

Programming with POSIX[®] Threads

David R. Butenhof



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

*Programming
with
POSIX[®] Threads*

Addison-Wesley Professional Computing Series

Brian W. Kernighan, Consulting Editor

- Matthew H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*
- David R. Butenhof, *Programming with POSIX® Threads*
- Brent Callaghan, *NFS Illustrated*
- Tom Cargill, *C++ Programming Style*
- William R. Cheswick/Steven M. Bellovin/Aviel D. Rubin, *Firewalls and Internet Security, Second Edition: Repelling the Wily Hacker*
- David A. Curry, *UNIX® System Security: A Guide for Users and System Administrators*
- Stephen C. Dewhurst, *C++ Gotchas: Avoiding Common Problems in Coding and Design*
- Dan Farmer/Wietse Venema, *Forensic Discovery*
- Erich Gamma/Richard Helm/Ralph Johnson/John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*
- Erich Gamma/Richard Helm/Ralph Johnson/John Vlissides, *Design Patterns CD: Elements of Reusable Object-Oriented Software*
- Peter Hagggar, *Practical Java™ Programming Language Guide*
- David R. Hanson, *C Interfaces and Implementations: Techniques for Creating Reusable Software*
- Mark Harrison/Michael McLennan, *Effective Tcl/Tk Programming: Writing Better Programs with Tcl and Tk*
- Michi Henning/Steve Vinoski, *Advanced CORBA® Programming with C++*
- Brian W. Kernighan/Rob Pike, *The Practice of Programming*
- S. Keshav, *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*
- John Lakos, *Large-Scale C++ Software Design*
- Scott Meyers, *Effective C++ CD: 85 Specific Ways to Improve Your Programs and Designs*
- Scott Meyers, *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*
- Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*
- Scott Meyers, *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*
- Robert B. Murray, *C++ Strategies and Tactics*
- David R. Musser/Gillmer J. Derge/Atul Saini, *STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library*
- John K. Ousterhout, *Tcl and the Tk Toolkit*
- Craig Partridge, *Gigabit Networking*
- Radia Perlman, *Interconnections, Second Edition: Bridges, Routers, Switches, and Internetworking Protocols*
- Stephen A. Rago, *UNIX® System V Network Programming*
- Eric S. Raymond, *The Art of UNIX Programming*
- Marc J. Rochkind, *Advanced UNIX Programming, Second Edition*
- Curt Schimmel, *UNIX® Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*
- W. Richard Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*
- W. Richard Stevens, *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX® Domain Protocols*
- W. Richard Stevens/Bill Fenner/Andrew M. Rudoff, *UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API*
- W. Richard Stevens/Stephen A. Rago, *Advanced Programming in the UNIX® Environment, Second Edition*
- W. Richard Stevens/Gary R. Wright, *TCP/IP Illustrated Volumes 1-3 Boxed Set*
- John Viega/Gary McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*
- Gary R. Wright/W. Richard Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*
- Ruixi Yuan/W. Timothy Strayer, *Virtual Private Networks: Technologies and Solutions*

Visit www.awprofessional.com/series/professionalcomputing for more information about these titles.

*Programming
with
POSIX[®] Threads*

David R. Butenhof

◆ Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sidney • Tokyo • Singapore • Mexico City

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

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

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, please contact:

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

For sales outside of the U.S., please contact:

International Sales
international@pearsontechgroup.com

Visit Addison-Wesley on the Web at www.awprofessional.com

Library of Congress Cataloging-in-Publication Data

Butenhof, David R., 1956–

Programming with POSIX threads / David R. Butenhof.
p. cm.—(Addison-Wesley professional computing series)

Includes bibliographical references and index.

ISBN 0-201-63392-2 (pbk.)

1. Threads (Computer programs) 2. POSIX (Computer software standard) 3. Electronic digital computers—Programming. I. Title. II. Series.

QA76.76.T55B88 1997

005.4'32—dc21

97-6635

CIP

Copyright © 1997 by Addison-Wesley

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.
Rights and Contracts Department
75 Arlington Street, Suite 300
Boston, MA 02116
Fax: (617) 848-7047

Trademark acknowledgments: UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Ltd. Digital, DEC, Digital UNIX, DECthreads, VMS, and OpenVMS are trademarks of Digital Equipment Corporation. Solaris, SPARC, SunOS, and Sun are trademarks of Sun Microsystems Incorporated. SGI and IRIX are trademarks of Silicon Graphics, Incorporated. HP-UX is a trademark of Hewlett-Packard Company. AIX, IBM, and OS/2 are trademarks or registered trademarks of the IBM Corporation. X/Open is a trademark of X/Open Company Ltd. POSIX is a registered trademark of the Institute of Electrical and Electronics Engineers, Inc.

ISBN 0201633922

Text printed in the United States on recycled paper at RR Donnelley Crawfordsville in Crawfordsville, Indiana.
21st Printing December 2008

*To Anne, Amy,
and
Alyssa.*

Quote acknowledgments:

American Heritage Dictionary of the English Language: page 1.

ISO/IEC 9945-1:1996, © 1996 by IEEE: page 29.

Lewis Carroll, *Alice's Adventures in Wonderland*: pages xv, 47, 70, 88, 97, 98, 106, 131, 142, 161, 189, 197, 241. Reproduced by permission of Macmillan Children's Books.

Lewis Carroll, *Through the Looking-Glass*: pages 1, 4, 8, 20, 25, 29, 35, 45, 172, 214, 283, 290, 302. Reproduced by permission of Macmillan Children's Books.

Lewis Carroll, *The Hunting of the Snark*: pages 3, 13, 28, 39, 120, 131, 134, 289, 367. Reproduced by permission of Macmillan Children's Books.

Contents

<i>List of Example Programs</i>	xii
<i>Preface</i>	xv
Intended audience	xvi
About the author	xvi
Acknowledgments	xvii
1 Introduction	1
1.1 <i>The “bailing programmers”</i>	3
1.2 <i>Definitions and terminology</i>	4
1.2.1 Asynchronous	4
1.2.2 Concurrency	4
1.2.3 Uniprocessor and multiprocessor	5
1.2.4 Parallelism	5
1.2.5 Thread safety and reentrancy	6
1.2.6 Concurrency control functions.	7
1.3 <i>Asynchronous programming is intuitive...</i>	8
1.3.1 . . . because UNIX is asynchronous	9
1.3.2 . . . because the world is asynchronous	11
1.4 <i>About the examples in this book</i>	12
1.5 <i>Asynchronous programming, by example</i>	13
1.5.1 The baseline, synchronous version	14
1.5.2 A version using multiple processes	15
1.5.3 A version using multiple threads	17
1.5.4 Summary	19
1.6 <i>Benefits of threading</i>	20
1.6.1 Parallelism	20
1.6.2 Concurrency	22
1.6.3 Programming model.	24
1.7 <i>Costs of threading</i>	25
1.7.1 Computing overhead	26
1.7.2 Programming discipline	26
1.7.3 Harder to debug	27
1.8 <i>To thread or not to thread?</i>	28
1.9 <i>POSIX thread concepts</i>	29
1.9.1 Architectural overview	30
1.9.2 Types and interfaces	30

1.9.3	Checking for errors.	31
2	Threads.	35
2.1	Creating and using threads	35
2.2	The life of a thread	39
2.2.1	Creation	40
2.2.2	Startup	41
2.2.3	Running and blocking	42
2.2.4	Termination	43
2.2.5	Recycling	44
3	Synchronization	45
3.1	Invariants, critical sections, and predicates	45
3.2	Mutexes.	47
3.2.1	Creating and destroying a mutex	49
3.2.2	Locking and unlocking a mutex	52
3.2.2.1	Nonblocking mutex locks	58
3.2.3	Using mutexes for atomicity	61
3.2.4	Sizing a mutex to fit the job	62
3.2.5	Using more than one mutex	63
3.2.5.1	Lock hierarchy	63
3.2.5.2	Lock chaining.	70
3.3	Condition variables.	70
3.3.1	Creating and destroying a condition variable	74
3.3.2	Waiting on a condition variable.	77
3.3.3	Waking condition variable waiters.	81
3.3.4	One final alarm program	82
3.4	Memory visibility between threads.	88
4	A few ways to use threads	97
4.1	Pipeline	98
4.2	Work Crew.	106
4.3	Client/Server.	120
5	Advanced threaded programming	131
5.1	One-time initialization.	131
5.2	Attributes objects	134
5.2.1	Mutex attributes.	135
5.2.2	Condition variable attributes	137
5.2.3	Thread attributes	138
5.3	Cancellation	142
5.3.1	Deferred cancelability.	147
5.3.2	Asynchronous cancelability	150

5.3.3	Cleaning up	154
5.4	<i>Thread-specific data</i>	161
5.4.1	Creating thread-specific data	163
5.4.2	Using thread-specific data	166
5.4.3	Using destructor functions	167
5.5	<i>Realtime scheduling</i>	172
5.5.1	POSIX realtime options	173
5.5.2	Scheduling policies and priorities	174
5.5.3	Contention scope and allocation domain	181
5.5.4	Problems with realtime scheduling	183
5.5.5	Priority-aware mutexes	185
5.5.5.1	Priority ceiling mutexes	186
5.5.5.2	Priority inheritance mutexes	188
5.6	<i>Threads and kernel entities</i>	189
5.6.1	Many-to-one (user level).	190
5.6.2	One-to-one (kernel level)	191
5.6.3	Many-to-few (two level)	193
6	POSIX adjusts to threads	197
6.1	<i>fork</i>	197
6.1.1	Fork handlers	199
6.2	<i>exec</i>	204
6.3	<i>Process exit</i>	204
6.4	<i>Stdio</i>	204
6.4.1	flockfile and funlockfile	205
6.4.2	getchar_unlocked and putchar_unlocked.	207
6.5	<i>Thread-safe functions</i>	209
6.5.1	User and terminal identification	210
6.5.2	Directory searching	212
6.5.3	String token	212
6.5.4	Time representation	212
6.5.5	Random number generation	213
6.5.6	Group and user database	213
6.6	<i>Signals</i>	214
6.6.1	Signal actions	215
6.6.2	Signal masks.	216
6.6.3	pthread_kill.	217
6.6.4	sigwait and sigwaitinfo	227
6.6.5	SIGEV_THREAD	230
6.6.6	Semaphores: synchronizing with a signal-catching function	234
7	“Real code”	241

7.1	<i>Extended synchronization</i>	241
7.1.1	Barriers	242
7.1.2	Read-write locks.	253
7.2	<i>Work queue manager</i>	270
7.3	<i>But what about existing libraries?</i>	283
7.3.1	Modifying libraries to be thread-safe.	284
7.3.2	Living with legacy libraries	285
8	Hints to avoid debugging.	289
8.1	<i>Avoiding incorrect code</i>	290
8.1.1	Avoid relying on “thread inertia”	291
8.1.2	Never bet your mortgage on a thread race.	293
8.1.3	Cooperate to avoid deadlocks	297
8.1.4	Beware of priority inversion	299
8.1.5	Never share condition variables between predicates	300
8.1.6	Sharing stacks and related memory corrupters	301
8.2	<i>Avoiding performance problems</i>	302
8.2.1	Beware of concurrent serialization	302
8.2.2	Use the right number of mutexes	303
8.2.2.1	Too many mutexes will not help	304
8.2.3	Never fight over cache lines.	304
9	POSIX threads mini-reference	307
9.1	<i>POSIX 1003.1c-1995 options</i>	307
9.2	<i>POSIX 1003.1c-1995 limits</i>	308
9.3	<i>POSIX 1003.1c-1995 interfaces</i>	309
9.3.1	Error detection and reporting	310
9.3.2	Use of void* type.	311
9.3.3	Threads	311
9.3.4	Mutexes	316
9.3.5	Condition variables.	319
9.3.6	Cancellation.	323
9.3.7	Thread-specific data.	325
9.3.8	Realtime scheduling.	326
9.3.9	Fork handlers.	336
9.3.10	<i>Stdio</i>	336
9.3.11	Thread-safe functions.	338
9.3.12	Signals.	342
9.3.13	Semaphores	345
10	Future standardization	347
10.1	<i>X/Open XSH5 [UNIX98]</i>	347

10.1.1	POSIX options for XSH5	348
10.1.2	Mutex type	349
10.1.3	Set concurrency level.	351
10.1.4	Stack guard size	353
10.1.5	Parallel I/O	354
10.1.6	Cancellation points	355
10.2	<i>POSIX 1003.1j</i>	356
10.2.1	Barriers.	358
10.2.2	Read-write locks	358
10.2.3	Spinlocks	359
10.2.4	Condition variable wait clock.	359
10.2.5	Thread abort	361
10.3	<i>POSIX 1003.14</i>	361
	<i>Bibliography</i>	363
	<i>Thread resources on the Internet</i>	367
	<i>Index</i>	369

Example programs

sample.c part 1 sample info	13
alarm.c	14
alarm_fork.c	15
alarm_thread.c part 1 definitions	17
alarm_thread.c part 2 alarm_thread	18
alarm_thread.c part 3 main	18
thread_error.c	32
errors.h	33
lifecycle.c	38
mutex_static.c	50
mutex_dynamic.c	50
alarm_mutex.c part 1 definitions	52
alarm_mutex.c part 2 alarm_thread	54
alarm_mutex.c part 3 main	56
trylock.c	59
backoff.c	66
cond_static.c	74
cond_dynamic.c	75
cond.c	78
alarm_cond.c part 1 declarations	83
alarm_cond.c part 2 alarm_insert	84
alarm_cond.c part 3 alarm_routine	86
alarm_cond.c part 4 main	87
pipe.c part 1 definitions	99
pipe.c part 2 pipe_send	100
pipe.c part 3 pipe_stage	101
pipe.c part 4 pipe_create	102
pipe.c part 5 pipe_start, pipe_result	104
pipe.c part 6 main	105
crew.c part 1 definitions	108
crew.c part 2 worker_routine	110
crew.c part 3 crew_create	115
crew.c part 4 crew_start	117
crew.c part 5 main	119
server.c part 1 definitions	121
server.c part 2 tty_server_routine	123
server.c part 3 tty_server_request	125
server.c part 4 client_routine	127

server.c part 5 main.....	129
once.c	133
mutex_attr.c.....	136
cond_attr.c.....	137
thread_attr.c.....	140
cancel.c	145
cancel_disable.c	149
cancel_async.c	152
cancel_cleanup.c	156
cancel_subcontract.c.....	158
tsd_once.c.....	164
tsd_destructor.c	169
sched_attr.c.....	176
sched_thread.c.....	179
atfork.c	201
flock.c	205
putchar.c	208
getlogin.c.....	211
susp.c part 1 signal-catchingfunctions.....	218
susp.c part 2 initialization	220
susp.c part 3 thd_suspend.....	221
susp.c part 4 thd_continue	223
susp.c part 5 sampleprogram	225
sigwait.c	228
sigev_thread.c.....	232
semaphore_signal.c	238
barrier.h part 1 barrier_t	245
barrier.h part 2 interfaces	245
barrier.c part 1 barrier_init.....	246
barrier.c part 2 barrier_destroy.....	247
barrier.c part 3 barrier_wait.....	250
barrier_main.c.....	251
rwlock.h part 1 rwlock_t.....	255
rwlock.h part 2 interfaces	257
rwlock.c part 1 rwl_init.....	257
rwlock.c part 2 rwl_destroy	259
rwlock.c part 3 cleanup handlers.....	260
rwlock.c part 4 rwl_readlock	261
rwlock.c part 5 rwl_readtrylock.....	262
rwlock.c part 6 rwl_readunlock.....	263
rwlock.c part 7 rwl_writelock.....	263
rwlock.c part 8 rwl_writetrylock.....	264
rwlock.c part 9 rwl_writeunlock.....	265
rwlock_main.c.....	266
workq.h part 1 workq_t.....	271

workq.h part 2 interfaces.....	272
workq.c part 1 workq_init.....	272
workq.c part 2 workq_destroy.....	274
workq.c part 3 workq_add.....	275
workq.c part 4 workq_server.....	277
workq_main.c	280
inertia.c.....	291

Preface

*The White Rabbit put on his spectacles,
“Where shall I begin, please your Majesty?” he asked.
“Begin at the beginning,” the King said, very gravely,
“and go on till you come to the end: then stop.”*

—Lewis Carroll, *Alice’s Adventures in Wonderland*

This book is about “threads” and how to use them. “Thread” is just a name for a basic software “thing” that can do work on a computer. A thread is smaller, faster, and more maneuverable than a traditional process. In fact, once threads have been added to an operating system, a “process” becomes just data—address space, files, and so forth—plus one or more threads that do something with all that data.

With threads, you can build applications that utilize system resources more efficiently, that are more friendly to users, that run blazingly fast on multiprocessors, and that may even be easier to maintain. To accomplish all this, you need only add some relatively simple function calls to your code, adjust to a new way of thinking about programming, and leap over a few yawning chasms. Reading this book carefully will, I hope, help you to accomplish all that without losing your sense of humor.

The threads model used in this book is commonly called “Pthreads,” or “POSIX threads.” Or, more formally (since you haven’t yet been properly introduced), the POSIX 1003.1c–1995 standard. I’ll give you a few other names later—but for now, “Pthreads” is all you need to worry about.

Pthreads interfaces are included with Sun’s Solaris; Hewlett-Packard’s Tru64 UNIX, OpenVMS, NonStop platform, and HP-UX; IBM’s AIX, OS/400, and OS/390; SGI’s IRIX; SCO’s UnixWare; Apple’s Mac OS X; and Linux (any major distribution). There’s even an Open Source emulation package that allows you to use portable Pthread interfaces on Win32 systems.

In the personal computer market, Microsoft’s Win32 API (the primary programming interface to both Windows NT and Windows 95) supports threaded programming, as does IBM’s OS/2. These threaded programming models are quite different from Pthreads, but the important first step toward using them productively is understanding concurrency, synchronization, and scheduling. The rest is (more or less) a matter of syntax and style, and an experienced thread programmer can adapt to any of these models.

The threaded model can be (and has been) applied with great success to a wide range of programming problems. Here are just a few:

- Large scale, computationally intensive programs
- High-performance application programs and library code that can take advantage of multiprocessor systems
- Library code that can be used by threaded application programs
- Realtime application programs and library code
- Application programs and library code that perform I/O to slow external devices (such as networks and human beings).

Intended audience

This book assumes that you are an experienced programmer, familiar with developing code for an operating system in “the UNIX family” using the ANSI C language. I have tried not to assume that you have any experience with threads or other forms of asynchronous programming. The *Introduction* chapter provides a general overview of the terms and concepts you’ll need for the rest of the book. If you don’t want to read the Introduction first, that’s fine, but if you ever feel like you’re “missing something” you might try skipping back to get introduced.

Along the way you’ll find examples and simple analogies for everything. In the end I hope that you’ll be able to continue comfortably threading along on your own. Have fun, and “happy threading.”

About the author

I have been involved in the Pthreads standard since it began, although I stayed at home for the first few meetings. I was finally forced to spend a grueling week in the avalanche-proof concrete bunker at the base of Snowbird ski resort in Utah, watching hard-working standards representatives from around the world wax their skis. This was very distracting, because I had expected a standards meeting to be a formal and stuffy environment. As a result of this misunderstanding, I was forced to rent ski equipment instead of using my own.

After the Pthreads standard went into balloting, I worked on additional thread synchronization interfaces and multiprocessor issues with several POSIX working groups. I also helped to define the Aspen threads extensions, which were fast-tracked into X/Open XSH5.

