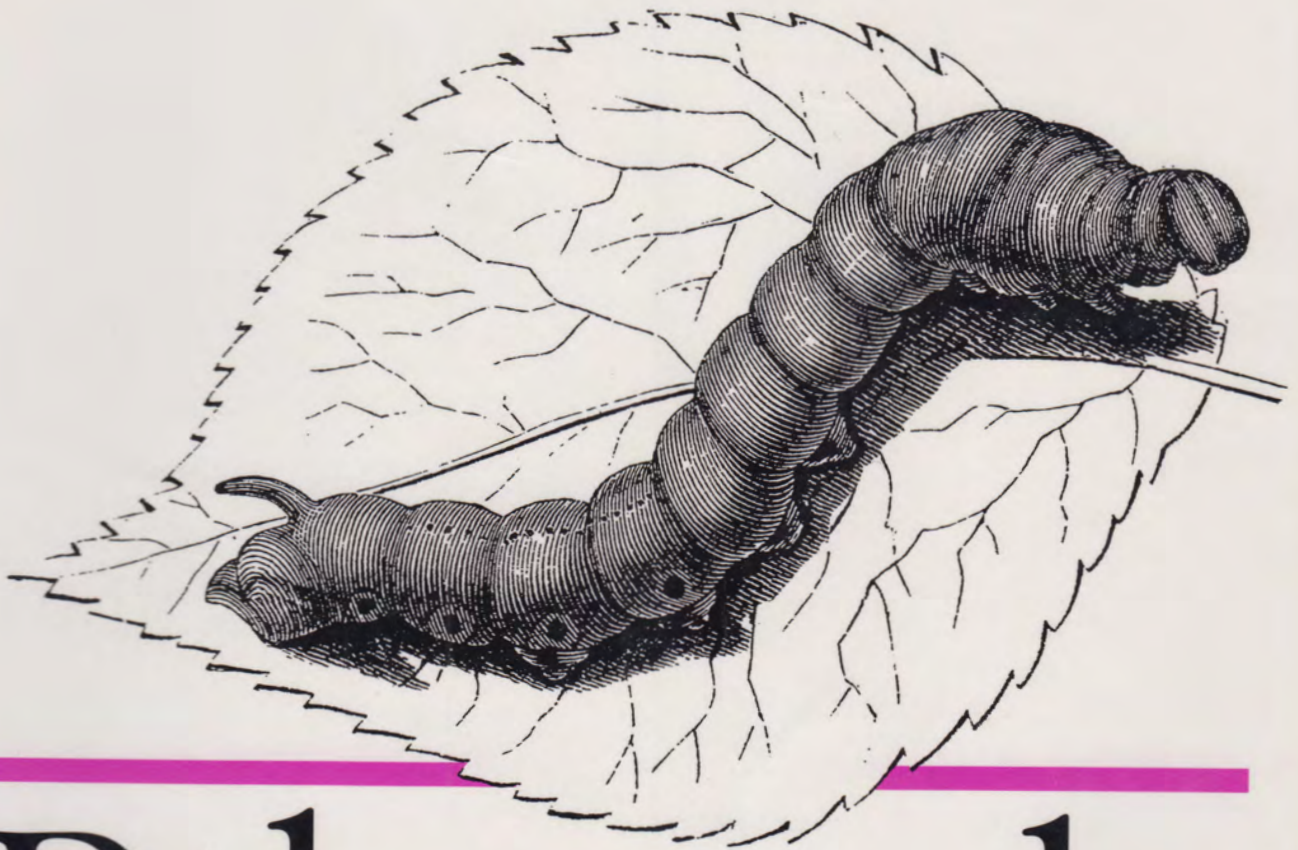


A POSIX Standard for Better Multiprocessing



Pthreads

Programming

A NUTSHELL®



HANDBOOK

Bradford Nichols, Dick Buttlar & Jacqueline Proulx Farrell

O'Reilly & Associates, Inc.

