`O_ASYNC`

Generate a signal when I/O becomes possible on the file descriptor returned by `open()`. This feature, termed *signal-driven I/O*, is available only for certain file types, such as terminals, FIFOs, and sockets. (The `O_ASYNC` flag is not specified in SUSv3; however, it, or the older synonym, `FASYNC`, is found on most UNIX implementations.) On Linux, specifying the `O_ASYNC` flag when calling `open()` has no effect. To enable signal-driven I/O, we must instead set this flag using the `fcntl()` `F_SETFL` operation (Section 5.3). (Several other UNIX implementations behave similarly.) Refer to Section 63.3 for more information about the `O_ASYNC` flag.

`O_CLOEXEC` (since Linux 2.6.23)

Enable the close-on-exec flag (`FD_CLOEXEC`) for the new file descriptor. We describe the `FD_CLOEXEC` flag in Section 27.4. Using the `O_CLOEXEC` flag allows a program to avoid additional `fcntl()` `F_SETFD` and `F_SETFD` operations to set the close-on-exec flag. It is also necessary in multithreaded programs to avoid the race conditions that could occur using the latter technique. These races can occur when one thread opens a file descriptor and then tries to mark it close-on-exec at the same time as another thread does a `fork()` and then an `exec()` of an arbitrary program. (Suppose that the second thread manages to both `fork()` and `exec()` between the time the first thread opens the file descriptor and uses `fcntl()` to set the close-on-exec flag.) Such races

could result in open file descriptors being unintentionally passed to unsafe programs. (We say more about race conditions in Section 5.1.)

`O_CREAT`

If the file doesn't already exist, it is created as a new, empty file. This flag is effective even if the file is being opened only for reading. If we specify `O_CREAT`, then we must supply a *mode* argument in the *open()* call; otherwise, the permissions of the new file will be set to some random value from the stack.

`O_DIRECT`

Allow file I/O to bypass the buffer cache. This feature is described in Section 13.6. The `_GNU_SOURCE` feature test macro must be defined in order to make this constant definition available from `<fcntl.h>`.

`O_DIRECTORY`

Return an error (*errno* equals `ENOTDIR`) if *pathname* is not a directory. This flag is an extension designed specifically for implementing *opendir()* (Section 18.8). The `_GNU_SOURCE` feature test macro must be defined in order to make this constant definition available from `<fcntl.h>`.

`O_DSYNC` (since Linux 2.6.33)

Perform file writes according to the requirements of synchronized I/O data integrity completion. See the discussion of kernel I/O buffering in Section 13.3.

`O_EXCL`

This flag is used in conjunction with `O_CREAT` to indicate that if the file already exists, it should not be opened; instead, *open()* should fail, with *errno* set to `EEXIST`. In other words, this flag allows the caller to ensure that it is the process creating the file. The check for existence and the creation of the file are performed *atomically*. We discuss the concept of atomicity in Section 5.1. When both `O_CREAT` and `O_EXCL` are specified in *flags*, *open()* fails (with the error `EEXIST`) if *pathname* is a symbolic link. SUSv3 requires this behavior so that a privileged application can create a file in a known location without there being a possibility that a symbolic link would cause the file to be created in a different location (e.g., a system directory), which would have security implications.

`O_LARGEFILE`

Open the file with large file support. This flag is used on 32-bit systems in order to work with large files. Although it is not specified in SUSv3, the `O_LARGEFILE` flag is found on several other UNIX implementations. On 64-bit Linux implementations such as Alpha and IA-64, this flag has no effect. See Section 5.10 for more information.

`O_NOATIME` (since Linux 2.6.8)

Don't update the file last access time (the *st_atime* field described in Section 15.1) when reading from this file. To use this flag, the effective user ID of the calling process must match the owner of the file, or the process must be privileged (`CAP_FOWNER`); otherwise, *open()* fails with the error `EPERM`.

(In reality, for an unprivileged process, it is the process's file-system user ID, rather than its effective user ID, that must match the user ID of the file when opening a file with the `O_NOATIME` flag, as described in Section 9.5.) This flag is a nonstandard Linux extension. To expose its definition from `<fcntl.h>`, we must define the `_GNU_SOURCE` feature test macro. The `O_NOATIME` flag is intended for use by indexing and backup programs. Its use can significantly reduce the amount of disk activity, because repeated disk seeks back and forth across the disk are not required to read the contents of a file and to update the last access time in the file's i-node (Section 14.4). Functionality similar to `O_NOATIME` is available using the `MS_NOATIME` `mount()` flag (Section 14.8.1) and the `FS_NOATIME_FL` flag (Section 15.5).

`O_NOCTTY`

If the file being opened is a terminal device, prevent it from becoming the controlling terminal. Controlling terminals are discussed in Section 34.4. If the file being opened is not a terminal, this flag has no effect.

