

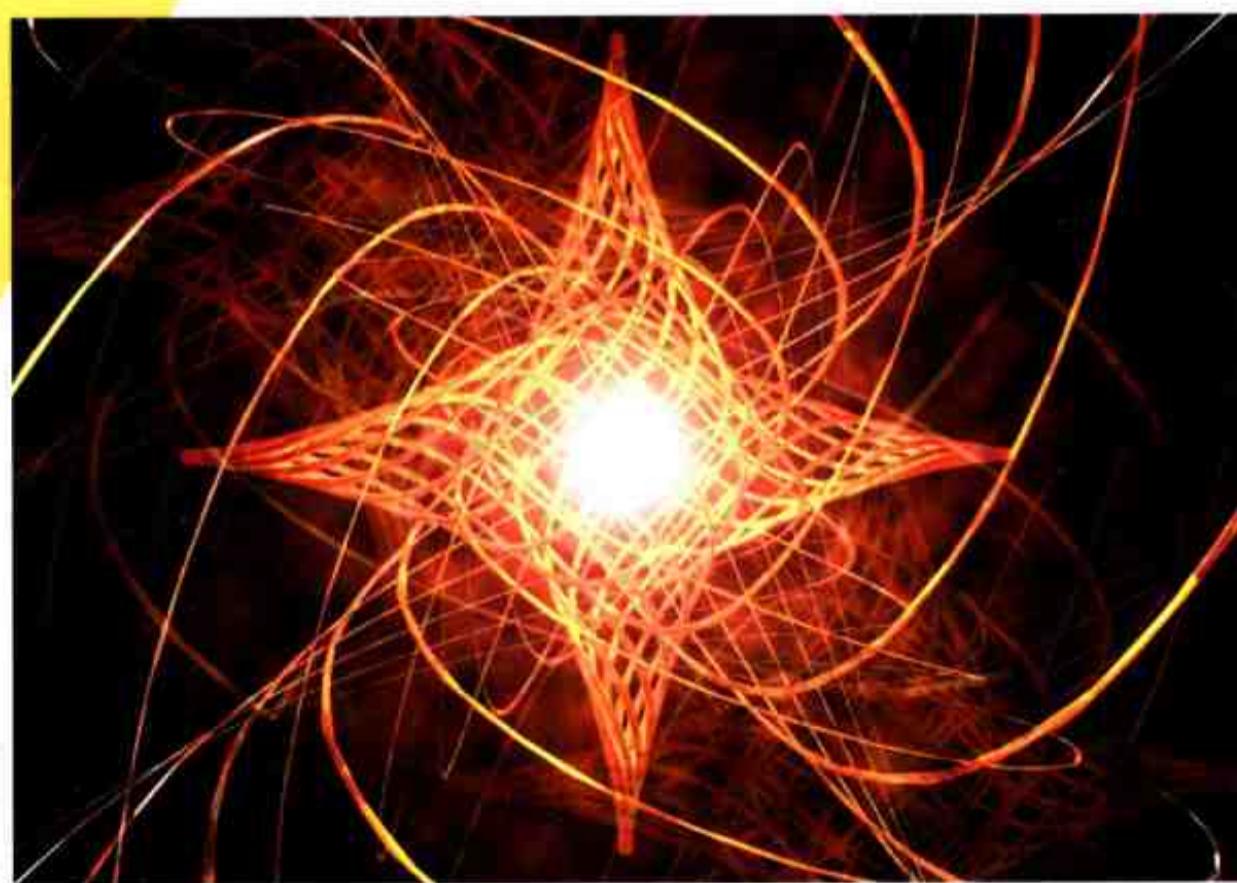
Programming With POSIX Threads

POSIX

多线程程序设计

[美]David R. Butenhof 著

于磊 曾刚 译



1.1

8



Addison
Wesley



中国电力出版社

www.infopower.com.cn

POSIX多线程程序设计

通过学习这本实用的参考书，你将理解有关线程的坚实基础，并学会如何将这一强大的编程模型应用到实际工作中。

多线程编程的主要优势在于通过利用多处理器的并行数字运算能力，或者通过在代码中自动使用I/O并发功能，以达到同时执行多项任务的目的，即使是在单处理器机器上。结果是：程序运行更快、响应速度更快、通常更容易维护。多线程编程尤其适用于网络编程，可帮助削弱慢速网络I/O带来的瓶颈问题。

本书深入描述了IEEE的开放系统接口标准POSIX (Portable Operating System Interface) 线程，通常称为Pthreads标准。本书的读者对象是有经验的C语言程序员，并假设以前没有线程知识。本书首先解释了基本的线程概念：异步编程、线程的生命周期和同步机制；然后讨论了一些高级话题，包括属性对象、线程私有数据和实时调度。第7章讨论了barrier、读/写锁、工作队列管理器，以及如何利用已有函数库。此外，本书还涉及了多线程编程者最头痛的问题——调试，并给出了如何从一开始就避免错误和性能问题的有价值的建议。

本书使用了大量注释过的实例来解释实际的概念，还包括Pthreads的微型参考和对标准化进程的展望。

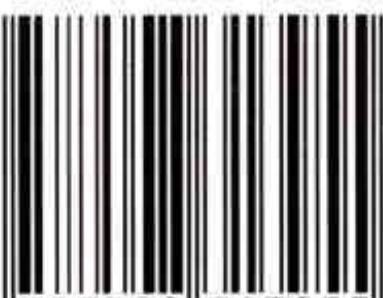
David R. Butenhof，是一个公认的Pthreads权威人士，深入参与过IEEE的POSIX 标准和X/OPEN线程扩展的制订。作为DEC公司的工程师，他还是Digital的线程体系的主要构架师和开发者，在Digital UNIX 4.0上设计并实现了大量的Pthreads接口。

新书推荐

深入C++系列最新图书

- ◆ More Effective C++中文版
- ◆ 泛型编程与STL
- ◆ Exceptional C++中文版
- ◆ C++经典问答
- ◆ C++设计新思维（影印版）
- ◆ Effective C++（影印版）

ISBN 7-5083-1395-X



9 787508 313955 >

ISBN 7-5083-1395-X

定价：39.00元

Programming
With POSIX
Threads

2B748

POSIX

多线程程序设计

[美]David R. Butenhof 著
于磊 曾刚 译

中国电力出版社

内 容 提 要

本书深入描述了 IEEE 的开放系统接口标准—POSIX 线程，通常称为 Pthreads 标准。本书首先解释了线程的基本概念，包括异步编程、线程的生命周期和同步机制；然后讨论了一些高级话题，包括属性对象、线程私有数据和实时调度。此外，本书还讨论了调度的问题，并给出了避免错误和提高性能等问题的有价值的建议。本书使用了大量注释过的实例来解释实际的概念，并包括 Pthreads 的简单索引和对标准化的展望。

本书适合有经验的 C 语言程序员阅读，也适合多线程编程人员参考。

图书在版编目 (CIP) 数据

POSIX 多线程程序设计 / (美) 布滕霍夫 (Butenhof,D.R.) 著;
于磊, 曾刚译. —北京: 中国电力出版社, 2003
ISBN 7-5083-1395-X

I.P... II.①布...②于...③曾... III.程序设计 IV.TP311.1

中国版本图书馆 CIP 数据核字 (2002) 第 110540 号

著作权合同登记号 图字: 01-2002-0712 号

Authorized translation from the English language edition, entitled Programming with
POSIX Threads by David R.Butenhof, published by Addison-Wesley, Copyright©1997

All rights reserved. NO part of this book may be reproduced or transmitted in any
form or by any means, electronic or mechanical, including photocopying, recording or by
any information storage retrieval system, without permission from the Publisher.

CHINESE SIMPLIFIED language edition published by China Electric Power Press
Copyright©2003

本书由培生集团授权出版。

中国电力出版社出版、发行

(北京三里河路 6 号 100044 <http://www.infopower.com.cn>)

汇鑫印务有限公司 印刷

各地新华书店经售

*

2003 年 4 月第一版 2003 年 4 月北京第一次印刷

787 毫米×1092 毫米 16 开本 20.75 印张 505 千字

定价 39.00 元

版 权 所 有 翻 印 必 究

(本书如有印装质量问题，我社发行部负责退换)

序 言

本书是有关“线程”(thread)和如何使用“线程”的。在计算机中，“线程”是一种能够实现某种功能的基本软件单元。线程比传统的进程(process)更小巧、更快捷、更易操作。实际上，一旦在操作系统中引入线程，就可以将进程看作包含了数据地址空间、文件和一个(或多个)数据处理线程的综合应用。

使用线程构建的应用程序能够更加有效地利用系统资源，使用户的界面更加友好，在多处理器系统中不但运行十分快速，而且更加易于维护。为达到上述目的，你只需要在程序中添加相应的几个简单函数调用，即可调整成另一种编程思路。通过仔细阅读本书，我希望能够帮助你实现上述目标。

本书讲述的线程模型通常被称为 Pthreads，或者 POSIX threads，更正式的名称应该是 POSIX 1003.1c-1995 标准。随后还将提供一些其他的名称，不过目前你只需记住 Pthreads 就够了。

在写本书时，SUN 公司的 Solaris、Digital 公司的 Digital UNIX、SGI 公司的 IRIX 已经支持 Pthreads。其他一些主要的商用 UNIX 操作系统，像 IBM 公司的 AIX 和 HP 公司的 HP-UX，不久也将支持线程模型，也许在你阅读本书的时候已经支持。Pthreads 也已经在 Linux 和其他 UNIX 系统中实现。

在个人电脑市场，微软公司的 Win32 编程接口和 IBM 的 OS/2 都支持线程编程。这些线程模型与 Pthreads 模型之间存在着一定的区别。为了有效地使用它们，首先必须理解并发、同步和调度等概念，剩下的就是语法和样式的问题，一个经验丰富的程序员可以适应这些模型中的任何一个。

线程模型已经很成功地在应用领域中广泛运用，下面仅是其中的一些：

- 有大规模科学计算的程序
- 能够充分利用多处理器系统的高性能程序和库代码
- 能被多线程程序使用的库代码
- 实时应用程序和库代码
- 对慢速外设(如网络和人类)执行输入输出操作的应用程序和库代码

读者对象

本书适合熟悉在 UNIX 系列操作系统上使用 ANSI C 开发代码的高级程序员阅读，并不要求具有线程或其他形状异步编程经验。第 1 章介绍有关概念和术语，使你能够继续阅读

本书后续部分，建议你不要跳过。

在阅读过程中，你将发现关于线程各方面的有趣比喻和实例。最后我希望你能够自己独立地使用线程编程。好了，祝你线程之旅愉快。

关于作者

我从一开始就参与 Pthreads 标准的有关工作，虽然最初的几次会议我没有参加。最后，我被迫在犹他州的雪鸟滑雪场的防雪崩掩体中度过了一周，观看来自世界各地的代表们向他们的滑雪板上涂蜡。我本以为这是一个十分正式、乏味的会议，所以我没有带自己的滑雪板，只能租用滑雪设备。

在 Pthreads 标准最后投票阶段，我同其他几个 POSIX 工作组设计线程同步接口和多处理器应用。我也帮助定义了 Aspen 线程扩展规范，该规范已经成功应用于 X/Open XSH5 中。

我曾在 DEC 公司工作数年，从麻省分部到新罕布什尔州分部。我是 DEC 公司线程架构的创始人之一，并在 Digital UNIX4.0 上设计并实现了大部分的 Pthreads 线程接口。我还帮助人们开发、调试线程代码超过八年之久。

我的一个不成文的座右铭是“并发使生活更美好”。线程不是面包片，程序员也不是面包师，所以我们只做能够做的事情。

致谢

可能读者并不关心这部分内容，但确实是我和朋友们以及本书合作者希望见到的。如果你是一个好奇的读者，请务必读下去。

尽管本书封面上只有我一个人的署名，但像本书这样的项目是不可能完全由一个人来完成的。因为我了解很多线程知识，至少在线程通信方面相当在行，所以我也可能不需任何帮助写出一本关于线程的书。但结果是，本书要比假设的那本书更好。

首先要感谢的是我的经理 Jean Fullerton，他给我时间并鼓励我在工作时写书。感谢 DEC threads 组的其他同仁，他们是：Brian Keane、Webb scales、Jacqueline Berg、Richard Love、Peter Portante、Brian Silver、Mark Simons 和 Steve Johnson。

感谢 Garret Swart，当他还在 Digital 系统研究中心工作时，就让我们了解 POSIX 标准。感谢 Nawaf Bitar，他和 Garret 一起通宵工作，实现了 Pthreads 的第一个草案，并且不遗余力地推广 POSIX 线程标准，让每个人都理解线程到底是个什么东西。没有 Garret，特别是如果没有 Nawaf，Pthreads 可能不会存在，至少不会像现在这么好（缺乏完美并不是他们的责任——生活本来如此）。

感谢参与设计 cma、Pthreads、UNIX98、DCE threads 和 DECthreads 的所有人的帮助，他们是：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、Finnbarr P. Murphy、Bill Noyce、Simon Patience、Harold Seigel、Al Simons、Jim Woodward 和 John Zolnowsky。

特别感谢所有耐心审阅本书草稿的人们，他们是：Brian Kernighan、Rich Stevens、Dave Brownell、Bill Gallmeister、Ilan Ginzburg、Will Morse、Bryan O’ Sullivan、Bob Robillard、Dave Ruddock 和 Bil Lewis。感谢对结构和细节提出改进意见和建议的人们：Devang Shah 和 Bart Smaalders 帮助回答了一些有关 Solaris 的问题，Bryan O’Sullivan 建议使用“舀水的程序员”的比喻。

感谢 Addison Wesley Longman 的 John Wait 和 Lana Langlois，他们耐心地等待并鼓励第一次写书的我努力写好这本书。感谢 Pamela Yee 和 Erin Sweeney，他们管理了本书的整个出版过程。感谢所有帮助过我的人们。

感谢我的妻子 Anne Lederhos 和我的女儿 Amy、Alyssa，感谢她们对我的支持和陪伴。感谢 Charles Dodgson (Lewis Carroll)，他在其经典小说 *Alice's Adventures in Wonderland* (《艾丽丝漫游仙境》)、*Through the looking-Glass* (《镜中漫游》) 和 *The Hunting of the Snark* (《捕猎蛇鲨》) 中写了大量的关于线程编程的事情 (译者注：是指小说中描写的多人之间的协调、并发工作，作者认为与线程间的同步和协调具有相似的含义)。

目 录

序 言

第1章 概述	1
1.1 酋长的程序员	2
1.2 术语定义	3
1.3 异步编程是直观的	7
1.4 关于本书的实例	10
1.5 异步编程举例	11
1.6 线程的好处	16
1.7 线程的代价	20
1.8 选择线程还是不用线程	22
1.9 POSIX 线程概念	23

第2章 线程	28
2.1 建立和使用线程	28
2.2 线程的生命周期	32

第3章 同步	37
3.1 不变量、临界区和谓词	37
3.2 互斥量	38
3.3 条件变量	59
3.4 线程间的内存可见性	74

第4章 使用线程的几种方式	81
4.1 流水线	81
4.2 工作组	89
4.3 客户/服务器	101

第5章 线程高级编程	111
5.1 一次性初始化	111
5.2 属性	114

5.3 取消	120
5.4 线程私有数据	137
5.5 实时调度	147
5.6 线程和核实体	161
第 6 章 POSIX 针对线程的调整	167
6.1 fork	167
6.2 exec	173
6.3 进程结束	173
6.4 stdio	174
6.5 线程安全的函数	178
6.6 信号	182
第 7 章 Real code	206
7.1 扩展同步	206
7.2 工作队列管理器	231
7.3 对现存库的处理	243
第 8 章 避免调试的提示	248
8.1 避免不正确的代码	249
8.2 避免性能问题	259
第 9 章 POSIX 多线程快速参考	263
9.1 POSIX 1003.1c-1995 选项	263
9.2 POSIX 1003.1c-1995 限制	264
9.3 POSIX 1003.1c-1995 接口	265
第 10 章 标准化过程展望	303
10.1 X/Open XSH5 [UNIX98]	303
10.2 POSIX 1003.1j	311
10.3 POSIX 1003.14	316
参考文献	317
因特网上的线程资源	320

概 述

*"The time has come," the Walrus said,
"To talk of many things;
Of shoes—and ships—and sealing wax—
Of cabbages—and kings—
And why the sea is boiling hot—
And whether pigs have wings."*

—Lewis Carroll, *Through the Looking-Glass*

在计算机专用术语中，线程是指机器中连续的、顺序的属性集合。一个线程包含执行一系列机器指令所必须的机器状态，包括当前指令位置、地址和数据寄存器等。

一个 UNIX 进程可以理解为一个线程加上地址空间、文件描述符和其他数据。某些 UNIX 版本支持“轻量级”或“变量级”进程，以便可以从进程中剔除部分或者所有数据，从而实现高效性能。既然线程和轻量级进程都需要地址空间、文件描述符等数据，那么区别何在？区别在于多个线程可以共享一个地址空间，而做不同的事情。在多处理器系统中，一个进程中的多个线程可以同时做不同的工作。

当计算机还活在玻璃洞穴中时（译者注：指计算机发展初期），需要处理事先准备好的穿孔卡片。整个外部世界都在等待计算的结果，顶多可能听到程序员的抱怨声。但是外部世界并不是一次只做一件事情，逐渐地，计算机开始模拟这种实际模式，增加多程序设计、多重处理、分时共享、多处理器系统的能力，最终，实现了线程。

线程能够帮助你的应用程序走出洞穴。Pthreads 则能帮助你以一种优雅、高效、可移植的方式完成这个工作。本章简单介绍理解和使用线程所需要的基本知识，其他章节则会针对各个环节做进一步的详细解释。

1.1 节给出了包含多个比喻的故事，以此说明线程的工作模式。这个故事并没有什么特别的，但在你理解我所讲的程序员和水桶的含义之前，可能显得有点怪。

1.2 节给出了本书使用的基本概念和术语。其中最重要的一个概念需要在此特别介绍，也与全书会对一些重点特别强调的习惯是一致的：

异步：

任何两个彼此独立运行的操作是异步的。

1.3 节论述了在一般意义上你是如何经历异步编程的，无论你是 UNIX 程序员和用户，还是作为现实世界中的人。我不敢说异步编程十分容易，但是它的基本概念十分简单和自然，以至于在试图将它应用到软件编程之前，你几乎是不会考虑到它。

1.4 节简单讨论了本书中的实例。你会了解构造这些实例的假设条件以及应该如何在线获取这些源代码。

从某种程度上讲，线程只是构造异步应用程序的另一种方式，但是它却比其他用来构造异步程序的模型更有优势。1.5 节使用了几种不同的编程模型来实现一个简单的闹钟实例，以此来说明线程的几个优势。该实例可让你立刻看到线程的运转，以及对构建该应用程序的几个 Pthreads 接口的简述。

在已经了解线程的各个方面之后，1.6 节继续列举了线程模型的基本优点。

尽管有很多理由使用线程，但使用线程也是有代价的。1.7 节对照以上各节，列举了使用线程的一些可能代价。不过，归根结底，是告诉你需要了解线程的工作方式，然后很好地使用它，这并不像某些人说得那么难。

虽然已经了解了线程的一些基本好处和代价，很显然你不会马上把线程用到所有的应用程序或库函数中。在 1.8 节中，对于“要线程还是不要线程”的问题，本书力图指导读者在不同的情况下做出恰当的回答。

至此，你就了解了线程是什么、做什么、何时使用它们。除了简单的实例外，你还没有接触到 Pthreads 包含的编程接口的任何细节问题。1.9 节列出了线程领域中的一些里程碑事件，可以给你一个正确的引导。该节最重要的部分是 1.9.3，其中讲述了 Pthreads 模型的报错机制，这与 UNIX 和 POSIX 的其他部分有些不同。

1.1 留水的程序员

*This was charming, no doubt: but they shortly found out
That the Captain they trusted so well
Had only one notion for crossing the ocean,
And that was to tingle his bell.*

—Lewis Carroll, *The Hunting of the Snark*

三个程序员乘一艘小船出海，他们尽情享受着阳光和海风，离岸越来越远。一会儿，天色变暗，暴风雨降临，小船在狂风骇浪中颠簸。当暴风雨终于减缓的时候，他们已经失去了桅杆和帆，而且小船也裂了一个小缝。他们看不到陆地的影子。

小船上配备了食物、水、桨和一个水桶。程序员们开始工作。一个人划船、监视船舱的渗水情况，其他两个人则可能去睡觉、检测水位、寻找陆地或者其他船只。

空闲的程序员可能看到水位上升，开始用桶舀水。当两个程序员都醒来的时候，可能同时发现了水位的上升，都要去拿水桶。显然只有一个人首先拿到水桶，而另

一个人则不得不等待。

如果两个人都睡着了，而划船的人认为该舀水的时候，通常他会推醒其中一个人而让另一个人继续睡觉。但是，如果划船的人心情不好，则可能大声喊叫，将两个人全都吵醒。在让其中一个继续该做的工作时，另一个可以继续睡觉。

当划船的人疲劳时，他可以唤醒其中一个接替他的工作，然后马上睡觉，直到他被再次唤醒。这样，三个程序员就可以坚持他们的旅程了。

那么，舀水的程序员和线程到底有什么关系呢？答案是，上述故事可用来比喻线程编程模型的思想。后面我们还会给出更多的比喻，甚至继续扩展这个故事。不过，目前让我们先考虑以下几个基本点：

一个程序员就是一个能独立活动的实体。在此处，程序员代表线程。尽管一个线程无法和一个程序员相比（后者是工程师、数学家和艺术家的神奇组合，没有什么计算机可以媲美），然而，在代表编程模式中的活动元素方面，还是足够了。

舀水的桶和划船的桨就是一次只能由一人拥有的令牌（token），它们也可以被理解为共享数据，或者是同步对象。顺便提一句，在 Pthreads 中的同步对象称为互斥量（mutex）。

轻推和喊叫是与同步对象相关的通信机制，个体等待这些事件的发生。Pthreads 中提供了条件变量（condition variable），可以通过信号和广播来指示共享数据的状态变化。

1.2 术语定义

*"When I use a word," Humpty Dumpty said, in rather a scornful tone,
"it means just what I choose it to mean—neither more nor less."*

—Lewis Carroll, Through the Looking-Glass

本书使用严格定义的术语，读者可能不太熟悉，除非已经具有了一些并行或异步编程的经验。即使如此，业界中使用的一些术语依然有混淆甚至矛盾的地方，不利于相互交流。在开始学习之前，首先需要对术语的含义达成共识，因为我是作者，所有应该遵从我的定义（多谢）。

1.2.1 异步

异步（asynchronous）表明事情相互独立地发生，除非有强加的依赖性。生活也是异步的。依赖性是大自然补充的，彼此互不相干的事件可以同时发生。没有桨，程序员无法划船；没有水桶，程序员也不能高效地舀水。但是，在一个程序员可以用桨划船的，同时，另一个程序是可以用桶舀水。不过，传统的计算机系统让所有的事件依次发生，为了让它们并发，程序员必须使用额外的方法。

异步带来的最大复杂性就是：如果你没有同时执行多个活动，那么异步就没有

什么优势。如果你开始了一个异步活动，然后什么也不做等待它结束，则你并没有从异步那儿获得太多好处。

1.2.2 并发

在字典中，并发（concurrency）的意思是指事情同时发生。在本书中，并发是指让实际上可能串行发生的事情好像同时发生一样。并发描述了单处理器系统中线程或进程的行为特点。在 POSIX 中，并发的定义要求“延迟调用线程的函数不应该导致其他线程的无限期延迟”。

并发操作之间可能任意交错，导致程序相互独立地运行（一个程序不必等到另一个结束后才开始运行），但是并发并不代表操作同时进行。然而，并发让程序能利用异步能力的优点，在无关操作运行的过程中继续工作。

大部分程序拥有异步特点，但可能不很明显。例如，用户更喜欢异步界面，他们希望在思考的同时让程序接受他们发出的命令，即使程序还没有处理完上一个请求。当窗口界面提供多个分离的窗口时，难道人们不是直觉地希望它们能够异步地工作吗？没有人喜欢看到“忙指针”。Pthreads 提供了并发和异步的能力，二者结合就可以容易地写出高效且有良好响应速度的程序。你的程序可以一面等待慢速的外 I/O 作，一面利用多处理器系统的优势并行地进行其他计算。

1.2.3 单处理器和多处理器

“单处理器（uniprocessor）”和“多处理器（multiprocessor）”是相当直接的两个词，不过为了避免混淆，我们还是要定义它们。单处理器是指一台计算机只有一个编程人员可见的执行单元（处理器）。对于拥有超标量体系结构、向量或者其他数学或 I/O 协处理器的单一通用处理器，我们仍然把它当成单处理器。

多处理器是指一台计算机拥有多个处理器，它们共享同一个指令集和相同的物理内存。虽然处理器不必同等地访问所有的物理内存，但是每一个应该都能访问大部分内存。就本书而言，一个大型并行处理系统（MPP）可能是也可能不是我们说的多处理器系统。许多 MPP 系统是属于多处理器系统的，因为所有的处理器都能访问所有的物理内存，即使访问时间的变化会很大。

1.2.4 并行

并行（parallelism）指并发序列同时执行。换言之，软件中的“并行”和日常语言中的“并发”是相同的意思，而区别于软件中的“并发”。并行的补充含义是指事情在相同的方向上独立进行（没有交错）。

真正的并行只能在多处理器系统中存在，但是并发可以在单处理器系统和多处理器系统中都存在。并发能够在单处理器系统中存在是因为并发实际上是并行的假象。并行要求程序能够同时执行多个操作，而并发只要求程序能够假装同时执行多个操作。

1.2.5 线程安全和可重入

线程安全是指代码能够被多个线程调用而不会产生灾难性结果。它不要求代码在多个线程中高效地运行，只要求能够安全地运行。大部分现行函数可以利用 Pthreads 提供的工具——互斥量、条件变量和线程私有数据，实现线程的安全。不需要保存永久状态的函数，可以通过整个函数调用的串行化来实现线程安全。比如，在进入函数时加锁，在退出函数时解锁。这样的函数可以被多个线程调用，但一次只能有一个线程调用它。

更有效的方式是将线程安全函数分为多个小的临界区。这样就允许多个线程进入该函数，虽然不能同时进入一个临界区。更好的方式是将代码改造为对临界数据的保护而不是对临界代码的保护，这样就可以令不会同时访问相同临界数据的线程完全并行地执行。

举例说，将字符写入标准 I/O (stdio) 缓冲区的 putchar 函数可以通过临界代码实现线程安全。即，putchar 可能先锁住一个互斥量，写数据，然后解锁互斥量。你可以在两个线程中调用该函数，这将是安全的。但是一次只能有一个线程写数据，另一个必须等待，哪怕它们向不同的 stdio 流写入数据。

正确的方法是将互斥量和数据流相关联，保护数据而不是代码。这样，只要线程向不同的数据流写入数据，就可以并行调用 putchar 函数。更重要的是，所有访问同一个数据流的线程，都可以使用一个互斥量来安全地协调它们的访问。

“可重入”有时用来表示“有效的线程安全”，即通过采用比将函数或库转换成一系列区域更为复杂的方式使代码成为线程安全的。通过引入互斥量和线程私有数据可以实现线程安全，但通常需要改变接口来使函数可重入。可重入的函数应该避免依赖任何静态数据，最好避免依赖线程间任何形式的同步。

通常，函数可以将状态保存在环境结构中，由调用者来负责状态数据的同步。例如，UNIX 中的 readdir 函数依次返回队列中的每一个目录项。为了让 readdir 函数是线程安全的，你可以添加一个互斥量，每次当 readdir 被调用时，锁住该变量，在函数返回时解锁。另一种办法，也是 Pthreads 对其 readdir_r 函数的做法，是避免函数内部的任何锁操作，而是让调用者在搜索目录时分配一个数据结构来保存 readdir_r 的环境。

乍看上去，好像仅仅是让调用者来执行本应由 readdir_r 来做的工作。但是，切记只有调用者才知道数据如何使用。如果只有一个线程使用保存的环境数据，则不需要任何的同步机制。即使多个线程共享数据，调用者也能够提供更有效的同步机制。例如，如果环境数据用互斥量来保护，应用程序也可以用互斥量来就像保护其他数据。

1.2.6 并发控制功能

并发系统必须提供基本的核心功能，包括创建并发执行的环境，并在你的库或

应用程序中对其操作进行控制。以下是任何并发系统都应该具有的基本功能：

1. “执行环境”是并发实体的状态。并发系统必须提供建立、删除执行环境和独立维护它们状态的方式。它必须能够不时地保存好一个执行环境，而去执行另外一个环境，例如，当程序需要等待外部事件的时候。它必须能够在随后的某个时刻重新恢复环境，从上一次执行时的位置开始，还有相同的寄存器数据。
2. “调度”决定在某个给定时刻该执行哪个环境（或环境组），并在不同的环境中切换。
3. “同步”为并发执行的环境提供了协调访问共享资源的一种机制。这里的同步和字典中的含义是对立的。通常意义上，同步是指“同时发生”，而这里我们是指那些“阻止同时发生”的事情。在辞典中你可能发现“同步”的同义词“协调”——同步就是让线程协调地完成工作的机制。本书使用“同步”这一术语，因为这是很流行的用法。

有很多种方式来提供上述功能，但总是表现为某种形式。本书采用的特殊选择是由本书的主题 Pthreads 决定的。表 1.1 列出了上述三方面的几个不同的实例。

表 1.1 执行环境、调度器和同步

环境	执行环境	调度	同步
交通	汽车	红绿灯	转弯信号和刹车灯
UNIX（无线程）	进程	优先级	等待和管道
Pthreads	线程	策略、优先级	条件变量和互斥量

系统调度功能可以让线程执行直到它自愿让出 CPU 给其他线程为止（一直运行到被阻塞），也可以提供时间片，使每个线程被迫周期性地让出 CPU，以使其他线程得以运行（轮转“round-robin”）。可以提供多种调度策略，让程序针对每个线程实现的功能来选择各线程的调度方式。还可以实现“分类调度器”，来描述线程间的依赖关系，例如，可以确保紧耦合的并行算法的各成员同时被调度。

同步可以通过很多种方式实现。其中最常见的是互斥量、条件变量、信号量和事件。还可以通过消息传递机制，如 UNIX 管道、SOCKET、POSIX 消息队列，或者是其他异步进程间（本地或者网络）的通信协议。任何通信协议里都包含某种相同的同步机制，因为如果没有同步，则传送数据将导致混乱而不是通信。

线程、互斥量和条件变量是本书的主要话题。目前，你只要记住线程是计算机中的可执行单元；互斥量阻止线程间发生不可预期的冲突；一旦避免了冲突，条件变量让线程等待直到可以安全地执行。互斥量和条件变量都是用来同步线程间操作的。

1.3 异步编程是直观的……

*"In most gardens," the Tiger-lily said,
"they make the beds too soft—so that the flowers are always asleep."
This sounded a very good reason, and Alice was quite
pleased to know it.
"I never thought of that before!" she said.*
—Lewis Carroll, *Through the Looking-Glass*

如果你没有经历过传统的实时编程，对异步编程方式可能会有些陌生，不过你可能一直在使用异步编程技术。例如，你可能使用过 UNIX，即使作为普通用户，通用的 UNIX 脚本（从 sh 到 ksh）已经设计成异步工作方式。而且，从你降临到这个世界开始，你就一直在使用异步“编程”技术。

大部分人在理解了形式化的、严格的复杂定义之后，就会对异步行为有了比预期的更加深入的理解。

1.3.1 因为 UNIX 是异步的

在任何 UNIX 系统中，进程彼此间异步地执行，即使只有一个处理器。的确，直到现在要给 UNIX 写一个异步执行的程序还是比较困难的，但是 UNIX 一直让你相当容易地执行异步操作。当输入脚本命令时，实际上启动了一个独立的程序，如果在后台运行该程序，则它和 shell 就是异步运行。当你使用管道（pipe）将一个命令的输出重定位到另一个命令的输入时，就是启动了几个独立的程序，它们彼此通过管道来同步。

| 时间就是同步机制

在很多情形中，你会在一系列过程间提供同步，即使你可能没有意识到。例如，你首先编辑源文件，然后运行编译工具，不可能首先或者甚至同时编译代码（与编辑代码相比），这就是实际中的同步。

| UNIX 管道和文件可以是同步机制

在其他情形中，可能用到更为复杂的软件同步机制。当在脚本环境中输入 ls | more，将 ls 命令的输出重定向到 more 命令的输入时，实际上是在通过设定数据依赖性来描述命令间的同步。shell 环境立刻启动两个命令，但是 more 命令在通过管道获得 ls 命令输出（对 more 命令而言是输入）之前，是不能产生任何输出的。两个命令可以并发地执行（在多处理器系统中甚至可以并行地执行），ls 提供数据，more 处理数据，彼此互相独立。如果管道缓冲足够大，则 ls 命令可能在启动 more 命令之前结束，但是 more 命令决不会在 ls 命令之前结束。

有些 UNIX 命令本身提供同步机制。例如，命令 cc -o thread thread.c

可能涉及多个进程。`cc` 命令作为 C 语言环境的前端，会首先运行过滤器来展开处理器相关命令（像`#include` 和`#if`），再运行编译器来将代码转为中间形式，再运行优化器来重排中间结果，再运行汇编器来翻译中间结果为目标代码，最后运行加载器将目标代码翻译成可执行的二进制文件。与`ls | more` 命令一样，所有上述过程可以同时启动，由管道或者中间文件来同步它们。

UNIX 进程包含了需要执行代码的所有信息，所以可以异步执行。操作系统可以保存一个进程的状态转而执行其他进程而不会有任何不良影响。任何通用意义的异步实体都必须包含让操作系统可以任意切换它们的足够状态。但 UNIX 进程还包括与执行环境不直接相关的附加状态数据，如地址空间和文件描述符。

线程就是进程里足以执行代码的部分。在大多数计算机系统中，这意味着线程应包括以下内容：当前指令位置指针（通常称为计数器或 PC）、栈顶指针（SP）、通用寄存器、浮点或地址寄存器。线程可能还包括像处理器状态和协处理器寄存器等数据。线程不包括进程中的其他数据，如地址空间和文件描述符。一个进程中的所有线程共享文件和内存空间，包括程序文本段和数据段。

| 线程比进程简单

可以将线程理解为一种减负的进程，精干、简单、快速。系统在线程间切换要比在进程间切换快得多，这很大程度上得益于线程间共享地址空间（包括代码段、数据段和堆栈）的事实。

当系统在进程间切换时，与进程相关的所有硬件状态都会失效。有些可能随环境切换过程而改变，如数据缓存和虚拟内存转换入口可能需要刷新。即使不必立刻刷新，这些数据对于新进程而言也是无用的。每一个进程有独立的虚拟内存空间，但是同一进程中的线程却共享相同的地址空间和其他进程数据。

线程使程序不同部分间的高带宽通信更加容易。你不再担心类似管道之类的消息传送机制或者在不同地址空间中共享内存指针的一致性。同步更快，编程更加自然。如果你打开或创建一个文件，则所有的线程都可以使用它。如果你通过调用`malloc` 分配了一块动态数据结构，就可以把地址传给其他线程，让其他线程引用该结构。线程可让我们更方便地利用并发的优点。

1.3.2 因为世界是异步的

在刚开始考虑异步模式时，可能显得比较笨拙，但有一些实践之后就会变得自然。要努力克服那种以为任何事都是顺序发生的不自然的想法：在单车道上一次只能通过一辆汽车，而在两车道上一次就可以通过两辆汽车。在计算机编译代码的时候，你可以出去喝一杯咖啡，并且完全相信没有你它会正常运行。世界上到处都存在并行，这也是人们期望的。

商场中的一排收款机在并行为顾客服务；每一列中的顾客依次等待它们的服务。你可以增加队列以改善流通量，只要有足够的收款机为顾客服务，并且有足够的

的顾客等待服务。为每个收款机开两个队列可以缩短队列的长度以减少混乱，但是并不会更快地提供服务。为两个顾客开三台收款机可能看上去不错，但其实只是对资源的浪费。

在生产线上，工人们并行地完成整个工作的不同部分，并把结果向下传。在生产线上增加站点可以分割重负荷的站点，减少下面站点的等待时间，可以改善系统性能。即，改善一个站点就可以使之产生更多的中间件给下一站点处理。

在办公室里，每个项目都分派给一个专门人员或部门，像市场部、管理部、工程部、打字组、销售、技术支持等。每个专业人员代表顾客或者其他专业人员独立地处理他的项目，在结束时以某种形式汇报。如果一个专业人员有太多的工作，则增加一个专业人员或者减少其负责范围将改善系统性能，而不应该让一些人闲着游戏却让其他人忙得晕头转向。

汽车在高速路上并行前进，它们可以有不同的速度，可以超车、独立进入或离开高速路。司机们必须遵守某种规则来避免发生撞车事件。除了速度限制和交通标志外，更多地要自愿遵守“道路规则”。相似地，线程也必须遵守约定的规则来保护程序，否则就有被送进线程医院中就诊的危险。

软件可以像我们在日常生活中的活动一样应用并行。当有多个能工作的“东西”时，你自然希望它们都能同时工作。一个多处理器系统可以执行多个计算，任何分时共享系统在等待外设响应的同时执行计算。软件并行也会像实际生活中一样带来复杂性或者问题，而且可能并不容易解决。需要足够的线程而不是太多的线程，需要足够的通信而不是太多的通信。线程编程的一个关键点就是学会在不同的情况下作出恰当的取舍。

每个线程可以像收款机一样执行相同的工作，也可以像生产线上的工人一样依次执行工作的不同部分。每个线程可以专用于特定的操作，并且代表其他线程反复执行该操作。可以将所有这些简单模式以各种方式组合在一起，例如，在并行生产线的某个环节上增加多个服务节点。

本书将向你介绍一些可能不太熟悉的概念，如互斥量、条件变量、竞争条件、死锁和优先级倒置等。线程编程可能让人感到畏惧或者不自然，我会在你通读本书的过程中解释这些概念。在你自己编写一些多线程代码后，就会在现实中找到许多与这些概念类似的现象。

如果你认为你已经锁住会话互斥量，别人不应该再中断你，此时你就开始真正理解线程编程了，这会帮你更容易地设计更好的线程代码^①。记住，如果某些事情在现实世界中没有意义的话，你也不要试图在程序里实现它。

^① 现在倒是休息一下的好时候，你可以读一会儿有利健康的小说。

1.4 关于本书的实例

本书所有实例都是完整的，都在 Digital UNIX 4.0d 和 Solaris 2.5 上实现并通过测试。

所有实例都会实现某个功能，但是很多实例并不做什么特别重要的工作。所有实例的目的都是为了展示线程管理和同步技术，它们在实际的应用程序中可能仅仅是额外的负担，但是却能很好地显示线程编程的各个细节。

本书的实例代码分节讨论，通常一节一个函数。源代码由头、尾标志块与正文隔离，标志块中包含文件名、函数名和节号。每行代码的左侧顶头标有行号。每节中的主要功能块都在源代码前面的段落中加以描述，这些描述段落的左侧顶头标有其描述的代码行号范围。例如：

1-2 这些行表明了大多数程序中包含的头文件。`<pthread.h>` 头文件声明了 Pthreads 函数用到的常量和函数原形。`errors.h` 头文件中包含了一些其他头文件和错误检查函数。1.9.3 节中列出了该头文件的全部代码。

■ sample.c part 1 sampleinfo

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

■ sample.c part 1 sampleinfo

错误检查应用在实例代码的各个位置，即，在每次函数调用时都会进行查错操作。只要你的程序编写得足够仔细，就并不需要这样做。有些专家建议只需在可能出现资源不足或者其他超出程序员控制的地方检查错误。我不同意他们的观点——除非你是那种从不犯错误的程序员。检查错误并不是那么无聊，还会省去你在调试程序中的许多麻烦。

你可以编译和运行本书所有代码，还可以从网址<http://www.awl.com/cseng/titles/0-201-63392-2/>上下载源代码。本书提供了一个 Make file 文件来编译实例代码，不过在不同的平台上可能需要稍加修改。在 Digital UNIX 平台上，用标志 CFLAGS= -pthread -std1 -w1 来编译代码，在 Solaris 系统中，用标志 CFLAGS=-D_REENTRANT -D_POSIX_C_SOURCE= 199506 -lpthread 来编译。可能你机器上的 Pthreads 库没有包含一些实例需要的接口函数，如 POSIX.1b 实时标准中提供的 clock_gettime 函数。需要使用 RTFLAGS 变量包含实时库，在 Digital UNIX 上让 RTFLAGS=-lrt，在 Solaris 系统上令 RTFLAGS=-lposix4。

在 Solaris 2.5 系统中，一些实例需要调用 `thr_setconcurrency` 函数来确保其运行正常，该函数使 Solaris 为进程提供额外的并发。在某些实例中，没有这个函数调用可能就根本无法运行，而在其他的实例中，可能会无法演示其全部的功能。

1.5 异步编程举例

*"In one moment I've seen what has hitherto been
Enveloped in absolute mystery,
And without extra charge I will give you at large
A Lesson in Natural History."*

—Lewis Carroll, *The Hunting of the Snark*

本节将通过一个简单的闹钟实例程序来演示基本的异步编程方法。这是一个有命令界面的闹钟，而且你不用花一分钱。由于本书的重点是讲述线程编程，所以在用户界面上不会花费太多力气。

该程序循环接受用户输入信息，直到出错或者输入完毕。用户输入的每行信息中，第一部分是闹钟等待的时间（以秒为单位），第二部分是闹钟时间到达时显示的文本消息。本书还提供了另外两个版本，一个使用多进程，一个使用多线程。我们将通过该例来比较各种方法。

1.5.1 基本的同步版本

包含头文件 `error.h`, `error.h` 中又包含了像`<unistd.h>`和`<stdio.h>`这样的头文件和本书所有实例都会使用到的报错宏定义。虽然本例中没有用到报错宏，但是为一致性起见，还是包含了这个头文件。

9-26 基本版本 `alarm.c` 是只有一个 `main` 函数的同步闹钟程序。`Main` 函数的大部分是一个循环，在里面处理简单的命令直到 `fgets` 返回 `NULL`（出错或者文件尾）。每行命令由 `sscanf` 函数解析成等待时间（`%d`, 数字序列）和打印的消息（`%64[^\\n]`, 最多 64 个字符）两部分。

■ alarm.c

```
1 #include "errors.h"
2
3 int main (int argc, char *argv[])
4 {
5     int seconds;
6     char line[128];
7     char message[64];
8
9     while (1) {
10         printf ("Alarm> ");
11         if (fgets (line, sizeof (line), stdin) == NULL) exit (0);
12         if (strlen (line) <= 1) continue;
13
14         /*
15          * Parse input line into seconds (%d) and a message
16          * (%64[^\\n]), consisting of up to 64 characters
17          * separated from the seconds by whitespace.
18         */
19 }
```

```

19     if (sscanf (line, "%d %64[^\\n]", 
20             &seconds, message) < 2) { 
21         fprintf (stderr, "Bad command\\n");
22     } else {
23         sleep (seconds);
24         printf ("(%d) %s\\n", seconds, message);
25     }
26 }
27 }
```

■ alarm.c

alarm.c 的问题是一次只能处理一个闹钟请求。如果你将闹钟设为 10 分钟 (600 秒) 等待时间，你就不能再让它过 5 分钟时闹铃。该程序实际上是把希望用异步方式实现的工作用同步方式实现了。

1.5.2 多进程版本

有多种异步实现该程序的办法，例如可以运行多个程序拷贝。其中一种方式就是为每个命令使用 fork 调用生成一个子进程，就像 alarm_fork.c 中那样。新版本是异步方式的——可以随时输入命令行，它们被彼此独立地执行。新版本并不比原先那个不错的版本复杂多少。

27~37 alarm.c 和 alarm_fork.c 的主要区别在于：新版的 main 函数没有直接调用 sleep 函数，而是创建了一个子进程，在子进程中异步地调用 sleep 函数（和 printf），而父进程则继续运行。

42~46 该版本的主要难点在于对所有已终止子进程的 reap。如果程序不做这个工作，则要等到程序退出的时候由系统回收，通常回收子进程的方法是调用某个 wait 系列函数。在本例中，我们调用 waitpid 函数，并设置 WNOHANG（译者注：父进程不必挂起等待子进程的结束）。如果有子进程终止，该函数调用回收该子进程的资源；如果没有子进程终止，该函数立即返回 pid=0。父进程继续回收终止的子进程直到没有子进程终止。当循环终止的时候，main 函数继续循环到行 13，读取下一个命令。

■ alarm_fork.c

```

1 #include <sys/types.h>
2 #include <wait.h>
3 #include "errors.h"
4
5 int main (int argc, char *argv[])
6 {
7     int status;
8     char line[128];
9     int seconds;
10    pid_t pid;
11    char message[64];
12
13    while (1) {
```

```
14     printf ("Alarm> ");
15     if (fgets (line, sizeof (line), stdin) == NULL) exit (0);
16     if (strlen (line) <= 1) continue;
17
18     /*
19      * Parse input line into seconds (%d) and a message
20      * (%64[^\\n]), consisting of up to 64 characters
21      * separated from the seconds by whitespace.
22      */
23     if (sscanf (line, "%d %64[^\\n]",
24                 &seconds, message) < 2) {
25         fprintf (stderr, "Bad command\\n");
26     } else {
27         pid = fork ();
28         if (pid == (pid_t)-1)
29             errno_abort ("Fork");
30         if (pid == (pid_t)0) {
31             /*
32              * In the child, wait and then print a message
33              */
34             sleep (seconds);
35             printf ("%d %s\\n", seconds, message);
36             exit (0);
37         } else {
38             /*
39              * In the parent, call waitpid() to collect children
40              * that have already terminated.
41              */
42             do {
43                 pid = waitpid ((pid_t)-1, NULL, WNOHANG);
44                 if (pid == (pid_t)-1)
45                     errno_abort ("Wait for child");
46             } while (pid != (pid_t)0);
47         }
48     }
49 }
50 }
```

■ alarm_fork.c

1.5.3 多线程版本

现在，让我们看另外一个闹钟程序 `alarm_thread.c`。它和多进程版本十分相似，只是使用线程而非子进程来实现异步闹钟。本例中用到了以下三个 Pthreads 函数：

- `pthread_create` 函数建立一个线程，运行由第三个参数 (`alarm_thread`) 指定的例程，并返回线程标识符 ID (保存在 `thread` 引用的变量中)。
- `pthread_detach` 函数允许 Pthreads 当线程终止时立刻回收线程资源。
- `pthread_exit` 函数终止调用线程。

4~7 数据结构 `alarm_t` 中定义了每个闹钟命令的信息，其中 `seconds` 中包含等待时间，`message` 中包含显示文本。

■ `alarm_thread.c`

part 1 definitions

```

1 #include <pthread.h>
2 #include "errors.h"
3
4 typedef struct alarm_tag {
5     int         seconds;
6     char        message[64];
7 } alarm_t;
```

■ `alarm_thread.c`

part 1 definitions

1~8 函数 `alarm_thread` 就是闹铃线程，即创建的每个闹铃线程执行该函数，当函数返回时，闹铃线程终止。该函数的参数 (`void * arg`) 就是传给 `pthread_create` 函数的第四个参数，即包含闹钟请求命令的 `alarm_t` 结构指针。线程首先将 `void *` 参数转换为 `alarm_t *` 类型，然后调用 `pthread_detach` 函数来分离自己，以通知 Pthreads 不必关心它的终止时间和退出状态。

9~12 线程睡眠指定的时间（由 `alarm_t` 结构中的 `seconds` 决定），然后打印指定的消息字符串，最后，线程释放 `alarm_t` 结构空间并返回，即线程终止。通常，Pthreads 会保存线程的资源以使其他线程了解它已经终止并获得其最终结果。由于本例中线程负责分离自己，所以不必做上述工作。

■ `alarm_thread.c`

part 2 `alarm_thread`

```

1 void *alarm_thread (void *arg)
2 {
3     alarm_t *alarm = (alarm_t*)arg;
4     int status;
5
6     status = pthread_detach (pthread_self ());
7     if (status != 0)
8         err_abort (status, "Detach thread");
9     sleep (alarm->seconds);
10    printf ("%d %s\n", alarm->seconds, alarm->message);
11    free (alarm);
12    return NULL;
13 }
```

■ `alarm_thread.c`

part 2 `alarm_thread`

`alarm_thread.c` 中的 `main` 函数和其他两个版本一样，它循环读取命令行、解释命令行直到不能从 `stdin` 中读取数据为止。

12~25 `main` 函数为每个命令行从堆中分配内存（作为 `alarm_t` 结构）。`alarm_t` 结构中保存了传给线程的闹铃时间和显示文本的信息。如果 `sscanf` 函数解析命令行失败，则释放已分配的堆内存。

26~29 创建一个闹铃线程，以 `alarm_t` 为线程参数运行函数 `alarm_thread`。

■ alarm_thread.c

part 3 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     while (1) {
9         printf ("Alarm> ");
10        if (fgets (line, sizeof (line), stdin) == NULL) exit (0);
11        if (strlen (line) <= 1) continue;
12        alarm = (alarm_t*)malloc (sizeof (alarm_t));
13        if (alarm == NULL)
14            errno_abort ("Allocate alarm");
15
16        /*
17         * Parse input line into seconds (%d) and a message
18         * (%64[^\\n]), consisting of up to 64 characters
19         * separated from the seconds by whitespace.
20         */
21        if (sscanf (line, "%d %64[^\\n]",
22                    &alarm->seconds, alarm->message) < 2) {
23            fprintf (stderr, "Bad command\\n");
24            free (alarm);
25        } else {
26            status = pthread_create (
27                &thread, NULL, alarm_thread, alarm);
28            if (status != 0)
29                err_abort (status, "Create alarm thread");
30        }
31    }
32 }
```

■ alarm_thread.c

part 3 main

1.5.4 总结

比较闹钟程序的两个异步版本是理解线程编程的很好途径。首先，在 fork 版本中，每个闹铃有一个从主进程拷贝的独立地址空间，这就意味着可以将闹铃时间和显示文本保存在局部变量中，一旦建立了子进程（fork 调用返回），父进程就可以改变这些变量而不会影响闹铃子进程。其次，在多线程版本中，所有线程共享同一个地址空间，所以可以为每个闹铃调用 malloc 建立新的 alarm_t 结构，并将它传递给新建线程。

在使用 fork 版本中，主程序需要调用 waitpid 或者其他 wait 系列函数来告诉系统释放其创建的子进程资源。例如，在 alarm_fork.c 中主程序在每个命令之后循环调用 waitpid，来回收所有结束的子进程。不需要等待线程结束，除非希望获得它的返回值。例如，在 alarm_thread.c 中，每个闹铃线程分离自己（part2 第 6 行），所以该线程的资源在它终止后会立刻回收。

在多线程版本中，基本活动（睡眠和打印消息）必须编码为独立的例程。而在 alarm.c 和 alarm_fork.c 中，这些活动不是由函数调用完成的。在像闹钟这样简单的实例中，将所有代码放在一起通常更易理解，所以好像 alarm_fork.c 更有可取之处。但是在更加复杂的应用中，很少会有如此简单的基本活动，可以置于一个例程中而不造成整体混乱。

在实际的应用中，可能不需要为每个闹铃建立一个进程。你可以轻易地设置上百个闹铃活动，但是系统可能无法创建那么多进程。另一方面，你可以在一个进程中创建几百个线程。由于不需要为每个闹铃请求保存堆栈和环境，所以这的确是个很有生命力的设计。

一个更加成熟的 alarm_thread.c 版本可以只包含两个线程：一个负责读取用户输入，一个等待闹铃的终止。我将在后面的章节给出这个版本，那时我们已经了解了更多的基本概念。当然也可以用两个进程来实现，但是会很麻烦。在线程间传送消息要快得多——因为不需要映射共享内存、不需要通过管道读写、不需要担心进程间传送的地址指针是否一致。线程间共享一切——一个线程内有效的地址指针在所有的线程内同样有效。

1.6 线程的好处

*"O Looking-Glass creatures," quoth Alice, 'draw near!
'Tis an honour to see me, a favour to hear:
'Tis a privilege high to have dinner and tea
Along with the Red Queen, the White Queen, and me!"*

—Lewis Carroll, Through the Looking-Glass

多线程编程模型具有以下优点：

1. 在多处理器系统中开发程序的并行性。除了并行性这一优点是需要特殊硬件支持外，其他优点对硬件不做要求。
2. 在等待慢速外设 I/O 操作结束的同时，程序可以执行其他计算，为程序的并发提供更有效、更自然的开发方式。
3. 一种模块化编程模型，能清晰地表达程序中独立事件间的相互关系。

以下各节将详细描述上述优点。

1.6.1 并行

在多处理器系统中，线程模式可以让一个进程同时执行多个独立运算。一个运行在双 CPU 上的计算密集型多线程程序几乎可以获得传统的单线程程序两倍的性能。“几乎两倍”是基于以下事实：创建额外的线程和执行线程间的同步会带来额外的开销。这种效果通常称为“可扩展性”。双 CPU 系统可以是单 CPU 系统性能的 1.9 倍；三 CPU 系统可以是它的 2.9 倍；四 CPU 系统可以是它的 3.8 倍，依次

类推。可扩展性总是随着 CPU 数量的增加而下降，因为可能会有更多耗时的锁操作和内存冲突。

可扩展性可以由图 1.1 所示的 Amdahl 法则预测，在 Amdahl 法则等式中， p 代表可并行代码与整个执行时间的比率， n 代表代码可以使用的处理器数目。并行工作的整个延续时间就等于非并行工作 ($1-p$) 的延续时间加上每个处理器执行并行工作 (p/n) 的延续时间。

$$\text{Speedup} = \frac{1}{(1-p) + \frac{p}{n}}$$

图 1.1 Amdahl 法则

Amdahl 法则显示了串行限制并行的简单关系。当程序没有可并行代码时 ($p=0$)，加速比等于 1，即不是一个并行程序。如果程序不需要任何同步或串行代码 ($p=1$)，则加速比为 n (处理器的数目)。需要的同步越多，则并行能带来的好处越少。换句话说，完全无关的活动比高度相关的活动更有可扩展性：因为独立活动之间需要更少的同步。

图 1.2 中显示了 Amdahl 法则的效果，时间轴从左向右递增。图 1.2 显示了任一时刻并行工作的处理器数量。只有一条水平线的区域表明程序串行执行。多条水平线的区域表明程序得益于多个处理器。如果你有 4 个处理器，而程序只有 10% 的执行时间并行，则 Amdahl 法则的加速比就是 $1/(0.9+(0.1/4))$ ，约等于 8%。

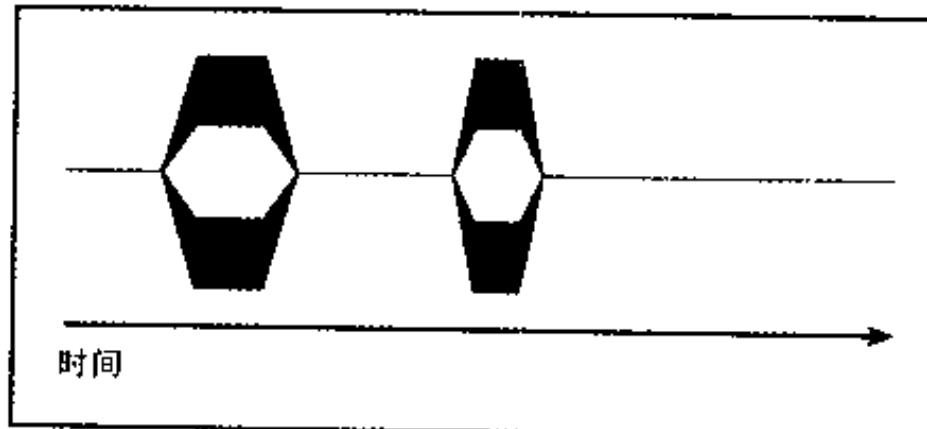


图 1.2 时间-并行图

对于大矩阵上的操作，可以通过将矩阵分块来实现并行操作。例如，让每个线程对矩阵的一组行或者列执行操作，这些操作不需要其他线程对其他分块的操作结果。通常还需要在处理矩阵的开始和结束时刻同步线程，通常使用 *barrier*^①。Amdahl

^① barrier 是一种简单的同步机制，它阻塞每个线程直到到达某个“限额”数，然后所有线程被解除阻塞。例如，可以用 barrier 机制来确保：只有当所有线程都准备好了才执行一段并行代码。在 7.11 节里会更详细地描述 barrier，并演示一个简单的 barrier 包的构造。

法则表明让每个线程做相对独立、无须频繁同步的一大块工作，要比让它们做小块工作能获得更好的并行。

Amdahl 法则是一个帮助你理解可扩展性的极好的思维练习，然而它却不是一个实用的工具，因为几乎不可能为每个程序精确计算 p 值。为精确计算 p 值，不仅需要考虑代码中所有的串行段，而且还要考虑操作系统内核甚至硬件的因素。多处理器系统的硬件必须提供一些同步访问内存的机制。当每个处理器有自己的数据高速缓存时，其中的数据必须与其他处理器缓存的数据以及内存中的数据保持一致。必须考虑所有这些串行以获得精确的 p 值计算。

1.6.2 并发

线程编程模式允许程序在等待像 I/O 之类的阻塞操作时继续其他计算，这对于网络服务器和客户端是有用的，也是客户/服务器系统（如 OSF DCE）使用线程的主要原因。当一个线程等待费时的网络读或写操作时，该线程被阻塞，而程序中的其他线程还可以独立地运行。某些系统支持异步 I/O 操作，可以具有类似的好处；但是大部分 UNIX 类系统不提供异步 I/O 操作^①。而且，异步 I/O 通常比线程更加复杂。

例如，你需要在 I/O 完成时处理异步通知，或者轮询（Poll）等待其完成。如果执行了一个异步 I/O 操作，然后进入 polling 循环，就失去了异步 I/O 操作的优势——应用程序就完全在等待。如果在程序的其他地方轮询，或者处理异步通知，然后执行 I/O 和处理结果数据，将使代码的分析和维护更加困难。当使用同步 I/O 时，可以执行 I/O 操作，然后继续工作。在多线程中使用同步 I/O，能提供与异步 I/O 几乎相同的好处。在大部分情况下，使用线程编写复杂的异步代码比传统的异步编程技术容易得多。

可以像 1.5 节一样写一个异步闹钟程序而不使用进程或线程。使用时钟信号作为闹铃，并把异步的读取时钟信号作为输入。使用时钟信号在很多方面更加复杂，因为你完全受限于一个信号处理程序中能够完成的工作。异步 I/O 不允许使用 stdio 函数库提供的便利。基本程序功能将由一系列的信号处理程序和函数组成，可能会更难让人理解。

不过，异步 I/O 的确比同步线程有一个优势：尽管线程比进程开销少（执行时间和存储空间），异步 I/O 操作的环境要求几乎总是比线程开销更为有效。如果你同时需要执行许多异步的 I/O 操作，则使用异步 I/O 模式可能更为有效。不过请注意，有些“异步”I/O 包的实现仅仅是将 I/O 请求在线程缓冲池内分发而已。

另一种编写异步应用的方法是将每个活动视为事件。事件由隐藏的后台进程排

^① UNIX 系统支持“非阻塞式 I/O”，但这与异步 I/O 不是一回事。非阻塞式 I/O 允许程序推迟执行 I/O 操作，直到它能够不被中断地完成该操作。但异步 I/O 能在程序执行其他操作的同时继续进行。

队，串行分派并被程序处理——通常使用向分派器注册过的回调函数。事件分派器已经广泛应用于窗口界面系统，如 Apple 的 Macintosh、Microsoft 的 Windows 和 UNIX 的 X Windows（如 Motif 和 CDE）。

事件机制大大减少了使用信号和异步 I/O 的复杂度。只要事件由事件分派器直接支持（例如，所有分派器处理来自键盘或鼠标输入事件），通常可以申请在某一时刻自动插入时钟事件。这样，用事件接口编写的闹钟程序只需初始化事件分派器并循环处理事件。输入事件应该转发到解析器，导致一个新的时钟事件请求；时钟事件则应该转发到一个以指定格式打印闹钟消息的处理函数中。

对于十分简单的应用（如本书的闹钟程序），基于事件的实现要比基于进程或者线程的实现更加简单，至少当除去初始化事件分派器的开销时是这样。但要创建更大、更复杂的应用时，事件的限制就比较明显了，因为事件是串行处理的。

事件不是并发的，程序一次只能做一件事。程序收到一个事件，处理它，再等待接收下一事件。如果需要花长时间处理一个事件，如大型数据库的排序，用户界面可能相当长的时间内无法响应用户的操作；如果事件引发长时间等待，如通过低速网络连接读取数据，用户也只能再次等待。

响应速度问题可以通过在事件分派器中处理额外操作来最小化，但是要在合适的位置处理它们而根本不影响操作性能可能很困难。另外，如果是从他人那儿购买的共享数据库，你可能没有机会在事件分派器中排序数据库。

而另一方面，你可以创建一个专门排序数据库的线程，或者从网上读取数据而让用户界面线程立即处理另一用户操作。慢速操作继续执行，而程序还可以响应。即使数据库包不能容忍在多个线程中运行，也可以将排序命令传到串行执行数据库操作的服务器线程中排队等待处理，而始终保持用户界面的响应。

1.6.3 编程模型

令人惊奇的是，即使你的代码从不在多处理器系统上运行，线程编程仍然是个好主意。的确如此，使用线程编程迫使你思考和安排程序中的同步需求。你总是需要考虑程序的依赖性，但线程帮助你从注释中将这种需求移到了程序执行结构中。

用汇编语言编写的程序和用高级语言编写的程序一样使用所有的串行控制结构（循环、条件语句），然而，很难分清一条分支指令是代表循环的头或者尾部，是条件 GOTO 语句，还是其他语句。而在高级语言中，如 C 语言中的 do、while、for、if 和 switch 语句，在源代码中就可以使这些串行编程结构更加一目了然。控制结构清楚意味着在你的大部分程序设计中源代码是清晰的，让其他人更容易理解。

类似地，C 语言程序（甚至汇编程序）可以通过遵循编程惯例使用数据封装和多态，而这些惯例应该仔细地记录成档并及时更新。但是，如果使用面向对象语言编写，则源程序中的封装和多态就很明显了。

在串行程序中，同步需求隐藏在操作顺序中。真正的同步需求只能由源代码中

的注释记录，如“文件必须首先打开，然后才能从中读取数据”。当使用线程编程时，串行假设应该限于小段连续代码，如单一函数内部。为安全起见，更加全局性的假设必须由显式的同步结构加以确保。

在传统的串行编程模型中，调用函数 A 做一件事，然后再调用函数 B 做另一件事，即使两个函数间不需要串行化，如果开发人员在跟踪错误时试图确定程序在做什么，可能无法明显地确定应该跟踪哪个函数调用。另外，严格的串行模型容易使函数 B 依赖于函数 A 的部分结果。这样在后来的修改中，如果将函数调用关系颠倒，程序可能就会无故崩溃。程序的依赖性可以使用源代码注释记录，但是注释常常是模棱两可的，而且在后来的代码变更时可能无法及时更新。

线程编程模型将独立的或者松耦合的功能执行流（线程）显式地分离。如果活动设计为线程，那么每个函数必须包括显式的同步以确保依赖关系。因为同步机制就是可执行的代码，所以依赖性改变时也无法忽略它。同步结构的存在让阅读代码的人了解代码中的时间依赖关系，使代码维护更加容易，尤其是对于包含大量独立代码的大型程序而言。

一个汇编程序员通过理解高级语言编程可以写出更好、更易维护的代码；一个 C 语言程序员通过理解面向对象编程可以写出更好、更易维护的代码。即使你从来不用多线程编程，也可以从理解独立函数线程编程中受益。这是或多或少独立于特定串行代码的“mental 模型”[或者是被太多人使用的词：“范式（paradigm）”]。清楚地分离功能独立的代码可以使串行程序更易理解和维护。

1.7 线程的代价



*All this time the Guard was looking at her, first through a telescope, then through a microscope, and then through an opera-glass.
At last he said, "You're travelling the wrong way,"
and shut up the window, and went away.*

—Lewis Carroll, *Through the Looking-Glass*

当然，事物总有反面，我在前面几节中已经论述了线程强大的优势，即使在单处理器系统中这种优势依然存在，在多处理器系统中它们甚至能够提供更多的好处。

那么你为什么不想使用线程呢？任何事情都有代价，线程也不例外。在很多情形下好处超过代价；在其他情形下则相反。为公平起见，以下几小节讨论线程编程的代价。

1.7.1 计算负荷

线程代码中的负荷代价包括由于线程间同步所导致的直接影响，如时间代价。很多聪明的算法能在某些情况下避免同步，但是没有一个是全能的。在几乎任何线

程代码中你都需要使用某种同步机制。使用太多的同步很容易损失性能。例如，对于两个总是同时使用的变量分别加以保护，这意味着你在同步上花费太多的时间而损失了并发，因为任何使用一个变量的线程一定要使用另一个变量。

线程编程的负荷还可能包含更多细微的影响。例如，一直向同一内存地址写数据的线程，在支持读写排序的处理器上可能需要大量的时间用于同步。其他处理器可能仅在线程使用特殊的指令时，如内存边界界限或者像测试-置位这样的多处理器原子操作，才花费时间来同步。3.4 节中将对于此类影响做进一步的论述。

从程序中排除瓶颈代码，如添加线程来执行多个并发 I/O 操作，可能会妨碍发现其他底层的瓶颈问题——ANSI C 库、操作系统、文件系统、设备驱动程序、内存和 I/O 结构或设备控制器。这些影响通常难以预测或度量，无法清晰记录。

一个很少被外部事件阻塞的计算密集型线程，无法与同类其他线程有效地共享一个处理器。一个 I/O 线程可能隔段时间会中断它一次，但是 I/O 线程会被其他外部事件阻塞，所以计算密集型线程还会再次运行。如果你创建的计算密集型线程比可用的处理器多，则可能比单线程实现获得更好的代码结构，但是程序性能也会更糟。性能的损失是由于多线程结构在你要完成的工作上增加了额外的同步和调度开销，而你可用的资源不变。

1.7.2 编程规则

尽管线程编程模型的基本思想简单，但是编写实际的代码不是件容易的事。编写能够在多个线程中良好工作的代码需要认真的思考和计划。你需要明白同步协议和程序中的不变量（invariant），你不得不避免死锁、竞争和优先级倒置。后面几小节将描述以上问题，教你如何设计代码避开此类问题，以及如何发现和解决这些问题。

你当然应该尽量使用库代码而不是自己编写。这些代码有些是操作系统提供的，通常大部分库函数能安全地用于多线程程序中。POSIX 标准保证由 ANSI C 和 POSIX 提供的大部分函数必须在多线程应用中安全使用。然而，肯定还有一些你可能感兴趣的、将会用到的一些函数没有包括在内。你经常需要调用一些库函数，它们不是由操作系统提供的，如数据库软件，其中一些代码就不是线程安全的。我会讨论一些使用不安全代码的技巧，但是它们并非总能有效，而且可能有些难以控制。

一个进程中的所有线程共享地址空间，线程间没有保护界限。如果一个线程使用未初始化的指针写内存，则可能破坏其他线程的堆栈或堆空间。最终的失败很可能发生在无辜的受害人那儿，或者可能“罪犯”已经去做其他事后很久才发生。如果允许在线程内运行任意代码，则这一点尤其重要。例如，若库函数支持回调函数，则必须确保库函数和回调函数都是线程安全的。

需要重点说明的是，好的串行代码不一定是好的线程代码，而不好的线程代码会更难于定位和修改错误。考虑现实中的并行会有所帮助，但是编程比实际生活需要更多细致的工作。

1.7.3 更难以调试

在第8章，会介绍更多有关调试线程代码的知识。你会看到一些以后可能遇到或用到的工具和调试技巧，那时你会了解所有关于互斥量和内存可视性的内容，并学会解决死锁和竞争问题。现在我们不必关心细节问题，本节只是说明你必须学会线程调试，虽然不像任何人希望的那么容易，不过调试什么时候又容易过？

提供线程功能的系统通常将传统的串行调试工具扩展用以提供基本的线程调试支持。系统会提供一个调试器，允许你看到所有线程的调用结构树，并设置只能在特定线程内激活的断点。系统可能提供某种形式的性能分析器，让你计算某个线程或者所有线程中函数的累计占有处理器时间。

不幸的是，这仅仅是调试异步代码的开始。调试不可避免地要改变事件的时序。这在调试串行代码时不会有大问题，但是在调试异步代码时却是致命的。如果一个线程因调试陷阱而运行得稍微慢了，则你要跟踪的问题就可能不会出现。每个程序员都会遇到此类在调试时无法再现的错误，这在线程编程中会更加普遍。

在串行程序中很难跟踪内存错误，如通过未初始化的指针写内存，这在线程程序中就会更难。这种错误是由于其他线程未通过互斥量访问内存吗？是由于使用了错误的互斥量了吗？是因为其他线程建立了指针而未显示同步吗？还是由于与传统串行程序类似的内存错误呢？

一些系统提供了各种调试辅助工具，但没有一个工具是标准的或广泛可用的。它们可能会检查源代码中明显的锁协议违规，并给出有关共享变量和如何加锁共享变量的定义；可能会记录程序运行过程中的线程间交互过程，以允许程序员分析甚至重现线程间的交互；可能会记录并度量同步竞争和负荷；可能会检测到互斥量集合中复杂的死锁条件。

通过旧式的人工查阅代码的方式，大脑是你最强大的、最通用的线程调试工具。你可能需要花大量时间建立几个断点和测试大量的状态来试图缩小错误发生的区域，然后再认真地阅读那部分代码，找出错误。最好让其他人来帮你查错，因为他们不需要详细了解程序功能，所以很多最难查的错误对他们来讲可能就是很明显的事。

1.8 选择线程还是不用线程

*"My poor client's fate now depends on your votes."
Here the speaker sat down in his place,
And directed the Judge to refer to his notes
And briefly to sum up the case.*

—Lewis Carroll, *The Hunting of the Snark*

线程无须给所有编程问题提供最好的解决方案。线程并非总是容易使用，而且

并非总是可达到更好的性能。

一些问题本身就是非并发的，添加线程只能降低程序的性能并使程序复杂。如果程序中的每一步都需要上一步的结果，则使用线程不会有任何帮助。每个线程不得不等待其他线程的结束。

最适合使用线程的是实现以下功能的应用：

1. 计算密集型应用，为了能在多处理器系统上运行，将这些计算分解到多个线程中实现；
2. I/O 密集型应用，为提高性能，将 I/O 操作重叠。很多线程可以同时等待不同的 I/O 操作。分布式服务器应用就是很好的实例，它们必须响应多个客户的请求，必须为通过慢速的网络连接主动提供 I/O 做好准备。

大部分程序有一些本质上的并发，即使那种在处理命令的同时从输入设备中读取下一个命令的简单并发。多线程程序通常比串行程序更快、响应性能更好，它们比实现同样功能的非线程异步程序更易于开发和维护。

那么，你该使用线程吗？你可能不必为你开发的所有程序使用它，但线程编程确实是所有软件开发人员应该理解的技术。

1.9 POSIX 线程概念

*"You seem very clever at explaining words, Sir," said Alice.
"Would you kindly tell me the meaning of the poem
called 'Jabberwocky'?"
"Let's hear it," said Humpty Dumpty. "I can explain all
the poems that ever were invented—and a good many
that haven't been invented just yet."
—Lewis Carroll, Through the Looking-Glass*

首先，本书重点讲述“POSIX 线程”。从技术角度讲，这意味着线程 API 遵循国际正式标准 POSIX 1003.1c -1995，该标准由 IEEE 于 1995 年 6 月通过。POSIX 1003.1 的一个新版本，即 ISO/IEC 9945-1: 1996 (ANSI/IEEE 标准 1003.1, 1996 版)，也由 IEEE^①通过。该新版本包含了 1003.1b-1993 (实时)、1003.1c-1995 (线程) 和 1003.1i-1995 (对 1003.1b-1993 的改进)。除非你要写标准的实现，或者极为严肃，你可能不需要购买关于 POSIX 标准的书。要编写线程程序，你会发现像本书这样的书会更加有用，因为它们为你提供了有关操作系统的编程文档。

^① Contact the IEEE at 1-800-678-IEEE.9945-1:1996 Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application: Program Interface (API) [C Language], ISBN 1-55937-573-6, order number SH94352.

正如我在前言中所说的那样，我将使用非正式的术语“Pthreads”代表“POSIX 1003.1c-1995”，用稍微正式的术语“POSIX.1b”来表示“POSIX 1003.1b-1993”，用“POSIX.14”表示“POSIX 1003.14 多处理器系统主题”，以及其他类似的缩写。在需要正规的场合，以及用在章节标题及表格标题上时，我会使用全称，如比较 POSIX 1003.1-1990 和 POSIX 1003.1-1996。

1.9.1 结构概览

在 1.2 中提过线程系统的三个基本要素：执行环境、调度和同步。在评估任一个线程系统或者比较任意两个线程系统时，都可以从比较它们对这三方面的支持程度开始。

使用 Pthreads，通过调用 `pthread_create` 来创建执行环境（线程）。创建一个线程同样也调度了该线程的执行，这将通过调用指定的“线程启动”函数开始。Pthreads 允许在创建线程时指定调度参数，或者在线程运行时设定。当线程调用 `pthread_exit` 时退出，或者从线程启动函数中返回时退出，当然以后还会遇到其他一些可能性。

基本的 Pthreads 同步模型使用互斥量来保护共享数据、使用条件变量来通信，还可以使用其他的同步机制，如信号量、管道和消息队列。互斥量允许线程在访问共享数据时锁定它，以避免其他线程的干扰。条件变量允许线程等待共享数据到达某个期望的状态（例如队列非空或者资源可用）。

1.9.2 类型和接口

本节简要列举 Pthreads 数据类型并解释它们的一些规则。每个数据类型代表“对象”的完全描述和在程序中如何创建、使用它们，请参考本书后续相关章节，如表 1.2 所示。

表 1.2 POSIX 线程数据类型

类 型	章 节	描 述
<code>pthread_t</code>	2	线程标识符
<code>pthreae_mutex_t</code>	3.2	互斥量
<code>pthread_cond_t</code>	3.3	条件变量
<code>pthread_key_t</code>	5.4	线程私有权键访问键
<code>pthread_attr_t</code>	5.2.3	线程属性对象
<code>pthread_mutexattr_t</code>	5.2.1	互斥量属性对象
<code>pthread_condattr_t</code>	5.2.2	条件变量属性对象
<code>pthread_once_t</code>	5.1	“一次性初始化”控制变量

| 所有数据类型都是“不透明的”。

| 可移植的代码不能对这些数据类型的实现做任何假设。

表 1.2 中列出的所有 Pthread₁ 类型都是不透明的。这些类型没有公共的定义，程序员也不应该对其实现做任何假设，只能按照标准中描述的方式使用它们。例如，线程标识符 ID 可能是整型，或者是浮点类型，或者是结构体类型，任何以不能兼容上述所有定义的方式使用线程 ID 的代码都是错误的。

1.9.3 错误检查

| Pthreads 引入了一种全新的报错方式，而没有使用 errno 变量。

Pthreads 修订版是 POSIX 中第一个与传统的 UNIX 和 C 语言报错机制相分离的部分。传统上的函数在成功返回时，会返回一个有效值或者指示调用成功的 0 值。在错误发生时，会返回特定的 -1 值，并对全局变量 errno 赋值以指示错误类型。

传统的报错机制有许多问题，包括很难创建在报错的同时返回一个有用的 -1 值的函数。当引入多线程时会有更严重的问题。在传统的 UNIX 系统和原来的 POSIX.1-1990 标准中，errno 是一个外部整型 Cextern int 变量。由于该变量一次只能有一个值，所以只能支持进程中的单一执行流程。

| Pthreads 函数有错时不会设置 errno 变量(而大部分其他 POSIX 函数会这样做)。

Pthreads 中的新函数通过返回值来表示错误状态，而不是用 errno 变量。当成功时，Pthreads 函数返回 0，并包含一个额外的输出参数来指向存有“有用结果”的地址。当发生错误时，函数返回一个包含在<errno.h>头文件中的错误代码。

Pthreads 同样提供了一个线程内的 errno 变量以支持其他使用 errno 的代码。这意味着当线程调用使用 errno 报错的函数时，该变量值不会被其他线程重写或读取。你可以像以前一样继续使用 errno 变量，不过在编写新的代码时，应该考虑按照 Pthreads 惯例报错。设置和读取线程内 errno 变量要比读写内存地址或返回函数值带来更多的开销。

例如，在等待线程和查错时，你可能会使用如以下实例 `thread_error.c` 中所示的代码。用来等待线程结束的 `pthread_join` 函数当遇到无效线程 ID 时会返回错误代码 ESRCH。在大多数实现中，未初始化的 `pthread_t` 线程变量就是无效的线程 ID。运行下列实例将显示错误消息“error 3: no such process”。

存在着一种不太可能发生的情况：未初始化的 `thread` 变量拥有一个有效的 `pthread_t` 值，则该值一定是初始线程的 ID（因为进程中没有其他线程）。此时，如果你的 Pthreads 实现了自死锁检测功能，`pthread_join` 函数返回错误代码 EDEADLK；否则，线程将挂起等待自己退出（即自死锁）。

■ `thread_error.c`

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <errno.h>
```

```

4
5 int main (int argc, char *argv[])
6 {
7     pthread_t thread;
8     int status;
9
10    /*
11     * Attempt to join with an uninitialized thread ID. On most
12     * implementations, this will return an ESRCH error code. If
13     * the local (and uninitialized) pthread_t happens to be a valid
14     * thread ID, it is almost certainly that of the initial thread,
15     * which is running main(). In that case, your Pthreads
16     * implementation may either return EDEADLK (self-deadlock),
17     * or it may hang. If it hangs, quit and try again.
18     */
19    status = pthread_join (thread, NULL);
20    if (status != 0)
21        fprintf (stderr, "error %d: %s\n", status, strerror (status));
22    return status;
23 }