Pthreads Programming

Bradford Nichols, Dick Butlar,
and Jacqueline Proulx Farrell

Pthreads Programming

**Bradford Nichols, Dick Buttlar,
and Jacqueline Proulx Farrell**

O'REILLY™

Bonn • Cambridge • Paris • Sebastopol • Tokyo

Pthreads Programming

by Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell

Copyright © 1996 O'Reilly & Associates, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472.

Editor: Andy Oram

Production Editor: Nancy Crumpton

Printing History:

September 1996: First Edition

Nutshell Handbook and the Nutshell Handbook logo are registered trademarks of O'Reilly & Associates, Inc.

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 O'Reilly & Associates, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.



This book is printed on acid-free paper with 85% recycled content, 15% post-consumer waste. O'Reilly & Associates is committed to using paper with the highest recycled content available consistent with high quality.

ISBN: 1-56592-115-1

Table of Contents

<i>Preface</i>	<i>xiii</i>
<i>1: Why Threads?</i>	<i>1</i>
What Are Pthreads?	6
Potential Parallelism	7
Specifying Potential Parallelism in a Concurrent Programming Environment	9
Parallel vs. Concurrent Programming	16
Synchronization	16
Who Am I? Who Are You?	21
Terminating Thread Execution	22
Why Use Threads Over Processes?	25
A Structured Programming Environment	26
Choosing Which Applications to Thread	26
<i>2: Designing Threaded Programs</i>	<i>29</i>
Suitable Tasks for Threading	29
Models	31
Buffering Data Between Threads	38
Some Common Problems	40
Performance	41
Example: An ATM Server	42
Example: A Matrix Multiplication Program	52

3: Synchronizing Pthreads	61
Selecting the Right Synchronization Tool	63
Mutex Variables	64
Condition Variables	79
Reader/Writer Locks	84
Synchronization in the ATM Server	89
Thread Pools	98
4: Managing Pthreads	109
Setting Thread Attributes	110
The pthread_once Mechanism	115
Keys: Using Thread-Specific Data	120
Cancellation	128
Scheduling Pthreads	143
Mutex Scheduling Attributes	156
5: Pthreads and UNIX	163
Threads and Signals	165
Threadsafe Library Functions and System Calls	176
Cancellation-Safe Library Functions and System Calls	183
Thread-Blocking Library Functions and System Calls	185
Threads and Process Management	186
Multiprocessor Memory Synchronization	191
6: Practical Considerations	193
Understanding Pthreads Implementation	194
Debugging	202
Performance	214
Conclusion	232
A: Pthreads and DCE	235
B: Pthreads Draft 4 vs. the Final Standard	243
C: Pthreads Quick Reference	249
Index	261

We'll spell out the advantages of threads over the multiprocess model of multitasking in our performance measurements in Chapter 6, *Practical Considerations*. In the meantime, we'll show you how to build a multithreaded program.

A Structured Programming Environment

Revisiting the techniques used to obtain concurrency that we discussed earlier—potential parallelism, overlapping I/O, asynchronous events, and real-time scheduling—we find that UNIX offers many disjointed mechanisms to accomplish them between processes. They include the *select* system call, signals, nonblocking I/O, and the *setjmp/longjmp* system call pair, plus many calls for real time (such as *aio_read* and *aio_write*) and parallel processing. Pthreads offers a clean, consistent way to address all of these motivations. If you're a disciplined programmer, designing and coding a multithreaded program should be easier than designing and coding a multiprocess program.

Now, we know that the example program we've been looking at in this chapter is far too simple to convince anyone that a particular programming style is more structured or elegant than another. Subsequent examples will venture into more complex territory and, in doing so, illustrate Pthreads mechanisms for a more practical set of coding problems. We hope that they may make the case for Pthreads.

Choosing Which Applications to Thread

The major benefit of multithreaded programs over nonthreaded ones is in their ability to concurrently execute tasks. However, in providing concurrency, multithreaded programs introduce a certain amount of overhead. If you introduce threads in an application that can't use concurrency, you'll add overhead without any performance benefit.