I have worked at Digital Equipment Corporation for (mumble, mumble) years, in various locations throughout Massachusetts and New Hampshire. I was one of the creators of Digital’s own threading architecture, and I designed (and implemented much of) the Pthreads interfaces on Digital UNIX 4.0. I have been helping people develop and debug threaded code for more than eight years.

My unofficial motto is “Better Living Through Concurrency.” Threads are not sliced bread, but then, we’re programmers, not bakers, so we do what we can.

Acknowledgments

This is the part where I write the stuff that I’d like to see printed, and that my friends and coworkers want to see. You probably don’t care, and I promise not to be annoyed if you skip over it—but if you’re curious, by all means read on.

No project such as this book can truly be accomplished by a single person, despite the fact that only one name appears on the cover. I could have written a book about threads without any help—I know a great deal about threads, and I am at least reasonably competent at written communication. However, the result would not have been *this* book, and *this* book is better than that hypothetical work could possibly have been.

Thanks first and foremost to my manager Jean Fullerton, who gave me the time and encouragement to write this book on the job—and thanks to the rest of the DECthreads team who kept things going while I wrote, including Brian Keane, Webb Scales, Jacqueline Berg, Richard Love, Peter Portante, Brian Silver, Mark Simons, and Steve Johnson.

Thanks to Garret Swart who, while he was with Digital at the Systems Research Center, got us involved with POSIX. Thanks to Nawaf Bitar who worked with Garret to create, literally overnight, the first draft of what became Pthreads, and who became *POSIX thread evangelist* through the difficult period of getting everyone to understand just what the heck this threading thing was all about anyway. Without Garret, and especially Nawaf, Pthreads might not exist, and certainly wouldn’t be as good as it is. (The lack of perfection is not their responsibility—that’s the way life is.)

Thanks to everyone who contributed to the design of cma, Pthreads, UNIX98, and to the users of DCE threads and DECthreads, for all the help, thought-provoking discourse, and assorted skin-thickening exercises, including Andrew Birrell, Paul Borman, Bob Conti, Bill Cox, Jeff Denham, Peter Gilbert, Rick Greer, Mike Grier, Kevin Harris, Ken Hobday, Mike Jones, Steve Kleiman, Bob Knighten, Leslie Lamport, Doug Locke, Paula Long, Finnbar P. Murphy, Bill Noyce, Simon Patience, Harold Seigel, Al Simons, Jim Woodward, and John Zolnowsky.

Many thanks to all those who patiently reviewed the drafts of this book (and even to those who didn’t seem so patient at times). Brian Kernighan, Rich Stevens, Dave Brownell, Bill Gallmeister, Ilan Ginzburg, Will Morse, Bryan O’Sullivan, Bob Robillard, Dave Ruddock, Bil Lewis, and many others suggested or motivated improvements in structure and detail—and provided additional skin-thickening exercises to keep me in shape. Devang Shah and Bart Smaalders answered some Solaris questions, and Bryan O’Sullivan suggested what became the “bailing programmers” analogy.

Thanks to John Wait and Lana Langlois at Addison Wesley Longman, who waited with great patience as a first-time writer struggled to balance writing a book with engineering and consulting commitments. Thanks to Pamela Yee and Erin Sweeney, who managed the book's production process, and to all the team (many of whose names I'll never know), who helped.

Thanks to my wife, Anne Lederhos, and our daughters Amy and Alyssa, for all the things for which any writers may thank their families, including support, tolerance, and just being there. And thanks to Charles Dodgson (Lewis Carroll), who wrote extensively about threaded programming (and nearly everything else) in his classic works *Alice's Adventures in Wonderland*, *Through the Looking-Glass*, and *The Hunting of the Snark*.

Dave Butenhof
Digital Equipment Corporation
110 Spit Brook Road, ZKO2-3/Q18
Nashua, NH 03062
butenhof@zko.dec.com
December 1996

3 Synchronization

“That’s right!” said the Tiger-lily. “The daisies are worst of all. When one speaks, they all begin together, and it’s enough to make one wither to hear the way they go on!”

—Lewis Carroll, Through the Looking-Glass

To write a program of any complexity using threads, you’ll need to share data between threads, or cause various actions to be performed in some coherent order across multiple threads. To do this, you need to *synchronize* the activity of your threads.

Section 3.1 describes a few of the basic terms we’ll be using to talk about thread synchronization: *critical section* and *invariant*.

Section 3.2 describes the basic Pthreads synchronization mechanism, the mutex.

Section 3.3 describes the *condition variable*, a mechanism that your code can use to communicate changes to the state of *invariants* protected by a mutex.

Section 3.4 completes this chapter on synchronization with some important information about threads and how they view the computer’s memory.

3.1 Invariants, critical sections, and predicates

***“I know what you’re thinking about,”
said Tweedledum; “but it isn’t so, nohow.”
“Contrariwise,” continued Tweedledee,
“if it was so, it might be; and if it were so, it would be;
but as it isn’t, it ain’t. That’s logic.”***

—Lewis Carroll, Through the Looking-Glass

Invariants are assumptions made by a program, especially assumptions about the relationships between sets of variables. When you build a queue package, for example, you need certain data. Each queue has a queue header, which is a pointer to the first queued data element. Each data element includes a pointer to the next data element. But the data isn’t all that’s important—your queue package relies on relationships between that data. The queue header, for example,

must either be `NULL` or contain a pointer to the first queued data element. Each data element must contain a pointer to the next data element, or `NULL` if it is the last. Those relationships are the *invariants* of your queue package.

It is hard to write a program that doesn't have invariants, though many of them are subtle. When a program encounters a broken invariant, for example, if it dereferences a queue header containing a pointer to something that is not a valid data element, the program will probably produce incorrect results or fail immediately.

Critical sections (also sometimes called “serial regions”) are areas of code that affect a shared state. Since most programmers are trained to think about program *functions* instead of program *data*, you may well find it easier to recognize critical sections than data invariants. However, a critical section can almost always be translated into a data invariant, and vice versa. When you remove an element from a queue, for example, you can see the code performing the removal as a critical section, or you can see the state of the queue as an invariant. Which you see first may depend on how you're thinking about that aspect of your design.

Most invariants can be “broken,” and are routinely broken, during isolated areas of code. The trick is to be sure that broken invariants are always repaired before “unsuspecting” code can encounter them. That is a large part of what “synchronization” is all about in an asynchronous program. Synchronization protects your program from broken invariants. If your code locks a mutex whenever it must (temporarily) break an invariant, then other threads that rely on the invariant, and which also lock the mutex, will be delayed until the mutex is unlocked—when the invariant has been restored.

Synchronization is voluntary, and the participants must cooperate for the system to work. The programmers must agree not to fight for (or against) possession of the bailing bucket. The bucket itself does not somehow magically ensure that one and only one programmer bails at any time. Rather, the bucket is a reliable shared token that, if used properly, can allow the programmers to manage their resources effectively.

“Predicates” are logical expressions that describe the state of invariants needed by your code. In English, predicates can be expressed as statements like “the queue is empty” or “the resource is available.” A predicate may be a boolean variable with a `TRUE` or `FALSE` value, or it may be the result of testing whether a pointer is `NULL`. A predicate may also be a more complicated expression, such as determining whether a counter is greater than some threshold. A predicate may even be a value returned from some function. For example, you might call `select` or `poll` to determine whether a file is ready for input.

3.2 Mutexes

*“How are you getting on?” said the Cat,
as soon as there was mouth enough for it to speak with.
Alice waited till the eyes appeared, and then nodded.
“It’s no use speaking to it,” she thought,
“till its ears have come, or at least one of them.”*
—Lewis Carroll, *Alice’s Adventures in Wonderland*

Most threaded programs need to share some data between threads. There may be trouble if two threads try to access shared data at the same time, because one thread may be in the midst of modifying some data invariant while another acts on the data as if it were consistent. This section is all about protecting the program from that sort of trouble.

The most common and general way to synchronize between threads is to ensure that all memory accesses to the same (or related) data are “mutually exclusive.” That means that only one thread is allowed to write at a time—others must wait for their turn. Pthreads provides mutual exclusion using a special form of Edsger Dijkstra’s semaphore [Dijkstra, 1968a], called a *mutex*. The word *mutex* is a clever combination of “mut” from the word “mutual” and “ex” from the word “exclusion.”

Experience has shown that it is easier to use mutexes correctly than it is to use other synchronization models such as a more general semaphore. It is also easy to build any synchronization models using mutexes in combination with condition variables (we’ll meet them at the next corner, in Section 3.3). Mutexes are simple, flexible, and can be implemented efficiently.

The programmers’ bailing bucket is something like a mutex (Figure 3.1). Both are “tokens” that can be handed around, and used to preserve the integrity of the concurrent system. The bucket can be thought of as protecting the bailing critical section—each programmer accepts the responsibility of bailing while holding the bucket, and of avoiding interference with the current bailer while not holding the bucket. Or, the bucket can be thought of as protecting the invariant that water can be removed by only one programmer at any time.

Synchronization isn’t important just when you modify data. You also need synchronization when a thread needs to read data that was written by another thread, if the order in which the data was written matters. As we’ll see a little later, in Section 3.4, many hardware systems don’t guarantee that one processor will see shared memory accesses in the same order as another processor without a “nudge” from software.



FIGURE 3.1 *Mutex analogy*

Consider, for example, a thread that writes new data to an element in an array, and then updates a `max_index` variable to indicate that the array element is valid. Now consider another thread, running simultaneously on another processor, that steps through the array performing some computation on each valid element. If the second thread “sees” the new value of `max_index` before it sees the new value of the array element, the computation would be incorrect. This may seem irrational, but memory systems that work this way can be substantially faster than memory systems that guarantee predictable ordering of memory accesses. A mutex is one general solution to this sort of problem. If each thread locks a mutex around the section of code that’s using shared data, only one thread will be able to enter the section at a time.

Figure 3.2 shows a timing diagram of three threads sharing a mutex. Sections of the lines that are above the rounded box labeled “mutex” show where the associated thread does not own the mutex. Sections of the lines that are below the center line of the box show where the associated thread owns the mutex, and sections of the lines hovering above the center line show where the thread is waiting to own the mutex.

Initially, the mutex is unlocked. Thread 1 locks the mutex and, because there is no contention, it succeeds immediately—thread 1’s line moves below the center

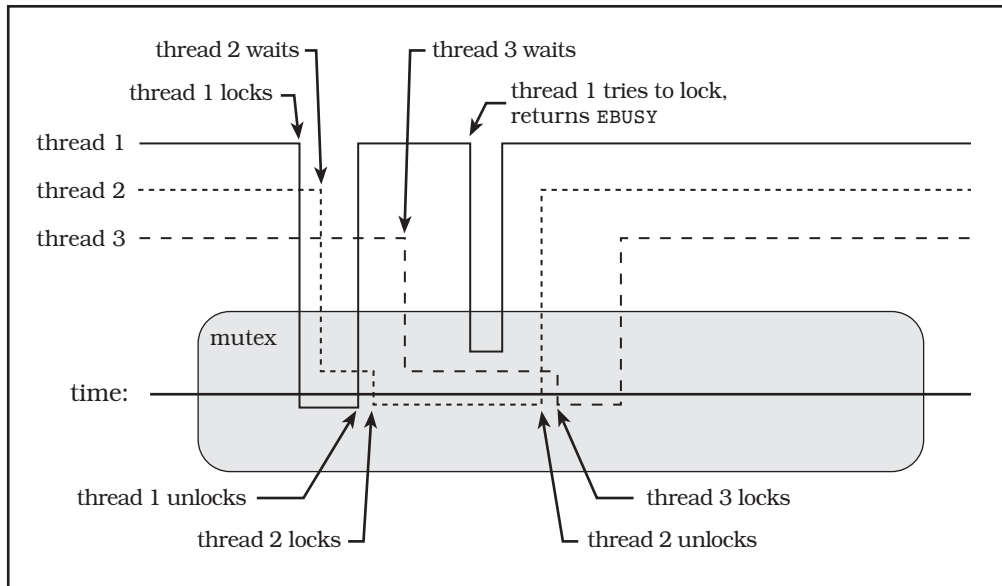


FIGURE 3.2 *Mutex operation*

of the box. Thread 2 then attempts to lock the mutex and, because the mutex is already locked, thread 2 blocks, its line remaining above the center line. Thread 1 unlocks the mutex, unblocking thread 2, which then succeeds in locking the mutex. Slightly later, thread 3 attempts to lock the mutex, and blocks. Thread 1 calls `pthread_mutex_trylock` to try to lock the mutex and, because the mutex is locked, returns immediately with `EBUSY` status. Thread 2 unlocks the mutex, which unblocks thread 3 so that it can lock the mutex. Finally, thread 3 unlocks the mutex to complete our example.

3.2.1 Creating and destroying a mutex

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init (
    pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

A mutex is represented in your program by a variable of type `pthread_mutex_t`. You should never make a copy of a mutex, because the result of using a copied mutex is undefined. You can, however, freely copy a pointer to a mutex so that various functions and threads can use it for synchronization.

Most of the time you'll probably declare mutexes using `extern` or `static` storage class, at "file scope," that is, outside of any function. They should have "normal" (`extern`) storage class if they are used by other files, or `static` storage class if used only within the file that declares the variable. When you declare a `static` mutex that has default attributes, you should use the `PTHREAD_MUTEX_INITIALIZER` macro, as shown in the `mutex_static.c` program shown next. (You can build and run this program, but don't expect anything interesting to happen, since `main` is empty.)

■ mutex_static.c

```
1 #include <pthread.h>
2 #include "errors.h"
3
4 /*
5  * Declare a structure, with a mutex, statically initialized. This
6  * is the same as using pthread_mutex_init, with the default
7  * attributes.
8  */
9 typedef struct my_struct_tag {
10     pthread_mutex_t    mutex; /* Protects access to value */
11     int                value; /* Access protected by mutex */
12 } my_struct_t;
13
14 my_struct_t data = {PTHREAD_MUTEX_INITIALIZER, 0};
15
16 int main (int argc, char *argv[])
17 {
18     return 0;
19 }
```

■ mutex_static.c

Often you cannot initialize a mutex statically, for example, when you use `malloc` to create a structure that contains a mutex. Then you will need to call `pthread_mutex_init` to initialize the mutex dynamically, as shown in `mutex_dynamic.c`, the next program. You can also dynamically initialize a mutex that you declare statically—but you must ensure that each mutex is initialized before it is used, and that each is initialized only once. You may initialize it before creating any threads, for example, or by calling `pthread_once` (Section 5.1). Also, if you need to initialize a mutex with nondefault attributes, you must use dynamic initialization (see Section 5.2.1).

■ mutex_dynamic.c

```
1 #include <pthread.h>
2 #include "errors.h"
3
```

```
4 /*
5  * Define a structure, with a mutex.
6  */
7 typedef struct my_struct_tag {
8     pthread_mutex_t    mutex; /* Protects access to value */
9     int                value; /* Access protected by mutex */
10 } my_struct_t;
11
12 int main (int argc, char *argv[])
13 {
14     my_struct_t *data;
15     int status;
16
17     data = malloc (sizeof (my_struct_t));
18     if (data == NULL)
19         errno_abort ("Allocate structure");
20     status = pthread_mutex_init (&data->mutex, NULL);
21     if (status != 0)
22         err_abort (status, "Init mutex");
23     status = pthread_mutex_destroy (&data->mutex);
24     if (status != 0)
25         err_abort (status, "Destroy mutex");
26     (void)free (data);
27     return status;
28 }
```

■ mutex_dynamic.c

It is a good idea to associate a mutex clearly with the data it protects, if possible, by keeping the definition of the mutex and data together. In `mutex_static.c` and `mutex_dynamic.c`, for example, the mutex and the data it protects are defined in the same structure, and line comments document the association.

When you no longer need a mutex that you dynamically initialized by calling `pthread_mutex_init`, you should destroy the mutex by calling `pthread_mutex_destroy`. You do not need to destroy a mutex that was statically initialized using the `PTHREAD_MUTEX_INITIALIZER` macro.

▮ You can destroy a mutex as soon as you are sure no threads are blocked on the mutex.

It is safe to destroy a mutex when you know that no threads can be blocked on the mutex, and the mutex is unlocked. The best way to know this is usually within a thread that has just unlocked the mutex, when program logic ensures that no threads will try to lock the mutex later. When a thread locks a mutex within some heap data structure to remove the structure from a list and free the storage, for example, it is safe (and a good idea) to unlock and destroy the mutex before freeing the storage that the mutex occupies.

3.2.2 Locking and unlocking a mutex

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

In the simplest case, using a mutex is easy. You lock the mutex by calling either `pthread_mutex_lock` or `pthread_mutex_trylock`, do something with the shared data, and then unlock the mutex by calling `pthread_mutex_unlock`. To make sure that a thread can read consistent values for a series of variables, you need to lock your mutex around any section of code that reads *or* writes those variables.

You cannot lock a mutex when the calling thread already has that mutex locked. The result of attempting to do so may be an error return (`EDEADLK`), or it may be a self-deadlock, where the unfortunate thread waits forever. You cannot unlock a mutex that is unlocked, or that is locked by another thread. Locked mutexes are owned by the thread that locks them. If you need an “unowned” lock, use a semaphore. (Section 6.6.6 discusses semaphores.)

The following program, `alarm_mutex.c`, is an improved version of `alarm_thread.c` (from Chapter 1). It lines up multiple alarm requests in a single “alarm server” thread.

12-17 The `alarm_t` structure now contains an absolute time, as a standard UNIX `time_t`, which is the number of seconds from the UNIX Epoch (Jan 1 1970 00:00) to the expiration time. This is necessary so that `alarm_t` structures can be sorted by “expiration time” instead of merely by the requested number of seconds. In addition, there is a link member to connect the list of alarms.

19-20 The `alarm_mutex` mutex coordinates access to the list head for alarm requests, called `alarm_list`. The mutex is statically initialized using default attributes, with the `PTHREAD_MUTEX_INITIALIZER` macro. The list head is initialized to `NULL`, or empty.

■ `alarm_mutex.c`

part 1 definitions

```
1 #include <pthread.h>
2 #include <time.h>
3 #include "errors.h"
4
5 /*
6  * The "alarm" structure now contains the time_t (time since the
7  * Epoch, in seconds) for each alarm, so that they can be
8  * sorted. Storing the requested number of seconds would not be
9  * enough, since the "alarm thread" cannot tell how long it has
10 * been on the list.
11 */
```

```
12 typedef struct alarm_tag {
13     struct alarm_tag    *link;
14     int                 seconds;
15     time_t              time;    /* seconds from EPOCH */
16     char                message[64];
17 } alarm_t;
18
19 pthread_mutex_t alarm_mutex = PTHREAD_MUTEX_INITIALIZER;
20 alarm_t *alarm_list = NULL;
```

■ alarm_mutex.c

part 1 definitions

The code for the `alarm_thread` function follows. This function is run as a thread, and processes each alarm request in order from the list `alarm_list`. The thread never terminates—when `main` returns, the thread simply “evaporates.” The only consequence of this is that any remaining alarms will not be delivered—the thread maintains no state that can be seen outside the process.

If you would prefer that the program process all outstanding alarm requests before exiting, you can easily modify the program to accomplish this. The main thread must notify `alarm_thread`, by some means, that it should terminate when it finds the `alarm_list` empty. You could, for example, have `main` set a new global variable `alarm_done` and then terminate using `pthread_exit` rather than `exit`. When `alarm_thread` finds `alarm_list` empty and `alarm_done` set, it would immediately call `pthread_exit` rather than waiting for a new entry.