```

■ `thread_error.c`

注意，Pthreads 中没有像 perror 函数一样按给定格式打印错误信息的等价函数。相反，它使用 strerror 函数获得错误代码的描述，然后将其打印到 stderr 文件流中。

为了避免在实例代码的每个函数调用中都增加报错和退出的代码段，我写了两个报错宏——使用 `err_abort` 检测标准的 Pthreads 错误，使用 `errno_abort` 检测传统的 `errno` 错误变量方式。以下的 `errors.h` 头文件包含了上述宏定义。`errors.h` 头文件还包含了大部分实例程序需要包含的几个系统头文件，以减少代码量。

■ `errors.h`

```

1 #ifndef __errors_h
2 #define __errors_h
3
4 #include <unistd.h>
5 #include <errno.h>
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <string.h>
9
10 /*
11  * Define a macro that can be used for diagnostic output from
12  * examples. When compiled -DDEBUG, it results in calling printf
13  * with the specified argument list. When DEBUG is not defined, it
14  * expands to nothing.
15  */
16 #ifdef DEBUG
17 # define DPRINTF(arg) printf arg

```

```
18 #else
19 # define DPRINTF(arg)
20 #endif
21
22 /*
23  * NOTE: the "do { ... } while (0);" bracketing around the macros
24  * allows the err_abort and errno_abort macros to be used as if they
25  * were function calls, even in contexts where a trailing ";" would
26  * generate a null statement. For example,
27  *
28  *     if (status != 0)
29  *         err_abort (status, "message");
30  *     else
31  *         return status;
32  *
33  * will not compile if err_abort is a macro ending with "}", because
34  * C does not expect a ";" to follow the "}". Because C does expect
35  * a ";" following the ")" in the do...while construct, err_abort and
36  * errno_abort can be used as if they were function calls.
37 */
38 #define err_abort(code,text) do { \
39     fprintf (stderr, "%s at \"%s\" : %d: %s\n", \
40             text, __FILE__, __LINE__, strerror (code)); \
41     abort (); \
42 } while (0)
43 #define errno_abort(text) do { \
44     fprintf (stderr, "%s at \"%s\" : %d: %s\n", \
45             text, __FILE__, __LINE__, strerror (errno)); \
46     abort (); \
47 } while (0)
48
49 #endif
```

■ errors.h

Pthreads 报错机制中的一个例外是 `pthread_getspecific` 函数，它仅返回线程私有数据——共享“Key”值。5.4 节将详细讲述线程私有数据，但是目前我们只关心报错。对于很多应用来说，管理线程私有数据的能力是至关重要的，函数必须尽量快地执行，所以 `pthread_getspecific` 函数根本不报错。如果 `pthread_key_t` 值无效或者线程未对它赋值，则 `pthread_getspecific` 函数返回 `NULL`。

2 线程

*If seven maids with seven mops
Swept it for half a year,
Do you suppose," the Walrus said,
"That they could get it clear?"
"I doubt it," said the Carpenter,
And shed a bitter tear.*

—Lewis Carroll, *Through the Looking-Glass*

线程是本书提倡的编程方式的本质基础（这一点大概不会有什么奇怪的）。尽管本章重点是讲述线程的，但是你不能指望仅通过本章，就可以掌握线程的全部。线程是本书的关键部分，但是只有线程你还不能做太多的事。不过，总是需要从某个部分开始学习，所以我们从线程开始。

2.1 节讲述创建和管理线程的编程内容，即如何建立线程，线程在程序中如何表示，以及建立线程后能对它们进行的基本操作。

2.2 节讲述线程的生命周期，从线程建立到线程回收。该节将讲述线程所有能够经历的调度状态。

2.1 建立和使用线程

*A loaf of bread," the Walrus said,
"Is what we chiefly need:
Pepper and vinegar besides
Are very good indeed—
Now, if you're ready, Oysters dear,
We can begin to feed."*

—Lewis Carroll, *Through the Looking-Glass*

```
pthread_t thread;
int pthread_equal (pthread_t t1, pthread_t t2);
int pthread_create (pthread_t *thread,
                    const pthread_attr_t *attr,
                    void *(*start) (void *), void *arg);
pthread_t pthread_self (void);
int sched_yield (void);
int pthread_exit (void *value_ptr);
int pthread_detach (pthread_t thread,);
int pthread_join (pthread_t thread, void **value_ptr);
```

第 1 章介绍了有关线程是什么、线程对计算机硬件的意义等基本概念。本节开始讲述第一章中未涉及的内容，如线程在程序中的表现方式、对程序的意义和能够对线程进行的操作。如果你没有阅读第一章，则现在最好回头阅读第一章（我会在这儿等你）。

程序中使用线程标识符 ID 来表示线程。线程 ID 属于封装的 `pthread_t` 类型。为建立线程，你需要在程序中声明一个 `pthread_t` 类型的变量。如果只需在某个函数中使用线程 ID，或者函数直到线程终止时才返回，则可以将线程 ID 声明为自动存储变量。不过大部分时间内，线程 ID 保存在共享（静态或外部）变量中，或者保存在堆空间的结构体中。

Pthreads 线程通过调用你提供的某些函数开始。这个“线程函数”应该只有一个 `void *` 类型参数，并返回相同的类型值。通过向 `pthread_create` 函数传送线程函数地址和线程函数调用的参数来创建线程。

当创建线程时，`pthread_create` 函数返回一个 `pthread_t` 类型的线程 ID，并保存在 `thread` 参数中。通过这个线程 ID，程序可以引用该线程。线程可以通过调用 `pthread_self` 来获得自身的 ID。除非线程的创建者或者线程本身将线程 ID 保存于某处，否则不可能获得一个线程的 ID。要对线程进行任何操作都必须通过线程 ID。例如，如果需要知道线程何时结束，就需要保存线程 ID。

Pthreads 提供了 `pthread_equal` 函数来比较两个线程 ID，只能比较二者是否相同。比较两个线程 ID 谁大谁小是没有任何意义的，因为线程 ID 之间不存在顺序。如果两个线程 ID 表示同一个线程，则 `pthread_equal` 函数返回非零值；否则返回零值。

| 初始线程（主线程）是特殊的。

当 C 程序运行时，首先运行 `main` 函数。在线程代码中，这个特殊的执行流被称为“初始线程”或“主线程”。你可以在初始线程中做任何你能在普通线程中做的事情。例如，它可以调用 `pthread_self` 来获得自己的 ID，也可以调用 `pthread_exit` 来终止自己。如果初始线程将其 ID 保存在一个其他线程可以访问的地方，则其他线程就可以等待初始线程的终止或者分离（`detach`）初始线程。

初始线程的特殊性在于 Pthreads 在 `main` 函数返回阶段保留了传统 UNIX 进程行为。即，进程结束时不会等待其他线程结束。通常，在线程代码中你并不希望如此，但有时这也会很方便。例如，在本书所列的多数实例中，线程一旦脱离进程范围就毫无意义。因此，一旦进程结束就不再关心线程在做什么。当进程结束时，所有线程、状态和它们的工作结果都会简单地“蒸发”——没有理由要清理什么。

分离一个正在运行的线程不会对线程带来任何影响，仅仅是通知系统当该线程结束时，其所属资源可以被回收。

尽管“线程蒸发”有时有用，但大部分时间进程要比你创建的线程“长命”。为确保终止线程的资源对进程可用，应该在每个线程结束时分离它们。一个没有被分离的线程终止时会保留其虚拟内存，包括它们的堆栈和其他系统资源。分离线程意味着通知系统不再需要此线程，允许系统将分配给它的资源回收。

如果要创建一个从不需要控制的线程，可以使用属性（attribute）来建立线程以使它总是可分离的（属性在 5.2.3 节讲述）。如果不想要创建的某个线程，而且知道不再需要控制它，可以随时调用 `pthread_detach` 函数来分离它。线程可以分离自己，任何获得其 ID 的其他线程也可以随时分离它。如果需要获得线程的返回值，或者需要获知其何时结束，应该调用 `pthread_join` 函数。`pthread_join` 函数将阻塞其调用者直到指定线程终止，然后，可以选择地保存线程的返回值。调用 `pthread_join` 函数自动分离指定的线程。

如前所述，进程内的线程能够同时使用不同的堆栈执行不同的指令。尽管线程彼此独立地执行，它们总是共享同一个地址空间和文件描述符。共享地址空间使得线程间能够高效地通信，这也是线程编程模型的一大优点。

一些程序可能建立执行不相关任务的线程，但大多数情况是一组线程共同工作以实现某个目标。例如，一组线程形成流水线工作方式，每个线程在共享数据流上执行特定的子任务并将结果传给下一线程。一组线程可以形成一个工作群体，以完成同一任务的不同部分。或者，某个控制线程负责在一组工作线程间分配任务。可以将上述模型以各种方式组合，举例来说，某工作线程执行流水线中的某个复杂步骤，如转换矩阵的一列数据。

以下例程 `lifecycle.c` 建立了一个线程。在线程生命周期一节中我们还会引用该例程。

7~10

线程函数 `thread_routine` 返回一个符合标准线程函数原型的值。本例中，线程返回其参数，该值总是为空（NULL）。

18~25

程序调用 `pthread_create` 函数创建线程，然后调用 `pthread_join` 等待线程结束。不一定需要等待线程结束，但如果这样做，就必须通过其他手段来确保进程一直运行到所有线程结束。从 `main` 函数返回将导致进程终止，也将使进程内所有的线程终止。可以在 `main` 函数中调用 `pthread_exit`，这样进程就必须等

到所有线程结束后才能终止。

26~29 当 `join` 完成时，程序检查线程的返回值以确保线程返回值与给定的值（NULL）相同。如果返回值为 NULL，程序退出值为 0（成功）；否则，退出值为 1（失败）。

让所有线程返回某个值是个很好的习惯，即使返回值为空。如果你在线程中忽略了返回语句，`pthread_join` 还是会返回某个值，不论线程在保存返回值的地方（可能是寄存器内）保存了什么数据，都会被当作返回值。

■ lifecycle.c

```
1 #include <pthread.h>
2 #include "errors.h"
3
4 /*
5  * Thread start routine.
6  */
7 void *thread_routine (void *arg)
8 {
9     return arg;
10}
11
12 main (int argc, char *argv[])
13 {
14     pthread_t thread_id;
15     void *thread_result;
16     int status;
17
18     status = pthread_create (
19         &thread_id, NULL, thread_routine, NULL);
20     if (status != 0)
21         err_abort (status, "Create thread");
22
23     status = pthread_join (thread_id, &thread_result);
24     if (status != 0)
25         err_abort (status, "Join thread");
26     if (thread_result == NULL)
27         return 0;
28     else
29         return 1;
30 }
```

■ lifecycle.c

如果“连接”（joining）线程不关心返回值，或者它知道被“被连接”（joined）的线程根本就不返回任何值，则它可以向 `pthread_join` 函数的 `&retval` 参数传递 NULL。被连接线程的返回值将被忽略。

当 `pthread_join` 调用返回时，被连接线程就已经被分离（detached），再也不能连接该线程了。在少数情况下，多个线程需要知道某个特定线程何时结束，则这些线程应该等待某个条件变量而不是调用 `pthread_join` 函数。被等待的线程应该将其返回值（或任何其他信息）保存在某个公共的位置，并将条件变量广播给

所有在其上等待的线程以唤醒它们。

2.2 线程的生命周期

*Come, listen, my men, while I tell you again
The five unmistakable marks
By which you may know, wheresoever you go,
The warranted genuine Snarks.*

—Lewis Carroll, *The Hunting of the Snark*

任意时刻，线程处于如表 2.1 所示的四个基本状态之一。在具体实现中，你可能会发现其他附加的“状态”，这些附加“状态”区别于进入四个基本状态的不同原因。如 Digital UNIX 将这些细微的差别表示为“子状态”，每个状态对应几个“子状态”。不管是称为子状态还是附加状态，“终止态”可以分为“退出”或“取消”两种情形；“阻塞态”可能分为“条件变量阻塞”、“互斥量阻塞”和“读阻塞”等。

表 2.1 线程状态

状 态	含 义
就绪 (ready)	线程能够运行，但在等待可用的处理器。可能刚刚启动，或刚刚从阻塞中恢复，或者被其他线程抢占
运行 (running)	线程正在运行。在多处理器系统中，可能有多个线程处于运行态
阻塞 (Blocked)	线程由于等待处理器外的其他条件无法运行，如条件变量的改变、加锁互斥量或 I/O 操作结束
终止 (Terminated)	线程从起始函数中返回，或调用 <code>pthread_exit</code> ，或者被取消，终止自己并完成所有资源清理工作。不是被分离，也不是被连接，一旦线程被分离或者连接，它就可以被收回

更细微的区别对于调试和分析线程程序很重要。但是，对于理解线程调度不会有本质的区别，所以本书不予讨论。

线程开始于就绪状态，当线程运行时，它调用特定的起始函数。它可能被其他线程抢占，或者因等待外来事件而阻塞自己。最终线程完成工作，或者从起始函数返回，或者调用 `pthread_exit` 函数，即进入终止态。如果线程已被分离，则它立刻被回收重用（这难道不是比“销毁”线程好吗？大部分系统可以重用资源来建立新线程）；否则，线程停留在终止态直到被分离或者被连接。图 2.1 显示了线程状态间的转换关系及激发状态转换的事件。

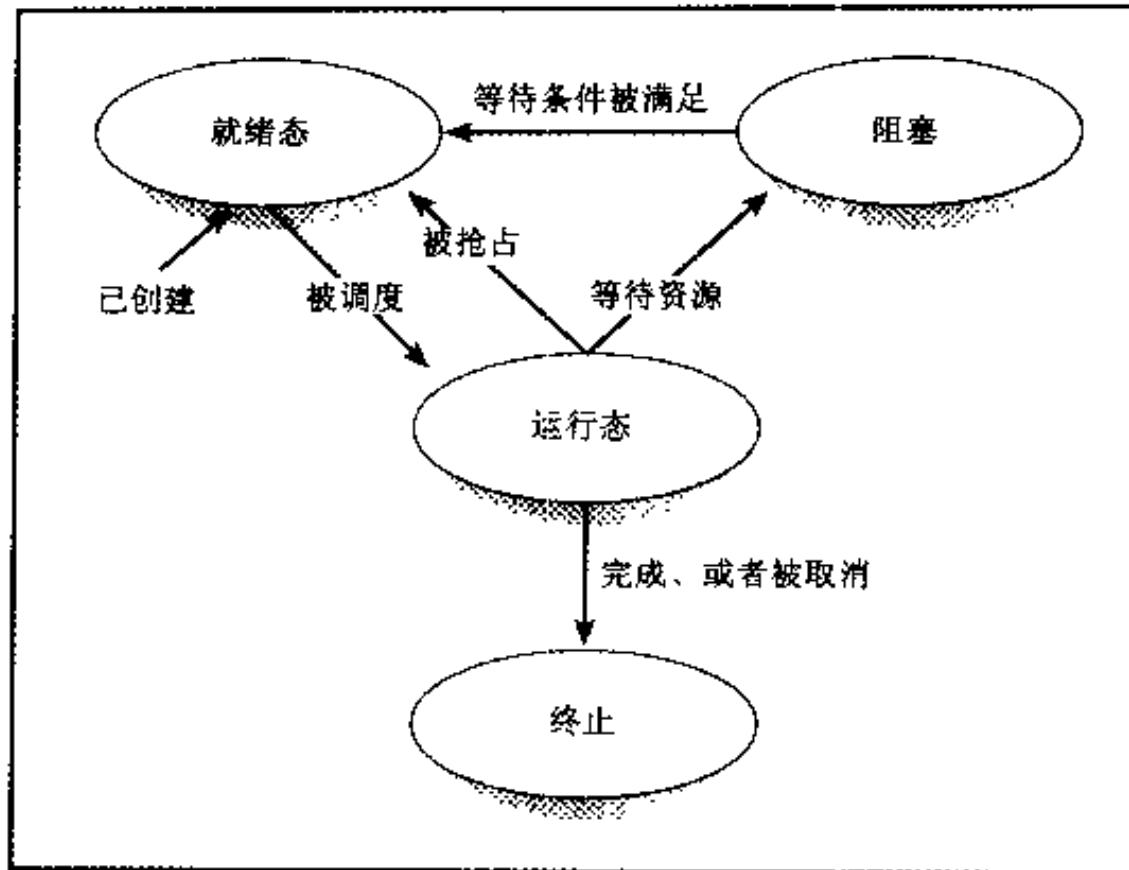


图 2.1 线程状态转换

2.2.1 创建线程

进程的初始线程随着进程的创建而创建。在一个完全支持线程编程的系统中，可能没有代码能够脱离线程运行。线程很可能是系统中包含执行代码所需硬件状态（寄存器、程序计数器、堆栈指针等）的惟一软件环境。

其他线程通过显示的函数调用建立。Pthreads 系统中建立线程的主要方式是调用 `pthread_create`。如果进程的信号通知机制设为 `SIGEV_THREAD`，则当进程收到一个 POSIX 信号时也会创建线程。你的系统还可能包含其他创建线程的方式。

在新线程建立后，它处于就绪态。受调度机制的限制，新线程可能在就绪态停留一段时间后才被执行。5.5 节中包含了线程调度的更多信息。回到例程 `lifecycle.c`，运行 `thread_routine` 函数的线程在 `main` 函数调用 `pthread_create`（第 18 行）时处于就绪态。

有关线程创建最重要的是，在当前线程从函数 `pthread_create` 中返回以及新线程被调度执行之间不存在同步关系。即，新线程可能在当前线程从 `pthread_create` 返回之前就运行了，甚至在当前线程从 `pthread_create` 返回之前，新线程就可能已经运行完毕了。为获得更多有关线程创建时需要注意的信息，请参考 8.1.1 节。

2.2.2 线程启动

一旦线程被创建，最终它将开始执行机器指令。初始指令序列将导致在 `pthread_create` 调用中指定的线程启动函数的执行。线程启动函数的运行参数也是在创建线程时指定。例如，在 `lifecycle.c` 中，新线程使用参数 `NULL` 开始运行 `thread_routine` 函数中的用户代码。

在初始线程中，线程的启动函数（即 `main` 函数）是从程序外部被调用的。例

如，很多 UNIX 系统中将程序链接到一个 crt0.o 文件上，该文件负责初始化进程，然后调用 main 函数。这虽是一个较小的实现区别，但应该记住初始线程与普通线程之间存在的一些不同。如，main 函数的调用参数（argc 和 argv）与普通线程的启动函数不同（void *参数）。另一方面，如果普通线程从启动函数中返回，则线程终止而其他线程依然可以运行；但初始进程从 main 函数中返回时，进程终止（进程内所有线程也被终止）。如果你希望在初始线程终止时，进程中的其他线程继续执行，则需要在初始线程中调用 pthread_exit 而不是从 main 函数中返回。

另一个重要的区别是：在大多数系统中，初始线程运行在默认进程堆栈上，该堆栈可以增长到足够的尺寸；而在某些实现中，普通线程的堆栈空间是受限的，如果线程堆栈溢出，则程序会因段错误或总线错误而失败。

2.2.3 运行和阻塞

像人们一样，线程通常也不能一生保持清醒。大多数线程会不时地睡眠休息。线程之所以会睡眠是因为它需要的某个资源不可用（即被阻塞），或者因为系统将处理器分配给其他线程（即被抢占）。线程大多数时间处于其生命周期中的三个状态：就绪、运行和阻塞。

当线程刚被创建时，或者当线程被解除阻塞再次可以运行时，线程处于就绪态。就绪的线程在等待可用的处理器。同样，当一个运行线程被抢占时，如被时间片机制抢占（因为它已经运行足够长的时间），线程立即进入就绪态。

当处理器选中一个就绪线程执行它时，该线程进入运行态。通常这意味着某个其他线程被阻塞或者被时间片机制抢占，则处理器会保存被阻塞（或抢占）线程的环境并恢复下一个就绪线程的环境。不过在多处理器系统中，一个未用的处理器可以执行一个就绪线程而不必由于其他线程被阻塞。

线程在以下情况时被阻塞：试图加锁一个已经被锁住的互斥量；等待某个条件变量；调用 Singwait 等待尚未发生的信号；执行无法立即完成的 I/O 操作。线程还会由于如内存页错误之类的系统操作而被阻塞。

在线程等到某个事件发生后，重新进入就绪态。如果处理器可用，它可以马上运行。在程序 lifecycle.c 中，主线程在第 23 行阻塞，即调用 pthread_join 等待它创建的线程运行结束。如果新线程此时没有运行，则它将在主线程被阻塞后从就绪态进入运行态。当新线程运行完毕并返回时，主线程才会被解除阻塞，返回就绪态。当处理器可用时，主线程或者立即执行或者等到创建的线程终止后重新运行直到结束。

2.2.4 终止

线程通常从启动函数中返回来终止自己。例如，程序 lifecycle.c 中的线程返回 NULL 后，线程终止。当调用 pthread_exit 退出线程或者调用 pthread_cancel 取消线程时，线程在调用完每个清理过程后也将进入终止态。

这些清理过程是由线程通过调用 `pthread_cleanup_push` 注册的，而且尚未通过调用 `pthread_cleanup_pop` 删除。清理过程将在 5.3.3 节讲述。

线程还会有私有的“线程特定数据”（线程私有数据将在 5.4 节讲述）。如果线程有非空的私有数据值，则与这些数据相关的 `destructor` 函数将被调用。

如果线程已经被分离，则它立刻进入下一节——回收；否则，线程处于终止态，它还可以被其他线程调用 `pthread_join` 连接。这就像 UNIX 中的进程已经结束但还没有被一个 `wait` 调用回收一样。有时这种线程被称为“僵”线程，因为即使它们已经死了但还存在。僵线程可能会保留其运行时的大部分甚至所有资源，因此不应该让线程长时间处于这种状态。当创建不需连接的线程时，应该使用 `detachstate` 属性建立线程使其自动分离（见 5.2.3 节）。

终止后线程至少保留了线程 ID（`pthread_t` 类型数据）和 `void *` 返回值。该返回值从线程启动函数中返回或者在 `pthread_exit` 调用时设定。通过函数返回或者 `pthread_exit` 调用正常终止的线程与通过取消调用终止的线程间的唯一外部区别是：被取消的线程其返回值总是 `PTHREAD_CANCELLED`（这也是不把取消视为一种独立线程状态的原因）。

如果有其他线程在等待连接进入终止态的线程，则该其他线程将被唤醒。它将从其 `pthread_join` 调用中返回相应的值。一旦 `pthread_join` 获得返回值，终止线程就被 `pthread_join` 函数分离，并且可能在 `pthread_join` 函数返回前被回收。这意味着，返回值一定不要是与终止线程堆栈相关的堆栈地址，因为该地址上的值可能在调用线程能够访问之前就被覆盖了。在程序 `lifecycle.c` 中，主线程从第 23 行的 `pthread_join` 调用中返回 `NULL`。

| `pthread_join` 是有用的，但不是规则。

即使当你需要获得返回值时，建立分离的线程并定制自己的返回机制通常与使用 `pthread_join` 一样简单。例如，如果传给一个工作线程某种形式的数据结构，且该结构能够被其他线程访问，你可以让工作线程简单地将结果数据保存在该数据结构中，然后广播某个条件变量。做完这些之后，该线程的环境数据（包括线程 ID）就可以被立即回收，而你关心的返回值始终保存在你随时可以找到的地方。

如果 `pthread_join` 恰恰实现了你想要的功能，则一定要使用它。不过要记住，这不过是最简单也是最受限的传递返回值的模型。如果 `pthread_join` 不能直接实现你想要的功能，则设计你自己的返回机制，而不要扭曲自己的设计来满足 `pthread_join` 的限制。

2.2.5 回收

如果使用 `detachstate` 属性（设为 `PTHREAD_CREATE_DETACH`）建立线程，或者调用 `pthread_detach` 分离线程，则当线程结束时将被立刻回收。

如果终止线程没有被分离，则它将一直处于终止态直到被分离（通过

`pthread_detach`) 或者被连接 (通过 `pthread_join`)。线程一旦被分离，就不能再访问它了。例如，在程序 `lifecycle.c` 中，运行 `thread_routine` 的线程在主线程从 `pthread_join` 调用返回时将被回收。

回收将释放所有在线程终止时未释放的系统和进程资源，包括保存线程返回值的内存空间、堆栈、保存寄存器状态的内存空间等。其中一些资源可能已在线程终止时被释放，但必须记住：在线程终止后上述资源就不该被访问了。例如，如果一个线程将其堆栈空间指针通过共享数据传给另一个线程，则当该线程终止后，此数据就是陈旧数据了。

一旦线程被回收，线程 ID 就无效了，不能再连接它、取消它或者执行其他任何操作。终止线程 ID（可能是系统数据结构地址）可能被分给新的线程。使用该 ID 调用 `pthread_cancel` 可能就会取消一个不同的线程，而不是返回 `ESRCH` 错误。

终止线程将释放所有系统资源，但你必须释放由该线程占有的程序资源。调用 `malloc` 或 `mmap` 分配的内存可以在任何时候、由任何线程释放。互斥量、条件变量和信号灯可以由任何线程销毁，只要它们被解锁并没有线程等待。但是，只有互斥量的主人能够解锁它。如果线程终止时还有加锁的互斥量，则该互斥量就不能被再次使用（因为不会被解锁）。

同步

3

*"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*

使用线程编写各种难度的程序，都需要在线程间共享数据或者需要以一致的顺序在线程间执行一组操作。为了实现该目标，需要同步线程的各种操作。

3.1 节给出了用于讨论线程同步问题的一些基本术语：临界区(critical section)和不变量(invariant)。

3.2 节讨论了 Pthreads 提供的基本同步机制：互斥量。

3.3 节讲述了条件变量，通过它的代码可以通知由互斥量控制的不变量(invariant)的状态变化。

3.4 节给出了线程的一些重要信息和线程对系统内存的影响，以此结束本章关于同步的讨论。

3.1 不变量、临界区和谓词

*"I know what you're thinking about,"
said Tweedledum; "but it isn't so, nohow."
"Contrariewise," 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*

不变量(invariant)是由程序作出的假设，特别是有关变量组间关系的假设。当编写队列包时，你需要某些特殊数据。你需要为每个队列指定一个队列头指针，指向队列第一个元素。每一个数据元素也包含指向下一个元素的指针。重要的并不完全是数据，程序还需要依赖于数据之间的关系。例如，队列头或者为空，或者包含一个指向队首元素的指针。数据元素包含的指针或者指向下一个队列元素，或者

为空（此时该元素为队尾元素）。上述关系就是队列包中的不变量。

即使不变量有时是不明显的，也很难遇到一个完全没有不变量的程序。当程序遇到被破坏的不变量时，系统可能会返回错误结果甚至立即失败。例如，当程序试图解除对指向无效数据元素的队列头指针的引用时。

临界区（有时称为“串行区域”）是指影响共享数据的代码段。由于大部分程序员习惯于思考程序功能而非程序数据，所以你会发现认识临界区比认识数据不变量更容易。不过，临界区总能够对应到一个数据不变量，反之亦然。例如，你从队列中删除数据时，你可以将删除数据的代码视为临界区，也可以将队列状态视为不变量。采取何种考虑方式将决定你最先的想法。

不变量可能会被破坏，而且会经常被独立的代码段破坏。其中的窍门是要确保在“不可预见”的线程访问被破坏的不变量之前将其修复，这也是异步程序中同步机制制作的大部分工作。同步机制使你的程序免于访问被破坏的不变量。如果一定要（临时的）破坏某个不变量，你需要锁住一个互斥量；当其他线程需要访问该不变量时，它也要试图锁住同一个互斥量。此时，线程就会一直等待，直到你将互斥量解锁并恢复不变量的值后才能访问不变量。

同步是自愿的，参与者必须协调工作使系统运行。程序员们应该一致同意不会为抢占舀水桶而打架。桶自己不会神奇地确保一次只有一个程序员舀水。桶更像一个可靠的共享令牌，如果使用恰当，可以使程序员有效地管理他们的资源。

“谓词”（Predicate）是描述代码所需不变量的状态的语句。在英语中，谓词可以是如“队列为空”、“资源可用”之类的陈述。谓词可以是一个布尔变量，也可以是测试指针是否为空的测试结果，还可以是更复杂的表达式（如判断计数器是否大于其下限）。谓词甚至可以是某些函数的返回值，如调用 `select` 或 `poll` 来判断输入文件是否可用。

3.2 互斥量

*"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

大部分多线程程序需要在线程间共享数据。如果两个线程同时访问共享数据就可能会有问题，因为一个线程可能在另一个线程修改共享数据的过程中使用该数据，并认为共享数据保持未变。本节就是讨论如何防止此类问题的。

使线程同步最通用和常用的方法就是确保对相同（或相关）数据的内存访问“互

斥地”进行，即一次只能允许一个线程写数据，其他线程必须等待。Pthreads 使用了一种特殊形式的 Edsger Dijkstra 信号灯[Dijkstra,1968a]——互斥量。互斥量（mutex）是由单词互相（mutual）的首部“mut”和排斥（exclusion）的首部“ex”组合而成的。

经验表明，正确使用互斥量比使用像通用信号灯之类的其他同步模型要容易，还能很容易地使用互斥量与条件变量（见 3.3 节）结合建立任何同步模型。互斥量简单、灵活，并能被有效实现。

程序员的舀水桶类似于一个互斥量（图 3.1）。二者都可以当作传递的令牌，用以保持并发系统的完整性。桶可以被用来保护“舀水”临界区——每个程序员在拿到水桶的时候负责舀水，在不拿水桶的时候不要干扰舀水的人。或者，可以将水桶理解为：用来确保一次只能由一个人舀水这一不变量。

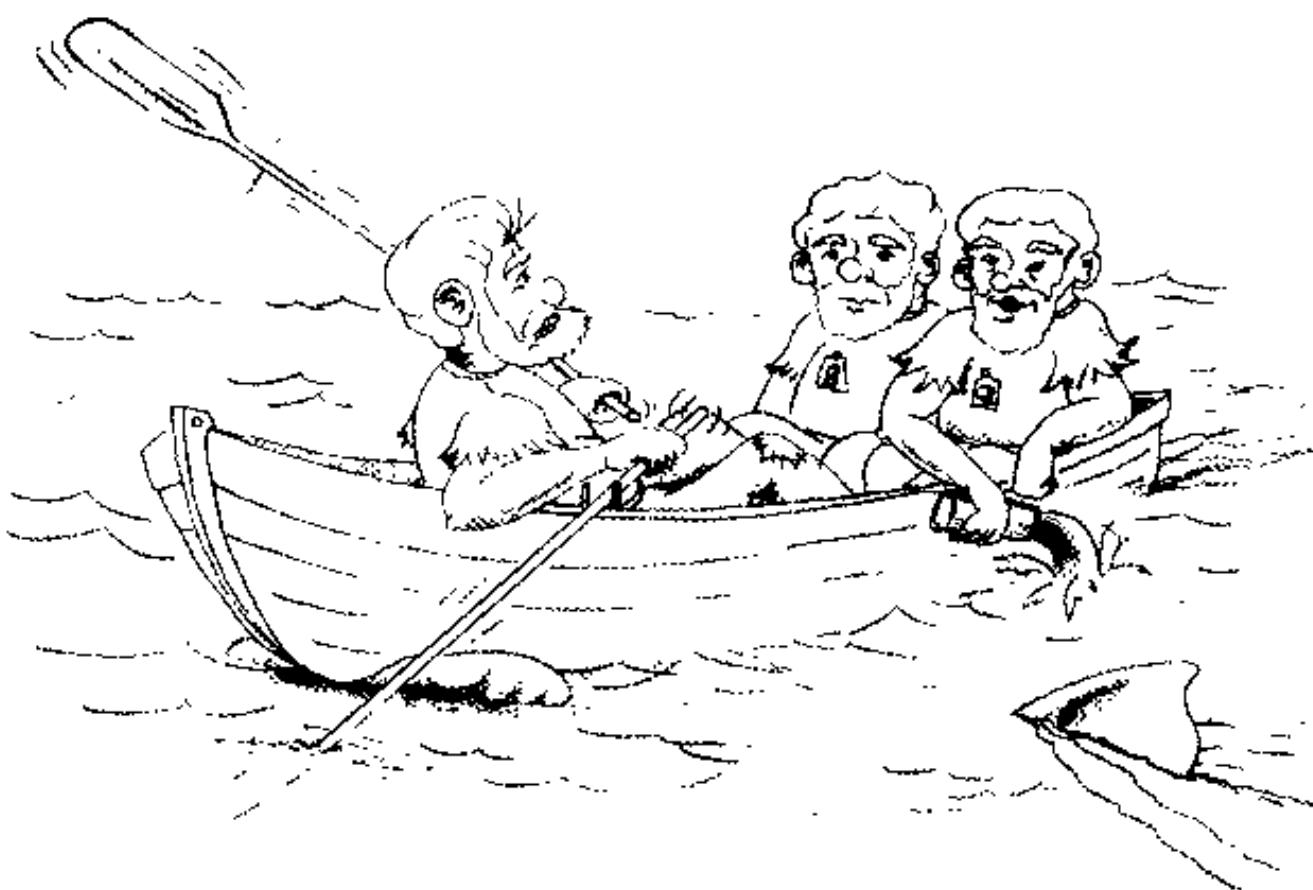


图 3.1 互斥量类比

同步不仅仅在修改数据时重要，当线程需要读取其他线程写入的数据时，而且数据写入的顺序也有影响时，同样需要同步。就像我们会在 3.4 节讲述的一样，很多硬件系统不能保证处理器以相同的顺序访问共享内存，除非有软件的支持。

考虑以下实例：一个线程向数组中某个元素填加新数据，并更新 max_index 变量以表明该数组元素有效。在另一个处理器上同时运行的处理线程，负责遍历数组并处理每个有效元素。如果处理线程在读取数组元素的新数据之前先读取了更新后的 max_index 值，计算就会出错。这可能显得有些不太合理，但是以这种方式工作的内存系统比按照确定顺序访问内存的系统要快得多。互斥量是解决此类问题的通用方法：在访问共享数据的代码段周围加锁互斥量，则一次只能有一个线程进入该代码段。

图 3.2 显示了共享互斥量的三个线程的时序图。处于标有“互斥量”的圆形框

之上的线段表示相关的线程没有拥有互斥量。处于圆形框中心线之下的线段表示相关的线程拥有互斥量。处于中心线之上的线段表明相关的线程等待拥有互斥量。

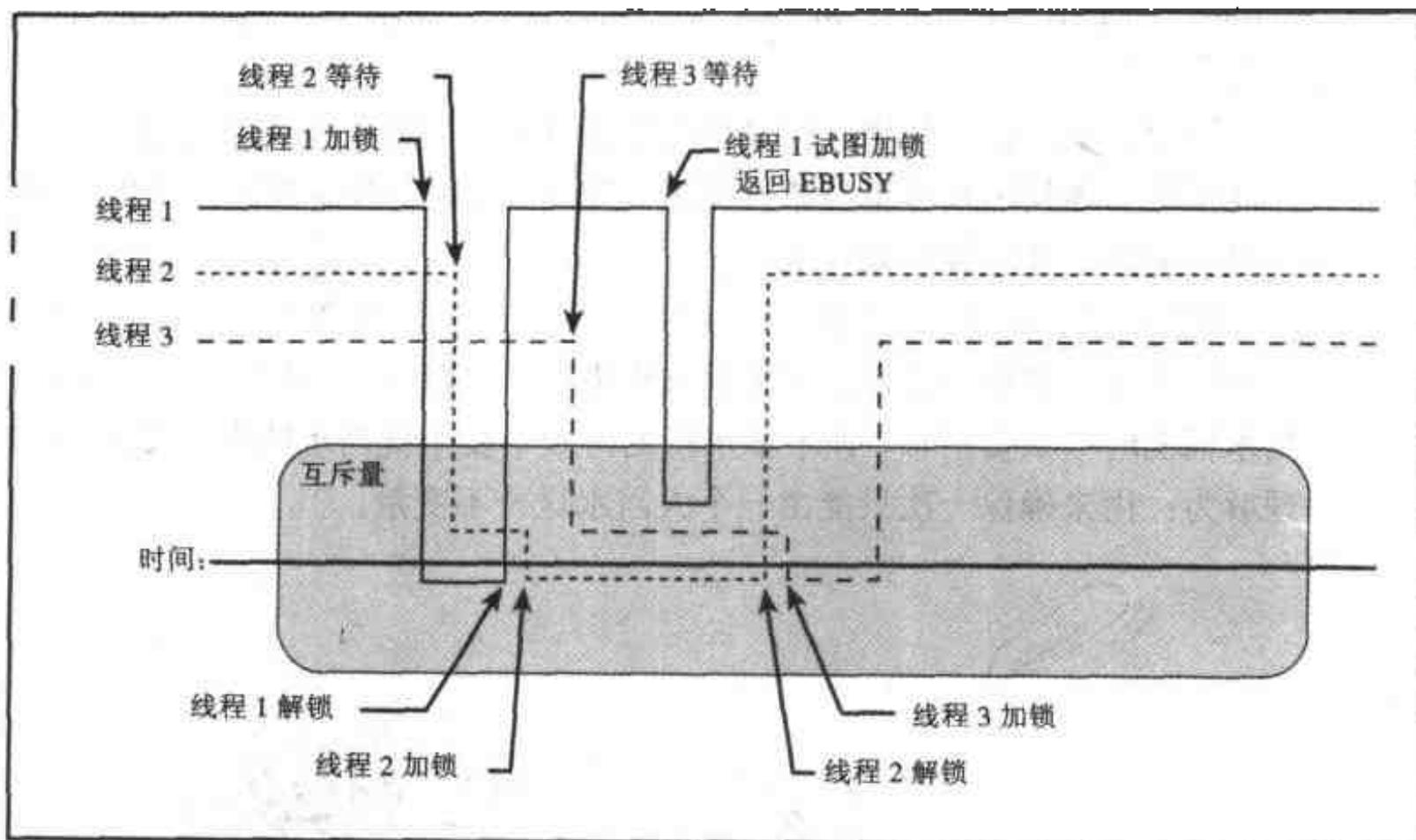


图 3.2 互斥量操作

最初，互斥量没有被加锁。当线程 1 试图加锁该互斥量时，由于没有竞争，线程 1 立即加锁成功，对应线段也移到中心线之下。然后线程 2 试图加锁互斥量，由于互斥量已经被锁住，所以线程 2 被阻塞，对应线段在中心线之上。线程 1 解锁互斥量，解除线程 2 的阻塞，使其对互斥量加锁成功。稍后，线程 3 试图加锁互斥量，被阻塞。线程 1 调用函数 `pthread_mutex_trylock` 试着锁住互斥量，而立刻返回 `EBUSY`。线程 2 解锁互斥量，解除线程 3 的阻塞，线程 3 加锁成功。最后，线程 3 完成工作，解锁互斥量。

3.2.1 创建和销毁互斥量

```

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);
  
```

程序中的互斥量是用 `pthread_mutex_t` 类型的变量来表示的。不能拷贝互斥量变量，因为使用拷贝的互斥量是不确定的。可以拷贝指向互斥量的指针，这样就可以使多个函数或线程共享互斥量来实现同步。

大部分时间你可能在“文件范围”内，即函数体外，声明互斥量为外部或静态存储类型。如果有其他文件使用互斥量，则将其声明为外部类型；如果仅在本文件

内使用，则将其声明为静态类型。你应该使用宏 PTHREAD_MUTEX_INITIALIZER 来声明具有默认属性的静态互斥量，如下面例程 mutex_static.c 所示（你可以编译并运行该程序，不过因为 main 函数为空，所以不会有任何结果）。

■ 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 }
```

通常不能静态地初始化一个互斥量，例如当使用 malloc 动态分配一个包含互斥量的数据结构时。这时，应该使用 pthread_mutex_init 调用来动态地初始化互斥量，如下面程序 mutex_dynamic.c 所示。也可以动态地初始化静态声明的互斥量，但必须保证每个互斥量在使用前被初始化，而且只被初始化一次。可以在创建任何线程之前初始化它，如通过调用 pthread_once（见 5.1 节）。如果需要初始化一个非缺省属性的互斥量，必须使用动态初始化（见 5.2.1 节）。

■ 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

将互斥量与它要保护的数据明显地联系起来是个不错的注意。如果可能的话，将互斥量和数据定义在一起。例如，在 mutex_static.c 和 mutex_dynamic.c 中，互斥量和它要保护的数据就被定义在同一个数据结构中，并通过注释语句记录了这种关系。

当不再需要一个通过 pthread_mutex_init 调用动态初始化的互斥量时，应该调用 pthread_mutex_destroy 来释放它。不需要释放一个使用 PTHREAD_MUTEX_INITIALIZER 宏静态初始化的互斥量。

| 当确信没有线程在互斥量上阻塞时，可以立刻释放它。

当知道没有线程在互斥量上阻塞，且互斥量也没有被锁住时，可以安全地释放它。获知此信息的最好方式是在刚刚解锁互斥量的线程内，程序逻辑确保随后不再有线程会加锁该互斥量。当线程在某个堆栈数据结构中锁住互斥量，以从列表中删除该结构并释放内存时，在释放互斥量占有的空间之前先将互斥量解锁和释放是个安全且不错的主意。

3.2.2 加锁和解锁互斥量

```

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

```

在最简单的情况下，使用互斥量是容易的事情。通过调用 pthread_mutex_lock 或 pthread_mutex_trylock 锁住互斥量，处理共享数据，然后调用 pthread_mutex_unlock 解锁互斥量。为确保线程能够读取一组变量的一致的值，需要在任何读写这些变量的代码段周围锁住互斥量。

当调用线程已经锁住互斥量之后，就不能再加锁该互斥量。试图这样做的结果可能是返回错误 (EDEADLK)，或者可能陷入“自死锁”，使不幸的线程永远等待。

下去。不能解锁一个已经解锁的互斥量，也不能解锁由其他线程锁住的互斥量。锁住的互斥量是属于加锁线程的。如果你需要一个 unowned 锁，可以使用信号灯（见 6.6.6 节）。

下面程序 alarm_mutex.c 是 alarm_thread.c（见第一章）的改进版本。它在一个 alarmserver 线程中处理多个闹铃的请求。

12~17 结构 alarm_t 现在包含了一个标准 UNIX time_t 类型的绝对时间，表示从 UNIX 纪元（1970 年 1 月 1 日 00:00）开始到闹铃时的秒数。这样 alarm_t 结构就可以按照闹铃时间排序，而不是按照请求的秒数排序。另外，还有一个 link 元素将所有请求链接起来。

19~20 互斥量 alarm_mutex 负责协调对闹铃请求列表 alarm_list 的头节点的访问。互斥量是使用默认属性调用宏 PTHREAD_MUTEX_INITIALIZER 静态初始化的。列首指针初始化为空。

■ 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

下面讲述函数 alarm_thread 的代码。该函数作为线程运行，并依次处理列表 alarm_list 中的每个闹铃请求。线程永不停止，当 main 函数返回时，线程“蒸发”。这种做法的惟一后果是任何剩余请求都不会被传送，线程没有保留任何能够在进程之外可见的状态。

如果希望程序在退出之前处理所有未完结的闹铃请求，可以简单地修改程序以达到该目标。当主线程发现列表 alarm_list 为空时，需要通过某种方式通知 alarm_thread 线程终止。例如，可以在主线程中设置一个全局变量 alarm_done

的值，然后调用 `pthread_exit` 而不是调用 `exit` 终止。当 `alarm_thread` 线程发现列表为空且 `alarm_done` 被置位时，它会立即调用 `pthread_exit`，而不是等待下一个请求。

29~30 如果列表中没有新的请求，`alarm_thread` 线程需要阻塞自己一小段时间，解锁互斥量，以便主线程能够添加新的闹铃请求。通过将 `sleep_time` 置为 1 秒来做到这点。

31~42 如果列表中发现请求，则将它从列表中删除。调用 `time` 函数获得当前时间，并将其与请求时间比较。如果闹铃时间已经过期，则 `alarm_thread` 线程将 `sleep_time` 置为 0；否则，`alarm_thread` 线程计算闹铃时间与当前时间的差，并将 `sleep_time` 置为该差值（以秒为单位）。

52~58 在线程睡眠或阻塞之前，总是要解锁互斥量。如果互斥量仍被锁住，则主线程就无法向列表中增加请求。这将使程序变成同步工作方式，因为用户必须等到闹铃之后才能做其他事（用户可能输入一个命令，但是必须等到下一闹钟到期时才能获得系统提示）。调用 `sleep` 将阻塞 `alarm_thread` 线程指定的时间，直到经过该时间后线程才能运行。

调用 `sched_yield` 有一点不同，我们将在后面章节详细讲述 `sched_yield` 函数（见 5.5.2 节）。现在，只要记住调用 `sched_yield` 的效果是将处理器交给另一个等待运行的线程，但是如果没就绪的线程，则立即返回。在程序中，调用 `sched_yield` 意味着：如果有等待处理的用户输入，则主线程运行，处理用户请求；如果用户没有输入请求，则该函数立即返回。

64~67 如果 `alarm` 指针非空，即如果已经从 `alarm_list` 列表中处理了一个闹铃请求，则函数打印消息显示闹铃已到期。然后，线程释放 `alarm` 结构，准备处理下一个闹铃请求。

■ 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 }
```

最后，我们来讨论 `alarm_mutex.c` 的主程序代码。基本结构与我们已经开发过的版本相同，包括一个循环、从 `stdin` 中读取用户输入的请求并依次处理它们。这一次，没有像 `alarm.c` 中那样同步地等待，也没有像 `alarm_fork.c` 或 `alarm_thread.c` 那样为每个请求创建一个异步处理实体（进程或线程），而是将所有请求排队，等待服务线程 `alarm_thread` 处理。一旦主线程将所有请求排队（译者注：实际是将请求添加到列表 `alarm_list` 中），它就可以读取下一个请求了。

- 8~11 建立一个服务线程来处理所有请求。返回的线程 ID 保存在局部变量 `thread` 中（尽管我们不使用它）。
- 13~28 与其他的闹铃版本一样读取并处理用户请求。就像在 `alarm_thread.c` 中那样，数据保存在 `malloc` 分配的堆结构中。
- 30~32 程序将闹铃请求添加到 `alarm_list` 列表中，该列表由主线程和 `alarm_thread` 线程共享。所以在访问共享数据之前，需要将互斥量 `alarm_mutex` 加锁。
- 33 由于线程 `alarm_thread` 串行地处理列表中的请求，所以没有办法知道从读取用户请求到处理请求的时间间隔。因此，`alarm` 结构中包含了闹铃的绝对时间。绝对时间是通过将用户输入的闹铃时间间隔加上由 `time` 调用返回的当前时间获得。
- 39~49 闹铃请求在列表 `alarm_list` 中按照闹铃时间先后顺序排序。插入代码遍历列表，直到找到第一个闹铃时间大于或等于新闹铃请求时间的节点，然后将新请求插入到找到的节点前。因为 `alarm_list` 是个简单的链表，遍历维护了两个指针：一个 `next` 指针指向当前节点，一个 `last` 指针指向前一个节点的 `link` 指针或者指向列表头指针。
- 56~59 如果没有找到大于或等于当前闹铃时间的节点，则将新请求节点插入列表尾部。即，当退出遍历时，如果当前节点指针为 `NULL`，则前一节点（或链表头）指向新请求节点。

■ 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     }

```

■ alarm_mutex.c

part 3 main

这个简单的例子存在几个严重的缺点。尽管与 `alarm_fork.c` 和 `alarm_thread.c` 相比，该实例具有占用更少资源的优势，但它的响应性能不够。一旦 `alarm_thread` 线程从列表中接收了一个闹铃请求，它就进入睡眠直到闹铃到期。当它发现列表中没有闹铃请求时，也会睡眠 1 秒，以允许主线程接收新的用户请求。当 `alarm_thread` 线程睡眠时，它就不能注意到由主线程添加到请求列表中的任何闹铃请求，直到它从睡眠中返回。

这个问题可以通过不同的方式解决。当然最简单的方式就是像 `alarm_thread.c` 那样为每个闹铃请求建立一个线程。当然这也不坏，因为线程还是比较廉价的。不过还是没有 `alarm_t` 数据结构廉价，并且我们更喜欢构造高效的程序，而不仅仅是响应性好的程序。最好的办法是使用条件变量来通知共享数据的状态变化，所以在 3.3.4 节中将给出闹铃程序的最终版本 `alarm_cond.c`。

3.2.2.1 非阻塞式互斥量锁

当调用 `pthread_mutex_lock` 加锁互斥量时，如果此时互斥量已经被锁住，则调用线程将被阻塞。通常这是你希望的结果，但有时你可能希望如果互斥量已被锁住，则执行另外的代码路线，你的程序可能做其他一些有益的工作而不仅仅是等待。为此，Pthreads 提供了 `pthread_mutex_trylock` 函数，当调用互斥量已被锁住时调用该函数将返回错误代码 `EBUSY`。

使用非阻塞式互斥量加锁函数时，需要确保只有当 `pthread_mutex_trylock` 函数调用成功时，才能解锁互斥量。只有拥有互斥量的线程才能解锁它。一个错误的 `pthread_mutex_trylock` 函数调用可能返回错误代码，或者可能解锁其他线程依赖的互斥量，这将导致程序产生难以调试的错误。

下列实例程序 `trylock.c` 使用 `pthread_mutex_trylock` 函数来间歇性地报告计数器的值，不过仅当它对计数器的访问与计数线程没有发生冲突时才报告。

第 4 行的定义用来控制 `counter_thread` 线程在更新计数器时保持互斥量的时间长短。增大该数值将使 `monitor_thread` 线程中 `pthread_mutex_trylock` 函数返回 `EBUSY` 值的概率增大。

14~39 计数线程大约每隔 1 秒醒来一次，锁住互斥量，增加计数器的值。计数器 `counter` 每秒增加 `SPIN`。

46~72 监控线程每隔 3 秒醒来一次，并试图加锁互斥量。如果调用返回 `EBUSY`，则监控线程计算失败的次数，然后再次等待 3 秒钟。如果 `pthread_mutex_trylock` 函数返回成功，则监控线程打印计数器 `counter` 的当前值。

80~88 如果在 Solaris 2.5 系统中，则调用 `thr_setconcurrency` 设置线程并发级别为 2 级。这将允许计数线程和监控线程在单处理器系统中并发地执行；否则，监控线程要等待计数线程终止后才能运行。

■ 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 locked, 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      * Loop until end_time, checking the counter every 3 seconds.
53      */
54     while (time (NULL) < end_time)
55     {
56         sleep (3);
57         status = pthread_mutex_trylock (&mutex);
58         if (status != EBUSY)
59         {
60             if (status != 0)
61                 err_abort (status, "Trylock mutex");
62             printf ("Counter is %ld\n", counter/SPIN);
63             status = pthread_mutex_unlock (&mutex);
64             if (status != 0)
65                 err_abort (status, "Unlock mutex");
66         } else
67             misses++;           /* Count "misses" on the lock */
68     }
69     printf ("Monitor thread missed update %d times.\n", misses);
70     return NULL;
71 }
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 使用互斥量实现原子操作

如 3.1 节所示，不变量是程序中必须为真的陈述。但是你会发现，不变量不可能总是为真，而且很多时候不能为真。为了让不变量总是为真，对其组成数据的修改必须是原子性的。然而，很少有可能使程序状态的多个修改是原子性的。如果没有对硬件、结构的丰富知识和对指令执行的控制，甚至无法确保对程序状态的单个修改原子化。

| “原子”是指不可分割，但是在此处意味着线程不会看到令它困惑的东西。

尽管某些硬件允许将设置数组元素和更新数组索引放入不可中断的单一指令中运行，但大部分硬件不提供这样的支持。即使硬件能够支持，编译器也不允许你控制代码到如此细节的程度。如果不是特别重要，谁又会使用汇编语言呢？更重要的是，大部分我们感兴趣的不变量都要比这复杂。

我们所说的“原子性”只是意味着：即使当多个线程同时运行在多个处理器上时，其他线程也不会发现被破坏（中间状态或者不一致状态）的不变量。当硬件无法支持不可分割或不可中断的操作时，有两种基本方法实现上述目的：一种是当检测到被破坏的不变量时重新再试；一种是重建原始状态。除非你对处理器结构有深入了解或者不想编写可移植代码，否则很难可靠地实现以上目的。

当没有办法确保真正的原子性时，你需要建立自己的同步机制。原子性是理想的，但同步在大部分情况下也同样可以做到。所以当需要“原子化”地更新数组元素和索引时，只要保证互斥量加锁时执行操作即可。

不管硬件是否无分割、不可中断地执行存储和增加操作，在操作的过程中不会有其他协作线程能够打扰你。这个事务从实际目标出发就是“原子性”的。当然，“协同”是关键。任何对不变量感兴趣的线程在修改或检测不变量状态时都必须使用同一个互斥量。

3.2.4 调整互斥量满足工作

互斥量有多大？我可不是指一个 `pthread_mutex_t` 类型的结构占了多少内存。我是使用了一种通俗的、不完全准确的、但是可以被大多数人接受的说法。这种有趣的用法是在有关如何将非线程代码改为线程安全代码的过程中流行起来的。实现线程安全的函数库的一个相对简单的做法是创建一个互斥量，在每次进入函数

库时锁住它，在退出库的时候解锁它。这样函数库就变成了一个串行区域，从而防止了线程间的任何冲突。我们就把保护如此大的串行区域的互斥量称为“大”互斥量，并形而上学地认为比那些只保护几行代码的互斥量要明显地大。

进一步扩展，保护两个变量的互斥量比保护一个变量的互斥量要“大”。那么到底互斥量该多大呢？答案只能是：足够大，但不要太大。

当需要保护两个共享变量时，你有两种基本策略：可以为每个变量指派一个“小”的互斥量，或者为两个变量指派一个“大”的互斥量。哪一种方法更好取决于很多因素。并且，在开发过程中影响因素可能发生改变，这依赖于有多少线程需要共享数据和如何使用共享变量。

以下是主要的设计因素：

1. 互斥量不是免费的，需要时间来加锁和解锁。锁住较少互斥量的程序通常运行得更快。所以，互斥量应该尽量少，够用即可，每个互斥量保护的区域应则尽量大。
2. 互斥量的本质是串行执行。如果很多线程需要频繁地加锁同一个互斥量，则线程的大部分时间就会在等待，这对性能是有害的。如果互斥量保护的数据（或代码）包含彼此无关的片段，则可以将大的互斥量分解为几个小的互斥量来提高性能。这样，任意时刻需要小互斥量的线程减少，线程等待时间就会减少。所以，互斥量应该足够多（到有意义的地步），每个互斥量保护的区域则应尽量的少。
3. 上述两方面看似互相矛盾，但是这不是我们头一次遇到的情况。一旦当你理解了互斥量的性能后，就能够正确地处理它们。

在复杂的程序中，通常需要一些经验来获得正确的平衡。在大多数情况下，如果你开始使用较大的互斥量，然后当经验或性能数据告诉你哪个地方存在频繁的竞争时，你应改用较小的互斥量，则你的代码通常会更为简单。简单是好的。除非发现问题，否则不要轻易花费太多时间优化你的代码。

另一方面，如果从一开始就能明晰你的算法必然导致频繁的竞争，则不要过于简单化。开始使用必须的互斥量和数据结构比后来增加它们容易得多。当然你也可能估计错误，因为直觉不总是对的，尤其在你刚开始介入线程编程时。不过正如人们说的那样，智慧来自于经验，经验来自于缺少智慧。

3.2.5 使用多个互斥量

有时，一个互斥量是不够的，特别是当你的代码需要跨越软件体系内部的界限时。例如，当多个线程同时访问一个队列结构时，你需要两个互斥量，一个用来保护队列头，一个用来保护队列元素内的数据。当为多线程建立一个树型结构时，你可能需要为每个节点设置一个互斥量。

同时使用多个互斥量会导致复杂度的增加。最坏的情况就是死锁的发生，即两

个线程分别锁住一个互斥量而等待对方的互斥量。更多的是像优先级倒置这样的微妙问题（当你将互斥量和优先级调度组合使用时）。有关死锁、优先级倒置和其他的同步问题，请参考 8.1 节。

3.2.5.1 加锁层次

如果可以在独立的数据上使用两个分离的互斥量，那么就应该这样做。这样，通过减少线程必须等待其他线程完成数据操作（甚至是该线程不需要的数据）的时间，你的程序会最终取得成功。如果数据独立，则某个特定函数就不太可能经常需要同时加锁两个互斥量。

当数据不是完全独立的时候，情况就复杂了。如果你的程序中有一个不变量，影响着由两个互斥量保护的数据，即使该不变量很少被改变或引用，你迟早需要编写同时锁住两个互斥量的代码，来确保不变量的完整。如表 3.1 所示，如果一个线程锁住互斥量 A 后，加锁互斥量 B；同时另一个线程锁住互斥量 B 而等待互斥量 A，则你的代码就产生了一个经典的死锁现象。

表 3.1 互斥量死锁

第一个线程	第二个线程
<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>

表 3.1 中的两个线程可能同时完成第一步。即使在单处理器系统中，一个线程完成了第一步后可能被时间片机制抢占，以使另一个线程完成第一步。至此，两个线程都无法完成第二步，因为它们彼此等待的互斥量已经被对方锁住。

针对上述类型的死锁，考虑以下两种通用的解决方法：

- **固定加锁层次：**所有需要同时加锁互斥量 A 和互斥量 B 的代码，必须首先加锁互斥量 A，然后加锁互斥量 B（译者注：即固定加锁顺序）。
- **试加锁和回退：**在锁住某个集合中的第一个互斥量后，使用 `pthread_mutex_trylock` 来加锁集合中的其他互斥量，如果失败则将集合中所有已加锁互斥量释放，并重新加锁。

有许多种方式定义固定加锁层次，但对于特定的互斥量，总有某个明显的加锁顺序。例如，如果有两个互斥量，一个保护队列头，一个保护队列元素内的数据，则很显然的一种固定加锁层次就是先将队列头互斥量加锁，然后再加锁另一个互斥量。

如果互斥量间不存在明显的逻辑层次，则可以建立任意的固定加锁层次。例如，你可以建立这样一个加锁互斥量集合的函数：将集合中的互斥量按照 ID 地址顺序排列，并以此顺序加锁互斥量。或者给每个互斥量指派名字，然后按照字母顺序加锁；或者给每个互斥量指派序列号，然后按照数字顺序加锁。

从某种程度上讲，只要总是保持相同的顺序，顺序本身就并不真正重要。另一方面，你也很少需要一次锁住一组互斥量。函数 A 需要锁住互斥量 1，然后调用函数 B，而 B 需要锁住互斥量 2。如果代码设计为函数加锁层次，则可能发现互斥量 1 和互斥量 2 按照合适的顺序被加锁，即先锁住互斥量 1，再锁住互斥量 2。如果采用任意的加锁层次，特别是代码无法直接控制的层次，则你可能会发现互斥量 2 应该在互斥量 1 之前被锁住。

如果代码不变量允许首先释放互斥量 1，然后再加锁互斥量 2，就可以避免同时拥有两个互斥量的需要。但是，如果存在被破坏的不变量需要锁住不变量 1，则互斥量 1 就不能被释放，直到不变量被恢复为止。在这种情况下，你应该考虑使用回退（或者试锁-回退）算法。

“回退”意味着你以正常的方式锁住集合中的第一个互斥量，而调用 `pthread_mutex_trylock` 函数有条件地加锁集合中其他互斥量。如果 `pthread_mutex_trylock` 返回 `EBUSY`，则你必须释放已经拥有的所有属于该集合的互斥量并重新开始。

回退方式没有固定加锁层次有效，它会浪费时间来试锁和回退。另一方面，你也不必定义和遵循严格的固定加锁层次，这使得回退方法更为灵活。可以组合使用两种算法来最小化回退的代价，即在定义良好的代码区遵循固定加锁层次，在更灵活的地方使用回退算法。

以下程序 `backoff.c` 演示了如何使用回退算法避免互斥量死锁。程序建立了两个线程，一个运行函数 `lock_forward`，一个运行函数 `lock_backward`。每个线程重复循环 `ITERATION` 次，每次循环两个线程都试图依次锁住三个互斥量。`lock_forward` 线程先锁住互斥量 1，再锁住互斥量 2，再锁住互斥量 3；`lock_backward` 线程则按照相反顺序加锁三个互斥量。如果没有特殊的防范机制，则上述程序很快进入死锁状态（在长时间片的单处理器系统中除外，因为时间片足够长到让一个线程在另一个线程运行之前就已经结束）。

如果运行程序 `backoff 0`，就会看到死锁现象。第一个参数用来设置 `backoff` 变量的值。如果 `backoff` 变量为 0，则两个线程都将调用 `pthread_mutex_lock` 来加锁每个互斥量。由于两个线程从不同端开始，所以当它们在中间遇到时就会发生死锁，程序挂起。当 `backoff` 变量非零时（默认值），线程使用 `pthread_mutex_trylock` 调用加锁互斥量，使用回退算法。当加锁互斥量返回失败信息 `EBUSY` 时，线程释放所有现有的互斥量并重新开始。

在某些系统中，可能不会看到任何互斥量冲突，因为一个线程总是能够在另一个线程有机会加锁互斥量之前锁住所有互斥量。可以设置 `yield_flag` 变量来解决这个问题。只要使用第二个参数运行程序即可，例如，执行 `backoff 1 1`。当 `yield_flag` 变量为 0 时（默认值），每个线程的互斥量加锁循环都不可中断，从而防止死锁的发生（至少在单处理器系统中）。当 `yield_flag` 变量大于 0 时，线程会

在锁住每个互斥量之后调用 `sched_yield`, 以确保其他线程有机会运行。如果 `yield_flag` 变量小于 0, 则线程在锁住每个互斥量之后睡眠 1 秒, 以真正确保其他线程有机会运行。

70~75 在锁住所有三个互斥量后, 每个线程报告成功信息和执行回退的总次数。在多处理器系统中, 或者当将 `yield_flag` 变量设为非零值时, 通常会看到更多的非零回退计数。线程按照加锁的相反方向释放所有锁, 以避免其他线程中的不必要的回退操作。在每个循环结束时调用 `sched_yield` 使事情变得复杂。有关 `sched_yield` 的描述见 5.5.2 节。

■ 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 >= 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 * This is a thread start routine that locks all mutexes in
81 * reverse order, to ensure a conflict with lock_forward, which
82 * does the opposite.
83 */
84 void *lock_backward (void *arg)
85 {
86     int i, iterate, backoffs;
87     int status;
88
89     for (iterate = 0; iterate < ITERATIONS; iterate++) {
90         backoffs = 0;
91         for (i = 2; i >= 0; i--) {
92             if (i == 2) {
93                 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 < 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

不管你选择哪种加锁策略，都应该完整地、仔细地、经常地记录下来。在任何使用互斥量的函数中记录下来，在定义互斥量的地方记录下来，在声明它们的头文件中做好记录，在工程设计文档中记录下来。把它写在黑板上，并在你的手指上系一根细绳以确保你不会忘记。

你可以按随意顺序解锁互斥量。解锁互斥量不会导致死锁的发生。在下一节，我们会介绍一类互斥量“交叠层次”，称为“链锁”。在锁链中，正常的工作模式是锁住一个互斥量，然后锁住下一个互斥量，解锁第一个互斥量，依次类推。不过，如果你使用“试锁和回退”算法，你应该总是以相反的顺序解锁互斥量。即，如果

你锁住互斥量 1、互斥量 2、互斥量 3，你应该解锁互斥量 3、互斥量 2、互斥量 1。如果你先解锁互斥量 1、互斥量 2，而互斥量 3 依然被锁住，则其他线程可能在加锁互斥量 3 之前首先锁住互斥量 1、互斥量 2，但是由于无法锁住整个层次，所以不得不释放互斥量 1 和 2，并再次开始加锁过程。按照相反的顺序解锁有利于减少线程做回退操作的可能性。

3.2.5.2 链锁

“链锁”是层次锁的一个特殊实例，即两个锁的作用范围互相交叠。当锁住第一个互斥量后，代码进入一个区域，该区域需要另一个互斥量。当锁住另一个互斥量后，第一个互斥量就不再需要，可以释放它了。这种技巧在遍历如树型结构或者链表结构时十分有用。每一个节点设置一个互斥量，而不是用一个互斥量锁住整个数据结构，阻止任何并行访问。遍历代码可以首先锁住队列头或者树根节点，找到期望的节点，锁住它，然后释放根节点或队列头互斥量。

由于“链锁”是层次锁的一个特殊实例，所以如果你小心使用，两者是兼容的。例如，当平衡或修剪树型结构时，可以使用层次锁；而当搜索某个节点时，可以使用链锁。

不过要小心使用链锁。很容易写出这样浪费处理器时间的代码：代码大部分时间在加锁和解锁时从来不会遇到竞争的互斥量。仅当多个线程几乎总是活跃在层次中的不同部分时才应该使用链锁。

3.3 条件变量

"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*

条件变量是用来通知共享数据状态信息的。可以使用条件变量来通知队列已空、或队列非空、或任何其他需要由线程处理的共享数据状态。我们的程序员航海家用一种类似条件变量的机制通信，如图 3.3 所示。当划船的人轻推睡觉的程序员应该醒来划船时，原先划船的人实际上是发出了条件通知信号。当一个疲惫的划船人进入沉睡时，他相信其他程序员会在合适的时机唤醒他，他实际上就是在等待一个条件的发生。当舀水的人惊恐地发现他一个人已经无法将渗进的水排出去时，他会大声喊叫求救，他实际上是在广播一个条件。

当一个线程互斥地访问共享状态时，它可能发现在其他线程改变状态之前它什么也做不了。状态可能是对的和一致的，即没有破坏不变量，但是线程就是对当前状态不感兴趣。例如，一个处理队列的线程发现队列为空时，它只能等待，直到有

一个节点被填加进队列中。



图 3.3 条件变量类比

例如，共享数据由一个互斥量保护。线程必须锁住互斥量来判定队列的当前状态，如判断队列是否为空。线程在等待之前必须释放锁（否则其他线程就不可能插入数据），然后等待队列状态的变化。例如，线程可以通过某种方式阻塞自己，以便插入线程能够找到它的 ID 并唤醒它。但是这里有一个问题，即线程是运行于解锁和阻塞之间。

如果线程仍在运行，而其他线程此时向队列中插入了一个新的元素，则其他线程无法确定是否有线程在等待新元素。等待线程已经查找了队列并发现队列为空，解锁互斥量，然后阻塞自己，所以无法知道队列不再为空。更糟糕的是，它可能没有说明它在等待队列非空，所以其他线程无法找到它的线程 ID，它只有永远等下去了。解锁和等待操作必须是原子性的，以防止其他线程在该线程解锁之后、阻塞之前锁住互斥量，这样其他线程才能够唤醒它。

| 等待条件变量总是返回锁住的互斥量。

这就是为什么要使用条件变量的原因。条件变量是与互斥量相关、也与互斥量保护的共享数据相关的信号机制。在一个条件变量上等待会导致以下原子操作：释放相关互斥量，等待其他线程发给该条件变量的信号（唤醒一个等待者）或广播该条件变量（唤醒所有等待者）。当等待条件变量时，互斥量必须始终锁住；当线程从条件变量等待中醒来时，它重新继续锁住互斥量。

与条件变量相关的共享数据就是我们在 3.1 节所说的“谓词”，如队列满或队列空条件。条件变量是程序用来等待某个谓词为真的机制，也是用来通知其他线程谓词为真的机制。换句话说，条件变量就是允许使用队列的线程之间交换队列状态信息的机制。

| 条件变量的作用是发信号，而不是互斥。

条件变量不提供互斥。需要一个互斥量来同步对共享数据（包括等待的谓词）的访问，这就是为什么在等待条件变量时必须指定一个互斥量。通过将解锁操作与等待条件变量原子化，Pthreads 系统确保了在释放互斥量和等待条件变量之间没有线程可以改变谓词。

为什么不将互斥量作为条件变量的一部分来创建呢？首先，互斥量不仅与条件变量一起使用，而且还要单独使用；其次，通常一个互斥量可以与多个条件变量相关。例如，队列可以为空，也可以为满。虽然可以设置两个条件变量让线程等待不同的条件，但只能有一个互斥量来协调对队列头的访问。

一个条件变量应该与一个谓词相关。如果试图将一个条件变量与多个谓词相关，或者将多个条件变量与一个谓词相关，就有陷入死锁或者竞争问题的危险。只要小心使用，可能不会有什麼问题，但是很容易搞混你的程序（计算机并不是十分聪明），并且通常也不值得冒险。随后我会进一步讲述有关细节，但原则是：第一，当你在多个谓词之间共享一个条件变量时，必须总是使用广播，而不是发信号；第二，信号要比广播有效。

条件变量和谓词都是程序中的共享数据。它们被多个线程使用，可能是同时使用。由于你认为条件变量和谓词总是一起被锁定的，所以容易让人记住它们总是被相同的互斥量控制。在没有锁住互斥量前就发信号或广播条件变量是可能的（合法的，通常也是合理的），但是更安全的方式是先锁住互斥量。

图 3.4 显示了三个线程与一个条件变量交互的时间图。圆形框代表条件变量，三条线段代表三个线程的活动。

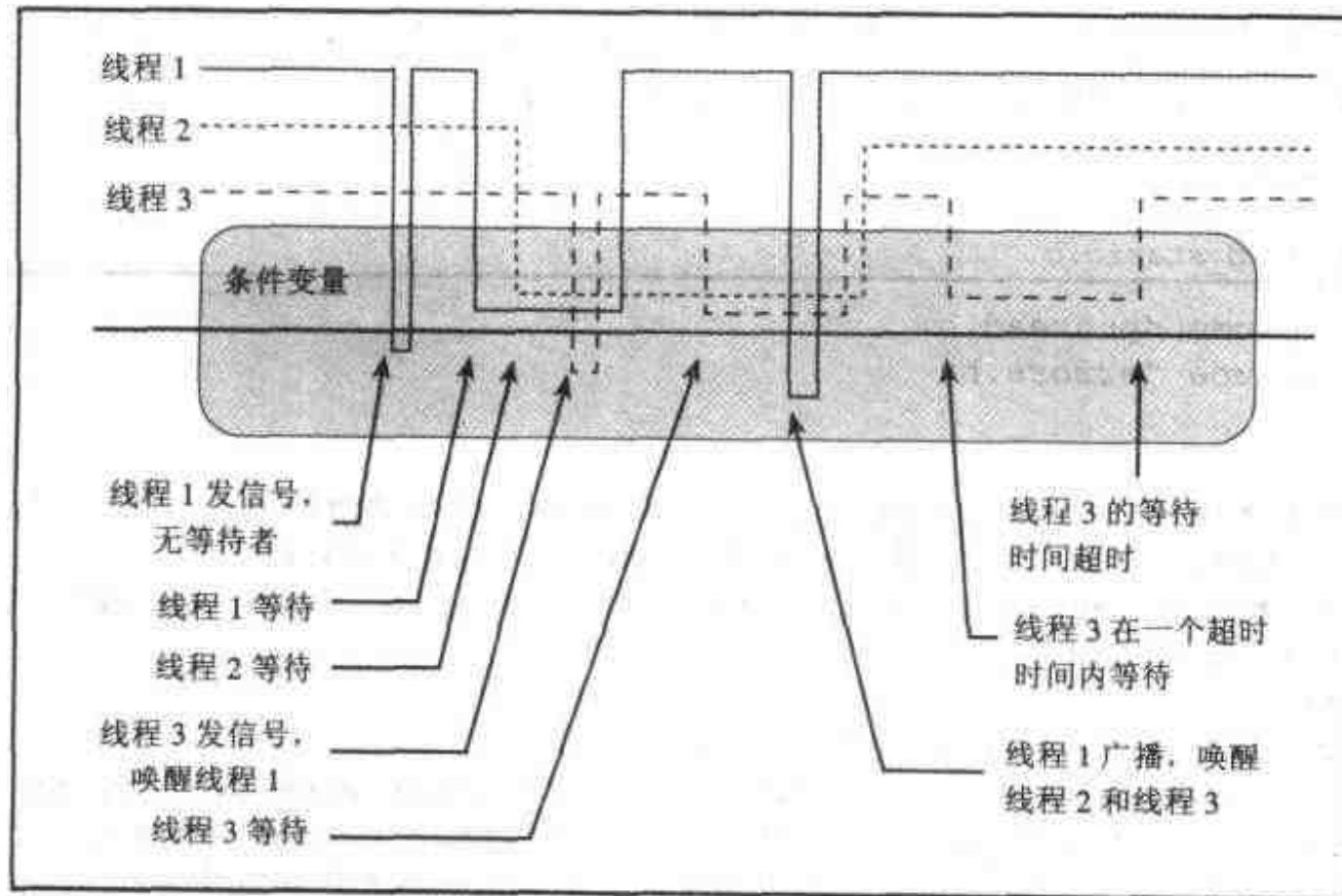


图 3.4 条件变量操作

当线段进入圆形框时，既表明线程使用条件变量做了一些事。当线程对应的线段在到达圆形框中线之前停止，表明线程在等待条件变量；当线程线段到达中线下时，表明它在发信号或广播来唤醒等待线程。

线程 1 等条件变量发信号，由于此时没有等待线程，所以没有任何效果。线程 1 然后在条件变量上等待。线程 2 同样在条件变量上阻塞，随后线程 3 发信号唤醒在条件变量上等待的线程 1。线程 3 然后在条件变量上等待。线程 1 广播条件变量，唤醒线程 2 和线程 3。随后，线程 3 在条件变量上等待。一段时间后，线程 3 的等待时间超时，线程 3 被唤醒。

3.3.1 创建和释放条件变量

```
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);
```

程序中由 `pthread_cond_t` 类型的变量来表示条件变量。永远不要拷贝条件变量，因为使用条件变量的备份是不可知的，这就像是打一个断线的电话号码并等待回答一样。例如，一个线程可能在等待条件变量的一个拷贝，同时其他线程可能广播或发信号给该条件变量的其他拷贝，则该等待线程就不能被唤醒。不过，可以传递条件变量的指针以使不同函数和线程可以使用它来同步。

大部分时间你可能在整个文件范围内（即不在任何函数内部）声明全局或静态类型条件变量。如果有其他文件需要使用，则使用全局（`extern`）类型；否则，使用静态（`static`）类型。如下面实例 `cond_static.c` 所示，如果声明了一个使用默认属性值的静态条件变量，则需要使用 `PTHREAD_COND_INITIALIZER` 宏初始化。

■ 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
```

| 为获得最好的结果，应该将条件变量与相关的谓词“链接”在一起对待。

当声明条件变量时，要记住条件变量与相关的谓词是“链接”在一起的。总是将二者一起声明将使你（或你的后来者）免于一些混淆。建议你将一组不变量、谓词和它们的互斥量，以及一个或多个条件变量封装为一个数据结构的元素，并仔细地记录下它们之间的关系。

有时无法静态地初始化一个条件变量，例如，当使用 `malloc` 分配一个包含条件变量的结构时。这时，你需要调用 `pthread_cond_init` 来动态地初始化条件变量，如以下实例 `cond_dynamic.c` 所示。还可以动态初始化静态声明的条件变量，但是必须确保每个条件变量在使用之前初始化且仅初始化一次。你可以在建立任何线程前初始化它，或者使用 `pthread_once`（见 5.1 节）。如果需要使用非默认属性初始化条件变量，必须使用动态初始化（见 5.2.2 节）。