So what makes concurrency possible? First, of course, your application must consist of some independent tasks—tasks that do not depend on the completion of other tasks to proceed. Secondly, you must be confident that concurrent execution of these tasks would be faster than their serial execution.

On a uniprocessing system, the concurrent execution of independent tasks will be faster than their serial execution if at least one of these tasks issues a lot of I/O requests and must wait for the device to complete each request. On a multiprocessor, even CPU-bound tasks can benefit from concurrency because they can truly proceed in parallel.

If you are writing an application for a uniprocessor, look at overlapping I/O and asynchronous events as the motivation for threading an application. If your program is hung up in doing a lot of disk, file, or network accesses when it could be doing other useful things, threads offer a means of doing them while the thread that handles the I/O is waiting.* If your program must deal with many asynchronous events, such as the receipt of an out-of-band message, threads give you an efficient way to structure its event handling where the only alternatives for a single-threaded process would be to either abruptly change context or put off handling the event to a more convenient time. The server portion of a client/server program often meets both of these criteria for concurrency: it must handle asynchronous requests and wait while retrieving and storing data in secondary storage.

If your application has been designed to use multiple processes, it's likely that it would benefit from threading. A common design model for a UNIX server daemon is to accept requests and fork a child image of itself to process the request. If the benefits of concurrency outweighed the overhead of using separate processes in the application, threading is bound to improve its performance because threads involve less overhead.

The remaining class of applications that can benefit from threads are those that execute on multiprocessing systems. Purely CPU-bound applications can achieve a performance boost with threads. A matrix-multiply program (or similar analytical program) with independent computational tasks but no excessive I/O requirements would not benefit from threads on a uniprocessing system. However, on a multiprocessor, this same application could speed up dramatically as the threads performed their computations in parallel.

As we'll see in Chapter 6, there are commonly three different types of Pthreads implementations. To take full advantage of a multiprocessing system, you'll need an implementation that's sophisticated enough to allow multiple threads in a single process to access multiple CPUs.

* A side benefit is that your code is ready to take advantage of multiprocessing systems in the future. Multiprocessing UNIX hosts are not restricted to exotic scientific number crunching anymore as two- to four-CPU server and desktop platforms have become commonplace.

Pthreads Programming

Computers are just as busy as the rest of us nowadays. They have lots of tasks to do at once and need some cleverness to get them all done at the same time. That's why threads are becoming a new model for programming. Microsoft Windows NT, the Distributed Computing Environment (DCE), and many UNIX-based operating systems support threads.

One advantage of most UNIX and DCE implementations is that they conform to a recently ratified POSIX standard (originally 1003.4a, now 1003.1c), which allows your programs to be portable between them. The POSIX threads standard, commonly known as Pthreads, is supported by most UNIX-based operating systems.

With threads programming, multiple tasks run concurrently within the same program. They can share a single CPU as processes do or take advantage of multiple CPUs when available. They provide a clean way to divide the tasks of a program while sharing data.

In this book you will learn when to use threads and how to make them efficient. The book delves into performance issues, comparing threads to processes, contrasting kernel threads to user threads, and showing how to measure speed. It describes in a simple, clear manner what all the advanced features are for and how threads interact with the rest of the UNIX system.

Topics include:

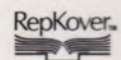
- Basic design techniques
- Mutexes, conditions, and specialized synchronization techniques
- Scheduling, priorities, and other real-time issues
- Cancellation
- UNIX libraries and re-entrant routines
- Debugging tips
- Measuring performance
- Special considerations for the Distributed Computing Environment (DCE)
- Full integration of threads with UNIX, including process management, library issues, and signals
- Many programming examples and models
- A Pthreads quick reference guide



X004H6BO1D

PThreads Programming: A POSIX Standard for
New

9 781565 921153



Printed on Recycled Paper

ISBN 1-56592-115-1