29-30 If there are no alarms on the list, `alarm_thread` needs to block itself, with the mutex unlocked, at least for a short time, so that `main` will be able to add a new alarm. It does this by setting `sleep_time` to one second.

31-42 If an alarm is found, it is removed from the list. The current time is retrieved by calling the `time` function, and it is compared to the requested time for the alarm. If the alarm has already expired, then `alarm_thread` will set `sleep_time` to 0. If the alarm has not expired, `alarm_thread` computes the difference between the current time and the alarm expiration time, and sets `sleep_time` to that number of seconds.

52-58 The mutex is always unlocked before sleeping or yielding. If the mutex remained locked, then `main` would be unable to insert a new alarm on the list. That would make the program behave synchronously—the user would have to wait until the alarm expired before doing anything else. (The user would be able to enter a single command, but would not receive another prompt until the next alarm expired.) Calling `sleep` blocks `alarm_thread` for the required period of time—it cannot run until the timer expires.

Calling `sched_yield` instead is slightly different. We’ll describe `sched_yield` in detail later (in Section 5.5.2)—for now, just remember that calling `sched_yield` will yield the processor to a thread that is ready to run, but will return immediately if there are no *ready* threads. In this case, it means that the main thread will be allowed to process a user command if there’s input waiting—but if the user hasn’t entered a command, `sched_yield` will return immediately.

64-67 If the alarm pointer is not NULL, that is, if an alarm was processed from `alarm_list`, the function prints a message indicating that the alarm has expired. After printing the message, it frees the alarm structure. The thread is now ready to process another alarm.

■ `alarm_mutex.c`

part 2 `alarm_thread`

```

1  /*
2  * The alarm thread's start routine.
3  */
4  void *alarm_thread (void *arg)
5  {
6      alarm_t *alarm;
7      int sleep_time;
8      time_t now;
9      int status;
10
11     /*
12     * Loop forever, processing commands. The alarm thread will
13     * be disintegrated when the process exits.
14     */
15     while (1) {
16         status = pthread_mutex_lock (&alarm_mutex);
17         if (status != 0)
18             err_abort (status, "Lock mutex");
19         alarm = alarm_list;
20
21         /*
22         * If the alarm list is empty, wait for one second. This
23         * allows the main thread to run, and read another
24         * command. If the list is not empty, remove the first
25         * item. Compute the number of seconds to wait -- if the
26         * result is less than 0 (the time has passed), then set
27         * the sleep_time to 0.
28         */
29         if (alarm == NULL)
30             sleep_time = 1;
31         else {
32             alarm_list = alarm->link;
33             now = time (NULL);
34             if (alarm->time <= now)
35                 sleep_time = 0;
36             else
37                 sleep_time = alarm->time - now;
38 #ifdef DEBUG
39             printf ("[waiting: %d(%d)\">%s\"]\n", alarm->time,
40                 sleep_time, alarm->message);
41 #endif
42         }
43     }

```

```

44     /*
45     * Unlock the mutex before waiting, so that the main
46     * thread can lock it to insert a new alarm request. If
47     * the sleep_time is 0, then call sched_yield, giving
48     * the main thread a chance to run if it has been
49     * readied by user input, without delaying the message
50     * if there's no input.
51     */
52     status = pthread_mutex_unlock (&alarm_mutex);
53     if (status != 0)
54         err_abort (status, "Unlock mutex");
55     if (sleep_time > 0)
56         sleep (sleep_time);
57     else
58         sched_yield ();
59
60     /*
61     * If a timer expired, print the message and free the
62     * structure.
63     */
64     if (alarm != NULL) {
65         printf ("%d %s\n", alarm->seconds, alarm->message);
66         free (alarm);
67     }
68 }
69 }

```

■ alarm_mutex.c

part 2 alarm_thread

And finally, the code for the main program for `alarm_mutex.c`. The basic structure is the same as all of the other versions of the alarm program that we've developed—a loop, reading simple commands from `stdin` and processing each in turn. This time, instead of waiting synchronously as in `alarm.c`, or creating a new asynchronous entity to process each alarm command as in `alarm_fork.c` and `alarm_thread.c`, each request is queued to a server thread, `alarm_thread`. As soon as `main` has queued the request, it is free to read the next command.

8-11 Create the server thread that will process all alarm requests. Although we don't use it, the thread's ID is returned in local variable `thread`.

13-28 Read and process a command, much as in any of the other versions of our alarm program. As in `alarm_thread.c`, the data is stored in a heap structure allocated by `malloc`.

30-32 The program needs to add the alarm request to `alarm_list`, which is shared by both `alarm_thread` and `main`. So we start by locking the mutex that synchronizes access to the shared data, `alarm_mutex`.

33 Because `alarm_thread` processes queued requests, serially, it has no way of knowing how much time has elapsed between reading the command and processing it. Therefore, the alarm structure includes the absolute time of the alarm expiration, which we calculate by adding the alarm interval, in seconds, to the

current number of seconds since the UNIX Epoch, as returned by the time function.

39-49 The alarms are sorted in order of expiration time on the `alarm_list` queue. The insertion code searches the queue until it finds the first entry with a time greater than or equal to the new alarm's time. The new entry is inserted preceding the located entry. Because `alarm_list` is a simple linked list, the traversal maintains a current entry pointer (`next`) and a pointer to the previous entry's link member, or to the `alarm_list` head pointer (`last`).

56-59 If no alarm with a time greater than or equal to the new alarm's time is found, then the new alarm is inserted at the end of the list. That is, if the alarm pointer is `NULL` on exit from the search loop (the last entry on the list always has a link pointer of `NULL`), the previous entry (or queue head) is made to point to the new entry.

■ `alarm_mutex.c`

part 3 main

```

1 int main (int argc, char *argv[])
2 {
3     int status;
4     char line[128];
5     alarm_t *alarm, **last, *next;
6     pthread_t thread;
7
8     status = pthread_create (
9         &thread, NULL, alarm_thread, NULL);
10    if (status != 0)
11        err_abort (status, "Create alarm thread");
12    while (1) {
13        printf ("alarm> ");
14        if (fgets (line, sizeof (line), stdin) == NULL) exit (0);
15        if (strlen (line) <= 1) continue;
16        alarm = (alarm_t*)malloc (sizeof (alarm_t));
17        if (alarm == NULL)
18            errno_abort ("Allocate alarm");
19
20        /*
21         * Parse input line into seconds (%d) and a message
22         * (%64[^\n]), consisting of up to 64 characters
23         * separated from the seconds by whitespace.
24         */
25        if (sscanf (line, "%d %64[^\n]",
26            &alarm->seconds, alarm->message) < 2) {
27            fprintf (stderr, "Bad command\n");
28            free (alarm);
29        } else {
30            status = pthread_mutex_lock (&alarm_mutex);

```

```

31         if (status != 0)
32             err_abort (status, "Lock mutex");
33         alarm->time = time (NULL) + alarm->seconds;
34
35         /*
36          * Insert the new alarm into the list of alarms,
37          * sorted by expiration time.
38          */
39         last = &alarm_list;
40         next = *last;
41         while (next != NULL) {
42             if (next->time >= alarm->time) {
43                 alarm->link = next;
44                 *last = alarm;
45                 break;
46             }
47             last = &next->link;
48             next = next->link;
49         }
50         /*
51          * If we reached the end of the list, insert the new
52          * alarm there. ("next" is NULL, and "last" points
53          * to the link field of the last item, or to the
54          * list header).
55          */
56         if (next == NULL) {
57             *last = alarm;
58             alarm->link = NULL;
59         }
60 #ifdef DEBUG
61         printf ("[list: ");
62         for (next = alarm_list; next != NULL; next = next->link)
63             printf ("%d(%d)[\"%s\"] ", next->time,
64                 next->time - time (NULL), next->message);
65         printf ("]\n");
66 #endif
67         status = pthread_mutex_unlock (&alarm_mutex);
68         if (status != 0)
69             err_abort (status, "Unlock mutex");
70     }
71 }
72 }

```

This simple program has a few severe failings. Although it has the advantage, compared to `alarm_fork.c` or `alarm_thread.c`, of using fewer resources, it is less responsive. Once `alarm_thread` has accepted an alarm request from the queue, it

sleeps until that alarm expires. When it fails to find an alarm request on the list, it sleeps for a second anyway, to allow `main` to accept another alarm command. During all this sleeping, it will fail to notice any alarm requests added to the head of the queue by `main`, until it returns from `sleep`.

This problem could be addressed in various ways. The simplest, of course, would be to go back to `alarm_thread.c`, where a thread was created for each alarm request. That wasn't so bad, since threads are relatively cheap. They're still not as cheap as the `alarm_t` data structure, however, and we'd like to make efficient programs—not just responsive programs. The best solution is to make use of condition variables for signaling changes in the state of shared data, so it shouldn't be a surprise that you'll be seeing one final version of the alarm program, `alarm_cond.c`, in Section 3.3.4.

3.2.2.1 Nonblocking mutex locks

When you lock a mutex by calling `pthread_mutex_lock`, the calling thread will block if the mutex is already locked. Normally, that's what you want. But occasionally you want your code to take some alternate path if the mutex is locked. Your program may be able to do useful work instead of waiting. Pthreads provides the `pthread_mutex_trylock` function, which will return an error status (`EBUSY`) instead of blocking if the mutex is already locked.

When you use a nonblocking mutex lock, be careful to *unlock* the mutex only if `pthread_mutex_trylock` returned with success status. Only the thread that owns a mutex may unlock it. An erroneous call to `pthread_mutex_unlock` may return an error, or it may unlock the mutex while some other thread relies on having it locked—and that will probably cause your program to break in ways that may be very difficult to debug.

The following program, `trylock.c`, uses `pthread_mutex_trylock` to occasionally report the value of a counter—but only when its access does not conflict with the counting thread.

- 4 This definition controls how long `counter_thread` holds the mutex while updating the counter. Making this number larger increases the chance that the `pthread_mutex_trylock` in `monitor_thread` will occasionally return `EBUSY`.
- 14-39 The `counter_thread` wakes up approximately each second, locks the mutex, and spins for a while, incrementing counter. The counter is therefore increased by `SPIN` each second.
- 46-72 The `monitor_thread` wakes up every three seconds, and tries to lock the mutex. If the attempt fails with `EBUSY`, `monitor_thread` counts the failure and waits another three seconds. If the `pthread_mutex_trylock` succeeds, then `monitor_thread` prints the current value of counter (scaled by `SPIN`).
- 80-88 On Solaris 2.5, call `thr_setconcurrency` to set the thread concurrency level to 2. This allows the `counter_thread` and `monitor_thread` to run concurrently on a uniprocessor. Otherwise, `monitor_thread` would not run until `counter_thread` terminated.

■ trylock.c

```
1 #include <pthread.h>
2 #include "errors.h"
3
4 #define SPIN 10000000
5
6 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
7 long counter;
8 time_t end_time;
9
10 /*
11  * Thread start routine that repeatedly locks a mutex and
12  * increments a counter.
13  */
14 void *counter_thread (void *arg)
15 {
16     int status;
17     int spin;
18
19     /*
20      * Until end_time, increment the counter each second. Instead of
21      * just incrementing the counter, it sleeps for another second
22      * with the mutex unlocked, to give monitor_thread a reasonable
23      * chance of running.
24      */
25     while (time (NULL) < end_time)
26     {
27         status = pthread_mutex_lock (&mutex);
28         if (status != 0)
29             err_abort (status, "Lock mutex");
30         for (spin = 0; spin < SPIN; spin++)
31             counter++;
32         status = pthread_mutex_unlock (&mutex);
33         if (status != 0)
34             err_abort (status, "Unlock mutex");
35         sleep (1);
36     }
37     printf ("Counter is %#lx\n", counter);
38     return NULL;
39 }
40
41 /*
42  * Thread start routine to "monitor" the counter. Every 3
43  * seconds, try to lock the mutex and read the counter. If the
44  * trylock fails, skip this cycle.
45  */
46 void *monitor_thread (void *arg)
```

```
47 {
48     int status;
49     int misses = 0;
50
51
52     /*
53      * Loop until end_time, checking the counter every 3 seconds.
54      */
55     while (time (NULL) < end_time)
56     {
57         sleep (3);
58         status = pthread_mutex_trylock (&mutex);
59         if (status != EBUSY)
60         {
61             if (status != 0)
62                 err_abort (status, "Trylock mutex");
63             printf ("Counter is %ld\n", counter/SPIN);
64             status = pthread_mutex_unlock (&mutex);
65             if (status != 0)
66                 err_abort (status, "Unlock mutex");
67         } else
68             misses++;          /* Count "misses" on the lock */
69     }
70     printf ("Monitor thread missed update %d times.\n", misses);
71     return NULL;
72 }
73
74 int main (int argc, char *argv[])
75 {
76     int status;
77     pthread_t counter_thread_id;
78     pthread_t monitor_thread_id;
79
80     #ifdef sun
81         /*
82          * On Solaris 2.5, threads are not timesliced. To ensure
83          * that our threads can run concurrently, we need to
84          * increase the concurrency level to 2.
85          */
86         DPRINTF (("Setting concurrency level to 2\n"));
87         thr_setconcurrency (2);
88     #endif
89
90     end_time = time (NULL) + 60;          /* Run for 1 minute */
91     status = pthread_create (
92         &counter_thread_id, NULL, counter_thread, NULL);
93     if (status != 0)
94         err_abort (status, "Create counter thread");
```

```
95     status = pthread_create (
96         &monitor_thread_id, NULL, monitor_thread, NULL);
97     if (status != 0)
98         err_abort (status, "Create monitor thread");
99     status = pthread_join (counter_thread_id, NULL);
100    if (status != 0)
101        err_abort (status, "Join counter thread");
102    status = pthread_join (monitor_thread_id, NULL);
103    if (status != 0)
104        err_abort (status, "Join monitor thread");
105    return 0;
106 }
```

■ trylock.c

3.2.3 Using mutexes for atomicity

Invariants, as we saw in Section 3.1, are statements about your program that must always be true. But we also saw that invariants probably aren't always true, and many can't be. To be always true, data composing an invariant must be modified atomically. Yet it is rarely possible to make multiple changes to a program state atomically. It may not even be possible to guarantee that a single change is made atomically, without substantial knowledge of the hardware and architecture and control over the executed instructions.

┆ "Atomic" means indivisible. But most of the time, we just mean that threads don't see things that would confuse them.

Although some hardware will allow you to set an array element and increment the array index in a single instruction that cannot be interrupted, most won't. Most compilers don't let you control the code to that level of detail even if the hardware can do it, and who wants to write in assembler unless it is *really* important? And, more importantly, most interesting invariants are more complicated than that.

By "atomic," we really mean only that other threads can't accidentally find invariants broken (in intermediate and inconsistent states), even when the threads are running simultaneously on separate processors. There are two basic ways to do that when the hardware doesn't support making the operation indivisible and noninterruptable. One is to detect that you're looking at a broken invariant and try again, or reconstruct the original state. That's hard to do reliably unless you know a lot about the processor architecture and are willing to design nonportable code.

When there is no way to enlist true atomicity in your cause, you need to create your own synchronization. Atomicity is nice, but synchronization will do just as well in most cases. So when you need to update an array element and the index variable atomically, just perform the operation while a mutex is locked.

Whether or not the store and increment operations are performed indivisibly and noninterruptably by the hardware, you know that no cooperating thread can peek until you're done. The transaction is, for all practical purposes, "atomic." The key, of course, is the word "cooperating." Any thread that is sensitive to the invariant must use the same mutex before modifying or examining the state of the invariant.

3.2.4 Sizing a mutex to fit the job

How big is a mutex? No, I don't mean the amount of memory consumed by a `pthread_mutex_t` structure. I'm talking about a colloquial and completely inaccurate meaning that happens to make sense to most people. This colorful usage became common during discussions about modifying existing nonthreaded code to be thread-safe. One relatively simple way to make a library thread-safe is to create a single mutex, lock it on each entry to the library, and unlock it on each exit from the library. The library becomes a single serial region, preventing any conflict between threads. The mutex protecting this big serial region came to be referred to as a "big" mutex, clearly larger in some metaphysical sense than a mutex that protects only a few lines of code.

By irrelevant but inevitable extension, a mutex that protects two variables must be "bigger" than a mutex protecting only a single variable. So we can ask, "How big should a mutex be?" And we can answer only, "As big as necessary, but no bigger."

When you need to protect two shared variables, you have two basic strategies: You can assign a small mutex to each variable, or assign a single larger mutex to both variables. Which is better will depend on a lot of factors. Furthermore, the factors will probably change during development, depending on how many threads need the data and how they use it.

These are the main design factors:

1. Mutexes aren't free. It takes time to lock them, and time to unlock them. Therefore, code that locks fewer mutexes will usually run faster than code that locks more mutexes. So use as few as practical, each protecting as much as makes sense.
2. Mutexes, by their nature, serialize execution. If a lot of threads frequently need to lock a single mutex, the threads will spend most of their time waiting. That's bad for performance. If the pieces of data (or code) protected by the mutex are unrelated, you can often improve performance by splitting the big mutex into several smaller mutexes. Fewer threads will need the smaller mutexes at any time, so they'll spend less time waiting. So use as many as makes sense, each protecting as little as is practical.
3. Items 1 and 2 conflict. But that's nothing new or unique, and you can deal with it once you understand what's going on.

In a complicated program it will usually take some experimentation to get the right balance. Your code will be *simpler* in most cases if you start with large mutexes and then work toward smaller mutexes as experience and performance data show where the heavy contention happens. Simple is good. Don't spend too much time optimizing until you know there's a problem.

On the other hand, in cases where you can tell from the beginning that the algorithms will make heavy contention inevitable, don't oversimplify. Your job will be a lot easier if you start with the necessary mutexes and data structure design rather than adding them later. You will get it wrong sometimes, because, especially when you are working on your first major threaded project, your intuition will not always be correct. Wisdom, as they say, comes from experience, and experience comes from lack of wisdom.

3.2.5 Using more than one mutex

Sometimes one mutex isn't enough. This happens when your code "crosses over" some boundary within the software architecture. For example, when multiple threads will access a queue data structure at the same time, you may need a mutex to protect the queue header and another to protect data within a queue element. When you build a tree structure for threaded programming, you may need a mutex for each node in the tree.

Complications can arise when using more than one mutex at the same time. The worst is deadlock—when each of two threads holds one mutex and needs the other to continue. More subtle problems such as priority inversion can occur when you combine mutexes with priority scheduling. For more information on deadlock, priority inversion, and other synchronization problems, refer to Section 8.1.

3.2.5.1 Lock hierarchy

If you can apply two separate mutexes to completely independent data, do it. You'll almost always win in the end by reducing the time when a thread has to wait for another thread to finish with data that this thread doesn't even need. And if the data is independent you're unlikely to run into many cases where a given function will need to lock both mutexes.

The complications arise when data isn't completely independent. If you have some program invariant—even one that's rarely changed or referenced—that affects data protected by two mutexes, sooner or later you'll need to write code that must lock *both* mutexes at the same time to ensure the integrity of that invariant. If one thread locks `mutex_a` and then locks `mutex_b`, while another thread locks `mutex_b` and then `mutex_a`, you've coded a classic deadlock, as shown in Table 3.1.

First thread	Second thread
<code>pthread_mutex_lock (&mutex_a);</code>	<code>pthread_mutex_lock (&mutex_b);</code>
<code>pthread_mutex_lock (&mutex_b);</code>	<code>pthread_mutex_lock (&mutex_a);</code>

TABLE 3.1 *Mutex deadlock*

Both of the threads shown in Table 3.1 may complete the first step about the same time. Even on a uniprocessor, a thread might complete the first step and then be timesliced (preempted by the system), allowing the second thread to complete its first step. Once this has happened, neither of them can ever complete the second step because each thread needs a mutex that is already locked by the other thread.