■ 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

当动态初始化条件变量时，应该在不需要它时调用 `pthread_cond_destroy` 来释放它。不必释放一个通过 `PTHREAD_COND_INITIALIZER` 宏静态初始化的条件变量。

当你确信没有其他线程在某条件变量上等待，或者将要等待、发信号或广播时，可以安全地释放该条件变量。判定上述情况的最好方式是在刚刚成功地广播了该条件变量、唤醒了所有等待线程的线程内，且确信不再有线程随后使用它时安全释放。

在线程从列表中删除了一个包含条件变量的节点后，然后广播唤醒所有等待线程，此时释放该条件变量（在释放它占有的空间之前）是安全的，也是很好的主意。被唤醒线程在继续执行时应该检查等待的谓词，所以必须确保没有释放该谓词需要的资源，这可能需要额外的同步。

3.3.2 等待条件变量

```

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);

```

每个条件变量必须与一个特定的互斥量、一个谓词条件相关联。当线程等待条件变量时，它必须将相关互斥量锁住。记住，在阻塞线程之前，条件变量等待操作将解锁互斥量；而在重新返回线程之前，会再次锁住互斥量。

所有并发地（同时）等待同一个条件变量的线程必须指定同一个相关互斥量。例如，Pthreads 不允许线程 1 使用互斥量 A 等待条件变量 A，而线程 2 使用互斥量 B 等待条件变量 A。不过，以下情况是十分合理的：线程 1 使用互斥量 A 等待条件变量 A，而线程 2 使用互斥量 A 等待条件变量 B。即，任何条件变量在特定时刻只能与一个互斥量相关联，而互斥量则可以同时与多个条件变量关联。

在锁住相关的互斥量之后和在等待条件变量之前，测试谓词是很重要的。如果线程发信号或广播一个条件变量，而没有线程在等待该条件变量时，则什么也没发生。如果在这之后，有线程调用 `pthread_cond_wait`，则它将一直等待下去而

无视该条件变量刚刚被广播的事实，这将意味着该线程可能永远不被唤醒。因为在线程等待条件变量之前，互斥量一直被锁住，所以，在测试谓词和等待条件变量之间无法设置谓词——互斥量被锁住，没有其他线程可以修改共享数据，包括谓词。

| 总是测试你的谓词，然后再次测试它。

当线程醒来时，再次测试谓词同样重要。应该总是在循环中等待条件变量，来避免程序错误、多处理器竞争和假唤醒。以下实例 cond.c，显示了如何等待条件变量。合适的谓词循环同样包含在其他使用条件变量的代码中，如在 3.3.4 节中的 alarm_cond.c 中。

20~37 wait_thread 线程睡眠一段时间以允许主线程在被唤醒之前条件变量等待操作，设置共享的谓词 (data.value)，然后发信号给条件变量。Wait_thread 线程等待的时间由 hibernation 变量控制，默认是 1 秒。

51~52 如果程序带参数运行，则将该参数解析为整数值，保存在 hibernation 变量中。这将控制 wait_thread 线程在发送条件变量的信号前等待的时间。

68~83 主线程调用 pthread_cond_timedwait 函数等待至多 2 秒（从当前时间开始）。如果 hibernation 变量设置为大于两秒的值，则条件变量等待操作将超时，返回 ETIMEDOUT。如果 hibernation 变量设为 2 秒，则主线程和 wait_thread 线程发生竞争，并且每次运行的结果可能不同。如果 hibernation 变量设置为小于 2 秒，则条件变量等待操作不会超时。

■ 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

有很多理由解释在线程醒来时检测谓词是否为真是个不错的主意。以下是几个主要的原因：

被拦截的唤醒：线程是异步执行的。从条件变量等待中唤醒包括加锁相应的互斥量。但是，如果其他线程首先获得了互斥量会发生什么？例如，它可能在等待之前先检查谓词。由于谓词现在为真，所以它不需要等待。如果谓词是“对于某工作可用的”，它将接受该工作。当它解锁互斥量时，可能已经没有可做的工作了。要确保最后被唤起的线程得到工作通常是一件昂贵、甚至起相反作用的事。

松散的谓词：由于很多原因，使用实际状态的近似值通常是容易的、方便的。例如，“可能有工作”而不是“有工作”。基于“松散的谓词”比基于真正的“严谨的谓词”发信号或广播通常更容易一些。如果你总是在等待条件变量之前和之后检测严谨的谓词，则当使用松散的谓词时，程序同样可用。而且，当条件变量被偶然地发信号或广播时，代码会更加健壮。使用松散谓词或偶然的唤醒可能是性能问题，但在很多情况下不会有什么区别。

假唤醒：这意味着当你在某个条件变量上等待时，等待可能（偶然地）返回而没有其他线程广播或发信号该条件变量。这听起来有些奇怪，但在某些多处理器系统中，使条件唤醒完全可预测将极大地减低所有条件变量操作的速度。导致假唤醒的竞争条件可以被认为是罕见的。

通常只需几条指令来重测谓词，而且这是个良好的编程规则。如果不重新检测谓词，可能导致后来难以跟踪的严重错误。所以不要做假设，总是在 while 循环中测试谓词来等待条件变量。

你还可以调用 `pthread_cond_timedwait` 函数，当到达某个时间后等待结束并返回 `ETIMEDOUT` 状态。时间是绝对时间，使用 POSIX.1b `struct timespec` 类型。由于超时是绝对时间而不是间隔（或时间增量），所以一旦你计算出了超时时间后，不管经过了假唤醒还是被拦截唤醒，它都将保持有效。使用时间段好像容易一些，但是在在线程每次被唤起之后和再次等待之前都需要重新计算超时时间，这需要知道已经等待了多长时间。

当超时的条件变量等待返回 `ETIMEDOUT` 错误时，应该在处理错误之前先检测谓词。如果等待的条件为真，则等待的时间过长就不重要了。记住，线程在从条件变量返回之前总是锁住互斥量，即使是等待超时。在超时之后等待一个锁住的互斥量将导致超时等待比你要求的时间长很多。

3.3.3 唤醒条件变量等待线程

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

一旦有线程为某个谓词在等待一个条件变量，你可能需要唤醒它。Pthreads 提供了两种方式唤醒等待的线程：一个是“发信号”，一个是“广播”。发信号只唤醒一个等待该条件变量的线程，而广播将唤醒所有等待该条件变量的线程。

术语“发信号”很容易与“POSIX 信号”机制相混淆。POSIX 信号机制允许你定义“信号活动”处理“信号掩码”等。然而，我们在这里使用的“发信号”已经独立地建立在线程编程的文献中和商业实现中，所以 Pthreads 工作组决定不修改“发信号”术语。幸运地是，很少有同时使用两个术语的可能，尽量避免在线程程序中使用“信号”(signal) 也是个很不错的主意。如果在可能有混淆的地方，我们小心地说“发信号条件变量”和“POSIX 信号”(或“UNIX 信号”)，就不太可能让人混淆。

很容易将“广播”视为一种通用的“发信号”，不过更准确的是将“发信号”视为“广播”的一种优化。记住，使用广播而不是发信号决不会错，因为等待线程必须自己解决被拦截唤醒和假唤醒。实际上，真正的区别是效率：广播将唤醒额外的等待线程，而这些线程会检测自己的谓词然后继续等待。通常，不能用发信号代替广播。“当有什么疑惑的时候，就使用广播”。

当只有一个线程需要被唤醒来处理改变后的状态(而任何等待线程都可以做这个工作)时，使用发信号。如果你为多个谓词条件使用一个条件变量，则不能使用发信号操作；因为你不能分辨是应该唤醒等待这个谓词条件的线程，还是唤醒等待那个谓词条件的线程。不要试图通过重发信号(当你检测谓词不为真时)绕过去，这可能不会如你所想的传递信号，一个假唤醒或被拦截唤醒可能导致一系列无意义的重发信号。

如果向队列中增加一个元素，而且只有等待该元素的线程在条件变量上阻塞，则可以使用发信号来唤醒一个线程，而让其他线程继续等待，以避免不必要的环境切换。另一方面，如果向队列中增加了多个元素，你可能需要使用广播。7.1.2 节中的“读/写锁”提供了同时使用广播和发信号操作的实例。

尽管必须在等待条件变量时将相关的互斥量锁住，但是当发信号或广播条件变量时可以解锁相应的互斥量(如果这样更方便的话)。这样做的优点是在很多系统

中可以更加高效。当一个等待线程被唤醒的时候，它必须首先加锁互斥量。如果线程被唤醒而此时通知线程仍然锁住互斥量，则被唤醒线程会立刻阻塞在互斥量上。经过两次环境切换，你又回到了起点^①。

反面的问题是，如果互斥量不被锁住，任何线程（不仅仅是被唤醒的线程）可以在被唤醒线程之前锁住互斥量。这将是一个产生被拦截唤醒的根源。例如，一个低优先级的线程可能锁住了互斥量，使高优先级的线程无法被其他线程唤醒，延迟了高优先级线程的调度。如果在发信号的过程中保持互斥量加锁，就不会发生这样的事情。高优先级等待线程将被调于低优先级线程之前（在互斥量等待队列中），所以将被首先调度。

3.3.4 闹铃实例的最终版本

是实现闹铃程序的最终版本的时候了。在 `alarm_mutex.c` 中，通过不为每个闹铃生成单独的执行体（线程或进程）来减少资源利用率。使用单个线程来处理所有的闹铃请求，不过这种方法有个问题，就是无法响应新的闹铃请求。它必须在处理完当前闹铃后，才能检测其他闹铃请求是否已经被加入了列表，即使新的请求的到期时间比当前请求早。例如，首先输入命令行“10 message 1”，然后输入“5 message 2”。

既然线程编程工具库中已经增加了条件变量，我们就可以解决这个问题了。新的版本 `alarm_cond.c` 使用一个超时条件变量等待操作代替睡眠操作，以等待闹铃到时。当主线程在列表中添加了一个新的请求时，将发信号给条件变量，立刻唤醒 `alarm_thread` 线程。`alarm_thread` 线程可以重排等待的闹铃请求，然后重新等待。

20~22 第一部分显示了 `alarm_cond.c` 中的声明。与 `alarm_mutex.c` 相比，增加了两部分：一个条件变量 `alarm_cond` 和一个 `current_alarm` 变量，以允许主线程决定 `alarm_thread` 线程等待的闹铃请求的到期时间。`Current_alarm` 变量是为优化程序而设置的，除非 `alarm_thread` 线程空闲或者在等待一个比新请求晚的闹铃时，否则主线程不会（不需要）唤醒它。

■ `alarm_cond.c` part 1 declarations

```
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
```

① 有一个最优化的方法，我称之为 wait morphing，在这种情况下，当互斥量锁住时，将线程直接从条件变量等待队列中移到互斥量等待队列中，不需要任何环境切换。这种方法将为很多应用带来本质的性能优化。

```

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**

第二部分显示了新的函数 `alarm_insert`。该函数与 `alarm_mutex.c` 中的列表插入代码几乎相同，区别在于当需要时给条件变量 `alarm_cond` 发信号。将函数 `alarm_insert` 分离出来是由于它需要被两个地方调用：一个是主线程调用来插入新的请求，一个是 `alarm_thread` 线程调用来重新插入被新请求抢占的当前闹铃。

9~14 我已经建议将互斥量加锁协议记录下来，本例就是如此：`alarm_insert` 函数明确要求必须在锁住互斥量 `alarm_mutex` 时才能调用它。

48~53 如果 `current_alarm`（当前闹铃的到期时间）为 0，则表明 `alarm_thread` 线程在等待处理新的闹铃请求。如果 `current_alarm` 大于新来的闹铃请求到期时间，则 `alarm_thread` 线程不能在处理完当前闹铃后继续处理新的请求。以上两种情况，主线程都要发信号给 `alarm_cond` 条件变量，以唤醒 `alarm_thread` 线程处理新到的请求。

■ 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;

```

```

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

第三部分显示了 alarm_thread 函数，即闹铃服务线程的启动函数。该函数与程序 alarm_mutex.c 中的 alarm_thread 函数整体结构相同，区别在于增加了条件变量。

如果 alarm_list 为空，则程序 alarm_mutex.c 只能睡眠以使主线程可以处理新的用户命令。结果是可能无法看到新来的请求（至少 1 秒内）。现在，alarm_thread 在条件变量 alarm_cond 上等待。它将一直等到你输入一个新的闹铃请求时，主线程将立刻唤醒它。将 current_alarm 变量设为 0，告诉主线程该线程空闲。记住，pthread_cond_wait 在等待之前解锁互斥量，而在返回调用者之前重新加锁互斥量。

- 35 新加的变量 `expired` 被初始化为 0；如果超时条件等待过期则将它置为 1。这将易于判断在循环底部是否打印闹铃消息。
- 36~42 如果刚从列表中删除的闹铃还没有到期，则需要等待它。由于我们使用的超时条件等待操作需要 POSIX.1b Struct `timespec` 的时间，所以需要将到期时间做相应的转换。这很容易，因为 Struct `timespec` 包含两个元素：`tv_sec` 表示从 UNIX 纪元开始的秒数，`tv_nsec` 表示纳秒数。我们只须将 `tv_nsec` 元素置 0，因为不需要如此高的精度。
- 43 在变量 `current_alarm` 中记录到期时间，以使主线程能够判断当新的闹铃到达时是否需要发信号给条件变量 `alarm_cond`。
- 44~53 线程等待，直到当前闹铃到期或者主线程要求 `alarm_thread` 处理新的、到期时间更早的闹铃。注意，为方便起见，此处的谓词检测被分离。`while` 语句中的表达式仅是谓词的一半——检测主线程已经插入了更早闹铃时间的请求，并改变了 `current_alarm` 的值。当超时等待返回 `ETIMEDOUT` 时，表明当前闹铃已经到期，则在第 49 行退出循环。
- 54~55 如果当前闹铃没有到期而 `while` 循环退出，则主线程一定是要求 `alarm_thread` 线程处理更早的闹铃，确保将当前的闹铃请求重新插入队列中以免丢失。
- 57 如果从 `alarm_list` 列表中删除已经到期的闹钟，则将 `expired` 变量置为 1 以确保打印闹铃信息。

■ 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

第四部分显示 alarm_cond.c 的最后部分—— main 函数，它与 alarm_mutex.c 中的主函数几乎相同。

由于已经将条件变量信号操作内建到 alarm_insert 函数中，我们在主函数中调用 alarm_insert 插入闹铃请求。

■ 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 线程间的内存可视性

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

本章已经讲述了如何使用互斥量和条件变量同步（或协调）线程活动。现在，让我们离题一会儿，看一看线程世界中的“同步”到底意味着什么。同步不仅仅意味着防止两个线程同时写数据，当然这也是同步的部分工作。正如本节标题所示，同步是关于线程如何看待计算机的内存的。

Pthreads 提供了一些有关内存可视性的基本规则。你可以指望所有的标准实现都遵循以下规则：

1. 当线程调用 `pthread_create` 时，它所能看到的内存值也是它建立的线程能够看到的。任何在调用 `pthread_create` 之后向内存写入的数据，可能不会被建立的线程看到，即使写操作发生在启动新线程之前。
2. 当线程解锁互斥量时看到的内存中的数据，同样也能被后来直接锁住（或通过等待条件变量锁住）相同互斥量的线程看到。同样，在解锁互斥量之后写入的数据不必被其他线程看见，即使写操作发生在其他线程锁互斥量之前。
3. 线程终止（或者通过取消操作，或者从启动函数中返回，或者调用 `pthread_exit`）时看到的内存数据，同样能够被连接该线程的其他线程（通过调用 `pthread_join`）看到。当然，终止后写入的数据不会被连接线程看到，即使写操作发生在连接之前。
4. 线程发信号或广播条件变量时看到的内存数据，同样可以被唤醒的其他线程看到。而在发信号或广播之后写入的数据不会被唤醒的线程看到，即使写操作发生在线程被唤醒之前。

图 3.5 和 3.6 证明了其中的一些结论。那么，你作为一个程序员应该怎样做呢？

本例工作正确。左侧的代码（运行在线程 A 中）在锁住互斥量的过程中设置了几个变量的值，右边的代码（运行在线程 B 中）在锁住互斥量的过程中正确地读取了变量中的数据。

线程 A	线程 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>

规则 2：从 `pthread_mutex_unlock` 到 `pthread_mutex_lock` 之间可视化。当线程 B 从 `pthread_mutex_lock` 中返回时，它将和线程 A 在调用 `pthread_mutex_unlock` 时看到同样的变量值，即相应的 1 和 2。

图 3.5 正确的内存可视性

本例工作错误。左侧的代码（运行在线程 A 中）在解锁互斥量后设置了几个变量的值，右边的代码（运行在线程 B 中）在锁住互斥量的过程中正确地读取了变量中的数据。

线程 A	线程 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>

规则 2：从 `pthread_mutex_unlock` 到 `pthread_mutex_lock` 之间可视化。当线程 B 从 `pthread_mutex_lock` 中返回时，它将和线程 A 在调用 `pthread_mutex_unlock` 时看到同样的变量值，即它将看到变量 `variableA` 的值为 1，但不一定能够看到 `variableB` 的值，因为 `variableB` 是在解锁互斥量之后写入的。

图 3.6 错误的内存可视化

首先，确保哪些可能的地方只有一个线程能访问某段数据。线程的寄存器变量不能被其他线程修改。线程分配的堆栈和堆空间是私有的，除非线程将指向该内存的指针传给其他线程。任何放在 `register` 或 `auto` 变量中的数据可以在随后的某时刻读取，就像在完全同步的程序中一样。每个线程与自己是同步的。在线程间共享的数据越少，需要做的工作越多。

其次，任何时候两个线程需要访问相同数据时，你就需要应用其中一条内存可视化规则，大多数情况下是指使用互斥量。这不仅是为了保护多个写操作，即使线程只是读数据，它也需要锁住互斥量以确保读到最新的数据值。

正如规则所说的，有一些特别的情况不需要使用互斥量来确保可视化。如果线程设置了一个全局变量，然后创建了一个新线程读取同一个变量，则新线程将看不到旧的变量值（只能看到改过的新的变量值）。但是，如果你创建了一个新线程然后再设置变量的值，则新线程可能看不到新的变量值，即使原线程在新线程读取数据之前向变量内写入了新值。

警告！

下面我们将详细讲解与 Pthreads API 相关的内存硬件结构问题。如果你不想了解，现在可以跳过这些解释，将来再读。

如果你完全相信我所讲的东西（或者你已经知道得足够多了），你可以跳过本节后面部分。本书不是关于多处理器内存体系结构的，所以我只是肤浅地讲述一些，但即使如此，细节问题仍显得有些深。如果不关心这些，你不需要读下去。不过，

以后有时间的时候，你可以回头来读完本节的后续部分。

在单一线程中，即在完全同步的程序中，在任何时间读写任何内存都是安全的。即，如果程序向某个内存地址写入数据，然后在某个时刻从相同的内存地址读取数据，它总是能够得到最后更新的数据。

当你在程序中增加异步特性时（包括多处理器），有关内存可视性假设就变得复杂了。例如，在程序执行的任何时刻都可能产生异步信号。如果程序向内存中写入数据，随后信号处理函数运行，并向同一内存地址写入了不同的数据，当主程序再次运行时，它将得不到上次写入的数据。

通常这不是主要的问题，因为声明和使用信号处理函数是件复杂的事。它们在独立于主程序的执行环境中运行。有经验的程序员知道只有十分小心地写全局数据，才有可能跟踪它们所做的事。如果这太过于笨拙，你可以阻塞信号处理函数中使用全局数据的代码段。

当你在程序中添加多个线程时，异步代码不再特殊了。每个线程执行正常的程序代码，所有线程在同一个自由空间中运行。很难确定你总能知道线程在做什么。很可能它们会同时读或写相同的数据。你的线程可能运行不可知的次数，甚至同时运行在多个处理器上。这才是我们关心的时候。

尽管我们在讨论多线程编程，但本节涉及的实例没有一个是专门针对线程的。它们是内存结构设计的产物，它们可以应用于任何两个事物同时访问相同内存的情况。它们可能是运行在不同处理器上的两个线程；也可能是运行在不同处理器上、共享内存的两个进程；或者一个单一处理器上运行的代码，而另一个是读写相同内存的独立的 I/O 控制器。

| 一个内存地址一次只能保存一个值；不要让线程竞争以优先获得访问权。

当两个线程一个接一个地向同一内存地址写不同的数据时，最终的结果就像是一个线程按照相同的顺序写两次一样。在这两种情况下，都只有一个值保存在内存中。问题在于如何知道哪个写操作最后发生。通过绝对的外部时间测量，可能是处理器 A 先写入 1，几个微秒后处理器 B 写入 2，但这并不意味着最终的结果是“2”。

为什么？因为我们从未说过有关机器高速缓存和内存总线工作的细节。处理器可能有高速缓存，是用来保存最近从主存中读出数据的拷贝的快速本地内存。在一个“回写”（write-back）高速缓存系统中，数据最初只是保存在高速缓存中，而在后面的时间写入（刷新）主存。在不保证读写顺序的系统中，高速缓存中的数据块可能在任何处理器认为方便的时候写入主存。如果两个处理器向同一内存地址写不同的数据，则不同的数据将分别保存在二者的高速缓存中。最终两个值将写到主存中，但是在随机的时刻写入，与写到相应的高速缓存中的顺序无关。

即使同一线程（或处理器）中的两个写操作也不需要在内存中表现相同的顺序。内存控制器可能发现以相反的顺序写入会更快或更方便，如图 3.7 所示。例如，它

们可能被写入不同的高速缓存块中。通常，程序不可能知道这些影响。如果程序依赖这些，它们可能无法在其他的处理器模式中正确运行，或者只能运行在更少类型的计算机上。

时 间	线 程 1	线 程 2
t	向地址 1 (cache) 中写入 “1”	
t+1	向地址 2 (cache) 中写入 “2”	从地址 1 中读出 “0”
t+2	Cache 系统刷新地址 2	
t+3		从地址 2 中读出 “2”
t+4	Cache 系统刷新地址 1	

图 3.7 没有同步的内存排序

问题还不仅局限于两个线程写内存。想像一下，一个线程在某个处理器上向内存中写入数据，而另一个线程在其他处理器上读相同的内存。好像很显然地，线程将得到最近写入的值，而在有些硬件上的确如此。这种现象有时被称为“内存一致性”或“读写排序”。但是在处理器之间确保这种同步是复杂的，它会降低内存系统的速度，并且对大部分代码没有任何意义。现在很多的计算机（通常是最快的计算机）不再保证不同处理器间内存访问的顺序，除非程序使用特殊的指令，这些指令就是常说的“内存屏障”（memory barrier）。

在这些计算机中的内存访问，至少原则上是被内存控制器排队，并以任意一种最高效的顺序处理。如果读取没有在 cache 中包含的地址数据，则读操作将一直等到随后 cache 添入需要的数据后结束。如果向“脏”的 cache 地址中写数据，则写操作将等到将 cache 刷新后才能完成。内存屏障确保：所有在设置内存屏障之前发起的内存访问，必须先于在设置屏障之后发起的内存访问之前完成。

| 内存屏障是一堵移动的墙，而不是刷新 cache 的命令。

对于内存屏障常见的误解是，内存屏障是将 cache 刷新到主存中，以保证数据对其他处理器可视。然而事实并非如此，内存屏障所做的是将一组操作排序。如果每个内存访问对应队列中的一个元素，你可以将内存屏障看作队列中特殊的令牌。不像其他的内存访问，内存控制器不能删除内存屏障，不能越过它，直到完成在内存屏障之前的所有操作。

例如，互斥量锁开始于锁住互斥量，而结束于发布一个内存屏障，结果是任何在互斥量锁住期间的内存操作不能在其他线程看到该互斥量被锁住之前完成。类似的，解锁互斥量开始于发布一个内存屏障，而结束于解锁互斥量，以确保任何在互斥量锁住期间的内存操作不能晚于其他线程看到该互斥量解锁完成。

内存屏障模型是隐藏在 Pthreads 内存规则描述后面的逻辑。对于每个规则而言，都有一个“源”事件（如线程调用 `pthread_mutex_unlock`）和“目标”事件（如其他线程从调用 `pthread_mutex_lock` 中返回）。由于在每个事件中小心地设置了内存屏障，所以“内存视图”从第一个线程传到第二个线程。

即使没有读写排序和内存屏障，写内存好象也必须是原子性的操作。即任何其他线程只能看到完整的旧数据或者看到完整的新数据。但这也并不总是正确的。大部分计算机有一个天然的内存粒度，依赖于内存的组织和总线结构。即使处理器读写 8 位数据，内存传输可能以 32 位或 64 位为单位。

这意味着，相对于其他覆盖相同 32 或 64 位数据单元的内存操作而言，8 位写操作不是原子操作。大部分计算机会写包含修改数据的整个内存数据单元（如 32 位）。如果两个线程向同一个 32 位内存单元中写入不同的 8 位数据，则结果是最后写入的线程将确定两个 8 位字节的值，即覆盖第一个线程写入的值。图 3.8 显示了这种效果。

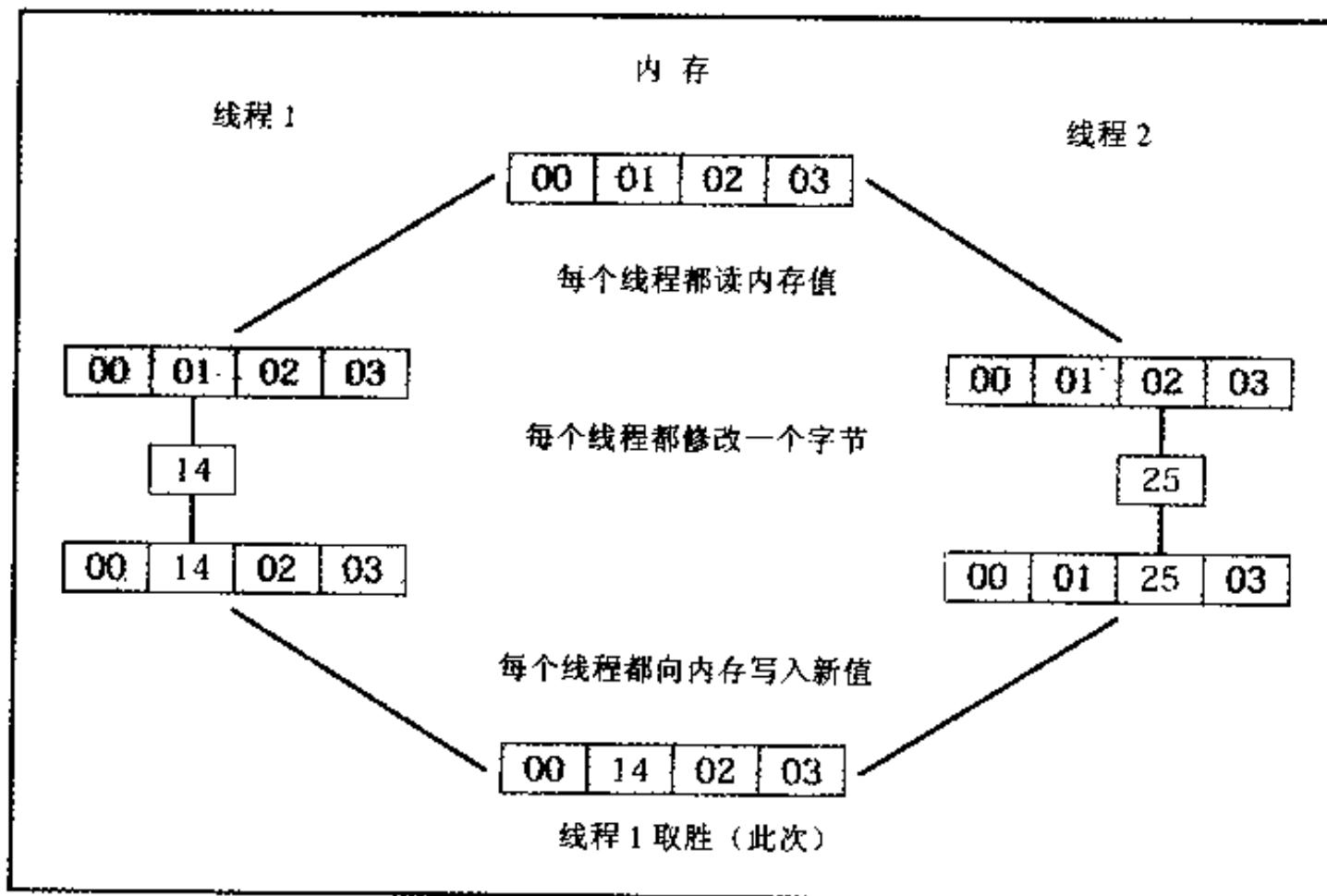


图 3.8 内存冲突

如果一个变量跨越了内存单元的界限（这将在允许非对齐内存访问的机器上出现），计算机不得不在两次总线事务中传送数据。例如，一个非对齐的 32 位数据可能需要写两个相邻的 32 位内存单元。如果其中一个内存单元同时被另一个处理器写入，则将丢失一半数据。这被称为“word tearing”，如图 3.9 所示。

最后我们回到本节开始给出的建议：如果想写可移植的代码，你要总使用 Pthreads 内存可视化规则来保证正确的内存可视性，而不能依赖于特定硬件和编译器相关的假设。现在，你应该理解为什么要这样做了。要获得更深入的有关多处理器内存体系结构的知识，请参考 *UNIX Systems for Modern Architectures*

[Schimmel, 1994]。

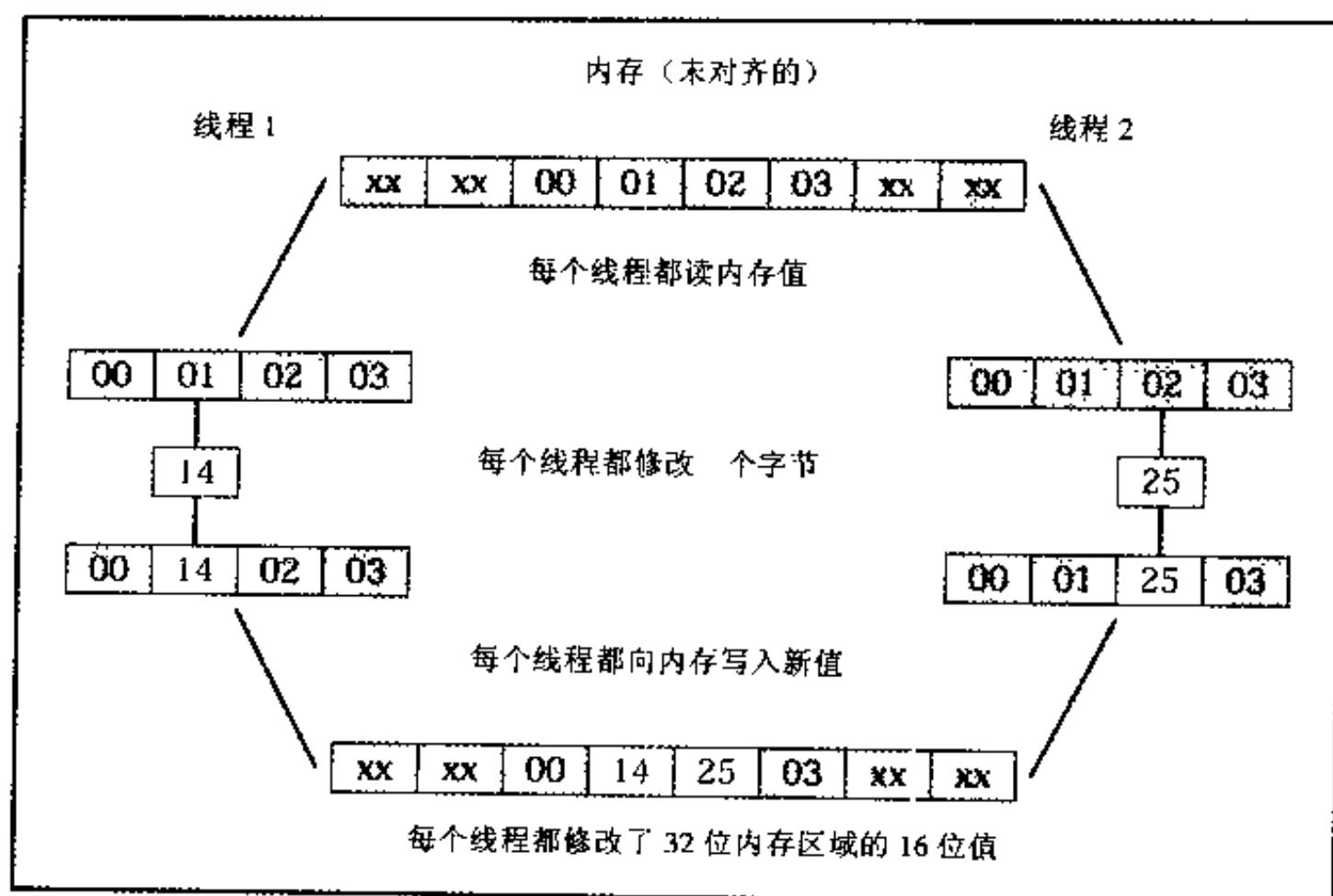


图 3.9 word tearing

图 3.10 显示了与图 3.7 相同的结果,但是它使用互斥量来保证期望的读写排序。图 3.10 中没有显示图 3.7 中的 cache 刷新步骤,因为这些步骤不再相关。内存可见性是通过在步骤 t+3 和 t+4 中传递互斥量所有权来保证的,通过相关联的内存屏障。即,当线程 2 成功地锁定了先前由线程 1 锁住的互斥量时,可以确保线程 2 看到的内存值和线程 1 在解锁互斥量时看到的内存值至少一样新。

时间	线程 1	线程 2
t	锁住互斥量 (内存屏障)	
t+1	向地址 1 (cache) 中写入 “1”	
t+2	向地址 2 (cache) 中写入 “2”	
t+3	(内存屏障) 解锁互斥量	
t+4		锁住互斥量 (内存屏障)
t+5		从地址 1 中读出 “1”
t+6		从地址 2 中读出 “2”
t+7		(内存屏障) 解锁互斥量

图 3.10 使用同步的内存排序

4

使用线程的几种方式

*"They were obliged to have him with them," the Mock Turtle said.
"No wise fish would go anywhere without a porpoise."
"Wouldn't it, really?" said Alice, in a tone of great surprise.
"Of course not," said the Mock Turtle. "Why, if a fish came to me,
and told me he was going on a journey, I should say 'With what porpoise?'"
—Lewis Carroll, Alice's Adventures in Wonderland*

在本书的第 1 章中，曾提及几种构建线程解决方案的方式。有无数种可能方式，表 4.1 显示了基本的线程编程模型。

表 4.1 线程编程模型

流水线	每个线程反复地在数据系列集上执行同一种操作，并把操作结果传递给下一步骤的其他线程，这就是“流水线”(assembly line) 方式
工作组	每个线程在自己的数据上执行操作。工作组中的线程可能执行同样的操作，也可能执行不同的操作，但是它们一定独立地执行
客户端/服务器	一个客户为每一件工作与一个独立的服务器“订契约”。通常“订契约”是匿名的——一个请求通过某种接口提交

所有上述模型可以以任意方式组合或修改来满足个人的要求。流水线中的一步可能包含向服务器线程提交请求，服务器线程可能使用工作组方式，一个或多个工作线程可能使用一条流水线，或者一个并行的搜索“引擎”可能启动多个线程，每个尝试不同的搜索算法。

4.1 流水线

*"I want a clean cup," interrupted the Hatter: "let's all move one place on."
He moved on as he spoke, and the Dormouse followed him: the March Hare
moved into the Dormouse's place, and Alice rather unwillingly took the
place of the March Hare. The Hatter was the only one who got any
advantage from the change; and Alice was a good deal worse off than
before, as the March Hare had just upset the milk-jug into his plate.
—Lewis Carroll, Alice's Adventures in Wonderland*

在流水线 (pipeline) 方式中，“数据元素”流串行地被一组线程顺序处理，如图 4.1 所示。每个线程依次在每个元素上执行一个特定的操作，并将结果传递给流水线中的下一个线程。

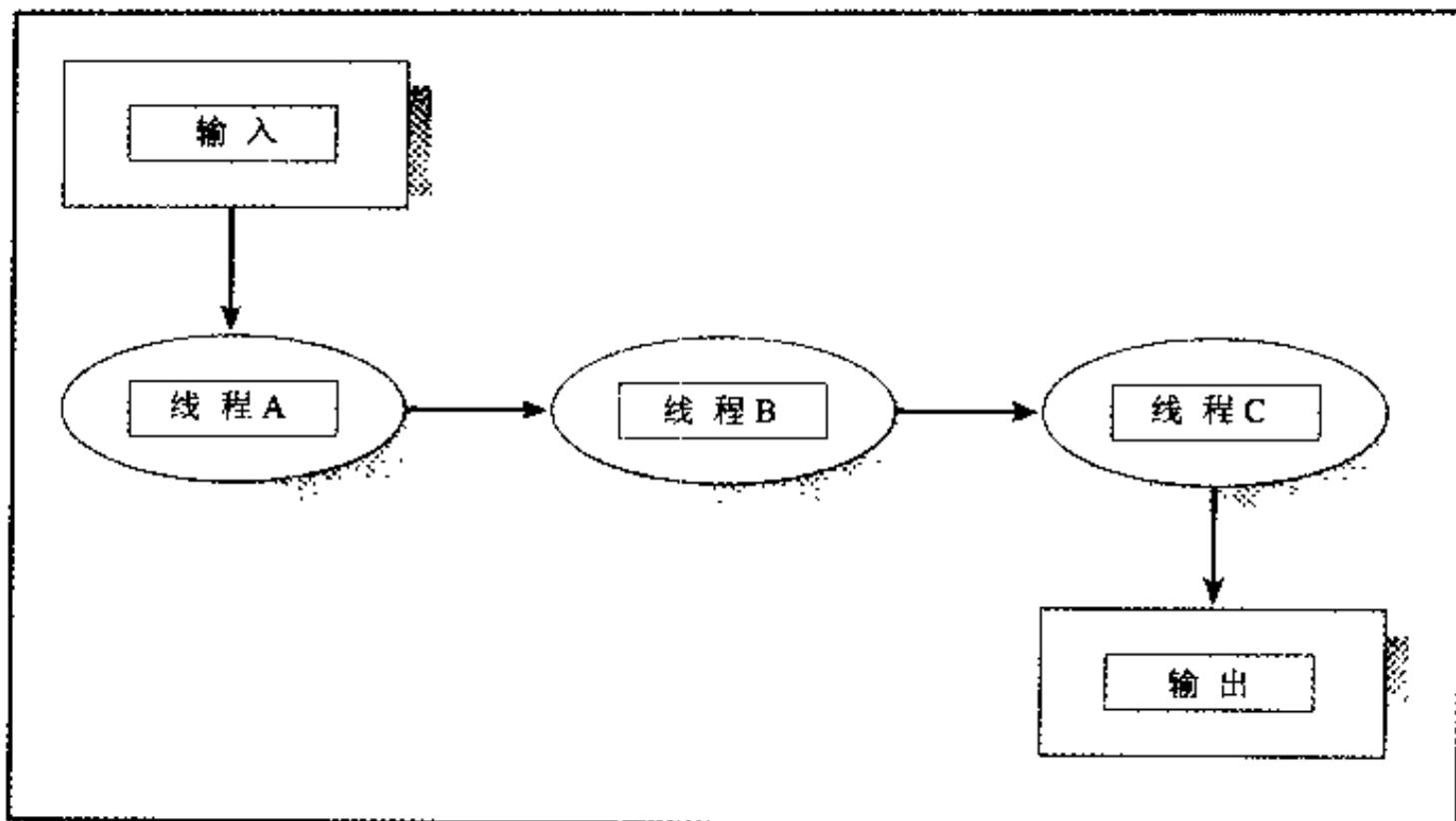


图 4.1 流水线

例如，数据可能是扫描的图像，线程 A 可能处理一个图像数组，线程 B 可能在处理的数据中搜索某个特定的属性集，而线程 C 可能控制从线程 B 中收集连续的搜索结果流并做出报告。或者每个线程可能执行某个数据修改序列中的一步。

下面的程序 pipe.c，显示了简单的流水线程序片段。流水线中的每个线程将它的输入加 1，并传给下一个线程。主线程从 stdin 中读取一系列的“命令行”。一个命令行或者是一个数字，它会被传给流水线的开始；或者是字符“=”，表示程序将从流水线结束处读取下一个结果并打印它。

9~17

流水线中的每一步是由一个 stage_t 类型的变量代表的。stage_t 包含一个同步访问的互斥量 mutex。条件变量 avail 用来通知某步要处理的数据已经准备好，每一步在准备好处理新数据时都会发信号给自己的 ready 条件变量。元素 data 是从前一步骤传递下来的数据，thread 表示执行该步骤的线程，next 指针指向下一步。

23~29

pipe_t 结构描述了一条流水线，它包含指向流水线第一步和最后一步的指针。第一步 head 表示流水线中的第一个线程，最后一步 tail 是一个特殊的 stage_t，因为没有对应的线程，它只是保存最终结果的地方。

■ pipe.c

part 1 definitions

```

1 #include <pthread.h>
2 #include "errors.h"
3
4 /*
5 * Internal structure describing a "stage" in the
  
```

```

6   * pipeline. One for each thread, plus a "result
7   * stage" where the final thread can stash the value.
8   */
9  typedef struct stage_tag {
10    pthread_mutex_t      mutex;          /* Protect data */
11    pthread_cond_t       avail;          /* Data available */
12    pthread_cond_t       ready;          /* Ready for data */
13    int                 data_ready;     /* Data present */
14    long                data;           /* Data to process */
15    pthread_t            thread;         /* Thread for stage */
16    struct stage_tag    *next;          /* Next stage */
17 } stage_t;
18
19 /*
20  * External structure representing the entire
21  * pipeline.
22 */
23 typedef struct pipe_tag {
24    pthread_mutex_t      mutex;          /* Mutex to protect pipe */
25    stage_t             *head;          /* First stage */
26    stage_t             *tail;          /* Final stage */
27    int                 stages;        /* Number of stages */
28    int                 active;        /* Active data elements */
29 } pipe_t;

```

■ pipe.c**part 1 definitions**

第二部分显示了 `pipe_send` 函数，该函数用来启动流水线中的数据，而且可以被每一步调用来向下传递数据。

- 17~23 开始等待特定流水线步骤的 `ready` 条件变量，直到该步能够接收新数据。
 28~30 保存新数据值，然后告诉该步骤数据可用。

■ pipe.c**part 2 pipe_send**

```

1 /*
2  * Internal function to send a "message" to the
3  * specified pipe stage. Threads use this to pass
4  * along the modified data item.
5  */
6 int pipe_send (stage_t *stage, long data)
7 {
8     int status;
9
10    status = pthread_mutex_lock (&stage->mutex);
11    if (status != 0)
12        return status;
13    /*
14     * If there's data in the pipe stage, wait for it
15     * to be consumed.
16     */
17    while (stage->data_ready) {
18        status = pthread_cond_wait (&stage->ready, &stage->mutex);
19        if (status != 0) {
20            pthread_mutex_unlock (&stage->mutex);

```

```

21         return status;
22     }
23 }
24 /*
25 * Send the new data
26 */
27 stage->data = data;
28 stage->data_ready = 1;
29 status = pthread_cond_signal (&stage->avail);
30 if (status != 0) {
31     pthread_mutex_unlock (&stage->mutex);
32     return status;
33 }
34 status = pthread_mutex_unlock (&stage->mutex);
35 return status;
36
37 }
```

■ pipe.c

part 2 pipe_send

第三部分显示了 pipe_stage 函数，该函数是流水线中每个线程的启动函数。线程参数是一个指向自身 stage_t 结构的指针。

16~27 线程一直循环下去，处理数据。因为 mutex 在循环外被锁住，线程似乎总是锁住流水线步骤的互斥量。然而，它的大部分时间用来在 avail 条件变量上等待新的数据。记住，线程在等待条件变量时，会自动地将相关的互斥量解锁。所以实际上在线程大部分的时间内，mutex 是处于解锁状态。

22~26 当接到新数据时，线程将自己的数据加 1，并把结果传递给下一步。然后，线程通过清除 data_ready 标志记录本步骤已经没有数据处理，并发信号 ready 条件变量来唤醒任何可能在该步骤上等待的线程。

■ pipe.c

part 3 pipe_stage

```

1 /*
2 * The thread start routine for pipe stage threads.
3 * Each will wait for a data item passed from the
4 * caller or the previous stage, modify the data
5 * and pass it along to the next (or final) stage.
6 */
7 void *pipe_stage (void *arg)
8 {
9     stage_t *stage = (stage_t*)arg;
10    stage_t *next_stage = stage->next;
11    int status;
12
13    status = pthread_mutex_lock (&stage->mutex);
14    if (status != 0)
15        err_abort (status, "Lock pipe stage");
16    while (1) {
17        while (stage->data_ready != 1) {
18            status = pthread_cond_wait (&stage->avail, &stage->mutex);
19            if (status != 0)
```

```

20             err_abort (status, "Wait for previous stage");
21         }
22         pipe_send (next_stage, stage->data + 1);
23         stage->data_ready = 0;
24         status = pthread_cond_signal (&stage->ready);
25         if (status != 0)
26             err_abort (status, "Wake next stage");
27     }
28     /*
29      * Notice that the routine never unlocks the stage->mutex.
30      * The call to pthread_cond_wait implicitly unlocks the
31      * mutex while the thread is waiting, allowing other threads
32      * to make progress. Because the loop never terminates, this
33      * function has no need to unlock the mutex explicitly.
34      */
35 }
```

■ pipe.c

part 3 pipe_stage

第四部分显示 pipe_create——建立流水线的函数。该函数可以建立包含任意步骤的流水线，并在列表中连接它们。

18~34 对于每一步，程序分配一个新的 stage_t 结构并初始化各个成员。注意，分配和初始化了一个额外的“步骤”来保存流水线的最终结果。

36~37 最后一步的连接成员被置为 Null，流水线的 tail 指针指向最后一步。tail 小指针允许 pipe_result 函数容易地发现流水线的最终结果，即保存在最后一步中。

52~59 在所有步骤数据初始化之后，pipe_create 为每一步建立一个线程。额外的“最后一步”没有对应的线程。for 循环的终止条件是当前步骤的 next 指针为空，即意味着不会处理最后一步。

■ pipe.c

part 4 pipe_create

```

1  /*
2   * External interface to create a pipeline. All the
3   * data is initialized and the threads created. They'll
4   * wait for data.
5   */
6  int pipe_create (pipe_t *pipe, int stages)
7  {
8      int pipe_index;
9      stage_t **link = &pipe->head, *new_stage, *stage;
10     int status;
11
12     status = pthread_mutex_init (&pipe->mutex, NULL);
13     if (status != 0)
14         err_abort (status, "Init pipe mutex");
15     pipe->stages = stages;
16     pipe->active = 0;
17
18     for (pipe_index = 0; pipe_index <= stages; pipe_index++) {
```

```

19         new_stage = (stage_t*)malloc (sizeof (stage_t));
20         if (new_stage == NULL)
21             errno_abort ("Allocate stage");
22         status = pthread_mutex_init (&new_stage->mutex, NULL);
23         if (status != 0)
24             err_abort (status, "Init stage mutex");
25         status = pthread_cond_init (&new_stage->avail, NULL);
26         if (status != 0)
27             err_abort (status, "Init avail condition");
28         status = pthread_cond_init (&new_stage->ready, NULL);
29         if (status != 0)
30             err_abort (status, "Init ready condition");
31         new_stage->data_ready = 0;
32         *link = new_stage;
33         link = &new_stage->next;
34     }
35
36     *link = (stage_t*)NULL;      /* Terminate list */
37     pipe->tail = new_stage;    /* Record the tail */
38
39 /*
40  * Create the threads for the pipe stages only after all
41  * the data is initialized (including all links). Note
42  * that the last stage doesn't get a thread, it's just
43  * a receptacle for the final pipeline value.
44  *
45  * At this point, proper cleanup on an error would take up
46  * more space than worthwhile in a "simple example," so
47  * instead of cancelling and detaching all the threads
48  * already created, plus the synchronization object and
49  * memory cleanup done for earlier errors, it will simply
50  * abort.
51 */
52     for (    stage = pipe->head;
53             stage->next != NULL;
54             stage = stage->next) {
55         status = pthread_create (
56             &stage->thread, NULL, pipe_stage, (void*)stage);
57         if (status != 0)
58             err_abort (status, "Create pipe stage");
59     }
60     return 0;
61 }
```

■ pipe.c

part 4 pipe_create

第五部分显示了函数 `pipe_start` 和 `pipe_result`。函数 `pipe_start` 将一个数据元素推入流水线的开始部分，然后不等待结果立即返回。函数 `pipe_result` 允许当需要时可以让调用者等待最后的结果。

9-22 `pipe_start` 函数向第一步骤传递数据。函数增加流水线中“活动”元素的计数值，以便 `pipe_result` 函数检测没有其他活动元素需要收集，并立即返回而不是阻塞。你可能不是总希望流水线像这样工作。对于本实例这种方式是有意义的，

因为单一线程交替地“喂”和“读”流水线，如果用户不小心多读了数据，程序也不会永远挂起。

28~47 pipe_result 函数首先检查流水线中是否存在活动元素。如果没有，则解锁流水线互斥量，返回 0。

49~55 如果流水线中还有活动元素，pipe_result 锁住最后一步，等待接收数据。它拷贝数据然后重置最后一步，以使它能够接收下一个数据元素。记住最后一步没有线程，所以不能重置自己。

■ pipe.c**part 5 pipe_start,pipe_result**

```
1  /*
2   * External interface to start a pipeline by passing
3   * data to the first stage. The routine returns while
4   * the pipeline processes in parallel. Call the
5   * pipe_result return to collect the final stage values
6   * (note that the pipe will stall when each stage fills,
7   * until the result is collected).
8   */
9  int pipe_start (pipe_t *pipe, long value)
10 {
11     int status;
12
13     status = pthread_mutex_lock (&pipe->mutex);
14     if (status != 0)
15         err_abort (status, "Lock pipe mutex");
16     pipe->active++;
17     status = pthread_mutex_unlock (&pipe->mutex);
18     if (status != 0)
19         err_abort (status, "Unlock pipe mutex");
20     pipe_send (pipe->head, value);
21     return 0;
22 }
23
24 /*
25  * Collect the result of the pipeline. Wait for a
26  * result if the pipeline hasn't produced one.
27 */
28 int pipe_result (pipe_t *pipe, long *result)
29 {
30     stage_t *tail = pipe->tail;
31     long value;
32     int empty = 0;
33     int status;
34
35     status = pthread_mutex_lock (&pipe->mutex);
36     if (status != 0)
37         err_abort (status, "Lock pipe mutex");
38     if (pipe->active <= 0)
39         empty = 1;
40     else
41         pipe->active--;
```

```

42     status = pthread_mutex_unlock (&pipe->mutex);
43     if (status != 0)
44         err_abort (status, "Unlock pipe mutex");
45     if (empty)
46         return 0;
47
48     pthread_mutex_lock (&tail->mutex);
49     while (!tail->data_ready)
50         pthread_cond_wait (&tail->avail, &tail->mutex);
51     *result = tail->data;
52     tail->data_ready = 0;
53     pthread_cond_signal (&tail->ready);
54     pthread_mutex_unlock (&tail->mutex);
55     return 1;
56 }

```

■ pipe.c**part 5 pipe_start,pipe_result**

第六部分显示了驱动流水线的 main 程序。该函数建立一个流水线，然后循环从 stdin 中读入命令行。如果输入的是单个“=”字符，则它从流水线中读出最终结果并打印它。否则，它将输入转为整数值，并将它传给流水线处理。

■ pipe.c**part 6 main**

```

1  /*
2   * The main program to "drive" the pipeline...
3   */
4  int main (int argc, char *argv[])
5  {
6      pipe_t my_pipe;
7      long value, result;
8      int status;
9      char line[128];
10
11     pipe_create (&my_pipe, 10);
12     printf ("Enter integer values, or \"=\" for next result\n");
13
14     while (1) {
15         printf ("Data> ");
16         if (fgets (line, sizeof (line), stdin) == NULL) exit (0);
17         if (strlen (line) <= 1) continue;
18         if (strlen (line) <= 2 && line[0] == '=') {
19             if (pipe_result (&my_pipe, &result))
20                 printf ("Result is %ld\n", result);
21             else
22                 printf ("Pipe is empty\n");
23         } else {
24             if (sscanf (line, "%ld", &value) < 1)
25                 fprintf (stderr, "Enter an integer value\n");
26             else
27                 pipe_start (&my_pipe, value);

```

```

28 }
29 }
30 }

■ pipe.c

```

part 6 main

4.2 工作组

*The twelve jurors were all writing very busily on slates.
 "What are they doing?" Alice whispered to the Gryphon.
 "They can't have anything to put down yet, before the trial's begun."
 "They're putting down their names," the Gryphon whispered in reply,
 "for fear they should forget them before the end of the trial."*

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

在工作组模式中，数据由一组线程分别独立地处理（如图 4.2 所示）。循环的“并行分解”通常就是属于这种模式。例如，可能建立一组线程，每个线程负责处理数组的某些行或列。单一数据集合在线程间分离成不同部分，结果是一个（过滤后的）数据集。由于所有的工作线程在不同的数据部分上执行相同的操作，这种模式通常被称为 SIMD (single instruction, multiple data, 单指令多数据流) 并行处理。最初 SIMD 是以一种完全不同的并行形式使用，而不是专用于线程的，但是概念是相似的。

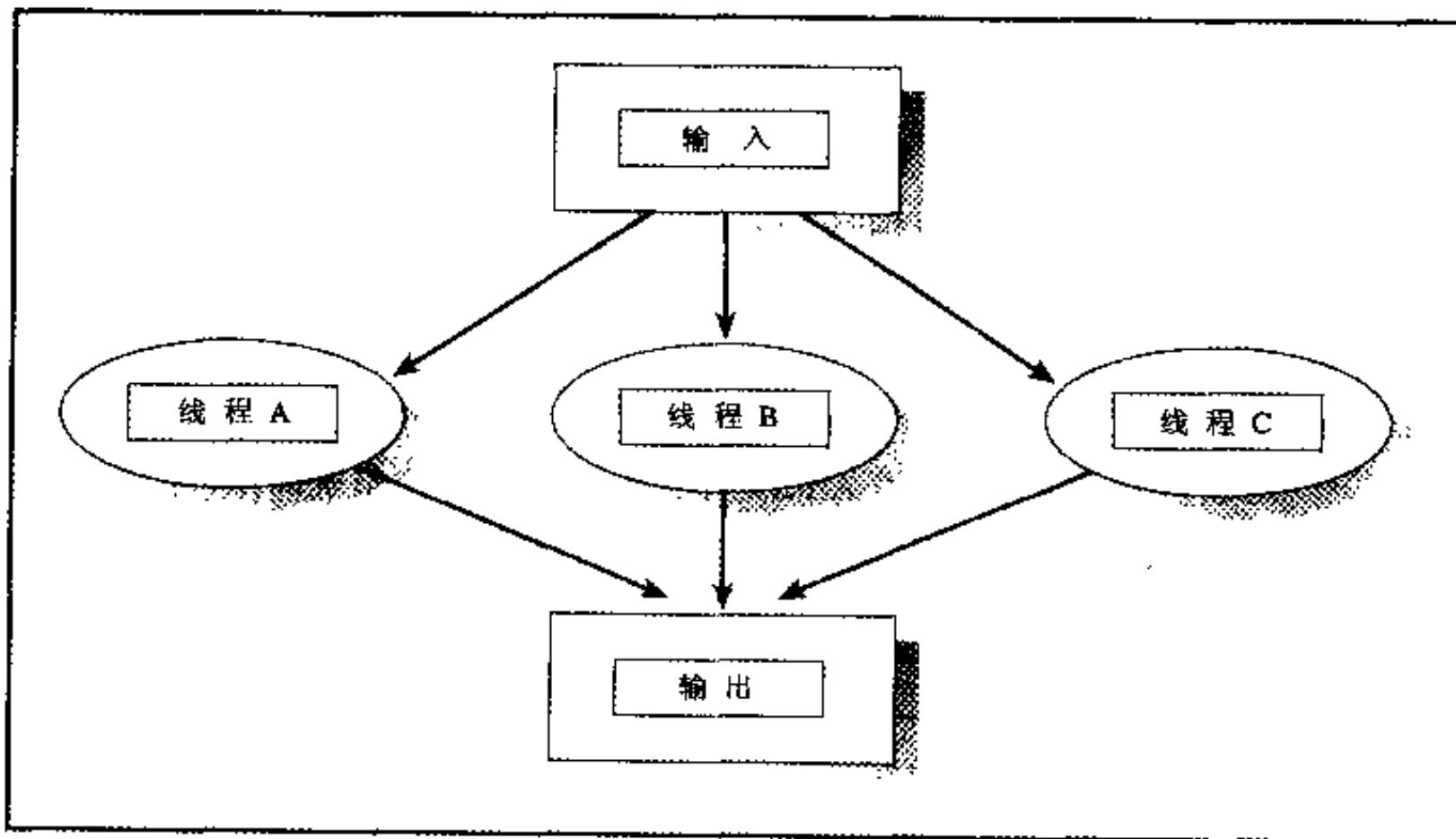


图 4.2 工作组模式

不过，工作组中的线程不必一定使用 SIMD 模型。它们完全可以在不同的数据上执行不同的操作。例如，我们工作组中的成员，每个从共享队列中删除工作请求、然后做请求的工作。每个排队可请求包可能描述不同的操作，但是相同的队列和“使

命”（处理队列内的请求）使得它们成为一个工作组而不是独立的工作线程。该模式可能与原始的 MIMD（multiple instruction, multiple data，多指令多数据）并行处理定义相似。

第 7.2 节显示了一个更健壮、更通用（当然也更复杂）的“工作队列管理器”包。“工作组”和“工作队列”之间的关系很像“不变量”和“临界区”的关系，即依赖于你如何看待发生的事情。工作组是一组独立处理数据的线程，而工作队列是一种机制，它要求数据被匿名的、独立的“代理”（agent）处理。因此在本节我们开发了一个“工作组”，而在 7.2 节我们会开发一个更加复杂的“工作队列”。尽管侧重点不同，但基本原则相同。

以下例程 `crew.c` 显示了一个简单的工作组。使用两个参数运行程序：一个字符串和一个文件路径。程序将文件路径排队给工作组。工作组成员将决定该路径是文件还是目录，如果是文件，它将在文件中搜索字符串；如果是路径，它将使用 `readdir_r` 来查找该目录中的所有子目录和常规文件，并将每一项作为一件新工作排队。每个包含指定字符串的文件将通过 `stdout` 报告。

第一部分显示了程序使用的头文件和定义。

- 7 符号 `CREW_SIZE` 决定为每个工作组建立多少线程。
- 13~17 每项工作由一个 `work_t` 结构描述。该结构包含一个指向下一工作的指针（设置 `NULL` 表明到达列表尾部），一个指向文件路径的指针和一个指向程序要搜索的字符串的指针。目前的设计是所有工作单元指向相同的搜索字符串。
- 23~27 每个工作组成员具有一个 `worker_t` 结构。该结构包含成员在工作组向量中的索引，对应的线程 ID 和指向 `crew_t` 结构（`crew`）的指针。
- 33~41 `crew_t` 结构描述工作组状态。它记录了工作组包含的成员数（`crew_size`）和一个 `worker_t` 结构（`crew`）数组。它还包含一个要处理工作项的数目（`work_count`）和一个待处理工作的链表（`first` 指向第一个工作，`last` 指向最后一项）。最后，它还包括几个 Pthreads 同步对象：访问控制的互斥量、等待其他成员完成任务的条件变量 `done` 和成员用以等待接收新工作的条件变量 `go`。
- 43~44 文件名和路径名的最大长度可能因不同的文件系统而不同。在一个成员启动后，程序通过调用 `pathconf` 来计算允许的文件名和指定文件的路径长度，并将值相应地保存在 `path_max` 和 `name_max` 中。

■ `crew.c`

part 1 definitions

```

1 #include <sys/types.h>
2 #include <pthread.h>
3 #include <sys/stat.h>
4 #include <dirent.h>
5 #include "errors.h"
6
7 #define CREW_SIZE      4
8
9 /*

```

```

10 * Queued items of work for the crew. One is queued by
11 * crew_start, and each worker may queue additional items.
12 */
13 typedef struct work_tag {
14     struct work_tag    *next;           /* Next work item */
15     char               *path;          /* Directory or file */
16     char               *string;        /* Search string */
17 } work_t, *work_p;
18
19 /*
20 * One of these is initialized for each worker thread in the
21 * crew. It contains the "identity" of each worker.
22 */
23 typedef struct worker_tag {
24     int                index;          /* Thread's index */
25     pthread_t          thread;         /* Thread for stage */
26     struct crew_tag   *crew;          /* Pointer to crew */
27 } worker_t, *worker_p;
28
29 /*
30 * The external "handle" for a work crew. Contains the
31 * crew synchronization state and staging area.
32 */
33 typedef struct crew_tag {
34     int                crew_size;      /* Size of array */
35     worker_t          crew[CREW_SIZE];/* Crew members */
36     long               work_count;    /* Count of work items */
37     work_t             *first, *last; /* First & last work item */
38     pthread_mutex_t   mutex;         /* Mutex for crew data */
39     pthread_cond_t    done;          /* Wait for crew done */
40     pthread_cond_t    go;            /* Wait for work */
41 } crew_t, *crew_p;
42
43 size_t  path_max;                  /* Filepath length */
44 size_t  name_max;                 /* Name length */

```

■ crew.c

part 1 definitions

第二部分显示了 `worker_routine`——工作线程的启动函数。外面的循环反复处理直到线程被告知终止。

20~23 POSIX 关于 `struct dirent` 类型的实际大小不太明确。实际上 `readdir_r` 要求传送一个缓冲区地址，该缓冲区足以包含一个 `struct dirent` 和一个至少 `NAME_MAX` 字节大小的名称元素。为确保有足够的空间，为缓冲区分配的大小应该等于一个 `struct dirent` 大小加上文件系统中文件名的最大长度。这可能比需要的大，应该保证它不会太小。

33~37 该条件变量循环阻塞每个新的工作成员直到有可做的工作。

61~65 这个等待语句有一点不同。当工作列表为空时，等待新的工作加入。成员线程永远不会终止——一旦完成了当前的任务，它们就准备好处理新的任务（本例中没

有利用该优点——一旦单个搜索命令完成，线程就终止)。

73~76 从队列中删除第一个工作元素。如果队列为空，则清除指向最后一个节点的指针：crew->last。

81~83 解锁工作组互斥量，以使工作组的大量工作可以并发地执行。

89 判定在工作元素中的路径字符串表示的文件类型。使用 lstat 调用返回符号链接的有关信息，而不是使用 stat 调用返回链接指向的文件信息。通过这种做法，减少了本例中的工作量，并且从根本上避免了跟踪链接进入其他文件系统，而在那个文件系统中我们指定的 name_max 和 path_max 大小可能不够。

91~95 如果指定文件是个链接文件，则报告文件名，除此之外不做其他处理。注意，每条消息包括线程在工作组中的索引 (mine->index)，以使你能够在本例中轻易地看到“并发工作”。

96~165 如果指定文件是个目录文件，则使用 opendir 调用打开目录。反复调用 readdir_r 获得目录中的所有条目。将每个目录条目作为新的工作元素进入。

166~206 如果指定文件是个常规文件，则打开该文件，读取所有文本数据，查找指定的搜索字符串。如果找到字符串，则退出搜索循环。

207~218 如果文件是其他类型，则写一条消息来试图辨认该文件类型。

232~252 重新加锁工作组互斥量，并报告完成一件工作。如果计数 count 为 0，则成员已经完成所分配的任务，广播唤醒任何等待发布新任务的等待线程。只有在工作元素都处理完毕时才能减少计数器 count 的值。如果任何成员仍然在忙（可能是在插入新的目录条目），则 count 就不会到达 0 值。

■ crew.c

part 2 worker_routine

```

1  /*
2   * The thread start routine for crew threads. Waits until "go"
3   * command, processes work items until requested to shut down.
4   */
5 void *worker_routine (void *arg)
6 {
7     worker_p mine = (worker_t*)arg;
8     crew_p crew = mine->crew;
9     work_p work, new_work;
10    struct stat filestat;
11    struct dirent *entry;
12    int status;
13
14    /*
15     * "struct dirent" is funny, because POSIX doesn't require
16     * the definition to be more than a header for a variable
17     * buffer. Thus, allocate a "big chunk" of memory, and use
18     * it as a buffer.
19     */
20    entry = (struct dirent*)malloc (
21        sizeof (struct dirent) + name_max);
22    if (entry == NULL)
```

```
23         errno_abort ("Allocating dirent");
24
25     status = pthread_mutex_lock (&crew->mutex);
26     if (status != 0)
27         err_abort (status, "Lock crew mutex");
28
29     /*
30      * There won't be any work when the crew is created, so wait
31      * until something's put on the queue.
32      */
33     while (crew->work_count == 0) {
34         status = pthread_cond_wait (&crew->go, &crew->mutex);
35         if (status != 0)
36             err_abort (status, "Wait for go");
37     }
38
39     status = pthread_mutex_unlock (&crew->mutex);
40     if (status != 0)
41         err_abort (status, "Unlock mutex");
42
43     DPRINTF (("Crew %d starting\n", mine->index));
44
45     /*
46      * Now, as long as there's work, keep doing it.
47      */
48     while (1) {
49         /*
50          * Wait while there is nothing to do, and
51          * the hope of something coming along later. If
52          * crew->first is NULL, there's no work. But if
53          * crew->work_count goes to zero, we're done.
54          */
55         status = pthread_mutex_lock (&crew->mutex);
56         if (status != 0)
57             err_abort (status, "Lock crew mutex");
58
59         DPRINTF (("Crew %d top: first is %#lx, count is %d\n",
60                   mine->index, crew->first, crew->work_count));
61         while (crew->first == NULL) {
62             status = pthread_cond_wait (&crew->go, &crew->mutex);
63             if (status != 0)
64                 err_abort (status, "Wait for work");
65         }
66
67         DPRINTF (("Crew %d woke: %#lx, %d\n",
68                   mine->index, crew->first, crew->work_count));
69
70         /*
71          * Remove and process a work item.
72          */
73         work = crew->first;
74         crew->first = work->next;
75         if (crew->first == NULL)
76             crew->last = NULL;
```

```
77
78     DPRINTF (("Crew %d took %#lx, leaves first %#lx, last %#lx\n",
79                 mine->index, work, crew->first, crew->last));
80
81     status = pthread_mutex_unlock (&crew->mutex);
82     if (status != 0)
83         err_abort (status, "Unlock mutex");
84
85     /*
86      * We have a work item. Process it, which may involve
87      * queuing new work items.
88      */
89     status = lstat (work->path, &filestat);
90
91     if (S_ISLNK (filestat.st_mode))
92         printf (
93             "Thread %d: %s is a link, skipping.\n",
94             mine->index,
95             work->path);
96     else if (S_ISDIR (filestat.st_mode)) {
97         DIR *directory;
98         struct dirent *result;
99
100        /*
101         * If the file is a directory, search it and place
102         * all files onto the queue as new work items.
103         */
104        directory = opendir (work->path);
105        if (directory == NULL) {
106            fprintf (
107                stderr, "Unable to open directory %s: %d (%s)\n",
108                work->path,
109                errno, strerror (errno));
110            continue;
111        }
112
113        while (1) {
114            status = readdir_r (directory, entry, &result);
115            if (status != 0) {
116                fprintf (
117                    stderr,
118                    "Unable to read directory %s: %d (%s)\n",
119                    work->path,
120                    status, strerror (status));
121                break;
122            }
123            if (result == NULL)
124                break;                  /* End of directory */
125
126            /*
127             * Ignore ".." and "..." entries.
128             */
129            if (strcmp (entry->d_name, ".") == 0)
```

```
130         continue;
131         if (strcmp (entry->d_name, ".") == 0)
132             continue;
133         new_work = (work_p)malloc (sizeof (work_t));
134         if (new_work == NULL)
135             errno_abort ("Unable to allocate space");
136         new_work->path = (char*)malloc (path_max);
137         if (new_work->path == NULL)
138             errno_abort ("Unable to allocate path");
139         strcpy (new_work->path, work->path);
140         strcat (new_work->path, "/");
141         strcat (new_work->path, entry->d_name);
142         new_work->string = work->string;
143         new_work->next = NULL;
144         status = pthread_mutex_lock (&crew->mutex);
145         if (status != 0)
146             err_abort (status, "Lock mutex");
147         if (crew->first == NULL) {
148             crew->first = new_work;
149             crew->last = new_work;
150         } else {
151             crew->last->next = new_work;
152             crew->last = new_work;
153         }
154         crew->work_count++;
155         DPRINTF ((
156             "Crew %d: add %#lx, first %#lx, last %#lx, %d\n",
157             mine->index, new_work, crew->first,
158             crew->last, crew->work_count));
159         status = pthread_cond_signal (&crew->go);
160         status = pthread_mutex_unlock (&crew->mutex);
161         if (status != 0)
162             err_abort (status, "Unlock mutex");
163     }
164
165     closedir (directory);
166 } else if (S_ISREG (filestat.st_mode)) {
167     FILE *search;
168     char buffer[256], *bufptr, *search_ptr;
169
170     /*
171      * If this is a file, not a directory, then search
172      * it for the string.
173      */
174     search = fopen (work->path, "r");
175     if (search == NULL)
176         fprintf (
177             stderr, "Unable to open %s: %d (%s)\n",
178             work->path,
179             errno, strerror (errno));
180     else {
181
182         while (1) {
```

```
183         bufptr = fgets (
184             buffer, sizeof (buffer), search);
185         if (bufptr == NULL) {
186             if (feof (search))
187                 break;
188             if (ferror (search)) {
189                 fprintf (
190                     stderr,
191                     "Unable to read %s: %d (%s)\n",
192                     work->path,
193                     errno, strerror (errno));
194                 break;
195             }
196         }
197         search_ptr = strstr (buffer, work->string);
198         if (search_ptr != NULL) {
199             printf (
200                 "Thread %d found \"%s\" in %s\n",
201                 mine->index, work->string, work->path);
202             break;
203         }
204     }
205     fclose (search);
206 }
207 } else
208     fprintf (
209         stderr,
210         "Thread %d: %s is type %o (%s))\n",
211         mine->index,
212         work->path,
213         filestat.st_mode & S_IFMT,
214         (S_ISFIFO (filestat.st_mode) ? "FIFO"
215          : (S_ISCHR (filestat.st_mode) ? "CHR"
216             : (S_ISBLK (filestat.st_mode) ? "BLK"
217                 : (S_ISSOCK (filestat.st_mode) ? "SOCK"
218                     : "unknown"))));
219
220     free (work->path);           /* Free path buffer */
221     free (work);                /* We're done with this */
222
223 /*
224  * Decrement count of outstanding work items, and wake
225  * waiters (trying to collect results or start a new
226  * calculation) if the crew is now idle.
227  *
228  * It's important that the count be decremented AFTER
229  * processing the current work item. That ensures the
230  * count won't go to 0 until we're really done.
231  */
232 status = pthread_mutex_lock (&crew->mutex);
233 if (status != 0)
234     err_abort (status, "Lock crew mutex");
235
236 crew->work_count--;
```

```

237     DPRINTF (( "Crew %d decremented work to %d\n", mine->index,
238             crew->work_count));
239     if (crew->work_count <= 0) {
240         DPRINTF (( "Crew thread %d done\n", mine->index));
241         status = pthread_cond_broadcast (&crew->done);
242         if (status != 0)
243             err_abort (status, "Wake waiters");
244         status = pthread_mutex_unlock (&crew->mutex);
245         if (status != 0)
246             err_abort (status, "Unlock mutex");
247         break;
248     }
249
250     status = pthread_mutex_unlock (&crew->mutex);
251     if (status != 0)
252         err_abort (status, "Unlock mutex");
253
254 }
255
256 free (entry);
257 return NULL;
258 }
```

■ crew.c**part 2 worker_routine**

第三部分显示了 `crew_create` 函数。该函数用来建立一个新的工作组，本例未提供删除工作组的功能，因为没有这个必要。工作组仅当程序准备退出时被删除，而且进程退出将删除所有的线程和处理数据。

- 12~15 `crew_create` 函数开始时检查 `crew_size` 参数。工作组成员数不能超过 `crew_t` 类型结构中工作组数组的大小。如果要求的大小可以接受，则将它拷贝到 `crew` 结构中。
- 16~31 开始时没有要处理的工作，工作队列为空。初始化工作组的同步对象。
- 36~43 对于每个成员，初始化成员的 `worker_t` 数据。记录成员在工作组数组中的索引值和指向 `crew_t` 的指针，然后，建立成员线程，使用指向成员 `worker_t` 结构的指针作为线程参数。

■ crew.c**part 3 crew_create**

```

1  /*
2   * Create a work crew.
3   */
4  int crew_create (crew_t *crew, int crew_size)
5  {
6      int crew_index;
7      int status;
8
9      /*
10       * We won't create more than CREW_SIZE members.
11      */
```

```

12     if (crew_size > CREW_SIZE)
13         return EINVAL;
14
15     crew->crew_size = crew_size;
16     crew->work_count = 0;
17     crew->first = NULL;
18     crew->last = NULL;
19
20     /*
21      * Initialize synchronization objects.
22      */
23     status = pthread_mutex_init (&crew->mutex, NULL);
24     if (status != 0)
25         return status;
26     status = pthread_cond_init (&crew->done, NULL);
27     if (status != 0)
28         return status;
29     status = pthread_cond_init (&crew->go, NULL);
30     if (status != 0)
31         return status;
32
33     /*
34      * Create the worker threads.
35      */
36     for (crew_index = 0; crew_index < CREW_SIZE; crew_index++) {
37         crew->crew[crew_index].index = crew_index;
38         crew->crew[crew_index].crew = crew;
39         status = pthread_create (&crew->crew[crew_index].thread,
40             NULL, worker_routine, (void*)&crew->crew[crew_index]);
41         if (status != 0)
42             err_abort (status, "Create worker");
43     }
44     return 0;
45 }
```

■ crew.c

part 3 crew_create

第四部分显示了 crew_start 函数，该函数用来为每个工作组指派新的路径名和搜索字符串。该函数是异步工作的——即在分派任务后，该函数等待工作组成员完成任务，然后返回给调用者。crew_start 函数假设已经通过调用 crew_create 创建了 crew_t 结构，如第三部分所示，但不会去验证结构的正确性。

20~26 等待工作组成员完成所有先前分配的工作。尽管 crew_start 函数异步执行，成员线程可能处理由其他线程分配的任务。在建立之初，工作组的 work_count 被置为 0，所以第一个对 crew_start 函数的调用不需要等待。

28~43 获得针对当前文件系统（由读入的文件路径指定）合适的 path_max 和 name_max 值。如果请求的值“无限制”，则函数 pathconf 可能返回-1 值，而不设置 errno 变量。为检测这种现象，需要在调用 pathconf 前清除 errno 变量。如果 pathconf 返回-1 且没有设置 errno，则可以设置为合理的值。

47~48 pathconf 返回的值不包括字符串的结束符，所以每个值加 1。
49~67 分配一个工作队列条目 (work_t)，并填充它，将它加到请求队列尾部。
68~75 我们已经排队了一个工作请求，所以通过发信号条件变量来唤醒其中一个等待成员。如果操作失败，则释放工作请求，清除工作队列，返回错误。
76~80 等待工作组完成任务。工作组成员处理所有的输出，所以当完成工作时简单地返回到调用者。

■ crew.c**part 4 crew_start**

```
1  /*
2   * Pass a file path to a work crew previously created
3   * using crew_create
4   */
5  int crew_start (
6      crew_p crew,
7      char *filepath,
8      char *search)
9  {
10     work_p request;
11     int status;
12
13     status = pthread_mutex_lock (&crew->mutex);
14     if (status != 0)
15         return status;
16
17     /*
18      * If the crew is busy, wait for them to finish.
19      */
20     while (crew->work_count > 0) {
21         status = pthread_cond_wait (&crew->done, &crew->mutex);
22         if (status != 0) {
23             pthread_mutex_unlock (&crew->mutex);
24             return status;
25         }
26     }
27
28     errno = 0;
29     path_max = pathconf (filepath, _PC_PATH_MAX);
30     if (path_max == -1) {
31         if (errno == 0)
32             path_max = 1024;           /* "No limit" */
33         else
34             errno_abort ("Unable to get PATH_MAX");
35     }
36     errno = 0;
37     name_max = pathconf (filepath, _PC_NAME_MAX);
38     if (name_max == -1) {
39         if (errno == 0)
40             name_max = 256;          /* "No limit" */
41         else
42             errno_abort ("Unable to get NAME_MAX");
```