`O_NOFOLLOW`

Normally, `open()` dereferences *pathname* if it is a symbolic link. However, if the `O_NOFOLLOW` flag is specified, then `open()` fails (with *errno* set to `ELLOOP`) if *pathname* is a symbolic link. This flag is useful, especially in privileged programs, for ensuring that `open()` doesn't dereference a symbolic link. To expose the definition of this flag from `<fcntl.h>`, we must define the `_GNU_SOURCE` feature test macro.

`O_NONBLOCK`

Open the file in nonblocking mode. See Section 5.9.

`O_SYNC`

Open the file for synchronous I/O. See the discussion of kernel I/O buffering in Section 13.3.

`O_TRUNC`

If the file already exists and is a regular file, then truncate it to zero length, destroying any existing data. On Linux, truncation occurs whether the file is being opened for reading or writing (in both cases, we must have write permission on the file). SUSv3 leaves the combination of `O_RDONLY` and `O_TRUNC` unspecified, but most other UNIX implementations behave in the same way as Linux.

4.3.2 Errors from `open()`

If an error occurs while trying to open the file, `open()` returns `-1`, and *errno* identifies the cause of the error. The following are some possible errors that can occur (in addition to those already noted when describing the *flags* argument above):

`EACCES`

The file permissions don't allow the calling process to open the file in the mode specified by *flags*. Alternatively, because of directory permissions, the file could not be accessed, or the file did not exist and could not be created.

EISDIR

The specified file is a directory, and the caller attempted to open it for writing. This isn't allowed. (On the other hand, there are occasions when it can be useful to open a directory for reading. We consider an example in Section 18.11.)

EMFILE

The process resource limit on the number of open file descriptors has been reached (RLIMIT_NOFILE, described in Section 36.3).

ENFILE

The system-wide limit on the number of open files has been reached.

ENOENT

The specified file doesn't exist, and `O_CREAT` was not specified, or `O_CREAT` was specified, and one of the directories in *pathname* doesn't exist or is a symbolic link pointing to a nonexistent pathname (a dangling link).

EROFS

The specified file is on a read-only file system and the caller tried to open it for writing.

ETXTBSY

The specified file is an executable file (a program) that is currently executing. It is not permitted to modify (i.e., open for writing) the executable file associated with a running program. (We must first terminate the program in order to be able to modify the executable file.)

When we later describe other system calls or library functions, we generally won't list the range of possible errors that may occur in the above fashion. (Such a list can be found in the corresponding manual page for each system call or library function.) We do so here for two reasons. One of these is that *open()* is the first system call that we describe in detail, and the above list illustrates that a system call or library function may fail for any of a number of reasons. Second, the specific reasons why *open()* may fail make an interesting list in themselves, illustrating a number of factors and checks that come into play when a file is accessed. (The above list is incomplete: see the *open(2)* manual page for more reasons why *open()* may fail.)

4.3.3 The *creat()* System Call

In early UNIX implementations, *open()* had only two arguments and could not be used to create a new file. Instead, the *creat()* system call was used to create and open a new file.

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

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

Returns file descriptor, or -1 on error

The *creat()* system call creates and opens a new file with the given *pathname*, or if the file already exists, opens the file and truncates it to zero length. As its function

result, `creat()` returns a file descriptor that can be used in subsequent system calls. Calling `creat()` is equivalent to the following `open()` call:

```
fd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

Because the `open()` *flags* argument provides greater control over how the file is opened (e.g., we can specify `O_RDWR` instead of `O_WRONLY`), `creat()` is now obsolete, although it may still be seen in older programs.

4.4 Reading from a File: `read()`

The `read()` system call reads data from the open file referred to by the descriptor `fd`.

```
#include <unistd.h>
```

```
ssize_t read(int fd, void *buffer, size_t count);
```

Returns number of bytes read, 0 on EOF, or -1 on error

The *count* argument specifies the maximum number of bytes to read. (The *size_t* data type is an unsigned integer type.) The *buffer* argument supplies the address of the memory buffer into which the input data is to be placed. This buffer must be at least *count* bytes long.

System calls don't allocate memory for buffers that are used to return information to the caller. Instead, we must pass a pointer to a previously allocated memory buffer of the correct size. This contrasts with several library functions that *do* allocate memory buffers in order to return information to the caller.

A successful call to `read()` returns the number of bytes actually read, or 0 if end-of-file is encountered. On error, the usual -1 is returned. The *ssize_t* data type is a signed integer type used to hold a byte count or a -1 error indication.

A call to `read()` may read less than the requested number of bytes. For a regular file, the probable reason for this is that we were close to the end of the file.

When `read()` is applied to other types of files—such as pipes, FIFOs, sockets, or terminals—there are also various circumstances where it may read fewer bytes than requested. For example, by default, a `read()` from a terminal reads characters only up to the next newline (`\n`) character. We consider these cases when we cover other file types in subsequent chapters.

Using `read()` to input a series of characters from, say, a terminal, we might expect the following code to work:

```
#define MAX_READ 20
char buffer[MAX_READ];

if (read(STDIN_FILENO, buffer, MAX_READ) == -1)
    errExit("read");
printf("The input data was: %s\n", buffer);
```

The output from this piece of code is likely to be strange, since it will probably include characters in addition to the string actually entered. This is because `read()`

doesn't place a terminating null byte at the end of the string that *printf()* is being asked to print. A moment's reflection leads us to realize that this must be so, since *read()* can be used to read any sequence of bytes from a file. In some cases, this input might be text, but in other cases, the input might be binary integers or C structures in binary form. There is no way for *read()* to tell the difference, and so it can't attend to the C convention of null terminating character strings. If a terminating null byte is required at the end of the input buffer, we must put it there explicitly:

```
char buffer[MAX_READ + 1];
ssize_t numRead;

numRead = read(STDIN_FILENO, buffer, MAX_READ);
if (numRead == -1)
    errExit("read");

buffer[numRead] = '\0';
printf("The input data was: %s\n", buffer);
```

Because the terminating null byte requires a byte of memory, the size of *buffer* must be at least one greater than the largest string we expect to read.

4.5 Writing to a File: *write()*

The *write()* system call writes data to an open file.

```
#include <unistd.h>

ssize_t write(int fd, void *buffer, size_t count);
```

Returns number of bytes written, or -1 on error

The arguments to *write()* are similar to those for *read()*: *buffer* is the address of the data to be written; *count* is the number of bytes to write from *buffer*; and *fd* is a file descriptor referring to the file to which data is to be written.

On success, *write()* returns the number of bytes actually written; this may be less than *count*. For a disk file, possible reasons for such a *partial write* are that the disk was filled or that the process resource limit on file sizes was reached. (The relevant limit is `RLIMIT_FSIZE`, described in Section 36.3.)

When performing I/O on a disk file, a successful return from *write()* doesn't guarantee that the data has been transferred to disk, because the kernel performs buffering of disk I/O in order to reduce disk activity and expedite *write()* calls. We consider the details in Chapter 13.

4.6 Closing a File: *close()*

The *close()* system call closes an open file descriptor, freeing it for subsequent reuse by the process. When a process terminates, all of its open file descriptors are automatically closed.

```
#include <unistd.h>
```

```
int close(int fd);
```

Returns 0 on success, or -1 on error

It is usually good practice to close unneeded file descriptors explicitly, since this makes our code more readable and reliable in the face of subsequent modifications. Furthermore, file descriptors are a consumable resource, so failure to close a file descriptor could result in a process running out of descriptors. This is a particularly important issue when writing long-lived programs that deal with multiple files, such as shells or network servers.

Just like every other system call, a call to *close()* should be bracketed with error-checking code, such as the following:

```
if (close(fd) == -1)
    errExit("close");
```

This catches errors such as attempting to close an unopened file descriptor or close the same file descriptor twice, and catches error conditions that a specific file system may diagnose during a close operation.

NFS (Network File System) provides an example of an error that is specific to a file system. If an NFS commit failure occurs, meaning that the data did not reach the remote disk, then this error is propagated to the application as a failure in the *close()* call.

4.7 Changing the File Offset: *lseek()*

For each open file, the kernel records a *file offset*, sometimes also called the *read-write offset* or *pointer*. This is the location in the file at which the next *read()* or *write()* will commence. The file offset is expressed as an ordinal byte position relative to the start of the file. The first byte of the file is at offset 0.

The file offset is set to point to the start of the file when the file is opened and is automatically adjusted by each subsequent call to *read()* or *write()* so that it points to the next byte of the file after the byte(s) just read or written. Thus, successive *read()* and *write()* calls progress sequentially through a file.

The *lseek()* system call adjusts the file offset of the open file referred to by the file descriptor *fd*, according to the values specified in *offset* and *whence*.