Consider these two common solutions to this type of deadlock:

- **Fixed locking hierarchy:** All code that needs both `mutex_a` and `mutex_b` must *always* lock `mutex_a` first and then `mutex_b`.
- **Try and back off:** After locking the first mutex of some set (which can be allowed to block), use `pthread_mutex_trylock` to lock additional mutexes in the set. If an attempt fails, release all mutexes in the set and start again.

There are any number of ways to define a fixed locking hierarchy. Sometimes there's an obvious hierarchical order to the mutexes anyway, for example, if one mutex controls a queue header and one controls an element on the queue, you'll probably have to have the queue header locked by the time you need to lock the queue element anyway.

When there's no obvious logical hierarchy, you can create an arbitrary hierarchy; for example, you could create a generic "lock a set of mutexes" function that sorts a list of mutexes in order of their identifier address and locks them in that order. Or you could assign them names and lock them in alphabetical order, or integer sequence numbers and lock them in numerical order.

To some extent, the order doesn't really matter as long as it is always the same. On the other hand, you will rarely need to lock "a set of mutexes" at one time. Function A will need to lock mutex 1, and then call function B, which needs to also lock mutex 2. If the code was designed with a functional locking hierarchy, you will usually find that mutex 1 and mutex 2 are being locked in the proper order, that is, mutex 1 is locked first and then mutex 2. If the code was designed with an arbitrary locking order, especially an order not directly controlled by the code, such as sorting pointers to mutexes initialized in heap structures, you may find that mutex 2 should have been locked before mutex 1.

If the code invariants permit you to unlock mutex 1 safely at this point, you would do better to avoid owning both mutexes at the same time. That is, unlock mutex 1, and then lock mutex 2. If there is a broken invariant that requires mutex 1 to be owned, then mutex 1 cannot be released until the invariant is restored. If this situation is possible, you should consider using a backoff (or "try and back off") algorithm.

"Backoff" means that you lock the first mutex normally, but any additional mutexes in the set that are required by the thread are locked conditionally by

calling `pthread_mutex_trylock`. If `pthread_mutex_trylock` returns `EBUSY`, indicating that the mutex is already locked, you must unlock *all* of the mutexes in the set and start over.

The backoff solution is less efficient than a fixed hierarchy. You may waste a lot of time trying and backing off. On the other hand, you don't need to define and follow strict locking hierarchy conventions, which makes backoff more flexible. You can use the two techniques in combination to minimize the cost of backing off. Follow some fixed hierarchy for well-defined areas of code, but apply a backoff algorithm where a function needs to be more flexible.

The program below, `backoff.c`, demonstrates how to avoid mutex deadlocks by applying a backoff algorithm. The program creates two threads, one running function `lock_forward` and the other running function `lock_backward`. The two threads loop `ITERATIONS` times, each iteration attempting to lock all of three mutexes in sequence. The `lock_forward` thread locks mutex 0, then mutex 1, then mutex 2, while `lock_backward` locks the three mutexes in the opposite order. Without special precautions, this design will always deadlock quickly (except on a uniprocessor system with a sufficiently long timeslice that either thread can complete before the other has a chance to run).

15 You can see the deadlock by running the program as `backoff 0`. The first argument is used to set the backoff variable. If backoff is 0, the two threads will use `pthread_mutex_lock` to lock each mutex. Because the two threads are starting from opposite ends, they will crash in the middle, and the program will hang. When backoff is nonzero (which it is unless you specify an argument), the threads use `pthread_mutex_trylock`, which enables the backoff algorithm. When the mutex lock fails with `EBUSY`, the thread will release all mutexes it currently owns, and start over.

16 It is possible that, on some systems, you may not see any mutex collisions, because one thread is always able to lock all mutexes before the other thread has a chance to lock any. You can resolve that problem by setting the `yield_flag` variable, which you do by running the program with a second argument, for example, `backoff 1 1`. When `yield_flag` is 0, which it is unless you specify a second argument, each thread's mutex locking loop may run uninterrupted, preventing a deadlock (at least, on a uniprocessor). When `yield_flag` has a value greater than 0, however, the threads will call `sched_yield` after locking each mutex, ensuring that the other thread has a chance to run. And if you set `yield_flag` to a value less than 0, the threads will sleep for one second after locking each mutex, to be *really* sure the other thread has a chance to run.

70-75 After locking all of the three mutexes, each thread reports success, and tells how many times it had to back off before succeeding. On a multiprocessor, or when you've set `yield_flag` to a nonzero value, you'll usually see a lot more nonzero backoff counts. The thread unlocks all three mutexes, in the reverse order of locking, which helps to avoid unnecessary backoffs in other threads. Calling `sched_yield` at the end of each iteration "mixes things up" a little so one thread doesn't always start each iteration first. The `sched_yield` function is described in Section 5.5.2.

■ backoff.c

```
1 #include <pthread.h>
2 #include "errors.h"
3
4 #define ITERATIONS 10
5
6 /*
7  * Initialize a static array of 3 mutexes.
8  */
9 pthread_mutex_t mutex[3] = {
10     PTHREAD_MUTEX_INITIALIZER,
11     PTHREAD_MUTEX_INITIALIZER,
12     PTHREAD_MUTEX_INITIALIZER
13 };
14
15 int backoff = 1;          /* Whether to backoff or deadlock */
16 int yield_flag = 0;      /* 0: no yield, >0: yield, <0: sleep */
17
18 /*
19  * This is a thread start routine that locks all mutexes in
20  * order, to ensure a conflict with lock_reverse, which does the
21  * opposite.
22  */
23 void *lock_forward (void *arg)
24 {
25     int i, iterate, backoffs;
26     int status;
27
28     for (iterate = 0; iterate < ITERATIONS; iterate++) {
29         backoffs = 0;
30         for (i = 0; i < 3; i++) {
31             if (i == 0) {
32                 status = pthread_mutex_lock (&mutex[i]);
33                 if (status != 0)
34                     err_abort (status, "First lock");
35             } else {
36                 if (backoff)
37                     status = pthread_mutex_trylock (&mutex[i]);
38                 else
39                     status = pthread_mutex_lock (&mutex[i]);
40                 if (status == EBUSY) {
41                     backoffs++;
42                     DPRINTF ((
43                         " [forward locker backing off at %d]\n",
44                         i));
45                     for (i--; i >= 0; i--) {
46                         status = pthread_mutex_unlock (&mutex[i]);
47                         if (status != 0)
```

```
48             err_abort (status, "Backoff");
49         }
50     } else {
51         if (status != 0)
52             err_abort (status, "Lock mutex");
53         DPRINTF ((" forward locker got %d\n", i));
54     }
55 }
56 /*
57  * Yield processor, if needed to be sure locks get
58  * interleaved on a uniprocessor.
59  */
60 if (yield_flag) {
61     if (yield_flag > 0)
62         sched_yield ();
63     else
64         sleep (1);
65 }
66 }
67 /*
68  * Report that we got 'em, and unlock to try again.
69  */
70 printf (
71     "lock forward got all locks, %d backoffs\n", backoffs);
72 pthread_mutex_unlock (&mutex[2]);
73 pthread_mutex_unlock (&mutex[1]);
74 pthread_mutex_unlock (&mutex[0]);
75 sched_yield ();
76 }
77 return NULL;
78 }
79
80 /*
81  * This is a thread start routine that locks all mutexes in
82  * reverse order, to ensure a conflict with lock_forward, which
83  * does the opposite.
84  */
85 void *lock_backward (void *arg)
86 {
87     int i, iterate, backoffs;
88     int status;
89
90     for (iterate = 0; iterate < ITERATIONS; iterate++) {
91         backoffs = 0;
92         for (i = 2; i >= 0; i--) {
93             if (i == 2) {
94                 status = pthread_mutex_lock (&mutex[i]);
95                 if (status != 0)
96                     err_abort (status, "First lock");
```

```

97         } else {
98             if (backoff)
99                 status = pthread_mutex_trylock (&mutex[i]);
100            else
101                status = pthread_mutex_lock (&mutex[i]);
102            if (status == EBUSY) {
103                backoffs++;
104                DPRINTF ((
105                    " [backward locker backing off at %d]\n",
106                    i));
107                for (i++; i < 3; i++) {
108                    status = pthread_mutex_unlock (&mutex[i]);
109                    if (status != 0)
110                        err_abort (status, "Backoff");
111                }
112            } else {
113                if (status != 0)
114                    err_abort (status, "Lock mutex");
115                DPRINTF ((" backward locker got %d\n", i));
116            }
117        }
118        /*
119         * Yield processor, if needed to be sure locks get
120         * interleaved on a uniprocessor.
121         */
122        if (yield_flag) {
123            if (yield_flag > 0)
124                sched_yield ();
125            else
126                sleep (1);
127        }
128    }
129    /*
130     * Report that we got 'em, and unlock to try again.
131     */
132    printf (
133        "lock backward got all locks, %d backoffs\n", backoffs);
134    pthread_mutex_unlock (&mutex[0]);
135    pthread_mutex_unlock (&mutex[1]);
136    pthread_mutex_unlock (&mutex[2]);
137    sched_yield ();
138    }
139    return NULL;
140 }
141
142 int main (int argc, char *argv[])
143 {
144     pthread_t forward, backward;

```

```
145     int status;
146
147 #ifdef sun
148     /*
149     * On Solaris 2.5, threads are not timesliced. To ensure
150     * that our threads can run concurrently, we need to
151     * increase the concurrency level.
152     */
153     DPRINTF (("Setting concurrency level to 2\n"));
154     thr_setconcurrency (2);
155 #endif
156
157     /*
158     * If the first argument is absent, or nonzero, a backoff
159     * algorithm will be used to avoid deadlock. If the first
160     * argument is zero, the program will deadlock on a lock
161     * "collision."
162     */
163     if (argc > 1)
164         backoff = atoi (argv[1]);
165
166     /*
167     * If the second argument is absent, or zero, the two threads
168     * run "at speed." On some systems, especially uniprocessors,
169     * one thread may complete before the other has a chance to run,
170     * and you won't see a deadlock or backoffs. In that case, try
171     * running with the argument set to a positive number to cause
172     * the threads to call sched_yield() at each lock; or, to make
173     * it even more obvious, set to a negative number to cause the
174     * threads to call sleep(1) instead.
175     */
176     if (argc > 2)
177         yield_flag = atoi (argv[2]);
178     status = pthread_create (
179         &forward, NULL, lock_forward, NULL);
180     if (status != 0)
181         err_abort (status, "Create forward");
182     status = pthread_create (
183         &backward, NULL, lock_backward, NULL);
184     if (status != 0)
185         err_abort (status, "Create backward");
186     pthread_exit (NULL);
187 }
```

■ backoff.c

Whatever type of hierarchy you choose, *document* it, carefully, completely, and often. Document it in each function that uses any of the mutexes. Document it where the mutexes are defined. Document it where they are declared in a project

header file. Document it in the project design notes. Write it on your whiteboard. And then tie a string around your finger to be sure that you do not forget.

You are free to unlock the mutexes in whatever order makes the most sense. Unlocking mutexes cannot result in deadlock. In the next section, I will talk about a sort of “overlapping hierarchy” of mutexes, called a “lock chain,” where the normal mode of operation is to lock one mutex, lock the next, unlock the first, and so on. If you use a “try and back off” algorithm, however, you should always try to release the mutexes in reverse order. That is, if you lock mutex 1, mutex 2, and then mutex 3, you should unlock mutex 3, then mutex 2, and finally mutex 1. If you unlock mutex 1 and mutex 2 while mutex 3 is still locked, another thread may have to lock both mutex 1 and mutex 2 before finding it cannot lock the entire hierarchy, at which point it will have to unlock mutex 2 and mutex 1, and then retry. Unlocking in reverse order reduces the chance that another thread will need to back off.

3.2.5.2 Lock chaining

“Chaining” is a special case of locking hierarchy, where the scope of two locks overlap. With one mutex locked, the code enters a region where another mutex is required. After successfully locking that second mutex, the first is no longer needed, and can be released. This technique can be very valuable in traversing data structures such as trees or linked lists. Instead of locking the entire data structure with a single mutex, and thereby preventing any parallel access, each node or link has a unique mutex. The traversal code would first lock the queue head, or tree root, find the desired node, lock it, and then release the root or queue head mutex.

Because chaining is a special form of hierarchy, the two techniques are compatible, if you apply them carefully. You might use hierarchical locking when balancing or pruning a tree, for example, and chaining when searching for a specific node.

Apply lock chaining with caution, however. It is exceptionally easy to write code that spends most of its time locking and unlocking mutexes that never exhibit any contention, and that is wasted processor time. Use lock chaining only when multiple threads will almost always be active within different parts of the hierarchy.

3.3 Condition variables

“There’s no sort of use in knocking,” said the Footman, “and that for two reasons. First, because I’m on the same side of the door as you are: secondly, because they’re making such a noise inside, no one could possibly hear you.”

—Lewis Carroll, *Alice’s Adventures in Wonderland*



FIGURE 3.3 *Condition variable analogy*

A condition variable is used for communicating information about the state of shared data. You would use a condition variable to signal that a queue was no longer empty, or that it had become empty, or that anything else needs to be done or can be done within the shared data manipulated by threads in your program.

Our seafaring programmers use a mechanism much like condition variables to communicate (Figure 3.3). When the rower nudges a sleeping programmer to signal that the sleeping programmer should wake up and start rowing, the original rower “signals a condition.” When the exhausted ex-rower sinks into a deep slumber, secure that another programmer will wake him at the appropriate time, he is “waiting on a condition.” When the horrified bailer discovers that water is seeping into the boat faster than he can remove it, and he yells for help, he is “broadcasting a condition.”

When a thread has mutually exclusive access to some shared state, it may find that there is no more it can do until some other thread changes the state. The state may be correct, and consistent—that is, no invariants are broken—but the current state just doesn’t happen to be of interest to the thread. If a thread servicing a queue finds the queue empty, for example, the thread must wait until an entry is added to the queue.

The shared data, for example, the queue, is protected by a mutex. A thread must lock the mutex to determine the current state of the queue, for example, to determine that it is empty. The thread must unlock the mutex before waiting (or

no other thread would be able to insert an entry onto the queue), and then it must wait for the state to change. The thread might, for example, by some means block itself so that a thread inserting a new queue entry can find its identifier and awaken it. There is a problem here, though—the thread is running between unlocking and blocking.

If the thread is still running while another thread locks the mutex and inserts an entry onto the queue, that other thread cannot determine that a thread is waiting for the new entry. The waiting thread has already looked at the queue and found it empty, and has unlocked the mutex, so it will now block itself without knowing that the queue is no longer empty. Worse, it may not yet have recorded the fact that it intends to wait, so it may wait forever because the other thread cannot find its identifier. The unlock and wait operations must be atomic, so that no other thread can lock the mutex before the waiter has become blocked, and is in a state where another thread can awaken it.

- A condition variable wait always returns with the mutex locked.

That's why *condition variables* exist. A condition variable is a “signaling mechanism” associated with a mutex and by extension is also associated with the shared data protected by the mutex. *Waiting* on a condition variable atomically releases the associated mutex and waits until another thread *signals* (to wake one waiter) or *broadcasts* (to wake all waiters) the condition variable. The mutex must always be locked when you wait on a condition variable and, when a thread wakes up from a condition variable wait, it always resumes with the mutex locked.

The shared data associated with a condition variable, for example, the queue “full” and “empty” conditions, are the *predicates* we talked about in Section 3.1. A condition variable is the mechanism your program uses to wait for a predicate to become true, and to communicate to other threads that it might be true. In other words, a condition variable allows threads using the queue to exchange information about the changes to the queue state.

- Condition variables are for *signaling*, not for mutual exclusion.

Condition variables do not provide mutual exclusion. You need a mutex to synchronize access to the shared data, including the predicate for which you wait. That is why you must specify a mutex when you wait on a condition variable. By making the unlock atomic with the wait, the Pthreads system ensures that no thread can change the predicate after you have unlocked the mutex but before your thread is waiting on the condition variable.

Why isn't the mutex created as part of the condition variable? First, mutexes are used separately from any condition variable as often as they're used with condition variables. Second, it is common for one mutex to have more than one associated condition variable. For example, a queue may be “full” or “empty.” Although you may have two condition variables to allow threads to wait for either

condition, you must have one and only one mutex to synchronize *all* access to the queue header.

A condition variable should be associated with a single predicate. If you try to share one condition variable between several predicates, or use several condition variables for a single predicate, you're risking deadlock or race problems. There's nothing wrong with doing either, as long as you're careful—but it is easy to confuse your program (computers aren't very smart) and it is usually not worth the risk. I will expound on the details later, but the rules are as follows: First, when you share a condition variable between multiple predicates, you must always *broadcast*, never *signal*; and second, *signal* is more efficient than *broadcast*.

Both the condition variable and the predicate are shared data in your program; they are used by multiple threads, possibly at the same time. Because you're thinking of the condition variable and predicate as being locked together, it is easy to remember that they're always controlled using the same mutex. It is possible (and legal, and often even reasonable) to *signal* or *broadcast* a condition variable without having the mutex locked, but it is safer to have it locked.

Figure 3.4 is a timing diagram showing how three threads, thread 1, thread 2, and thread 3, interact with a condition variable. The rounded box represents the condition variable, and the three lines represent the actions of the three threads.

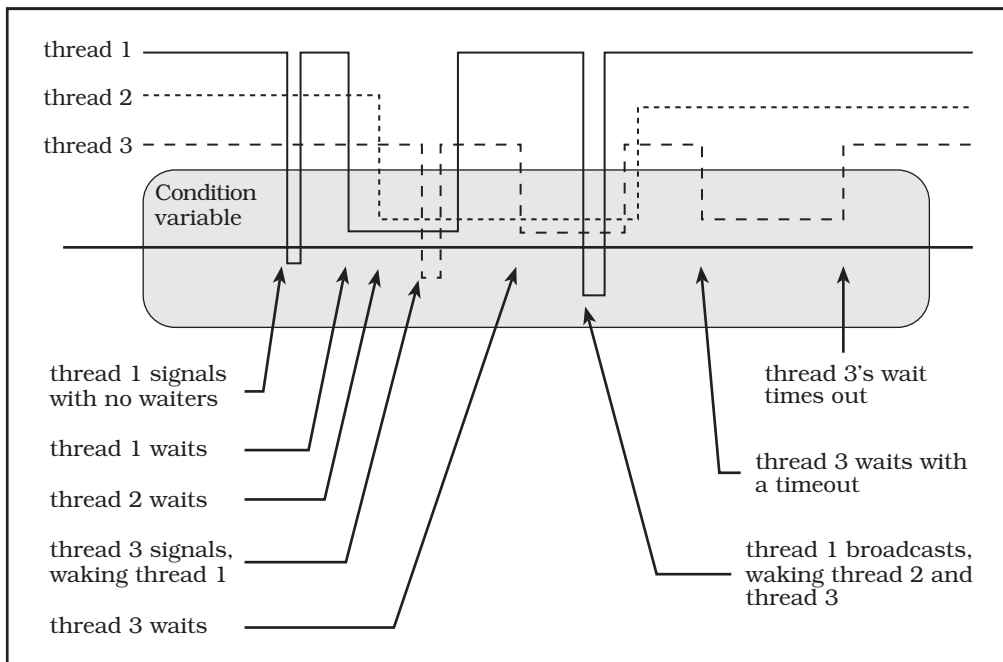


FIGURE 3.4 Condition variable operation

When a line goes within the box, it is “doing something” with the condition variable. When a thread’s line stops before reaching below the middle line through the box, it is waiting on the condition variable; and when a thread’s line reaches below the middle line, it is signaling or broadcasting to awaken waiters.