```

43     }
44     DPRINTF ((  

45         "PATH_MAX for %s is %ld, NAME_MAX is %ld\n",  

46         filepath, path_max, name_max));  

47     path_max++; /* Add null byte */  

48     name_max++; /* Add null byte */  

49     request = (work_p)malloc (sizeof (work_t));  

50     if (request == NULL)  

51         errno_abort ("Unable to allocate request");  

52     DPRINTF (("Requesting %s\n", filepath));  

53     request->path = (char*)malloc (path_max);  

54     if (request->path == NULL)  

55         errno_abort ("Unable to allocate path");  

56     strcpy (request->path, filepath);  

57     request->string = search;  

58     request->next = NULL;  

59     if (crew->first == NULL) {  

60         crew->first = request;  

61         crew->last = request;  

62     } else {  

63         crew->last->next = request;  

64         crew->last = request;  

65     }  

66  

67     crew->work_count++;  

68     status = pthread_cond_signal (&crew->go);  

69     if (status != 0) {  

70         free (crew->first);  

71         crew->first = NULL;  

72         crew->work_count = 0;  

73         pthread_mutex_unlock (&crew->mutex);  

74         return status;  

75     }  

76     while (crew->work_count > 0) {  

77         status = pthread_cond_wait (&crew->done, &crew->mutex);  

78         if (status != 0)  

79             err_abort (status, "waiting for crew to finish");  

80     }  

81     status = pthread_mutex_unlock (&crew->mutex);  

82     if (status != 0)  

83         err_abort (status, "Unlock crew mutex");  

84     return 0;  

85 }

```

■ crew.c

part 4 crew_start

第五部分显示了该简单实例的初始线程 (main)。

- 10~13 程序要求三个参数：程序名、搜索字符串和路径名。例如，“crewbutenhof~”。
 15~23 在 Solaris 系统上，调用 thr_setconcurrency 来确保至少为每个工作组成员建立一个内核执行环境 (kernel execution context) LWP。没有该调用，程序仍能工作，但是在单处理器系统上，你不能看到任何并发。可以参考 5.6.3 小节以获得有关 many to few (多对少) 调度模型的更多信息，参考 10.1.3 小节获得 “set

concurrency (设置并发)”函数的信息。

24~30 建立一个工作组，将文件并发搜索任务分派给它。

```
■ crew.c                                         part 5    main
1  /*
2   * The main program to "drive" the crew...
3   */
4  int main (int argc, char *argv[])
5  {
6      crew_t my_crew;
7      char line[128], *next;
8      int status;
9
10     if (argc < 3) {
11         fprintf (stderr, "Usage: %s string path\n", argv[0]);
12         return -1;
13     }
14
15 #ifdef sun
16     /*
17      * On Solaris 2.5, threads are not timesliced. To ensure
18      * that our threads can run concurrently, we need to
19      * increase the concurrency level to CREW_SIZE.
20      */
21     DPRINTF (("Setting concurrency level to %d\n", CREW_SIZE));
22     thr_setconcurrency (CREW_SIZE);
23 #endif
24     status = crew_create (&my_crew, CREW_SIZE);
25     if (status != 0)
26         err_abort (status, "Create crew");
27
28     status = crew_start (&my_crew, argv[2], argv[1]);
29     if (status != 0)
30         err_abort (status, "Start crew");
31
32     return 0;
33 }
```

■ crew.c

part 5 main

4.3 客户/服务器

*But the Judge said he never had summed up before;
So the Snark undertook it instead,
And summed it so well that it came to far more
Than the Witnesses ever had said!*

—Lewis Carroll, *The Hunting of the Snark*

在客户服务器系统中，客户请求服务器对一组数据执行某个操作（如图 4.3 所示）。服务器独立地执行操作——客户端或者等待服务器执行，或者并行地执行并

在后面需要时查找结果。尽管让客户等待是最简单的，但这种方式很少有用——因为它不会为客户带来性能上的提高。另一方面，这又是一种对某些公共资源同步管理的简单方式。

如果一组线程都需要从 `stdin` 中读取输入，对它们来说，每个都独立地执行（提示-读）（prompt-and-read）操作可能有些混乱。想像两个线程，每个线程使用 `printf` 写它的提示，然后每个使用 `gets` 读取响应。你没有办法知道是在响应哪个线程。如果一个线程问“确认发送 email 吗？”，而另一个问“确认删除根目录吗？”你可能希望知道哪个线程将收到你的响应。当然有办法“连接地”保持提示并响应，而不需要引入服务线程。例如，在 prompt-and-read（提示-读）操作序列周围使用 `flockfile` 和 `funlockfile` 函数来锁住 `stdin` 和 `stdout`。但是，服务器线程更有趣并且与本节内容相关。

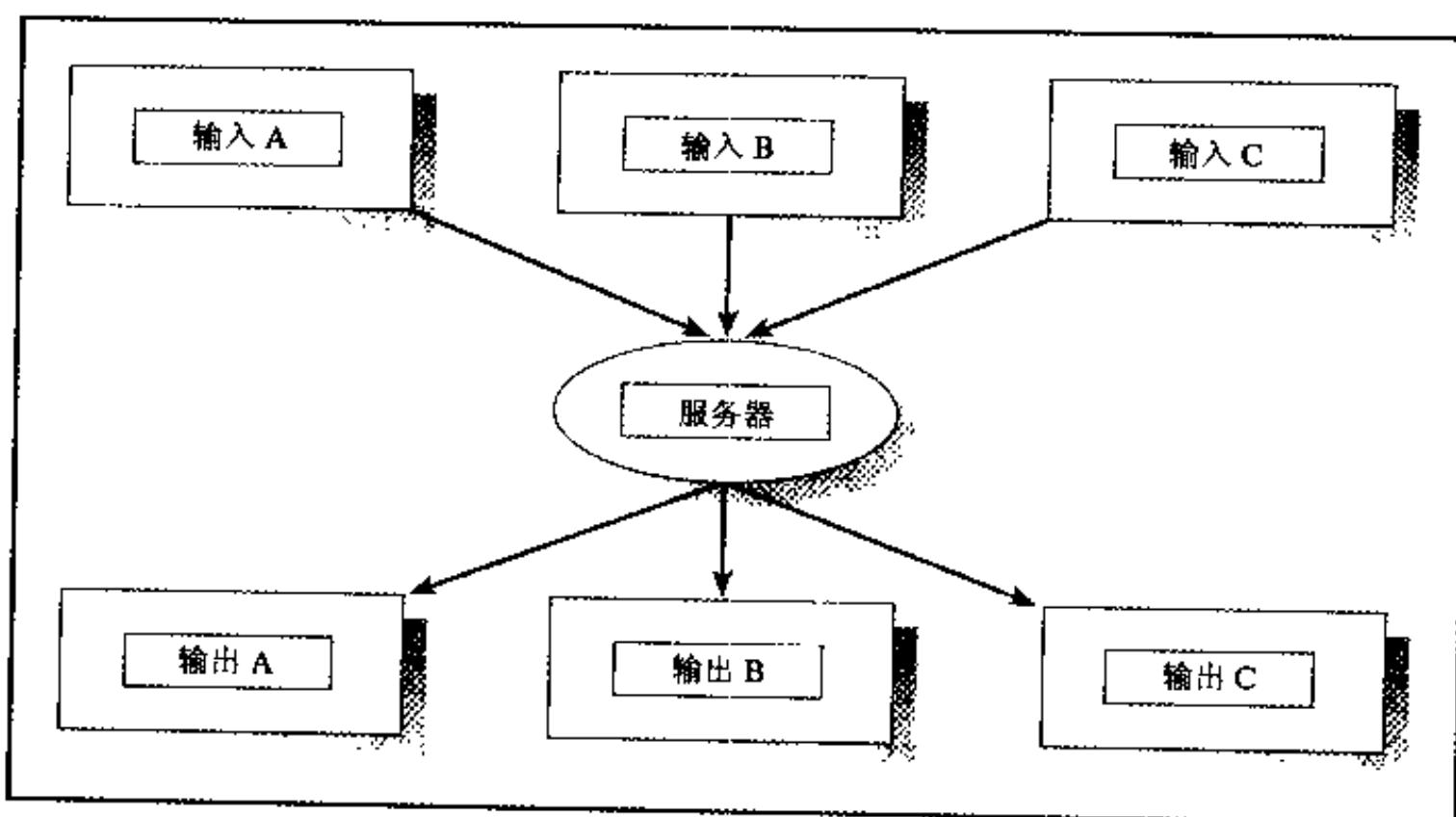


图 4.3 Client/Server

在下列程序 `server.c` 中，四个线程都将反复地读、响应输入行。当程序运行时，你应该看到线程提示以可变的顺序出现，其他线程可能会在当前线程响应之前提示。但是，你决不会在由“提示服务器”执行的提示和读操作之间看到一个提示或响应信息。

7~9

这些符号定义了传送给“提示服务器”的命令。它们可以用来要求读输入、写输出或退出。

14~22

`request_t` 结构定义了传给服务器的每个请求。当前请求通过 `next` 元素链接成链表。`operation` 元素包含一个请求编码（读、写或退出）。如果客户端希望等待操作完成（同步地），则 `synchronous`（同步）成员非 0；否则为 0。

27~33

`tty_server_t` 结构提供了服务器线程的环境。它包括同步对象（`mutex` 和 `request`）、一个表示服务器是否运行的标志和一个尚未处理的请求链表（`first` 和 `last`）。

- 35~37 该程序有一个服务器线程，控制结构 (tty_server) 静态被分配和初始化。请求链表为空，服务器没有运行。互斥量和条件变量被静态初始化。
- 43~45 主程序和客户线程通过同步对象 (client_mutex 和 clients_done) 协调它们的终止运行，而不是使用 pthread_join 调用。

■ server.c**part 1 definitions**

```

1 #include <pthread.h>
2 #include <math.h>
3 #include "errors.h"
4
5 #define CLIENT_THREADS 4           /* Number of clients */
6
7 #define REQ_READ      1           /* Read with prompt */
8 #define REQ_WRITE     2           /* Write */
9 #define REQ_QUIT      3           /* Quit server */
10
11 /*
12  * Internal to server "package" -- one for each request.
13  */
14 typedef struct request_tag {
15     struct request_tag *next;      /* Link to next */
16     int                 operation;   /* Function code */
17     int                 synchronous; /* Nonzero if synchronous */
18     int                 done_flag;   /* Predicate for wait */
19     pthread_cond_t      done;       /* Wait for completion */
20     char                prompt[32];  /* Prompt string for reads */
21     char                text[128];   /* Read/write text */
22 } request_t;
23
24 /*
25  * Static context for the server
26  */
27 typedef struct tty_server_tag {
28     request_t          *first;
29     request_t          *last;
30     int                running;
31     pthread_mutex_t    mutex;
32     pthread_cond_t     request;
33 } tty_server_t;
34
35 tty_server_t tty_server = {
36     NULL, NULL, 0,
37     PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER};
38
39 /*
40  * Main program data
41  */
42
43 int client_threads;
44 pthread_mutex_t client_mutex = PTHREAD_MUTEX_INITIALIZER;
45 pthread_cond_t clients_done = PTHREAD_COND_INITIALIZER;

```

■ server.c**part 1 definitions**

第二部分显示了服务器线程函数 `tty_server_routine`, 该函数循环连续地处理请求直到被要求退出。

- 25~30 服务器使用 `request` 条件变量等待请求出现。
- 31~34 从队列中删除第一个请求——如果队列现在为空，则清除指向最后一个元素的指针 (`tty_server.last`)。
- 43~66 `switch` 语句按照请求包中给定的 `operation`(操作)执行请求的工作。`REQ_QUIT` 告诉服务器退出。`REQ_READ` 告诉服务器读数据，包括可选的提示字符串。`REQ_WRITE` 告诉服务器写字符串。
- 67~79 如果请求标识为“同步”(`synchronous` 标志非 0)，则服务器置位 `done_flag` 标志并发信号 `done` 条件变量。当请求同步时客户端负责释放请求包。如果请求是异步的，则服务器在完成时负责释放请求。
- 80~81 如果请求是 `REQ_QUIT`，则通过跳出 `while` 循环来终止服务器线程。

■ server.c

part 2 tty_server_routine

```

1  /*
2   * The server start routine. It waits for a request to appear
3   * in tty_server.requests using the request condition variable.
4   * It processes requests in FIFO order. If a request is marked
5   * "synchronous" (synchronous != 0), the server will set done_flag
6   * and signal the request's condition variable. The client is
7   * responsible for freeing the request. If the request was not
8   * synchronous, the server will free the request on completion.
9   */
10 void *tty_server_routine (void *arg)
11 {
12     static pthread_mutex_t prompt_mutex = PTHREAD_MUTEX_INITIALIZER;
13     request_t *request;
14     int operation, len;
15     int status;
16
17     while (1) {
18         status = pthread_mutex_lock (&tty_server.mutex);
19         if (status != 0)
20             err_abort (status, "Lock server mutex");
21
22         /*
23          * Wait for data
24          */
25         while (tty_server.first == NULL) {
26             status = pthread_cond_wait (
27                 &tty_server.request, &tty_server.mutex);
28             if (status != 0)
29                 err_abort (status, "Wait for request");
30         }
31         request = tty_server.first;
32         tty_server.first = request->next;
33         if (tty_server.first == NULL)
34             tty_server.last = NULL;

```

```
35     status = pthread_mutex_unlock (&tty_server.mutex);
36     if (status != 0)
37         err_abort (status, "Unlock server mutex");
38
39     /*
40      * Process the data
41      */
42     operation = request->operation;
43     switch (operation) {
44         case REQ_QUIT:
45             break;
46         case REQ_READ:
47             if (strlen (request->prompt) > 0)
48                 printf (request->prompt);
49             if (fgets (request->text, 128, stdin) == NULL)
50                 request->text[0] = '\0';
51             /*
52              * Because fgets returns the newline, and we don't
53              * want it, we look for it, and turn it into a null
54              * (truncating the input) if found. It should be the
55              * last character, if it is there.
56              */
57             len = strlen (request->text);
58             if (len > 0 && request->text[len-1] == '\n')
59                 request->text[len-1] = '\0';
60             break;
61         case REQ_WRITE:
62             puts (request->text);
63             break;
64         default:
65             break;
66     }
67     if (request->synchronous) {
68         status = pthread_mutex_lock (&tty_server.mutex);
69         if (status != 0)
70             err_abort (status, "Lock server mutex");
71         request->done_flag = 1;
72         status = pthread_cond_signal (&request->done);
73         if (status != 0)
74             err_abort (status, "Signal server condition");
75         status = pthread_mutex_unlock (&tty_server.mutex);
76         if (status != 0)
77             err_abort (status, "Unlock server mutex");
78     } else
79         free (request);
80     if (operation == REQ_QUIT)
81         break;
82 }
83 return NULL;
84 }
```

(REQ_QUIT、REQ_READ 或 REQ_WRITE), 是否同步操作 (sync), 对 REQ_READ 操作可选的提示字符串 (prompt) 和一个字符串指针 (REQ_WRITE 的输入信息, 或者用来返回 REQ_READ 操作结果的缓冲区)。

16~40 如果 tty 服务器还没有运行, 则启动一个服务器。创建一个临时的线程属性对象 (detached_attr), 并将 detachstate 属性置为 PTHREAD_CREATE_DETACHED。线程数据将在 5.2.3 节解释。在本例中, 只是说明在创建线程后我们不再需要使用线程 ID。

45~76 分配和初始化一个服务器请求 (request_t) 包。如果是同步请求, 则初始化请求中的条件变量 (done), 否则不使用条件变量。新的请求被链接到请求队列中。

81~83 唤醒服务器线程来处理排队的请求。

88~105 如果是同步请求, 则等待服务器设置 done_flag 标志并发信号 done 条件变量。如果是 REQ_READ 操作, 则拷贝结果字符串到输出缓冲区中。最后, 释放条件变量, 释放请求包。

■ server.c

part 3 tty_server_request

```

1  /*
2   * Request an operation
3   */
4  void tty_server_request (
5      int          operation,
6      int          sync,
7      const char  *prompt,
8      char         *string)
9 {
10    request_t *request;
11    int status;
12
13    status = pthread_mutex_lock (&tty_server.mutex);
14    if (status != 0)
15        err_abort (status, "Lock server mutex");
16    if (!tty_server.running) {
17        pthread_t thread;
18        pthread_attr_t detached_attr;
19
20        status = pthread_attr_init (&detached_attr);
21        if (status != 0)
22            err_abort (status, "Init attributes object");
23        status = pthread_attr_setdetachstate (
24            &detached_attr, PTHREAD_CREATE_DETACHED);
25        if (status != 0)
26            err_abort (status, "Set detach state");
27        tty_server.running = 1;
28        status = pthread_create (&thread, &detached_attr,
29                               tty_server_routine, NULL);
30        if (status != 0)
31            err_abort (status, "Create server");

```

```
32
33         /*
34          * Ignore an error in destroying the attributes object.
35          * It's unlikely to fail, there's nothing useful we can
36          * do about it, and it's not worth aborting the program
37          * over it.
38          */
39         pthread_attr_destroy (&detached_attr);
40     }
41
42     /*
43      * Create and initialize a request structure.
44      */
45     request = (request_t*)malloc (sizeof (request_t));
46     if (request == NULL)
47         errno_abort ("Allocate request");
48     request->next = NULL;
49     request->operation = operation;
50     request->synchronous = sync;
51     if (sync) {
52         request->done_flag = 0;
53         status = pthread_cond_init (&request->done, NULL);
54         if (status != 0)
55             err_abort (status, "Init request condition");
56     }
57     if (prompt != NULL)
58         strncpy (request->prompt, prompt, 32);
59     else
60         request->prompt[0] = '\0';
61     if (operation == REQ_WRITE && string != NULL)
62         strncpy (request->text, string, 128);
63     else
64         request->text[0] = '\0';
65
66     /*
67      * Add the request to the queue, maintaining the first and
68      * last pointers.
69     */
70     if (tty_server.first == NULL) {
71         tty_server.first = request;
72         tty_server.last = request;
73     } else {
74         (tty_server.last)->next = request;
75         tty_server.last = request;
76     }
77
78     /*
79      * Tell the server that a request is available.
80      */
81     status = pthread_cond_signal (&tty_server.request);
82     if (status != 0)
83         err_abort (status, "Wake server");
84
```

```

85      /*
86       * If the request was "synchronous", then wait for a reply.
87       */
88     if (sync) {
89         while (!request->done_flag) {
90             status = pthread_cond_wait (
91                 &request->done, &tty_server.mutex);
92             if (status != 0)
93                 err_abort (status, "Wait for sync request");
94         }
95         if (operation == REQ_READ) {
96             if (strlen (request->text) > 0)
97                 strcpy (string, request->text);
98             else
99                 string[0] = '\0';
100        }
101        status = pthread_cond_destroy (&request->done);
102        if (status != 0)
103            err_abort (status, "Destroy request condition");
104        free (request);
105    }
106    status = pthread_mutex_unlock (&tty_server.mutex);
107    if (status != 0)
108        err_abort (status, "Unlock mutex");
109 }

```

■ server.c**part 3 tty_server_request**

第四部分显示了客户端线程的启动函数，该函数反复地将 tty 操作请求送至服务器排队。

12~22 通过 tty 服务器读一行请求。如果结果字符串为空，则跳出循环并终止。否则，以 1 秒的间隔循环四次打印结果字符串。为什么是四次？只是让事情“显得复杂”一些罢了。

26~31 减少客户端线程计数，如果最后的客户端线程终止，则唤醒主线程。

■ server.c**part 4 client_routine**

```

1 /*
2  * Client routine -- multiple copies will request server.
3  */
4 void *client_routine (void *arg)
5 {
6     int my_number = (int)arg, loops;
7     char prompt[32];
8     char string[128], formatted[128];
9     int status;
10
11    sprintf (prompt, "Client %d> ", my_number);
12    while (1) {
13        tty_server_request (REQ_READ, 1, prompt, string);
14        if (strlen (string) == 0)
15            break;

```

```

16         for (loops = 0; loops < 4; loops++) {
17             sprintf (
18                 formatted, "(%d%d) %s", my_number, loops, string);
19             tty_server_request (REQ_WRITE, 0, NULL, formatted);
20             sleep (1);
21         }
22     }
23     status = pthread_mutex_lock (&client_mutex);
24     if (status != 0)
25         err_abort (status, "Lock client mutex");
26     client_threads--;
27     if (client_threads <= 0) {
28         status = pthread_cond_signal (&clients_done);
29         if (status != 0)
30             err_abort (status, "Signal clients done");
31     }
32     status = pthread_mutex_unlock (&client_mutex);
33     if (status != 0)
34         err_abort (status, "Unlock client mutex");
35     return NULL;
36 }
```

■ server.c**part 4 client_routine**

第五部分显示了 server.c 的主程序，它建立了一组使用 tty 服务器的客户端线程，并等待它们运行。

7~15 在 Solaris 系统中，通过调用 `thr_setconcurrency` 设置并发级别为客户端线程的数量。由于所有的客户端线程将花费部分时间等待条件变量，我们并不真正需要增加并发级别。不过，那样做将减少运行效果的可预见性。

20~26 建立客户端线程。

27~35 该结构与 `pthread_join` 很相似，除了它只在所有客户端线程已经终止时完成外。正如我曾说过的，`pthread_join` 并没有什么神奇的，除非它恰恰符合你的功能要求，否则没有理由使用它来检测线程是否终止。很少有人希望在一个循环中使用 `pthread_join` 连接多个线程，而像此处显示的多连结（multiple join）更容易实现。

■ server.c**part 5 main**

```

1 int main (int argc, char *argv[])
2 {
3     pthread_t thread;
4     int count;
5     int status;
6
7 #ifdef sun
8     /*
9      * On Solaris 2.5, threads are not timesliced. To ensure
10     * that our threads can run concurrently, we need to
11     * increase the concurrency level to CLIENT_THREADS.
12     */

```

```
13     DPRINTF (( "Setting concurrency level to %d\n", CLIENT_THREADS));
14     thr_setconcurrency (CLIENT_THREADS);
15 #endif
16
17     /*
18      * Create CLIENT_THREADS clients.
19      */
20     client_threads = CLIENT_THREADS;
21     for (count = 0; count < client_threads; count++) {
22         status = pthread_create (&thread, NULL,
23             client_routine, (void*)count);
24         if (status != 0)
25             err_abort (status, "Create client thread");
26     }
27     status = pthread_mutex_lock (&client_mutex);
28     if (status != 0)
29         err_abort (status, "Lock client mutex");
30     while (client_threads > 0) {
31         status = pthread_cond_wait (&clients_done, &client_mutex);
32         if (status != 0)
33             err_abort (status, "Wait for clients to finish");
34     }
35     status = pthread_mutex_unlock (&client_mutex);
36     if (status != 0)
37         err_abort (status, "Unlock client mutex");
38     printf ("All clients done\n");
39     tty_server_request (REQ_QUIT, 1, NULL, NULL);
40     return 0;
41 }
```

■ server.c

part 5 main

5

线程高级编程

*"Tis the voice of the Jubjub!" he suddenly cried.
(This man, that they used to call "Dunce.")
"As the Bellman would tell you," he added with pride,
"I have uttered that sentiment once."*

—Lewis Carroll, *The Hunting of the Snark*

Pthreads 标准提供了很多简单项目用不到的功能。为了保持第 2 章和第 3 章的结构相对简单，更高级的功能被收集进本章。

5.1 节描述了管理数据初始化的功能，尤其是在多线程环境中函数库中的数据初始化。

5.2 节描述“属性对象”，通过它可以在创建线程、互斥量和条件变量时，控制它们各种各样的特性。

5.3 节描述取消线程，当你不需要它们继续时让它们“消失”的一种方式。

5.4 节描述线程的私有数据，这是一种数据库机制，允许函数库把数据与它遇到的单个线程联系起来，以便随后找到数据。

5.5 节描述 Pthreads 为实时调度提供的功能，帮助你的程序以可预测的方式与冷酷、残忍的世界交互。

5.1 一次性初始化

*"Take some more tea," the March Hare said to Alice, very earnestly.
"I've had nothing yet," Alice replied in an offended tone:
"so I can't take more."
"You mean you can't take less," said the Hatter.
"It's very easy to take more than nothing."
—Lewis Carroll, *Alice's Adventures in Wonderland**

```
Pthread_once_t once_control = PTHREAD_ONCE_INIT;  
int pthread_once (pthread_once_t *once_control,  
void (*init_routine) (void));
```

一些事情仅仅需要做一次，不管是什么。在主函数中并且在调用任何其他依赖于初始化的事物之前初始化应用最容易，特别是在创造任何线程之前初始化它需要的数据，如互斥量、线程特定数据键等。

如果你正在编写一个函数库，通常没有必要做这些，但是必须确保在你能使用需要被初始化的任何东西以前已完成必要的初始化。静态初始化的互斥量能给我们很多帮助，但是有时你会发现“一次性初始化”特性方便得多。

在传统的顺序编程中，一次性初始化经常通过使用布尔变量来管理。控制变量被静态地初始化为 0，而任何依赖于初始化的代码都能测试该变量。如果变量值仍然为 0，则它能实行初始化，然后将变量置为 1。以后检查的代码将跳过初始化。

正在使用多线程时，上述工作就不是那么容易了。如果多个线程并发地执行初始化序列代码，2 个线程可能都发现 `initializer` 为 0，并且都实行初始化，而该过程本该仅仅执行一次。初始化的状态是必须由互斥量保护的共享不变量。

可以使用一个布尔变量和一个静态初始化的互斥量来编写一次性初始化代码。在许多情况中，这比 `pthread_once` 更有效、更方便。使用 `pthread_once` 的主要原因是原来不能静态地初始化一个互斥量。这样，为使用一个互斥量，不得不首先调用 `pthread_mutex_init` 初始化互斥量。必须仅仅初始化互斥量一次，因此初始化调用应在一次性初始化代码中进行。`pthread_once` 解决了这个递归的问题。当互斥量的静态初始化被加到标准中时，`pthread_once` 作为便利功能被保留下来。如果它是方便的，就使用它，但是不必一定使用它。

首先，需要声明类型为 `pthread_once_t` 的一个控制变量。控制变量必须使用 `PTHREAD_ONCE_INIT` 宏静态地初始化，如以下实例 `once.c` 中显示的那样。还必须创建一个包含与控制变量相关联的所有初始化代码的函数。现在，线程可以在任何时间调用 `pthread_once`，指定一个指向控制变量的指针和指向相关初始化函数的指针。

`pthread_once` 函数首先检查控制变量，以判断是否已经完成初始化。如果完成，`pthread_once` 简单地返回；否则，`pthread_once` 调用初始化函数（没有参数），并且记录下初始化被完成。如果在一个线程初始化时，另外的线程调用 `pthread_once`，则调用线程将等待，直到那个线程完成初始化后返回。换句话说，当调用 `pthread_once` 成功返回时，调用者能够肯定所有的状态已经初始化完毕。

13~20

函数 `once_init_routine` 用来初始化互斥量。`pthread_once` 的使用保证它只被调用一次。

29

线程函数 `thread_routine` 在使用互斥量前调用 `pthread_once`，保证就算它没有被主函数创建，它也会存在。

51

主程序也在使用互斥量前调用 `pthread_once`，以便程序无论何时都正确执行。注意，虽然我通常强调所有共享数据必须在创造使用它的任何线程前被初始化，

而在这种情况下，惟一的临界共享数据实际是 once_block，它是不关心互斥量是否被初始化，因为 pthread_once 的使用保证了合适的同步。

■ once.c

```
1 #include <pthread.h>
2 #include "errors.h"
3
4 pthread_once_t once_block = PTHREAD_ONCE_INIT;
5 pthread_mutex_t mutex;
6
7 /*
8  * This is the one-time initialization routine. It will be
9  * called exactly once, no matter how many calls to pthread_once
10 * with the same control structure are made during the course of
11 * the program.
12 */
13 void once_init_routine (void)
14 {
15     int status;
16
17     status = pthread_mutex_init (&mutex, NULL);
18     if (status != 0)
19         err_abort (status, "Init Mutex");
20 }
21
22 /*
23  * Thread start routine that calls pthread_once.
24 */
25 void *thread_routine (void *arg)
26 {
27     int status;
28
29     status = pthread_once (&once_block, once_init_routine);
30     if (status != 0)
31         err_abort (status, "Once init");
32     status = pthread_mutex_lock (&mutex);
33     if (status != 0)
34         err_abort (status, "Lock mutex");
35     printf ("thread_routine has locked the mutex.\n");
36     status = pthread_mutex_unlock (&mutex);
37     if (status != 0)
38         err_abort (status, "Unlock mutex");
39     return NULL;
40 }
41
42 int main (int argc, char *argv[])
43 {
44     pthread_t thread_id;
45     char *input, buffer[64];
46     int status;
47
48     status = pthread_create (&thread_id, NULL, thread_routine, NULL);
```

```

49     if (status != 0)
50         err_abort (status, "Create thread");
51     status = pthread_once (&once_block, once_init_routine);
52     if (status != 0)
53         err_abort (status, "Once init");
54     status = pthread_mutex_lock (&mutex);
55     if (status != 0)
56         err_abort (status, "Lock mutex");
57     printf ("Main has locked the mutex.\n");
58     status = pthread_mutex_unlock (&mutex);
59     if (status != 0)
60         err_abort (status, "Unlock mutex");
61     status = pthread_join (thread_id, NULL);
62     if (status != 0)
63         err_abort (status, "Join thread");
64     return 0;
65 }

```

■ once.c

5.2 属性

*The fifth is ambition. It next will be right
To describe each particular batch:
Distinguishing those that have feathers, and bite,
From those that have whiskers, and scratch.*

—Lewis Carroll, *The Hunting of the Snark*

到目前为止，当我们创建线程或动态初始化互斥量和条件变量时，通常使用空指针作为第二个参数，这个参数实际上是指向一个属性对象的指针。空指针表明，Pthreads 应该为所有属性假定默认值，就像静态初始化互斥量或条件变量时一样。

一个属性对象是当初始化一个对象时提供的一个扩展参数表，它允许主要的接口（例如，`pthread_create`）比较简单，同时当你需要时又提供“高级”的功能。以后的 POSIX 标准将增加选择功能而不必要求改变现有的代码。除了由 Pthreads 提供的标准属性外，具体实现可以提供专业化的选择（不用创建非标准的参数）。

可以将属性对象视为一个私有结构。通过特殊的函数读或写结构的“成员”，而非通过存取公共的成员命名。例如，通过 `pthread_attr_getstacksize` 调用从线程属性对象中读 `stacksize` 属性，或调用 `pthread_attr_setstacksize` 写该属性。

在 Pthreads 的简单实现中，类型 `pthread_attr_t` 可能是一 `typedef` 结构，并且 `get` 和 `set` 函数可能是读或写变量成员的宏。另外的实现可能在初始化属性对象时分配内存，并且它可以将 `get` 和 `set` 函数实现为执行有效性检查的真

正的函数。

线程、互斥量和条件变量都有自己特殊的属性对象类型，分别是 `pthread_attr_t`、`pthread_mutexattr_t` 和 `pthread_condattr_t`。

5.2.1 互斥量属性

```
pthread_mutexattr_t attr;
int pthread_mutexattr_init (pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy (
    pthread_mutexattr_t *attr);
#ifndef _POSIX_THREAD_PROCESS_SHARED
int pthread_mutexattr_getpshared (
    pthread_mutexattr_t *attr, int *pshared);
int pthread_mutexattr_setpshared (
    pthread_mutexattr_t *attr, int pshared);
#endif
```

Pthreads 为互斥量创建定义下列属性：`pshared`、`protocol` 和 `prioceiling`，然而并没有要求系统一定实现这些属性，所以在使用它们前要检查系统文档。

通过调用 `pthread_mutexattr_init` 初始化互斥量属性，指定一个指向类型 `pthread_mutexattr_t` 变量的指针，如以下实例 `mutex_attr.c` 所示。通过把它的地址（而不是我们到目前为止一直在使用的空值）传递给 `pthread_mutex_init` 来使用那个属性对象。

如果系统提供 `_POSIX_THREAD_PROCESS_SHARED` 选项，则它支持 `pshared` 属性，可以调用 `pthread_mutexattr_setpshared` 来设置它。如果将 `pshared` 属性置为 `PTHREAD_PROCESS_SHARED`，就能使用互斥量在多个进程间同步线程，这些进程能访问用来初始化互斥量(`pthread_mutex_t`)的存储器。该属性的默认值是 `PTHREAD_PROCESS_PRIVATE`。

`mutex_attr.c` 程序显示了如何设置属性对象来创建使用 `pshared` 属性的互斥量。这个例子使用默认值 `PTHREAD_PROCESS_PRIVATE`，避免创建共享存储器和派生进程所带来的附加复杂性。另外的互斥量属性、协议和 `prioceiling` 将以后的 5.5.5 节中讨论。

■ mutex_attr.c

```
1 #include <pthread.h>
2 #include "errors.h"
3
4 pthread_mutex_t mutex;
5
6 int main (int argc, char *argv[])
7 {
8     pthread_mutexattr_t mutex_attr;
9     int status;
10}
```

```

11     status = pthread_mutexattr_init (&mutex_attr);
12     if (status != 0)
13         err_abort (status, "Create attr");
14 #ifdef _POSIX_THREAD_PROCESS_SHARED
15     status = pthread_mutexattr_setpshared (
16         &mutex_attr, PTHREAD_PROCESS_PRIVATE);
17     if (status != 0)
18         err_abort (status, "Set pshared");
19 #endif
20     status = pthread_mutex_init (&mutex, &mutex_attr);
21     if (status != 0)
22         err_abort (status, "Init mutex");
23     return 0;
24 }

```

■ mutex_attr.c

5.2.2 条件变量

```

pthread_condattr_t attr;
int pthread_condattr_init (pthread_condattr_t *attr);
int pthread_condattr_destroy (
    pthread_condattr_t *attr);
#endif _POSIX_THREAD_PROCESS_SHARED
int pthread_condattr_getpshared (
    pthread_condattr_t *attr, int *pshared);
int pthread_condattr_setpshared (
    pthread_condattr_t *attr, int pshared);
#endif

```

Pthreads 为条件变量的创建仅定义了一个属性 `pshared`。没有系统被要求实现该属性，这样在使用它之前应检查系统文档。使用 `pthread_condattr_init` 初始化条件变量属性对象，设置一个指向类型 `pthread_condattr_t` 变量的指针，如下例 `cond_attr.c` 所示。通过将其地址传递给 `pthread_cond_init`（而不是我们到目前为止一直在使用的空值）使用该属性对象。

如果系统定义了 `_POSIX_THREAD_PROCESS_SHARED`，则它支持 `pshared` 属性。通过调用 `pthread_condattr_setpshared` 设置 `pshared`。如果将 `pshared` 属性置为 `PTHREAD_PROCESS_SHARED`，条件变量能被多个进程中的线程使用，这些进程能访问初始化条件变量 (`pthread_cond_t`) 的存储器。该属性的默认值是 `PTHREAD_PROCESS_PRIVATE`。

`cond_attr.c` 程序显示了如何设置一个条件变量属性来创建使用 `pshared` 属性的条件变量。这个例子使用默认值 `PTHREAD_PROCESS_PRIVATE`，避免创建共享存储器和派生进程带来的附加复杂性。

■ cond_attr.c

```

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

```

```
3
4     pthread_cond_t cond;
5
6     int main (int argc, char *argv[])
7     {
8         pthread_condattr_t cond_attr;
9         int status;
10
11         status = pthread_condattr_init (&cond_attr);
12         if (status != 0)
13             err_abort (status, "Create attr");
14 #ifdef _POSIX_THREAD_PROCESS_SHARED
15         status = pthread_condattr_setpshared (
16             &cond_attr, PTHREAD_PROCESS_PRIVATE);
17         if (status != 0)
18             err_abort (status, "Set pshared");
19 #endif
20         status = pthread_cond_init (&cond, &cond_attr);
21         if (status != 0)
22             err_abort (status, "Init cond");
23         return 0;
24 }
```

■ cond_attr.c

为使用一个 PTHREAD_PROCESS_SHARED 条件变量，必须使用一个 PTHREAD_PROCESS_SHARED 互斥量，因为同步使用一个条件变量的两个线程必须使用一样的互斥量。等待一个条件变量会自动地解锁、然后加锁相关的互斥量，因此如果互斥量没与 PTHREAD_PROCESS_SHARED 一起被创建，同步不会工作。

5.2.3 线程属性

```
pthread_attr_t attr;
int pthread_attr_init (pthread_attr_t *attr);
int pthread_attr_destroy (pthread_attr_t *attr);
int pthread_attr_getdetachstate (
    pthread_attr_t *attr, int *detachstate);
int pthread_attr_setdetachstate (
    pthread_attr_t *attr, int *detachstate);
#endif _POSIX_THREAD_ATTR_STACKSIZE
int pthread_attr_getstacksize (
    pthread_attr_t *attr, size_t *stacksize);
int pthread_attr_sttstacksize (
    pthread_attr_t *attr, size_t stacksize);
#endif
#endif _POSIX_THREAD_ATTR_STACKADDR
int pthread_attr_getstackaddr (
    pthread_attr_t *attr, void *stackaddr);
int pthread_attr_setstackaddr (
    pthread_attr_t *attr, void **stackaddr);
#endif
```

POSIX 为线程创建定义下列属性：*detachstate*、*stacksize*、*stackaddr*、*scope*、*inheritsched*、*schedpolicy* 和 *schedparam*。一些系统不会支持所有这些属性，因此需要在使用他们前检查系统文档。使用 `pthread_attr_init` 初始化属性对象，指定一个指向 `pthread_attr_t` 类型变量的指针，如程序 `thread_attr.c` 所示。将属性对象地址作为第二参数传递给，（而不是我们到目前为止一直在使用的空值）来使用创建的属性对象。

所有的 Pthreads 系统都支持 *detachstate* 属性，该属性的值可以是 `PTHREAD_CREATE_JOINABLE` 或 `PTHREAD_CREATE_DETACHED`。默认的，线程被创建为可连接的（joinable），即意味着由 `pthread_create` 创建的该线程 ID 能被用来与线程连接并获得它的返回值，或取消它。如果将 *detachstate* 属性设为 `PTHREAD_CREATE_DETACHED`，使用该属性对象创建的线程 ID 不能被使用，也就意味着，当线程终止时，它使用的任何资源将立刻被系统回收。

当创建已经知道不需要取消或连接的线程时，应该以可分离的方式创建它们。记住，在许多情况中，就算你想要知道一个线程什么时候终止，或获得它的返回值，也不必需要使用 `pthread_join`。如果你提供自己的通知机制，如使用一个条件变量，仍然能创建可分离的线程。

| 设置堆栈大小不是可移植的。

如果你的系统定义了标志 `_POSIX_THREAD_ATTR_STACKSIZE`，就可以设置 *stacksize* 属性，指定使用属性对象创建的线程栈的最小值。大多数系统将支持该选项，但是因为栈大小不是可移植的，你应该小心使用它。所需的栈空间数量取决于每个系统使用的调用标准和数据格式。

Pthreads 定义 `PTHREAD_STACK_MIN` 标志，指定每个线程要求的最小栈大小；如果确实需要指定栈大小，能最好以实现要求的最小值计算你的要求。或者，能在实现选择的默认 *stacksize* 属性基础上计算你的要求，如默认值的两倍或缺省值的一半。程序 `thread_attr.c` 显示了如何调用 `pthread_attr_getstacksize` 读取一个初始化的属性的默认 *stacksize* 值。

| 设置堆栈地址将减低可移植性！

如果系统定义了标志 `_POSIX_THREAD_ATTR_STACKADDR`，就可以设置 *stackaddr* 属性，为任何使用该属性对象的线程指定一个存储器区域作为堆栈使用。栈至少一定要和 `PTHREAD_STACK_MIN` 那样大。你可能需要使用符合某个对齐 *granularity* 的地址来指定内存区域。在从上向下的栈空间的机器上，指定的地址应该是栈的最高地址，而不是最低地址。如果栈增大，需要指定最低的地址。

你也需要知道在写新数据之前或之后，机器是否增加（或减少）栈——这决定了你指定的地址是否应该在分配的堆栈“里面”或“外面”。系统不能告诉你是否分配了足够的空间，或是否指定了正确的地址，因此它不得不信任你。如果你错了，

将发生不希望发生的事情。

要特别小心地使用 `stackaddr` 属性，并且注意它可能是 Pthreads 中最不可移植的。尽管合理的 `stacksize` 属性值将可能在大量机器上正常工作，但是很难让任何特别的 `stackaddr` 属性值能在任何两台机器上工作。另外，必须记住你能使用 `stackaddr` 属性的任何值创建一个线程。如果使用相同的 `stackaddr` 属性值创建两个并发线程，线程将在相同的栈上运行（那可不太好）。

以下 `thread_attr.c` 程序演示了实际中的某些属性对象，程序中使用判断条件语句以避免在系统不支持 `stacksize` 属性时使用它。如果 `stacksize` 被支持（并且它将在大多数 UNIX 系统上被支持），程序将打印缺省的和最小的栈大小，并且将 `stacksize` 设为最小值的两倍。代码还创建可分离的线程，即没有线程能连接它们以判断它们何时结束。相反，主函数调用 `pthread_exit` 退出，意味着当最后的线程退出时，进程将终止。

这个例子不包括优先级调度属性，它将在 5.5.2 节中讨论。它也没有演示 `stackaddr` 属性的应用——正如我曾说过的，没有办法以可移植性的方式使用 `stackaddr`，尽管我为完整性考虑提及它，但强烈建议不要在任何程序中使用 `stackaddr`。

■ `thread_attr.c`

```
1 #include <limits.h>
2 #include <pthread.h>
3 #include "errors.h"
4
5 /*
6  * Thread start routine that reports it ran, and then exits.
7  */
8 void *thread_routine (void *arg)
9 {
10     printf ("The thread is here\n");
11     return NULL;
12 }
13
14 int main (int argc, char *argv[])
15 {
16     pthread_t thread_id;
17     pthread_attr_t thread_attr;
18     struct sched_param thread_param;
19     size_t stack_size;
20     int status;
21
22     status = pthread_attr_init (&thread_attr);
23     if (status != 0)
24         err_abort (status, "Create attr");
25
26     /*
27      * Create a detached thread.
28      */
```

```

29     status = pthread_attr_setdetachstate (
30             &thread_attr, PTHREAD_CREATE_DETACHED);
31     if (status != 0)
32         err_abort (status, "Set detach");
33 #ifdef _POSIX_THREAD_ATTR_STACKSIZE
34     /*
35      * If supported, determine the default stack size and report
36      * it, and then select a stack size for the new thread.
37      *
38      * Note that the standard does not specify the default stack
39      * size, and the default value in an attributes object need
40      * not be the size that will actually be used. Solaris 2.5
41      * uses a value of 0 to indicate the default.
42     */
43     status = pthread_attr_getstacksize (&thread_attr, &stack_size);
44     if (status != 0)
45         err_abort (status, "Get stack size");
46     printf ("Default stack size is %u; minimum is %u\n",
47             stack_size, PTHREAD_STACK_MIN);
48     status = pthread_attr_setstacksize (
49             &thread_attr, PTHREAD_STACK_MIN*2);
50     if (status != 0)
51         err_abort (status, "Set stack size");
52 #endif
53     status = pthread_create (
54             &thread_id, &thread_attr, thread_routine, NULL);
55     if (status != 0)
56         err_abort (status, "Create thread");
57     printf ("Main exiting\n");
58     pthread_exit (NULL);
59     return 0;
60 }

```

■ `thread_attr.c`

5.3 取消

```

int pthread_cancel (pthread_t thread);
int pthread_setcancelstate (int state, int *oldstate);
int pthread_setcanceltype (int type, int *oldtype);
void pthread_testcancel (void);
void pthread_cleanup_push (
    void (*routine) (void *), void *arg);
void pthread_cleanup_pop (int execute);

```

大部分时间每个线程独立地运行着，完成一个特定的工作，并且自己退出。但是有时一个线程被创建做不一定需要被完成的一些东西。用户可能单击“取消”

(cancel) 按钮停止长时间的搜索操作。或者，线程可能是某个冗余算法的一部分，并且由于另外线程成功而不再是有用的。当你想要一个线程消失时，你该做什么？这正是 Pthreads 取消接口的任务。

取消一个线程就像告诉一个人停止他正在做的工作一样。有个程序员执拗地着迷于到达陆地，直到到达安全地带也拒绝停止划船（如图 5.1 所示）。当船最后到沙滩上时，他是如此的集中精力以至于没能认识到已经到达陆地。此时，另外的程序员必须想办法使他停止，并且强制地把桨从他的手上移开——他必须被停止。那就是取消。我能在舀水的程序员故事里为取消想到另外的类比，也许你也能。

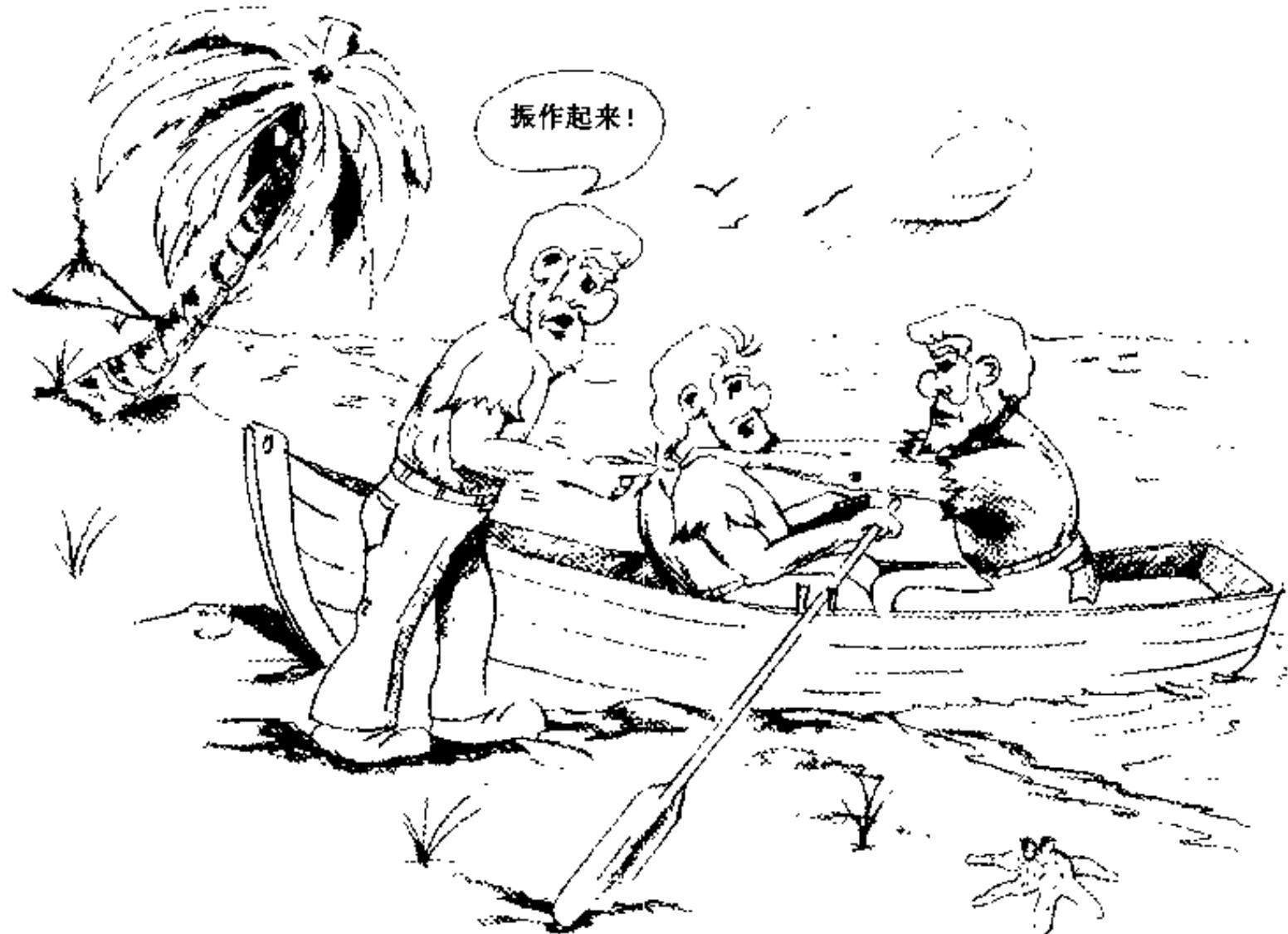


图 5.1 线程取消类比

取消允许你告诉一个线程关掉自己。你不经常需要它，但是有时它是极其有用的。取消不是任意的外部结束操作，它更像一个礼貌的（尽管不一定“友好”）请求。最可能想要取消一个线程的时候是在你发现该线程对完成事情不再是必要时。除非你确实想要目标线程消失，否则应该尽量避免使用取消操作。取消是结束机制，不是通信隧道。那么，为什么想要取消你可能为一些原因创建的线程呢？

当用户继续工作时，应用可能使用线程来执行长操作，也许在后台。这种操作可能包括保存一个大文件、准备打印一个文件或排序一张大表。大多数这样的接口可能需要有一些方法让用户取消操作，不管是单击 ESC 键或 Ctrl-C 键或单击屏幕上的“停止”图标。收到用户取消请求的线程将判断一个或多个后台操作，并且使用 `pthread_cancel` 取消适当的线程。

通常，线程被用来为一些启发式的解决方案并行地“探索”一个数据集。例如，

为获得局部最小值或最大值解方程。一旦你得到了一个足够好的答案，剩下的线程就可以不再需要。如果是这样，你就能取消它们以避免浪费处理器时间，并且开始另外的工作。Pthreads 允许每个线程控制自己的结束，它能恢复程序不变量并解锁互斥量。当线程完成一些重要的操作时，它甚至能推迟取消。例如，当两个写操作必须同时完成时，如果任何一个未完成，则取消它们中的一个操作是不可接受的。

Pthreads 支持三种取消模式，如表 5.1 所示。模式为两位二进制编码，称为“取消状态”和“取消类型”。每种模式实质上包括开、关两种状态（因为编码技术上提供了四种模式，其中一个是冗余的）。如表所示，取消状态可以是“启用”(enable) 或“禁用”(disable)，取消类型可以是被“推迟”或“异步”。

表 5.1 取消状态

模 式	状 态	类 型	含 义
Off (关)	禁用	二者之一	取消 pending 被推迟 直到启用取消模式
Deferred (推迟)	启用	推迟	在下一个取消点执 行取消
Asynchronous (异步)	启用	异步	可以随时执行取消

默认情况下，取消被推迟执行，并且仅仅能在程序中特定的点发生。该点检查线程是否被请求要求终止，被称为取消点。可能等待一无界时间的大多数函数应该成为被推迟取消点。推迟的取消点包括等待条件变量、读或写文件，以及线程可能被阻塞一段可观的时间的其他函数。也有被称为 `pthread_testcancel` 的特殊函数，该函数仅仅是一个推迟的取消点。如果线程没被要求终止，它将很快返回，这允许你将任何函数转变为取消点。

一些系统提供了立刻终止线程的函数。尽管听起来很有用，但要安全地使用该功能是困难的。事实上，它在正常模块化编程环境中是几乎不可能的。例如，如果一个线程尚未解锁互斥量就被终止，则试图加锁该互斥量的下一个线程将被阻塞永远等待。

看上去，系统好像能自动地释放互斥量；但是在大部分时间内，那是没有帮助的。线程锁住互斥量是因为它们正在修改共享的数据。没有另外的线程能知道什么数据被修改了或线程正在试着改变什么，这使恢复数据十分困难。现在，程序被破坏了。当留下锁住的互斥量时，你通常能知道某些东西被破坏了，因为一个以上的线程将被挂起等待互斥量。

从一个锁住互斥量的终止线程中恢复数据的惟一方法是：应用程序能分析所有的共享数据并且恢复它到一个一致、正确的状态。那不是不可能的，并且当应用必须是防失败时，这也值得那些努力。然而，除了应用程序设计者在进程中控制每一

位共享状态的嵌入系统外，这通常是不实际的。你不仅需要重建自己程序或库的状态，而且需要重建可能被线程调用的任何库函数的状态（如 ANSIC 库）。

为取消一个线程，你需要线程的标识符 ID，即由 `pthread_create` 返回给创建者或由 `pthread_self` 返回给线程自己的 `pthread_t` 值。取消一个线程是异步的，即，当 `pthread_cancel` 调用返回时，线程未必已经被取消，可能仅仅被通知有一个针对它的未解决的取消请求。如果需要知道线程在何时实际终止，就必须在取消它之后调用 `pthread_join` 与它连接。

如果线程有一个异步取消类型集，或当线程下次到达一个推迟的取消点时，取消请求将被系统释放。当发生这种情况时，系统将设置线程的取消类型为 `PTHREAD_CANCEL_DELIVEROD`，取消状态为 `PTHREAD_CANCEL_DISABLE`。即，线程能清理并终止自己，而不必担心再次被取消。

当作为一个取消点的函数检测到一个未解决的取消请求时，函数将不返回调用者。如果有任何活跃的清除函数，就调用它们，线程终止。没有方法“处理”取消并继续执行——线程必须或者被彻底推迟取消，或者终止。这好像是 C++ 对象的 `destructors`，而非 C++ 异常——允许对象在运行后销毁自己但是不允许避免销毁。

下列程序 `cancel.c`，显示出通过在一个循环内调用 `pthread_testcancel` 来实现一个对推迟取消反应“相当快速”的线程。

11~19 线程函数 `thread_routine` 无限循环，直到被取消，周期性地测试未解决的取消请求。它每重复 1000 次调用一次（第 17 行），以使调用 `pthread_testcancel` 的开销减到最小。

27~35 在一个 Solaris 系统上，调用 `thr_setconcurrency` 将线程并发级别设置为 2。如果没有调用 `thr_setconcurrency`，程序将挂起。因为 `thread_routine` 是计算密集型函数，不会被阻塞。一旦 `thread_routine` 开始运行，主程序永远不再有机会运行，也无法调用 `pthread_cancel`。

36~54 主程序创造运用 `thread_routine` 的一个线程，睡眠 2 秒，然后取消线程。它与线程连接，并且检查返回值，该值应该是 `PTHREAD_CANCELED`，表明它被取消，而非正常终止。

■ cancel.c

```
1 #include <pthread.h>
2 #include "errors.h"
3
4 static int counter;
5
6 /*
7  * Loop until canceled. The thread can be canceled only
8  * when it calls pthread_testcancel, which it does each 1000
9  * iterations.
10 */
11 void *thread_routine (void *arg)
12 {
```

```

13    DPRINTF (( "thread_routine starting\n" ));
14    for (counter = 0; ; counter++)
15        if ((counter % 1000) == 0) {
16            DPRINTF (( "calling testcancel\n" ));
17            pthread_testcancel ();
18        }
19    }
20
21 int main (int argc, char *argv[])
22 {
23     pthread_t thread_id;
24     void *result;
25     int status;
26
27 #ifdef sun
28     /*
29      * On Solaris 2.5, threads are not timesliced. To ensure
30      * that our two threads can run concurrently, we need to
31      * increase the concurrency level to 2.
32     */
33     DPRINTF (( "Setting concurrency level to 2\n" ));
34     thr_setconcurrency (2);
35 #endif
36     status = pthread_create (
37         &thread_id, NULL, thread_routine, NULL);
38     if (status != 0)
39         err_abort (status, "Create thread");
40     sleep (2);
41
42     DPRINTF (( "calling cancel\n" ));
43     status = pthread_cancel (thread_id);
44     if (status != 0)
45         err_abort (status, "Cancel thread");
46
47     DPRINTF (( "calling join\n" ));
48     status = pthread_join (thread_id, &result);
49     if (status != 0)
50         err_abort (status, "Join thread");
51     if (result == PTHREAD_CANCELED)
52         printf ("Thread canceled at iteration %d\n", counter);
53     else
54         printf ("Thread was not canceled\n");
55     return 0;
56 }

```

■ cancel.c

在需要无中断完成的代码段附近，线程可以调用 `pthread_setcancelstate` 来停用取消。例如，如果数据库更改操作执行两个分开的写调用，你不会希望完成第一个而让第二个取消。如果当取消被停用时，你请求取消一个线程，则线程将记得它被取消但是不会做任何动作，直到取消被重新启用。因为启用取消操作不是一个取消点，如果希望未解决的取消请求很快地被处理，你需要测试未解决的取消请求。

当被取消线程持有私有资源时，如加锁的互斥量或不会被任何其他线程释放的堆存储空间，当线程被取消时，那些资源需要被释放。如果线程有一个加锁的互斥量时，它可能也需要“修理”共享数据以恢复程序的不变量。清除处理函数提供机制完成清除，有点类似进程的 atexit 处理器。在获得一个资源以后，并且在任何取消点以前，通过调用 `pthread_cleanup_push` 声明一个清除处理函数。在释放资源前，但是在任何取消点以后，通过调用 `pthread_cleanup_pop` 删除清除处理函数。

如果没有一个线程的标识符 ID，就不能取消线程。这意味着，至少使用可移植的 POSIX 函数，你不能编写一个可以在进程中任意终止线程的“理想的线程杀手”。你只能取消创建的线程，或线程创建者（或线程自己）给了你一个标识符。这通常意味着取消被限制在一个子系统内操作。

5.3.1 推迟取消

“推迟取消”意味着线程的取消类型被设置为 `PTHREAD_CANCEL_DEFERRED`，线程的取消使能属性被设置为 `PTHREAD_CANCEL_ENABLE`。线程将仅仅在到达取消点时才响应取消请求。

下列函数在任何 Pthreads 系统上总是取消点：

<code>pthread_cond_wait</code>	<code>fsync</code>	<code>sigwaitinfo</code>
<code>pthread_cond_timedwait</code>	<code>mq_receive</code>	<code>sigsuspend</code>
<code>pthread_join</code>	<code>mq_send</code>	<code>sigtimedwait</code>
<code>pthread_testcancel</code>	<code>msync</code>	<code>sleep</code>
<code>sigwait</code>	<code>nanosleep</code>	<code>system</code>
<code>aio_suspend</code>	<code>open</code>	<code>tcdrain</code>
<code>close</code>	<code>pause</code>	<code>wait</code>
<code>creat</code>	<code>read</code>	<code>waitpid</code>
<code>fcntl (F_SETLCKW)</code>	<code>sem_wait</code>	<code>write</code>

以下函数列表可能是取消点。你应该编写代码以便使这些函数是取消点时它将正确工作；如果它们不是取消点，也不会出问题。如果依赖于任何特别的行为，就可能限制了代码的可移植性。你必须查看一致性文档来找出（如果有的话。）哪个是你正在使用的系统的取消点。

<code>closedir</code>	<code>getc_unlocked</code>	<code>printf</code>
<code>ctermid</code>	<code>getchar</code>	<code>putc</code>
<code>fclose</code>	<code>getchar_unlocked</code>	<code>putc_unlocked</code>
<code>fcntl (except F_SETLCKW)</code>	<code>getcwd</code>	<code>putchar</code>
<code>fflush</code>	<code>getrggid</code>	<code>putchar_unlocked</code>
<code>fgetc</code>	<code>getrggid_r</code>	<code>puts</code>
<code>fgets</code>	<code>getrtnam</code>	<code>readdir</code>

fopen	getgrnam_r	remove
fprintf	getlogin	rename
fputc	getlogin_r	rewind
fputs	getpwnam	rewinddir
fread	getpwnam_r	scanf
freopen	getpwuid	tmpfile
fscanf	getpwuid_r	tmpname
fseek	gets	ttyname
ftell	lseek	ttyname_r
fwrite	opendir	ungetc
getc	perror	

Pthreads 要求任何没在上述两张表中指定的 ANSI C 或 POSIX 函数不能是一个取消点。然而，你的系统可能有许多附加的取消点，那是因为很少有 UNIX 系统是“POSIX”。即，它们也支持另外的编程接口如 BSD 4.3、System V Release 4、UNIX95 等。POSIX 认不出如 select 或 poll 函数的存在，因此不能说它们是否是取消点。然而两者显然都可能阻塞任意的时间，所以程序员可以合理地希望它们表现为取消点。通过扩大 Pthreads 取消点列表，X/Open 目前为正在 UNIX98 解决这个问题（X/Open 系统接口，第 5 期）。

大多数取消点包含可以“无限”时间阻塞线程的 I/O 操作，它们是可取消的，以便等待能被打断。当一个线程到达一个取消点时，系统决定是否有一个未解决的针对当前（“目标”）线程的取消请求。如果目标线程从上一个取消点返回后，另外的线程针对目标线程调用了 pthread_cancel，就存在一个未解决的取消。如果这一取消是未解决的，系统将很快开始调用清除函数，然后线程终止。

如果没有取消是当前未解决的，函数将继续执行。如果当线程正在等一些东西（例如 I/O）时，另外的线程请求取消该线程，那么等待将被打断并且线程将开始它的取消清除。

如果需要保证取消不能在一个特别的取消点或取消点的一些顺序期间发生，可以暂时在代码的那个区域停用取消。下列程序 cancel_disable.c 是 cancel.c 的变体。“目标”线程周期性地调用睡眠函数，并且不想调用被取消。

23~32 在每个周期 755 次重复以后，thread_routine 将调用 sleep 等待 1 秒（值 755 只是突然在我头脑中出现的一个任意数字。有任意的数字曾经突然闪现在你的头脑中吗？）。在睡眠以前，thread_routine 通过设置 cancelability 状态为 PTHREAD_CANCEL_DISABLE 停用取消。在睡眠返回后，它再调用 pthread_setcancelstate 恢复保存的 cancelability 状态。

33~35 与 cancel.c 中一样，每重复 1000 次测试未解决的取消。

```
■ cancel_disable.c

1 #include <pthread.h>
2 #include "errors.h"
3
4 static int counter;
5
6 /*
7  * Thread start routine.
8  */
9 void *thread_routine (void *arg)
10 {
11     int state;
12     int status;
13
14     for (counter = 0; ; counter++) {
15
16         /*
17          * Each 755 iterations, disable cancellation and sleep
18          * for one second.
19          *
20          * Each 1000 iterations, test for a pending cancel by
21          * calling pthread_cancel().
22          */
23
24         if ((counter % 755) == 0) {
25             status = pthread_setcancelstate (
26                 PTHREAD_CANCEL_DISABLE, &state);
27             if (status != 0)
28                 err_abort (status, "Disable cancel");
29             sleep (1);
30             status = pthread_setcancelstate (
31                 state, &state);
32             if (status != 0)
33                 err_abort (status, "Restore cancel");
34         } else
35             if ((counter % 1000) == 0)
36                 pthread_cancel ();
37     }
38
39 int main (int argc, char *argv[])
40 {
41     pthread_t thread_id;
42     void *result;
43     int status;
44
45     status = pthread_create (
46         &thread_id, NULL, thread_routine, NULL);
47     if (status != 0)
48         err_abort (status, "Create thread");
49     sleep (2);
50     status = pthread_cancel (thread_id);
51     if (status != 0)
52         err_abort (status, "Cancel thread");
```

```

53
54     status = pthread_join (thread_id, &result);
55     if (status != 0)
56         err_abort (status, "Join thread");
57     if (result == PTHREAD_CANCELED)
58         printf ("Thread canceled at iteration %d\n", counter);
59     else
60         printf ("Thread was not canceled\n");
61     return 0;
62 }
```

■ cancel_disable.c

5.3.2 异步取消

异步取消是有用的，因为“目标线程”不需要使用取消点来查询取消请求。对于运行一个紧密计算循环的线程（例如，在找一个素数因素）而言是珍贵的，因为那种情况下调用 `pthread_testcancel` 的开销在可能是严重的。

避免异步的取消！

很难正确使用异步取消，并且很少有用。

问题是：你能在异步取消过程中做的事是有限的。你不能获得任何资源，如加锁互斥量，因为清除代码没有办法决定互斥量是否被锁住了。异步的取消能用任何硬件指令发生，在一些计算机上甚至可能在一些指令的中间打断它们，这使决定取消的线程正在做什么变地困难。

例如，当你调用 `malloc` 时，系统为你分配一些堆内存，将指向分配内存的指针存储到某地（可能在硬件寄存器中），然后返回给的代码，代码可能为以后的使用将返回值保存到本地变量。`malloc` 可能在很多地方被异步取消打断，不同的地方决定了变化的效果。它可能在分配内存前被打断，或可能在分配内存后、但在保存地址返回前被打断，或者甚至可能返回代码，但是在返回值被拷贝到本地变量前被打断。无论哪种情况，你的代码中保存指向分配内存指针的变量将是未初始化的。你不能判断内存是否确实被分配，不能释放内存，以至于内存（如果它被分配给你）将在程序整个生命周期内保持被分配状态。这就是内存泄漏，一个我们不想看到的特征。

或者当你调用 `pthread_mutex_lock` 时，系统可能在函数调用内被打断（在锁住互斥量之前或之后）。同样，程序没有办法为你发现，因为中断可能在任何两条指令之间发生，甚至在 `pthread_mutex_lock` 函数内部发生，它可能让互斥量不可用。如果互斥量被锁住，因为它决不会被解锁，应用程序将多半会挂起在上面结束。

除非你写了可以安全地异步取消的代码，否则在异步取消启用时不要调用任何代码。即使要这样做，也要三思！

当异步取消被启用时，你不允许调用任何获得资源的函数。事实上，除非当函数被记录为“异步取消安全”的，否则当异步取消被启用时你不该调用任何函数。由 Pthreads 规定的异步取消安全的函数是 `pthread_cancel`、`pthread_setcancelstate` 和 `pthread_setcanceltype`（并且当启用异步取消时，没有必要调用 `pthread_cancel`）。没有其他 POSIX 或 ANSIC 函数需要是异步取消安全的，并且你从来不该在启用异步取消时调用它们。

Pthreads 建议所有的库函数应该记录它们是否是异步取消安全的。如果函数的描述没有具体的说明，则你应该总是假定它不是异步取消安全的。在不是异步取消安全的函数中执行异步取消的后果将是严重的。而且更坏的是，后果是与时间相关的。即，事实上，在试验期间好像是异步取消安全的函数，在一个稍微不同的地方被取消时，可能就会引起各种问题。

下列程序 `cancel_async.c`，显示了在一个计算密集的循环中异步取消的使用。异步取消的使用使该循环比在 `cancel.c` 中使用推迟取消的循环“响应性更好”。然而，如果在循环内有任何函数调用，程序将变得不可靠，而推迟取消的版本将能继续正确工作。在大多数情况下，同步取消更好。

24~28 为了能让线程以比一个空循环更有趣的方式运行片刻，`cancel_async.c` 使用了一个简单的矩阵嵌套乘循环。`matrixa` 和 `matrixb` 数组被分别用它们的主要或次要数组索引初始化。

34~36 取消类型被改变为 `PTHREAD_CANCEL_ASYNCHRONOUS`，以使在矩阵乘循环内部允许异步取消。

39~44 线程重复矩阵乘直到被取消，在每次重复中用先前增加的结果（`matrixc`）代替第一个源数组（`matrixa`）。

66~74 同样，在 Solaris 系统上将线程并发级别设为 2，以允许主线程和 `thread_routine` 并发地在一台单处理机上运行。没有这个步骤，程序将挂起，因为 Solaris 上的用户模式线程是不被时间片控制的。

■ `cancel_async.c`

```
1 #include <pthread.h>
2 #include "errors.h"
3
4 #define SIZE    10      /* array size */
5
6 static int matrixa[SIZE][SIZE];
7 static int matrixb[SIZE][SIZE];
8 static int matrixc[SIZE][SIZE];
9
10 /*
11  * Loop until canceled. The thread can be canceled at any
12  * point within the inner loop, where asynchronous cancellation
13  * is enabled. The loop multiplies the two matrices matrixa
14  * and matrixb.
15 */
```

```
16 void *thread_routine (void *arg)
17 {
18     int cancel_type, status;
19     int i, j, k, value = 1;
20
21     /*
22      * Initialize the matrices to something arbitrary.
23      */
24     for (i = 0; i < SIZE; i++)
25         for (j = 0; j < SIZE; j++) {
26             matrixa[i][j] = i;
27             matrixb[i][j] = j;
28         }
29
30     while (1) {
31         /*
32          * Compute the matrix product of matrixa and matrixb.
33          */
34         status = pthread_setcanceltype (
35             PTHREAD_CANCEL_ASYNCHRONOUS,
36             &cancel_type);
37         if (status != 0)
38             err_abort (status, "Set cancel type");
39         for (i = 0; i < SIZE; i++)
40             for (j = 0; j < SIZE; j++) {
41                 matrixc[i][j] = 0;
42                 for (k = 0; k < SIZE; k++)
43                     matrixc[i][j] += matrixa[i][k] * matrixb[k][j];
44             }
45         status = pthread_setcanceltype (
46             cancel_type,
47             &cancel_type);
48         if (status != 0)
49             err_abort (status, "Set cancel type");
50
51         /*
52          * Copy the result (matrixc) into matrixa to start again
53          */
54         for (i = 0; i < SIZE; i++)
55             for (j = 0; j < SIZE; j++)
56                 matrixa[i][j] = matrixc[i][j];
57     }
58 }
59
60 int main (int argc, char *argv[])
61 {
62     pthread_t thread_id;
63     void *result;
64     int status;
65
66 #ifdef sun
67     /*
68      * On Solaris 2.5, threads are not timesliced. To ensure
```

```
69      * that our two threads can run concurrently, we need to
70      * increase the concurrency level to 2.
71      */
72      DPRINTF (( "Setting concurrency level to 2\n" ));
73      thr_setconcurrency ( 2 );
74 #endif
75      status = pthread_create (
76          &thread_id, NULL, thread_routine, NULL );
77      if (status != 0)
78          err_abort (status, "Create thread");
79      sleep (1);
80      status = pthread_cancel (thread_id);
81      if (status != 0)
82          err_abort (status, "Cancel thread");
83      status = pthread_join (thread_id, &result);
84      if (status != 0)
85          err_abort (status, "Join thread");
86      if (result == PTHREAD_CANCELED)
87          printf ("Thread canceled\n");
88      else
89          printf ("Thread was not canceled\n");
90      return 0;
91 }
```

■ cancel_async.c

警告:

别将“DCE 线程”的习惯带到 Pthreads 上！

我将以警告结束本节。DCE 线程是开放软件基础（OSF）的分布式计算环境（DCE）的一个关键部件，被设计为独立于内在的 UNIX 核心。根本不支持线程的系统经常在用户模式上使用非阻塞的 I/O 模式仿效“同步的线程”I/O，以便使在一个忙文件上尝试 I/O 的线程在一个条件变量上阻塞，直到以后的 select 或 poll 操作显示 I/O 可以完成。DCE 监听线程可能无止境地在一个 socket 读操作上阻塞，所以能取消这种读操作是重要的。

当 DCE 被移植到支持线程（然而并非支持 Pthreads）的新内核时，通常省略用户模式 I/O 包装器，导致线程在不支持推迟取消的内核内阻塞。在许多情况下，用户发现这些系统是采用以下的方法实现了异步取消：相当巧合地，如果在内核调用前线程很快地切换到异步取消模式，而在内核调用以后又很快地切换回推迟取消模式，一个内核等待就可以“安全地”被取消。这一观察结果也被发布在 DCE 文档上，但这是很危险的，即使在它似乎可以正常工作的系统上。不应该在任何 Pthreads 系统上试用这种方式！如果你的系统遵循 POSIX 1003.1c-1995（或 POSIX 1003.1、1996 年版，或以后），它至少支持如读、写等内核功能上的推迟取消。你不需要异步取消，并且使用它可能极其危险。

5.3.3 清除

当编写任何库代码时，应将其设计为可以优雅地处理推迟取消。在不适当的地方停用取消，并且总是在取消点使用清除处理器。

当一个代码段被取消时，需要恢复一些状态，必须使用清除处理器。当线程在等待一个条件变量时被取消，它将被唤醒，并保持互斥量加锁状态。在线程终止前，通常需要恢复不变量，且它总是需要释放互斥量。

可以把每个线程考虑为有一个活动的清除处理函数的栈。调用 `pthread_cleanup_push` 将清除处理函数加到栈中，调用 `pthread_cleanup_pop` 删除最近增加的处理函数。当线程被取消时或当它调用 `pthread_exit` 退出时，Pthreads 从最近增加的清除处理函数开始，依次调用各个活动的清除处理函数。当所有活动的清除处理函数返回时，线程被终止。

Pthreads 清除处理函数甚至被设计为当线程没被取消时，也能经常使用清除处理函数。不论是否被取消或正常完成，运行同样的清除函数经常是有用的。当 `pthread_cleanup_pop` 以非零值被调用时，就算线程没被取消，清除处理函数也要被执行。

你不能在一个函数内压入一个清除处理函数而在另外的函数中弹出它。`pthread_cleanup_push` 和 `pthread_cleanup_pop` 操作可能作为宏被定义，这样 `pthread_cleanup_push` 中可能包含块开始大括号 “{”，而 `pthread_cleanup_pop` 中则包含了匹配的块结束大括号 “}”。当使用清除处理函数时，如果希望代码可移植，你必须总是记住这一限制。

下列程序 `cancel_cleanup.c`，显示了当一个条件变量等待被取消时，使用一个清除处理函数来释放互斥量。

10~17 控制结构 (`control`) 被所有的线程使用来维护共享同步对象和不变量。每个线程在开始时将成员 `counter` 加 1，并且在结束时将 `counter` 减 1。成员 `busy` 作为一个哑元条件等待谓语，被初始化为 1，并且从来不被清零，即意味着条件等待循环决不会终止（在这个例子中）直到线程被取消。

24~34 函数 `cleanup_handler` 作为每个线程的取消清除处理函数被安装。当线程被取消或正常结束时，该函数被调用，减少活动的线程数并且解锁互斥量。

47 函数 `thread_routine` 建立 `cleanup_handler` 作为活动的取消清除处理函数。

54~58 等待直到 `control` 结构的 `busy` 成员被置为 0，这在这个例子中决不会发生。条件等待循环只有当等待被取消时才能退出。

63 尽管条件等待循环不会退出，函数通过删除活动的清除处理函数执行清理工作。记住，只要设置 `pthread_cleanup_pop` 的参数非零，则即使没有发生取消，清除处理函数仍将被调用。

在一些情况中，你可以省略像这个 `pthread_cleanup_pop` 调用一样“不能到达的语句”。然而在本例中，没有它你的代码可能就不能编译通过。`pthread_cleanup_push` 和

`pthread_cleanup_pop` 宏是特殊的，并且分别可以扩展成块的开始和结束。例如，Digital UNIX 在由操作系统提供的结构化异常处理上面以这种方式实现了取消。

```
■ cancel_cleanup.c
1 #include <pthread.h>
2 #include "errors.h"
3
4 #define THREADS 5
5
6 /*
7  * Control structure shared by the test threads, containing
8  * the synchronization and invariant data.
9  */
10 typedef struct control_tag {
11     int             counter, busy;
12     pthread_mutex_t mutex;
13     pthread_cond_t  cv;
14 } control_t;
15
16 control_t control =
17     {0, 1, PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER};
18
19 /*
20  * This routine is installed as the cancellation cleanup
21  * handler around the cancelable condition wait. It will
22  * be called by the system when the thread is canceled.
23  */
24 void cleanup_handler (void *arg)
25 {
26     control_t *st = (control_t *)arg;
27     int status;
28
29     st->counter--;
30     printf ("cleanup_handler: counter == %d\n", st->counter);
31     status = pthread_mutex_unlock (&st->mutex);
32     if (status != 0)
33         err_abort (status, "Unlock in cleanup handler");
34 }
35
36 /*
37  * Multiple threads are created running this routine (controlled
38  * by the THREADS macro). They maintain a "counter" invariant,
39  * which expresses the number of running threads. They specify a
40  * nonzero value to pthread_cleanup_pop to run the same
41  * "finalization" action when cancellation does not occur.
42  */
43 void *thread_routine (void *arg)
```