```
#include <unistd.h>
```

```
off_t lseek(int fd, off_t offset, int whence);
```

Returns new file offset if successful, or -1 on error

The *offset* argument specifies a value in bytes. (The *off_t* data type is a signed integer type specified by SUSv3.) The *whence* argument indicates the base point from which *offset* is to be interpreted, and is one of the following values:

SEEK_SET

The file offset is set *offset* bytes from the beginning of the file.

SEEK_CUR

The file offset is adjusted by *offset* bytes relative to the current file offset.

SEEK_END

The file offset is set to the size of the file plus *offset*. In other words, *offset* is interpreted with respect to the next byte after the last byte of the file.

Figure 4-1 shows how the *whence* argument is interpreted.

In earlier UNIX implementations, the integers 0, 1, and 2 were used, rather than the SEEK_* constants shown in the main text. Older versions of BSD used different names for these values: L_SET, L_INCR, and L_XTND.

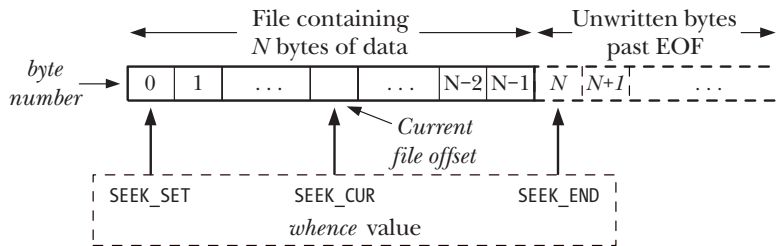


Figure 4-1: Interpreting the *whence* argument of *lseek()*

If *whence* is SEEK_CUR or SEEK_END, *offset* may be negative or positive; for SEEK_SET, *offset* must be nonnegative.

The return value from a successful *lseek()* is the new file offset. The following call retrieves the current location of the file offset without changing it:

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

Some UNIX implementations (but not Linux) have the nonstandard *tell(fd)* function, which serves the same purpose as the above *lseek()* call.

Here are some other examples of *lseek()* calls, along with comments indicating where the file offset is moved to:

```
lseek(fd, 0, SEEK_SET);      /* Start of file */
lseek(fd, 0, SEEK_END);      /* Next byte after the end of the file */
lseek(fd, -1, SEEK_END);     /* Last byte of file */
lseek(fd, -10, SEEK_CUR);    /* Ten bytes prior to current location */
lseek(fd, 10000, SEEK_END);  /* 10001 bytes past last byte of file */
```

Calling *lseek()* simply adjusts the kernel's record of the file offset associated with a file descriptor. It does not cause any physical device access.

We describe some further details of the relationship between file offsets, file descriptors, and open files in Section 5.4.

We can't apply *lseek()* to all types of files. Applying *lseek()* to a pipe, FIFO, socket, or terminal is not permitted; *lseek()* fails, with *errno* set to *ESPIPE*. On the other hand, it is possible to apply *lseek()* to devices where it is sensible to do so. For example, it is possible to seek to a specified location on a disk or tape device.

The *l* in the name *lseek()* derives from the fact that the *offset* argument and the return value were both originally typed as *long*. Early UNIX implementations provided a *seek()* system call, which typed these values as *int*.

File holes

What happens if a program seeks past the end of a file, and then performs I/O? A call to *read()* will return 0, indicating end-of-file. Somewhat surprisingly, it is possible to write bytes at an arbitrary point past the end of the file.

The space in between the previous end of the file and the newly written bytes is referred to as a *file hole*. From a programming point of view, the bytes in a hole exist, and reading from the hole returns a buffer of bytes containing 0 (null bytes).

File holes don't, however, take up any disk space. The file system doesn't allocate any disk blocks for a hole until, at some later point, data is written into it. The main advantage of file holes is that a sparsely populated file consumes less disk space than would otherwise be required if the null bytes actually needed to be allocated in disk blocks. Core dump files (Section 22.1) are common examples of files that contain large holes.

The statement that file holes don't consume disk space needs to be qualified slightly. On most file systems, file space is allocated in units of blocks (Section 14.3). The size of a block depends on the file system, but is typically something like 1024, 2048, or 4096 bytes. If the edge of a hole falls within a block, rather than on a block boundary, then a complete block is allocated to store the data in the other part of the block, and the part corresponding to the hole is filled with null bytes.

Most native UNIX file systems support the concept of file holes, but many nonnative file systems (e.g., Microsoft's VFAT) do not. On a file system that doesn't support holes, explicit null bytes are written to the file.

The existence of holes means that a file's nominal size may be larger than the amount of disk storage it utilizes (in some cases, considerably larger). Writing bytes into the middle of the file hole will decrease the amount of free disk space as the kernel allocates blocks to fill the hole, even though the file's size doesn't change. Such a scenario is uncommon, but nevertheless one to be aware of.

SUSv3 specifies a function, *posix_fallocate(fd, offset, len)*, that ensures that space is allocated on disk for the byte range specified by *offset* and *len* for the disk file referred to by the descriptor *fd*. This allows an application to be sure that a later *write()* to the file won't fail because disk space is exhausted (which could otherwise occur if a hole in the file was filled in, or some other application consumed space on the disk). Historically, the *glibc* implementation of this function achieved the desired result by writing a 0 byte into each block in the specified range. Since version 2.6.23, Linux provides an *fallocate()* system call, which provides a more efficient way of ensuring that the necessary space is allocated, and the *glibc* *posix_fallocate()* implementation makes use of this system call when it is available.