Thread 1 signals the condition variable, which has no effect since there are no waiters. Thread 1 then waits on the condition variable. Thread 2 also blocks on the condition variable and, shortly thereafter, thread 3 signals the condition variable. Thread 3’s signal unblocks thread 1. Thread 3 then waits on the condition variable. Thread 1 broadcasts the condition variable, unblocking both thread 2 and thread 3. Thread 3 waits on the condition variable shortly thereafter, with a timed wait. Some time later, thread 3’s wait times out, and the thread awakens.

3.3.1 Creating and destroying a condition variable

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init (pthread_cond_t *cond,
    pthread_condattr_t *condattr);
int pthread_cond_destroy (pthread_cond_t *cond);
```

A condition variable is represented in your program by a variable of type `pthread_cond_t`. You should never make a copy of a condition variable, because the result of using a copied condition variable is undefined. It would be like telephoning a disconnected number and expecting an answer. One thread could, for example, wait on one copy of the condition variable, while another thread signaled or broadcast the other copy of the condition variable—the waiting thread would not be awakened. You can, however, freely pass pointers to a condition variable so that various functions and threads can use it for synchronization.

Most of the time you’ll probably declare condition variables using the `extern` or `static` storage class at file scope, that is, outside of any function. They should have normal (`extern`) storage class if they are used by other files, or `static` storage class if used only within the file that declares the variable. When you declare a static condition variable that has default attributes, you should use the `PTHREAD_COND_INITIALIZER` initialization macro, as shown in the following example, `cond_static.c`.

■ `cond_static.c`

```
1 #include <pthread.h>
2 #include "errors.h"
3
4 /*
5  * Declare a structure, with a mutex and condition variable,
6  * statically initialized. This is the same as using
```

```

7  * pthread_mutex_init and pthread_cond_init, with the default
8  * attributes.
9  */
10 typedef struct my_struct_tag {
11     pthread_mutex_t    mutex; /* Protects access to value */
12     pthread_cond_t     cond;  /* Signals change to value */
13     int                value; /* Access protected by mutex */
14 } my_struct_t;
15
16 my_struct_t data = {
17     PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, 0};
18
19 int main (int argc, char *argv[])
20 {
21     return 0;
22 }

```

■ cond_static.c

Condition variables and their predicates are “linked”—for best results, treat them that way!

When you declare a condition variable, remember that a condition variable and the associated predicate are “locked together.” You may save yourself (or your successor) some confusion by always declaring the condition variable and predicate together, if possible. I recommend that you try to encapsulate a set of invariants and predicates with its mutex and one or more condition variables as members in a structure, and carefully document the association.

Sometimes you cannot initialize a condition variable statically; for example, when you use `malloc` to create a structure that contains a condition variable. Then you will need to call `pthread_cond_init` to initialize the condition variable dynamically, as shown in the following example, `cond_dynamic.c`. You can also dynamically initialize condition variables that you declare statically—but you must ensure that each condition variable is initialized before it is used, and that each is initialized only once. You may initialize it before creating any threads, for example, or by using `pthread_once` (Section 5.1). If you need to initialize a condition variable with nondefault attributes, you must use dynamic initialization (see Section 5.2.2).

■ cond_dynamic.c

```

1 #include <pthread.h>
2 #include "errors.h"
3
4 /*
5  * Define a structure, with a mutex and condition variable.
6  */
7 typedef struct my_struct_tag {

```

```
8     pthread_mutex_t    mutex; /* Protects access to value */
9     pthread_cond_t     cond;  /* Signals change to value */
10    int                 value; /* Access protected by mutex */
11 } my_struct_t;
12
13 int main (int argc, char *argv[])
14 {
15     my_struct_t *data;
16     int status;
17
18     data = malloc (sizeof (my_struct_t));
19     if (data == NULL)
20         errno_abort ("Allocate structure");
21     status = pthread_mutex_init (&data->mutex, NULL);
22     if (status != 0)
23         err_abort (status, "Init mutex");
24     status = pthread_cond_init (&data->cond, NULL);
25     if (status != 0)
26         err_abort (status, "Init condition");
27     status = pthread_cond_destroy (&data->cond);
28     if (status != 0)
29         err_abort (status, "Destroy condition");
30     status = pthread_mutex_destroy (&data->mutex);
31     if (status != 0)
32         err_abort (status, "Destroy mutex");
33     (void)free (data);
34     return status;
35 }
```

■ cond_dynamic.c

When you dynamically initialize a condition variable, you should destroy the condition variable when you no longer need it, by calling `pthread_cond_destroy`. You do not need to destroy a condition variable that was statically initialized using the `PTHREAD_COND_INITIALIZER` macro.

It is safe to destroy a condition variable when you know that no threads can be blocked on the condition variable, and no additional threads will try to wait on, signal, or broadcast the condition variable. The best way to determine this is usually within a thread that has just successfully broadcast to unblock all waiters, when program logic ensures that no threads will try to use the condition variable later.

When a thread removes a structure containing a condition variable from a list, for example, and then broadcasts to awaken any waiters, it is safe (and also a very good idea) to destroy the condition variable before freeing the storage that the condition variable occupies. The awakened threads should check their wait predicate when they resume, so you must make sure that you don't free resources required for the predicate before they've done so—this may require additional synchronization.

3.3.2 Waiting on a condition variable

```
int pthread_cond_wait (pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_timedwait (pthread_cond_t *cond,
                            pthread_mutex_t *mutex,
                            struct timespec *expiration);
```

Each condition variable must be associated with a specific mutex, and with a predicate condition. When a thread waits on a condition variable it must always have the associated mutex locked. Remember that the condition variable wait operation will *unlock* the mutex for you before blocking the thread, and it will *relock* the mutex before returning to your code.

All threads that wait on any one condition variable concurrently (at the same time) must specify the *same* associated mutex. Pthreads does not allow thread 1, for example, to wait on condition variable A specifying mutex A while thread 2 waits on condition variable A specifying mutex B. It is, however, perfectly reasonable for thread 1 to wait on condition variable A specifying mutex A while thread 2 waits on condition variable B specifying mutex A. That is, each condition variable must be associated, at any given time, with only one mutex—but a mutex may have any number of condition variables associated with it.

It is important that you test the predicate after locking the appropriate mutex and before waiting on the condition variable. If a thread signals or broadcasts a condition variable while no threads are waiting, nothing happens. If some other thread calls `pthread_cond_wait` right after that, it will keep waiting regardless of the fact that the condition variable was just signaled, which means that if a thread waits when it doesn't have to, it may never wake up. Because the mutex remains locked until the thread is blocked on the condition variable, the predicate cannot become set between the predicate test and the wait—the mutex is locked and no other thread can change the shared data, including the predicate.

■ Always test your predicate; and then test it again!

It is equally important that you test the predicate again when the thread wakes up. You should always wait for a condition variable in a loop, to protect against program errors, multiprocessor races, and spurious wakeups. The following short program, `cond.c`, shows how to wait on a condition variable. Proper predicate loops are also shown in all of the examples in this book that use condition variables, for example, `alarm_cond.c` in Section 3.3.4.

20-37 The `wait_thread` sleeps for a short time to allow the main thread to reach its condition wait before waking it, sets the shared predicate (`data.value`), and then signals the condition variable. The amount of time for which `wait_thread` will sleep is controlled by the `hibernation` variable, which defaults to one second.

- 51-52 If the program was run with an argument, interpret the argument as an integer value, which is stored in `hibernation`. This controls the amount of time for which `wait_thread` will sleep before signaling the condition variable.
- 68-83 The main thread calls `pthread_cond_timedwait` to wait for up to two seconds (from the current time). If `hibernation` has been set to a value of greater than two seconds, the condition wait will time out, returning `ETIMEDOUT`. If `hibernation` has been set to two, the main thread and `wait_thread` race, and, in principle, the result could differ each time you run the program. If `hibernation` is set to a value less than two, the condition wait should not time out.

■ `cond.c`

```

1 #include <pthread.h>
2 #include <time.h>
3 #include "errors.h"
4
5 typedef struct my_struct_tag {
6     pthread_mutex_t    mutex;    /* Protects access to value */
7     pthread_cond_t     cond;    /* Signals change to value */
8     int                value;   /* Access protected by mutex */
9 } my_struct_t;
10
11 my_struct_t data = {
12     PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, 0};
13
14 int hibernation = 1;           /* Default to 1 second */
15
16 /*
17  * Thread start routine. It will set the main thread's predicate
18  * and signal the condition variable.
19  */
20 void *
21 wait_thread (void *arg)
22 {
23     int status;
24
25     sleep (hibernation);
26     status = pthread_mutex_lock (&data.mutex);
27     if (status != 0)
28         err_abort (status, "Lock mutex");
29     data.value = 1;           /* Set predicate */
30     status = pthread_cond_signal (&data.cond);
31     if (status != 0)
32         err_abort (status, "Signal condition");
33     status = pthread_mutex_unlock (&data.mutex);
34     if (status != 0)
35         err_abort (status, "Unlock mutex");
36     return NULL;
37 }

```

```
38
39 int main (int argc, char *argv[])
40 {
41     int status;
42     pthread_t wait_thread_id;
43     struct timespec timeout;
44
45     /*
46      * If an argument is specified, interpret it as the number
47      * of seconds for wait_thread to sleep before signaling the
48      * condition variable. You can play with this to see the
49      * condition wait below time out or wake normally.
50      */
51     if (argc > 1)
52         hibernation = atoi (argv[1]);
53
54     /*
55      * Create wait_thread.
56      */
57     status = pthread_create (
58         &wait_thread_id, NULL, wait_thread, NULL);
59     if (status != 0)
60         err_abort (status, "Create wait thread");
61
62     /*
63      * Wait on the condition variable for 2 seconds, or until
64      * signaled by the wait_thread. Normally, wait_thread
65      * should signal. If you raise "hibernation" above 2
66      * seconds, it will time out.
67      */
68     timeout.tv_sec = time (NULL) + 2;
69     timeout.tv_nsec = 0;
70     status = pthread_mutex_lock (&data.mutex);
71     if (status != 0)
72         err_abort (status, "Lock mutex");
73
74     while (data.value == 0) {
75         status = pthread_cond_timedwait (
76             &data.cond, &data.mutex, &timeout);
77         if (status == ETIMEDOUT) {
78             printf ("Condition wait timed out.\n");
79             break;
80         }
81         else if (status != 0)
82             err_abort (status, "Wait on condition");
83     }
84
85     if (data.value != 0)
86         printf ("Condition was signaled.\n");
```

```
87     status = pthread_mutex_unlock (&data.mutex);
88     if (status != 0)
89         err_abort (status, "Unlock mutex");
90     return 0;
91 }
```

■ cond.c

There are a lot of reasons why it is a good idea to write code that does not assume the predicate is always true on wakeup, but here are a few of the main reasons:

Intercepted wakeups: Remember that threads are asynchronous. Waking up from a condition variable wait involves locking the associated mutex. But what if some other thread acquires the mutex first? It may, for example, be checking the predicate before waiting itself. It doesn't have to wait, since the predicate is now true. If the predicate is "work available," it will accept the work. When it unlocks the mutex there may be no more work. It would be expensive, and usually counterproductive, to ensure that the latest awakened thread got the work.

Loose predicates: For a lot of reasons it is often easy and convenient to use approximations of actual state. For example, "there may be work" instead of "there is work." It is often much easier to signal or broadcast based on "loose predicates" than on the real "tight predicates." If you always test the tight predicates before and after waiting on a condition variable, you're free to signal based on the loose approximations when that makes sense. And your code will be much more robust when a condition variable is signaled or broadcast accidentally. Use of loose predicates or accidental wakeups may turn out to be a performance issue; but in many cases it won't make a difference.

Spurious wakeups: This means that when you wait on a condition variable, the wait may (occasionally) return when no thread specifically broadcast or signaled that condition variable. Spurious wakeups may sound strange, but on some multiprocessor systems, making condition wakeup completely predictable might substantially slow all condition variable operations. The race conditions that cause spurious wakeups should be considered rare.

It usually takes only a few instructions to retest your predicate, and it is a good programming discipline. Continuing without retesting the predicate could lead to serious application errors that might be difficult to track down later. So don't make assumptions: Always wait for a condition variable in a while loop testing the predicate.

You can also use the `pthread_cond_timedwait` function, which causes the wait to end with an `ETIMEDOUT` status after a certain time is reached. The time is an absolute clock time, using the POSIX.1b `struct timespec` format. The timeout is absolute rather than an interval (or "delta time") so that once you've computed the timeout it remains valid regardless of spurious or intercepted

wakeups. Although it might seem easier to use an interval time, you'd have to recompute it every time the thread wakes up, before waiting again—which would require determining how long it had already waited.

When a timed condition wait returns with the `ETIMEDOUT` error, you should test your predicate before treating the return as an error. If the condition for which you were waiting is true, the fact that it may have taken too long usually isn't important. Remember that a thread always relocks the mutex before returning from a condition wait, even when the wait times out. Waiting for a locked mutex after timeout can cause the timed wait to appear to have taken a lot longer than the time you requested.

3.3.3 Waking condition variable waiters

```
int pthread_cond_signal (pthread_cond_t *cond);
int pthread_cond_broadcast (pthread_cond_t *cond);
```

Once you've got a thread waiting on a condition variable for some predicate, you'll probably want to wake it up. Pthreads provides two ways to wake a condition variable waiter. One is called "signal" and the other is called "broadcast." A signal operation wakes up a single thread waiting on the condition variable, while broadcast wakes up all threads waiting on the condition variable.

The term "signal" is easily confused with the "POSIX signal" mechanisms that allow you to define "signal actions," manipulate "signal masks," and so forth. However, the term "signal," as we use it here, had independently become well established in threading literature, and even in commercial implementations, and the Pthreads working group decided not to change the term. Luckily, there are few situations where we might be tempted to use both terms together—it is a very good idea to avoid using signals in threaded programs when at all possible. If we are careful to say "signal a condition variable" or "POSIX signal" (or "UNIX signal") where there is any ambiguity, we are unlikely to cause anyone severe discomfort.

It is easy to think of "broadcast" as a generalization of "signal," but it is more accurate to think of signal as an optimization of broadcast. Remember that it is never wrong to use broadcast instead of signal since waiters have to account for intercepted and spurious wakes. The only difference, in fact, is efficiency: A broadcast will wake additional threads that will have to test their predicate and resume waiting. But, in general, you can't replace a broadcast with a signal. "When in doubt, broadcast."

Use signal when only one thread needs to wake up to process the changed state, and when *any* waiting thread can do so. If you use one condition variable for several program predicate conditions, you can't use the signal operation; you couldn't tell whether it would awaken a thread waiting for that predicate, or for

another predicate. Don't try to get around that by resignaling the condition variable when you find the predicate isn't true. That might not pass on the signal as you expect; a spurious or intercepted wakeup could result in a series of pointless resignals.

If you add a single item to a queue, and only threads waiting for an item to appear are blocked on the condition variable, then you should probably use a signal. That'll wake up a single thread to check the queue and let the others sleep undisturbed, avoiding unnecessary context switches. On the other hand, if you add more than one item to the queue, you will probably need to broadcast. For examples of both broadcast and signal operations on condition variables, check out the "read/write lock" package in Section 7.1.2.

Although you must have the associated mutex locked to wait on a condition variable, you can signal (or broadcast) a condition variable with the associated mutex unlocked if that is more convenient. The advantage of doing so is that, on many systems, this may be more efficient. When a waiting thread awakens, it must first lock the mutex. If the thread awakens while the signaling thread holds the mutex, then the awakened thread must immediately block on the mutex—you've gone through two context switches to get back where you started.

Weighing on the other side is the fact that, if the mutex is not locked, any thread (not only the one being awakened) can lock the mutex prior to the thread being awakened. This race is one source of intercepted wakeups. A lower-priority thread, for example, might lock the mutex while another thread was about to awaken a very high-priority thread, delaying scheduling of the high-priority thread. If the mutex remains locked while signaling, this cannot happen—the high-priority waiter will be placed before the lower-priority waiter on the mutex, and will be scheduled first.

3.3.4 One final alarm program

It is time for one final version of our simple alarm program. In `alarm_mutex.c`, we reduced resource utilization by eliminating the use of a separate execution context (thread or process) for each alarm. Instead of separate execution contexts, we used a single thread that processed a list of alarms. There was one problem, however, with that approach—it was not responsive to new alarm commands. It had to finish waiting for one alarm before it could detect that another had been entered onto the list with an earlier expiration time, for example, if one entered the commands "10 message 1" followed by "5 message 2."

*There is an optimization, which I've called "wait morphing," that moves a thread directly from the condition variable wait queue to the mutex wait queue in this case, without a context switch, when the mutex is locked. This optimization can produce a substantial performance benefit for many applications.

Now that we have added condition variables to our arsenal of threaded programming tools, we will solve that problem. The new version, creatively named `alarm_cond.c`, uses a timed condition wait rather than `sleep` to wait for an alarm expiration time. When `main` inserts a new entry at the head of the list, it signals the condition variable to awaken `alarm_thread` immediately. The `alarm_thread` then requeues the alarm on which it was waiting, to sort it properly with respect to the new entry, and tries again.

20,22 Part 1 shows the declarations for `alarm_cond.c`. There are two additions to this section, compared to `alarm_mutex.c`: a condition variable called `alarm_cond` and the `current_alarm` variable, which allows `main` to determine the expiration time of the alarm on which `alarm_thread` is currently waiting. The `current_alarm` variable is an optimization—`main` does not need to awaken `alarm_thread` unless it is either idle, or waiting for an alarm later than the one `main` has just inserted.