```
44 {
45     int status;
46
47     pthread_cleanup_push (cleanup_handler, (void*)&control);
48
49     status = pthread_mutex_lock (&control.mutex);
50     if (status != 0)
51         err_abort (status, "Mutex lock");
52     control.counter++;
53
54     while (control.busy) {
55         status = pthread_cond_wait (&control.cv, &control.mutex);
56         if (status != 0)
57             err_abort (status, "Wait on condition");
58     }
59
60     pthread_cleanup_pop (1);
61     return NULL;
62 }
63
64 int main (int argc, char *argv[])
65 {
66     pthread_t thread_id[THREADS];
67     int count;
68     void *result;
69     int status;
70
71     for (count = 0; count < THREADS; count++) {
72         status = pthread_create (
73             &thread_id[count], NULL, thread_routine, NULL);
74         if (status != 0)
75             err_abort (status, "Create thread");
76     }
77
78     sleep (2);
79
80     for (count = 0; count < THREADS; count++) {
81         status = pthread_cancel (thread_id[count]);
82         if (status != 0)
83             err_abort (status, "Cancel thread");
84
85         status = pthread_join (thread_id[count], &result);
86         if (status != 0)
87             err_abort (status, "Join thread");
88         if (result == PTHREAD_CANCELED)
89             printf ("thread %d canceled\n", count);
90         else
91             printf ("thread %d was not canceled\n", count);
92     }
93     return 0;
94 }
```

■ cancel_cleanup.c

如果你的一个线程创建了一套线程来“转包”一些功能（如并行算术运算），并且当分包线程在进行中时“承包线程”被取消，你可能不希望留着分包线程继续

运行。相反，可以把取消操作“传递”到每个“分包线程”，让它们独立地处理自己的终止过程。

如果你原来打算连接“分包线程”，记得它们将继续消费一些资源直到它们被连接或分离。当“承包线程”取消它们时，不应该连接分包线程来推迟取消；相反，可以取消每个线程并且使用 `pthread_detach` 很快地分离它。当它们完成时，分包线程的资源就能够很快被重用，而“承包线程”同时能独立地完成一些事情。

以下程序 `cancel_subcontract.c` 显示了传播取消到“分包线程”的一个方法。

9~12 `team_t` 结构定义了分包线程团队的状态。`join_i` 成员记录了与承包线程连接的最后一个分包线程的索引，这样从在 `pthread_join` 内部取消时，它能取消它还没连接的线程。`workers` 成员是记录分包线程标识符 ID 的一个数组。

18~25 分包线程从 `worker_routine` 函数开始运行。该函数一直循环直到被取消，每重复 1000 次调用一次 `pthread_testcancel`。

31~46 清除函数在承包线程内作为活动的清除处理函数被建立。当承包线程被取消时，依次清除留下（未连接）的分包线程，取消并且分离它们。注意它不连接分包线程——通常在一个清除处理函数内等待不是一个好主意。不管怎么说，该线程是期望来清理并终止的，而不是等待一些东西发生的。但是如果您的清除处理函数确实需要等待一些东西，别害怕，它将会工作得很好。

54~76 承包线程从 `thread_routine` 开始运行。该函数创建一套分包线程，然后与每个分包线程连接。当连接每个线程时，它在 `team_t` 成员 `join_i` 中记录下 `workers` 数组中的当前索引。清除处理函数有一指向 `team` 结构的指针，以便它能决定最后的偏移量并且开始取消留下的分包线程。

78~104 主程序创建承包线程，运行 `thread_routine`，然后睡眠 5 秒。当它醒来时，取消承包线程并等它终止。

■ `cancel_subcontract.c`

```
1 #include <pthread.h>
2 #include "errors.h"
3
4 #define THREADS 5
5
6 /*
7  * Structure that defines the threads in a "team."
8 */
9 typedef struct team_tag {
10     int          join_i;           /* join index */
11     pthread_t    workers[THREADS]; /* thread identifiers */
12 } team_t;
13
14 /*
15  * Start routine for worker threads. They loop waiting for a
16  * cancellation request.
```

```
17  */
18 void *worker_routine (void *arg)
19 {
20     int counter;
21
22     for (counter = 0; ; counter++)
23         if ((counter % 1000) == 0)
24             pthread_testcancel ();
25 }
26
27 /*
28  * Cancellation cleanup handler for the contractor thread. It
29  * will cancel and detach each worker in the team.
30 */
31 void cleanup (void *arg)
32 {
33     team_t *team = (team_t *)arg;
34     int count, status;
35
36     for (count = team->join_i; count < THREADS; count++) {
37         status = pthread_cancel (team->workers[count]);
38         if (status != 0)
39             err_abort (status, "Cancel worker");
40
41         status = pthread_detach (team->workers[count]);
42         if (status != 0)
43             err_abort (status, "Detach worker");
44         printf ("Cleanup: canceled %d\n", count);
45     }
46 }
47
48 /*
49  * Thread start routine for the contractor. It creates a team of
50  * worker threads, and then joins with them. When canceled, the
51  * cleanup handler will cancel and detach the remaining threads.
52 */
53 void *thread_routine (void *arg)
54 {
55     team_t team;                      /* team info */
56     int count;
57     void *result;                     /* Return status */
58     int status;
59
60     for (count = 0; count < THREADS; count++) {
61         status = pthread_create (
62             &team.workers[count], NULL, worker_routine, NULL);
63         if (status != 0)
64             err_abort (status, "Create worker");
65     }
66     pthread_cleanup_push (cleanup, (void*)&team);
67
68     for (team.join_i = 0; team.join_i < THREADS; team.join_i++) {
69         status = pthread_join (team.workers[team.join_i], &result);
70         if (status != 0)
```

```
71             err_abort (status, "Join worker");
72     }
73
74     pthread_cleanup_pop (0);
75     return NULL;
76 }
77
78 int main (int argc, char *argv[])
79 {
80     pthread_t thread_id;
81     int status;
82
83 #ifdef sun
84     /*
85      * On Solaris 2.5, threads are not timesliced. To ensure
86      * that our threads can run concurrently, we need to
87      * increase the concurrency level to at least 2 plus THREADS
88      * (the number of workers).
89     */
90     DPRINTF (("Setting concurrency level to %d\n", THREADS+2));
91     thr_setconcurrency (THREADS+2);
92 #endif
93     status = pthread_create (&thread_id, NULL, thread_routine, NULL);
94     if (status != 0)
95         err_abort (status, "Create team");
96     sleep (5);
97     printf ("Cancelling...\n");
98     status = pthread_cancel (thread_id);
99     if (status != 0)
100         err_abort (status, "Cancel team");
101     status = pthread_join (thread_id, NULL);
102     if (status != 0)
103         err_abort (status, "Join team");
104     return 0;
105 }
```

■ cancel_subcontract.c

5.4 线程私有数据

No, I've made up my mind about it: if I'm Mabel, I'll stay down here. It'll be no use their putting their heads down and saying "Come up again, dear!" I shall only look up and say "Who am I, then? Tell me that first, and then, if I like being that person, I'll come up: if not, I'll stay down here till I'm somebody else."

—Lewis Carroll, Alice's Adventures in Wonderland

当单线程程序中的一个函数需要创造私有数据时，该私有数据在对该函数的调用之间保持一致，数据能静态地被分配在存储器中。命名范围可能限制于使用它的函数或文件（静态）或者它能被全局使用（extern）。

当你使用线程时它就不是那么简单了。在进程内的所有线程共享相同的地址空间，即意味着任何声明为静态或外部的变量，或在进程堆声明的变量，都可以被进程内所有的线程读写。要在一系列函数调用之间存储“稳定不变”的数据对于代码来说有若干重要的含意：

- 静态变量（static）、外部变量（extern）或堆变量的值，将是上次任何线程改写的值。在一些情况中这可能是你想要的，如维护伪随机数字顺序的种子。在其他情况下，它可能就不是你想要的。
- 一个线程真正拥有的惟一私有存储是处理器寄存器。甚至栈地址也能被共享，尽管只有在“主人”故意暴露一个地址给另外的线程时。在任何情况下，寄存器和“私有”堆栈都不能代替非线程代码中使用的持久静态存储。

因此，当需要一个私有变量时，必须首先决定所有的线程是否共享相同的值，或者线程是否应该有它自己的值。如果它们共享变量，则可以使用静态或外部数据，就像你能在单线程程序做的那样；然而，必须同步跨越多线程对共享数据的存取，通常通过增加一个或多个互斥量来完成。

如果每个线程都需要一个私有变量值，则必须在某处存储所有的值，并且每个线程一定能定位合适的值。在一些情况中可能能够使用静态数据，例如，能在一张表格中为每个线程查找惟一的值，如线程的 `pthread_t`。在许多有趣的情况下，你不能预言有多少线程可能调用这个函数——想像一下你正在实现能被任意多个线程中的任意代码调用的线程安全的函数库。

最一般的解决方案是在每个线程的堆栈中分配内存并将值保存在那里，但是你的代码将需要能够在任何线程中发现合适的私有数据。你能创建所有私有数值的一张连接表，存储创建线程的标识符（`pthread_t`）以使它能被再次发现，但是如果有许多线程，那将是很慢的。你需要在表中寻找合适的值，并且要收回终止线程分配的存储将是困难的。你的函数不会知道一个线程什么时候终止。

| 新接口不应该依靠隐式的持久存储！

正在设计新接口时，有一个更好的解决方案。你应该要求调用者分配必要的持久状态，并且告诉你它在哪儿。这个模型有许多优点，最重要地包括以下两点。

- 在许多情况中，使用这个模型能避免内部同步，并且在调用者希望在线程之间共享持久状态的稀罕情况中，能提供需要的同步。
- 相反，调用者可以选择分配多个在单线程内使用的状态缓冲区，结果是若干独立的相同线程内的函数调用序列之间没有冲突。

问题是经常需要支持隐式的持久状态。你可能正在将一个现存接口改为线程安全的，并且不能在函数中增加一个参数，或者要求调用者因为你的需求维护一个新的数据结构（这就是线程私有数据存在的地方）。

线程私有数据允许每个线程保有一份变量的拷贝，好像每个线程有一连串通过

公共的“键”值索引的线程私有数据值、想像舀水的程序员正穿着标有他们社团的标志徽章的衬衫（图 5.2）。尽管每个程序员的信息不同，你能很容易地发现信息而不需要知道你在检验哪个程序员。

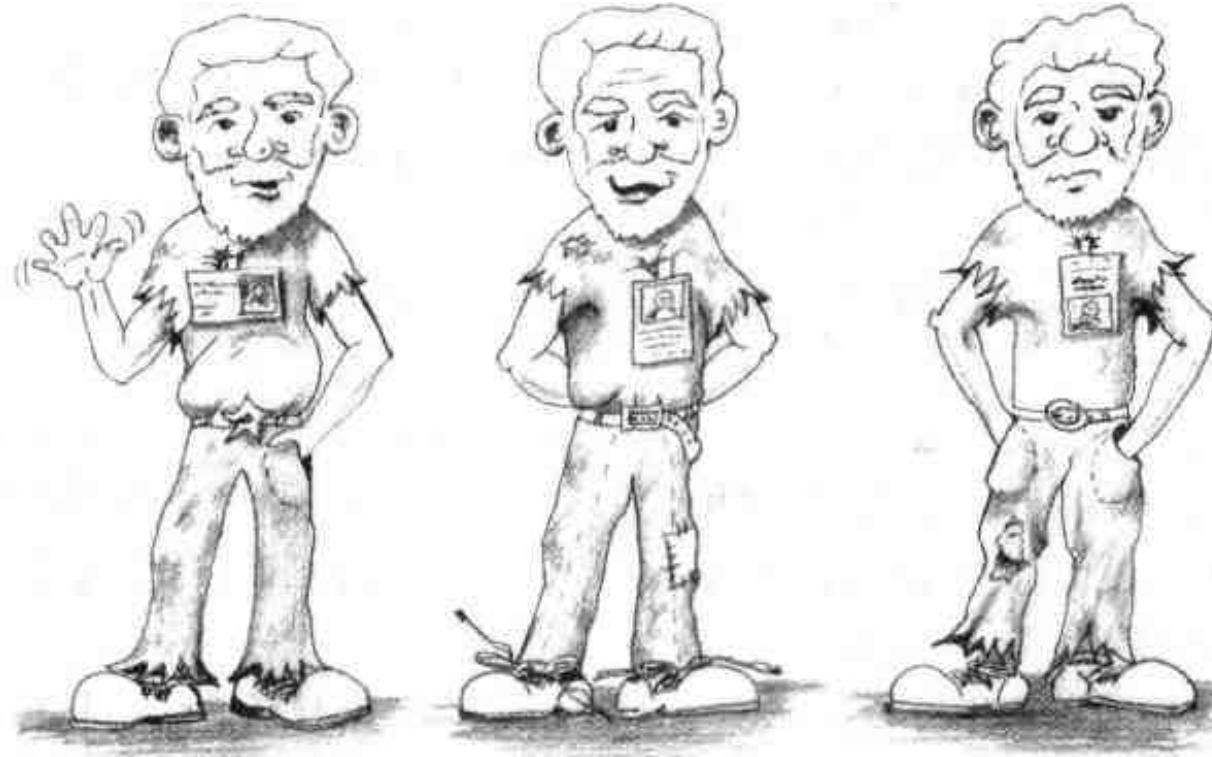


图 5.2 线程特定数据分析

程序创建一个键（就像设定一条社团的规则，徽章总是被戴在雇员衬衫或夹克衫的左衣袋上），然后每个线程就能独立地设定或得到自己的键值（尽管徽章总是在左边的衣袋上，每个雇员有唯一的徽章编号，和在大多数情况中一个唯一的名字）。键对于所有的线程是相同的，但是每个线程能将它独立的键值与共享键联系。每个线程能在任何时间为键改变它的私有值，而不会影响键或任何另外线程拥有的键值。

5.4.1 建立线程私有数据

```
pthread_key_t key;
int pthread_key_create (
    pthread_key *key, void (*destructor)(void *));
int pthread_key_delete (pthread_key_t key);
```

线程私有数据键在程序中是由类型 `pthread_key_t` 的一个变量表示的。像大多数 Pthreads 类型一样，`pthread_key_t` 是不透明的并且你永远不要做任何关于其结构或内容的假设。在任何线程试图使用键以前，创建线程私有数据键最容易的方法是调用 `pthread_key_create`，例如在程序的主函数中。

如果你以后需要创建一个线程私有数据键，必须保证 `pthread_key_create` 对于每个 `pthread_key_t` 变量仅仅被调用一次。因为，如果将一个键创建两次，其实是在创建两个不同的键。第二个键将覆盖第一个，第一个键与任何线程可能为其设置的值一起将永远地丢失。

当你不能把代码加到 main 函数时，保证一个线程私有数据键仅被创建一次的最容易的方法是使用 `pthread_once`，即一次性的初始化函数，如下列程序 `tsd_once.c` 所示。

7~10 `tsd_t` 结构用于包含每个线程的数据。每个线程分配私有的 `tsd_t` 结构，并且将一指向该结构的指针作为其私有数据键 `tsd_key` 的值保存。`thread_id` 成员保持线程的标识符 (`pthread_t`)，`string` 成员保持一个指向线程名字字符串的指针。`tsd_key` 变量保存线程私有数据键，该键用来存取 `tsd_t` 结构。

19~27 ---次性的初始化函数 (`pthread_once`) 被用来保证 `tsd_key` 键在第一次存取前被创建。

33~56 线程从启动函数 `thread_routine` 开始运行。参数 `arg` 是指向线程名字字符串的一个指针。每个线程调用 `pthread_once` 保证线程私有数据键被创建。线程然后分配 `tsd_t` 结构，用线程的标识符初始化 `thread_id` 成员，并且拷贝它的参数到 `string` 成员。

线程通过调用 `pthread_getspecific` 得到当前线程私有数据值，并且使用线程名字打印一条消息。它然后睡眠几秒，并且再次打印一条消息以表明线程私有数据值保持一样，尽管另外的线程把不同的 `tsd_t` 结构地址分派给了相同的线程私有数据键。

■ tsd_once.c

```

1 #include <pthread.h>
2 #include "errors.h"
3
4 /*
5  * Structure used as the value for thread-specific data key.
6  */
7 typedef struct tsd_tag {
8     pthread_t    thread_id;
9     char        *string;
10 } tsd_t;
11
12 pthread_key_t tsd_key;           /* Thread-specific data key */
13 pthread_once_t key_once = PTHREAD_ONCE_INIT;
14
15 /*
16  * One-time initialization routine used with the pthread_once
17  * control block.
18 */
19 void once_routine (void)
20 {
21     int status;
22
23     printf ("initializing key\n");
24     status = pthread_key_create (&tsd_key, NULL);
25     if (status != 0)
26         err_abort (status, "Create key");
27 }
```

```
28
29  /*
30   * Thread start routine that uses pthread_once to dynamically
31   * create a thread-specific data key.
32  */
33 void *thread_routine (void *arg)
34 {
35     tsd_t *value;
36     int status;
37
38     status = pthread_once (&key_once, once_routine);
39     if (status != 0)
40         err_abort (status, "Once init");
41     value = (tsd_t*)malloc (sizeof (tsd_t));
42     if (value == NULL)
43         errno_abort ("Allocate key value");
44     status = pthread_setspecific (tsd_key, value);
45     if (status != 0)
46         err_abort (status, "Set tsd");
47     printf ("%s set tsd value %p\n", arg, value);
48     value->thread_id = pthread_self ();
49     value->string = (char*)arg;
50     value = (tsd_t*)pthread_getspecific (tsd_key);
51     printf ("%s starting...\n", value->string);
52     sleep (2);
53     value = (tsd_t*)pthread_getspecific (tsd_key);
54     printf ("%s done...\n", value->string);
55     return NULL;
56 }
57
58 void main (int argc, char *argv[])
59 {
60     pthread_t thread1, thread2;
61     int status;
62
63     status = pthread_create (
64         &thread1, NULL, thread_routine, "thread 1");
65     if (status != 0)
66         err_abort (status, "Create thread 1");
67     status = pthread_create (
68         &thread2, NULL, thread_routine, "thread 2");
69     if (status != 0)
70         err_abort (status, "Create thread 2");
71     pthread_exit (NULL);
72 }
```

■ tsd_once.c

当程序不再需要时, Pthreads 允许你调用 `pthread_key_delete` 释放一个线程私有数据键。Pthreads 标准保证在一次只能有 128 个线程私有数据键。因此, 释放你不再需要的键是有用的。你的 Pthreads 系统实际支持的键数目由`<limits.h>` 中定义的 `PTHREAD_KEYS_MAX` 标志的值指定。

当释放线程私有数据键时, 不会影响任何线程对该键设置的当前值, 甚至不影

响调用线程的当前键值。那意味着你的代码将完全负责在所有的线程内释放与线程私有数据键相关的任何内存。当然，任何使用已删除的线程私有数据键（`pthread_key_t`）将导致未定义的行为。

| 仅当你肯定没有线程持有该键的值时，才能删除线程私有数据键！
| 否则，根本别释放它们。

当一些线程仍然持有某键的值时，你就不该释放该键。例如，如果在随后调用 `pthread_key_create`，可能重用被已删除键使用过的 `pthread_key_t` 标识符。当曾经为旧键赋值的现存线程请求新键的值时，它将会收到旧值。程序多半将对收到的不正确的数据反应糟糕，因此你决不应该删除线程私有数据键直到你肯定没有线程持有该键值为止。例如，通过为该键维护一个“引用数”，如 5.4.3 节程序 `tsd_destructor.c` 所示。

甚至更好的作法是不释放线程私有数据键。很少需要这样做，并且如果你试图这样（译者注：释放键），你将几乎陷入困境。很少有程序要求 128 个线程私有数据键（Pthreads 最小限制），几乎没有程序需要更多。总的来说，使用线程私有数据的每个部件将使用少数几个键来维护指向包含相关数据结构的指针。用尽所有的键将需要很多部件！

5.4.2 使用线程私有数据

```
int pthread_setspecific (
    pthread_key_t key, const void *value);
void *pthread_getspecific (pthread_key_t key);
```

可以使用 `pthread_getspecific` 函数来获得线程当前的键值，或调用 `pthread_setspecific` 来改变当前的键值。7.3.1 节给出了使用线程私有数据将依靠静态数据的旧库改造为线程安全的方法。

| 线程私有数据值为空（NULL）对于 Pthreads 意味着一些特殊的东西，所以
| 除非你确实需要，不要将一个线程私有数据置空。

对于任何新键，其初始值（在所有的线程）为 NULL。另外，当一个线程终止时^①，Pthreads 在调用键的 `destructor` 函数前将对应的线程私有数据置为 NULL（传

^① 不幸的是，标准不是这样说的。这是标准问题之一——他们说是说了，但不是他们原本想说的。不论如何，错误发生了，这句话（“调用 `destructor` 函数之前应清除线程私有数据值”）被漏掉了。没有人注意到这一点，标准被批准通过。所以，标准上讲“如果你想使用线程私有数据写一个可移植程序，且该程序在线程终止后不被挂起，你必须在 `destructor` 函数中调用 `pthread_setspecific` 来将私有数据值置空。”这将是愚蠢的，任何严谨的 Pthreads 实现都会在这方面违背标准。当然，标准将会被修订，可能在 1003.In 修订版中（对 1003.1c-1995 的修改），但还需要等一段时间。

递键的先前值）。例如，如果你的线程私有数据值是堆存储的地址，并且你想要在 `destructor` 函数中释放存储，就必须使用传递给 `destructor` 的参数，而非调用 `pthread_getspecific` 的参数。

如果终止线程对于某键的值为 `NULL` 也拒绝停止划船，则 Pthreads 将不会为线程私有数据键调用 `destructor`。`NULL` 有特殊的含义，意味着该键没有值。如果你曾经使用 `pthread_setspecific` 将线程私有数据键的值置为 `NULL`，需要记得你不是在赋空值，而是说该键在当前的线程中不再有值。

| `Destructor` 功能仅仅当线程终止时被调用，而不是当线程私有数据键的值改变时。

要记住的另外一件重要的事情是：线程私有数据键的 `destructor` 函数在你替代现存的键值时不会被调用。即，如果在堆中分配一个结构并将指向该结构的指针作为线程私有数据键的值，则在以后分配一个新结构并且把指向新结构的指针赋给相同的线程私有数据键时，你需要负责将旧结构释放。Pthreads 不会释放旧结构，也不会使用指向旧结构的指针调用你的 `destructor` 函数。

5.4.3 使用 `destructor` 函数

当一个线程退出时，它有一些为线程私有数据键定义的值，通常需要处理它们。如果键值是指向堆存储器的指针，需要释放存储器避免每次线程终止时留下内存泄漏。当你创建一个线程私有数据键时，Pthreads 允许你定义 `destructor` 函数。当具有非空的私有数据键值的一个线程终止时，键的 `destructor`（如果存在）将以键的当前值为参数被调用。

| 线程私有数据的 `destructors` 以“未特别指出的顺序”被调用。

当一个线程退出时，Pthreads 在进程中检查所有的线程私有数据键，并且将所有不是空的线程私有数据键置为空，然后调用键的 `destructor` 函数。回到我们的类比，某人为收集所有程序员的身份徽章，可能从每个程序员的左侧衬衫衣袋取下某样东西——因为他清楚地知道那就是身份徽章。小心，因为被调用的 `destructors` 顺序是未定义的，所以试着使每个 `destructor` 尽可能独立。

线程私有数据的 `destructors` 将为正在被破坏或为任何其他键赋予新值。不应该直接这样做，但是如果从 `destructor` 中调用其他函数，它就能很容易地间接发生。例如，ANSIC 库的 `destructors` 可能在你的 `destructors` 之前被调用——并且，调用一个 ANSIC 函数（如使用 `fprintf` 写一条记录消息到一个文件中）将导致新值被指派给一个线程私有数据键。在所有 `destructors` 被调用后，系统必须再核对一次线程私有数据值列表。

| 如果你的线程私有数据的 `destructor` 创建了新的线程私有数据值，你将得到另外的机会。也许是太多机会！

标准要求 Pthreads 实现在核对列表某个固定的次数后再放弃。当它放弃时，最终的线程私有数据值没有被破坏。如果值是指向堆存储器的指针，结果可能是一个内存泄漏，因此一定要小心。头文件 `<limits.h>` 中定义的 `_PTHREAD_DESTRUCTOR_ITERATIONS` 规定了系统检查列表的次数，并且值必须至少为 4。或者，系统被允许永远不停地检查，则一个总是设置线程私有数据值的 `destructor` 函数可能引起一个无限循环。

通常，仅当子系统 1 使用了取决于另外的独立子系统 2 的线程私有数据时，新的线程私有数据值才在 `destructor` 函数内被设置。因为 `destructor` 函数执行的顺序是未特别指出的，两个 `destructor` 函数可能被以错误的顺序调用。如果子系统 1 的 `destructor` 需要调用子系统 2，它可能无意中导致为子系统 2 分配新的线程私有数据。尽管子系统 2 的 `destructor` 需要被再次调用以释放新数据，子系统 1 的线程私有数据仍然保持为 `NULL`，所以循环将终止。

下列程序 `tsd_destructor.c` 表明了当一个线程终止时使用线程私有数据的 `destructors` 释放存储器。它还跟踪有多少线程正在使用线程私有数据，并且当最后线程的 `destructor` 被调用时，删除线程私有数据键。这个程序与 5.3 节的 `tsd_once.c` 的结构是类似的，所以这里仅仅将相关的差别加以解释。

- 12~14 除了键值 (`identity_key`) 外，程序还维护一个正在使用键的线程数 (`identity_key_counter`)，它被一个互斥量 (`identity_key_mutex`) 保护。
- 22~42 函数 `identity_key_destructor` 是线程私有数据键的 `destructor` 函数。它由打印一条消息开始，所以当它在每个线程中运行时，我们能观察到它。它释放用来维护线程私有数据的内存 `private_t` 结构，然后锁住与线程私有数据键相关的互斥量 (`identity_key_mutex`)，并且减少使用键的线程计数。如果计数为 0，它将删除键并打印一条消息。
- 48~63 函数 `identity_key_get` 能在任何地方使用（在本例中，它在每个线程中仅使用一次）来获得调用线程的 `identity_key` 值。如果当前没有值（值为 `NULL`），则它将给键分配一个新的 `private_t` 结构，以备将来引用。
- 68~78 函数 `thread_routine` 是本例中使用的线程启动函数，它通过调用 `identity_key_get` 获得键值，并设置结构的成员。`string` 成员被设置为线程的参数，为线程创建一个全局的名字，用于打印消息。
- 80~114 主程序创建线程私有数据键 `tsd_key`。注意，不同于 `tsd_once.c`，这个程序不需使用 `pthread_once`。正如我在那个例子的注解中提及的，在一个主程序中、在创建任何线程前，创建键是绝对安全的并且更有效。
- 101 主程序初始化引用计数 (`identity_key_counter`) 为 3。你预先定义有多少线程将引用一个键是重要的，而且如我们所希望的，该键将基于一个引用计数被删除。计数必须在使用键的任何线程可能终止前被设置。

例如，你不能将 identity_key_get 编码为当它首次为 identity_key 分配线程私有值时就动态地增加记数器。因为一个线程可能为 identity_key 分配线程私有值，然后在使用键的另外线程有机会开始前就终止。如果发生这种情况，第一个线程的 destructor 函数将发现没有其他的线程引用键，它将删除键，然后的线程在尝试设置线程私有数据值时将失败。

■ tsd_destructor.c

```
1 #include <pthread.h>
2 #include "errors.h"
3
4 /*
5  * Structure used as value of thread-specific data key.
6  */
7 typedef struct private_tag {
8     pthread_t    thread_id;
9     char         *string;
10 } private_t;
11
12 pthread_key_t identity_key;           /* Thread-specific data key */
13 pthread_mutex_t identity_key_mutex = PTHREAD_MUTEX_INITIALIZER;
14 long identity_key_counter = 0;
15
16 /*
17  * This routine is called as each thread terminates with a value
18  * for the thread-specific data key. It keeps track of how many
19  * threads still have values, and deletes the key when there are
20  * no more references.
21  */
22 void identity_key_destructor (void *value)
23 {
24     private_t *private = (private_t*)value;
25     int status;
26
27     printf ("thread \"%s\" exiting...\n", private->string);
28     free (value);
29     status = pthread_mutex_lock (&identity_key_mutex);
30     if (status != 0)
31         err_abort (status, "Lock key mutex");
32     identity_key_counter--;
33     if (identity_key_counter <= 0) {
34         status = pthread_key_delete (identity_key);
35         if (status != 0)
36             err_abort (status, "Delete key");
37         printf ("key deleted...\n");
38     }
39     status = pthread_mutex_unlock (&identity_key_mutex);
40     if (status != 0)
41         err_abort (status, "Unlock key mutex");
42 }
43
44 /*
45  * Helper routine to allocate a new value for thread-specific
```

```
46  * data key if the thread doesn't already have one.
47  */
48 void *identity_key_get (void)
49 {
50     void *value;
51     int status;
52
53     value = pthread_getspecific (identity_key);
54     if (value == NULL) {
55         value = malloc (sizeof (private_t));
56         if (value == NULL)
57             errno_abort ("Allocate key value");
58         status = pthread_setspecific (identity_key, (void*)value);
59         if (status != 0)
60             err_abort (status, "Set TSD");
61     }
62     return value;
63 }
64
65 /*
66  * Thread start routine to use thread-specific data.
67  */
68 void *thread_routine (void *arg)
69 {
70     private_t *value;
71
72     value = (private_t*)identity_key_get ();
73     value->thread_id = pthread_self ();
74     value->string = (char*)arg;
75     printf ("thread \"%s\" starting...\n", value->string);
76     sleep (2);
77     return NULL;
78 }
79
80 void main (int argc, char *argv[])
81 {
82     pthread_t thread_1, thread_2;
83     private_t *value;
84     int status;
85
86     /*
87      * Create the TSD key, and set the reference counter to
88      * the number of threads that will use it (two thread_routine
89      * threads plus main). This must be done before creating
90      * the threads! Otherwise, if one thread runs the key's
91      * destructor before any other thread uses the key, it will
92      * be deleted.
93      *
94      * Note that there's rarely any good reason to delete a
95      * thread-specific data key.
96      */
97     status = pthread_key_create (
98         &identity_key, identity_key_destructor);
```

```
99      if (status != 0)
100         err_abort (status, "Create key");
101     identity_key_counter = 3;
102     value = (private_t*)identity_key_get ();
103     value->thread_id = pthread_self ();
104     value->string = "Main thread";
105     status = pthread_create (&thread_1, NULL,
106         thread_routine, "Thread 1");
107     if (status != 0)
108         err_abort (status, "Create thread 1");
109     status = pthread_create (&thread_2, NULL,
110         thread_routine, "Thread 2");
111     if (status != 0)
112         err_abort (status, "Create thread 2");
113     pthread_exit (NULL);
114 }
```

■ tsd_destructor.c

5.5 实时调度

*"Well, it's no use your talking about waking him," said Tweedledum,
"when you're only one of the things in his dream. You know
very well you're not real."
"I am real!" said Alice, and began to cry.
"You won't make yourself a bit realer by crying," Tweedledee remarked:
"there's nothing to cry about."*
—Lewis Carroll, Through the Looking-Glass

从前，实时编程被认为是神秘且稀罕的艺术，实时程序员是在做不平常的事情，超脱编程主流，就像控制核反应堆或飞机导航系统一样。但是 POSIX.1b 的实时扩展定义了实时作为“操作系统在有限制的响应时间内提供特定水平的服务的能力”。适用于操作系统的同样也适用于你的应用程序或库。

“受限制”的响应时间不一定是“快”的反应，而是确实意味着“可预知”的响应速度。必须有一些方法来定义一个时间跨度，在该时间段内一系列操作保证被完成。控制一个核反应堆的系统比你将写的大多数程序有更严格的响应要求，并且没能满足反应堆要求的后果是更严重的。但是你写的很多代码将需要在“确定的反应时间”内提供一些“达到要求水平的服务”。实时编程就意味着软件生活在真实的世界。

实时编程涵概了一个如此广阔的区域，通常把它划分成两个独立的范畴。“硬实时”是传统上大多数人想到的那种实时。如果燃料杆的调整被推迟几微秒，你的核反应堆将会很危险；或者，如果航行系统用半秒的时间对偏离航向做出反应，你的飞机将会坠毁，这就是硬实时。硬实时是不可原谅的，因为服务要求的水平和反应时间被实际中或同等不可妥协的一些东西定义。“软实时”意味着你大部分时间

需要满足调度要求，但是如果不能能满足，后果也不是很严重。

许多与人交互的系统应该根据软实时的原则设计。尽管在计算机看来人反应慢，但是它们对响应时间敏感。当屏幕在接受下一鼠标点击前，让用户太长等待重换屏幕时，用户将很反感。没人喜欢“忙光标”——大多数人期望响应至少可预知，即使当它不能很快时。

线程对于所有类型的实时编程是有用的，因为当你能操作隔离时，更容易编写可预知响应时间的代码。“用户输入函数”不必等你的排序操作或屏幕更新操作，因为它是独立执行的。

然而，获得可预测性比把操作分在不同的线程中要求得更多。首先，需要保证你“不久”要运行的线程不会被留在运行队列中等待另外的线程使用处理器。默认地，大多数系统将差不多公平地在线程之间试着分配资源。在很多情况下这是好的——但是实时不是公平的。实时意味着小心地给予限制外部反应时间的程序部分优先级。

5.5.1 POSIX 实时选项

POSIX 标准是灵活的，因为它们被设计为在广泛的环境内有用。特别地，由于传统的 UNIX 系统不支持任何形式的实时调度控制，所有为控制实时反应的工具是可选的。一个给定的“遵循 1003.1c-1995 的 UNIX 实现”并不意味着你能编写可预知的实时程序。

如果系统定义了_POSIX_THREAD_PRIORITY_SCHEDULING，它为线程指派实时调度优先级提供支持。POSIX 的调度模型比传统 UNIX 的优先级模型更复杂一些，但是原则是类似的。优先级调度允许程序员给系统提供了任何两个线程间相对对方有多么重要的一个想法。无论何时当多个线程准备好执行时，系统将选择最高优先级的线程。

5.5.2 调度策略和优先级

```
int sched_get_priority_max (int policy);
int sched_get_priority_min (int policy);
int pthread_attr_getinheritsched (
    const pthread_attr_t *attr, int *inheritsched);
int pthread_attr_setinheritsched (
    pthread_attr_t *attr, int inheritsched);
int pthread_attr_getschedparam (
    const pthread_attr_t *attr,
    struct sched_param *param);
int pthread_attr_setschedparam (
    pthread_attr_t *attr,
    const struct sched_param *param);
int pthread_attr_getschedpolicy (
    const pthread_attr_t *attr, int *policy);
```

```
int pthread_attr_setschedpolicy (
    pthread_attr_t *attr, int policy);
int pthread_getschedparam (pthread_t thread,
    int *policy, struct sched_param *param);
int pthread_setschedparam(
    pthread_t thread, int policy,
    const struct sched_param *param);
```

支持`_POSIX_THREAD_PRIORITY_SCHEDULING`的 Pthreads 系统必须提供至少包括成员`sched_priority`的`struct sched_param`结构的定义。`sched_priority`成员是标准 Pthreads 调度策略 (`SCHED_FIFO` 和 `SCHED_RR`) 使用的惟一参数。允许各个调度策略的最小和最大优先级 (`sched_priority` 成员) 可以分别通过调用 `sched_get_priority_min` 或 `sched_get_priority_max` 设定。支持附加的、非标准调度策略的 Pthreads 系统可以包括附加的成员。

`SCHED_FIFO` (先入先出) 策略允许一个线程运行直到有更高优先级的线程准备好, 或者直到它自愿阻塞自己。在 `SCHED_FIFO` 调度策略下, 当有一个线程准备好时, 除非有平等或更高优先级的线程已经在运行, 否则它会很快开始执行。

`SCHED_RR` (轮循) 策略是基本相同的, 不同之处在于: 如果有一个 `SCHED_RR` 策略的线程执行了超过一个固定的时期 (时间片间隔) 没有阻塞, 而另外的 `SCHED_RR` 或 `SCHED_FIFO` 策略的相同优先级的线程准备好时, 运行的线程将被抢占以便准备好的线程可以执行。

当有 `SCHED_FIFO` 或 `SCHED_RR` 策略的线程在一个条件变量上等待或等待加锁同一个互斥量时, 它们将以优先级顺序被唤醒。即, 如果一个低优先级的 `SCHED_FIFO` 线程和一个高优先级的 `SCHED_FIFO` 线程都在等待锁相同的互斥量, 则当互斥量被解锁时, 高优先级线程将总是被首先解除阻塞。

Pthreads 定义了一条附加调度策略的名字——`SCHED_OTHER`, 然而, 关于这条调度策略根本没说任何东西。非官方的 POSIX 哲学的一个解释, 被称为“实现非标准内容的一个标准方法”(或者, “实现不可移植代码的一个可移植的方法”)。即, 当使用支持优先级调度选项的 Pthreads 实现时, 可以写一个可移植的程序, 它创建了以 `SCHED_OTHER` 策略运行的线程, 但是那个程序的行为是不可移植的 (`SCHED_OTHER` 的官方的解释是指为程序声明它不需要实时调度策略提供一个可移植的方法)。

`SCHED_OTHER` 策略既可以是 `SCHED_FIFO` 的别名, 也可以是 `SCHED_RR`, 或者是全然不同的其他东西。有关该歧义的实际问题不是你不知道 `SCHED_OTHER` 做什么, 而是你没法知道它可能要求的调度参数。因为 `SCHED_OTHER` 的意思是未定义的, 它不必要使用结构 `struct sched_param` 的 `sched_priority` 成员, 并且它可能使用 POSIX 实现加入该结构的非标准中的成员参数。如果这点有任何意义的话, 就是 `SCHED_OTHER` 是不可移植的。如果使用 `SCHED_OTHER` 编口写

的代码，你应该知道该代码是不可移植的——由定义，你是依赖于你在其上写代码的特别 Pthreads 实现中的 SCHED_OTHER。

`schedpolicy` 和 `schedparam` 属性分别由 `pthread_attr_setschedpolicy` 和 `pthread_attr_setschedparam` 调用设定，为属性对象指定显式的调度策略和参数。Pthreads 不为这些属性的任何一个指定缺省值，即意味着每个实现都可以选择一些“适当”的值。例如，一个为嵌入式控制应用设计的实时操作系统，可能选择默认地用 SCHED_FIFO 策略创建线程，并且，也许会使用一些中间范围的优先级。

默认地，大多数多用户操作系统更可能使用一种非标准的“分时”(*timeshare*) 调度策略，导致线程或多或少地被调度，好像程序总是运行一样。例如，系统可以暂时减少“CPU 猪”(译者注：大量占用 CPU 的线程)的优先级以便它们不能阻止另外的线程获得执行。

一个多用户操作系统的例子是 Digital UNIX，它支持两种非标准的分时共享调度策略。默认是前台策略(SCHED_FG_NP)，被用于正常的交互活动，并对应于非线程过程的调度。后台策略(SCHED_BG_NP)能被用于少量的重要支持活动。

| 当在属性对象中设置调度策略或优先级时，必须同时设置 `inheritsched` 属性！

可能调用 `pthread_attr_setinheritsched` 函数来设置 `inheritsched` 属性。该属性控制你创建的线程是从创建线程那儿继承调度信息，还是使用在 `schedpolicy` 和 `schedparam` 属性中的显式设置的调度信息。Pthreads 不为 `inheritsched` 指定默认值，因此如果你关心线程的调度策略和参数，必须总是设置该属性。

将 `inheritsched` 属性设置为 `PTHREAD_INHERIT_SCHED` 将使新线程继承创建线程的调度策略和参数。当你正在创建“助理线程”(代表创建线程工作的线程)时，调度继承是有用的——它们使用相同的策略和优先级执行通常是有意义的。无论何时，当你需要控制一个线程的调度策略或参数时，必须将 `inheritsched` 属性置为 `PTHREAD_EXPLICIT_SCHED`。

58~118 下列程序 `sched_attr.c` 显示了如何使用属性对象来创建一个具有显式的调度策略和优先级的线程。注意，程序使用了条件代码来判断在编译时是否支持 Pthreads 的优先级调度特征。如果选择没被支持，它将打印一条消息然后继续，尽管在这种情况下程序将不做很多工作(它用默认的调度策略创建一个线程，仅仅能说该线程可运行)。

尽管 Solaris 2.5 定义了`_POSIX_THREAD_PRIORITY_SCHEDULING`，它不支持 POSIX 的实时调度策略，并且试图将策略属性设置为 `SCHED_RR` 的操作将失败。该程序视为 Solaris 没有定义`_POSIX_THREAD_PRIORITY_SCHEDULING` 选项。

```
■ sched_attr.c

---



```
1 #include <unistd.h>
2 #include <pthread.h>
3 #include <sched.h>
4 #include "errors.h"
5
6 /*
7 * Thread start routine. If priority scheduling is supported,
8 * report the thread's scheduling attributes.
9 */
10 void *thread_routine (void *arg)
11 {
12 int my_policy;
13 struct sched_param my_param;
14 int status;
15
16 /*
17 * If the priority scheduling option is not defined, then we
18 * can do nothing with the output of pthread_getschedparam,
19 * so just report that the thread ran, and exit.
20 */
21 #if defined (_POSIX_THREAD_PRIORITY_SCHEDULING) && !defined (sun)
22 status = pthread_getschedparam (
23 pthread_self (), &my_policy, &my_param);
24 if (status != 0)
25 err_abort (status, "Get sched");
26 printf ("thread_routine running at %s/%d\n",
27 (my_policy == SCHED_FIFO ? "FIFO"
28 : (my_policy == SCHED_RR ? "RR"
29 : (my_policy == SCHED_OTHER ? "OTHER"
30 : "unknown"))),
31 my_param.sched_priority);
32 #else
33 printf ("thread_routine running\n");
34 #endif
35 return NULL;
36 }
37
38 int main (int argc, char *argv[])
39 {
40 pthread_t thread_id;
41 pthread_attr_t thread_attr;
42 int thread_policy;
43 struct sched_param thread_param;
44 int status, rr_min_priority, rr_max_priority;
45
46 status = pthread_attr_init (&thread_attr);
47 if (status != 0)
48 err_abort (status, "Init attr");
49
50 /*
51 * If the priority scheduling option is defined, set various
52 * scheduling parameters. Note that it is particularly important
```


```

```
53     * that you remember to set the inheritsched attribute to
54     * PTHREAD_EXPLICIT_SCHED, or the policy and priority that you've
55     * set will be ignored! The default behavior is to inherit
56     * scheduling information from the creating thread.
57     */
58 #if defined (_POSIX_THREAD_PRIORITY_SCHEDULING) && !defined (sun)
59     status = pthread_attr_getschedpolicy (
60         &thread_attr, &thread_policy);
61     if (status != 0)
62         err_abort (status, "Get policy");
63     status = pthread_attr_getschedparam (
64         &thread_attr, &thread_param);
65     if (status != 0)
66         err_abort (status, "Get sched param");
67     printf (
68         "Default policy is %s, priority is %d\n",
69         (thread_policy == SCHED_FIFO ? "FIFO"
70          : (thread_policy == SCHED_RR ? "RR"
71              : (thread_policy == SCHED_OTHER ? "OTHER"
72                  : "unknown"))),
73         thread_param.sched_priority);
74
75     status = pthread_attr_setschedpolicy (
76         &thread_attr, SCHED_RR);
77     if (status != 0)
78         printf ("Unable to set SCHED_RR policy.\n");
79     else {
80         /*
81         * Just for the sake of the exercise, we'll use the
82         * middle of the priority range allowed for
83         * SCHED_RR. This should ensure that the thread will be
84         * run, without blocking everything else. Because any
85         * assumptions about how a thread's priority interacts
86         * with other threads (even in other processes) are
87         * nonportable, especially on an implementation that
88         * defaults to System contention scope, you may have to
89         * adjust this code before it will work on some systems.
90         */
91         rr_min_priority = sched_get_priority_min (SCHED_RR);
92         if (rr_min_priority == -1)
93             errno_abort ("Get SCHED_RR min priority");
94         rr_max_priority = sched_get_priority_max (SCHED_RR);
95         if (rr_max_priority == -1)
96             errno_abort ("Get SCHED_RR max priority");
97         thread_param.sched_priority =
98             (rr_min_priority + rr_max_priority)/2;
99         printf (
100             "SCHED_RR priority range is %d to %d: using %d\n",
101             rr_min_priority,
102             rr_max_priority,
103             thread_param.sched_priority);
104         status = pthread_attr_setschedparam (
105             &thread_attr, &thread_param);
```

```
106     if (status != 0)
107         err_abort (status, "Set params");
108     printf (
109         "Creating thread at RR/%d\n",
110         thread_param.sched_priority);
111     status = pthread_attr_setinheritsched (
112         &thread_attr, PTHREAD_EXPLICIT_SCHED);
113     if (status != 0)
114         err_abort (status, "Set inherit");
115 }
116 #else
117     printf ("Priority scheduling not supported\n");
118 #endif
119     status = pthread_create (
120         &thread_id, &thread_attr, thread_routine, NULL);
121     if (status != 0)
122         err_abort (status, "Create thread");
123     status = pthread_join (thread_id, NULL);
124     if (status != 0)
125         err_abort (status, "Join thread");
126     printf ("Main exiting\n");
127     return 0;
128 }
```

■ sched_attr.c

下一个程序 sched_thread.c 显示了如何为一个正在运行的线程修改实时调度策略和参数。当在线程属性对象中改变调度策略和参数时，要记得，你使用了两个分开的操作：修改调度策略和修改调度参数。

不能独立于线程的参数来修改一个执行线程的调度策略，为了调度正确操作，策略和参数一定是一致的。每个调度策略有一个优先级的唯一范围，并且一个线程不能以一个对当前调度策略而言无效的优先级执行。为保证策略和参数的一致性，它们被设置于单个调用中。

不同于 sched_attr.c，sched_thread.c 不检查编译时特征宏 _POSIX_THREAD_PRIORITY_SCHEDULING。这意味着，在一个不支持该选择的系统上，它可能不被编译，并且几乎不会正确运行。以这种方式写程序没有什么不对的，事实上，这可能是你大部分时间所做的工作。如果需要优先级调度，应记录下你的应用要求 _POSIX_THREAD_PRIORITY_SCHEDULING 选项，并且使用它。

Solaris 2.5，尽管定义了 _POSIX_THREAD_PRIORITY_SCHEDULING，但不支持实时调度策略。因此，从 sched_get_priority_min 返回的 ENOSYS 被作为一个特殊的情况处理。

■ sched_thread.c

```
1 #include <unistd.h>
2 #include <pthread.h>
3 #include <sched.h>
```

```
4 #include "errors.h"
5
6 #define THREADS 5
7
8 /*
9  * Structure describing each thread.
10 */
11 typedef struct thread_tag {
12     int          index;
13     pthread_t    id;
14 } thread_t;
15
16 thread_t      threads[THREADS];
17 int           rr_min_priority;
18
19 /*
20  * Thread start routine that will set its own priority.
21  */
22 void *thread_routine (void *arg)
23 {
24     thread_t *self = (thread_t*)arg;
25     int my_policy;
26     struct sched_param my_param;
27     int status;
28
29     my_param.sched_priority = rr_min_priority + self->index;
30     DPRINTF ((
31         "Thread %d will set SCHED_FIFO, priority %d\n",
32         self->index, my_param.sched_priority));
33     status = pthread_setschedparam (
34         self->id, SCHED_RR, &my_param);
35     if (status != 0)
36         err_abort (status, "Set sched");
37     status = pthread_getschedparam (
38         self->id, &my_policy, &my_param);
39     if (status != 0)
40         err_abort (status, "Get sched");
41     printf ("thread_routine %d running at %s/%d\n",
42             self->index,
43             (my_policy == SCHED_FIFO ? "FIFO"
44              : (my_policy == SCHED_RR ? "RR"
45              : (my_policy == SCHED_OTHER ? "OTHER"
46              : "unknown"))),
47             my_param.sched_priority);
48     return NULL;
49 }
50
51 int main (int argc, char *argv[])
52 {
53     int count, status;
54
55     rr_min_priority = sched_get_priority_min (SCHED_RR);
56     if (rr_min_priority == -1) {
```

```
57 #ifdef sun
58     if (errno == ENOSYS) {
59         fprintf (stderr, "SCHED_RR is not supported.\n");
60         exit (0);
61     }
62 #endif
63     errno_abort ("Get SCHED_RR min priority");
64 }
65     for (count = 0; count < THREADS; count++) {
66         threads[count].index = count;
67         status = pthread_create (
68             &threads[count].id, NULL,
69             thread_routine, (void*)&threads[count]);
70         if (status != 0)
71             err_abort (status, "Create thread");
72     }
73     for (count = 0; count < THREADS; count++) {
74         status = pthread_join (threads[count].id, NULL);
75         if (status != 0)
76             err_abort (status, "Join thread");
77     }
78     printf ("Main exiting\n");
79     return 0;
80 }
```

■ sched_thread.c

5.5.3 竞争范围和分配域 (Contention scope and allocation domain)

```
int pthread_attr_getscope (
    const pthread_attr_t *attr, int *contentionscope);
int pthread_attr_setscope (
    pthread_attr_t *attr, int contentionscope);
```

除调度策略和参数以外，另外两个控制量在实时调度中也是重要的。除非你正在写实时应用，否则它们可能没有关系。如果你正在写一个实时的应用程序，应该知道系统对这些控制量设置的支持。

第一个控制量被称为竞争范围。它描述了线程为处理器资源而竞争的方式。系统竞争范围意味着线程与进程之外的线程竞争处理器资源。一个进程内的高优先级系统竞争范围线程能阻止其他进程内的系统竞争范围线程运行(或反过来也如此)。进程竞争范围指线程仅仅在同一进程内相互竞争。进程竞争范围通常意味着，操作系统选择一个进程执行，可能仅仅使用传统型的 UNIX 优先级，然后一些附加的调度程序在进程内部应用 POSIX 调度规则决定哪个线程执行。

Pthreads 提供了线程范围属性，以便能指定创建的线程应有进程或系统竞争范围。一个 Pthreads 系统可以选择支持 PTHREAD_SCOPE_PROCESS、PTHREAD_SCOPE_SYSTEM，或同时支持二者。如果你试着用没被系统支持的一

个范围来创建一个线程，`pthread_attr_setscope` 将返回 `ENOTSUP`。

第二个控制量是分配域。分配域是系统内线程可以为其竞争的处理器的集合。一个系统可以有一个以上的分配领域，每个包含一个以上的处理器。在一个单处理机系统内，分配域将只包含一个处理器，但是你仍然可以有多个分配域。在一台多处理机上，各个分配领域可以包含从一个处理器到系统中所有的处理器。

没有设置线程分配域的 Pthreads 接口。POSIX.14（多处理机专题）工作组考虑了建议标准的接口，但是仍受阻于对广泛的硬件体系结构和现存的软件接口的处理。尽管缺乏一个标准，支持多处理机的任何系统将有接口去影响一个线程的分配领域。

因为没有控制分配域的标准接口，没有办法精确描述任何特别的假想状况的所有效果。如果你使用支持多处理机的一个系统，仍然需要担心这些事情。考虑以下的事情：

1. 在相同的分配域内，系统竞争范围线程和进程竞争范围线程如何与对方交互？它们以某种方式进行资源竞争，但是该行为没被标准定义。
2. 如果系统支持“重叠”的分配域，换句话说，如果一个处理器能在系统内的多个分配域内出现，并且在每个重叠的分配领域内有一个系统竞争范围线程，会发生什么呢？

| 系统竞争范围是可预知的。

| 进程竞争范围是低廉的。

在大多数系统上，仅仅使用进程竞争范围，你将获得更好的性能价格比。系统竞争范围线程之间的环境切换通常要求至少一次内核调用，并且那些调用与在用户模式下保存和恢复线程状态的费用相比是相对昂贵的。每个系统竞争范围线程将永久地与一个“核实体”联系，并且核实体的数目通常比 Pthreads 线程的数目更有限。进程竞争范围线程可以共享一个核实体，或少量的核实体。例如，在给定的系统配置上，你也许能创建几千个进程竞争范围线程，但是仅能创建几百个系统竞争范围线程。

另一方面，进程竞争范围在线程优先级调度上没有给你真正的控制——尽管高优先级线程可以优先于进程内的其他线程，但让它优先于其他进程的线程就没有意义了。系统竞争范围允许控制给你好一些的可预测性，经常使你的线程比在操作系统内核运行的线程“更加重要”。

| 系统竞争范围在一个大于 1 的分配域内有更少的可预言性。

当一个线程被分配到超过一个处理器的分配域时，应用程序不能再依靠完全可预知的调度行为。例如，高优先级和低优先级线程可以同时执行，调度程序将不允许因为一个高优先级线程正在运行，而使处理器闲置。单处理机行为在一台多处理机上没什么意义。

当线程 1 解锁互斥量唤醒线程 2，并且线程 2 比线程 1 有更高的优先级时，线程 2 将抢占线程 1 并且开始立刻运行。然而，如果线程 1 和线程 2 同时在一个人大于 1 的分配域内运行，并且线程 1 唤醒线程 3，它比线程 1 优先级低但是比线程 2 高，线程 3 不能很快抢占线程 2。线程 3 仍然将处于就绪态直到线程 2 阻塞。

对于一些应用，在上述的情况下保证抢占的可预测性是重要的。在大多数情况下，只要线程 3 最后运行，它不是那么重要。尽管 POSIX 不要求任何 Pthreads 系统实现这类跨处理器的抢占，但当你使用系统竞争范围线程时，更可能会发现它。当然，如果可预测性是关键的，无论如何都应该使用系统竞争范围。

5.5.4 实时调度的问题

依靠实时调度的问题之一是它不是模块化的。在真实的应用中，你通常使用多种来源的库，并且这些库可能依靠线程实现像网络通信和资源管理那样重要的功能。现在，为库中“最重要的线程”使用 SCHED_FIFO 策略和最大优先级可能似乎合理。然而，这样产生的线程将不仅是库中最重要的线程——而且是（或至少表现为）在整个进程中最重要的线程，包括主程序和任何其他库。高优先级线程可能阻止所有另外的库（在一些情况下甚至是操作系统）完成应用程序需要的操作。

另外一个问题确实不是优先级调度的问题，而是与许多人考虑优先级调度的方式有关，即它并没有像人们期望的那样。许多人认为“实时优先级线程某种程度上比另外的线程更快”，但那不总是真的。实时优先级线程实际上可能变得更慢，因为它包含了更多的要求抢占检查的开销，特别是在一台多处理机上。

固定优先级调度的一个更严重的问题被称为优先级倒置。优先级倒置是指一个低优先级线程能阻止一个高优先级线程的执行——它是调度和同步之间的一个不干净的相互作用结果。调度规则要求一个线程运行，但是同步要求运用另行的线程，所以两个线程的优先级好像被颠倒了。

当低优先级线程获得一个共享的资源（如一互斥量），并且被一个随后在同样资源上阻塞的高优先级线程抢占时，优先级倒置发生。当只有两个线程时，低优先级线程然后将被允许执行，最后（我们假定）释放互斥量。然而，如果在它们之间有第三个线程准备好时，它能阻止低优先级线程运行，因为低优先级线程拥有高优先级线程需要的互斥量，中间优先级线程也就阻止了高优先级线程的执行。

有很多方法可以阻止优先级倒置。最简单的是避免使用实时调度，但那并不总是实际的。Pthreads 提供了互斥量加锁协议帮助避免优先级倒置，这些将在 5.5.5 节讨论。

| 大部分线程程序不需要实时调度。

最后一个问题是：优先级调度不是完全可移植的。Pthreads 定义在一个选项下的优先级调度特征，而那些不是首先为实时编程计划的实现可能不支持该选项。就算选项被支持，还有许多没被标准覆盖的优先级调度的重要方面。例如，当你使用

系统竞争范围时，你的线程可以直接与操作系统内的线程竞争，提高你的线程的优先级可能会阻止内核 I/O 驱动程序在一些系统上工作。

Pthreads 不指定一个线程的调度策略或缺省优先级，或标准调度策略如何与非标准策略交互。因此，当你设置线程调度策略和优先级时，应使用可移植的接口。标准不提供方法预测哪个设置将怎么影响进程或系统中的任何其他线程。

如果你确实需要优先级调度，那么使用它——并且知道它有超越简单 Pthreads 的特殊要求。如果你需要优先级调度，记在以下几点：

1. 进程竞争范围比系统竞争范围“更好”，因为你将不会阻止其他进程或内核中的某个线程运行。
2. SCHED_RR 比 SCHED_FIFO“更好”，并且更具可移植性，因为 SCHED_RR 线程将在与具有相同优先级的线程共享可用处理器时间间隔中被抢占。
3. 对 SCHED_FIFO 和 SCHED_RR 策略而言，低优先级比高优先级好，因为这更少可能妨碍另外重要的东西。

除非你的代码确实需要优先级调度，否则应避免使用它。在大多数情况下，与它将解决的问题相比，优先级调度将引起更多的问题。

5.5.5 优先级相关互斥量