```

1  #include <pthread.h>
2  #include <time.h>
3  #include "errors.h"
4
5  /*
6   * The "alarm" structure now contains the time_t (time since the
7   * Epoch, in seconds) for each alarm, so that they can be
8   * sorted. Storing the requested number of seconds would not be
9   * enough, since the "alarm thread" cannot tell how long it has
10  * been on the list.
11  */
12  typedef struct alarm_tag {
13      struct alarm_tag  *link;
14      int                seconds;
15      time_t            time;  /* seconds from EPOCH */
16      char              message[64];
17  } alarm_t;
18
19  pthread_mutex_t alarm_mutex = PTHREAD_MUTEX_INITIALIZER;
20  pthread_cond_t alarm_cond = PTHREAD_COND_INITIALIZER;
21  alarm_t *alarm_list = NULL;
22  time_t current_alarm = 0;

```

```

■ alarm_cond.c                                     part 1  declarations

```

Part 2 shows the new function `alarm_insert`. This function is nearly the same as the list insertion code from `alarm_mutex.c`, except that it signals the condition variable `alarm_cond` when necessary. I made `alarm_insert` a separate function because now it needs to be called from two places—once by `main` to insert a new alarm, and now also by `alarm_thread` to reinsert an alarm that has been “pre-empted” by a new earlier alarm.

9-14 I have recommended that mutex locking protocols be documented, and here is an example: The `alarm_insert` function points out explicitly that it must be called with the `alarm_mutex` locked.

48-53 If `current_alarm` (the time of the next alarm expiration) is 0, then the `alarm_thread` is not aware of any outstanding alarm requests, and is waiting for new work. If `current_alarm` has a time greater than the expiration time of the new alarm, then `alarm_thread` is not planning to look for new work soon enough to handle the new alarm. In either case, signal the `alarm_cond` condition variable so that `alarm_thread` will wake up and process the new alarm.

■ `alarm_cond.c`

part 2 `alarm_insert`

```

1 /*
2  * Insert alarm entry on list, in order.
3  */
4 void alarm_insert (alarm_t *alarm)
5 {
6     int status;
7     alarm_t **last, *next;
8
9     /*
10    * LOCKING PROTOCOL:
11    *
12    * This routine requires that the caller have locked the
13    * alarm_mutex!
14    */
15    last = &alarm_list;
16    next = *last;
17    while (next != NULL) {
18        if (next->time >= alarm->time) {
19            alarm->link = next;
20            *last = alarm;
21            break;
22        }
23        last = &next->link;
24        next = next->link;
25    }
26    /*
27    * If we reached the end of the list, insert the new alarm
28    * there. ("next" is NULL, and "last" points to the link
29    * field of the last item, or to the list header.)
30    */
31    if (next == NULL) {
32        *last = alarm;
33        alarm->link = NULL;
34    }
35 #ifdef DEBUG
36    printf ("[list: ");

```

```

37     for (next = alarm_list; next != NULL; next = next->link)
38         printf ("%d(%d)[\"%s\"] ", next->time,
39             next->time - time (NULL), next->message);
40     printf ("]\n");
41 #endif
42     /*
43     * Wake the alarm thread if it is not busy (that is, if
44     * current_alarm is 0, signifying that it's waiting for
45     * work), or if the new alarm comes before the one on
46     * which the alarm thread is waiting.
47     */
48     if (current_alarm == 0 || alarm->time < current_alarm) {
49         current_alarm = alarm->time;
50         status = pthread_cond_signal (&alarm_cond);
51         if (status != 0)
52             err_abort (status, "Signal cond");
53     }
54 }

```

■ alarm_cond.c

part 2 alarm_insert

Part 3 shows the `alarm_thread` function, the start function for the “alarm server” thread. The general structure of `alarm_thread` is very much like the `alarm_thread` in `alarm_mutex.c`. The differences are due to the addition of the condition variable.

26-31 If the `alarm_list` is empty, `alarm_mutex.c` could do nothing but sleep anyway, so that `main` would be able to process a new command. The result was that it could not see a new alarm request for at least a full second. Now, `alarm_thread` instead waits on the `alarm_cond` condition variable, with no timeout. It will “sleep” until you enter a new alarm command, and then `main` will be able to awaken it immediately. Setting `current_alarm` to 0 tells `main` that `alarm_thread` is idle. Remember that `pthread_cond_wait` unlocks the mutex before waiting, and relocks the mutex before returning to the caller.

35 The new variable `expired` is initialized to 0; it will be set to 1 later if the timed condition wait expires. This makes it a little easier to decide whether to print the current alarm’s message at the bottom of the loop.

36-42 If the alarm we’ve just removed from the list hasn’t already expired, then we need to wait for it. Because we’re using a timed condition wait, which requires a POSIX.1b `struct timespec`, rather than the simple integer time required by `sleep`, we convert the expiration time. This is easy, because a `struct timespec` has two members—`tv_sec` is the number of seconds since the Epoch, which is exactly what we already have from the `time` function, and `tv_nsec` is an additional count of nanoseconds. We will just set `tv_nsec` to 0, since we have no need of the greater resolution.

43 Record the expiration time in the `current_alarm` variable so that `main` can determine whether to signal `alarm_cond` when a new alarm is added.

- 44-53 Wait until either the current alarm has expired, or main requests that `alarm_thread` look for a new, earlier alarm. Notice that the predicate test is split here, for convenience. The expression in the while statement is only half the predicate, detecting that main has changed `current_alarm` by inserting an earlier timer. When the timed wait returns `ETIMEDOUT`, indicating that the current alarm has expired, we exit the while loop with a `break` statement at line 49.
- 54-55 If the while loop exited when the current alarm had not expired, main must have asked `alarm_thread` to process an earlier alarm. Make sure the current alarm isn't lost by reinserting it onto the list.
- 57 If we remove from `alarm_list` an alarm that has already expired, just set the expired variable to 1 to ensure that the message is printed.

```

■ alarm_cond.c part 3 alarm_routine
1  /*
2  * The alarm thread's start routine.
3  */
4  void *alarm_thread (void *arg)
5  {
6      alarm_t *alarm;
7      struct timespec cond_time;
8      time_t now;
9      int status, expired;
10
11     /*
12     * Loop forever, processing commands. The alarm thread will
13     * be disintegrated when the process exits. Lock the mutex
14     * at the start -- it will be unlocked during condition
15     * waits, so the main thread can insert alarms.
16     */
17     status = pthread_mutex_lock (&alarm_mutex);
18     if (status != 0)
19         err_abort (status, "Lock mutex");
20     while (1) {
21         /*
22         * If the alarm list is empty, wait until an alarm is
23         * added. Setting current_alarm to 0 informs the insert
24         * routine that the thread is not busy.
25         */
26         current_alarm = 0;
27         while (alarm_list == NULL) {
28             status = pthread_cond_wait (&alarm_cond, &alarm_mutex);
29             if (status != 0)
30                 err_abort (status, "Wait on cond");
31         }
32         alarm = alarm_list;
33         alarm_list = alarm->link;
34         now = time (NULL);

```

```

35     expired = 0;
36     if (alarm->time > now) {
37 #ifdef DEBUG
38         printf ("[waiting: %d(%d)\">%s%\"]\n", alarm->time,
39             alarm->time - time (NULL), alarm->message);
40 #endif
41         cond_time.tv_sec = alarm->time;
42         cond_time.tv_nsec = 0;
43         current_alarm = alarm->time;
44         while (current_alarm == alarm->time) {
45             status = pthread_cond_timedwait (
46                 &alarm_cond, &alarm_mutex, &cond_time);
47             if (status == ETIMEDOUT) {
48                 expired = 1;
49                 break;
50             }
51             if (status != 0)
52                 err_abort (status, "Cond timedwait");
53         }
54         if (!expired)
55             alarm_insert (alarm);
56     } else
57         expired = 1;
58     if (expired) {
59         printf ("%d) %s\n", alarm->seconds, alarm->message);
60         free (alarm);
61     }
62 }
63 }

```

■ alarm_cond.c

part 3 alarm_routine

Part 4 shows the final section of `alarm_cond.c`, the main program. It is nearly identical to the main function from `alarm_mutex.c`.

38 Because the condition variable `signal` operation is built into the new `alarm_insert` function, we call `alarm_insert` rather than inserting a new alarm directly.

■ alarm_cond.c

part 4 main

```

1 int main (int argc, char *argv[])
2 {
3     int status;
4     char line[128];
5     alarm_t *alarm;
6     pthread_t thread;
7
8     status = pthread_create (
9         &thread, NULL, alarm_thread, NULL);

```

```

10     if (status != 0)
11         err_abort (status, "Create alarm thread");
12     while (1) {
13         printf ("Alarm> ");
14         if (fgets (line, sizeof (line), stdin) == NULL) exit (0);
15         if (strlen (line) <= 1) continue;
16         alarm = (alarm_t*)malloc (sizeof (alarm_t));
17         if (alarm == NULL)
18             errno_abort ("Allocate alarm");
19
20         /*
21          * Parse input line into seconds (%d) and a message
22          * (%64[^\n]), consisting of up to 64 characters
23          * separated from the seconds by whitespace.
24          */
25         if (sscanf (line, "%d %64[^\n]",
26                 &alarm->seconds, alarm->message) < 2) {
27             fprintf (stderr, "Bad command\n");
28             free (alarm);
29         } else {
30             status = pthread_mutex_lock (&alarm_mutex);
31             if (status != 0)
32                 err_abort (status, "Lock mutex");
33             alarm->time = time (NULL) + alarm->seconds;
34             /*
35              * Insert the new alarm into the list of alarms,
36              * sorted by expiration time.
37              */
38             alarm_insert (alarm);
39             status = pthread_mutex_unlock (&alarm_mutex);
40             if (status != 0)
41                 err_abort (status, "Unlock mutex");
42         }
43     }
44 }

```

■ alarm_cond.c

part 4 main

3.4 Memory visibility between threads

The moment Alice appeared, she was appealed to by all three to settle the question, and they repeated their arguments to her, though, as they all spoke at once, she found it very hard to make out exactly what they said.

—Lewis Carroll, *Alice's Adventures in Wonderland*

In this chapter we have seen how you should use mutexes and condition variables to synchronize (or “coordinate”) thread activities. Now we’ll journey off on a tangent, for just a few pages, and see what is really meant by “synchronization” in the world of threads. It is more than making sure two threads don’t write to the same location at the same time, although that’s part of it. As the title of this section implies, it is about how threads see the computer’s memory.

Pthreads provides a few basic rules about memory visibility. You can count on all implementations of the standard to follow these rules:

1. Whatever memory values a thread can see when it calls `pthread_create` can also be seen by the new thread when it starts. Any data written to memory after the call to `pthread_create` may not necessarily be seen by the new thread, even if the write occurs before the thread starts.
2. Whatever memory values a thread can see when it unlocks a mutex, either directly or by waiting on a condition variable, can also be seen by any thread that later locks the same mutex. Again, data written after the mutex is unlocked may not necessarily be seen by the thread that locks the mutex, even if the write occurs before the lock.
3. Whatever memory values a thread can see when it terminates, either by cancellation, returning from its start function, or by calling `pthread_exit`, can also be seen by the thread that joins with the terminated thread by calling `pthread_join`. And, of course, data written after the thread terminates may not necessarily be seen by the thread that joins, even if the write occurs before the join.
4. Whatever memory values a thread can see when it signals or broadcasts a condition variable can also be seen by any thread that is awakened by that signal or broadcast. And, one more time, data written after the signal or broadcast may not necessarily be seen by the thread that wakes up, even if the write occurs before it awakens.

Figures 3.5 and 3.6 demonstrate some of the consequences. So what should you, as a programmer, do?

First, where possible make sure that only one thread will ever access a piece of data. A thread’s registers can’t be modified by another thread. A thread’s stack and heap memory a thread allocates is private unless the thread communicates pointers to that memory to other threads. Any data you put in `register` or `auto` variables can therefore be read at a later time with no more complication than in a completely synchronous program. Each thread is synchronous with itself. The less data you share between threads, the less work you have to do.

Second, any time two threads need to access the same data, you have to apply one of the Pthreads memory visibility rules, which, in most cases, means using a mutex. This is not only to protect against multiple writes—even when a thread only reads data it must use a mutex to ensure that it sees the most recent value of the data written while the mutex was locked.

This example does everything correctly. The left-hand code (running in thread A) sets the value of several variables while it has a mutex locked. The right-hand code (running in thread B) reads those values, also while holding the mutex.

Thread A	Thread B
<pre>pthread_mutex_lock (&mutex1); variableA = 1; variableB = 2; pthread_mutex_unlock (&mutex1);</pre>	<pre>pthread_mutex_lock (&mutex1); localA = variableA; localB = variableB; pthread_mutex_unlock (&mutex1);</pre>

Rule 2: visibility from `pthread_mutex_unlock` to `pthread_mutex_lock`. When thread B returns from `pthread_mutex_lock`, it will see the same values for `variableA` and `variableB` that thread A had seen at the time it called `pthread_mutex_unlock`. That is, 1 and 2, respectively.

FIGURE 3.5 *Correct memory visibility*

This example shows an error. The left-hand code (running in thread A) sets the value of variables after unlocking the mutex. The right-hand code (running in thread B) reads those values while holding the mutex.

Thread A	Thread B
<pre>pthread_mutex_lock (&mutex1); variableA = 1; pthread_mutex_unlock (&mutex1); variableB = 2;</pre>	<pre>pthread_mutex_lock (&mutex1); localA = variableA; localB = variableB; pthread_mutex_unlock (&mutex1);</pre>

Rule 2: visibility from `pthread_mutex_unlock` to `pthread_mutex_lock`. When thread B returns from `pthread_mutex_lock`, it will see the same values for `variableA` and `variableB` that thread A had seen at the time it called `pthread_mutex_unlock`. That is, it will see the value 1 for `variableA`, but may not see the value 2 for `variableB` since that was written after the mutex was unlocked.

FIGURE 3.6 *Incorrect memory visibility*

As the rules state, there are specific cases where you do not need to use a mutex to ensure visibility. If one thread sets a global variable, and then creates a new thread that reads the same variable, you know that the new thread will not see an old value. But if you create a thread and *then* set some variable that the new thread reads, the thread may not see the new value, even if the creating thread succeeds in writing the new value before the new thread reads it.

Warning! We are now descending below the Pthreads API into details of hardware memory architecture that you may prefer not to know. You may want to skip this explanation for now and come back later.

If you are willing to just trust me on all that (or if you've had enough for now), you may now skip past the end of this section. This book is not about multiprocessor memory architecture, so I will just skim the surface—but even so, the details are a little deep, and if you don't care right now, you do not need to worry about them yet. You will probably want to come back later and read the rest, though, when you have some time.

In a single-threaded, fully synchronous program, it is “safe” to read or write any memory at any time. That is, if the program writes a value to some memory address, and later reads from that memory address, it will always receive the last value that it wrote to that address.

When you add asynchronous behavior (which includes multiprocessors) to the program, the assumptions about memory visibility become more complicated. For example, an asynchronous signal could occur at any point in the program's execution. If the program writes a value to memory, a signal handler runs and writes a different value to the same memory address, when the main program resumes and reads the value, it may not receive the value it wrote.

That's not usually a major problem, because you go to a lot of trouble to declare and use signal handlers. They run “specialized” code in a distinctly different environment from the main program. Experienced programmers know that they should write global data only with extreme care, and it is possible to keep track of what they do. If that becomes awkward, you block the signal around areas of code that use the global data.

When you add multiple threads to the program the asynchronous code is no longer special. Each thread runs normal program code, and all in the same unrestricted environment. You can hardly ever be sure you always know what each thread may be doing. It is likely that they will all read and write some of the same data. Your threads may run at unpredictable times or even simultaneously on different processors. And that's when things get interesting.

By the way, although we are talking about programming with multiple threads, none of the problems outlined in this section is specific to threads. Rather, they are artifacts of memory architecture design, and they apply to any situation where two “things” independently access the same memory. The two things may be threads running on separate processors, but they could instead be processes running on separate processors and using shared memory. Or one “thing” might be code running on a uniprocessor, while an independent I/O controller reads or writes the same memory.

| A memory address can hold only one value at a time; don't let threads “race” to get there first.

When two threads write different values to the same memory address, one after the other, the final state of memory is the same as if a single thread had

written those two values in the same sequence. Either way only one value remains in memory. The problem is that it becomes difficult to know which write occurred last. Measuring some absolute external time base, it may be obvious that “processor B” wrote the value “2” several microseconds after “processor A” wrote the value “1.” That doesn’t mean the final state of memory will have a “2.”

Why? Because we haven’t said anything about how the machine’s cache and memory bus work. The processors probably have cache memory, which is just fast, local memory used to keep quickly accessible copies of data that were recently read from main memory. In a write-back cache system, data is initially written only to cache, and copied (“flushed”) to main memory at some later time. In a machine that doesn’t guarantee read/write ordering, each cache block may be written whenever the processor finds it convenient. If two processors write different values to the same memory address, each processor’s value will go into its own cache. Eventually both values will be written to main memory, but at essentially random times, not directly related to the order in which the values were written to the respective processor caches.

Even two writes from within a single thread (processor) need not appear in memory in the same order. The memory controller may find it faster, or just more convenient, to write the values in “reverse” order, as shown in Figure 3.7. They may have been cached in different cache blocks, for example, or interleaved to different memory banks. In general, there’s no way to make a program aware of these effects. If there was, a program that relied on them might not run correctly on a different model of the same processor family, much less on a different type of computer.

The problems aren’t restricted to two threads *writing* memory. Imagine that one thread writes a value to a memory address on one processor, and then another thread reads from that memory address on another processor. It may seem obvious that the thread will see the last value written to that address, and on some hardware that will be true. This is sometimes called “memory coherence” or “read/write ordering.” But it is complicated to ensure that sort of synchronization between processors. It slows the memory system and the overhead provides no benefit to most code. Many modern computers (usually among the fastest) don’t guarantee any ordering of memory accesses between different processors, unless the program uses special instructions commonly known as *memory barriers*.

Time	Thread 1	Thread 2
t	write “1” to address 1 (cache)	
t+1	write “2” to address 2 (cache)	read “0” from address 1
t+2	cache system flushes address 2	
t+3		read “2” from address 2
t+4	cache system flushes address 1	

FIGURE 3.7 Memory ordering without synchronization

Memory accesses in these computers are, at least in principle, queued to the memory controller, and may be processed in whatever order becomes most efficient. A read from an address that is not in the processor's cache may be held waiting for the cache fill, while later reads complete. A write to a "dirty" cache line, which requires that old data be flushed, may be held while later writes complete. A memory barrier ensures that all memory accesses that were initiated by the processor prior to the memory barrier have completed before any memory accesses initiated after the memory barrier can complete.

■ A "memory barrier" is a moving wall, not a "cache flush" command.

A common misconception about memory barriers is that they "flush" values to main memory, thus ensuring that the values are visible to other processors. That is not the case, however. What memory barriers do is ensure an order between sets of operations. If each memory access is an item in a queue, you can think of a memory barrier as a special queue token. Unlike other memory accesses, however, the memory controller cannot remove the barrier, or look past it, until it has completed all previous accesses.

A mutex lock, for example, begins by locking the mutex, and completes by issuing a memory barrier. The result is that any memory accesses issued while the mutex is locked cannot complete before other threads can see that the mutex was locked. Similarly, a mutex unlock begins by issuing a memory barrier and completes by unlocking the mutex, ensuring that memory accesses issued while the mutex is locked cannot complete after other threads can see that the mutex is unlocked.

This memory barrier model is the logic behind my description of the Pthreads memory rules. For each of the rules, we have a "source" event, such as a thread calling `pthread_mutex_unlock`, and a "destination" event, such as another thread returning from `pthread_mutex_lock`. The passage of "memory view" from the first to the second occurs because of the memory barriers carefully placed in each.

Even without read/write ordering and memory barriers, it may seem that writes to a single memory address must be atomic, meaning that another thread will always see either the intact original value or the intact new value. But that's not always true, either. Most computers have a natural memory granularity, which depends on the organization of memory and the bus architecture. Even if the processor naturally reads and writes 8-bit units, memory transfers may occur in 32- or 64-bit "memory units."

That may mean that 8-bit writes aren't atomic with respect to other memory operations that overlap the same 32- or 64-bit unit. Most computers write the full memory unit (say, 32 bits) that contains the data you're modifying. If two threads write different 8-bit values within the same 32-bit memory unit, the result may be that the last thread to write the memory unit specifies the value of both bytes, overwriting the value supplied by the first writer. Figure 3.8 shows this effect.

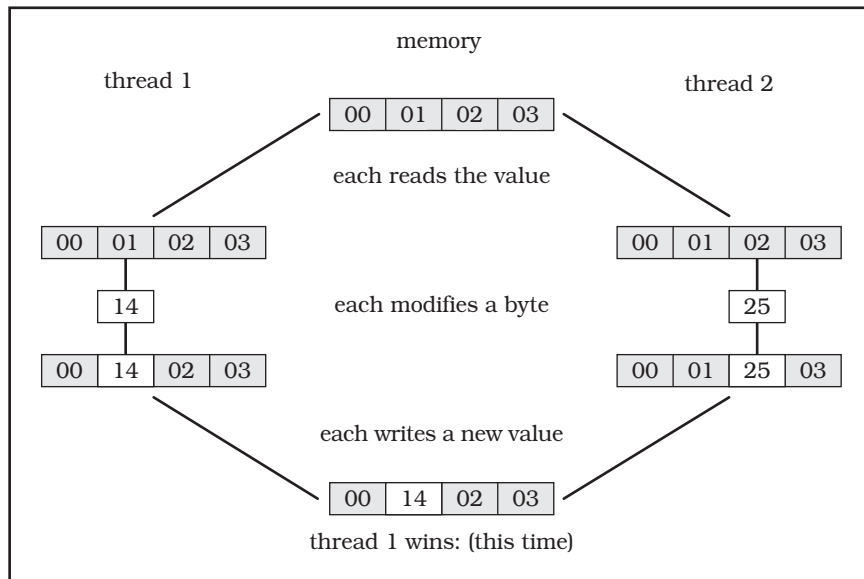


FIGURE 3.8 *Memory conflict*

If a variable crosses the boundary between memory units, which can happen if the machine supports unaligned memory access, the computer may have to send the data in two bus transactions. An unaligned 32-bit value, for example, may be sent by writing the two adjacent 32-bit memory units. If either memory unit involved in the transaction is simultaneously written from another processor, half of the value may be lost. This is called “word tearing,” and is shown in Figure 3.9.

We have finally returned to the advice at the beginning of this section: If you want to write portable Pthreads code, you will always guarantee correct memory visibility by using the Pthreads memory visibility rules instead of relying on any assumptions regarding the hardware or compiler behavior. But now, at the bottom of the section, you have some understanding of why this is true. For a substantially more in-depth treatment of multiprocessor memory architecture, refer to *UNIX Systems for Modern Architectures* [Schimmel, 1994].

Figure 3.10 shows the same sequence as Figure 3.7, but it uses a mutex to ensure the desired read/write ordering. Figure 3.10 does not show the cache flush steps that are shown in Figure 3.7, because those steps are no longer relevant. Memory visibility is guaranteed by passing mutex ownership in steps t+3 and t+4, through the associated memory barriers. That is, when thread 2 has

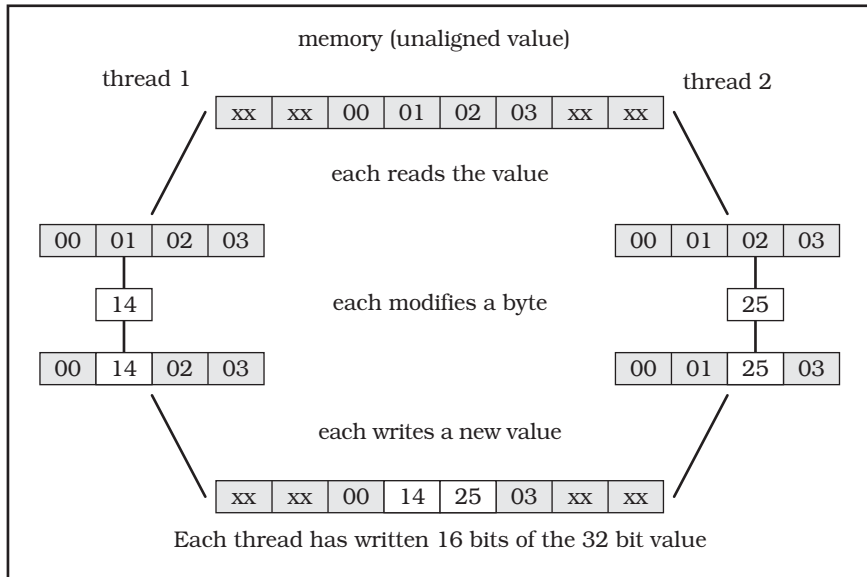


FIGURE 3.9 Word tearing

successfully locked the mutex previously unlocked by thread 1, thread 2 is guaranteed to see memory values “at least as recent” as the values visible to thread 1 at the time it unlocked the mutex.

Time	Thread 1	Thread 2
t	lock mutex (memory barrier)	
t+1	write “1” to address 1 (cache)	
t+2	write “2” to address 2 (cache)	
t+3	(memory barrier) unlock mutex	
t+4		lock mutex (memory barrier)
t+5		read “1” from address 1
t+6		read “2” from address 2
t+7		(memory barrier) unlock mutex

FIGURE 3.10 Memory ordering with synchronization

This page intentionally left blank

Index

A

Abort thread, 361
aio_read, 230
aio_write, 230
Allocation domain, 181–183
Amdahl's Law, 21–22, 297
ANSI C
 and cancellation, 151
 and deadlocks, 298
 and destructor functions, 168
 and fork handlers, 199–200
 and kernel entities, 189–190
 library, runtime, 284
 and priority inversions, 300
 prototype for Pthread interfaces, 310–346
 and sequence races, 296
 and signals, 217
 and threading, 24–25
 void *, use of, 311
 See also Stdio (function)
Apple Macintosh toolbox, 23
asctime_r (function), 213, 339–340
Assembly language and threading, 24–25
Assembly line. *See* Pipeline programming model
Asynchronous
 cancelability, 150–154, 249
 communication between processes, 8
 definition of, 2, 4
 I/O operations, 22–23
 programming, 8–12, 15–21
 sigwait, 227–230
 and UNIX, 9–11
 See also Memory, visibility
Async-signal safe functions, 198, 234
Atomicity, 61, 72–73, 93
Attribute objects
 condition variable attributes, 137–138
 definition of, 134–135
 and detaching, 37, 43, 231

 mutex attributes, 135–136, 185
 thread attributes, 138–141, 174, 181, 231

B

Bailing programmers
 barriers analogy, 242
 cancellation analogy, 143
 condition variable analogy, 71
 mutex analogy, 47–48
 overview of, 3–4
 thread-specific data analogy, 162–163
Barriers
 definition of, 241
 memory, 92–95
 POSIX 1003.1j, 358
 and synchronization, extended, 242–253
 See also Condition variables; Mutexes
Blocking threads, 42
Broadcast
 and condition variables, 72–76, 80–82
 and memory visibility, 89
 and work crews, 109

C

C language. *See* ANSI C
Cache lines, 304–305
Cancellation
 asynchronous, 150–154, 249
 cleanup handlers, 147, 154–161
 deferred, 147–150, 249
 definition of, 142–147
 fail-safe in POSIX 1003.1j, 361
 interfaces, 323–325
 points, list of, 147–148
 points in XSH5, 355–356
 state, 143–144
 type, 143–144
Cleanup handlers, 147, 154–161
Client server programming model, 120–129

- clock_gettime (function), 13
 - CLOCK_MONOTONIC (value), 360
 - Code. *See* Programs, examples of
 - Communication. *See* Condition variables
 - Computing overhead and threads, 26
 - Concurrency
 - benefit of threading, 22–24
 - control functions, definition of, 7–8
 - definition of, 4–5
 - level in XSH5, 351–353
 - and parallelism, 5–6
 - serialization, avoidance of, 302–303
 - and thread implementation, 190
 - See also* Asynchronous; Threads
 - Condition variables
 - attributes, 137–138
 - and blocked threads, 42
 - and client-server programming model, 121
 - broadcasting, 81–82
 - creating and destroying, 74–76
 - definition of, 8
 - interfaces, 319–322
 - and pipeline programming model, 101
 - and semaphores, 237
 - sharing, avoidance of, 300
 - signaling, 81–82
 - as a synchronization mechanism, 4, 8, 30, 70–73
 - and thread-safety, 6–7
 - wait clock (POSIX 1003.1j), 359–361
 - waiting on, 77–81
 - waking waiters, 81–82
 - and work crew programming model, 107
 - See also* Barriers; Mutexes; Predicates; Signals
 - Contention scope, 181–183
 - Context structure, 6–7
 - Critical sections, 46
 - ctime_r (function), 213, 340
- D**
- DCE threads, 154, 290, 353
 - Deadlocks
 - avoidance of, 26–27, 297–299
 - and lock hierarchy, 63–69
 - and signals, 215
 - and thread-safe libraries, 284–285
 - See also* Priority inversions
 - Debugging, threads
 - cache lines, 304–305
 - concurrent serialization, avoidance of, 302–303
 - condition variable sharing, avoidance of, 300
 - and costs of threading, 13, 27–28
 - deadlocks, avoidance of, 297–299
 - introduction to, 13
 - memory corrupters, avoidance of, 301
 - mutexes, right number of, 303–304
 - priority inversions, avoidance of, 299–300
 - thread inertia, avoidance of, 291–293
 - thread race, avoidance of, 293–297
 - tools for, 290, 302
 - See also* Errors
 - Default mutexes, 349–351
 - Deferred cancelability, 147–150, 249
 - Destructor functions, 167–172
 - Detach
 - creating and using threads, 37–39
 - and multiple threads, 17–19
 - termination, 43
 - and thread attributes, 138–141
 - Digital UNIX
 - mutex error detection and reporting, 311
 - thread debugging library, 290
 - programming examples, introduction to, 12–13
 - detecting memory corrupters, 301
 - SIGEV-THREAD implementation, 231
 - Dijkstra, Edsger W., 47
 - Directory searching function, 212
- E**
- EAGAIN (error), 314, 318, 326, 344, 353
 - EBUSY (error), 49, 58, 65, 318–321, 345
 - EDEADLK (error), 32, 318–319, 346
 - EINTR (error), 346
 - EINVAL (error), 312–326, 343–346, 351–355
 - Encapsulation, 24, 75
 - ENOMEM (error), 313, 317–321, 326, 336
 - ENOSPC (error), 345
 - ENOSYS (error), 312–316, 344–346

ENXIO (error), 355
 EOVERFLOW (error), 355
 EPERM (error), 318, 345
 ERANGE (error), 210, 341–342
 err_abort, 33–34
 errno, 31–34, 117, 228
 errno_abort, 33–34
 errno.h (header file), 32–33
 Error detection and reporting, 310–311
 Errorcheck mutexes, 349–351
 Errors
 EAGAIN, 314, 318, 326, 344, 353
 EBUSY, 49, 58, 65, 318–321, 345
 EDEADLK, 32, 318–319, 346
 EINTR, 346
 EINVAL, 312–326, 343–346, 351–355
 ENOMEM, 313, 317–321, 326, 336
 ENOSPC, 345
 ENOSYS, 312–316, 344–346
 ENXIO, 355
 EOVERFLOW, 355
 EPERM, 318, 345
 ERANGE, 210, 341–342
 ESPIPE, 355
 ESRCH, 32, 314, 323, 343
 ETIMEDOUT, 78, 80, 86, 322
 errors.h (header file), 13
 ESPIPE (error), 355
 ESRCH (error), 32, 314, 323, 343
 ETIMEDOUT (error), 78, 80, 86, 322
 Events
 and concurrency, 23–24
 as a synchronization mechanism, 8
 Exec, 204
 Execution context
 architectural overview, 30
 definition of, 7–8
F
 fgets (function), 14
 flockfile (function), 121, 205–207,
 336–337
 Fork
 in asynchronous program, 15, 19
 definition of, 197–198
 handlers, 199–203, 336
 frylockfile (function), 207, 337
 funlockfile (function), 121, 205–207, 337

G

Gallmeister, Bill, 294
 getc (function), 207
 getchar (function), 207
 getchar_unlocked (function), 207–209,
 338