```
$if defined (_POSIX_THREAD_PRIO_PROTECT) \
|| defined (_POSIX_THREAD_PRIO_INHERIT)
int pthread_mutexattr_getprotocol (
    const pthread_mutexattr_t *attr, int *protocol);
int pthread_mutexattr_setprotocol (
    pthread_mutexattr_t *attr, int protocol);
#endif
#ifndef _POSIX_THREAD_PRIO_PROTECT
int pthread_mutexattr_getprioceiling (
    const pthread_attr_t *attr, int *prioceiling);
int pthread_mutexattr_setprioceiling (
    pthread_mutexattr_t *attr, int prioceiling);
int pthread_mutex_getprioceiling (
    const pthread_mutex_t *mutex, int *prioceiling);
int pthread_mutex_setprioceiling (
    pthread_mutex_t *mutex,
    int prioceiling, int *old_ceiling);
#endif
```

Pthreads 提供了若干能帮助避免优先级倒置死锁的特殊互斥量属性。加锁或等待这些属性之一的某个互斥量可以改变线程的优先级——或另外线程的优先级，来保证拥有互斥量的线程不能被需要锁住相同互斥量的其他线程抢占。

这些互斥量属性可能没有被你的 Pthreads 实现支持，因为它们是可选的特征。如果你的代码需要选择或不选择这些功能，你可以基于特征测试宏

`_POSIX_THREAD_PRIO_PROTECT` 或 `_POSIX_THREAD_PRIO_INHERIT`（定义在`<unistd.h>`中）对引用进行条件编译；或在程序执行期间调用 `sysconf` 来检查 `_SC_THREAD_PRIO_PROTECT` 或 `_SC_THREAD_PRIO_INHERIT`。

一旦创建了一个使用这些属性之一的互斥量，就能像对待任何其他互斥量一样加锁和解锁该互斥量。结果，通过改变初始化互斥量的代码就能容易地变换创建的任何互斥量（必须调用 `pthread_mutex_init`，因为不能使用非确省属性静态地初始化一个互斥量）。

“优先级上限 (ceiling)” 协议意味着当一个线程拥有互斥量时，它将以指定的优先级运行。

如果你的系统定义 `_POSIX_THREAD_PRIO_PROTECT`，则它支持“优先级 ceiling”协议和 `prioceiling` 属性。通过调用 `pthread_mutexattr_setprotocol` 设置协议属性。如果将协议属性置为 `PTHREAD_PRIO_PROTECT`，也可以通过设置 `prioceiling` 属性为使用属性对象创建的互斥量设置优先级 `ceiling`。

通过调用 `pthread_mutexattr_setprioceiling` 设置 `prioceiling` 属性。当任何线程锁住与该属性对象相关定义的一个互斥量时，线程的优先级将被设置为互斥量的优先级 `ceiling`，除非线程的优先级已经是相同或更高的。注意，在高于互斥量优先级 `ceiling` 的线程内加锁该互斥量会打破协议，失去对优先级倒置的保护。

“优先级继承” 意味着，当一个线程在由另一个低优先级线程拥有的互斥量上等待时，后者的优先级将被增加到等待线程的优先级。

如果你的系统定义了 `_POSIX_THREAD_PRIO_INHERIT`，那它就支持协议属性。如果你将协议属性设置为 `PTHREAD_PRIO_INHERIT`，那么保持互斥量的线程就不能被另外的与等待互斥量的线程相比优先级低的任何线程抢占。当任何线程试图加锁互斥量、同时一个低优先级线程拥有该互斥量时，只要它拥有互斥量，当前拥有互斥量的线程的优先级将被提高到等待线程的优先级。

如果你的系统或者未定义 `_POSIX_THREAD_PRIO_PROTECT`，或者未定义 `_POSIX_THREAD_PRIO_INHERIT`，则协议属性可能没有被定义。协议属性的默认值（或是属性没被定义的有效值）是 `POSIX_PRIO_NONE`，它意味着线程优先级没有因加锁（或等待）互斥量的行为而被修改。

5.5.5.1 优先级 ceiling 互斥量

两种最简单类型的优先级相关 (priority aware) 互斥量是优先级 ceiling (或“优先级保护”) 协议 (如图 5.3 所示)。当使用优先级 ceiling 创建一个互斥量时，你要指定当线程锁住互斥量时可以运用的最高优先级。任何锁住互斥量的线程将自动地将它的优先级提高到那个值，这将允许在它在被任何另外试图加锁该互斥量的线程抢占以前，完成对互斥量的操作。你还可以检验或修改使用优先级 ceiling (保护) 协议创建的互斥量的优先级 ceiling。

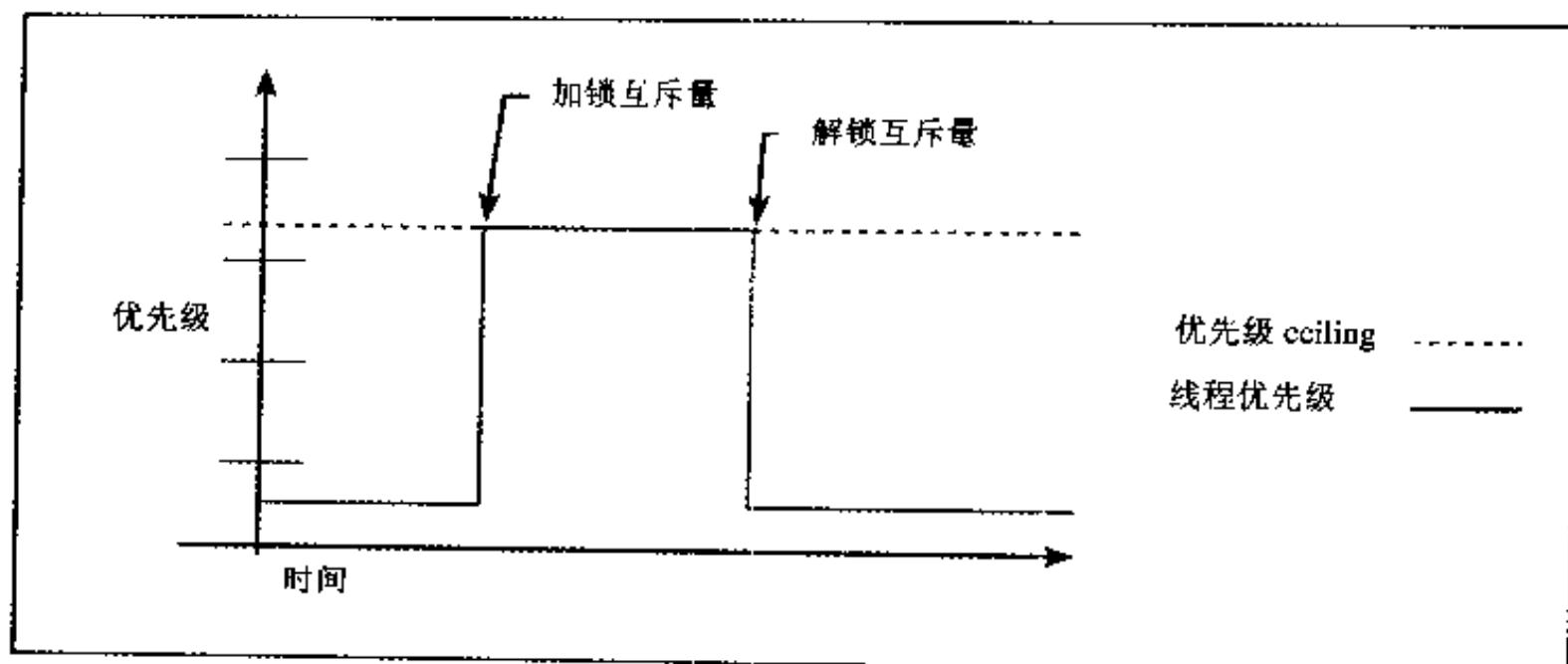


图 5.3 优先级 ceiling 互斥量操作

一个优先级 ceiling 互斥量在一个函数库（该库可能由你无法控制的线程调用）内是没有用的。如果有任何以比 ceiling 高的优先级运行的线程锁住优先级 ceiling 互斥量，协议将被破坏。这也不是说一定会发生优先级倒置，而是对于优先级倒置的所有保护都被取消了。与正常的“没有保护”的互斥量相比，由于优先级 ceiling 协议为每个互斥量操作增加了额外的开销，你可能浪费了处理器时间而不做任何事情。

对于开发者可以控制系统内所有同步的嵌入式实时应用程序而言，优先级 ceiling 是完美的。当设计代码时，优先级 ceiling 能被安全地判断，并且与更一般的方案相比，使用更少的性能开销来避免优先级倒置。当然，通过避免使用优先级调度或仅仅在相等优先级的线程内使用给定的互斥量，可以避免优先级倒置，而且总是最有效的方式。同样，当你最需要它们时，这些选择很少是可行的。

你能在几乎任何主程序内使用优先级 ceiling，甚至当使用你无法控制的库代码时。因为线程通常调用库函数来加锁库中的互斥量，而不是由库创建了调用代码来加锁应用程序互斥量。当使用有回调函数的库时，你必须保证要么那些回调函数（以及它们调用的任何函数）不使用优先级 ceiling 互斥量，要么激活回调函数的那些线程不会以高于互斥量优先级 ceiling 的优先级运行。

5.5.5.2 优先级继承互斥量

另外的 Pthreads 互斥量协议是优先级继承。在优先级继承协议中，当一个线程锁住一个互斥量时，线程的优先级就被互斥量控制（图 5.4）。当另外的线程在那个互斥量上阻塞时，它会查看拥有互斥量的线程的优先级。如果拥有互斥量的线程比试图在互斥量上阻塞的线程优先级低，则拥有互斥量的线程的优先级将被提升到阻塞线程的优先级。

除非等待的线程也将被抢占，提高优先级确保拥有互斥量的线程不能被抢占；或者说，拥有互斥量的线程是在代表高优先级的线程工作。当线程解锁互斥量时，线程的优先级自动被降到它原来的优先级，高优先级等待线程被唤醒。如果又有一

一个更高优先级的线程在互斥量上阻塞，则拥有互斥量的线程将再次增加优先级。当互斥量被解锁时，线程将仍然回到它原来的优先级。

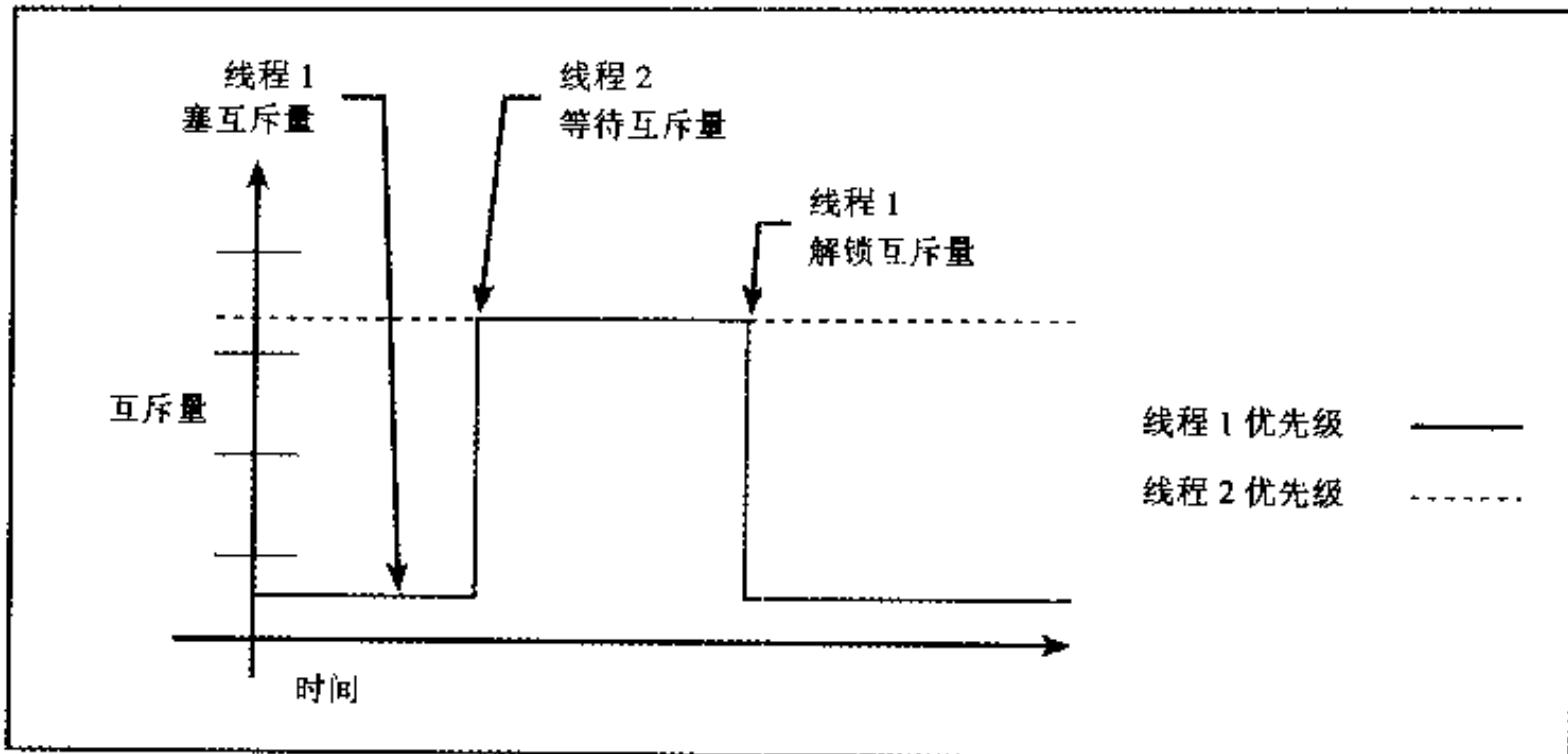


图 5.4 优先级继承互斥量操作

优先级继承协议比优先级 ceiling 更通用和强大，当然也更复杂和昂贵。如果一个函数库必须使用优先级调度，并且无法避免从不同优先级的线程内使用同一互斥量，则优先级继承是目前惟一可用的解决方案。如果你正在写一个主程序，并且知道你的互斥量不会被函数库内创建的线程加锁，则优先级 ceiling 将和优先级继承完成相同的结果，且带来更少的开销。

5.6 线程和核实体

*"Two lines!" cried the Mock Turtle. "Seals, turtles, salmon, and so on:
then, when you've cleared all the jelly-fish out of the way—"
"That generally takes some time," interrupted the Gryphon.
"—you advance twice—" "
"Each with a lobster as a partner!" cried the Gryphon.*
—Lewis Carroll, Alice's Adventures in Wonderland

Pthreads 几乎特意不说实现细节的事，这让每个供应商可以自由地基于用户的需要做决定并且通过允许革新使之继续发展。标准对实现只规定了一些必要的要求，这些已足够让你写出严格遵守标准的使用线程的 POSIX 应用^①，并且能在遵守标准的所有实现上正确运行。

^① 严格遵守 POSIX 的含义是指程序不依靠标准的任何可选项或扩展，对于所有具体实现的限制只需要取规定的最小值（但是对于任何允许的值都将工作正常）。

任何 Pthreads 实现必须保证“由一个线程调用的系统服务不能推迟另外的线程”，这使你不需要担心调用读 (read) 操作可能在一些系统上阻塞进程中所有的线程。另一方面，这意味着你的进程将不总是有最大可能的并发水平。

然而，当使用一个系统时，理解系统可能被实现的方法经常是有用的。例如，当写 ANSI C 表达式时，理解代码生成器、甚至硬件如何处理那些表达式经常是有用的。下列各节简短地描述了你可能遇见的一些主要变化。

在这些节中使用的重要术语是“Pthreads 线程”、“核实体”和“处理器”。Pthreads 线程意味着由你调用 `pthread_create` 创建的一个线程，由类型为 `pthread_t` 的一个标识符代表，这些是你使用 Pthreads 接口可以控制的。处理器是指物理硬件，特别指超过一个处理器的多处理机情况。

大多数操作系统至少在 Pthreads 线程和处理器之间有一层附加的抽象，这就是我所说的核实体（因为那是 Pthreads 使用的术语）。在一些系统中，核实体可能是一个传统型的 UNIX 进程；也可能是一个 Digital UNIX Mach 线程；或者是一个 Solaris 2.x LWP；或者是一个 IRIX sproc 进程。核实体的准确意思以及它怎么与 Pthreads 线程交互，在下列三个模型之间存在很大的差别。

5.6.1 多对一（用户级）

多对一 (many-to-one) 方法有时也被称为“库实现”。多对一实现通常是为了没有支持线程的操作系统设计的。在通用的 UNIX 内核上运行的 Pthreads 实现通常都属于这个范畴—例如，经典的 DCE 线程参考实现或 SunOS 4.x LWP 包（与 Solaris 2.x LWP 没有关系，后者是一个核实体）。

多对一实现不能在一台多处理机上利用并行的优势，并且任何阻塞式系统服务（如读调用）将阻塞进程中所有的线程。一些实现可能使用一些可用的 UNIX 特征（如非阻塞的 I/O 或 POSIX.1b 异步 I/O）来帮你避免这个问题。然而，这些特征具有限制。例如，不是所有的设备驱动程序都支持非阻塞 I/O，并且传统型的 UNIX 磁盘文件系统反应通常是被认为即时的，将忽略非阻塞的 I/O 模式。

一些多对一实现不能与 ANSI C 库的支持函数紧密结合，这可能引起严重的问题。例如，当一个线程等待你输入命令时，`stdio` 函数可能阻塞整个进程（和所有的线程）。然而，遵循 Pthreads 标准的任何多对一实现都避免了上述问题，也许还包括了 `stdio` 和其他函数的特殊版本。

当你要求并发但不需要并行时，多对一实现可以提供最好的线程创建性能和因互斥量和条件变量带来的环境切换效率。因为 Pthreads 库全部在用户模式下保存和恢复线程的环境，所以它是快的。例如，你能很快地创建很多线程并且在条件变量上阻塞它们的大多数（等一些外部的事件），而根本没有涉及内核。

图 5.5 显示了 Pthreads 线程（左列）映射到核实体（中间列）、再到物理的处理器（右列）的过程。在这种情况下，进程有四个 Pthreads 线程，分别从“Pthread 1”~“Pthread 4”标记。Pthreads 库通过切换状态寄存器（SP，通用寄存器等）在用户模

式下的单个进程上调度四个线程。库可以使用一个定时器来抢占运行太长的 Pthreads 线程。内核将进程在两个物理的处理器间调度，分别标记为“处理器 1”和“处理器 2”，该模型的重要特征如表 5.2 所示。

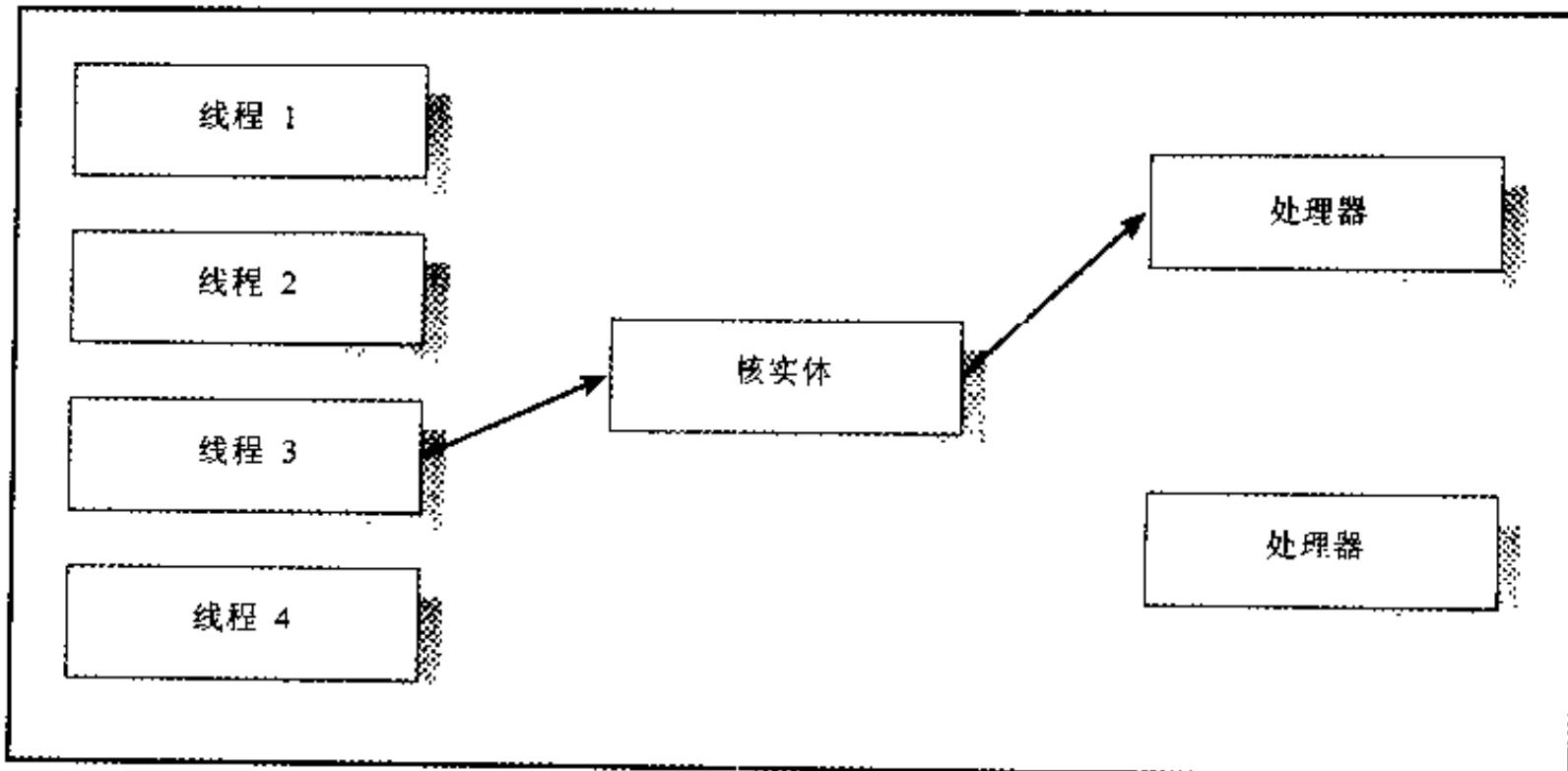


图 5.5 多对一线程图

表 5.2 多对一线程特征

优 点	缺 点
最快的环境切换时间	系统服务阻塞期间潜在的长延迟
简单；实现甚至可以是（大部分）可移植*	单个进程应用程序不能利用多处理机硬件

* DCE 用户模式线程调度器通常能在几天内移植到新的操作系统上，主要包括新的寄存器环境切换汇编语言例程。我们使用“一些汇编要求”的短句。

5.6.2 一对一（内核级）

一对一（one-to-one）线程映射有时也被称为一个“内核线程”实现。Pthreads 库把每个线程分派到一个核实体。通常它必须使用阻塞式内核功能在互斥量和条件变量上等待。由于同步可以在内核或用户模式中发生，线程调度在内核层实现。

在一对一的实现之中，Pthreads 线程能充分利用多处理机硬件优势而不需要你做任何额外的努力，例如把代码分开成多个的进程。与阻塞一个正常的 UNIX 进程不会影响另外的进程一样，当一个线程在内核阻塞时，不会影响另外的线程。一个线程甚至能在不影响另外线程的前提下处理页差错。

一对一的实现主要有两个问题。第一个是它们的可扩展性不好。即，应用程序的每个线程对应一个核实体。内核内存是宝贵的，例如进程和线程核对象通常受到预分配的数组限制，并且大多数实现将限制你能创建的线程数目。它还将限制全部系统能创建的线程数量——所以你的进程可能因为其他进程的原因而不能到达自

己的限制。

第二个问题是在一个互斥量上阻塞和在一个条件变量上等待时(这是在许多应用程序中经常发生的事情), 因为它们要求进入机器的内核保护模式, 所以大多数一对一实现花费更多的开销。尤其是当加锁一个尚未锁住的互斥量, 或当解锁一个没有线程等待的互斥量时, 要比多对一实现更加昂贵, 因为在大多数系统上这些功能可以在用户模式下完成。

一对一实现对于 CPU 密集型应用(不会经常阻塞)而言是个不错的选择。许多高效的并行应用程序开始时为系统中每个物理处理器创建一个工作线程, 并且一旦开始, 线程可以独立地运行一段较长的时间。因为它们不会要求内核创建很多线程, 这样的应用将工作得很好, 并且它们不要求很多进入内核的系统调用来阻塞和解除阻塞它们的线程。

图 5.6 显示了 Pthreads 线程(左列)映射到核实体(中间列)、再到物理处理器(右列)的过程。在这种情况下, 进程有四个 Pthreads 线程, 从“Pthread 1”~“Pthread 4”依次标记。每个 Pthreads 线程永久地与相应的核实体绑定。内核在两个物理处理器(处理器 1 和处理器 2)之间调度四个核实体(与另外的进程对应的那些核实体一起)。该模型的重要特征如表 5.3 所示。

5.6.3 多对少(两级)

多对少(many-to-few)模型试图兼有多对一模型和一对一同的优点, 同时避免它们的缺点。该模型要求建立用户水平 Pthreads 库和内核之间的合作, 它们共同承担调度责任并且可以在对方之间传达线程的信息。

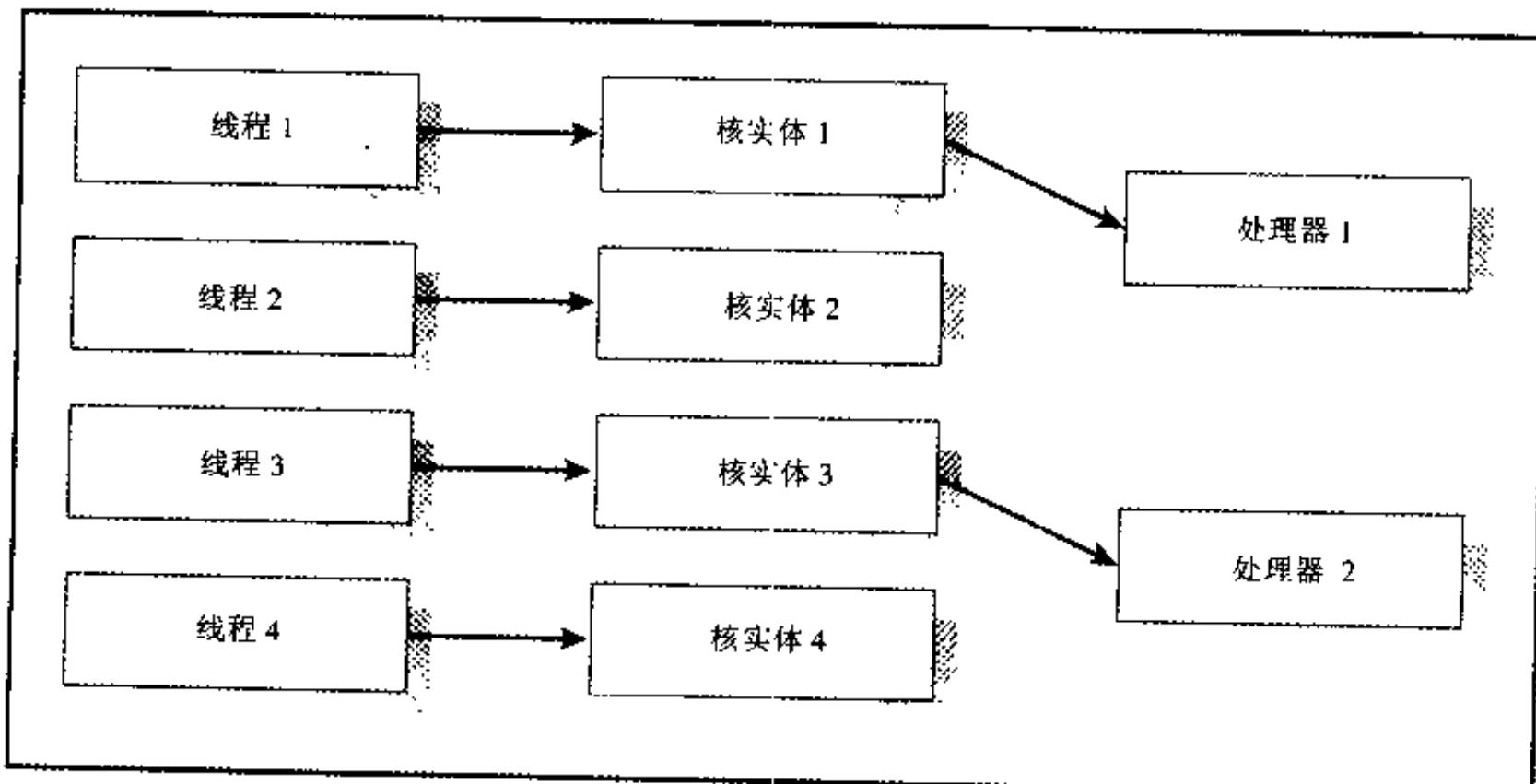


图 5.6 一对一线程图

表 5.3 一对一线程特征

优 点	缺 点
能在单个进程中利用多处理器硬件的优势	线程环境切换速度相对较慢(调用进内核)
在系统服务阻塞期间没有延迟	当使用许多线程时可扩展性差, 因为每个Pthreads线程从系统中使用核资源

当 Pthreads 库需要在两个线程之间切换时, 它能直接在用户模式下这样做。新的 Pthreads 线程在不需打扰内核的前提下在相同的核实体上运行。在大多数情况下, 当一个线程在一个互斥量或条件变量上阻塞时, 或者当一个线程终止时, 将获得多对一实现的性能优势。

当内核需要阻塞一个线程来等待 I/O 或另外的资源时, 它将按下面的方法来做。内核可以通知 Pthreads 库, 如在 Digital UNIX 4.0 中那样, 以便库能通过很快调度一个新的 Pthreads 线程来保持进程的并发性, 有点像原来由著名的华盛顿 research 大学提出的“调度器激活”模型[Anderson,1991]。或者, 内核可以简单地阻塞核实体, 在此情况下, 可以允许程序员增加分配给进程的核实体的数量, 如在 Solaris 2.5 中那样——否则, 即使另外的用户线程准备好运行, 当所有的核实体阻塞时进程可能被“阻塞”。

多对少模型在大多数实际应用程序中是胜任的, 因为在大多数应用程序中, 线程执行的操作既有 CPU 密集型也有 I/O 密集型操作, 既可能阻塞在 I/O 操作上, 也可能阻塞在 Pthreads 同步上。与物理的处理器相比, 大多数应用程序也将创建更多的线程, 或者直接创建, 或者通过并行线程函数库创建等。

图 5.7 显示了 Pthreads 线程(左列)映射到核实体(中间列), 再到物理的处理器(右列)的过程。在这种情况下, 进程有四个 Pthreads 线程, 从“线程 1”到“线程 4”依次标记。Pthreads 库在初始化时创建一些核实体(可能以后再创建)。典型地, 库将为每个物理处理器启动一个核实体(标记为“核实体 1”和“核实体 2”)。内核在两个物理处理器(标记为“处理器 1”和“处理器 2”)之间调度这些核实体(与另外进程对应的核实体一起)。该模型的重要特征如表 5.4 所示。

表 5.4 Many-to-few 线程特性

优 点	缺 点
能在一个进程中利用多处理器硬件的优势	比其他模型复杂
大多数环境在用户模式切换(速度快)	程序员在核实体上失去直接的控制, 因为线程的优先级可能仅仅在用户模式中有意义
可扩展性好, 进程可以为每个物理处理器使用一个核实体, 或者更多一些	
在系统服务的阻塞期间延迟很小	

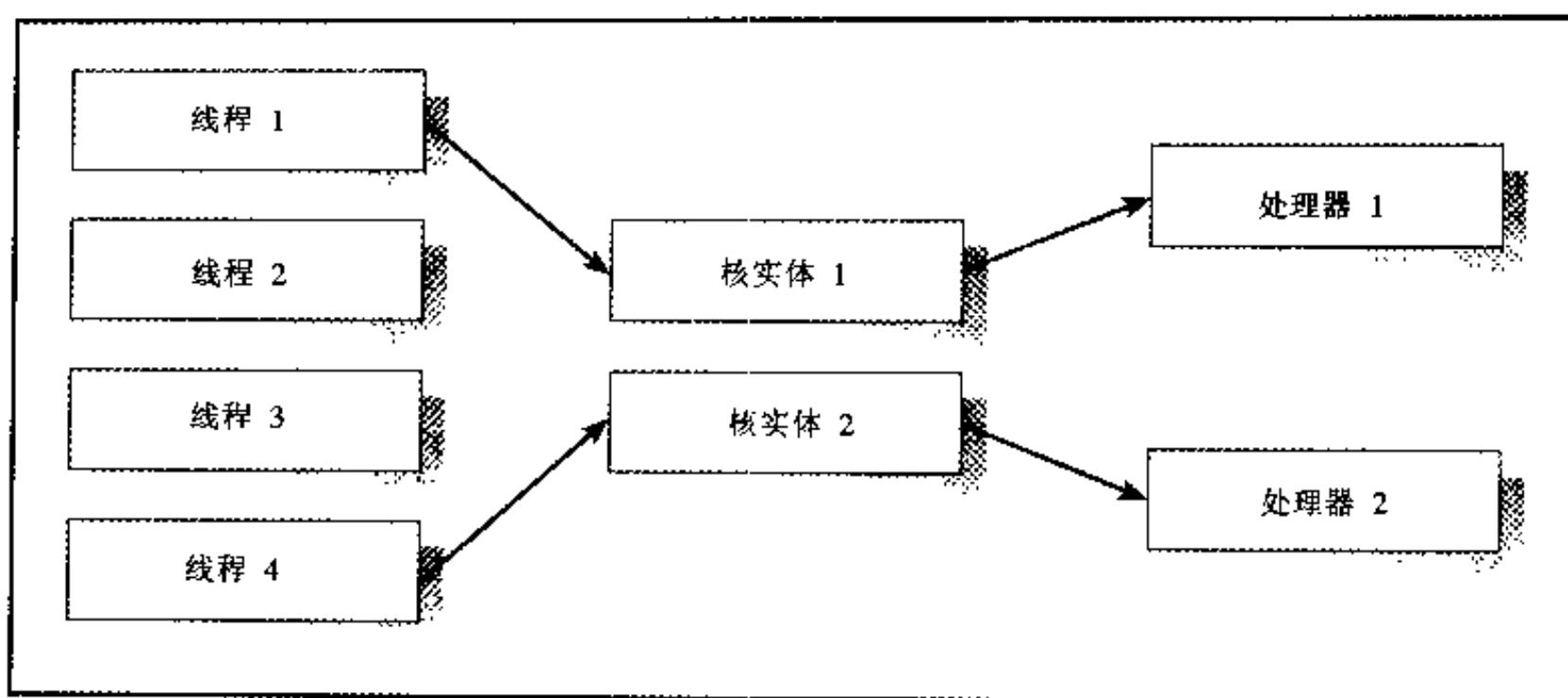


图 5.7 多对少线程图

6

POSIX 针对线程的调整

*“Who are you?” said the Caterpillar.
This was not an encouraging opening for a conversation.
Alice replied, rather shyly, “I-I hardly know, Sir,
just at present—at least /know who/was when/got up this morning, but
/think/must have been changed several times since then.”*

—Lewis Carroll, Alice's Adventures in Wonderland

Pthreads 改变了很多传统型的 POSIX 进程函数的含义。大多数变化是明显的，并且即使标准没有增加特定的说明你也可能感受到。例如，当一个线程因为 I/O 被阻塞时，只有调用线程被阻塞，而进程内的其他线程可以继续运行。

但是，还有另外一类 POSIX 的函数没有这样相当明白地扩展到线程的世界。例如，当你 fork 一个多线程进程时，在线程身上会发生什么呢？exec（执行外部程序）函数在一个多线程的进程中会做什么呢？当多线程进程中的一条线程调用 exit 退出时，又会发生什么？

6.1 fork

除非你打算很快地 exec 一个新程序，否则应避免（如果你能的话）在一个多线程的程序中使用 fork。

当多线程进程调用 fork 创造子进程时，Pthreads 指定只有那个调用 fork 的线程在子进程中存在。尽管当从 fork 调用返回时，只有调用线程在子进程中存在，所有其他的 Pthreads 线程状态仍保留为与调用 fork 时相同的状态。在子进程中，线程拥有与在父进程中相同的状态。它拥有相同的互斥量，同样的线程私有数据键值等。尽管当调用 fork 时在同步对象上等待的任何线程不再等待，所有的互斥量和条件变量仍然存在（它们不在子进程中存在，所以它们怎么能等待呢？）。

在一个 forked 的进程中，Pthreads 不会终止其他线程，好像它们调用

`pthread_exit` 退出了或是好像它们被取消。它们只是简单地不再存在，即，线程不再运用线程私有数据 `destructors` 或清除处理函数。如果子进程准备调用 `exec` 运行一个新程序，这不是一个问题，但是如果你使用 `fork` 克隆一个多线程程序，注意你可能失去对存储器的存取，特别是仅仅存储线程私有数据值的堆存储器。

| `fork` 调用不会影响互斥量的状态。如果它在父进程中被锁住，则它在子进程中被锁！

如果一个互斥量在 `fork` 调用时被锁，则它在子进程中仍然被锁。因为一个加锁的互斥量被锁住它的线程拥有，只有锁住互斥量的线程是调用 `fork` 的那个线程时，互斥量可以在子进程中被开锁。这是重要的如果当你调用 `fork` 时另外的线程把一个互斥量锁住，则你将失去对该互斥量和由该互斥量控制的任何数据的存取。

尽管复杂，你可以 `fork` 一个继续运行甚至继续使用 Pthreads 的子进程。必须小心地使用 `fork` 处理函数来保护你的互斥量和由互斥量保护的共享数据。`fork` 处理函数将在 6.1.1 节讨论。

因为没有调用线程私有数据销毁和清除处理函数，你可能需要担心存储泄漏问题。一个可能的办法是在 `prepare` `fork` 处理函数中取消你的子系统创造的线程，并且等待它们终止（当返回时）才允许 `fork` 继续运行；然后，在 `fork` 完成以后，在 `parent` `fork` 处理函数（详细解释见 6.1.1）中创建新线程。这很容易变得凌乱，所以我不推荐它为一个好的解决方案。相反，我还是如本节开始时警告的那样：应避免在线程的代码中使用 `fork`，除非子进程想很快地 `exec` 一个新程序。

POSIX 指定了一个小的函数集合，它们可以被安全地从信号处理函数（“异步信号安全”的函数）内调用，其中包括 `fork`。然而，没有一个 POSIX 线程函数是异步信号安全的（并且有很好的理由如此，因为异步信号安全的函数通常开销更大）。不过，随着 `fork` 处理器的引入，对 `fork` 的调用也就是对一组 `fork` 处理器的调用。

`fork` 处理器的目的是允许线程的代码越过 `fork` 调用保护同步状态和数据不变量，而在大多数情况下都需要加锁互斥量，但是不能从信号处理函数中加锁互斥量。因此尽管从一个信号处理函数内调用 `fork` 是合法的，这样做可能需要执行一些无法在信号处理函数内部执行的额外操作（超出了调用者的控制或知识范围）。

这是 POSIX 标准需要在将来修订的一个矛盾，还没有人知道最终的答案会是什么。我的忠告是避免在信号处理函数内使用 `fork`。

6.1.1 `fork` 处理器

```
int pthread_atfork (void (*prepare) (void),
                    void (*parent) (void), void (*child) (void));
```

Pthreads 增加了 `pthread_atfork` “fork 处理器”机制以允许你的代码越过 `fork` 调用保护数据和不变量。这与 `atexit` 有点类似，后者在一个进程终止时允许程序执行清除操作。使用 `pthread_atfork`，你需要提供三个独立的处理函数地址。`prepare fork` 处理器在父进程调用 `fork` 之前调用，`parent fork` 处理器在 `fork` 执行后在父进程中被调用，`child fork` 处理器在 `fork` 执行后在子进程中被调用。

| 如果编写一个使用互斥量并且不建立 fork 处理器的子系统，则该子系统在 `fork` 调用后在子进程中将不能正确工作。

通常，`prepare fork` 处理器以正确的顺序锁住所有的由相关代码（对于一个库或一个应用程序而言）使用的互斥量以阻止死锁的发生。调用 `fork` 的线程将在 `prepare fork` 处理器中阻塞直到它锁住了所有的互斥量后，这保证了其他线程不能锁住某个互斥量或修改子进程可能需要的数据。`parent fork` 处理器只需要开锁所有互斥量即可，以允许父进程和所有线程继续正常工作。

`child fork` 处理器经常可以与 `parent fork` 处理器一样；但是有时需要重置程序或库的状态。例如，如果使用 `daemon` 线程在后台执行函数，你或者需要记录那些线程不再存在的事实，或者在子进程中创建新线程来执行同样的函数。你可能需要重置计数器、释放堆存储等。

| 你的 `fork` 处理器与其他人的 `fork` 处理器一样好。

当任何线程调用 `fork` 时，系统将运行所有在进程中声明的 `prepare fork` 处理器。如果正确地编码 `prepare fork` 和 `child fork` 处理器，原则上你将能在子进程中继续操作。但是如果某人没有提供 `fork` 处理器或没有提供正确的处理器，该怎么办？例如，在一个多线程系统上的 ANSI C 库，必须使用一套互斥量同步内部的数据，如 `stdio` 文件流。

例如，如果你使用一个不提供 `fork` 处理器（来为 `fork` 调用准备合适的互斥量）的 ANSI C 库，则有时你会发现当它调用 `printf` 时，你的子进程可能挂起，因为当你的线程调用 `fork` 时，父进程中的其他线程锁住了互斥量。对于这类问题，通常你除了记录系统问题之外没有什么可以做的。这些互斥量通常对库而言是私有的，并且对你的代码不可见。你不能在 `prepare` 处理器或在调用 `fork` 前加锁它们。

程序 `atfork.c` 显示了如何使用 `fork` 处理器。当不带参数运行时，或者运行一个非零参数时，程序将安装 `fork` 处理器。当运行一个零参数时，如 `atfork 0`，它将不安装 `fork` 处理器。

在安装 `fork` 处理器后，结果会生成两行 `fork` 调用的报告结果，报告当前进程的 `pid`。如果没有 `fork` 处理器，子进程将被创建，同时起始线程拥有互斥量。因为起始线程不在而子进程存在，互斥量不能被开锁，所以子进程将被挂起，只有父进程能打印消息。

被 fork 调用。该函数改变的任何状态（特别是被锁住的互斥量）将被拷贝进子进程。fork_prepare 函数锁住程序的互斥量。

31~42 函数 fork_parent 是 parent 处理器。在创建子进程以后，它将在父进程中被 fork 调用。总的来说，一个 parent 处理器应该取消在 prepare 处理器中做的处理，以便父进程能正常继续。fork_parent 函数解锁被 fork_prepare 锁住的互斥量。

48~60 函数 fork_child 是 child 处理器。它将在子进程中被 fork 调用。在大多数情况下，child 处理器需要执行在 fork_parent 处理器中做过的处理，解锁状态以便子进程能继续运行。它可能也需要执行附加的清除操作，例如，fork_child 设置 self_pid 变量为子进程的 pid，同时解锁进程互斥量。

65~91 在创建子进程以后，它将继续执行 thread_routine 代码，thread_routine 函数锁住互斥量。当运行 fork 处理器时，fork 调用将被阻塞（当 prepare 处理器锁住互斥量时）直到互斥量可用。没有 fork 处理器，线程将在主函数解锁互斥量前调用 fork，并且线程将在这个点上在子进程中挂起。

99~106 除非程序运行参数 0，否则主程序声明 fork 处理器。

108~123 主程序在创建将调用 fork 的线程之前锁住互斥量。然后，它睡眠若干秒以保证当互斥量被锁住时，线程能够调用 fork，然后解锁互斥量。运用 thread_routine 的线程将总是在父进程中成功，因为它将简单地阻塞直到主程序释放锁。

然而，如果没有 fork 处理器，则子进程将在互斥量被锁住时被创建。锁住互斥量的线程（主线程）不在子进程内存在，所以不能在子进程中解锁互斥量。只有当拥有互斥量的线程调用 fork 时，互斥量才能在子进程中被解锁。fork 处理器提供了确保这种要求的最好方法。

■ atfork.c

```

1 #include <sys/types.h>
2 #include <pthread.h>
3 #include <sys/wait.h>
4 #include "errors.h"
5
6 pid_t self_pid;                      /* pid of current process */
7 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
8
9 /*
10  * This routine will be called prior to executing the fork,
11  * within the parent process.
12  */
13 void fork_prepare (void)
14 {
15     int status;
16
17     /*
18      * Lock the mutex in the parent before creating the child,
19      * to ensure that no other thread can lock it (or change any

```

```
20     * associated shared state) until after the fork completes.
21     */
22     status = pthread_mutex_lock (&mutex);
23     if (status != 0)
24         err_abort (status, "Lock in prepare handler");
25 }
26
27 /*
28  * This routine will be called after executing the fork, within
29  * the parent process
30 */
31 void fork_parent (void)
32 {
33     int status;
34
35     /*
36      * Unlock the mutex in the parent after the child has been
37      * created.
38      */
39     status = pthread_mutex_unlock (&mutex);
40     if (status != 0)
41         err_abort (status, "Unlock in parent handler");
42 }
43
44 /*
45  * This routine will be called after executing the fork, within
46  * the child process.
47 */
48 void fork_child (void)
49 {
50     int status;
51
52     /*
53      * Update the file scope "self_pid" within the child process, and
54      * unlock the mutex.
55      */
56     self_pid = getpid ();
57     status = pthread_mutex_unlock (&mutex);
58     if (status != 0)
59         err_abort (status, "Unlock in child handler");
60 }
61
62 /*
63  * Thread start routine, which will fork a new child process.
64 */
65 void *thread_routine (void *arg)
66 {
67     pid_t child_pid;
68     int status;
69
70     child_pid = fork ();
71     if (child_pid == (pid_t)-1)
72         errno_abort ("Fork");
73
74     /*
75      * Lock the mutex -- without the atfork handlers, the mutex will
```

```
76     * remain locked in the child process and this lock attempt will
77     * hang (or fail with EDEADLK) in the child.
78     */
79     status = pthread_mutex_lock (&mutex);
80     if (status != 0)
81         err_abort (status, "Lock in child");
82     status = pthread_mutex_unlock (&mutex);
83     if (status != 0)
84         err_abort (status, "Unlock in child");
85     printf ("After fork: %d (%d)\n", child_pid, self_pid);
86     if (child_pid != 0) {
87         if ((pid_t)-1 == waitpid (child_pid, (int*)0, 0))
88             errno_abort ("Wait for child");
89     }
90     return NULL;
91 }
92
93 int main (int argc, char *argv[])
94 {
95     pthread_t fork_thread;
96     int atfork_flag = 1;
97     int status;
98
99     if (argc > 1)
100         atfork_flag = atoi (argv[1]);
101     if (atfork_flag) {
102         status = pthread_atfork (
103             fork_prepare, fork_parent, fork_child);
104         if (status != 0)
105             err_abort (status, "Register fork handlers");
106     }
107     self_pid = getpid ();
108     status = pthread_mutex_lock (&mutex);
109     if (status != 0)
110         err_abort (status, "Lock mutex");
111     /*
112     * Create a thread while the mutex is locked. It will fork a
113     * process, which (without atfork handlers) will run with the
114     * mutex locked.
115     */
116     status = pthread_create (
117         &fork_thread, NULL, thread_routine, NULL);
118     if (status != 0)
119         err_abort (status, "Create thread");
120     sleep (5);
121     status = pthread_mutex_unlock (&mutex);
122     if (status != 0)
123         err_abort (status, "Unlock mutex");
124     status = pthread_join (fork_thread, NULL);
125     if (status != 0)
126         err_abort (status, "Join thread");
127     return 0;
128 }
```

现在，想像你正在写一个管理网络服务器连接的函数库，并且你为每个网络连接创建一个线程来接收服务请求。在 `prepare fork` 处理器中，你锁住库的所有互斥量以保证子进程的状态是一致的和可恢复的。在 `parent fork` 处理器中，你解锁那些互斥量然后返回。当设计 `child fork` 处理器时，你需要确切地决定 `fork` 对于库而言意味着什么。如果你想要在子进程中保留所有的网络连接，则你将为每个连接创造一个新的接收线程并且在释放互斥量前在合适的数据结构中记录它们的标识符。如果你希望子进程在没有连接打开的情况下开始，则你应该查找现存的父进程连接数据结构然后释放它们，并关闭被 `fork` 调用复制的相关文件。

6.2 exec

`exec` 函数没有因为引入线程而受到很多影响。`exec` 函数的功能是消除当前程序的环境并且用一个新程序代替它。对 `exec` 的调用，将很快地终止进程内除调用 `exec` 的线程外的所有线程。它们不执行清除处理器或线程私有数据 `destructors`——线程只是简单地停止存在。

所有的同步对象也消失了，除了 `pshared` 互斥量（使用 `PTHREAD_PROCESS_SHARED` 属性值创建的互斥量）和 `pshared` 条件变量。因为只要共享内存被一些进程映射，后者仍然是有用的。然而，你应该解锁当前进程可能锁住的任何 `pshared` 互斥量——系统不会为你解锁它们。

6.3 进程结束

在一个非线程程序中，对于 `exit` 函数的显式调用和从程序主函数返回有一样的效果，指进程退出。`Pthreads` 增加了 `pthread_exit` 函数，该函数能在进程继续运行的同时导致单个线程的退出。

在一个多线程程序中，主函数是“进程主线程的启动函数”。尽管从任何其他线程的启动函数返回就像调用 `pthread_exit` 终止线程一样，但是从主函数返回将终止整个进程。与进程相关的所有内存（和线程）将消失。线程不会执行清除处理器或线程私有数据 `destructors` 函数。调用 `exit` 具有同样的效果。

当你不想使用起始线程或让它等待其他线程结束时，可以通过调用 `pthread_exit` 而非返回或调用 `exit` 退出主函数。从主函数中调用 `pthread_exit` 将在不影响进程内其他线程的前提下终止起始线程，允许它们继续和正常完成。

`exit` 函数提供了一个简单的方法停止整个进程。例如，如果一个线程判定数据一定已经被一些错误严重地破坏，则允许程序继续操作数据将是危险的。当程序

某种程度上被破坏时，试图干净地停止应用线程可能是危险的。在这种情况下，你可以调用 `exit` 来很快地停止所有的处理。

6.4 stdio

Pthreads 要求 ANSI C 标准 I/O (stdio) 函数是线程安全的。因为 stdio 包需要为输出缓冲区和文件状态指定静态存储区，stdio 实现将使用互斥量或信号灯等同步机制。

6.4.1 flockfile 和 funlockfile

```
void flockfile (FILE *file);
int ftrylockfile (FILE *file);
void funlockfile (FILE *file);
```

在一些情况里，一系列 stdio 操作以不被中断的顺序执行是重要的。例如，一个提示符后面跟着一个从终端的读操作，或者需要一起在输出文件出现的两个写操作，即使其他线程试图在两个 stdio 调用之间写数据。因此，Pthreads 增加了锁住一个文件的机制并且规定文件锁如何与 stdio 内部锁交互。为了向 `stdout` 写提示符号并从 `stdin` 读响应，而不让其他线程在两个操作之间读 `stdin` 或者写 `stdout`，你应该在两个调用前后锁住 `stdin` 和 `stdout`，如下列程序 `flock.c` 所示。

19~20 这是重要的部分：分别为两个文件流执行了一次 `flockfile` 调用。为了在 stdio 内避免可能的死锁问题，Pthreads 推荐当你必须锁住两个文件流时，总是在输出流之前锁住输入流。这是很好的建议，所以我这样做了。

29~30 当然，对 `funlockfile` 的两个调用必须以相反的顺序执行。尽管标准调用让你有效地在 stdio 库中锁住互斥量，还是应该考虑一个一致的锁层次。

■ flock.c

```
1 #include <pthread.h>
2 #include "errors.h"
3
4 /*
5  * This routine writes a prompt to stdout (passed as the thread's
6  * "arg"), and reads a response. All other I/O to stdin and stdout
7  * is prevented by the file locks until both prompt and fgets are
8  * complete.
9 */
10 void *prompt_routine (void *arg)
11 {
12     char *prompt = (char*)arg;
```

```
13     char *string;
14     int len;
15
16     string = (char*)malloc (128);
17     if (string == NULL)
18         errno_abort ("Alloc string");
19     flockfile (stdin);
20     flockfile (stdout);
21     printf (prompt);
22     if (fgets (string, 128, stdin) == NULL)
23         string[0] = '\0';
24     else {
25         len = strlen (string);
26         if (len > 0 && string[len-1] == '\n')
27             string[len-1] = '\0';
28     }
29     funlockfile (stdout);
30     funlockfile (stdin);
31     return (void*)string;
32 }
33
34 int main (int argc, char *argv[])
35 {
36     pthread_t thread1, thread2, thread3;
37     void *string;
38     int status;
39
40 #ifdef sun
41     /*
42      * On Solaris 2.5, threads are not timesliced. To ensure
43      * that our threads can run concurrently, we need to
44      * increase the concurrency level.
45     */
46     DPRINTF (("Setting concurrency level to 4\n"));
47     thr_setconcurrency (4);
48 #endif
49     status = pthread_create (
50         &thread1, NULL, prompt_routine, "Thread 1> ");
51     if (status != 0)
52         err_abort (status, "Create thread");
53     status = pthread_create (
54         &thread2, NULL, prompt_routine, "Thread 2> ");
55     if (status != 0)
56         err_abort (status, "Create thread");
57     status = pthread_create (
58         &thread3, NULL, prompt_routine, "Thread 3> ");
59     if (status != 0)
60         err_abort (status, "Create thread");
61     status = pthread_join (thread1, &string);
62     if (status != 0)
63         err_abort (status, "Join thread");
64     printf ("Thread 1: \"%s\"\n", (char*)string);
65     free (string);
66     status = pthread_join (thread2, &string);
```

```

67     if (status != 0)
68         err_abort (status, "Join thread");
69     printf ("Thread 1: \"%s\"\n", (char*)string);
70     free (string);
71     status = pthread_join (thread3, &string);
72     if (status != 0)
73         err_abort (status, "Join thread");
74     printf ("Thread 1: \"%s\"\n", (char*)string);
75     free (string);
76     return 0;
77 }

```

■ flock.c

你也能使用 `flockfile` 和 `funlockfile` 函数来确保一系列写操作不会被从其他线程的文件存取打断。`ftrylockfile` 函数在试图锁住文件方面像 `pthread_mutex_trylock` 一样工作，并且如果文件已经被锁住，将返回错误状态而不是阻塞。

6.4.2 getchar_unlocked 和 putchar_unlocked

```

int getc_unlocked (FILE *stream);
int getchar_unlocked (void);
int putc_unlocked (int c, FILE *stream);
int putchar_unlocked (int c);

```

ANSI C 提供了向 `stdio` 缓冲区高效地读取和写入单个字符的函数。函数 `getchar` 和 `putchar` 分别操作 `stdin` 和 `stdout`，而 `getc` 和 `putc` 能在任何 `stdio` 文件流上被使用。这些函数传统上被作为宏实现以获得最大的性能，直接读或写文件流数据缓冲区。然而，Pthreads 要求这些函数锁住 `stdio` 流数据来防止代码对 `stdio` 缓冲区的偶然破坏。

加锁和开锁互斥量的开销极有可能将超过执行字符拷贝花费的时间，因此这些函数不再是高效的。Pthreads 本可以定义新函数以提供锁的变化而非重新定义现存函数，然而，结果将是现存代码可能在线程中无法安全地使用。工作组最终确定：更好的方法是让现存代码慢一些，而不是使它不正确。

Pthreads 增加了新函数来代替旧的高效宏，这些函数本质上与传统的宏实现一样。函数 `getc_unlocked`、`putc_unlocked`、`getchar_unlocked` 和 `putchar_unlocked` 不执行任何锁操作，因此你必须在这些操作附近使用 `flockfile` 和 `funlockfile`。如果你想要读或写单个字符，通常应该使用加锁的变量而非锁住文件流，调用新的解锁的 `get` 或 `put` 函数，然后解锁文件流。

如果想要执行一系列快速字符存取，以前你可能使用 `getchar` 和 `putchar` 函数，现在你可以使用 `getchar_unlocked` 和 `putchar_unlocked`。下列程序，`putchar.c`，显示了使用 `putchar` 和在一个文件锁内调用一系列 `putchar_`

unlocked 之间的差别。

- 9~20 当程序运行一个非零参数或没有参数时，它将运行 lock_routine 函数创造线程。该函数锁住 stdout 文件流，然后将参数（一个字符串）写到 stdout，每调用 putchar_unlocked 一次写入一个字符。
- 28~37 当程序运行零参数时，它调用 unlock_routine 函数创造线程。该函数将参数写到 stdout 中，每个字符使用 putchar 写入。尽管 putchar 内部被同步以确保 stdio 缓冲区不被破坏，但是单个字符可以以任意顺序出现。

■ putchar.c

```
1 #include <pthread.h>
2 #include "errors.h"
3
4 /*
5  * This function writes a string (the function's arg) to stdout,
6  * by locking the file stream and using putchar_unlocked to write
7  * each character individually.
8 */
9 void *lock_routine (void *arg)
10 {
11     char *pointer;
12
13     flockfile (stdout);
14     for (pointer = arg; *pointer != '\0'; pointer++) {
15         putchar_unlocked (*pointer);
16         sleep (1);
17     }
18     funlockfile (stdout);
19     return NULL;
20 }
21
22 /*
23  * This function writes a string (the function's arg) to stdout,
24  * by using putchar to write each character individually.
25  * Although the internal locking of putchar prevents file stream
26  * corruption, the writes of various threads may be interleaved.
27 */
28 void *unlock_routine (void *arg)
29 {
30     char *pointer;
31
32     for (pointer = arg; *pointer != '\0'; pointer++) {
33         putchar (*pointer);
34         sleep (1);
35     }
36     return NULL;
37 }
38
39 int main (int argc, char *argv[])
40 {
41     pthread_t thread1, thread2, thread3;
42     int flock_flag = 1;
```

```

43     void *(*thread_func)(void *);
44     int status;
45
46     if (argc > 1)
47         flock_flag = atoi (argv[1]);
48     if (flock_flag)
49         thread_func = lock_routine;
50     else
51         thread_func = unlock_routine;
52     status = pthread_create (
53         &thread1, NULL, thread_func, "this is thread 1\n");
54     if (status != 0)
55         err_abort (status, "Create thread");
56     status = pthread_create (
57         &thread2, NULL, thread_func, "this is thread 2\n");
58     if (status != 0)
59         err_abort (status, "Create thread");
60     status = pthread_create (
61         &thread3, NULL, thread_func, "this is thread 3\n");
62     if (status != 0)
63         err_abort (status, "Create thread");
64     pthread_exit (NULL);
65 }

```

■ putchar.c

6.5 线程安全的函数

尽管 ANSI C 和 POSIX 1003.1-1990 在开发时没有考虑到线程，它们定义的大多数函数可以在不改变外部接口的前提下变为线程安全的。例如，尽管 malloc 和 free 必须被改为支持线程，调用这些函数的代码却不必知道该变化。当你调用 malloc 时，它锁住一个互斥量（或者若干互斥量）来执行操作，或可能使用其他等价的同步机制。而你的代码只是与以前一样调用 malloc，并且该函数像平时一样实现相同的功能。

对于下列两类主要函数，这不是真的：

- 传统地返回指向内部静态缓冲区指针的函数，如asctime。使用内部互斥量不会有所帮助，因为在函数返回以后，调用者将在某时（在互斥量被解锁之后）读取格式化的时间字符串。
- 在一系列调用之间要求静态环境的函数，例如 strtok，它在一个本地静态变量中保存 token 串内的当前位置。对于 strtok，在其内部使用互斥量同样不会有所帮助，因为其他线程能在两个调用之间覆盖当前位置。

在这些情况下，Pthreads 定义了现存函数的线程安全的变体，它们在相应函数名结尾处添加后缀_r，这些变体在调用者的控制下将环境移出库外。当每个线程使用一个私有缓冲区或环境时，函数是线程安全的。如果你想要，你也能在线程之间共享环境——但是调用者必须在线程之间提供同步。如果想要两个线程并行地搜

索一个目录，必须同步它们对传给 `readdir_r` 函数的共享 `struct dirent` 的使用。

一些现存函数，如 `ctermid`，只要对参数做某些限制就已经是线程安全的。这些限制在下面各节论述。

6.5.1 用户和终端 ID

```
int getlogin_r (char *name, size_t namesize);
char *ctermid (char *s);
int ttyname_r (int fildes,
    char *name, size_t namesize);
```

这些函数将数据返回到一个调用者指定的缓冲区中。对于函数 `getlogin_r`，`namesize` 必须至少是 `LOGIN_NAME_MAX` 个字符。对于 `ttyname_r`，`namesize` 必须至少是 `TTY_NAME_MAX` 个字符。每个函数在成功时返回 0 值，在失败时返回一个错误数字。除了可能返回的错误 `getlogin` 和 `ttyname`，`getlogin_r` 和 `ttyname_r` 外，还可以返回 `ERANGE` 以表明名字缓冲区太小。

Pthreads 要求当在线程环境中使用 `ctermid`（它没有变化）时，返回参数必须是一个指向至少有 `L_ctermid` 个字节的字符缓冲区指针。好像该限制已经足够，不需要定义新的变体来指定缓冲区的大小。程序 `getlogin.c` 显示了如何调用这些函数。注意，这些函数不以任何方式依赖于线程或`<pthread.h>`，甚至可以在不支持线程的系统上被提供。

■ `getlogin.c`

```
1 #include <limits.h>
2 #include "errors.h"
3
4 /*
5  * If either TTY_NAME_MAX or LOGIN_NAME_MAX are undefined
6  * (this means they are "indeterminate" values), assume a
7  * reasonable size (for simplicity) rather than using sysconf
8  * and dynamically allocating the buffers.
9 */
10 #ifndef TTY_NAME_MAX
11 #define TTY_NAME_MAX 128
12#endif
13 #ifndef LOGIN_NAME_MAX
14 #define LOGIN_NAME_MAX 32
15#endif
16
17 int main (int argc, char *argv[])
18 {
19     char login_str[LOGIN_NAME_MAX];
20     char stdin_str[TTY_NAME_MAX];
21     char cterm_str[L_ctermid], *cterm_str_ptr;
```

```

22     int status;
23
24     status = getlogin_r (login_str, sizeof (login_str));
25     if (status != 0)
26         err_abort (status, "Get login");
27     cterm_str_ptr = ctermid (cterm_str);
28     if (cterm_str_ptr == NULL)
29         errno_abort ("Get cterm");
30     status = ttyname_r (0, stdin_str, sizeof (stdin_str));
31     if (status != 0)
32         err_abort (status, "Get stdin");
33     printf ("User: %s, cterm: %s, fd 0: %s\n",
34             login_str, cterm_str, stdin_str);
35     return 0;
36 }

```

■ getlogin.c

6.5.2 目录搜索

```

int readdir_r (DIR *dirp, struct dirent *entry,
               struct dirent **result);

```

该函数实质上与 `readdir` 执行一样的功能。即，它返回由 `dirp` 指定的目录流中的下一个目录入口。差别在于它不是返回指向那个入口的指针，而是将入口拷贝到由 `entry` 指定的缓冲区中。如果成功，它返回 0 并设置 `result` 指针指向 `entry` 缓冲区。如果搜索到目录流的结尾，它返回 0 并将 `result` 置空。如果失败，它返回一个如 `EBADF` 之类的错误数字。

为演示使用 `readdir_r` 来使线程并发地搜索多个目录，请参考 4.2 节的程序 `crew`。

6.5.3 字符串 token

```

char *strtok_r (
    char *s, const char *sep, char **lasts);

```

该函数返回字符串 `s` 中的下一个 `token`。不同于 `strtok`，环境（原来串内的当前指针）在 `lasts` 中保存，它由调用者指定，而非保存在函数内部的一个静态指针。

当第一次调用该函数时，参数 `s` 中保存了指向字符串的指针。在后续的函数调用中，为返回该字符串的后续 `token`，`s` 必须被指定为空。用 `strtok_r` 设置 `lasts` 值来保存函数在字符串内的位置，并且在随后的每个调用中你必须返回同样的 `lasts` 值。`strtok_r` 函数返回一个指向下一个 `token` 的指针，或当没有更多的 `token` 时返回 `NULL`。

6.5.4 时间表示

```
char *asctime_r (const struct tm *tm, char *buf);
char *ctime_r (const time_t *clock, char *buf);
struct tm *gmtime_r (
    const time_t *clock, struct tm *result);
struct tm *localtime_r (
    const time_t *clock, struct tm *result);
```

输出缓冲区 (buf 和 result) 由调用者提供, 而不是返回一个指向函数内部的静态存储指针。否则, 它们和传统型的变体相同。asctime_r 和 ctime_r 例程返回表示系统时间的 ASCII 字符, 二者都要求 buf 参数指向一个至少 26 个字节的字符串。

6.5.5 随机数产生

```
int read_r (unsigned int *seed);
```

种子被维持在调用者提供的存储 (seed) 而非使用函数内部的静态存储。该接口的主要问题是: 在一个程序内让所有的应用和库代码共享单个种子通常是不实际的。结果是, 应用程序和各个库通常将产生分离的随机数流。这样, 即使没有创建线程 (创建线程可能将 rand 调用的顺序改变, 因此不管怎样会改变结果), 使用 rand_r 代替 rand 的程序很可能产生不同的结果。

6.5.6 组和用户数据库

组数据库:

```
int getrgid_r (
    gid_t gid, struct group *grp, char *buffer,
    size_t bufsize, struct group **result);
int getrnam_r (
    const char *name, struct group *grp,
    char *buffer, size_t bufsize,
    struct group **result);
```

用户数据库:

```
int getpwuid_r (
    uid_t uid, struct passwd *pwd, char *buffer,
    size_t bufsize, struct passwd **result);
int getpwnam_r (
    const char *name, struct passwd *pwd,
    char *buffer, size_t bufsize,
    struct passwd **result);
```

这些函数针对特定的组或用户 (gid、uid 或 name)，将组或用户记录的一个拷贝（分别是 grp 或 pwd）保存到由参数 buffer 和 bufsize 指定的一个缓冲区中。在每种情况中，函数返回值或者为 0（成功），或者为一个错误数字（例如当缓冲区太小时返回 ERANGE）。如果请求的记录不在组或 passwd 数据库中，则函数可能返回成功但是将 result 指针置空。如果记录被发现并且缓冲区足够大，则 result 指针指向 buffer 内的 struct group 或 struct passwd 结构记录。

缓冲区要求的最大尺寸可以通过调用 sysconf 函数来设置，其中参数为 _SC_GETGR_R_SIZE_MAX（对于组数据）或参数为 _SC_GETPW_R_SIZE_MAX（对于用户数据）。

6.6 信号

*Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the JubJub bird, and shun
The frumious Bandersnatch!*
—Lewis Carroll, Through the Looking-Glass

Pthreads 信号处理模型的历史是标准中最容易让人糊涂的部分。有若干个不同的观点，并且要满足工作组中每个人的妥协设计是困难的（更不用说更大、更多样的投票组了）。因为信号是复杂的，所以这种状况不管怎样也不足为奇，并且在工业有广泛分布的历史。

有两个互相矛盾的主要目标：

- 首先是“信号应该与传统型的 UNIX 完全兼容”，既信号处理器和信号掩码应该与进程相关。因为信号的复杂语义使得信号很难与线程同时共存，所以实际上对于多线程信号几乎无用。任务应该是使用线程同步完成而非异步地使用信号。
- 其次是“信号应该与传统的 UNIX 完全兼容”，这里的“兼容”意味着信号处理器和信号掩码应该完全是线程私有的。大多数现存的 UNIX 代码实质上仍然有相同的作用，不管是在线程内运行还是在进程内运行。代码移植将被简化。

问题是“兼容”的定义本身就不兼容。尽管参与协商的许多人可能不同意最后的结果，但是几乎每个人都同意那些设计妥协的人们做了一个格外好的工作，并且他们相当勇敢地尝试了这一过程。

当编写多线程代码时，尽量要把信号当作 Jabberwocks-curious（无须好奇的）和有潜在危险的动物来小心使用。

避免与线程一起使用信号总是最好的。同时，将它们分开经常又是不可能或不实际的。当信号和线程遭遇时，就要注意了。只要有可能的话，仅仅在主线程内使用 `pthread_sigmask` 来屏蔽信号，然后同步地在专用线程中使用 `sigwait` 来处理信号。如果必须在线程内使用 `sigaction`（或等价的）来处理同步信号（如 `SIGSEGV`），要特别小心。在信号处理函数内应尽量少做的工作。

6.6.1 信号行为

所有的信号行为都是过程范围内的。一个程序必须在线程之间协调 `sigaction` 的任何使用。这是非模块化的，而且相对简单，信号从来不是模块化的。在一个多线程的系统上，要编写一个动态地修改信号行为（如当它执行浮点操作时处理或忽略 `SIGFPE` 信号，或者当它执行网络 I/O 时 `SIGPIPE`）的函数将是困难的。

尽管修改进程信号行为本身是线程安全的，但是不能防止其他线程随后很快地设置一个新的信号行为。即使代码试着通过“保存原来的信号行为随后恢复它”来做好这件事，它也可能被其他线程阻挠，如图 6.1 所示。

线 程 1	线 程 2	注 释
<code>sigaction(SIGFPE)</code>		线程1的信号行为激活
产生 <code>SIGFPE</code>	<code>sigaction(SIGFPE)</code>	线程2的信号行为激活
恢复信号行为	恢复信号行为	线程1的信号被线程2的信号行为处理（但是仍然在线程1的执行环境中） 线程1恢复原有的信号行为 线程2恢复线程1的信号行为——原有信号行为丢失

图 6.1 信号行为的非模块化

与“硬件环境”同步的信号包括 `SIGFPE`、`SIGSEGV` 和 `SIGTRAP`，它们被送到引起该硬件状况的线程，而绝不传给其他线程。

| 你不能通过传送一个 `SIGKILL` 信号来杀死一个线程，也不能通过传送一个 `SIGSTOP` 信号来停止一个线程。

当多线程工作时，任何影响进程的信号仍然影响进程，即意味着向进程或向进程内任何线程传送一个 `SIGKILL` 信号（`pthread_kill` 将在 6.6.3 节讨论）将终止进程。传送一个 `SIGSTOP` 信号将导致所有的线程停止直到收到 `SIGCONT` 信号。这保证了现存的进程处理函数能够继续工作——否则，当你传送一个 `SIGSTOP` 停止命令时，进程中的大多数线程可以继续运行。这也适用于其他信号的默认行为，如 `SIGSEGV`，如果不做处理，将终止进程并且产生一个核心文件——它将不仅仅

终止产生 SIGSEGV 的线程。

这对一个程序员而言意味着什么？与库代码不应该改变信号行为一样明智，这应该是主程序的范围。当你使用线程编程时，该思想甚至变得更加明智。信号行为必须总是在单一部件的控制下，让主程序承担该责任，这对于所有情况都有意义。

6.6.2 信号掩码 (mask)

```
int pthread_sigmask (int how,
                     const sigset_t *set, sigset_t *oset);
```

每个线程都有自己私有的信号掩码，可以通过调用 `pthread_sigmask` 来修改。`Pthreads` 不指定 `sigprocmask` 在一个多线程进程中做什么——它可以不做任何事情。可移植的线程代码不会调用 `sigprocmask`。一个线程可以阻塞发信号或解除阻塞而不影响其他线程的处理信号能力，这对于同步信号特别重要。如果线程 A 因为线程 B 正在处理自己的 SIGFPE，或者更坏的因为线程 C 阻塞了 SIGFPE 信号而不能处理 SIGFPE，则是十分尴尬的。当一个线程被创建时，它继承了创造它的线程的信号掩码——如果你想要在所有地方屏蔽一个信号，则首先在主线程中屏蔽它。

6.6.3 `pthread_kill`

```
int pthread_kill (pthread_t thread, int sig);
```

在一个进程中，一个线程可以通过调用 `pthread_kill` 向一个特定的线程（包括自己）发信号。当调用 `pthread_kill` 时，你不仅要指定要传送的信号数值，还要指定目标线程的 `pthread_t` 标识符。然而，你不能使用 `pthread_kill` 向进程外的线程传送信号，因为线程标识符（`pthread_t`）仅在创建它的进程中才有意义。

`pthread_kill` 传送的信号像任何其他信号一样被处理。如果“目标”线程被信号屏蔽，它将被标记为未解决。如果线程正在等待在 `sigwait` 中的信号（见 6.6.4 节），线程将收到信号。如果线程没有将信号屏蔽，并且没在 `sigwait` 中阻塞，当前的信号行为将被执行。

记住，除了信号处理函数外，信号行为也影响进程。使用 `pthread_kill` 向一特定线程传送 SIGKILL 信号将杀死整个进程，而不仅是指定的线程。使用 `pthread_cancel` 来删除一个特定线程（见 5.3 节）。向一个线程传送 SIGSTOP 信号将停止进程内所有的线程直到其他的进程传送来一个 SIGCONT 信号。

ANSI C 指定的 `raise` 函数已经传统地被映射到杀死当前进程的函数。即，

`raise (SIGABRT)` 通常和 `kill (getpid(), SIGABRT)` 一样。

对于多线程而言，调用 `raise` 的代码很可能打算将信号送到调用的线程，而不是到一些进程内的任意线程。Pthreads 指定 `raise (SIGABRT)` 与 `pthread_kill` 一样 (`pthread_self()`, `SIGABRT`)。

下列程序 `susp.c` 使用 `pthread_kill` 实现了一个可移植的“推迟和恢复”（或者说是“推迟和继续”）能力，很像由 Solaris 提供的“UI 线程”接口：`thr_suspend` 和 `thr_continue`^①。当你使用一个线程的 `pthread_t` 调用 `thd_suspend` 函数，则当函数返回时，指定的线程被推迟执行。线程不能执行直到使用一样的 `pthread_t` 调用 `thd_continue` 以后。

对于已经被推迟线程的推迟请求没有任何效果。对于一个推迟的线程调用一次 `thd_continue` 将使它恢复执行，即使它被多次 `thd_suspend` 调用推迟。而当前没有被推迟的线程调用 `thd_continue` 没有任何效果。

“推迟和继续”一般用来解决一些问题，如多线程垃圾收集，并且如果程序员很小心，甚至可以工作一段时间。所以，对于“推迟和继续”的模拟可能对确实需要这些功能的少数程序员十分珍贵。然而，要注意的是，如果你推迟一个占有资源（如一个互斥量）的线程，将容易导致应用程序的死锁。

6 标志 `ITERATIONS` 定义了目标线程循环的次数。如果该值太小，在主线程能够如愿地推迟和继续它们之前，一些或所有线程就将终止。如果发生这种情况，程序将报告一个错误消息而失败——增加 `ITERATIONS` 的值直到问题消失。

12 变量 `sentinel` 被用来在信号处理函数和其他线程之间同步。你可能会不相信地问：“噢”？该机制不是完美的——推迟的线程（调用 `thd_suspend` 的线程）在一个循环中等待，让出处理器直到 `sentinel` 改变状态。`volatile` 存储属性保证信号处理函数将变量值写到内存^②。记住，你不能在信号处理函数内使用互斥量。

22~40 `suspend_signal_handler` 函数将作为“推迟”信号 `SIGUSR1` 的信号处理函数被建立。它初始化一个信号掩码来屏蔽除了 `SIGUSR2`（继续信号）外的所有信号，然后通过调用 `sigsuspend` 等待那个信号。在推迟自己以前，置位 `sentinel` 变量以通知推迟线程它不再执行用户代码——因为最实际的原因，它已经被推迟。

在信号处理函数与 `thd_suspend` 之间的同步最有用目的是：调用 `thd_suspend` 的线程一定要能够知道目标线程何时已经成功地推迟了。简单地调用 `pthread_kill` 是不够的，因为系统可能较长一段时间内不交付信号；我们需要知道信号什么时候被收到了。

① 程序 `susp.c` 的算法（和大部分代码）是有我的合作者 Brian Silver 开发的。为演示需要在这儿给了简化版的代码。

② 正如在后面的 6.6.6 节中所描述的那样，信号灯会提供更清晰、更安全的同步。函数 `thd_suspend` 将使用初始值 0 在信号灯上调用 `sem_wait`，信号捕获函数将调用 `sem_post` 来唤醒信号灯。

47~51 resume_signal_handler 函数作为“继续”信号 SIGUSR1 的信号处理函数被建立。该函数不是很重要，因为信号仅仅被送来中断 suspend_signal_handler 中对 sigsuspend 的调用。

■ susp.c	part1 signal-catchingfunctions
----------	--------------------------------

```
1 #include <pthread.h>
2 #include <signal.h>
3 #include "errors.h"
4
5 #define THREAD_COUNT      20
6 #define ITERATIONS        40000
7
8 unsigned long thread_count = THREAD_COUNT;
9 unsigned long iterations = ITERATIONS;
10 pthread_mutex_t the_mutex = PTHREAD_MUTEX_INITIALIZER;
11 pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
12 volatile sig_atomic_t sentinel = 0;
13 pthread_once_t once = PTHREAD_ONCE_INIT;
14 pthread_t *array = NULL, null_pthread = {0};
15 int bottom = 0;
16 int initied = 0;
17
18 /*
19  * Handle SIGUSR1 in the target thread, to suspend it until
20  * receiving SIGUSR2 (resume).
21  */
22 void
23 suspend_signal_handler (int sig)
24 {
25     sigset_t signal_set;
26
27     /*
28      * Block all signals except SIGUSR2 while suspended.
29      */
30     sigfillset (&signal_set);
31     sigdelset (&signal_set, SIGUSR2);
32     sentinel = 1;
33     sigsuspend (&signal_set);
34
35     /*
36      * Once I'm here, I've been resumed, and the resume signal
37      * handler has been run to completion.
38      */
39     return;
40 }
41
42 /*
43  * Handle SIGUSR2 in the target thread, to resume it. Note that
44  * the signal handler does nothing. It exists only because we need
45  * to cause sigsuspend() to return.
46  */
47 void
```

```

48 resume_signal_handler (int sig)
49 {
50     return;
51 }

```

■ susp.c part 1 signal-catchingfunctions

当第一次调用 thd_suspend 时, suspend_init_routine 函数动态地初始化推迟/恢复包。它实际上由 pthread_once 间接地调用。

- 15~16 它分配一个线程标识符的初始数组, 用来记录被推迟的所有线程的标识符。该数组被用来保证对于 thd_suspend 的多次调用不会对目标线程产生附加的效果, 以及调用 thd_continue 对于没被推迟的线程没有效果。
- 21~35 它为 SIGUSR1 和 SIGUSR2 信号建立信号行为, 它们将被分别使用来推迟和恢复线程的执行。

■ susp.c

part 2 initialization

```

1 /*
2  * Dynamically initialize the "suspend package" when first used
3  * (called by pthread_once).
4 */
5 void
6 suspend_init_routine (void)
7 {
8     int status;
9     struct sigaction sigusr1, sigusr2;
10
11    /*
12     * Allocate the suspended threads array. This array is used
13     * to guarantee idempotency
14     */
15    bottom = 10;
16    array = (pthread_t*) calloc (bottom, sizeof (pthread_t));
17
18    /*
19     * Install the signal handlers for suspend/resume.
20     */
21    sigusr1.sa_flags = 0;
22    sigusr1.sa_handler = suspend_signal_handler;
23    sigemptyset (&sigusr1.sa_mask);
24    sigaddset (&sigusr1.sa_mask, SIGUSR2);
25    sigusr2.sa_flags = 0;
26    sigusr2.sa_handler = resume_signal_handler;
27    sigemptyset (&sigusr2.sa_mask);
28
29    status = sigaction (SIGUSR1, &sigusr1, NULL);
30    if (status == -1)
31        errno_abort ("Installing suspend handler");
32
33    status = sigaction (SIGUSR2, &sigusr2, NULL);
34    if (status == -1)
35        errno_abort ("Installing resume handler");

```

```

36
37     initied = 1;
38     return;
39 }

```

■ susp.c

part 2 initialization

9~40 thd_suspend 函数推迟一个线程，并且当那个线程停止执行用户代码时返回。它首先保证推迟/恢复包已经通过调用 pthread_once 初始化。在一个互斥量的保护下，它在保存推迟的线程标识符的数组中查找目标线程的标识符。如果线程已经被推迟，thd_suspend 返回成功。

47~60 判断推迟线程数组中是否有一个空的入口，如果没有则 realloc 数组增加一个额外的入口。

65~78 变量 sentinel 被初始化为 0，以检测目标线程何时被推迟执行。线程首先调用 pthread_kill 传送一个 SIGUSR1 信号，循环调用 thd_suspend，然后调用 sched_yield 避免垄断一个处理器，直到目标线程通过设置 sentinel 做出响应。最后，推迟线程的标识符被存储到数组中，

■ susp.c

part 3 thd_suspend

```

1 /*
2  * Suspend a thread by sending it a signal (SIGUSR1), which will
3  * block the thread until another signal (SIGUSR2) arrives.
4  *
5  * Multiple calls to thd_suspend for a single thread have no
6  * additional effect on the thread -- a single thd_continue
7  * call will cause it to resume execution.
8 */
9 int
10 thd_suspend (pthread_t target_thread)
11 {
12     int status;
13     int i = 0;
14
15     /*
16      * The first call to thd_suspend will initialize the
17      * package.
18      */
19     status = pthread_once (&once, suspend_init_routine);
20     if (status != 0)
21         return status;
22
23     /*
24      * Serialize access to suspend, makes life easier.
25      */
26     status = pthread_mutex_lock (&mut);
27     if (status != 0)
28         return status;
29
30     /*
31      * Threads that are suspended are added to the target_array;

```

```
32     * a request to suspend a thread already listed in the array
33     * is ignored. Sending a second SIGUSR1 would cause the
34     * thread to resuspend itself as soon as it is resumed.
35     */
36     while (i < bottom)
37         if (pthread_equal (array[i++], target_thread)) {
38             status = pthread_mutex_unlock (&mut);
39             return status;
40         }
41
42     /*
43     * Ok, we really need to suspend this thread. So, let's find
44     * the location in the array that we'll use. If we run off
45     * the end, realloc the array for more space.
46     */
47     i = 0;
48     while (i < bottom && array[i] != 0)
49         i++;
50
51     if (i == bottom) {
52         array = (pthread_t*) realloc (
53             array, (++bottom * sizeof (pthread_t)));
54         if (array == NULL) {
55             pthread_mutex_unlock (&mut);
56             return errno;
57         }
58
59         array[bottom] = null_pthread; /* Clear new entry */
60     }
61
62     /*
63     * Clear the sentinel and signal the thread to suspend.
64     */
65     sentinel = 0;
66     status = pthread_kill (target_thread, SIGUSR1);
67     if (status != 0) {
68         pthread_mutex_unlock (&mut);
69         return status;
70     }
71
72     /*
73     * Wait for the sentinel to change.
74     */
75     while (sentinel == 0)
76         sched_yield ();
77
78     array[i] = target_thread;
79
80     status = pthread_mutex_unlock (&mut);
81     return status;
82 }
```

■ susp.c

part 3 thd_suspend

化。

33~39 如果指定的线程标识符在推迟线程数组中没被发现，则它没有被推迟，函数再次返回成功。

45~51 传送恢复信号 SIGUSR2。没有需要等待——无论何时，如有可能线程将恢复运行，调用 thd_continue 的线程不需要知道。

■ susp.c

part 4 thd_continue

```

1  /*
2   * Resume a suspended thread by sending it SIGUSR2 to break
3   * it out of the sigsuspend() in which it's waiting. If the
4   * target thread isn't suspended, return with success.
5   */
6  int
7  thd_continue (pthread_t target_thread)
8  {
9      int status;
10     int i = 0;
11
12     /*
13      * The first call to thd_continue will initialize the
14      * package.
15      */
16     status = pthread_once (&once, suspend_init_routine);
17     if (status != 0)
18         return status;
19
20     /*
21      * Serialize access to suspend/resume, to make life easier
22      */
23     status = pthread_mutex_lock (&mut);
24     if (status != 0)
25         return status;
26 }
27
28 /*
29  * Make sure the thread is in the suspend array. If not, it
30  * hasn't been suspended (or it has already been resumed) and
31  * we can just carry on.
32  */
33     while (i < bottom && !pthread_equal (array[i], target_thread))
34         i++;
35
36     if (i == bottom) {
37         pthread_mutex_unlock (&mut);
38         return 0;
39     }
40
41 /*
42  * Signal the thread to continue, and remove the thread from
43  * the suspended array.
44  */

```

```

45     status = pthread_kill (target_thread, SIGUSR2);
46     if (status != 0) {
47         pthread_mutex_unlock (&mut);
48         return status;
49     }
50
51     array[i] = 0;           /* Clear array element */
52     status = pthread_mutex_unlock (&mut);
53     return status;
54 }
```

susp.c**part 4 thd_continue**

2~25 `thread_routine` 函数是由程序创建的各个目标线程的起始函数。它简单地循环一段时间，周期性地打印一条状态消息。每次重复，它让位给其他线程以保证处理器时间“越过所有线程”公平分配。

注意，函数没有使用 `printf`，而是使用 `sprintf` 格式化一条消息然后由调用 `write` 函数在 `stdout`（文件描述符 1）上显示它。这说明了使用推迟和恢复（`thd_suspend` 和 `thd_continue`）进行同步的问题之一。推迟和恢复是调度函数，而不是同步函数，并且同时使用调度和同步控制会产生严重的后果。

| 不小心使用推迟和恢复可能死锁你的应用程序。

在这种情况下，如果当线程修改 `stdio` 流时被推迟，那试图修改该 `stdio` 流的所有其他线程可能被阻塞，等待由被推迟的线程锁住的某个互斥量。另一方面，`(write)` 函数通常是到内核的调用——与信号相比，内核是原子的，因此不会被推迟。所以，使用 `write` 函数不会引起死锁。

总的来说，你不能推迟一个可能拥有任何资源的线程（如果该资源可能被一些其他线程在推迟线程被恢复之前要求）。特别地，如果负责将推迟线程恢复的线程首先需要获得资源，结果就是死锁。特别地，该禁止包括由你调用的库使用的互斥量——如 `malloc` 和 `free` 函数使用的互斥量或者由 `stdio` 使用的互斥量。

36~42 使用一个属性对象集创建可分离的线程，而非可连接的线程。结果是一旦线程终止，它们将停止存在，而不是保留到主线程调用 `pthread_join`。如果你试图向一个终止线程传送信号，`pthread_kill` 函数不一定确实失败（标准在这点上是沉默的），并且你可能仅仅将一个未解决的信号放在一个决不会对它起作用的线程中。如果这种情况发生，`thd_suspend` 函数将挂起等待线程响应。尽管向一个终止线程发信号时，`pthread_kill` 可能不失败，但是当向一个不存在的线程发信号时，函数将失败——所以当程序使用太低的 `ITERATIONS` 设置运行时，该特性将把可能遇到的挂起转换为进入一条错误消息，而程序失败。

51~85 主线程在创建线程后睡眠 2 秒以允许它们到达一个“稳定的状态”。它然后循环遍历前一半线程，逐个推迟它们。接着等待另一个 2 秒钟，然后逐个恢复它推迟的线程。它再等待 2 秒，逐个推迟留下的线程（后一半），然后在的 2 秒后恢复它们。通过查看由各个线程打印的状态消息，你能看见当线程被推迟和恢复时输出模

式的变化。

```
■ susp.c                                         part 5    sampleprogram

---

1 static void *  
2 thread_routine (void *arg)  
3 {  
4     int number = (int)arg;  
5     int status;  
6     int i;  
7     char buffer[128];  
8  
9     for (i = 1; i <= iterations; i++) {  
10         /*  
11             * Every time each thread does 2000 interations, print  
12             * a progress report.  
13             */  
14         if (i % 2000 == 0) {  
15             sprintf (  
16                 buffer, "Thread %02d: %d\n",  
17                 number, i);  
18             write (1, buffer, strlen (buffer));  
19         }  
20  
21         sched_yield ();  
22     }  
23  
24     return (void *)0;  
25 }  
26  
27 int  
28 main (int argc, char *argv[])  
29 {  
30     pthread_t threads[THREAD_COUNT];  
31     pthread_attr_t detach;  
32     int status;  
33     void *result;  
34     int i;  
35  
36     status = pthread_attr_init (&detach);  
37     if (status != 0)  
38         err_abort (status, "Init attributes object");  
39     status = pthread_attr_setdetachstate (  
40         &detach, PTHREAD_CREATE_DETACHED);  
41     if (status != 0)  
42         err_abort (status, "Set create-detached");  
43  
44     for (i = 0; i < THREAD_COUNT; i++) {  
45         status = pthread_create (  
46             &threads[i], &detach, thread_routine, (void *)i);  
47         if (status != 0)  
48             err_abort (status, "Create thread");  
49     }  
50 }
```

```

51     sleep (2);
52
53     for (i = 0; i < THREAD_COUNT/2; i++) {
54         printf ("Suspending thread %d.\n", i);
55         status = thd_suspend (threads[i]);
56         if (status != 0)
57             err_abort (status, "Suspend thread");
58     }
59
60     printf ("Sleeping ... \n");
61     sleep (2);
62
63     for (i = 0; i < THREAD_COUNT/2; i++) {
64         printf ("Continuing thread %d.\n", i);
65         status = thd_continue (threads[i]);
66         if (status != 0)
67             err_abort (status, "Suspend thread");
68     }
69
70     for (i = THREAD_COUNT/2; i < THREAD_COUNT; i++) {
71         printf ("Suspending thread %d.\n", i);
72         status = thd_suspend (threads[i]);
73         if (status != 0)
74             err_abort (status, "Suspend thread");
75     }
76
77     printf ("Sleeping ... \n");
78     sleep (2);
79
80     for (i = THREAD_COUNT/2; i < THREAD_COUNT; i++) {
81         printf ("Continuing thread %d.\n", i);
82         status = thd_continue (threads[i]);
83         if (status != 0)
84             err_abort (status, "Continue thread");
85     }
86
87     pthread_exit (NULL);          /* Let threads finish */
88 }
```

■ susp.c

part 5 sampleprogram

6.6.4 sigwait 和 sigwaitinfo

```

int sigwait (const sigset_t *set, int *sig);
#ifndef _POSIX_REALTIME_SIGNALS
int sigwaitinfo (
    const sigset_t *set, siginfo_t *info);
int sigtimedwait (
    const sigset_t *set, siginfo_t *info,
    const struct timespec *timeout);
#endif
```

| 在多线程代码中总是使用 `sigwait` 与异步信号一起工作。

Pthreads 增加了允许线程代码同步地处理异步信号的函数。即不允许信号在任意点上打断一个线程，线程能够选择同步地接收信号。通过调用 `sigwait` 或 `sigwait` 的某个兄弟函数来实现该目的。

| 调用 `sigwait` 等待的信号必须在调用线程中被屏蔽，并且通常应该在所有的线程中被屏蔽。

`sigwait` 函数使用一个信号集作为它的参数，并且在集合中的任一信号发生时返回一个信号值。可以创建等待一些信号（如 `SIGINT` 信号）的一个线程，并且当它发生时执行一些应用程序活动。隐含的规则是必须在调用 `sigwait` 前屏蔽要等待的信号。事实上，理想地情况是你应该在主线程中、在程序的开始屏蔽这些信号。因为信号掩码会被创建的线程继承，所以所有的线程将（默认地）屏蔽相应的信号。这保证信号决不会被送到除调用 `sigwait` 的任何其他线程。

信号仅仅被交付一次。如果两个线程在 `sigwait` 上阻塞，只有一个线程将收到送给进程的信号。这意味着你不能让两个独立的子系统使用 `sigwait` 捕获 `SIGINT`。这也意味着信号不会被一个线程的 `sigwait` 捕获并且送到另外线程的信号处理函数。这也不是特别糟糕，因为你在旧的非线程模型中同样做不到——一次只有一个信号行为可以是激活的。

`sigwait` 作为一个 Pthreads 函数，通过返回一个错误数字来报告错误；而它的兄弟函数 `sigwaitinfo` 和 `sigtimedwait` 在 Pthreads 以前就被加到 POSIX 中，它们使用旧的 `errno` 机制。这点迷惑人和让人尴尬，而且是不幸的。问题是因为它们在 Pthreads 修订之前就处理完了 POSIX 提供的关于实时信号选项（在 `<unistd.h>` 中定义标志 `_POSIX_REALTIME_SIGNALS`）的附加信息以及 POSIX 实时修订版 POSIX.1b。

函数 `sigwaitinfo` 和 `sigtimedwait` 都返回收到信号的实时信号信息 `siginfo_t`。另外，`sigtimedwait` 允许调用者指定：如果在指定的间隔内没有收到选定信号，则返回 `EAGAIN` 错误。

程序 `sigwait.c` 创建了一个处理 `SIGINT` 信号的“`sigwait` 线程”。

23~41 `signal_waiter` 线程重复地调用 `sigwait`，等待一个 `SIGINT` 信号。它记数 5 次 `SIGINT` 的出现（每次打印一条消息），然后向一个主线程正在等待的条件变量上发信号。此时，主线程将退出。

61~65 主程序开始时屏蔽 `SIGINT` 信号。因为所有的线程将从它们的创造者那儿继承起始信号掩码，`SIGINT` 将在所有的线程中被屏蔽。这避免了 `SIGINT` 信号在除 `signal_waiter` 线程调用 `sigwait` 并准备好接收信号外的任何其他时间被交付。

■ sigwait.c

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <pthread.h>
4 #include <signal.h>
5 #include "errors.h"
6
7 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
8 pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
9 int interrupted = 0;
10 sigset_t signal_set;
11
12 /*
13  * Wait for the SIGINT signal. When it has occurred 5 times, set the
14  * "interrupted" flag (the main thread's wait predicate) and signal a
15  * condition variable. The main thread will exit.
16 */
17 void *signal_waiter (void *arg)
18 {
19     int sig_number;
20     int signal_count = 0;
21     int status;
22
23     while (1) {
24         sigwait (&signal_set, &sig_number);
25         if (sig_number == SIGINT) {
26             printf ("Got SIGINT (%d of 5)\n", signal_count+1);
27             if (++signal_count >= 5) {
28                 status = pthread_mutex_lock (&mutex);
29                 if (status != 0)
30                     err_abort (status, "Lock mutex");
31                 interrupted = 1;
32                 status = pthread_cond_signal (&cond);
33                 if (status != 0)
34                     err_abort (status, "Signal condition");
35                 status = pthread_mutex_unlock (&mutex);
36                 if (status != 0)
37                     err_abort (status, "Unlock mutex");
38                 break;
39             }
40         }
41     }
42     return NULL;
43 }
44
45 int main (int argc, char *argv[])
46 {
47     pthread_t signal_thread_id;
48     int status;
49
50 /*
51  * Start by masking the "interesting" signal, SIGINT in the
52  * initial thread. Because all threads inherit the signal mask
53  * from their creator, all threads in the process will have
```

```

54     * SIGINT masked unless one explicitly unmasks it. The
55     * semantics of sigwait requires that all threads (including
56     * the thread calling sigwait) have the signal masked, for
57     * reliable operation. Otherwise, a signal that arrives
58     * while the sigwaiter is not blocked in sigwait might be
59     * delivered to another thread.
60     */
61     sigemptyset (&signal_set);
62     sigaddset (&signal_set, SIGINT);
63     status = pthread_sigmask (SIG_BLOCK, &signal_set, NULL);
64     if (status != 0)
65         err_abort (status, "Set signal mask");
66
67     /*
68     * Create the sigwait thread.
69     */
70     status = pthread_create (&signal_thread_id, NULL,
71         signal_waiter, NULL);
72     if (status != 0)
73         err_abort (status, "Create sigwaiter");
74
75     /*
76     * Wait for the sigwait thread to receive SIGINT and signal
77     * the condition variable.
78     */
79     status = pthread_mutex_lock (&mutex);
80     if (status != 0)
81         err_abort (status, "Lock mutex");
82     while (!interrupted) {
83         status = pthread_cond_wait (&cond, &mutex);
84         if (status != 0)
85             err_abort (status, "Wait for interrupt");
86     }
87     status = pthread_mutex_unlock (&mutex);
88     if (status != 0)
89         err_abort (status, "Unlock mutex");
90     printf ("Main terminating with SIGINT\n");
91     return 0;
92 }
```