 getc_unlocked (function), 207–208, 337
 getgrgid_r (function), 341
 getgrnam_r (function), 341
 getlogin_r (function), 210, 339
 getpwnam_r (function), 342
 getpwuid_r (function), 342
 gmtime_r (function), 340
 Group and user database function,
 213–214

H

Hard realtime, 172–173

I

inheritsched (attribute), 138–141, 176
 Initial (main) thread
 and creation, 40–42
 initialization, one time, 131–134
 overview of, 36–37
 signal mask, 216
 Initialization, 131–134
 Invariants
 and condition variables, 71, 75
 definition of, 45–46
 and mutexes, 61–64
 See also Predicates
 I/O operations
 and blocked threads, 42
 and cancellation points, 148–149
 candidate for threads, 29
 and concurrency, 22–23
 parallel in XSH5, 354–355

J

Join, 37–39, 139, 145

K

Kernel entities
 and contention scope thread, 182
 implementation of, 189
 many to few (two level), 193–195
 many to one (user level), 190–191

Kernel entities (*continued*)

- one to one (kernel level), 191–193
- setting in XSH5, 351–353

L

Libraries

- for debugging, 290
- implementation, 190
- legacy, 285–287
- thread-safe, 283–285

Light weight processes, 1

limits.h (file header), 308

localtime_r (function), 340

LWP. *See* Kernel entities

M

Main (initial) thread

- and creation, 40–42
- initialization, one time, 131–134
- overview of, 36–37
- signal mask, 216

Many to few (two level) implementation, 193–195

Many to one (user level) implementation, 190–191

Memory

- barriers, 92–95
- coherence, 47, 92
- conflict, 94
- corrupters, avoidance of, 301
- leaks, 198
- ordering, 48, 90, 92–93
- stack guard size in XSH5, 353–354
- visibility, 26, 88–95, 294

Message queues, 30

Microsoft Windows, 23

MIMD (multiple instruction, multiple data), 107

MPP (massively parallel processor), 5

Multiprocessors

- and allocation domain, 182
- candidate for threads, 29
- and concurrency, 1, 4–5
- and deadlock, 65
- definition of, 5
- memory architecture, 91–95
- and parallelism, 5–6, 20–22
- and thread implementation, 191–193

and thread inertia, 291–293

and thread race, 293–297

Multithreaded programming model. *See* Threads

Mutexes

- and atomicity, 61, 72–73
 - attributes, 135–136
 - and blocked threads, 42, 51
 - and client-server programming model, 121
 - creating and destroying, 49–51
 - and deadlocks, 26–27, 63, 66–69, 297–299
 - definition of, 8, 47–49
 - interfaces, 316–319
 - and invariants, 61–64
 - lock chaining, 70
 - lock hierarchy, 63–70
 - locking and unlocking, 52–58
 - multiple, use of, 63–64
 - non-blocking locks, 58–61
 - number of, 303–304
 - and pipeline programming model, 100–101
 - priority ceiling, 186–187, 300
 - priority inheritance, 186–188, 300, 307
 - priority inversions, 26–27, 63, 184–186, 299–300
 - priority-aware, 185–186
 - and semaphores, 236
 - sizing of, 62–63
 - as a synchronization mechanism, 8, 30
 - and thread-safety, 6–7, 62
 - types in XSH5, 349–351
 - and work crew programming model, 108–110
- See also* Barriers; Condition variables; Memory, visibility; Read/write locks

Mutually exclusive. *See* Mutexes

N

Network servers and clients, 22–23

Normal mutexes, 349–351

NULL (value), 167

O

- Object oriented programming and threading, 25
- One to one (kernel level) implementation, 191–193
- Opaque, 31, 36, 163
- Open Software Foundation's Distributed Computing Environment, 154

P

- Paradigms, 25
- Parallel decomposition. *See* Work crews
- Parallelism
 - and asynchronous programming, 11–12
 - benefit of threading, 20–25
 - candidate for threads, 29
 - definition of, 5–6
 - and thread-safety, 6–7
 - See also* Concurrency
- Performance
 - as a benefit of threading, 20–22
 - as a cost of threading, 26
 - and mutexes, 62–63
 - problems, avoidance of, 302–305
- Pipeline programming model, 97–105
- Pipes, 30
- Polymorphism, 24
- POSIX
 - architectural overview, 30
 - conformance document, 307, 308
 - error checking, 31–34
 - realtime scheduling options, 173
 - signal mechanisms, 40–41, 81–82
 - types and interfaces, 30–31
- POSIX 1003.1 (ISO/IEC 9945-1:1996), 29–30
- POSIX 1003.1-1990, 31, 209
- POSIX 1003.1b-1993 (realtime)
 - and condition variables, 3.94, 80
 - and semaphores, 236–237, 345
 - and signals, 230–232
 - thread concepts, 29–30
 - void *, use of, 311
- POSIX 1003.1c-1995 (threads)
 - and cancellation, 154
 - cancellation, interfaces, 323–325
 - condition variables, interfaces, 319–322
 - error detection and reporting, 310–311
 - fork handlers, interfaces, 336
 - interfaces, overview, 309–310
 - limits, 308–309
 - mutexes, interfaces, 316–319
 - options, 307–308
 - and realtime scheduling, 173
 - realtime scheduling, interfaces, 326–335
 - semaphores, interfaces, 345–346
 - signals, interfaces, 342–345
 - stdio, interfaces, 336–338
 - thread concepts, 29–30
 - threads, interfaces, 311–316
 - thread-safe, interfaces, 338–342
 - thread-specific data, interfaces, 325–326
 - void *, use of, 311
- POSIX 1003.1i-1995 (corrections to 1003.1b-1993), 29–30
- POSIX 1003.1j (additional realtime extension)
 - barriers, 249, 356–358
 - read/write locks, 358
 - spinlocks, 359
 - thread abort, 361
 - wait clock, condition variable, 359–361
- POSIX 1003.14 (multiprocessor profile), 30, 182, 361–362
- POSIX_Prio_INHERIT, 186
- POSIX_Prio_NONE (value), 186
- POSIX_Prio_PROTECT, 186
- _POSIX_REALTIME_SIGNALS (option), 228
- _POSIX_SEMAPHORES (option), 235
- _POSIX_THREAD_ATTR_STACKADDR (option), 139, 308, 348
- _POSIX_THREAD_ATTR_STACKSIZE (option), 139, 308, 348
- _POSIX_THREAD_Prio_INHERIT (option), 185–186, 308, 349
- _POSIX_THREAD_Prio_PROTECT (option), 185–186, 308, 349
- _POSIX_THREAD_Prio_PRIORITY_SCHEDULING (option), 173–176, 179, 308, 349
- _POSIX_THREAD_PROCESS_SHARED (option), 136–137, 308, 348

- `_POSIX_THREADS` (option), 308, 348
- `_POSIX_THREAD_SAFE_FUNCTIONS` (option), 308, 349
- `_POSIX_TIMERS` (option), 360
- `pread` (function), 354–355
- Predicates
 - definition of, 46
 - loose, 80
 - wakeup, 80–81
 - waking waiters, 81
 - See also* Condition variables
- `printf` (function), 15
- `prprioceiling` (attribute), 135–138
- Priority ceiling, 186–187, 300
- Priority inheritance
 - definition of, 186
 - mutexes, 188
 - and POSIX 1003.1c options, 307
 - priority inversion, avoidance of, 300
- Priority inversions
 - avoidance of, 299–300
 - as a cost of threading, 26–27
 - mutexes and priority scheduling, 63
 - as a realtime scheduling problem, 184
 - See also* Deadlocks
- Process contention, 181–185
- Process exit, 204
- Processes
 - asynchronous, 8
 - light weight, 1
 - threads, compared to, 10, 20
 - variable weight, 1
- Processors
 - and blocked threads, 42
 - and thread implementation, 190–195
 - See also* Multiprocessors; Uniprocessors
- Programs, examples of
 - barriers, 245–248, 250–253
 - cancellation, 145–161
 - client server, 121–129
 - condition variables, creating and destroying, 74–76
 - condition variables, timed condition wait, 83–88
 - condition variables, waiting on, 78–80
 - creating and using threads, 38–39
 - errors, 32–34
 - flockfile, 205–207
 - fork handlers, 201–203
 - initialization, 133–134
 - multiple processes, 15–16
 - multiple threads, 17–19
 - mutex attributes object, 136
 - mutexes, deadlock avoidance, 66–69
 - mutexes, dynamic, 50–68
 - mutexes, locking and unlocking, 52–57
 - mutexes, non-blocking locks, 58–61
 - mutexes, static mutex, 50
 - pipeline, 99–105
 - `putchar`, 208–209
 - read/write locks, 255–269
 - realtime scheduling, 175–181
 - sample information, 13
 - semaphore, 238–240
 - `SIGEV_THREAD`, 232–234
 - `sigwait`, 228–230
 - suspend and resume, 218–227
 - synchronous programming, 13–15, 27
 - thread attributes, 140–141
 - thread inertia, 292
 - thread-specific, 164–165, 169–172
 - user and terminal identification, 211
 - work crews, 108–120
 - work queue manager, 271–283
- `protocol` (attribute), 135–138, 186
- `pshared` (attribute), 135–138, 204
- `pthread_abort` (function), 361
- `pthread_atfork` (function), 199, 336
- `pthread_attr_destroy` (function), 312
- `pthread_attr_getdetachstate` (function), 312
- `pthread_attr_getguardsize` (function), 353
- `pthread_attr_getinheritsched` (function), 327
- `pthread_attr_getschedparam` (function), 327
- `pthread_attr_getschedpolicy` (function), 327–328
- `pthread_attr_getscope` (function), 328
- `pthread_attr_getstackaddr` (function), 312–313
- `pthread_attr_getstacksize` (function), 135, 139, 313
- `pthread_attr_init` (function), 139, 313

- pthread_attr_setdetachstate (function), 313
- pthread_attr_setguardsize (function), 354
- pthread_attr_setinheritsched (function), 176, 329
- pthread_attr_setschedparam (function), 175, 329
- pthread_attr_setschedpolicy (function), 175, 329–330
- pthread_attr_setscope (function), 182, 330
- pthread_attr_setstackaddr (function), 314
- pthread_attr_setstacksize (function), 135, 314
- pthread_attr_t (datatype), 135, 139, 231
- pthread_cancel (function)
 - asynchronous cancelability, 151
 - deferred cancelability, 148
 - definition of, 323
 - and pthread_kill, 217
 - termination, 43, 143–145
- PTHREAD_CANCEL_ASYNCHRONOUS (value), 152, 324
- PTHREAD_CANCEL_DEFERRED (value), 145, 147, 324
- PTHREAD_CANCEL_DISABLE (value), 145, 149, 324
- PTHREAD_CANCELED (value), 43, 145
- PTHREAD_CANCEL_ENABLE (value), 147, 324
- pthread_cleanup_pop (function), 43, 147, 155, 323
- pthread_cleanup_push (function), 43, 147, 155, 323
- pthread_condattr_destroy (function), 319
- pthread_condattr_getclock (function), 360
- pthread_condattr_getpshared (function), 320
- pthread_condattr_init (function), 137, 320
- pthread_condattr_setclock (function), 360
- pthread_condattr_setpshared (function), 137, 320
- pthread_condattr_t (datatype), 135
- pthread_cond_broadcast (function), 81, 256, 300, 321
- pthread_cond_destroy (function), 76, 321
- pthread_cond_init (function), 75, 137, 321
- pthread_cond_signal (function), 81, 300, 322
- pthread_cond_t (datatype), 74, 137
- pthread_cond_timedwait (function), 78, 80, 322
- pthread_cond_wait (function), 77, 85, 322
- pthread_create (function)
 - and attributes objects, 139
 - creating and using threads, 36–42, 189
 - definition of, 314
 - execution context, 30
 - and memory visibility, 89
 - and multiple threads, 17
 - and thread identifier, 144–145, 266
- PTHREAD_CREATE_DETACHED (value), 44, 125, 139, 231, 312–313
- PTHREAD_CREATE_JOINABLE (value), 139, 231, 312–313
- PTHREAD_DESTRUCTOR_ITERATIONS (limit), 168, 309
- pthread_detach (function)
 - cleaning up, 158
 - creating and using threads, 37
 - definition of, 315
 - and multiple threads, 17
 - termination, 43–44
- pthread_equal (function), 36, 315
- pthread_exit (function)
 - and attributes objects, 140
 - cleaning up, 155
 - creating and using threads, 37–38
 - definition of, 204, 315
 - and fork, 197
 - and memory visibility, 89
 - and multiple threads, 17
 - termination, 30, 40–44, 53
- PTHREAD_EXPLICIT_SCHED (value), 176, 327–329
- pthread_getconcurrency (function), 352

- pthread_getschedparam (function), 331
- pthread_getspecific (function), 34, 164, 166, 325
- pthread.h (header file), 13
- PTHREAD_INHERIT_SCHED (value), 176, 327–329
- pthread_join (function)
 - and attributes objects, 139
 - cleaning up, 158
 - creating and using threads, 37–38
 - definition of, 315
 - and error checking, 32
 - and memory visibility, 89
 - and pthread_kill, 225
 - termination, 43–44, 128, 145
- pthread_key_create (function), 163–166, 325–326
- pthread_key_delete (function), 166, 326
- PTHREAD_KEYS_MAX (limit), 166, 309
- pthread_key_t (datatype), 163–166
- pthread_kill (function), 217–227, 343
- pthread_mutexattr_destroy (function), 316
- pthread_mutexattr_getprioceiling (function), 332
- pthread_mutexattr_getprotocol (function), 332–333
- pthread_mutexattr_getpshared (function), 317
- pthread_mutexattr_gettype (function), 350–351
- pthread_mutexattr_init (function), 135, 317
- pthread_mutexattr_setprioceiling (function), 333
- pthread_mutexattr_setprotocol (function), 186, 333–334
- pthread_mutexattr_setpshared (function), 136, 317
- pthread_mutexattr_settype (function), 351
- pthread_mutexattr_t (datatype), 135
- PTHREAD_MUTEX_DEFAULT (value), 349–351
- pthread_mutex_destroy (function), 51, 318, 350
- PTHREAD_MUTEX_ERRORCHECK (value), 349
- pthread_mutex_getprioceiling (function), 331
- pthread_mutex_init (function)
 - and attributes objects, 135
 - creating and destroying mutexes, 50–51
 - definition of, 318
 - initialization of, 132, 186
 - standardization, future, 350
- PTHREAD_MUTEX_INITIALIZER (macro), 50–52, 74–76
- pthread_mutex_lock (function)
 - asynchronous cancelability, 151
 - definition of, 318
 - lock hierarchy, 64–65
 - locking and unlocking, 52, 58
 - and memory visibility, 90, 93
 - mutexes, number of, 303–304
 - standardization, future, 350
 - XSHS mutex types, 350
- PTHREAD_MUTEX_NORMAL (value), 349
- PTHREAD_MUTEX_RECURSIVE (value), 349
- pthread_mutex_setprioceiling (function), 332
- pthread_mutex_t (datatype), 49, 62, 136
- pthread_mutex_trylock (function)
 - creating and destroying mutexes, 49
 - definition of, 319
 - and flockfile and funlockfile, 207
 - lock hierarchy, 64–65
 - locking and unlocking, 52, 58
 - mutexes, number of, 303–304
 - read/write locks, 257
 - standardization, future, 350
 - XSHS mutex types, 350
- pthread_mutex_unlock (function)
 - creating and destroying mutexes, 49
 - definition of, 319
 - and flockfile and funlockfile, 207
 - lock hierarchy, 64–65
 - locking and unlocking, 52, 58
 - and memory visibility, 90

- mutexes, number of, 303–304
 - standardization, future, 350
 - XSHS mutex types, 350
 - pthread_once (function)
 - definition of, 131
 - initialization, condition variables, 75
 - initialization, mutexes, 50
 - or statically initialized mutex, 132
 - in suspend/resume, 220–221
 - and thread races, 295–296
 - thread-specific data, 163–164
 - PTHREAD_ONCE_INIT (macro), 132
 - pthread_once_t (datatype), 132
 - PTHREAD_PRIO_INHERIT (value), 186, 333–334
 - PTHREAD_PRIO_NONE (value), 333–334
 - PTHREAD_PRIO_PROTECT (value), 186, 333–334
 - PTHREAD_PROCESS_PRIVATE (value), 136–137, 317–320
 - PTHREAD_PROCESS_SHARED (value), 136–138, 204, 317–320
 - PTHREAD_SCOPE_PROCESS (value), 182, 328–330
 - PTHREAD_SCOPE_SYSTEM (value), 182, 328–330
 - pthread_self (function), 17, 36–37, 144–145, 316
 - pthread_setcancelstate (function), 147, 149, 151, 324
 - pthread_setcanceltype (function), 151, 324
 - pthread_setconcurrency (function), 352–353
 - pthread_setschedparam (function), 334
 - pthread_setspecific (function), 166, 326
 - pthread_sigmask (function), 215–216, 343
 - pthread_spin_lock (function), 359
 - PTHREAD_STACK_MIN (limit), 139, 309
 - pthread_t (datatype)
 - creating and using threads, 36–37, 189, 266
 - and pthread_kill, 217
 - termination, 43, 144–145
 - thread-specific data, 161–162
 - pthread_testcancel (function), 144–145, 150, 158, 325
 - PTHREAD_THREAD_MAX (limit), 309
 - putc (function), 207
 - putchar (function), 6, 207
 - putchar_unlocked (function), 207–209, 338
 - putc_unlocked (function), 207–208, 338
 - pwrite (function), 355
- ## R
- ### Races
- avoidance of, 26–27
 - and condition variables, 73
 - and memory visibility, 91
 - overview, 293–295
 - sequence race, 284–285, 295–297
 - synchronization race, 294–296
 - thread inertia, 291–293
- raise (function), 217
- Random number generation function, 213
- rand_r (function), 213, 341
- readdir_r (function)
 - definition of, 339
 - directory searching, 212
- reentrancy, 7, 297
- thread-safe function, 210
- and work crews, 107–109
- Read/write locks, 242, 253–269, 358
- Read/write ordering, 92–95
- Ready threads, 39–42, 53
- Realtime scheduling
 - allocation domain, 181–183
 - architectural overview, 30
 - contention scope, 181–183
 - definition of, 7–8, 172–173
 - hard realtime, 172–173
 - interfaces, 326–335
 - mutexes, priority ceiling, 186–187, 300
 - mutexes, priority inheritance, 186–188, 300, 307
 - mutexes, priority-aware, 185–186
 - policies and priorities, 174–181
 - POSIX options, 173
 - and priority inversion, 299–300, 326
 - problems with, 183–185
 - soft realtime, 172–173
 - and synchronization, 295
- Recursive mutexes, 349–351

- Recycling threads, 43–44
- Reentrant, 6–7, 297
 - See also* Readdir_r (function)
- Resume. *See* Pthread_kill (function)
- Running threads, 42

- S**
- Scaling, 20–22
- SC_GETGR_R_SIZE_MAX (value), 214
- SC_GETPW_R_SIZE_MAX (value), 214
- SCHED_BG_NP (value), 175
- SCHED_FG_NP (value), 175
- SCHED_FIFO (value)
 - problems with, 183–185
 - as a scheduling interface value, 328–331, 334–335
 - scheduling policies and priorities, 174–175
 - and thread race, 295
- sched_get_priority_max (function), 174, 335
- sched_get_priority_min (function), 174, 335
- SCHED_OTHER (value), 175, 328–331, 334–335
- schedparam (attribute), 138–141, 175
- schedpolicy (attribute), 138–141, 175
- SCHED_RR (value), 174, 185, 328–331, 334–335
- Scheduler Activations model, 194
- Scheduling. *See* Realtime scheduling
- sched_yield (function), 53–54, 65, 221, 316
- Schimmel, Curt, 94
- scope (attribute), 138–141, 182
- _SC_THREAD_ATTR_STACKADDR (option), 308
- _SC_THREAD_ATTR_STACKSIZE (option), 308
- _SC_THREAD_DESTRUCTOR_ITERATIONS (limit), 309
- _SC_THREAD_KEYS_MAX (limit), 309
- _SC_THREAD_PRIO_INHERIT (option), 185, 308
- _SC_THREAD_PRIO_PROTECT (option), 185–186, 308
- _SC_THREAD_PRIORITY_SCHEDULING (option), 308
- _SC_THREAD_PROCESS_SHARED (option), 308
- _SC_THREADS (limit), 308
- _SC_THREAD_SAFE_FUNCTIONS (option), 308
- _SC_THREAD_STACK_MIN (limit), 309
- _SC_THREAD_THREADS_MAX (limit), 309
- Semaphores
 - functions, definition of, 345–346
 - as a synchronization mechanism, 8, 30
 - synchronization with signal catching, 234–240
- sem_destroy (function), 237, 345
- sem_getvalue (function), 237
- sem_init (function), 237, 345
- sem_post (function), 235–237, 346
- sem_t (value), 237
- sem_trywait (function), 237, 346
- sem_wait (function), 236–237, 346
- Sequence races, 284–285, 295–297
- Serial programming, 25
- Serial regions. *See* Predicates
- Serialization, 21
- Shared data, 3
- sigaction (function), 215
- SIG_BLOCK (value), 343
- SIGCONT (action), 217
- sigevent, 311
- sigev_notify_attributes, 231
- sigev_notify_function, 231
- SIGEV_THREAD (function), 40, 231–234
- sigev_value (function), 311
- SIGFPE (function), 215–216
- SIGKILL (function), 216–217
- Signals
 - actions, 215–216
 - background, 214–215
 - and condition variables, 72–76, 80–81
 - handlers, 91–92
 - interfaces, 342–345
 - masks, 216
 - and memory visibility, 89
 - pthread_kill, 217–227
 - running and blocking threads, 42
 - semaphores, 234–240

- SIGEV_THREAD, 231–234
 - sigwait, 227–230
 - See also* Condition variables
 - SIGPIPE (action), 215
 - sigprocmask (function), 216
 - sigqueue (function), 230
 - SIGSEGV (action), 216
 - SIG_SETMASK (value), 343
 - SIGSTOP (action), 216–217
 - sigtimedwait (function), 228, 343–344
 - SIGTRAP signal, 216, 290
 - SIG_UNBLOCK (value), 343
 - sigwait (function)
 - definition of, 227–230, 344
 - running and blocking, 42
 - and semaphores, 234
 - sigwaitinfo (function), 228, 234, 344–345
 - SIMD (single instruction, multiple data), 106
 - sleep (function), 15
 - SMP. *See* Multiprocessors
 - Soft realtime, 172–173
 - Solaris 2.5
 - concurrency level, setting of, 58, 119, 128, 145, 152, 266
 - thread debugging library, 290
 - programming examples, introduction to, 12–13
 - realtime scheduling, 176, 179
 - SIGEV_THREAD implementation, 231
 - Spinlocks, 359
 - Spurious wakeups, 80–81
 - stackaddr (attribute), 138–141
 - stacksize (attribute), 138–141
 - Startup threads, 41–42
 - stderr, 33
 - stdin (function)
 - in asynchronous program example, 14, 18
 - and client servers, 121
 - in pipeline program example, 98, 105
 - and stdio, 205–207
 - stdio (function)
 - and concurrency, 23
 - and fork handlers, 199
 - interfaces, 336–338
 - and realtime scheduling, 190
 - thread-safety, 6, 205–207
 - stdout (function)
 - and client servers, 121
 - in pipeline program example, 98
 - and stdio, 205
 - in suspend program example, 224
 - strerror (function), 33
 - String token function, 212
 - strtok_r (function), 212, 339
 - struct aiocb, 230
 - struct dirent, 109, 210
 - struct sigevent, 230
 - Suspend. *See* Pthread_kill (function)
 - Synchronization
 - architectural overview, 30
 - and computing overhead, 26
 - critical sections, 46
 - definition of, 7–8
 - objects, 3
 - and programming model, 24–25
 - protocols, 26
 - races, 284–285
 - and reentrant code, 6–7
 - and scheduling, 295
 - and semaphores, 234–240
 - and sequence race, 294–297
 - and UNIX, 9–11
 - See also* Barriers; Condition variables; Invariants; Memory, visibility; Mutexes; Parallelism; Read/write locks
 - Synchronous
 - I/O operations, 22–23
 - programming, 13–15, 27
 - sysconf (function), 185, 214, 307–308
 - System contention, 181–185
- ## T
- Termination of threads, 43–44
 - thd_continue (interface), 217, 223–224
 - thd_suspend (interface), 217–221, 224
 - Threads
 - abort (POSIX 1003.1J), 361
 - architectural overview, 30
 - and asynchronous programming, 8–12
 - attributes, 138–141
 - benefits of, 10–11, 20–25
 - blocking, 42

- Threads (*continued*)
 - candidates for, 28–29
 - client server programming model, 120–129
 - costs of, 25–28
 - creating and using, 25–41
 - definition of, 8
 - error checking, 31–34
 - identifier, 36
 - implementation of, 189
 - initial (main), 36–42, 119, 131–134
 - interfaces, 311–316
 - introduction to, 1–3
 - many to few (two level), 193–195
 - many to one (user level), 190–191
 - one to one (kernel level), 191–193
 - as part of a process, 10
 - pipeline programming model, 97–105
 - processes, compared to, 10, 20
 - programmers, compared to, 3
 - programming model, 24–25
 - ready, 39–42, 53
 - recycling, 43–44
 - running, 42
 - startup, 41–42
 - states of, 39–41
 - termination, 43–44
 - and traditional programming, 4, 27
 - types and interfaces, 30–31
 - work crew programming model, 105–120
 - See also* Concurrency; Debugging, threads
 - Thread-safe
 - definition of, 6–7
 - interfaces, 286–287, 338–342
 - libraries, 283–285
 - library, 303–304
 - and mutexes, 62
 - and programming discipline, 26–27
 - Thread-safe functions
 - directory searching, 212
 - group and user database, 213–214
 - random number generation, 213
 - string token, 212
 - time representation, 212–213
 - user and terminal identification, 209–211
 - Thread-specific data
 - creating, 163–166
 - destructor functions, 167–172
 - interfaces, 325–326
 - overview, 161–163
 - and termination, 43
 - and thread-safety, 6–7
 - use of, 166–167
 - thr_setconcurrency (function), 13, 58, 119, 128, 145, 152
 - Time representation function, 212–213
 - Timer signals, 23
 - timer_create (function), 230
 - Timeslice, 8, 42, 174
 - Tokens, 3, 47
 - tty_name_r (function), 210
- U**
- Uniprocessors
 - and allocation domain, 182
 - and concurrency, 4–5
 - and deadlock, 65
 - definition of, 5
 - and thread inertia, 291–293
 - and thread race, 293–297
 - unistd.h (header file), 307
 - University of Washington, 194
 - UNIX
 - and asynchronous, 9–11, 22
 - and error checking, 31–34
 - kernel, 154
 - programming examples, introduction to, 12–13
 - UNIX Systems for Modern Architectures*, 94
 - UNIX98. *See* XSH5
 - User and terminal identification function, 210–211
- V**
- Variable weight processes, 1
 - void *
 - creating and using threads, 36
 - definition of, 311
 - thread startup, 42
 - thread termination, 43

W

waitpid (function), 15, 19
WNOHANG (flag), 15
Word tearing, 95
Work crews, 105–120, 270–283
Work queue. *See* Work crews

X

X Windows, 23
X/Open. *See* XSH5
*X/Open CAE Specification, System Inter-
faces and Headers, Issue 5. See*
XSH5

XSH5

cancellation points, 148, 355–356
concurrency level, 351–353
mutex error detection and reporting,
311
mutex types, 349–351
parallel I/O, 354–355
POSIX options for, 348–349
stack guard size, 353–354