■ sigwait.c

6.6.5 SIGEV_THREAD

POSIX.1b 实时标准中用来异步通知的一些函数允许程序员给出关于通知如何被完成的明确指示。例如，当使用 `aio_read` 或 `aio_write` 初始化一台异步设备读或写时，程序员指定一个 `struct aiocb`，其中包含了一个 `struct sigevent` 成员。接受 `struct sigevent` 的其他函数包括 `timer_create`（它创造一个进程范围内定时器）和 `sigqueue`（它将信号送到进程队列）。

在 POSIX.1b 中，`struct sigevent` 提供了一种允许程序员指定一个信号是否被产生以及如果产生应该使用什么信号数字的“通知机制”。Pthreads 增加了被

称为 SIGEV_THREAD 的新通知机制。该通知机制使得信号通知函数像线程起始函数一样运行。

Pthreads 把若干成员加到 POSIX.1b `struct sigevent` 结构中。新成员包括 `sigev_notify_function`（一个指向线程起始函数的指针）和 `sigev_notify_attributes`（一个指向包含了期望的线程创建属性的线程属性对象 `pthread_attr_t` 的指针）。如果 `sigev_notify_attributes` 为空，与将 `detachstate` 属性设置为 `PTHREAD_CREATE_DETACHED` 一样创建通知线程。这避免了内存泄漏——一般地讲，通知线程的标识符并不是对任何其他线程来讲都可得到的。而且，Pthreads 认为将 `detachstate` 属性集合设置为 `PTHREAD_CREATE_JOINABLE` 的结果是“不可知的”（结果很可能将是一个内存泄漏，因为线程是不可连接的——如果你幸运，系统可以忽视你的选择，并不管怎样创建可分离的线程）。

SIGEV_THREAD 通知函数可能无法在一个新的线程中实际运行——Pthreads 很小心地规定它的表现就像在一个新线程中运用一样。例如，系统可以排队 SIGEV_THREAD 事件，在一些内部的“服务线程”中连续地调用起始函数。这种差别对于应用程序而言很难有效地区分。使用服务线程的系统必须很小心地为通知线程指定属性——例如，调度策略和优先级、竞争范围和最小栈空间都必须仔细考虑。

对于任何“传统的”信号产生机制，如 `setitimer`、`SIGCHLD`、`SIGINT` 等，SIGEV_THREAD 特性是不可用的。那些使用 POSIX.1b 实时信号接口（包括定时器和异步的 I/O）编程的人可能发现该新功能有用。

下列程序 `sigev_thread.c` 显示了怎么为一个 POSIX.1b 定时器使用 SIGEV_THREAD 通知机制。

20~37 函数 `timer_thread` 作为 SIGEV_THREAD 定时器的“通知函数”（线程起始函数）。每次定时器到期时都会调用该函数。它记录到期的次数，在 5 次以后唤醒主线程。注意，不同于信号处理函数，SIGEV_THREAD 通知函数能充分利用 Pthreads 同步操作。这在许多情况下是一个很大的优点。

45~51 不幸地，Solaris 2.5 和 Digital UNIX 4.0 都没有正确地实现 SIGEV_THREAD。因此，不同于本书中所有其他例子，该代码将不在 Solaris 2.5 上编译。`#ifdef` 块允许代码编译，并且如果程序被运行，则以一条错误消息体面地失败。尽管程序将在 Digital UNIX 4.0 上编译，但它不会运行。SIGEV_THREAD 的实现 in Digital UNIX 4.0D 被修订了，当你读本书时，它应该已经可以得到它，而且它应该也在 Solaris 2.6 上被修订。

56~59 这些语句初始化 `sigevent` 结构，该结构描述当一个事件发生时系统应该如何通知应用程序。在本例中，当定时器到期时我们告诉它使用默认属性调用 `timer_thread`。

■ sigev_thread.c

```
1 #include <pthread.h>
2 #include <sys	signal.h>
3 #include <sys/time.h>
4 #include "errors.h"
5
6 timer_t timer_id;
7 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
8 pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
9 int counter = 0;
10
11 /*
12  * Thread start routine to notify the application when the
13  * timer expires. This routine is run "as if" it were a new
14  * thread, each time the timer expires.
15  *
16  * When the timer has expired 5 times, the main thread will
17  * be awakened, and will terminate the program.
18  */
19 void
20 timer_thread (void *arg)
21 {
22     int status;
23
24     status = pthread_mutex_lock (&mutex);
25     if (status != 0)
26         err_abort (status, "Lock mutex");
27     if (++counter >= 5) {
28         status = pthread_cond_signal (&cond);
29         if (status != 0)
30             err_abort (status, "Signal condition");
31     }
32     status = pthread_mutex_unlock (&mutex);
33     if (status != 0)
34         err_abort (status, "Unlock mutex");
35
36     printf ("Timer %d\n", counter);
37 }
38
39 int main (int argc, char *argv[])
40 {
41     int status;
42     struct itimerspec ts;
43     struct sigevent se;
44
45 #ifdef sun
46     fprintf (
47         stderr,
48         "This program cannot compile on Solaris 2.5.\n"
49         "To build and run on Solaris 2.6, remove the\n"
50         "\"#ifdef sun\" block in main().\n");
51 #else
```

```
52     /*
53      * Set the sigevent structure to cause the signal to be
54      * delivered by creating a new thread.
55      */
56     se.sigev_notify = SIGEV_THREAD;
57     se.sigev_value.sival_ptr = &timer_id;
58     se.sigev_notify_function = timer_thread;
59     se.sigev_notify_attributes = NULL;
60
61     /*
62      * Specify a repeating timer that fires every 5 seconds.
63      */
64     ts.it_value.tv_sec = 5;
65     ts.it_value.tv_nsec = 0;
66     ts.it_interval.tv_sec = 5;
67     ts.it_interval.tv_nsec = 0;
68
69     DPRINTF (( "Creating timer\n" ));
70     status = timer_create(CLOCK_REALTIME, &se, &timer_id);
71     if (status == -1)
72         errno_abort ("Create timer");
73
74     DPRINTF ((
75         "Setting timer %d for 5-second expiration...\n", timer_id));
76     status = timer_settime(timer_id, 0, &ts, 0);
77     if (status == -1)
78         errno_abort ("Set timer");
79
80     status = pthread_mutex_lock (&mutex);
81     if (status != 0)
82         err_abort (status, "Lock mutex");
83     while (counter < 5) {
84         status = pthread_cond_wait (&cond, &mutex);
85         if (status != 0)
86             err_abort (status, "Wait on condition");
87     }
88     status = pthread_mutex_unlock (&mutex);
89     if (status != 0)
90         err_abort (status, "Unlock mutex");
91
92 #endif /* Sun */
93     return 0;
94 }
```

■ sigev_thread.c

6.6.6 信号灯:与信号捕获函数同步

```
#ifdef _POSIX_SEMAPHORES
int sem_init (sem_t *sem,
    int pshared, unsigned int value);
```

```

int sem_destroy (sem_t *sem);
int sem_wait (sem_t *sem);
int sem_trywait (sem_t *sem);
int sem_post (sem_t *sem);
int sem_getvalue (sem_t *sem, int *sval);
#endif

```

尽管互斥量和条件变量为大多数同步需要提供了一个理想的答案,但它们不能满足所有的需要。这样的一个例子就是在 POSIX 信号处理函数与等待异步事件的线程之间通信的需要。在新的代码中,最好使用 `sigwait` 或 `sigwaitinfo` 而非依靠信号处理函数,这就干脆地避免了上述问题。然而,使用异步的 POSIX 信号处理函数已经很普遍,并且目前大多数程序员将可能遇见这种多线程代码。

从 POSIX 信号处理函数功能唤醒一个线程,你需要 POSIX 信号重入机制(异步信号安全的)。POSIX 提供了很少的相应函数,并且没有包括任何 Pthreads 函数。这主要是因为一个异步信号安全的互斥量锁操作将比不是异步信号安全的互斥量锁操作慢许多倍。在内核之外使函数成为异步信号安全的,通常要求函数在运行时屏蔽(阻塞)信号——这是昂贵的。

如果你好奇,下面是完整的异步信号安全的 POSIX 1003.1-1996 函数表(某些函数仅仅当特定 POSIX 选项被定义时才存在,如`_POSIX_ASYNCHRONOUS_IO`或`_POSIX_TIMERS`):

access	getoverrun	sigismember
aio_error	getgroups	sigpending
aio_return	getpgrp	sigprocmask
aio_suspend	getpid	sigqueue
alarm	getppid	sigsuspend
cfgetispeed	getuid	sleep
cfgetispeed	kill	stat
cfgetispeed	link	sysconf
cfgetospeed	lseek	tcdrain
chdir	mkdir	tcflow
chmod	mkfifo	tcflush
chown	open	tcgetattr
clock_gettime	pathconf	tcgetpgrp
close	pause	tcsendbreak
creat	pipe	tcsetattr
dup2	read	tcsetpgrp
dup	rename	time
execle	rmdir	timer_getoverrun
execve	sem_post	timer_gettime

_exit	setgid	timer_settime
fcntl	setpgid	times
fdatasync	setsid	umask
fork	setuid	uname
fstat	sigaction	unlink
fsync	sigaddset	utime
getegid	sigdelset	wait
geteuid	sigemptyset	waitpid
getgid	sigfillset	write

POSIX.1b 提供了计数信号灯，并且支持 Pthreads 的大多数系统也支持 POSIX.1b 信号灯。你可能注意到了 `sem_post` 函数（它唤醒在一个信号灯上等待的线程）也出现在异步信号安全的函数表中。如果你的系统支持 POSIX 信号灯（`<unistd.h>` 定义 `_POSIX_SEMAPHORES` 选项），则 Pthreads 增加了在进程内的线程间使用信号灯的能力。那意味着可以从一个 POSIX 信号处理函数内公布信号灯，来唤醒同一进程或不同进程中的一个线程。

一个信号灯是一种不同类型的同步对象——它有点像互斥量，又有点像条件变量。差别是：对于许多普通任务信号灯稍难使用，但是对于某些特定目的信号灯实质上更易使用。特别地，信号灯能从 POSIX 信号处理函数中被（解锁或发信号）。信号灯是通用的同步机制。

| 我们就是没有原因那样使用它们。

我是在强调使用信号灯从信号处理函数中传递信息，而并非为一般的使用。这样做的原因有其中一个原因是信号灯是不同标准的一个部分。我说过支持 Pthreads 的大多数系统将也支持 POSIX.1b，但是标准中任何地方也没有如此的要求。因此没有信号灯你也可能发现自己很好，并且你不应该感到要依赖于它们（当然，你也可以只要信号灯而没有线程——但是这种情况不是本书该讲述的内容）。

在这里与信号一起讲信号灯的另一个原因是，尽管信号灯是完全通用的同步机制，但是使用信号灯解决很多问题要比使用互斥量和条件变量更困难。如果你已经使用了 Pthreads，你仅仅需要使用信号灯来处理这个特定函数——从信号处理函数中唤醒一个等待线程。只要记住当它们易用并可得到时，你能使用它们做其他事情。

POSIX 信号灯包含一个计数量，但是没有“主人”，所以尽管它们实质上能作为锁使用，但它们也能被用来等待事件。POSIX 信号灯操作使用的术语也强调“等待”行为而非“锁”行为。但是不要被这些名字迷惑：在一个信号灯上“等待”与在信号灯上“加锁”之间没有差别。

一个线程通过调用 `sem_wait` 在一个信号灯上等待（以锁住一个资源或等待一个事件）。如果信号灯记数比零大，`sem_wait` 函数将记数器减 1 并马上返回；

否则，线程阻塞。一个线程能通过 `sem_post` 来发布（post）一个信号灯（开锁一个资源或唤醒等待线程）。如果一个以上的线程在信号灯上等待，`sem_post` 将唤醒其中一个（最高优先级或最早等待线程）。如果没有线程在等待，信号灯记数器将被加 1。

信号灯计数器的初始值可以区别“锁”信号灯和“等待”信号灯。通过创建一个初始值为 1 的信号灯，允许一个线程不必阻塞就完成 `sem_wait` 操作——这就是“锁”信号灯。通过创建一个初始值为 0 的信号灯，强迫调用 `sem_wait` 的线程阻塞直到其他线程调用 `sem_post`。

信号灯在其工作方式上的差别，为信号灯提供了两个胜过互斥量和条件变量的重要优点（可以在多线程程序中使用）：

1. 不同于互斥量，信号灯没有“主人”的概念。这意味着任何线程可以释放在一个信号灯上堵住的线程，就好像任何线程能释放某线程已锁住的互斥量一样（尽管通常这不是一个好的编程模型，不过有时它还是方便的）。
2. 不同于条件变量，信号灯能独立于任何外部的状态。条件变量依赖于一个共享的谓词和等待互斥量——信号灯不需要。

一个信号灯在程序中由一个 `sem_t` 类型变量表示。你从来不该拷贝一个 `sem_t` 变量——在 `sem_wait`、`sem_trywait`、`sem_post` 和 `sem_destroy` 中使用一个 `sem_t` 变量的拷贝，其结果是未定义的。针对我们的目的，一个 `sem_t` 变量通过调用 `sem_init` 函数被初始化。POSIX.1b 提供了另外的方法创建不通过共享内存而在进程间共享的“命名”信号灯，但是当在一个进程内使用信号灯时，就没有必要这样做。不同于 Pthreads 的其他函数，POSIX 信号灯函数使用 `errno` 报告错误。即，返回 0 表示成功，当有错误时返回 -1 并将 `errno` 变量设置为一个错误代码。

如果你有一个代码段，期望最多有两个线程同时在里面执行（而其他线程必须等待），可以只使用一个信号灯而不需任何附加状态。初始化信号灯值为 2；然后在代码段开始时调用 `sem_wait`，在结尾处调用 `sem_post`。然后，两个线程能够在信号灯上等待而不被阻塞，但是第三个线程将发现信号灯记数器为 0 并且阻塞。当每个线程退出代码区时，它信号灯，释放一等待线程（如果有的话）或恢复记数器。

如果没有线程等待，`sem_getvalue` 函数返回信号灯计数器的当前值。如果有线程正在等待，`sem_getvalue` 返回一个负数。该数字的绝对值说明有多少线程正在信号灯上等待。记住：它返回的值可能已经不正确了——因为该值可能由于其他线程的行为而随时变化。

使用 `sem_getvalue` 最好的方法是唤醒多个等待线程，有点像条件变量的广播。没有 `sem_getvalue`，你无法知道多少线程可能在一个信号灯上被阻塞。为了“广播”一个信号灯，可以在一个循环中调用 `sem_getvalue` 和 `sem_post` 直

到 `sem_getvalue` 报告没有更多的等待线程为止。

但是要记住，另外的线程可能在这个循环期间调用 `sem_post`，并且对于各种 `sem_post` 和 `sem_getvalue` 的并发调用之间没有同步。你最后可能轻易地做出额外的 `sem_post` 调用，这将使得下一个调用 `sem_wait` 的线程发现信号灯值大于 0 且很快地返回（没有阻塞）。

下面程序 `semaphore_signal.c` 使用一个信号灯从一个 POSIX 信号处理函数内唤醒线程。注意 `sem_init` 调用将初始值置 0 以便调用 `sem_wait` 的每个线程将阻塞，然后，主程序请求一个间隔定时器和 POSIX 信号处理函数（将调用 `sem_post` 唤醒一个等待线程）。POSIX 定时器信号的每次出现将唤醒一个等待线程。当每个线程被唤醒了 5 次时，程序将退出。

32~35 注意，检查 `EINTR` 信号的代码从 `sem_wait` 调用返回状态信息。当一个以上的线程在 `sem_wait` 上阻塞时，程序中的 POSIX 定时器信号将总会发生。当一个信号对进程发生时（如一个定时器信号），系统可以交付那个信号到进程内的任意线程的环境中。例如，可能的“牺牲品”包括内核知道的正在一个信号灯上等待的线程。因此，至少有时有相当好的机会选中调用 `sem_wait` 的线程。如果这样，对 `sem_wait` 的调用将返回 `EINTR`。线程然后必须再次调用 `sem_wait`。把返回 `EINTR` 当作“成功”，将使得每调用 `sem_post` 一次好像唤醒了两个线程：被打断的线程和被 `sem_post` 调用唤醒的线程。

■ `semaphore_signal.c`

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <pthread.h>
4 #include <semaphore.h>
5 #include <signal.h>
6 #include <time.h>
7 #include "errors.h"
8
9 sem_t semaphore;
10
11 /*
12  * Signal-catching function.
13  */
14 void signal_catcher (int sig)
15 {
16     if (sem_post (&semaphore) == -1)
17         errno_abort ("Post semaphore");
18 }
19
20 /*
21  * Thread start routine which waits on the semaphore.
22  */
23 void *sem_waiter (void *arg)
24 {
25     int number = (int)arg;
```

```
26     int counter;
27
28     /*
29      * Each thread waits 5 times.
30      */
31     for (counter = 1; counter <= 5; counter++) {
32         while (sem_wait (&semaphore) == -1) {
33             if (errno != EINTR)
34                 errno_abort ("Wait on semaphore");
35         }
36         printf ("%d waking (%d)...\\n", number, counter);
37     }
38     return NULL;
39 }
40
41 int main (int argc, char *argv[])
42 {
43     int thread_count, status;
44     struct sigevent sig_event;
45     struct sigaction sig_action;
46     sigset_t sig_mask;
47     timer_t timer_id;
48     struct itimerspec timer_val;
49     pthread_t sem_waiters[5];
50
51 #if !defined(_POSIX_SEMAPHORES) || !defined(_POSIX_TIMERS)
52 # if !defined(_POSIX_SEMAPHORES)
53     printf ("This system does not support POSIX semaphores\\n");
54 # endif
55 # if !defined(_POSIX_TIMERS)
56     printf ("This system does not support POSIX timers\\n");
57 # endif
58     return -1;
59 #else
60     sem_init (&semaphore, 0, 0);
61
62     /*
63      * Create 5 threads to wait on a semaphore.
64      */
65     for (thread_count = 0; thread_count < 5; thread_count++) {
66         status = pthread_create (
67             &sem_waiters[thread_count], NULL,
68             sem_waiter, (void*)thread_count);
69         if (status != 0)
70             err_abort (status, "Create thread");
71     }
72
73     /*
74      * Set up a repeating timer using signal number SIGRTMIN,
75      * set to occur every 2 seconds.
76      */
77     sig_event.sigev_value.sival_int = 0;
78     sig_event.sigev_signo = SIGRTMIN;
79     sig_event.sigev_notify = SIGEV_SIGNAL;
```

```
80     if (timer_create (CLOCK_REALTIME, &sig_event, &timer_id) == -1)
81         errno_abort ("Create timer");
82     sigemptyset (&sig_mask);
83     sigaddset (&sig_mask, SIGRTMIN);
84     sig_action.sa_handler = signal_catcher;
85     sig_action.sa_mask = sig_mask;
86     sig_action.sa_flags = 0;
87     if (sigaction (SIGRTMIN, &sig_action, NULL) == -1)
88         errno_abort ("Set signal action");
89     timer_val.it_interval.tv_sec = 2;
90     timer_val.it_interval.tv_nsec = 0;
91     timer_val.it_value.tv_sec = 2;
92     timer_val.it_value.tv_nsec = 0;
93     if (timer_settime (timer_id, 0, &timer_val, NULL) == -1)
94         errno_abort ("Set timer");
95
96     /*
97      * Wait for all threads to complete.
98      */
99     for (thread_count = 0; thread_count < 5; thread_count++) {
100         status = pthread_join (sem_waiters[thread_count], NULL);
101         if (status != 0)
102             err_abort (status, "Join thread");
103     }
104     return 0;
105 #endif
106 }
```

■ semaphore_signal.c

7

Real code

*"When we were still little," the Mock Turtle went on at last, more calmly,
though still sobbing a little now and then, "we went to school in the sea.
The master was an old Tortoise—we used to call him Tortoise—"
"Why did you call him Tortoise, if he wasn't one?" Alice asked.
"We called him Tortoise because he taught us," said the
Mock Turtle angrily.*

—Lewis Carroll, Through the Looking-Glass

本章内容构造在本书前几节的基础上，但主要是基于互斥量和条件变量两节。你应该已经理解如何创建两种类型的同步对象和它们如何工作。我将演示利用互斥量、条件变量和少量数据设计和构造 barrier 和读/写锁同步机制。barrier 和读/写锁是通用的，并且已被建议在不久的将来标准化。接着我将继续讨论一个在线程池中分配任务的工作排队服务器。

这一切的目的就是教你更多关于使用所有这些新的线程编程工具（即互斥量、条件变量和线程）的微妙之处。不过，在这里我放弃了一些东西，省略了一些可能在真实代码中有价值的复杂问题。例如，错误检测和恢复代码就是相当基本的。

7.1 扩展同步

互斥量和状况变量是灵活且有效的同步工具，使用它们可以构造所需的任何同步形式。但是不应该在每次需要它们时从头构造，最好是从不需要每次调试的、通用的模块化实现开始。本节显示了一些通用和有用的工具，你不用每次在写需要它们的应用程序时重新设计。

首先我们将构造一个 barrier。关于 barrier 的功能可能你已经猜到了——它用来停止线程。一个 barrier 被初始化来停止某些线程——当要求的线程数量到达 barrier 时，所有线程被允许继续运行。

然后我们将构造所谓的读/写锁。读/写锁允许多个线程同时读数据，但是禁止任何线程修改正在被其他线程读或修改的数据。

7.1.1 barriers

`barrier` 是将一组成员保持在一起的一种方式。例如，如果我们无畏的“舀程序员”被抛弃在一个荒芜的岛上，他们冒昧地进入丛林探索，他们将因为人多安全的想法希望待在一起（如图 7.1 所示）。任何发现自己领先其他人很远的程序员将在继续前进之前等待他们。

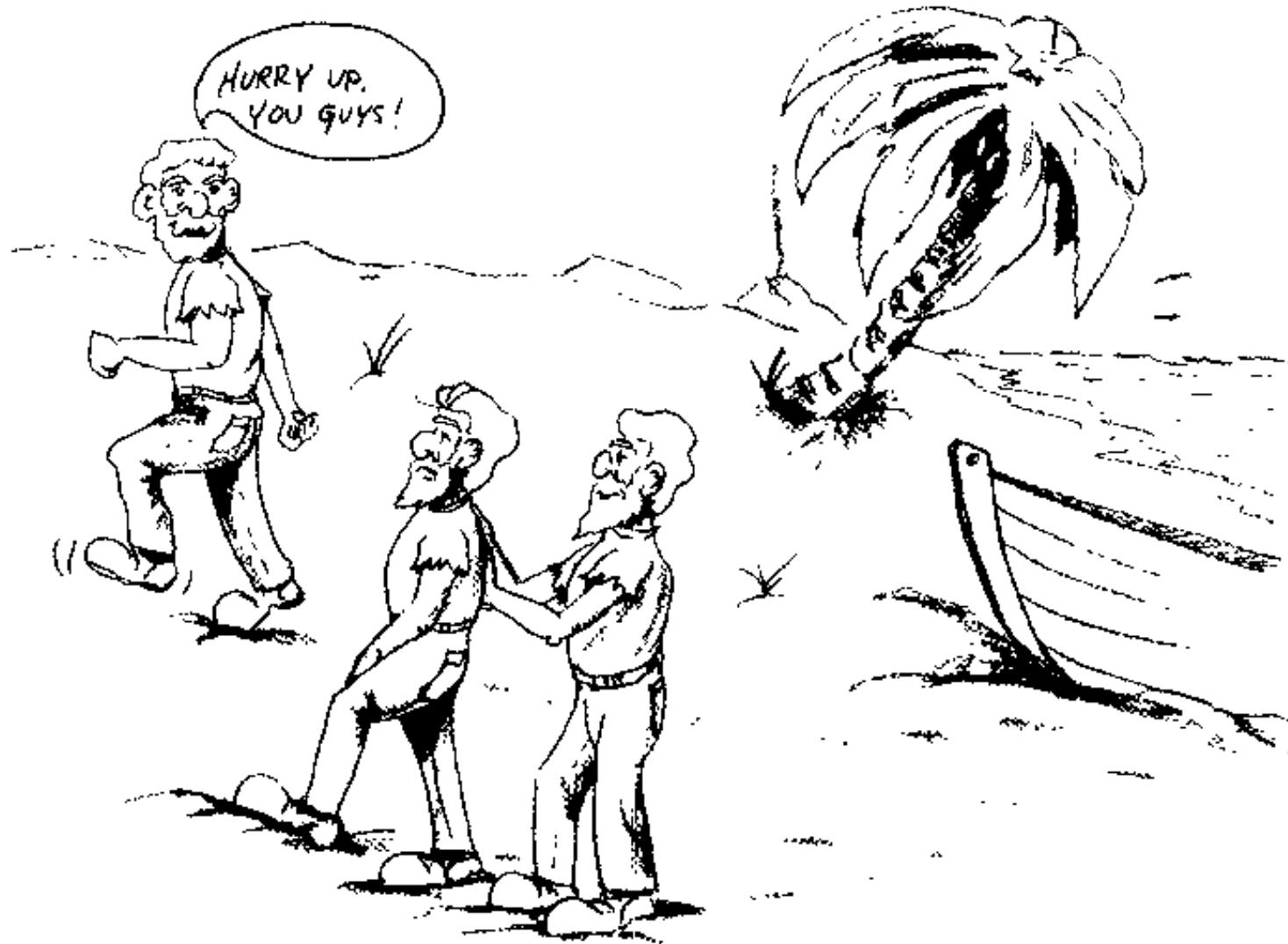


图 7.1 Barrier 类比

一个 `barrier` 通常被用来确保某些并行算法中的所有合作线程在任何线程可以继续运行之前到达算法中的一个特定点。这在以良好的并行自动分解的编译代码中尤其通用。所有的线程可以执行相同的代码，线程在一些代码段中处理共享数据集（如一个数组）的独立部分而在另外的代码段中并行地处理私有数据。但是，另外的代码段（如并发区域的安装或清除代码）还必须保证仅仅有一个线程执行。在这些区域之间的边界经常使用 `barrier` 来实现。这样，完成矩阵计算的线程可以在一个 `barrier` 等待直到所有线程完成。然后，一个线程可以为下一个并发区域执行安装操作，而其他的线程可以继续执行到下一个 `barrier`。当安装线到达那个 `barrier` 时，所有的线程开始进入下一个并发区域。

图 7.2 显示了一个用来同步三个线程（分别为线程 1、线程 2 和线程 3）操作的 `barrier`。本图是一个类时序图，时间从左向右递增。从左上方标签开始的各个线段表明一个特定线程的行为——实线表示线程 1，点线表示线程 2，虚线表示线程 3。当线段落入圆形矩形时，表示它们正在与 `barrier` 交互。如果线段在中心线以下，表明线程被阻塞，等待其他线程到达 `barrier`。在中心线上面停止的线段表示最后的线程

到达 barrier 并唤醒所有的等待线程。

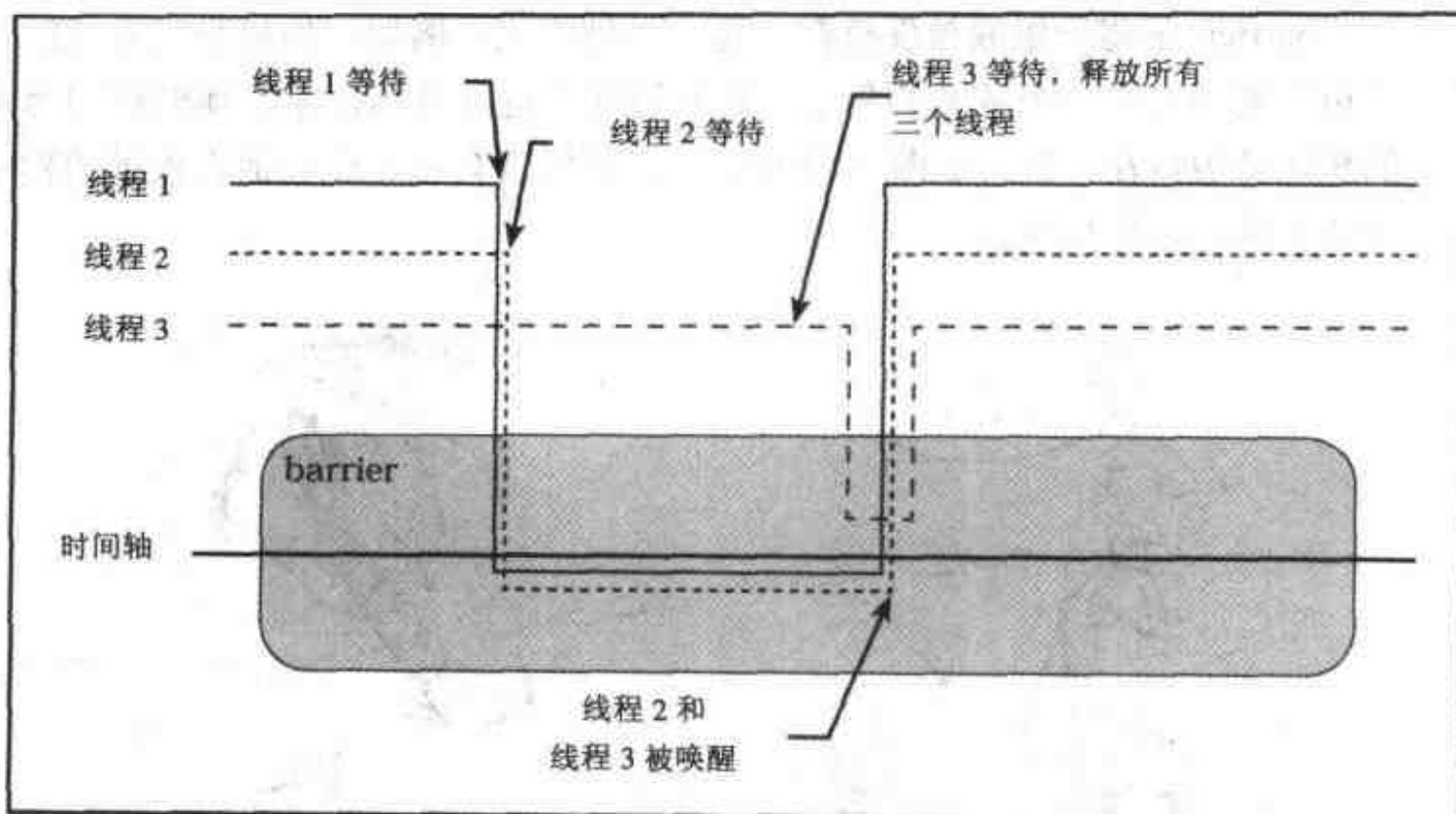


图 7.2 Barrier 操作

在本例中，线程 1 和线程 2 先后在 barrier 上等待。过一段时间后，线程 3 在 barrier 上等待，发现 barrier 已满，唤醒所有的等待线程。然后所有三个线程从 barrier 等待返回。

barrier 的核心是一个计数器。计数器被初始化为“旅行组”线程的数量，即在所有等待线程返回前，在一个 barrier 上必须等待的线程数。为给它一个简单的名字，我将称其为“阀值”。当每个线程到达 barrier 时，计数器减 1。如果计数器值没有到达 0，则线程等待。如果计数器值为 0，它将唤醒所有等待线程。

因为计数器将被多个线程修改，它必须由一个互斥量来保护。因为线程将等待一些事件（计数器值为 0），barrier 需要有一个条件变量和一个谓语表达式。当计数器到达 0 且 barrier 打开时，我们需要重置计数器，这意味着 barrier 必须存储原来的阀值。

显然的谓语是简单地等待计数器为 0，但是那将复杂化重置 barrier 的过程。我们什么时候能重置计数器为原来值？当计数器为 0 时，我们不能重置它，因为此时大多数线程正在条件变量上等待。当它们醒来时，计数器必须是 0，否则它们将继续等待。记住：条件变量等待发生在重置谓语的循环里面。

最好的办法是为谓语增加一个独立的变量。我们将使用一个 cycle 变量，该变量在每次 barrier 的周期完成时逻辑取反。即，无论何时重置计数器值时，在广播条件变量之前，线程将反转 cycle（周期）标志。只要周期标志与在线程进入 barrier 时看到的值一样，则线程将在循环中等待，这也意味着每个线程必须保存 cycle 的初始值。

头文件 barrier.h 和源文件 barrier.c 演示了使用标准 Pthreads 互斥量和条件变量的一个 barrier 实现，这是一个相对容易理解的可移植的实现。当然，一个

人能够基于不可移植的硬件和操作系统特征为任何特定系统创建更有效的实现。

6~13 第 1 部分显示了一个由类型 `barrier_t` 表示的 `barrier` 结构。你可以看见互斥量(`mutex`)和条件变量(`cv`)。`threshold` 成员是组中线程的数量，而 `counter` 是必须在 `barrier` 加入这个组的线程数量。`cycle` 是前一段讨论的标志。它被用来确保一个从 `barrier` 醒来的等待线程将很快地返回调用者，但是如果它在所有线程恢复执行以前再次调用了 `wait` 操作，它将在 `barrier` 阻塞。

15 `BARRIER_VALID` 宏定义了一个“魔术数字”，我们将它保存到 `valid` 成员，然后检查它来判定传给其他 `barrier` 接口的一个地址是否“很可能是”一个 `barrier`。这是一个可以捕获大多数错误的较容易、快捷的错误检查^①。

■ barrier.h	part 1 barrier_t
<pre>1 #include <pthread.h> 2 3 /* 4 * Structure describing a barrier. 5 */ 6 typedef struct barrier_tag { 7 pthread_mutex_t mutex; /* Control access to barrier */ 8 pthread_cond_t cv; /* wait for barrier */ 9 int valid; /* set when valid */ 10 int threshold; /* number of threads required */ 11 int counter; /* current number of threads */ 12 unsigned long cycle; /* count cycles */ 13 } barrier_t; 14 15 #define BARRIER_VALID 0xdbcafe</pre>	
■ barrier.h	part 1 barrier_t

第 2 部分显示了允许你处理 `barrier_t` 结构的函数定义和原型。首先，你将希望初始化一个新的 `barrier`。

4~6 你能在编译期间使用宏 `BARRIER_INITIALIZER` 初始化一个静态 `barrier`。同样，你还可以调用 `barrier_init` 动态地初始化一个 `barrier`。

11~13 一旦你初始化了一个 `barrier`，你将希望能够使用它，并且使用一个 `barrier` 做的主要事情就是在它上面等待。当我们用完一个 `barrier` 时，最好能破坏 `barrier` 并且回收它使用的资源。我们把这些操作称为 `barrier_init`、`barrier_wait` 和 `barrier_destroy`。所有的操作需要指定在哪个 `barrier` 上执行。因为 `barrier` 是同步对象，并且包含一个互斥量和一个条件变量（二者都不能被拷贝），我们总是传递一个指向 `barrier` 的指针。只有初始化操作要求第二个参数，即在 `barrier` 打开之前要

^① 我总是喜欢使用一些读起来像英文单词的十六进制数来定义“魔术”数字。对于 `barrier`，我发明了自己的饭馆名 DBCafe，或者是 C 语言语法表示为 0Xdbcafe。很多有趣的英文词可以只用字母 a~f 来拼写。如果使用数字 1 表示字母 l，用 0 表示字母 o，则就会有更多的可能（你是否喜欢这种效果很大程度上取决于代码的打印字体）。

求等待的线程数。

为了与 Pthreads 习惯一致，函数都返回整数值，代表一个在<errno.h>中定义的错误数字。0 代表返回成功。

■ barrier.h

part 2 interfaces

```

1  /*
2   * Support static initialization of barriers.
3   */
4 #define BARRIER_INITIALIZER(cnt) \
5     {PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, \
6      BARRIER_VALID, cnt, cnt, 0}
7
8 /*
9  * Define barrier functions
10 */
11 extern int barrier_init (barrier_t *barrier, int count);
12 extern int barrier_destroy (barrier_t *barrier);
13 extern int barrier_wait (barrier_t *barrier);

```

■ barrier.h

part 2 interfaces

既然你知道了接口定义，你可以写一个使用 barrier 的程序。但是本节的重点不是告诉你如何使用 barrier，而是通过显示如何构造一个 barrier 来增进你对多线程编程的理解。以下例子显示了由 barrier.c 提供的函数，来实现 barrier.h 中的接口。

第 1 部分显示 barrier_init，你将动态地调用它来初始化一个 barrier，例如，当你使用 malloc 分配一个 barrier 时。

12 counter（计数器）和 threshold（阀值）被设置为相同的值。counter 是工作计数器，并且将在每个 barrier 周期中被重置为 threshold。

14~16 如果互斥量初始化失败，barrier_init 返回失败状态。

17~21 如果条件变量（cv）初始化失败，barrier_init 释放它已经创建的互斥量（mutex）并返回失败状态——因为创建条件变量的失败比释放互斥量的失败更重要，pthread_mutex_destroy 的状态被忽略。

22 仅在所有初始化完成后，barrier 被标记为有效。这不能完全确保：当其他线程试图在那个 barrier 上等待时将检测到 barrier 无效，而非进入一些难于诊断的失败方式中，但是至少它是一种象征意义上的尝试。

■ barrier.c

part 1 barrier_init

```

1 #include <pthread.h>
2 #include "errors.h"
3 #include "barrier.h"
4
5 /*
6  * Initialize a barrier for use.
7  */
8 int barrier_init (barrier_t *barrier, int count)
9 {

```

```

10     int status;
11
12     barrier->threshold = barrier->counter = count;
13     barrier->cycle = 0;
14     status = pthread_mutex_init (&barrier->mutex, NULL);
15     if (status != 0)
16         return status;
17     status = pthread_cond_init (&barrier->cv, NULL);
18     if (status != 0) {
19         pthread_mutex_destroy (&barrier->mutex);
20         return status;
21     }
22     barrier->valid = BARRIER_VALID;
23     return 0;
24 }
```

■ barrier.c**part 1 barrier_init**

第 2 部分显示了 `barrier_destroy` 函数，它释放 `barrier_t` 结构中的互斥量（`mutex`）和条件变量（`cv`）。如果我们为 `barrier` 分配了任何附加资源，我们也应该释放那些资源。

8~9 首先通过查看 `valid` 成员来检查 `barrier` 是否有效和被初始化的。我们先不锁住互斥量，因为如果互斥量被破坏或没被初始化，那将可能类似于内存段错误一样失败。因为我们不先加锁互斥量，确认检查不是完全可靠的，但是有总比没有好，而且该检查仅当检测到一些竞争条件时（如线程试图破坏一个 `barrier` 而另外一个线程正在初始化它时，或者两个线程几乎同时试图破坏一个 `barrier` 时）失败。

19~22 如果当前有任何线程在 `barrier` 上等待，返回 `EBUSY`。

24~27 在当前点，`barrier` 被破坏——它留下的所有东西被清除。如果在我们完成之前其他线程试图在 `barrier` 上等待，为将混淆的机会减到最小，在改变任何其他状态前，将 `valid` 变量清空以标记 `barrier` 为无效。然后解锁互斥量，因为当它被锁住时，我们不能释放它。

33~35 释放互斥量和状况变量。如果互斥量释放失败，返回状态；否则，返回条件变量释放的状态。或者，如果任何一个释放失败，返回错误状态；否则，返回成功。

■ barrier.c**part 2 barrier_destroy**

```

1  /*
2   * Destroy a barrier when done using it.
3   */
4  int barrier_destroy (barrier_t *barrier)
5  {
6      int status, status2;
7
8      if (barrier->valid != BARRIER_VALID)
9          return EINVAL;
10
11     status = pthread_mutex_lock (&barrier->mutex);
12     if (status != 0)
13         return status;
14 }
```

```

15  /*
16   * Check whether any threads are known to be waiting; report
17   * "BUSY" if so.
18   */
19  if (barrier->counter != barrier->threshold) {
20      pthread_mutex_unlock (&barrier->mutex);
21      return EBUSY;
22  }
23
24  barrier->valid = 0;
25  status = pthread_mutex_unlock (&barrier->mutex);
26  if (status != 0)
27      return status;
28
29  /*
30   * If unable to destroy either 1003.1c synchronization
31   * object, return the error status.
32   */
33  status = pthread_mutex_destroy (&barrier->mutex);
34  status2 = pthread_cond_destroy (&barrier->cv);
35  return (status != 0 ? status : status2);
36 }

```

■ barrier.c

part 2 barrier_destroy

最后，第 3 部分演示了 barrier_wait 的实现。

首先我们验证 barrier 参数是一个有效的 barrier_t。我们在锁住互斥量前执行该检查，以便 barrier_destroy 能在它清除了有效的成员后安全地破坏互斥量。如果一个线程试图在一个 barrier 上等待而同时另外的线程正在初始化或释放那个 barrier 时，这将是使损坏减到最小的一个简单尝试。

我们不能完全地避免问题，因为没有互斥量，barrier_wait 就不能保证它将看见正确的（最新的）valid 值。当 barrier 正在被置为无效时，或者当 barrier 在被置为有效过程中失败时，有效性检查可能成功。首先锁住互斥量也没用，因为如果 barrier 没有被完全初始化，或如果它正在被破坏时，互斥量不能存在。只要你正确地使用 barrier，这就不是一个大问题——即，在任何线程可能试图使用它之前，你初始化 barrier；并且直到你肯定没有线程将再次试图使用它时，才释放 barrier。

将 barrier 中 cycle 的当前值拷贝到一个本地变量中。我们的条件等待谓词就是对 barrier_t 结构的 cycle 成员和本地的 cycle 变量进行比较。谓词保证：当最后的等待线程广播条件变量时，所有等待线程将从 barrier_wait 返回；但是再次调用 barrier_wait 的任何线程将等待下一个广播（这是正确实现一个 barrier “困难的部分”）。

现在我们减少计数器的值，该值表示被要求但还没在 barrier 上等待的线程数。当计数器到达 0 时，不需要更多的线程——它们都在这儿了，并且焦急地等待继续到下一个吸引它们的地点。现在我们需要做的就是告诉它们醒来。我们前进到下一个周期，重置计数器，并且广播 barrier 的条件变量。

28~29 先前我曾提及程序通常需要一个线程在并发区域间执行一些清除或安装操作。每个线程可以锁住一个互斥量并检查一个标志，以便只有一个线程可以执行安装操作。然而，安装可能不需要附加的同步，例如，因为其他线程将在一个 barrier 上等待下一个并发区域，则在这种情况下，最好避免锁住一个额外的互斥量。

barrier_wait 函数有这样的能力。有且仅有一个线程将返回特殊值-1，而其他线程返回 0。在这个特别实现中，最后等待并唤醒其他线程的线程将获此殊荣，但是原则上哪个线程返回-1 是“未特别指出”的。返回-1 的线程可以执行安装，而其他线程争着向前继续。如果不需要特殊的返回状态，那么将-1 作为另外的成功形式对待。提议的 POSIX.1j 标准有一个类似的功能——一个(未特别指出)完成 barrier 的线程将返回状态 BARRIER_SERIAL_THREAD。

35 任何使用条件变量的线程代码应该总是或者支持推迟取消或者停用取消。记得有两种不同类型的取消：推迟和异步。处理异步取消的代码是罕见的。总的来说，在需要获得资源（包括锁住一个互斥量）的任何代码中支持异步取消是困难的或不可能的。除非文档特别这样说明，程序员们不能假设任何函数支持异步取消，因此我们不需要担心异步取消。

我们可以将 barrier_wait 编码为处理推迟的取消，但是那将增加问题的难度。例如，如果线程之一被取消了，barrier 等待如何被满足？并且如果它不会被满足，那些已经在那个 barrier 上等待（或准备等待）的所有其他线程会怎样？对于这些问题有各种各样的回答。其中一个是：在 barrier_wait 中记录在 barrier 上等待的所有线程的标识符，以及当任何线程在等待过程中被取消时，取消所有其他的等待线程。

或者，我们可以通过设置一些特殊的错误标志并广播条件变量来处理取消，并当在那种情况下醒来时，修改 barrier_wait 返回一个特殊的错误。然而，对于取消正在使用一个 barrier 的线程，这样做没有什么意义。我们将否认通过在等待以前停用取消、并在后来恢复取消的先前状态的做法。这也是提议的 POSIX.1j 标准采取的方式，顺便说一句，barrier 等待不是取消点。

42~46 如果有更多的线程尚未到达 barrier，我们需要等待它们。我们在条件变量上等待直到 barrier 进展到了下一个周期——即，barrier 的 cycle 不再匹配本地的拷贝。

■ barrier.c

part 3 barrier_wait

```
1  /*
2   * Wait for all members of a barrier to reach the barrier. When
3   * the count (of remaining members) reaches 0, broadcast to wake
4   * all threads waiting.
5   */
6  int barrier_wait (barrier_t *barrier)
7  {
8      int status, cancel, tmp, cycle;
9
10     if (barrier->valid != BARRIER_VALID)
```

```

11         return EINVAL;
12
13     status = pthread_mutex_lock (&barrier->mutex);
14     if (status != 0)
15         return status;
16
17     cycle = barrier->cycle; /* Remember which cycle we're on */
18
19     if (--barrier->counter == 0) {
20         barrier->cycle++;
21         barrier->counter = barrier->threshold;
22         status = pthread_cond_broadcast (&barrier->cv);
23         /*
24          * The last thread into the barrier will return status
25          * -1 rather than 0, so that it can be used to perform
26          * some special serial code following the barrier.
27          */
28         if (status == 0)
29             status = -1;
30     } else {
31         /*
32          * Wait with cancellation disabled, because barrier_wait
33          * should not be a cancellation point.
34          */
35         pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, &cancel);
36
37         /*
38          * Wait until the barrier's cycle changes, which means
39          * that it has been broadcast, and we don't want to wait
40          * anymore.
41          */
42         while (cycle == barrier->cycle) {
43             status = pthread_cond_wait (
44                 &barrier->cv, &barrier->mutex);
45             if (status != 0) break;
46         }
47
48         pthread_setcancelstate (cancel, &ttmp);
49     }
50     /*
51      * Ignore an error in unlocking. It shouldn't happen, and
52      * reporting it here would be misleading -- the barrier wait
53      * completed, after all, whereas returning, for example,
54      * EINVAL would imply the wait had failed. The next attempt
55      * to use the barrier *will* return an error, or hang, due
56      * to whatever happened to the mutex.
57      */
58     pthread_mutex_unlock (&barrier->mutex);
59     return status; /* error, -1 for waker, or 0 */
60 }
```

■ barrier.c

part 3 barrier_wait

最后，barrier_main.c是使用barrier的一个简单程序。每个线程在一个私

有数组内循环计算。

35~47 在每次循环的开始和结束, 运行函数 `thread_routine` 的线程都在一个 `barrier` 上等待同步操作。

56~61 在每次循环的结束, “领头线程”(从 `barrier_wait` 返回-1的那个)将修改所有线程的数据, 为下一次循环做准备。其他线程则直接转到循环的开始并在 `barrier` 上等待(第35行)。

■ barrier_main.c

```
1 #include <pthread.h>
2 #include "barrier.h"
3 #include "errors.h"
4
5 #define THREADS 5
6 #define ARRAY 6
7 #define INLOOPS 1000
8 #define OUTLOOPS 10
9
10 /*
11  * Keep track of each thread.
12  */
13 typedef struct thread_tag {
14     pthread_t    thread_id;
15     int          number;
16     int          increment;
17     int          array[ARRAY];
18 } thread_t;
19
20 barrier_t barrier;
21 thread_t thread[THREADS];
22
23 /*
24  * Start routine for threads.
25  */
26 void *thread_routine (void *arg)
27 {
28     thread_t *self = (thread_t*)arg; /* Thread's thread_t */
29     int in_loop, out_loop, count, status;
30
31     /*
32      * Loop through OUTLOOPS barrier cycles.
33      */
34     for (out_loop = 0; out_loop < OUTLOOPS; out_loop++) {
35         status = barrier_wait (&barrier);
36         if (status > 0)
37             err_abort (status, "Wait on barrier");
38
39         /*
40          * This inner loop just adds a value to each element in
41          * the working array.
42          */
43         for (in_loop = 0; in_loop < INLOOPS; in_loop++)
```

```
44         for (count = 0; count < ARRAY; count++)
45             self->array[count] += self->increment;
46
47         status = barrier_wait (&barrier);
48         if (status > 0)
49             err_abort (status, "Wait on barrier");
50
51         /*
52          * The barrier causes one thread to return with the
53          * special return status -1. The thread receiving this
54          * value increments each element in the shared array.
55          */
56         if (status == -1) {
57             int thread_num;
58
59             for (thread_num = 0; thread_num < THREADS; thread_num++)
60                 thread[thread_num].increment += 1;
61         }
62     }
63     return NULL;
64 }
65
66 int main (int arg, char *argv[])
67 {
68     int thread_count, array_count;
69     int status;
70
71     barrier_init (&barrier, THREADS);
72
73     /*
74      * Create a set of threads that will use the barrier.
75      */
76     for (thread_count = 0; thread_count < THREADS; thread_count++) {
77         thread[thread_count].increment = thread_count;
78         thread[thread_count].number = thread_count;
79
80         for (array_count = 0; array_count < ARRAY; array_count++)
81             thread[thread_count].array[array_count] = array_count + 1;
82
83         status = pthread_create (&thread[thread_count].thread_id,
84             NULL, thread_routine, (void*)&thread[thread_count]);
85         if (status != 0)
86             err_abort (status, "Create thread");
87     }
88
89     /*
90      * Now join with each of the threads.
91      */
92     for (thread_count = 0; thread_count < THREADS; thread_count++) {
93         status = pthread_join (thread[thread_count].thread_id, NULL);
94         if (status != 0)
95             err_abort (status, "Join thread");
96
97         printf ("%02d: (%d) ",
```

```
98         thread_count, thread[thread_count].increment);
99
100        for (array_count = 0; array_count < ARRAY; array_count++)
101            printf ("%010u ",
102                    thread[thread_count].array[array_count]);
103            printf ("\n");
104        }
105
106    /*
107     * To be thorough, destroy the barrier.
108     */
109    barrier_destroy (&barrier);
110    return 0;
111 }
```

■ barrier_main.c

7.1.2 读/写锁

读/写锁很像一个互斥量，它是阻止多个线程同时修改共享数据的另外一种方法。但是不同于互斥量的是它区分读数据和写数据。一个互斥量排除所有的其他线程，而一个读/写锁允许多个线程同时读数据，只要它们不需要改变数据。

读/写锁被用来保护经常需要读但是通常不需要修改的信息。例如，当你建造最近存储信息的缓存时，许多线程可以同时没有冲突地检验缓存。当一个线程需要更新缓存时，它必须有独占的存取权。

当一个线程锁住一个读/写锁时，它选择共享读访问或者独占写访问。当有任何线程在写访问时，想要读访问的线程不能继续。当其他线程正在进行写存取或读存取时，试图获得写存取的一个线程不能继续。

当写锁被释放时，如果读数据者和写数据者同时正在等待存取，则读数据者被优先给予访问权。因为潜在地允许许多线程同时完成工作，读访问优先有利于并发。另一方面，写访问优先将保证在数据被使用以前，对于共享数据的未解决的修正必须被完成。没有绝对正确或错误的政策，并且如果你认为这里的实现不适合你，可以很容易地改变它。

图 7.3 显示了使用一个读/写锁来同步三个线程的操作。本图是一种时序图，时间从左向右递增。从左上方标签开始的各个线段分别表示指定线程的行为：实线表示线程 1，点线表示线程 2，虚线表示线程 3。当线段在圆型矩形内落下时，表示它们正在与读/写锁交互。如果线段落在中心线以下，表示线程锁住读/写锁（或者为独占写或者为共享读）。在中心线上面徘徊的线段代表等待锁的线程。

在本例中，线程 1 锁住读/写锁为独占写。线程 2 试图锁读/写锁为共享读，发现它已经锁为独占写，则阻塞。当线程 1 释放锁时，它唤醒线程 2，线程 2 然后将读/写锁成功地锁为共享读。然后，线程 3 试图锁读/写锁为共享读，因为读/写锁已经被锁为共享读，它很快地成功。线程 1 然后再次试图锁读/写锁为独占写，因为读/写锁已经锁为读存取，所以线程 1 阻塞。当线程 3 解开读/写锁时，它不能唤醒线程 1，

因为有其他的读线程。仅当线程 2 也开锁时，线程 1 才能被唤醒，锁住读/写锁为独占写。

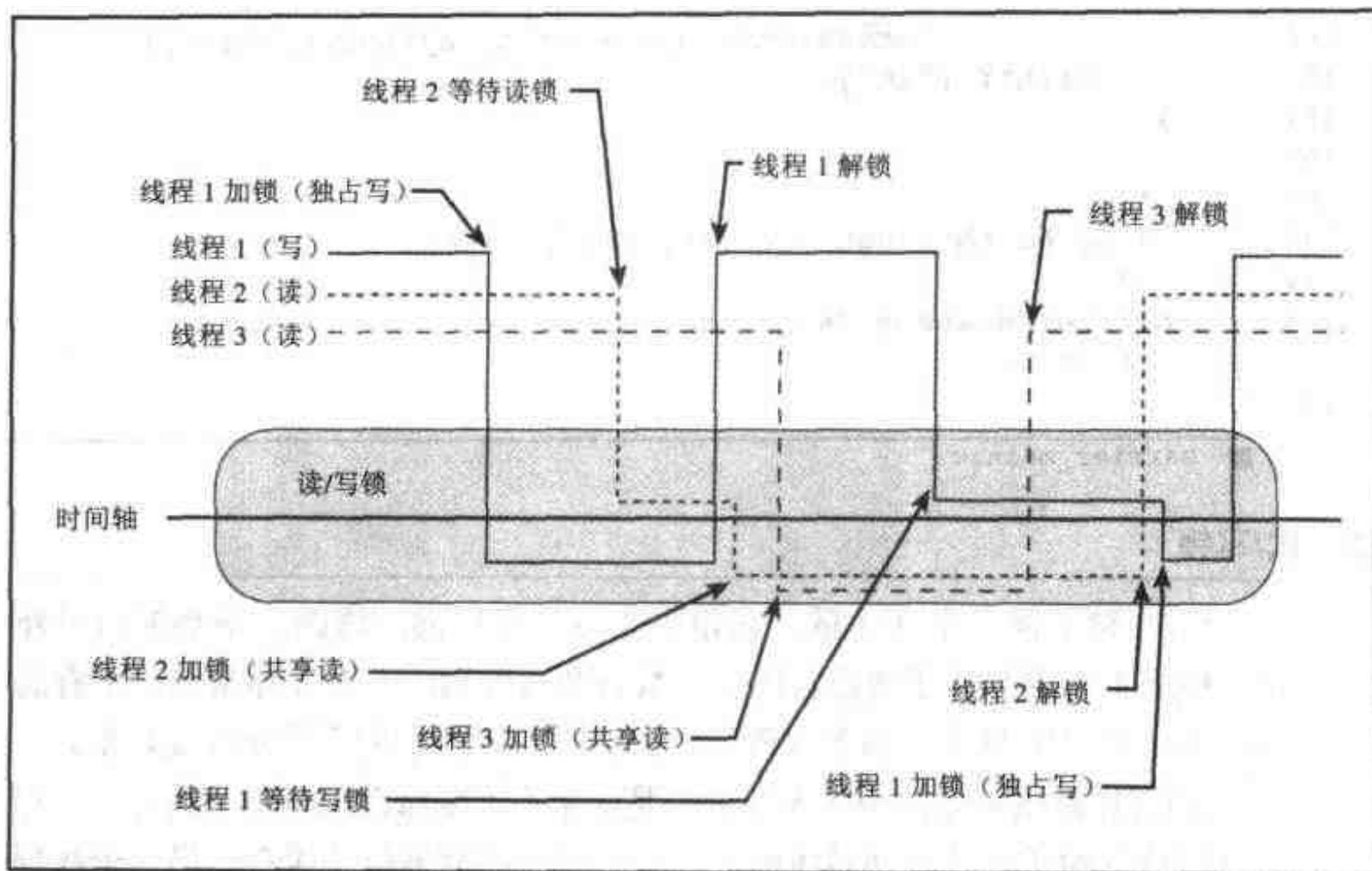


图 7.3 读/写锁操作

头文件 `rwlock.h` 和源文件 `rwlock.c` 演示了如何使用标准的 Pthreads 互斥量和状况变量实现读/写锁，这是相对容易理解的可移植的实现。当然，可以基于不可移植的硬件和操作系统特征为任何特定系统创建更有效的实现。

本节剩余部分显示了该读/写锁包（read/write lock package）的细节。首先，`rwlock.h` 描述接口，然后 `rwlock.c` 提供实现。第 1 部分演示了由类型 `rwlock_t` 代表的读/写锁的结构。

7~9 当然，有一个用于序列化结构存取的互斥量。我们将使用两个独立的状况变量，一个用于等待读存取（调用 `read`），一个用于等待写存取（调用 `write`）。

10 `rwlock_t` 结构有一个 `valid` 成员，用来容易地检测普通的用法错误，如试图加锁一个没被初始化的读/写锁。当读/写锁被初始化时，该成员被设置为一个魔术数字，就像在 `barrier_init` 中一样。

11~12 使我们能够判断条件变量上是否有等待线程，我们将保持活跃的读线程数（`r_active`）和一个指示活跃的写线程数（`w_active`）的标志。

13~14 我们也保留了等待读存取的线程数（`r_wait`）和等待写存取的线程数（`w_wait`）。

17 最后，我们的 `valid` 成员需要一个“magic 数”（如果你错过了 `barrier` 例子中的这一部分，见 7.1.1 节的脚注）。

■ rwlock.h

part 1 rwlock_t

```
1 #include <pthread.h>
2
3 /*
4  * Structure describing a read/write lock.
5  */
6 typedef struct rwlock_tag {
7     pthread_mutex_t      mutex;
8     pthread_cond_t       read;           /* wait for read */
9     pthread_cond_t       write;          /* wait for write */
10    int                valid;          /* set when valid */
11    int                r_active;        /* readers active */
12    int                w_active;        /* writer active */
13    int                r_wait;          /* readers waiting */
14    int                w_wait;          /* writers waiting */
15 } rwlock_t;
16
17 #define RWLOCK_VALID 0xfacade
```

■ rwlock.h

part 1 rwlock_t

通过使用单一的条件变量以及让读线程等待和写线程等待使用分开的谓词表达式，我们节省了代码空间并简化了代码。我们将为每个谓词使用一个条件变量，因为这样更有效。这是常用的折衷。主要的考虑是当两个谓词共享一个条件变量时，你必须总是使用 `pthread_cond_broadcast` 唤醒它们，这意味着每次解开读/写锁时都要唤醒所有的等待线程。

我们为“活动的写线程”跟踪一个布尔变量，因为只能有一个激活的写线程。对于“活跃的读线程”、“等待的读线程”和“等待的写线程”也有对应的计数器。即使没有“等待的读线程”和“等待的写线程”的计数器，也可以通过。使用广播同时唤醒所有的读线程，因此有多少读线程并没有关系。只要没有读线程，写线程就会被唤醒，因此我们无须跟踪是否有线程在等待写（当没有等待线程时，代价就是偶尔浪费的条件变量信号的费用）。

因为条件变量可能被取消，我们记录等待读的线程数。如果没有取消，我们就能使用一个简单的标志——“线程正在等待读”或“没有线程正在等待读”。每个线程可以在等待前将它置位，并且我们能在广播唤醒所有的等待读线程前清除它。然而，因为我们没有记录在一个条件变量上等待的线程数，当一个等待的读线程被取消时，我们不知道是否要清除该标志。这个信息是很关键的，因为如果没有等待的读线程，当读/写锁被解开时，我们必须唤醒一个写线程——但是如果有正在等待的读线程，我们就不能唤醒写线程。一个等待的读线程计数器（当等待线程被取消时，就减少该计数器）帮助我们解决了这个问题。

对于写线程，“弄错它”的后果相对读线程不那么重要。因为我们首先检查读线程，而确实不需要知道是否有写线程。在任何时候读/写锁被释放且没有等待读线程时，我们可以发信号给一个“潜在的写线程”。但是记录等待的写线程数允许我们在没有线程等待时避免发一个条件变量信号。

第 2 部分显示了其余的函数定义和原型。

4~6 RWLOCK_INITIALIZER 宏允许你静态地初始化一个读/写锁。

11~18 当然，你一定也能初始化不能静态分配的读/写锁，因此我们提供了 rwl_init 来动态地初始化，并且在用完它时使用 rwl_destroy 释放读/写锁。另外，还有一些用来加锁或解开读/写锁的函数。通过调用 rwl_readtrylock 或 rwl_writetrylock，可以试着加锁一个读/写锁，或为读操作或为写操作，就像可以调用 thread_mutex_trylock 试着加锁互斥量一样。

■ rwlock.h

part 2 interfaces

```

1 /*
2  * Support static initialization of barriers.
3 */
4 #define RWL_INITIALIZER \
5     {PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER, \
6      PTHREAD_COND_INITIALIZER, RWLOCK_VALID, 0, 0, 0, 0}
7
8 /*
9  * Define read/write lock functions.
10 */
11 extern int rwl_init (rwlock_t *rwlock);
12 extern int rwl_destroy (rwlock_t *rwlock);
13 extern int rwl_readlock (rwlock_t *rwlock);
14 extern int rwl_readtrylock (rwlock_t *rwlock);
15 extern int rwl_readunlock (rwlock_t *rwlock);
16 extern int rwl_writelock (rwlock_t *rwlock);
17 extern int rwl_writetrylock (rwlock_t *rwlock);
18 extern int rwl_writeunlock (rwlock_t *rwlock);

```

■ rwlock.h

part 2 interfaces

文件 rwlock.c 包含了读/写锁的实现。下列例子分别划分了实现 rwlock.h 接口的各个函数。

第 1 部分显示 rwl_init，它初始化一个读/写锁，它初始化 Pthreads 同步对象，初始化计数器和标志，最后设置 valid 标记以使读/写锁对于其他接口可识。如果不能初始化读条件变量，就释放已经创建的互斥量。同样，如果不能初始化写条件变量，同时释放互斥量和读条件变量。

■ rwlock.c

part 1 rwl_init

```

1 #include <pthread.h>
2 #include "errors.h"
3 #include "rwlock.h"
4
5 /*
6  * Initialize a read/write lock.
7 */
8 int rwl_init (rwlock_t *rwl)
9 {
10     int status;
11

```

```

12     rwl->r_active = 0;
13     rwl->r_wait = rwl->w_wait = 0;
14     rwl->w_active = 0;
15     status = pthread_mutex_init (&rw1->mutex, NULL);
16     if (status != 0)
17         return status;
18     status = pthread_cond_init (&rw1->read, NULL);
19     if (status != 0) {
20         /* if unable to create read CV, destroy mutex */
21         pthread_mutex_destroy (&rw1->mutex);
22         return status;
23     }
24     status = pthread_cond_init (&rw1->write, NULL);
25     if (status != 0) {
26         /* if unable to create write CV, destroy read CV and mutex */
27         pthread_cond_destroy (&rw1->read);
28         pthread_mutex_destroy (&rw1->mutex);
29         return status;
30     }
31     rw1->valid = RWLOCK_VALID;
32     return 0;
33 }
```

■ rwlock.c

part 1 rwl_init

第 2 部分显示了 rwl_destroy 函数，它负责释放读/写锁。

8~9 我们首先检查 valid 成员来证实读/写锁已经被初始化。这虽然不是检查不正确用法的安全保护，但是它是廉价的，并且它将捕获一些最普通的错误。可以参考第 2 部分 barrier.c 的注解以获得 valid 成员的更多用法。

10~30 检查读/写锁是否在使用。我们查找正在使用或等待读或写操作的任何一个线程。尽管没有其他好处，使用两个分开的陈述使得测试稍微更加可读。

36~39 如 barrier_destroy 一样，释放所有的 Pthreads 同步对象，并且存储每个返回状态。如果任何释放调用失败，返回非零值，rwl_destroy 将返回那个状态；如果它们都成功，将返回 0 表示成功。

■ rwlock.c

part 2 rwl_destroy

```

1  /*
2   * Destroy a read/write lock.
3   */
4  int rwl_destroy (rwlock_t *rw1)
5  {
6      int status, status1, status2;
7
8      if (rw1->valid != RWLOCK_VALID)
9          return EINVAL;
10     status = pthread_mutex_lock (&rw1->mutex);
11     if (status != 0)
12         return status;
13
14     /*
15      * Check whether any threads own the lock; report "BUSY" if
```

```

16     * SO.
17     */
18     if (rw1->r_active > 0 || rw1->w_active) {
19         pthread_mutex_unlock (&rw1->mutex);
20         return EBUSY;
21     }
22
23     /*
24     * Check whether any threads are known to be waiting; report
25     * EBUSY if so.
26     */
27     if (rw1->r_wait > 0 || rw1->w_wait > 0) {
28         pthread_mutex_unlock (&rw1->mutex);
29         return EBUSY;
30     }
31
32     rw1->valid = 0;
33     status = pthread_mutex_unlock (&rw1->mutex);
34     if (status != 0)
35         return status;
36     status = pthread_mutex_destroy (&rw1->mutex);
37     status1 = pthread_cond_destroy (&rw1->read);
38     status2 = pthread_cond_destroy (&rw1->write);
39     return (status != 0 ? status
40             : (status1 != 0 ? status1 : status2));
41 }

```

■ rwlock.c

part 2 rwl_destroy

第 3 部分显示了 rwl_readcleanup 和 rwl_writecleanup 代码, 这是在加锁读/写锁时使用的两个取消清除处理器。由此可以推断, 不同于 barrier, 读/写锁是取消点。当一个等待被取消时, 等待线程需要相应地减少等待读或写锁的线程数, 并且解锁互斥量。

■ rwlock.c

part 3 cleanuphandlers

```

1 /*
2  * Handle cleanup when the read lock condition variable
3  * wait is canceled.
4  *
5  * Simply record that the thread is no longer waiting,
6  * and unlock the mutex.
7  */
8 static void rwl_readcleanup (void *arg)
9 {
10     rwlock_t      *rw1 = (rwlock_t *)arg;
11
12     rw1->r_wait--;
13     pthread_mutex_unlock (&rw1->mutex);
14 }
15
16 /*
17  * Handle cleanup when the write lock condition variable
18  * wait is canceled.

```

```

19  *
20  * Simply record that the thread is no longer waiting,
21  * and unlock the mutex.
22  */
23 static void rwl_writecleanup (void *arg)
24 {
25     rwlock_t *rwl = (rwlock_t *)arg;
26
27     rwl->w_wait--;
28     pthread_mutex_unlock (&rwl->mutex);
29 }

```

■ rwlock.c**part 3 cleanuphandlers**

10-26 第4部分显示了 rwl_readlock，它为读操作加锁读/写锁。如果一个写线程当前是活动的 (w_active 非零)，我们等待它广播读条件变量。r_wait 成员记录等待读线程的数量。这也可以是一个简单的布尔变量，只是有一个问题——当等待线程被取消时，我们需要知道是否仍然有任何等待线程。而维持一个计数使这一工作变得容易，因为清除处理器仅需要减少计数器。

这是代码必须从“读线程优先”到“写线程优先”（如果你期望这样选择）变换读/写锁的地方之一。为实现写线程优先，当有正在等待的写线程 (w_wait>0) 时，读线程必须阻塞，而不仅仅是在有活动的写线程时（我们在这里采取的方式）。

15,21 注意清除处理器在条件等待附近的使用。另外，注意我们向 pthread_cleanup_pop 传递参数 0，以便只有当等待被取消时才调用清除代码。当等待没被取消时，我们需要施行稍微不同的行动。如果等待没被取消，需要在解锁互斥量前增加活动的读线程数。

■ rwlock.c**part 4 rwl_readlock**

```

1 /*
2  * Lock a read/write lock for read access.
3  */
4 int rwl_readlock (rwlock_t *rwl)
5 {
6     int status;
7
8     if (rwl->valid != RWLOCK_VALID)
9         return EINVAL;
10    status = pthread_mutex_lock (&rwl->mutex);
11    if (status != 0)
12        return status;
13    if (rwl->w_active) {
14        rwl->r_wait++;
15        pthread_cleanup_push (rwl_readcleanup, (void*)rwl);
16        while (rwl->w_active) {
17            status = pthread_cond_wait (&rwl->read, &rwl->mutex);
18            if (status != 0)
19                break;
20        }
21        pthread_cleanup_pop (0);

```

```

23     }
24     if (status == 0)
25         rwl->r_active++;
26     pthread_mutex_unlock (&rw1->mutex);
27     return status;
28 }
```

■ rwlock.c**part 4 rw1_readlock**

第 5 部分显示了 `rwl_readtrylock`。除了当一个写线程活动时，它将返回 `EBUSY` 而不是等待存取，该函数与 `rwl_readlock` 几乎相同。它不需要一个清除处理器，也不需要增加等待读线程数。

该函数必须也被修改以实现“写线程优先”的读/写锁：当一个写线程正在等待时，而不是当一个写线程激活时，函数返回 `EBUSY`。

■ rwlock.c**part 5 rw1_readtrylock**

```

1 /*
2  * Attempt to lock a read/write lock for read access (don't
3  * block if unavailable).
4 */
5 int rw1_readtrylock (rwlock_t *rw1)
6 {
7     int status, status2;
8
9     if (rw1->valid != RWLOCK_VALID)
10     return EINVAL;
11     status = pthread_mutex_lock (&rw1->mutex);
12     if (status != 0)
13         return status;
14     if (rw1->w_active)
15         status = EBUSY;
16     else
17         rw1->r_active++;
18     status2 = pthread_mutex_unlock (&rw1->mutex);
19     return (status2 != 0 ? status2 : status);
20 }
```

■ rwlock.c**part 5 rw1_readtrylock**

第 6 部分显示了 `rwl_readunlock`。该函数实质上通过减少活跃的读线程数 (`r_active`) 颠倒了 `rwl_readlock` 或 `rwl_tryreadlock` 的效果。

如果不再有活跃的读线程，并且至少有一个线程正在等待写操作，发写条件变量信号来唤醒其中的一个。注意这里有竞争，并且你或许应该担心它取决于你对发生事情的观点。如果在被唤醒的写线程执行以前，对于读操作感兴趣的其他线程调用了 `rwl_readlock` 或 `rwl_tryreadlock`，读线程可能“赢”，不管我们是否刚选择了一个写线程的事实。

因为我们的读/写锁的版本是“读线程优先”，这是我们通常想要发生的——写线程将判断自己失败并将恢复等待（它收到了一个假的唤醒）。如果实现变为写线程优先，假的唤醒不会发生，因为潜在的读线程将不得不被堵住。我们刚刚唤醒的等

待线程直到它实际拥有锁之前不能减少 w_wait 的值。

■ rwlock.c part 6 rwl_readunlock

```

1  /*
2   * Unlock a read/write lock from read access.
3   */
4  int rwl_readunlock (rwlock_t *rwl)
5  {
6      int status, status2;
7
8      if (rwl->valid != RWLOCK_VALID)
9          return EINVAL;
10     status = pthread_mutex_lock (&rwl->mutex);
11     if (status != 0)
12         return status;
13     rwl->r_active--;
14     if (rwl->r_active == 0 && rwl->w_wait > 0)
15         status = pthread_cond_signal (&rwl->write);
16     status2 = pthread_mutex_unlock (&rwl->mutex);
17     return (status2 == 0 ? status : status2);
18 }
```

■ rwlock.c part 6 rwl_readunlock

13 第 7 部分显示了 rwl_writelock。该函数很像 rwl_readlock，除了在条件变量上等待的谓词条件。在第 1 部分中，我曾解释，要从“读优先”变换为“写优先”，一个潜在的读线程将必须等待，直到没有活跃的或等待的写线程。而现在它仅仅等待活跃的写线程（读优先）。rwl_writelock 中的谓词倒置条件。因为理论上我们支持“读优先”，所以如果有活跃的或等待的读线程，我们必须等待。事实上，它是有点简单，因为如果有任何活跃的读线程，就不能有任何等待的读线程——读/写锁的整个要点就是多个线程能同时读。另一方面，如果有任何活跃的写线程，我们确实必须等待，因为一次只允许一个写线程。

25 不同于 r_active（一个计数器），w_active 被当作一个布尔变量。或者它也是一个计数器？确实没有语义的差别，因为 1 既可以被认为是布尔真也可以被当成是 1 值——任何时候只能有一个活跃的写线程。

■ rwlock.c part 7 rwl_writelock

```

1  /*
2   * Lock a read/write lock for write access.
3   */
4  int rwl_writelock (rwlock_t *rwl)
5  {
6      int status;
7
8      if (rwl->valid != RWLOCK_VALID)
9          return EINVAL;
10     status = pthread_mutex_lock (&rwl->mutex);
11     if (status != 0)
12         return status;
```

```

13     if (rw1->w_active || rw1->r_active > 0) {
14         rw1->w_wait++;
15         pthread_cleanup_push (rw1_writecleanup, (void*)rw1);
16         while (rw1->w_active || rw1->r_active > 0) {
17             status = pthread_cond_wait (&rw1->write, &rw1->mutex);
18             if (status != 0)
19                 break;
20         }
21         pthread_cleanup_pop (0);
22         rw1->w_wait--;
23     }
24     if (status == 0)
25         rw1->w_active = 1;
26     pthread_mutex_unlock (&rw1->mutex);
27     return status;
28 }
```

■ rwlock.c**part 7 rw1_writelock**

第 8 部分显示了 `rw1_writetrylock`。该函数很像 `rw1_writelock`，除了如果读/写锁当前被使用(或者被一个读线程或者被一个写线程)，它返回 `EBUSY` 而不是等待它变得可用。

■ rwlock.c**part 8 rw1_writetrylock**

```

1 /*
2  * Attempt to lock a read/write lock for write access. Don't
3  * block if unavailable.
4 */
5 int rw1_writetrylock (rwlock_t *rw1)
6 {
7     int status, status2;
8
9     if (rw1->valid != RWLOCK_VALID)
10     return EINVAL;
11     status = pthread_mutex_lock (&rw1->mutex);
12     if (status != 0)
13         return status;
14     if (rw1->w_active || rw1->r_active > 0)
15         status = EBUSY;
16     else
17         rw1->w_active = 1;
18     status2 = pthread_mutex_unlock (&rw1->mutex);
19     return (status != 0 ? status : status2);
20 }
```

■ rwlock.c**part 8 rw1_writetrylock**

最后，第 9 部分显示了 `rw1_writeunlock`。该函数被一个线程调用来释放写锁。

13~19

当一个写线程释放读/写锁时，它总是可用的；如果有任何线程等待，必须唤醒其中一个。因为我们实现“读优先”存取，所以首先寻找正在等待的读线程。如果有，将广播读条件变量来唤醒它们。

20~26 如果没有等待的读线程，但是有一个以上的写线程等待，通过发写条件变量信号唤醒其中一个。要实现一个“写优先”的锁，需要颠倒上述两个测试，即唤醒一个等待的写线程（如果有的话），然后再寻找等待的读线程。

■ rwlock.c

part 9 rwl_writeunlock

```

1  /*
2   * Unlock a read/write lock from write access.
3   */
4  int rwl_writeunlock (rwlock_t *rwl)
5  {
6      int status;
7
8      if (rwl->valid != RWLOCK_VALID)
9          return EINVAL;
10     status = pthread_mutex_lock (&rwl->mutex);
11     if (status != 0)
12         return status;
13     rwl->w_active = 0;
14     if (rwl->r_wait > 0) {
15         status = pthread_cond_broadcast (&rwl->read);
16         if (status != 0) {
17             pthread_mutex_unlock (&rwl->mutex);
18             return status;
19         }
20     } else if (rwl->w_wait > 0) {
21         status = pthread_cond_signal (&rwl->write);
22         if (status != 0) {
23             pthread_mutex_unlock (&rwl->mutex);
24             return status;
25         }
26     }
27     status = pthread_mutex_unlock (&rwl->mutex);
28     return status;
29 }
```

■ rwlock.c

part 9 writelock

既然我们已经完成了所有的部分，则 rwlock_main.c 显示了一个使用读/写锁的程序。

11~17 各个线程被类型 thread_t 的结构描述。thread_num 成员是线程在 thread_t 结构数组中的索引。thread_id 成员是当线程被创建时由 pthread_create 返回的 pthread_t (线程标识符)。updates 和 reads 成员是表示线程执行的读锁和写锁操作的计数器。interval 成员在每个线程创建时被随机产生，来决定在线程执行写操作之前，线程将重复读的循环次数。

22~26 线程循环遍历一个 data_t 元素数组。每个 data_t 元素有一个读/写锁，一个 data 元素和某个线程已经更新该元素的计数。

48~58 程序创建一组运行 thread_routine 函数的线程。每个线程循环 ITERATIONS 次使用读/写锁。它循环遍历 data 数组，当它到达结尾时重置索引

(element) 为 0。在由每个线程的 interval 成员指定的间隔到来时，线程将修改当前的 data 单元而不是读它。线程加锁读/写锁为写锁，将 thread_num 作为新的 data 值保存，并增加 updates 计数器。

59~73 在所有其他的循环中，thread_routine 读当前的 data 单元，锁住读/写锁为读锁。通过将它的 thread_num 变量与 data 值做比较，判断自己是否是最近更新 data 单元的线程。如果是，则增加计数器的值。

95~103 在 Solaris 系统上，增加线程并发度以使程序表现更多有趣的行为。没有用户线程的时间片机制，每个线程将趋于顺序执行。

■ rwlock_main.c

```
1 #include "rwlock.h"
2 #include "errors.h"
3
4 #define THREADS          5
5 #define DATASIZE         15
6 #define ITERATIONS       10000
7
8 /*
9  * Keep statistics for each thread.
10 */
11 typedef struct thread_tag {
12     int      thread_num;
13     pthread_t thread_id;
14     int      updates;
15     int      reads;
16     int      interval;
17 } thread_t;
18
19 /*
20  * Read/write lock and shared data.
21 */
22 typedef struct data_tag {
23     rwlock_t   lock;
24     int        data;
25     int        updates;
26 } data_t;
27
28 thread_t threads[THREADS];
29 data_t data[DATASIZE];
30
31 /*
32  * Thread start routine that uses read/write locks.
33 */
34 void *thread_routine (void *arg)
35 {
36     thread_t *self = (thread_t*)arg;
37     int repeats = 0;
38     int iteration;
39     int element = 0;
40     int status;
```

```
41
42     for (iteration = 0; iteration < ITERATIONS; iteration++) {
43         /*
44          * Each "self->interval" iterations, perform an
45          * update operation (write lock instead of read
46          * lock).
47          */
48         if ((iteration & self->interval) == 0) {
49             status = rwl_writelock (&data[element].lock);
50             if (status != 0)
51                 err_abort (status, "Write lock");
52             data[element].data = self->thread_num;
53             data[element].updates++;
54             self->updates++;
55             status = rwl_writeunlock (&data[element].lock);
56             if (status != 0)
57                 err_abort (status, "Write unlock");
58         } else {
59             /*
60              * Look at the current data element to see whether
61              * the current thread last updated it. Count the
62              * times, to report later.
63              */
64             status = rwl_readlock (&data[element].lock);
65             if (status != 0)
66                 err_abort (status, "Read lock");
67             self->reads++;
68             if (data[element].data == self->thread_num)
69                 repeats++;
70             status = rwl_readunlock (&data[element].lock);
71             if (status != 0)
72                 err_abort (status, "Read unlock");
73         }
74         element++;
75         if (element >= DATASIZE)
76             element = 0;
77     }
78
79     if (repeats > 0)
80         printf (
81             "Thread %d found unchanged elements %d times\n",
82             self->thread_num, repeats);
83     return NULL;
84 }
85
86 int main (int argc, char *argv[])
87 {
88     int count;
89     int data_count;
90     int status;
91     unsigned int seed = 1;
92     int thread_updates = 0;
93     int data_updates = 0;
94 }
```

```
95 #ifdef sun
96     /*
97      * On Solaris 2.5, threads are not timesliced. To ensure
98      * that our threads can run concurrently, we need to
99      * increase the concurrency level to THREADS.
100     */
101    DPRINTF (( "Setting concurrency level to %d\n", THREADS));
102    thr_setconcurrency (THREADS);
103#endif
104
105    /*
106     * Initialize the shared data.
107     */
108    for (data_count = 0; data_count < DATASIZE; data_count++) {
109        data[data_count].data = 0;
110        data[data_count].updates = 0;
111        status = rwl_init (&data[data_count].lock);
112        if (status != 0)
113            err_abort (status, "Init rw lock");
114    }
115
116    /*
117     * Create THREADS threads to access shared data.
118     */
119    for (count = 0; count < THREADS; count++) {
120        threads[count].thread_num = count;
121        threads[count].updates = 0;
122        threads[count].reads = 0;
123        threads[count].interval = rand_r (&seed) % 71;
124        status = pthread_create (&threads[count].thread_id,
125                               NULL, thread_routine, (void*)&threads[count]);
126        if (status != 0)
127            err_abort (status, "Create thread");
128    }
129
130    /*
131     * Wait for all threads to complete, and collect
132     * statistics.
133     */
134    for (count = 0; count < THREADS; count++) {
135        status = pthread_join (threads[count].thread_id, NULL);
136        if (status != 0)
137            err_abort (status, "Join thread");
138        thread_updates += threads[count].updates;
139        printf ("%02d: interval %d, updates %d, reads %d\n",
140               count, threads[count].interval,
141               threads[count].updates, threads[count].reads);
142    }
143
144    /*
145     * Collect statistics for the data.
146     */
147    for (data_count = 0; data_count < DATASIZE; data_count++) {
```

```
148     data_updates += data[data_count].updates;
149     printf ("data %02d: value %d, %d updates\n",
150            data_count, data[data_count].data,
151            data[data_count].updates);
152     rwl_destroy (&data[data_count].lock);
153 }
154
155 printf ("%d thread updates, %d data updates\n",
156         thread_updates, data_updates);
157 return 0;
158 }
```

■ rwlock_main.c

7.2 工作队列管理器

我已经简短地讲述了线程合作的各种模型。包括管道、工作组成员、客户机/服务器等等。在本节中，我演示一个“工作队列”的开发，即一组线程从同一个队列中接受工作请求，并且并行地处理它们（潜在地）。

也可以认为工作队列管理器是一个工作组成员管理器，这取决于你的观察点。如果你将它认为是在一组线程间分派工作的方法，那么“工作组成员”可能更适当些。我更喜欢把它想像为在后台为你工作的一个神奇排队，因为工作成员对于调用者而言几乎完全不可见。

当创建工作队列时，可以指定需要的最大并发级别，工作队列管理器把它解释为最大的“引擎”线程（可以处理请求的）数量。依据工作量的要求，线程将被开始或停止。没有发现任何请求的一个线程将等待一段时间后终止。最优的时间段取决于在你的系统上创建一个新线程的开销、维护一个不做任何工作的线程的系统资源的开销，以及你将再次需要线程的可能性。这里我选择了 2 秒，它可能有些太长了。

头文件 workq.h 和源文件 workq.c 显示了一个工作队列管理器的实现。第 1 部分显示了工作队列包使用的两种结构类型。`workq_t` 类型是一个工作队列的外部表示，`workq_ele_t` 是队列中工作元素的内部表示。

6~9 `workq_ele_t` 结构用于维持工作项目的一个链表。包括一个连接指针(`next`)和一个 `data` 值，当工作元素排队时被存储并且不做解释地传递给调用者的“引擎函数”。

14~16 当然，还有一个互斥量用来序列化对 `workq_t` 的访问，还有一个条件变量(`cv`)引擎线程在上面等待队列中的工作。

17 `attr` 成员是一个线程属性对象，当创建新的引擎线程时使用。属性对象也可以作为 `workq.c` 内的一个静态变量实现，但是我还是选择了为每个工作队列增加少量的内存开销，而不是增加为静态数据元素执行一次性初始化的较小复杂度。

18 成员 `first` 指向工作队列中的第一个元素。为使队列更容易在队尾增加新项目，`last` 成员指向队列中的最后一个元素。

19~24 这些成员记录了工作队列的各种信息。与先前的 `barrier` 以及读/写锁中的一样，`vaild` 成员是一个魔术数字（在这里，魔术数字是我女儿生日的年月），当工作队列被初始化时设置。`quit` 成员是一个标志，一旦排队为空，它允许“工作队列管理器”告诉引擎线程终止。`parallelism` 成员记录创建者允许工作队列使用多少线程，`counter` 记录创建的线程数，`idle` 记录当前正在等待工作的线程数。`engine` 成员是用户的“引擎函数”，当工作队列被创建时提供。正如你所见，引擎函数接收一个“无类型”的(`void*`)参数，并且没有返回值。

■ workq.h**part 1 workq_t**

```

1 #include <pthread.h>
2
3 /*
4  * Structure to keep track of work queue requests.
5  */
6 typedef struct workq_ele_tag {
7     struct workq_ele_tag          *next;
8     void                         *data;
9 } workq_ele_t;
10
11 /*
12  * Structure describing a work queue.
13  */
14 typedef struct workq_tag {
15     pthread_mutex_t      mutex;           /* control access to queue */
16     pthread_cond_t       cv;              /* wait for work */
17     pthread_attr_t       attr;            /* create detached threads */
18     workq_ele_t         *first, *last;   /* work queue */
19     int                 valid;           /* valid */
20     int                 quit;            /* workq should quit */
21     int                 parallelism;    /* maximum threads */
22     int                 counter;         /* current threads */
23     int                 idle;            /* number of idle threads */
24     void                (*engine)(void *arg); /* user engine */
25 } workq_t;
26
27 #define WORKQ_VALID      0xdec1992

```

■ workq.h**part 1 workq_t**

第 2 部分显示了我们将为工作队列创建的接口。我们需要创建和破坏工作队列管理器，因此将定义 `workq_init` 和 `workq_destroy`，二者都采用一个 `workq_t` 结构指针。另外，初始化函数还需要允许管理器创建的排队线程的最大数字和引擎函数。最后，程序需要能够将工作元素排队处理——我们将把该接口称为 `workq_add`。它包含一个 `workq_t` 指针和传递给引擎函数的参数。

■ workq.h

part 2 interfaces

```

1  /*
2   * Define work queue functions.
3   */
4  extern int workq_init (
5      workq_t      *wq,
6      int          threads,           /* maximum threads */
7      void         (*engine)(void *)); /* engine routine */
8  extern int workq_destroy (workq_t *wq);
9  extern int workq_add (workq_t *wq, void *data);

```

■ workq.h

part 2 interfaces

文件 workq.c 包含了工作队列的实现。下列例子分别显示了实现 workq.h 接口的各个函数。

第 1 部分显示了 workq_init 功能，它初始化一个工作队列。我们创建需要的 Pthreads 同步对象，并填写其他成员。

14~22 初始话线程属性对象 attr 以便我们创建可分离的 (detached) 引擎线程。这意味着我们不需要跟踪它们的线程标识符，或担心连接它们。

34~40 我们还没有准备好停止（我们几乎还没开始呢！），所以清除 quit 标志。parallelism 成员记录我们允许创建的最大线程数 (workq_init 参数线程)。counter 成员将记录当前活跃的引擎线程数，初始值为 0；idle 记录等待更多工作的活跃线程数。当然，最后我们要设置 valid 成员。

■ workq.c

part 1 workq_init

```

1 #include <pthread.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include "errors.h"
5 #include "workq.h"
6
7 /*
8  * Initialize a work queue.
9  */
10 int workq_init (workq_t *wq, int threads, void (*engine)(void *arg))
11 {
12     int status;
13
14     status = pthread_attr_init (&wq->attr);
15     if (status != 0)
16         return status;
17     status = pthread_attr_setdetachstate (
18         &wq->attr, PTHREAD_CREATE_DETACHED);
19     if (status != 0) {
20         pthread_attr_destroy (&wq->attr);
21         return status;
22     }
23     status = pthread_mutex_init (&wq->mutex, NULL);

```

```

24     if (status != 0) {
25         pthread_attr_destroy (&wq->attr);
26         return status;
27     }
28     status = pthread_cond_init (&wq->cv, NULL);
29     if (status != 0) {
30         pthread_mutex_destroy (&wq->mutex);
31         pthread_attr_destroy (&wq->attr);
32         return status;
33     }
34     wq->quit = 0;                                /* not time to quit */
35     wq->first = wq->last = NULL;                 /* no queue entries */
36     wq->parallelism = threads;                  /* max servers */
37     wq->counter = 0;                            /* no server threads yet */
38     wq->idle = 0;                               /* no idle servers */
39     wq->engine = engine;
40     wq->valid = WORKQ_VALID;
41     return 0;
42 }
```

■ workq.c

part 1 workq_init

第2部分显示了 `workq_destroy` 函数。关闭工作队列的过程与我们看到的其他关闭过程有所不同。记住当你试图破坏一个正在使用的对象时，Pthreads 互斥量和条件变量破坏函数失败并返回 `EBUSY`。我们为 barrier 和读/写锁使用了相同的模型，但是我们不能为工作队列这样做——调用程序无法知道工作队列是否在使用，因为调用者仅仅将请求排队，而请求异步地被处理。

工作队列管理器将随时接受一个关掉请求，但是它将等待所有现存的引擎线程完成它们的工作后终止。只有当最后的工作队列元素被处理且最后的引擎线程退出时，`workq_destroy` 才会成功返回。

如果工作队列没有线程，或者是因为它从来没被使用，或者是因为自从上次使用后所有的线程超时并关掉。那么事情变得容易了，我们可以省去所有的关闭复杂度。

如果有引擎线程，通过设置 `workq_t` 结构中的 `quit` 标志要求线程关闭自己，然后广播条件变量唤醒任何等待（闲散）的引擎线程。每个引擎线程将最终运行并且看见这个标志，当看见标志并且发现没有更多的工作时，它们将关掉自己。

最后关掉的线程将唤醒在 `workq_destroy` 上等待的线程，关闭完成。最后的线程会在相同的条件变量上发信号通知闲散的引擎线程做新工作，而不是创建一个只用来唤醒 `workq_destroy` 的条件变量。此时，所有的等待线程已经被广播唤醒，并且因为停止标志被设置，它们不会再等待。`Shutdown` 只在工作队列管理器的生命周期中发生一次，所以没有必要为这个目的创建另一个条件变量。

■ workq.c

part 2 workq_destroy

```

1  /*
2  * Destroy a work queue.
```

```
3  */
4  int workq_destroy (workq_t *wq)
5 {
6     int status, status1, status2;
7
8     if (wq->valid != WORKQ_VALID)
9         return EINVAL;
10    status = pthread_mutex_lock (&wq->mutex);
11    if (status != 0)
12        return status;
13    wq->valid = 0;                                /* prevent any other operations */
14
15 /*
16  * Check whether any threads are active, and run them down:
17  *
18  * 1.      set the quit flag
19  * 2.      broadcast to wake any servers that may be asleep
20  * 4.      wait for all threads to quit (counter goes to 0)
21  *          Because we don't use join, we don't need to worry
22  *          about tracking thread IDs.
23  */
24    if (wq->counter > 0) {
25        wq->quit = 1;
26        /* if any threads are idling, wake them. */
27        if (wq->idle > 0) {
28            status = pthread_cond_broadcast (&wq->cv);
29            if (status != 0) {
30                pthread_mutex_unlock (&wq->mutex);
31                return status;
32            }
33        }
34
35 /*
36  * Just to prove that every rule has an exception, I'm
37  * using the "cv" condition for two separate predicates
38  * here. That's OK, since the case used here applies
39  * only once during the life of a work queue -- during
40  * rundown. The overhead is minimal and it's not worth
41  * creating a separate condition variable that would
42  * wait and be signaled exactly once!
43  */
44    while (wq->counter > 0) {
45        status = pthread_cond_wait (&wq->cv, &wq->mutex);
46        if (status != 0) {
47            pthread_mutex_unlock (&wq->mutex);
48            return status;
49        }
50    }
51 }
52 status = pthread_mutex_unlock (&wq->mutex);
53 if (status != 0)
54     return status;
55 status = pthread_mutex_destroy (&wq->mutex);
```

```

56     status1 = pthread_cond_destroy (&wq->cv);
57     status2 = pthread_attr_destroy (&wq->attr);
58     return (status ? status : (status1 ? status1 : status2));
59 }

```

■ workq.c**part 2 workq_destroy**

第3部分显示了 `workq_add`, 它为排队管理器系统接受工作。

16~35 它分配一个新工作队列单元并用参数初始化它。它将该单元排队，并更新 `first` 和 `last` 指针。

40~45 如果有闲散的引擎线程（即已经被创建但是没有工作），在条件变量上发信号唤醒其中一个。

46~59 如果没有闲散的引擎线程，并且 `parallelism` 值允许，创建一个新的引擎线程。如果没有闲散的线程且不能创建一个新的引擎线程，`workq_add` 返回，将新单元留给下一个线程来完成它的当前任务。

■ workq.c**part 3 workq_add**

```

1  /*
2   * Add an item to a work queue.
3   */
4  int workq_add (workq_t *wq, void *element)
5  {
6      workq_ele_t *item;
7      pthread_t id;
8      int status;
9
10     if (wq->valid != WORKQ_VALID)
11         return EINVAL;
12
13     /*
14      * Create and initialize a request structure..
15      */
16     item = (workq_ele_t *)malloc (sizeof (workq_ele_t));
17     if (item == NULL)
18         return ENOMEM;
19     item->data = element;
20     item->next = NULL;
21     status = pthread_mutex_lock (&wq->mutex);
22     if (status != 0) {
23         free (item);
24         return status;
25     }
26
27     /*
28      * Add the request to the end of the queue, updating the
29      * first and last pointers.
30      */
31     if (wq->first == NULL)
32         wq->first = item;
33     else
34         wq->last->next = item;

```

```

35     wq->last = item;
36
37     /*
38      * if any threads are idling, wake one.
39      */
40     if (wq->idle > 0) {
41         status = pthread_cond_signal (&wq->cv);
42         if (status != 0) {
43             pthread_mutex_unlock (&wq->mutex);
44             return status;
45         }
46     } else if (wq->counter < wq->parallelism) {
47         /*
48          * If there were no idling threads, and we're allowed to
49          * create a new thread, do so.
50          */
51         DPRINTF (("Creating new worker\n"));
52         status = pthread_create (
53             &id, &wq->attr, workq_server, (void*)wq);
54         if (status != 0) {
55             pthread_mutex_unlock (&wq->mutex);
56             return status;
57         }
58         wq->counter++;
59     }
60     pthread_mutex_unlock (&wq->mutex);
61     return 0;
62 }
```

■ workq.c

part 3 workq_add

上述就是所有的外部接口，但我们还需要最后一个函数，即引擎线程的启动函数。该函数在第 4 部分显示，被称为 `workq_server`。尽管我们可以为每个请求启动一个线程，用适当的参数运行调用者的引擎，但这样做更有效。`workq_server` 函数将把下一个请求从队列中移出并传递给引擎函数，然后寻找新的工作。如果需要的话，它将等待；只有当某个时间内没有出现任何新的工作时，或当被 `workq_destroy` 告知时才关闭。

注意，服务器由锁住排队互斥量开始工作，并且匹配的解锁操作直到引擎线程准备终止时才发生。尽管如此，在线程的大部分时间里互斥量是解开的（或者在条件变量上等待工作，或者在调用线程的引擎函数内）。

29~62 当一个线程完成条件等待循环时，或者有工作可以做，或者队列正在被关掉 (`wq->quit` 非零)。

67~80 首先，我们检查是否有工作，如果有则处理工作队列单元。当 `workq_destroy` 被调用时，还仍然可能有工作被排队，而且都必须在任何引擎线程终止前被处理。用户的引擎函数在互斥量解锁时被调用，以便用户的引擎线程可以运行很长时间，或在不影响其他引擎线程运行的时候阻塞。这并不意味着引擎函数能够并行地运行——调用者提供的引擎函数负责确保需要的同步以允许期望的并发或并行级别。

理想的引擎函数将要求很少的或不需要同步，并且将并行执行。

86~104 当没有更多的工作且队列要被关闭时，线程终止。如果这是最后关闭的引擎线程，则唤醒 `workq_destroy`。

110~114 最后，我们检查寻找工作的引擎线程是否超时，这将意味着引擎线程已经等待足够长的时间。如果仍然没有发现要做的工作，引擎线程退出。

■ workq.c

part 4 workq_server

```

1  /*
2   * Thread start routine to serve the work queue.
3   */
4  static void *workq_server (void *arg)
5  {
6      struct timespec timeout;
7      workq_t *wg = (workq_t *)arg;
8      workq_ele_t *we;
9      int status, timedout;
10
11     /*
12      * We don't need to validate the workq_t here... we don't
13      * create server threads until requests are queued (the
14      * queue has been initialized by then!) and we wait for all
15      * server threads to terminate before destroying a work
16      * queue.
17     */
18     DPRINTF (( "A worker is starting\n" ));
19     status = pthread_mutex_lock (&wg->mutex);
20     if (status != 0)
21         return NULL;
22
23     while (1) {
24         timedout = 0;
25         DPRINTF (( "Worker waiting for work\n" ));
26         clock_gettime (CLOCK_REALTIME, &timeout);
27         timeout.tv_sec += 2;
28
29         while (wg->first == NULL && !wg->quit) {
30             /*
31              * Server threads time out after spending 2 seconds
32              * waiting for new work, and exit.
33             */
34             status = pthread_cond_timedwait (
35                 &wg->cv, &wg->mutex, &timeout);
36             if (status == ETIMEDOUT) {
37                 DPRINTF (( "Worker wait timed out\n" ));
38                 timedout = 1;
39                 break;
40             } else if (status != 0) {
41                 /*
42                  * This shouldn't happen, so the work queue
43                  * package should fail. Because the work queue
44                  * API is asynchronous, that would add

```

```
45      * complication. Because the chances of failure
46      * are slim, I choose to avoid that
47      * complication. The server thread will return,
48      * and allow another server thread to pick up
49      * the work later. Note that if this were the
50      * only server thread, the queue wouldn't be
51      * serviced until a new work item is
52      * queued. That could be fixed by creating a new
53      * server here.
54      */
55      DPRINTF ((
56          "Worker wait failed, %d (%s)\n",
57          status, strerror (status)));
58      wq->counter--;
59      pthread_mutex_unlock (&wq->mutex);
60      return NULL;
61  }
62 }
63 DPRINTF (("Work queue: 0x%p, quit: %d\n",
64     wq->first, wq->quit));
65 we = wq->first;
66
67 if (we != NULL) {
68     wq->first = we->next;
69     if (wq->last == we)
70         wq->last = NULL;
71     status = pthread_mutex_unlock (&wq->mutex);
72     if (status != 0)
73         return NULL;
74     DPRINTF (("Worker calling engine\n"));
75     wq->engine (we->data);
76     free (we);
77     status = pthread_mutex_lock (&wq->mutex);
78     if (status != 0)
79         return NULL;
80 }
81 /*
82  * If there are no more work requests, and the servers
83  * have been asked to quit, then shut down.
84  */
85 if (wq->first == NULL && wq->quit) {
86     DPRINTF (("Worker shutting down\n"));
87     wq->counter--;
88
89 /*
90  * NOTE: Just to prove that every rule has an
91  * exception, I'm using the "cv" condition for two
92  * separate predicates here. That's OK, since the
93  * case used here applies only once during the life
94  * of a work queue -- during rundown. The overhead
95  * is minimal and it's not worth creating a separate
96  * condition variable that would wait and be
97 }
```

```

98         * signaled exactly once!
99         */
100        if (wq->counter == 0)
101            pthread_cond_broadcast (&wq->cv);
102            pthread_mutex_unlock (&wq->mutex);
103            return NULL;
104        }
105    /*
106     * If there's no more work, and we wait for as long as
107     * we're allowed, then terminate this server thread.
108     */
109    if (wq->first == NULL && timeout) {
110        DPRINTF (("engine terminating due to timeout.\n"));
111        wq->counter--;
112        break;
113    }
114}
115}
116
117 pthread_mutex_unlock (&wq->mutex);
118 DPRINTF (("Worker exiting\n"));
119 return NULL;
120}

```

■ workq.c

part 4 workq_server

最后, workq_main.c 是使用我们的工作排队管理器的一个实例。两个线程并行地向工作队列中加入工作单元。引擎函数被设计为收集一些关于引擎使用的统计数据。为完成该工作, 它使用了线程私有数据。当实例运行完时, 主函数收集所有的线程私有数据并报告一些统计信息。

15~19 每个引擎线程有一个与线程私有数据键 engine_key 相关联的 engine_t 结构。引擎函数从调用线程那得到该键的值, 如果当前值为空, 则创建新的 engine_t 结构并且把它分派给键。engine_t 结构的 calls 成员记录每个线程调用引擎函数的次数。

29~37 线程私有数据键的 destructor 函数, 把终止线程的 engine_t 添加到一张表中 (engine_list_head), 主线程随后可以用来产生最后的报告。

43~68 引擎函数的工作相对是令人厌烦的。参数是一个 power_t 结构指针, 包含 value 和 power 成员。它使用一个循环来获得 value 的 power 次方值。结果在本例中被丢弃, power_t 结构被释放。

73~98 主函数启动一个线程, 执行 thread_routine 函数。另外, 主函数调用 thread_routine.thread_routine 函数循环若干次(由 ITERATIONS 宏决定), 创建并且排队工作队列单元。power_t 结构的 value 和 power 成员使用 rand_r 半随机产生。函数随机睡眠一段时间 (0~4 秒), 以允许引擎线程偶尔超时并终止。当运行本例时, 你可能期望看见报告少量引擎线程的摘要信息, 每个线程处理一些调用——总计是 50 个调用。

■ workq_main.c

```
1 #include <pthread.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <time.h>
5 #include "workq.h"
6 #include "errors.h"
7
8 #define ITERATIONS      25
9
10 typedef struct power_tag {
11     int          value;
12     int          power;
13 } power_t;
14
15 typedef struct engine_tag {
16     struct engine_tag *link;
17     pthread_t           thread_id;
18     int                calls;
19 } engine_t;
20
21 pthread_key_t engine_key;           /* Keep track of active engines */
22 pthread_mutex_t engine_list_mutex = PTHREAD_MUTEX_INITIALIZER;
23 engine_t *engine_list_head = NULL;
24 workq_t workq;
25
26 /*
27  * Thread-specific data destructor routine for engine_key.
28  */
29 void destructor (void *value_ptr)
30 {
31     engine_t *engine = (engine_t*)value_ptr;
32
33     pthread_mutex_lock (&engine_list_mutex);
34     engine->link = engine_list_head;
35     engine_list_head = engine;
36     pthread_mutex_unlock (&engine_list_mutex);
37 }
38
39 /*
40  * This is the routine called by the work queue servers to
41  * perform operations in parallel.
42  */
43 void engine_routine (void *arg)
44 {
45     engine_t *engine;
46     power_t *power = (power_t*)arg;
47     int result, count;
48     int status;
49
50     engine = pthread_getspecific (engine_key);
51     if (engine == NULL) {
```

```
52         engine = (engine_t*)malloc (sizeof (engine_t));
53         status = pthread_setspecific (
54             engine_key, (void*)engine);
55         if (status != 0)
56             err_abort (status, "Set tsd");
57         engine->thread_id = pthread_self ();
58         engine->calls = 1;
59     } else
60         engine->calls++;
61     result = 1;
62     printf (
63         "Engine: computing %d^%d\n",
64         power->value, power->power);
65     for (count = 1; count <= power->power; count++)
66         result *= power->value;
67     free (arg);
68 }
69
70 /*
71  * Thread start routine that issues work queue requests.
72  */
73 void *thread_routine (void *arg)
74 {
75     power_t *element;
76     int count;
77     unsigned int seed = (unsigned int)time (NULL);
78     int status;
79
80     /*
81      * Loop, making requests.
82      */
83     for (count = 0; count < ITERATIONS; count++) {
84         element = (power_t*)malloc (sizeof (power_t));
85         if (element == NULL)
86             errno_abort ("Allocate element");
87         element->value = rand_r (&seed) % 20;
88         element->power = rand_r (&seed) % 7;
89         DPRINTF ((
90             "Request: %d^%d\n",
91             element->value, element->power));
92         status = workq_add (&workq, (void*)element);
93         if (status != 0)
94             err_abort (status, "Add to work queue");
95         sleep (rand_r (&seed) % 5);
96     }
97     return NULL;
98 }
99
100 int main (int argc, char *argv[])
101 {
102     pthread_t thread_id;
103     engine_t *engine;
```

```
104     int count = 0, calls = 0;
105     int status;
106
107     status = pthread_key_create (&engine_key, destructor);
108     if (status != 0)
109         err_abort (status, "Create key");
110     status = workq_init (&workq, 4, engine_routine);
111     if (status != 0)
112         err_abort (status, "Init work queue");
113     status = pthread_create (&thread_id, NULL, thread_routine, NULL);
114     if (status != 0)
115         err_abort (status, "Create thread");
116     (void)thread_routine (NULL);
117     status = pthread_join (thread_id, NULL);
118     if (status != 0)
119         err_abort (status, "Join thread");
120     status = workq_destroy (&workq);
121     if (status != 0)
122         err_abort (status, "Destroy work queue");
123
124     /*
125      * By now, all of the engine_t structures have been placed
126      * on the list (by the engine thread destructors), so we
127      * can count and summarize them.
128      */
129     engine = engine_list_head;
130     while (engine != NULL) {
131         count++;
132         calls += engine->calls;
133         printf ("engine %d: %d calls\n", count, engine->calls);
134         engine = engine->link;
135     }
136     printf ("%d engine threads processed %d calls\n",
137             count, calls);
138     return 0;
139 }
```

■ workq_main.c

7.3 对现存库的处理

*"The great art of riding, as I was saying is—
to keep your balance properly. Like this, you know—"
He let go the bridle, and stretched out both his arms to
show Alice what he meant, and this time he fell flat on
his back, right under the horse's feet.
—Lewis Carroll, Through the Looking-Glass*

当创建一个新的函数库时，你要做的就是小心设计以保证函数库是线程安全的。你可以决定函数需要什么状态，你可以决定哪个状态需要在线程间共享，哪个状态应该被调用者通过外部环境管理，哪个状态可以被保存在本地变量中等，你可以定

义接口函数以最有效的方式支持那个状态。但是当你正在修改一个现存的函数库来与线程工作时，通常没有那么运气。而当你正在使用其他人的函数库时，可能只需要简单地去使用。

7.3.1 将函数库修改为线程安全的

许多函数依赖于跨越一系列调用的静态存储，如 `strtok`。其他函数依赖于返回一个指向静态存储的指针，如 `asctime`。本节使用 ANSI C 执行库中一些著名的例子，指出当你需要将“遗留”（legacy）库变为线程安全时，可以帮助你的一些技术。

最简单的技术是把一个互斥量指派到每个子系统。当任何调用进入子系统时，锁住互斥量；从子系统退出时，解锁互斥量。因为单个的互斥量涵盖了全部子系统，我们经常将该机制称为“大互斥量”（见 3.2.4 节）。互斥量阻止了多个线程同时在子系统内执行。注意，这种方法仅解决同步竞争，而不是顺序竞争（8.1.2 节描述了二者间的区别）。实现该方法的最好候选函数是：除维持一些内部数据库外做很少工作的函数。这也包括如 `malloc` 和 `free` 之类的函数，它们管理一个内部的资源池但是限制（或禁止）资源池的外部可见性。

使用“大互斥量”的一个问题就是必须对“子系统”的定义很小心。你需要包括共享数据的或相互调用的所有函数。如果 `malloc` 和 `free` 使用一个互斥量，而 `realloc` 使用另外一个时，则一旦一个线程调用 `realloc` 而其他线程调用 `malloc` 或 `free` 时，就会遇到竞争。

如果 `realloc` 被实现为调用 `malloc`、拷贝数据，然后在旧指针上调用 `free`，会怎么样？`realloc` 函数将锁住堆互斥量并且调用 `malloc`。`malloc` 函数将很快地自己试图锁住堆互斥量，导致死锁。有若干个解决方法。一个是将每个外部接口小心地分成两部分：一个做实际工作的内部引擎函数和一个锁住子系统互斥量并调用引擎函数的外部入口点。子系统内需要调用同一引擎函数的其他入口点将直接调用该函数而非使用正常的入口点。这通常是最有效的方法，但也是更难实现的。另外可能的方法是构造一个递归的互斥量以允许子系统重新锁住自己的互斥量而不会死锁^①。现在 `malloc` 和 `free` 被允许重新加锁 `realloc` 拥有的互斥量，但是试图调用它们的其他线程将被堵住直到 `realloc` 完全释放递归的互斥量。

大多数拥有持久状态的函数要求比只是“大互斥量”更本质的变化，尤其是避免改变接口。例如，`asctime` 函数返回一个表示二进制时间的字符串指针。传统的

^① 很容易构造一个“递归互斥量”：用一个互斥量、一个条件变量、一个表示当前“拥有线程”（拥有递归互斥量的线程）的 `Pthread_t` 值，一个表示当前“拥有线程”递归深度的计数变量。如果深度为 0，表示递归互斥量没有被锁住；大于 0 则表示被锁住。互斥量用来保护对递归深度和当前“拥有线程”变量的访问；条件变量用来等待递归深度变为 0（如果一个线程想锁住递归互斥量，而同时另一个线程已经锁住该互斥量时）。

实现是，字符串被格式化到一个在 `asctime` 函数内部声明的静态缓冲区中，并返回该缓冲区指针。

在 `asctime` 内锁住一个互斥量不足以保护数据，事实上，它甚至不是特别有用。在 `asctime` 返回后，互斥量被解锁。调用者需要读缓冲区，并且没有任何保护阻止其他线程在第一个线程完成读或拷贝数据以前调用 `asctime`（这将“破坏”第一个线程的结果）。解决使用一个互斥量的问题，调用者需要在调用 `asctime` 前锁住一个互斥量，然后仅在完成读或拷贝缓冲区的数据后才解锁。

相反地，这个问题可以通过重新编码 `asctime` 来解决，即让 `asctime` 使用 `malloc` 分配一个堆缓冲区，格式化时间串到那个缓冲区中，并返回缓冲区地址。函数可以使用线程私有数据键来追踪堆地址，以便它能在线程内的下一个调用时被再次使用。当线程终止时，`destructor` 函数将释放存储。

避免使用 `malloc` 和线程私有数据将是更有效的，但是那将要求改变到 `asctime` 的接口。Pthreads 使用一个新的线程安全函数 `asctime_r` 替代 `asctime`，它要求调用者传递一个缓冲区的地址和长度。`asctime_r` 函数格式化时间串进调用者的缓冲区中，这允许调用线程以任何方便的方式管理缓冲区，它可以在线程的栈中、堆中或甚至在线程之间被共享。尽管某种程度上该方法是在放弃现存的函数并定义新函数，但它经常是最好的方法（并且有时是惟一实际的方法）使函数线程安全。

7.3.2 使用遗留库

有时你不得不使用不是你写的代码，并且不能变化代码。很多代码现在正在成为线程安全的，并且支持线程的大多数操作系统将能够提供通用函数库的线程安全实现。随着更多的开发人员要求线程安全的函数库，线程安全的函数库作为核心集团将逐渐增加并成为规则而非例外。

但是你将不可避免地发现你会需要一个非线程安全的函数库，如一个旧的 X Windows 系统版本，或一个数据库引擎，或模拟包，而且你不会有它们的源代码。当然你可以很快地向库供应商抱怨并说服他们使下一个版本充分线程安全。但是在新版本到来以前你能做什么呢？

如果你确实需要一个库，答案是“随便使用它”。你有使用它的很多技术，从简单到复杂。适当的复杂性水平全部取决于库的接口和如何在你的代码中使用库。

| 将不安全的库转变为一个服务者线程。

在一些情况中，你可能觉得把库的使用限制到一个线程内会很方便，使那个线程成为一个代表不安全库提供能力的“服务器”。这技术通常被使用，例如当使用非线程安全的 X11 协议顾客库的版本时。主线程或某个线程（就是为该目的创建的）代表其他线程的利益处理排队的 X11 请求。只有服务者线程调用 X11 库，因此 X11 是否线程安全没有关系。

| 在接口附近编写自己的“大互斥量”包装器代码。

如果你需要的函数有一个“线程安全的接口”而非“线程安全的实现”时，你也许可以将每个调用封装到包装器函数（或一个宏）中，在包装器中将首先锁住一个互斥量，调用函数，然后解锁互斥量。这只是“大互斥量”的一个扩展版本。在这里，“线程安全的接口”意味着函数依靠于静态的状态，但是任何返回给调用者的数据不能被以后的调用改变。例如，`malloc` 就是属于这一范畴。存储器的分配包含需要被保护的静态数据，一旦内存块被分配给一个调用者，那个地址（和指向的存储器）将不会被以后的 `malloc` 调用影响。对于像 X11 或任何其他网络协议之类的可能阻塞较长时间的函数库而言，扩展的“大互斥量”不是一个好方法。尽管结果可能是安全的，但它将是很低效的，除非你很少使用函数库。因为当远程操作正在进行时，其他线程可能被锁住很长一段时间。

| 使用外部状态扩展实现。

“大互斥量”不能修复像 `asctime` 之类的函数。`asctime` 函数将数据写进一个静态缓冲区中并返回缓冲区地址；返回的数据必须被保护直到调用者使用完它，并且数据在包装器之外被使用。对于像 `strtok` 这样的函数，数据被使用直到全部的标志序列被分析。总的来说，具有永久的静态数据的函数更困难于封装。

像 `asctime` 之类的函数可以通过创建以下包装器来封装：加锁一个互斥量，调用函数，拷贝返回值到一个线程安全的缓冲区，解锁互斥量，然后返回。线程安全的缓冲区能被包装器使用 `malloc` 动态地分配工作。当完成时，你可以要求调用者释放缓冲区，或者你可以让包装器使用线程私有数据为每一个线程保持一个缓冲区。

另外，你可以发明一个要求调用者提供缓冲区的新接口。调用者可以使用一个栈缓冲区，或一个堆缓冲区，或者，如果适当同步的话（由调用者决定），可以在线程之间共享缓冲区。记住，如果包装器使用线程私有数据跟踪线程私有的堆缓冲区，包装器就能与原来的接口兼容。其他变量要求接口变化：调用者必须提供不同的输入或必须知道需要释放返回的缓冲区。

一个要在一系列调用间保持永久状态的函数更难于封装得好。静态数据必须被通盘保护。实现这一点最容易的方法是简单地改变调用者：在第一次调用前锁住一个互斥量，并使其被锁直到最后一个调用为止。但是要记住，在互斥量被解锁前其他线程都不能使用函数。如果调用者在调用之间处理大量数据，一个主要的处理瓶颈将发生。当然，这种方法也可能很难或不可能集成到一个简单的包装器中——因为包装器必须能够识别第一个和最后一个函数调用。

一个更好但更难的方法是找到某种方式将函数（或一组相关函数）封装进一个新的线程安全接口。对于此类转变没有通用的模型，而且在许多情况下它可能是不现实的。但是通常你就需要有创造力，并且可能应用一些限制。当库函数不能被容

易地封装时，你也许能封装你需要的特殊情况。例如，尽管 `strtok` 允许你在每次调用时改变标志分隔符，但大多数代码都没有利用这一灵活性。没有了分隔符变化的复杂性，你可以在 `strtok` 上定义一个新的标志解析模型，让一个线程安全的安装函数寻找字符串中所有的标志，并将它们存储到一个不需调用 `strtok` 就能获取的地方。这样，由于安装函数将锁住一个普通的互斥量并在所有线程间序列化访问，因此信息检索函数可以不需要任何序列化执行。

8

避免调试的提示

*"Other maps are such shapes, with their islands and capes!
But we've got our brave Captain to thank"
(So the crew would protest) "that he's bought us the best—
A perfect and absolute blank!"*

—Lewis Carroll, *The Hunting of the Snark*

写一个复杂的多线程程序比写一个简单的同步程序难得多，但是一旦你学会了规则，它不比写一个复杂的同步程序难。写一个多线程程序来执行复杂的异步函数，通常比使用更传统的异步编程技术写同样的程序容易。

当需要调试或分析你的多线程程序时，复杂度产生。并不是因为使用线程那么难，而相反是因为线程代码的调试和分析工具没有像编程接口那样被开发得很好且容易理解。你可能感到自己好像在一张空白的地图上遨游。这并不意味着你现在不能利用多线程编程的力量，但是这确实意味着你需要小心，并且也许更要有了一些创造性，来避免未知水域中的岩石和浅滩。

尽管本章提及了一些线程调试和分析工具，并建议你能用它们完成什么，但我的目标不是告诉你解决问题的工具本身。相反，我将描述你可能遇到的问题，并在你必须调试它们之前，给予你一些避免问题的“忠告”——或者，也许更实际地，告诉你如何识别出可能要遇见的问题。

| 在门口检查你的假设。

多线程编程对你可能是新的，异步编程对你可能也是新的。如果这样，你需要小心对待你的假设。你已经穿过了一座桥，并且在同步陆地上可接受（甚至被要求）的行为，在河对岸的异步陆地就可能是危险的。你可以容易地学习新规则，并且在实践中甚至可能感到很舒适。但是你开始时必须牢记一些东西已经变化了。

8.1 避免不正确的代码

"For instance, now," she went on, sticking a large piece of plaster on her finger as she spoke, "there's the King's Messenger. He's in prison now, being punished: and the trial doesn't even begin till next Wednesday: and of course the crime comes last of all."

"Suppose he never commits the crime?" said Alice.

"That would be all the better, wouldn't it?" the Queen said, as she bound the plaster round her finger with a bit of ribbon.

—Lewis Carroll, *Through the Looking-Glass*

Pthreads 没有为调试多线程代码提供许多帮助。那并不让人吃惊，因为 POSIX 根本不承认调试的概念，甚至在解释为什么标准没有包括几乎通用的 SIGTRAP 信号时也是如此。尽管每个多线程系统将提供某种形式的调试工具，但没有标准的方法来与你的程序交互，或当它运行时观察其行为。即使（不大可能的情况）系统开发者不关心你这个可怜的程序员，他们也需要调试他们自己的代码。

给线程提供操作系统的供应商将至少在调试工具中提供一个基本的线程“观察窗口”。至少你应该可以获得显示正在运行的线程及其当前状态的能力，以及显示互斥量和条件变量状态，并显示所有线程堆栈跟踪窗口的一张表。你应该也能为指定线程设置断点并指定一个“当前线程”来检验寄存器、变量和堆栈跟踪窗口。

因为 Pthreads 实现可能在用户模式的进程内维持很多状态，像 ptrace 或 proc 文件系统这样传统的 UNIX 调试机制可能难于发挥作用。一个通用的解决方案是提供一个被调试器调用的专用库，它知道如何在整个被调试进程地址空间中搜索线程和同步对象的状态。例如，Solaris 提供了 libthread_db.so 共享库，而 DIGITAL UNIX 提供了 libpthreaddebug.so 库。

在第三方操作系统的基础上实现的线程包将不能提供太多与调试器的集成。例如，便携式“DCE 线程”库提供了一个内建的调试命令解析器，你可以使用 print 或 call 命令激活调试器，来报导进程内的线程和同步对象的状态^①。这个受限的调试支持充其量是不方便的——你不能在一个程序失败后在 core 文件中分析线程状态，而且它不能理解（或报告）程序变量的符号名字。

下列各节描述了一些最常见的线程编程错误，目的是帮助你在设计时避免这些问题，同时可能使它们在调试时更容易被确认。

8.1.1 避免依赖“线程惯量”

记住：线程总是异步的。当你在可能“稍微同步”的单处理机系统上开发线

^① 因为历史原因，该功能被称为 cma_debug。如果你发现自己被困在 DCE 线程代码中，可以试试这个功能。输入 help 命令可以获得更多的命令列表。

程代码时，记住这点尤其重要。没有东西同时在一台单处理机上发生，准备就绪的线程被连续地以相对可知的间隔分时执行。当你在一台单处理机上创建一个新线程时，或唤醒一个在互斥量或在条件变量上等待的线程时，除非比创建它或唤醒它的线程有更高的优先级，否则它不能立刻运行。如果你达到了“进程的并发限制”，同样的现象甚至可能在一台多处理机上发生，例如，当就绪线程超过处理器数目时。如果有相等的优先级，创建线程或唤醒它的其他线程将继续运行直到被堵住或直到下一个时间片（可能是许多毫微秒后）。

这意味着，对于当前只有一个处理器的线程有一个优点。它具有不断运动的趋势，显示出与物理惯性有点类似的行为。结果是，你可能会侥幸逃脱这样潜在的错误：如果新创建或唤醒的线程能立刻运行——当有空闲的处理器时，它将导致代码神秘地失败。下列程序 `inertia.c` 显示了该现象如何能破坏你的程序。

27~41 问题是线程函数 `printer_thread` 是否将看到 `stringPtr` 的值，该值或者在 `pthread_create` 调用之前被设置，或者在 `pthread_create` 调用之后被设置。期望的值是“After value”。这是一类很常见的编程错误。当然，在大多数情况下问题不会像这个简单例子那么明显。通常，变量是未初始化的，没有设置为合适的值，而结果可能是数据破坏或内存分段错。

39 现在注意延期循环。即使在一台多处理机上，这个程序也不会总是失败。在新线程能够开始执行之前，程序通常会改变 `stringPtr`——不管怎么说，需要花一段时间让新建线程进入你的代码，而在该特殊程序中发生该现象的“机会窗口”仅仅是几条指令。该循环通过推迟主线程给打印线程足够长的时间开始来演示这个问题。如果将给循环设置得足够长，甚至在一台单处理机上也将会看见这个问题的发生（如果主线程最终被时间片机制抢占）。

■ `inertia.c`

```
1 #include <pthread.h>
2 #include "errors.h"
3
4 void *printer_thread (void *arg)
5 {
6     char *string = *(char**)arg;
7
8     printf ("%s\n", string);
9     return NULL;
10 }
11
12 int main (int argc, char *argv[])
13 {
14     pthread_t printer_id;
15     char *string_ptr;
16     int i, status;
17
18 #ifdef sun
19     /*
```

```
20      * On Solaris 2.5, threads are not timesliced. To ensure
21      * that our two threads can run concurrently, we need to
22      * increase the concurrency level to 2.
23      */
24  #ifndef _P_H_
25      DPRINTF (( "Setting concurrency level to 2\n" ));
26      thr_setconcurrency ( 2 );
27  #endif
28      string_ptr = "Before value";
29      status = pthread_create (
30          &printer_id, NULL, printer_thread, (void*)&string_ptr );
31      if (status != 0)
32          err_abort (status, "Create thread");
33
34      /*
35      * Give the thread a chance to get started if it's going to run
36      * in parallel, but not enough that the current thread is likely
37      * to be timesliced. (This is a tricky balance, and the loop may
38      * need to be adjusted on your system before you can see the bug.)
39      */
40      for (i = 0; i < 10000000; i++);
41
42      string_ptr = "After value";
43      status = pthread_join (printer_id, NULL);
44      if (status != 0)
45          err_abort (status, "Join thread");
46  }
```

■ inertia.c

修复 inertia.c 的一个方法就是在创建线程之前设置 After value 为你希望线程看到的值。这不是很难，对吧？可能仍然有一个 Before value，或者是未初始化的变量或者是事先用于一些其他目的的值，但是你创建的线程无法看见它。依据 3.4 节中的内存可视性规则，新线程可以看到所有在 pthread_create 调用前发生的内存写。因此，总是将你的代码设计成这种方式：在线程需要的资源被创建并恰当地初始化后，才让线程启动。

| 绝不要假设你创建的线程会等待你。

假设一个线程不会很快地运行，与假定它会很快地运行将给你带来同样多的问题。创建一个依赖于创建线程中的“中间存储”的线程几乎总是一个坏想法。我看到过这样的代码：创建一系列线程，把指向同一个本地结构的指针传给每个线程，每次改变结构成员的值。问题是不能假设线程将以任何特定的顺序开始。所有线程可能在最后一个创建调用后开始执行，这样它们都会得到数据的最后值。或者线程的启动可能有点乱，这样第一个和第二个线程将得到相同的数据，但是其他线程得到你期望它们得到的数据值。

线程惯性是线程竞争中的一个特殊情况。尽管线程竞争在 8.1.2 节被更广泛地讨论，线程惯性则是一个微妙的效果，而且许多人不认为它是竞争。因此，如果可

能的话，在一台多处理机上彻底地测试你的代码。在开发过程中尽早地做这件工作，并且要连续地贯穿整个开发过程。即使在一台多处理机上测试将比在一台单处理机上调试更困难，特别当没有一个完美的线程调试器时，也应该这样做。当然，你应该仔细地阅读下列各节。

8.1.2 别将你的赌押在线程竞争上

当两个以上的线程同时试图得到一些地方或做一些东西时，竞争发生。只有一个能赢。哪个线程赢由很多因素决定，并不是一切都在你的控制下。结果可以受到下列因素的影响：系统中有多少处理器，有多少其他进程正在运行，系统正在处理多少网络开销，以及此类的其他事情。这些是不可确定的竞争。如果你连续两次不断地执行同样的程序，它可能出现不一样的结果。你不该为这样的竞争做假设^①。

当你写多线程代码时，应该假定在任意点上、在程序的任何语句内，每个线程可能睡眠一段不可知的时间。

处理器可以以不同的速率执行你的线程，这取决于处理器的负担、中断等。处理器上的时间片机制可以在任意点上将线程打断一段未指定的时间。当一个线程不在运行时，任何其他线程可以运行并且做（代码的同步协议没有具体阻止它做的）任何东西。这意味着对于不同的活跃线程集，线程在两条指令间可能发现一幅完全不同的存储器图像。防止一个线程以奇怪的方式看世界的方法只能依靠线程之间的显式同步。

如果你在一台多处理机上调试，大多数同步问题将会很快地显示出来。没有足够同步的线程将为最后到达内存的荣誉而竞争。有些讽刺的是，因为内存系统将保留最后的值，“输家”（最后的到达者）通常在线程竞争中取胜。有时你根本不会注意竞争，但是有时你将得到令人奇怪的错误结果，而有时你将会得到内存分段错误。

竞争通常是难于诊断的。因为竞争要求并发执行，经常这个问题根本不会在一个单处理机系统上发生。在一台单处理机上，即使有时间片机制，其并发水平也是相当低的；而且，一些未同步的写操作经常在其他线程有机会读不一致数据之前就完成了。甚至在一台多处理机上，竞争也可能是很难重复的，而且它们经常拒绝向调试器揭示自己。竞争取决于线程执行的相对时序——这些很可能在调试器中被改变。

竞争更多与内存可视性相关（与多个写操作的同步相比）。还记得内存可视性的基本规则（见 3.4 节）：一个线程总是能看到被同一处理器上的前面执行的线程做的存储器变化。在一台单处理机上所有的线程在同样的处理器上执行，这使在调试期间检测内存可见性问题变得困难。在一台多处理机上，只有当线程被安排

^① 我的女儿在 3 岁时就已经指出了这一点——当她想要竞争时，她会提前告诉我而不管我的工作会怎样。在这么重要的事情上听天由命真的没有用！

到不同的处理器上执行特定的脆弱（易被破坏的）代码段时，才可能看到可见性竞争。

| 除非你导致排序，线程间不存在顺序。

| 比尔·加尔梅斯特的推论：“线程将可能以最邪恶的顺序执行”。

你一定不希望自己要调试线程竞争。你可能从来不会看见一样的结果两次。当你试图调试代码时，症状将变化——可能化装为一种不同类型的问题，而且发生在不同地方。甚至更坏的是，这个问题可能根本从来没有发生直到一个顾客运行代码，然后它可能每次都失败，但是仅仅在顾客大块的应用程序中发生，而且仅仅在它运行了几天后发生。它可能是要在在一个没有网络存取的保密系统上运行，他们不可能给你看专利代码，并且不能使用一个简单的测试程序重现这个问题。

| 调度和同步不一样。

最初看上去，将一个线程置为 SCHED_FIFO 调度策略和最大优先级将允许你避免使用昂贵的同步机制（通过确保没有其他线程能够运行直到线程阻塞自己或降低自己的优先级）。这种方法有若干问题，但是主要的问题是它不能在一台多处理机上良好工作。SCHED_FIFO 策略阻止其他线程抢占当前线程，但是在一台多处理机上其他线程可以不需任何形式的抢占运行。

调度告诉系统一个特定工作（线程）对于你的应用程序如何重要，因此它能安排你最需要的工作。同步告诉系统没有其他线程能被允许进入临界区直到调用线程退出。

在真实的生活中，确定性的竞争（获胜者从一开始就被确定）不是很令人激动的（除非对一个三岁的小孩）。但是确定性的竞争实质上代表更安全的赌注，并且那也是你想要在程序中设计的竞争类型。正如你可能猜测的那样，确定性的竞争根本就不完全是竞争。它们更像是有序地等待，组织得很好且可预测。复杂度被过高估计，特别当调试复杂的多线程应用时。

最简单形式的竞争是当多个线程在没有合适的同步前提下试图写共享状态时，例如，当两个线程增加一个共享计数器的值时。两个线程可以从内存取一样的值，独立地增加它，然后保存一样的结果到内存中；计数器被增加了 1 而非 2，并且两个线程有同样的结果。

当一个线程正在写共享数据而同时其他线程在读那个数据时，更微妙的竞争将发生。如果读操作以不同的顺序发生，或如果读线程赶上了写线程，则读线程可能得到不一致的结果。例如，一个线程增加一共享数组的索引然后在那个索引处的数据单元写入数据。其他线程在写线程填完全部元素之前获得共享的索引，并且读那个元素。因为数组单元还没有被完全建立，读线程将发现不一致的数据，它可能因为它看见的数据或可能跟随一个错误的指针去执行一条意外的代码路径。

在设计和编码时总是假设线程与你想像的更同步。有经验的程序员知道计算机

有一些乐于使你愤怒的小东西。记住当你使用线程编码时，有更多的小东西被放纵。别心存侥幸，别做什么假设。保证任何共享状态在使用它的线程被创建之前就被建立并且可见；或使用静态互斥量或 `pthread_once` 来创建它。使用一个互斥量保证线程不能读不一致的数据。如果你必须在线程之间共享栈数据，要确定使用数据的所有线程在从分配内存的函数返回前终止。

当你假设事件按照一定的顺序发生，但没有把顺序编码到应用程序时，顺序竞争可能发生。甚至当你小心地使用同步控制保证数据一致性时，顺序竞争仍然可能发生。你只能通过确保这种竞争不重要，或增加强迫任何事情以需要发生的顺序发生的代码，才能保证避免此类竞争。

例如，想像一下三个线程共享一个计数器变量。每个线程将保存当前值的一个私有拷贝并增加共享的计数器。如果三个线程正在执行相同的函数，并且他们不在乎得到的计数器的值，则在取值和增加操作附近锁住一个互斥量就足够了。该互斥量确保每个线程获得不同的值，而且没有值被忽略。因为线程不在乎谁会赢，所以没有竞争。

但是如果在意每个线程读到哪个值，则这个简单的代码将无法工作。例如，你可能想像线程以它们被创建的顺序启动，以便第一个线程得到值 1、第二个得到值 2 等。有时（可能当你调试时）线程将得到你期望的值，而且一切正常工作；而在其他时间，线程将碰巧以不同的顺序执行。

有解决该问题的若干方法。例如，通过在创建计数器的线程中增加计数器的值并以数据结构把适当的值传递给每个线程，你可以先给各个线程分配合适的值。不过，最好的方法是通过设计代码以使线程启动顺序没有关系来避免这个问题。你的线程越对称，对环境做的假设越少，这种竞争发生的机会越少。

竞争不总是发生在内存地址引用的水平上，它们能在任何地方发生。例如，当你在多线程应用程序中使用传统的 ANSI C 库中的某个函数时，将导致很多顺序竞争。例如，`readdir` 函数在函数内部依靠静态存储区在一系列相同的 `readdir` 调用间维持环境。如果一个线程调用 `readdir`，而其他线程正在自己的 `readdir` 调用序列中间时，静态存储将被新的环境覆盖。

| 甚至当所有的代码都使用互斥量保护共享数据时，“顺序竞争”也可能发生！

即使 `readdir` 是“线程相关”的并且通过锁住一个互斥量来保护静态存储，顺序竞争还会发生。它不是同步竞争，而是顺序竞争。例如，线程 A 可能调用 `readdir` 扫描目录 `/usr/bin`，它锁住互斥量，返回第一个入口，然后解锁互斥量。然后线程 B 可能调用 `readdir` 扫描目录 `/usr/include`，它也锁住互斥量，返回第一个入口，然后解锁互斥量。现在线程 A 再次调用 `readdir` 期望获得 `/usr/bin` 的第二个入口；但是相反地它获得了 `/usr/include` 中的第二个入口，没有接口函数表现地不适当，但是结果却是错误的。`readdir` 接口就是不适合线程使用。

因此，Pthreads 指定了一套新的线程可重入的函数，其中包括 `readdir_r`，它包

括一个用来跨调用维持环境的附加参数。附加参数通过避免对共享数据的任何需要来解决顺序竞争。线程 A 对 `readdir_r` 的调用将在 A 的缓冲区中返回 /usr/bin 的第一个入口，而线程 B 对 `readdir_r` 的调用将在 B 的缓冲区中返回 /usr/include 中的第一个入口。现在，线程 A 的第二次调用将返回 /usr/bin 的第二个入口（在线程 A 的缓冲区中）。参考 4.1 节中的 `pipe.c`，该程序使用了 `readdir_r`。

顺序竞争也能在高水平的编码中被发现。例如，进程中的文件描述符被所有的线程共享。如果两个线程从同一个文件中试图读一个字符 (`getc`)，文件中的每个字符只能被一个线程读到。尽管 `getc` 自己是线程安全的，每个线程看见的字符顺序不是确定的——它取决于每个线程独立的 `getc` 调用顺序。他们可以轮流出现，在整个文件中各自得到第 2 个字符。或者，一个线程可以连续不断地得到一行中的 2 个或 100 个字符，而另一个线程则可能在被抢占（由于其他一些可能的原因）之前只得到 1 个字符。

有很多方法让你解决 `getc` 竞争。你可以在两个分开的文件描述符中打开文件并把它们分给每个线程。这样，每个线程就可以依次看见每个字符。这是通过去掉顺序依赖性来解决竞争。或者，你可以在每个线程中越过全部调用序列锁住文件，它通过强制所需的顺序来解决竞争。在 6.4.2 节中，程序 `putchar.c` 显示了一种类似的情况。

通常，不关心顺序的程序比强制特定顺序的程序运行地更高效。这是因为强制顺序将总是引入与工作不直接相关的计算开销。记住 Amdahl 法则：因为多线程编程的最大力量就是事情能并发地发生，“无序”的程序是更有效的，同步阻止了并发。如果大多数处理器在等待另一个处理器完成一些工作，则在一台多处理机上运行一个应用程序不会有很多帮助。

有时，“线程安全”函数还不够。例如，函数 `readdir` 通常将返回指向 `DIR` 结构（作为参数传给函数）的指针。这样，不同线程使用独立的 `DIR` 结构调用该函数可能总是线程安全的。你或许会认为 `readdir` 允许不同线程共享一个 `DIR` 结构——在 `DIR` 结构上增回一个互斥量，每个调用时锁住该互斥量。这种情况并不是如此简单，因为竞争的范围已经超出了 `readdir` 函数的界限。

线程 A 可能会锁住互斥量，调用 `readdir`，在目录中查找下一个文件“文件 A”。解锁互斥量，返回指向文件 A 的 `dirent` 结构的指针。线程 B 可能使用同一个 `DIR` 结构并发地调用 `readdir` 函数来查找下一个文件（它们可能是实现一个并行 `find` 命令）。第二个调用同样锁住互斥量，前进到下一个目录入口，解锁互斥量，返回文件 B。现在，线程 A 返回的指针指向的是文件 B。没有不合适的接口行为，也没有数据被破坏，但文件 A 却被遗忘了。为避免这种问题，调用者必须在调用 `readdir` 前锁住互斥量，并保持加锁状态直到返回的数据使用完毕或者被拷贝到其他地方。

这就是 Pthreads 引入 `readdir_r` 函数的原因。该函数多了一个参数，调用者

通过该参数设置一个 `dirent` 结构指针, `readdir_r` 函数将下一个目录入口返回到该参数指向的 `dirent` 结构中。尽管 `readdir_r` 函数可能不加锁互斥量——因为多线程对同一目录进行遍历并不常见, 但是你可以通过在每个 `readdir_r` 调用前加锁自己的互斥量来容易地实现共享遍历。改进的接口允许每个线程保留、使用返回的 `dirent` 结构数据, 而不需要在 `readdir_r` 调用返回后仍锁住互斥量。可以参考 4.2 节的 `crew.c` 程序, 该程序中使用了 `readdir_r` 函数。

8.1.3 合作避免僵局

与竞争一样, 死锁是一个程序中同步问题的结果。竞争是由于同步不够引起的资源冲突, 而死锁通常是有关同步使用的冲突。当任何两个线程共享资源时, 死锁可能发生。实质上, 如果线程 A 拥有资源 1, 但是直到它拥有资源 2 时才能继续执行, 而线程 B 拥有资源 2, 并且直到它拥有资源 1 时才能继续, 此时死锁发生。

在一个 Pthreads 程序中, 最常见的死锁类型是互斥量死锁, 即资源都是互斥量时。与竞争相比, 死锁的确实重要的一个优点是: 调试问题更容易。在竞争中, 线程不正确地做了一些事情并继续前进, 该问题在某段时间后才显示出来(通常作为一个附加影响)。但是发生死锁的线程将仍然在那里等待, 而且将总是在等待——如果它们能去任何地方, 它就不会是死锁了。因此当你使用调试器连接进程或者查看毁坏的存储区时, 可以看到涉及了哪些资源。通常进行小部分检查工作就能决定为什么发生死锁。

最可能的原因是对资源索取顺序的不一致。对于死锁的学习可以回归到设计操作系统时。任何学过计算机科学课的人可能都学过经典的进餐哲学家的问题。一些哲学家坐在一张圆桌旁吃意大利面条: 每个人交替地吃或讨论哲学。尽管讨论哲学不需要什么器具, 但每个哲学家需要两个叉子吃饭。桌子的摆设是在每对哲学家之间放一个叉子。哲学家需要同步他们的吃和讨论并阻止死锁。最显然的死锁形式是当所有的哲学家同时拣起一个叉子且拒绝放下它时。

总是有办法保证哲学家最后都能吃上东西。例如, 一个哲学家可以先拿右边的叉子, 然后看她的左边。如果叉子可得到, 她就能拿起它并且吃。如果不能得到左边的叉子, 她应该将手中的叉子放下(右边)然后继续聊一会儿天(那就是在 3.2.5.1 节讨论的互斥量回退策略)。因为哲学家心情都很好并且没有人最近出版了严重批评临近人的文章, 所以那些吃到东西的人都愿意很快地放下两个叉子以便他们的同事能继续吃。

更可靠(并且更干净)的方法是跳过细面条, 提供一盘能用一个叉子吃的菜。如果每个线程一次只有一个互斥量被锁住, 则互斥量死锁就不能发生。避免在锁住一个互斥量时调用函数是一个好主意。第一, 如果那个函数(或它调用的函数)锁住了其他互斥量, 你最后将遇到死锁; 第二, 锁住互斥量尽量短的时间是一个好主意(记住, 锁住一个互斥量将阻止其他线程并发地执行)。但是, 调用 `printf` 函数

数不太可能在代码中引起死锁，因为你不锁住任何 ANSI C 库中的互斥量，而且 ANSI C 函数库也不锁住你的任何互斥量。如果函数调用是进入自己的代码，或如果你调用一个可能调回到你的代码的函数库，你可要小心了。

如果你一次需要锁住超过一个互斥量，可以通过使用一个严格的锁层次或回退算法避免死锁。互斥量回退算法的主要缺点是：如果有大量其他线程锁住互斥量，即使它们没有任何死锁的可能，回退循环可能运行很长时间。回退算法假设其他线程在锁住一个或多个其他互斥量后，可能锁住第一个互斥量。如果所有的线程总是在回退循环中被锁的顺序锁住互斥量，那你就有了一个固定加锁层次，并且不需要回退算法。

当一个程序因为死锁挂起时，你需要线程调试器具有两个重要的能力。首先，应允许你在一个互斥量所有权被记录的模式下运行程序，并且可以使用调试器命令显示。如果发现一个线程拥有互斥量的同时在其他互斥量上阻塞，则这可能是你遇到死锁的指示。其次，你会希望检验拥有互斥量的线程的调用栈，以决定互斥量为什么仍被锁住。

但是，调用栈不总是足够的。死锁的一个常见原因是—一些线程没有解锁互斥量就从一个函数中返回。在这种情况下，你可能需要一个更复杂的工具来跟踪程序的同步行为，这样的一个工具将允许你检验数据并且判定哪个函数锁住了互斥量且没有解锁它。

8.1.4 小心优先级倒置

优先级倒置是依赖于实时优先级调度的应用（或库）所独有的一个问题。优先级倒置至少涉及三个具有不同优先级的线程。不同的优先级是重要的——优先级倒置是在同步与调度之间的冲突。优先级倒置允许一个低优先级线程无止境地阻止一个高优先级线程运行。结果通常不是死锁（尽管它可能是），但是它总是一个严重的问题。有关优先级倒置更多的信息可以参考 5.5.4 节。

最常见的优先级倒置就是三个不同优先级的线程共享资源。一个优先级倒置的例子是：一个低优先级线程锁住一互斥量，并且被一个高优先级线程抢占，然后在由低优先级线程锁住的互斥量上阻塞。通常，低优先级线程将恢复执行，允许它解锁互斥量，它将唤醒高优先级线程继续执行。然而，如果一个中等优先级的线程被唤醒（可能由于高优先级线程的一些动作），它可能阻止低优先级线程运行。中等优先级的线程（或它唤醒的其他线程）可以无止境地阻止低优先级线程释放互斥量，因此一个高优先级线程被一个低优先级线程的行动阻塞。

如果中等优先级线程阻塞，低优先级线程将被允许恢复执行并释放互斥量。由于这个原因，许多优先级倒置死锁在一个短时间后会自己解决。如果一个程序中所有的优先级倒置问题能可靠地在短时间内自己解决，优先级倒置就会成为一个性能问题而非正确性问题。在任何一种情况中，优先级倒置都可能是一个严重的问题。

以下是避免优先级倒置的一些想法：

- 完全避免实时调度。然而，这显然在许多实时应用中是不实际的。
- 设计你的线程使不同优先级的线程不需要使用同一个互斥量。这可能也是不切实际的：例如，许多ANSI C函数使用互斥量。
- 使用优先级 ceiling 互斥量（5.5.5.1节）或优先级继承（5.5.5.2节）。这些是Pthreads 的可选特性并且不是随处可得的。另外，你不能为不是你创建的互斥量设置优先级协议，包括那些由ANSI C函数使用的互斥量。
- 避免调用这样的函数：它可能锁住不是你创建的互斥量并提升互斥量的优先级。

8.1.5 绝不要在谓词之间共享条件变量

如果避免使用单个条件变量来管理多个谓词条件，你的代码通常将更干净和更有效。例如，你不应该定义单个的 `queue` 条件变量，既用它来唤醒等待队列为空的线程，又来唤醒其他等待向队列增加单元的线程。

但是这不只是一个性能问题（否则它将在其他节中讨论）。如果你使用 `pthread_cond_signal` 来唤醒在这些共享条件变量上等待的线程，程序可能因为某线程在条件变量上等待却没有其他线程唤醒它而挂起。

为什么？因为当你知道有单个线程需要被唤醒时，你只能在一个条件变量上发信号，在那个条件变量上等待的任何一个线程将被选择。当多个谓词共享一个条件变量时，你永远不能肯定唤醒的线程正在等待你设置的谓词。如果不是，则它将看见一个假的唤醒信号然后再次等待。你的信号就丢失了，因为等待你的谓词的线程没有机会看到它的变化。

当它得到一个假的唤醒信号时，为一个线程重新发条件变量信号也是不够的。线程可能不是以它们等待的顺序醒来，特别是当你使用优先级调度时。“重新发信号”可能在一些高优先级线程间导致一个无限循环（都使用错误的谓词），交替地唤醒对方。

当你确实想要在谓词之间共享一个条件变量时，最好的方法就是总要使用 `pthread_cond_broadcast`。但是当你广播信号时，所有的等待线程醒来重新评估它们的谓词。你总是知道那一个集合或其他集合不能继续执行——所以为什么要使它们都醒来呢？例如，如果 1 个线程正在等待写访问，而 100 个线程正在等待读访问，当广播意味着现在可以写了，所有的 101 个线程必须醒来，但是只有 1 个写线程能继续——其他 100 个线程必须再次等待。这种不精确的结果就是浪费了很多环境切换的时间，有更有用的途径让你的计算机一直忙碌。

8.1.6 共享堆栈和相应的内存破坏

在线程间共享堆栈内存没有什么不对的。即，线程在自己的堆栈中分配变量并将地址通知给其他线程是合法的，有时也是合理的。程序编写无误，则共享堆栈地址根本没有危险；然而，并不是每个程序都正确编写，即使你希望它正确时。共享

堆栈地址能使小的编程错误成为灾难，而且很难将这种错误隔离。

| 当其他线程可能仍在使用共享数据时，从共享堆栈分配函数中返回，将导致不可预知的后果。

如果共享堆栈内存，你必须保证拥有堆栈的线程决不会在其他线程停止使用共享数据之前将共享数据从堆栈中“弹出”。例如，当拥有共享堆栈的线程调用其他函数并因而重新分配共享变量的空间时，线程将从调用堆栈中返回。下列两个（或其中之一）可能的结果将被最终发现：

1. 由其他线程写入的数据将被保存的寄存器值、返回程序指针或其他什么东西所覆盖。导致共享数据被破坏。
2. 保存的寄存器值、返回程序指针或其他什么东西，将被其他修改共享数据的线程改写，导致拥有堆栈的线程的调用栈被破坏。

如果小心确保了拥有堆栈的线程不会在其他线程使用共享数据的时候弹出堆栈数据，你就安全了吗？可能还不会。我们只是触及一二，记住，我们是在讨论编程错误——例如，很可能是像没有对一个自动存储类型的指针初始化一样的愚蠢错误。指向共享数据的指针必须保存在某个有用的地方——其他线程没有别的办法找到共享堆栈地址。有时，指针很可能出现在每个使用共享数据的线程的堆栈内。当线程不再使用堆栈时，这些指针不必被删除。

无论有没有线程，使用未初始化指针写数据是一个常见的编程错误，所以从某种程度上说这不是什么新问题。然而，在线程和共享堆栈中，每个线程有机会破坏由其他线程异步使用的数据。数据破坏的征兆可能直到一段时间后才能显示出来，那将导致明显不同的调试任务。

在你的程序中，如果共享堆栈数据是方便的，则一定要利用这个功能。但是如果在调试过程中出现了未预料的事情，你需要特别仔细地检查共享堆栈数据的代码。如果一贯使用分析工具来报告未初始化的变量使用情况（如 Digital UNIX 上的 Third Degree），你可能不需要担心此类问题——或者很多其他问题。

8.2 避免性能问题

"Well, in our country," said Alice, still panting a little, "you'd generally get to somewhere else—if you ran very fast for a long time as we've been doing."

"A slow sort of country!" said the Queen. "Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"

—Lewis Carroll, Through the Looking-Glass

有时，一旦程序运行，工作就结束了。至少，在你希望让它做其他事情之前是

这样。但是，在很多情况下，“工作”并不就是足够好了。程序需要满足性能目标。有时性能目标是清晰的：“必须在这段时间内完成这么多事务”。有时目标又是模糊的：“必须很快”。

本节给出了一些观点帮助你判断程序有多快、什么使程序变慢，以及如何分辨你是否足够快。有一些很好的工具可以帮你，并且随着工业界转向支持热情的多线程程序员，将会有更多的辅助工具。但是对于多线程分析工具没有通用的标准。如果你的供应商支持多线程，你可能至少发现一个线程安全的 prof 版本（一个几乎通用的 UNIX 工具）。每个系统可能要求不同的参数和环境来线程安全地使用它，并且输出将有所不同。

性能调整要求的不仅是回答传统的问题：“程序在每个函数上花费多少时间？”例如，你必须分析互斥量竞争。具有高竞争可能性的互斥量可能需要被分为几个控制更特殊数据的互斥量（并发粒度优化），这将通过增加并发来改进程序性能。如果细粒度互斥量上竞争较少，则将它们合并将改进性能（通过减少加锁开销）。

8.2.1 了解并发串行化

理想的并行代码是一组完全计算密集型的任务。它们从不同步，从不阻塞——它们仅仅“思考”。如果一个程序开始时顺序调用三个计算密集型的函数，则当它创建三个线程分别运行每个函数时，程序的速度将是原来速度的（接近）三倍。至少当你在有三个空闲 CPU 的多处理器系统上运行程序时情况是这样。

理想的并发代码是一组完全 I/O 密集型任务。它们从不同步，很少计算——它们仅仅执行 I/O 请求并等待。如果你开始时写一个程序将多块数据写到三个独立文件中（理想情况是写到有独立控制器的三个硬盘上），则当你将程序改为创建三个线程，每个线程分别写部分数据时，所有三个 I/O 操作将能同时进行。

但是，如果你在写一组计算密集型或 I/O 密集型并行线程时遇到了上述问题，并显示你只是将串行程序改为了一个多线程的串行程序，结果又会怎么样呢？结果会是一个更慢的程序，它使用更多的开销来完成相同的结果。很可能这不是你期望的结果，如果那样你将怎么办呢？

假设你的计算密集型操作调用 malloc 和 free 函数，这些函数修改静态进程状态，所以需要某种程度的同步，很可能它们会锁住一个互斥量。如果你的线程循环调用 malloc 和 free 函数，导致它们的相当一部分时间花费在这些函数内，你可能发现几乎没有真正的并行。当其他线程分配或释放内存时，线程将花费很多时间阻塞在互斥量上。

类似的，并发 I/O 线程可能使用串行化资源。例如，如果线程使用相同的 stdio FILE 系统执行“并发”I/O，它们将锁住互斥量来更新文件流的共享缓冲区。即使线程使用独立的文件，如果它们在同一个磁盘上，它们将在文件系统中被锁住以同步文件缓冲区等。即使使用不同的磁盘，真实的并发也可能取决于 I/O 总线或磁盘控制子系统的限制。

所有这些情况的要点是写一个使用线程的程序并不能保证并行甚至并发。当你分析性能时，要了解你的程序可能受你无法控制的因素影响。你可能甚至无法看到文件系统中发生的事，但是无法看到的事情就可够伤害你。

8.2.2 使用正确数目的互斥量

使函数库成为线程安全的第一步可能是创建一个“大互斥量”来保护函数库的所有入口。如果一次只有一个线程能够执行库中的函数，则大部分函数将是线程安全的，至少没有静态数据被破坏。如果库没有包含需要在多个调用间保持一致的永久状态，“大互斥量”好像是足够了，很多函数库就是这种状态。标准 X11 客户端库（Xlib）为这种“大互斥量”提供有限的支持已经很多年了。

现在你希望函数库在线程内执行得很好，所以线程安全不再是足够的了。在大多数情况下，那将需要重新设计库以便多个线程能同时使用它。大互斥量串行化库中的所有操作，因此你在库中不会得到任何并发或并行。如果使用那个函数库是线程的主要功能，则使用单个线程（没有同步）将使程序跑得更快。记住，Xlib 中的大互斥量将阻止所有其他线程使用任何 Xlib 函数直到第一个线程从服务器收到应答，这可能会花费相当一段时间。

规划你的函数库功能，决定哪个操作可以合理地并行执行。通常的策略是为每个数据结构创建一个独立的互斥量，并且使用那些互斥量来串行化对共享数据的访问，而非使用“大互斥量”来串行化对函数库的访问。

使用一个支持线程的分析器，通过查找对 `pthread_mutex_lock`、`pthread_mutex_unlock` 和 `pthread_mutex_trylock` 的调用热点，你可以判定有太多的互斥量活动。然而，这个数据不具备总结性，而且可能很难判定这种高活动率是由于太多的互斥量竞争还是由于没有竞争的锁太多。你需要关于互斥量竞争更特定的信息，而那通常要求特殊的工具。一些线程开发系统提供了详细的显示同步开销的可视化跟踪信息。其他系统则提供了单个互斥量上的测量信息，告诉你该互斥量被锁住了几次，以及多长时间线程会发现互斥量已经被锁住。

8.2.2.1 太多互斥量不会有帮助

同样需要注意的是，用很多“小”互斥量交换一个“大”互斥量，你可能适得其反。记住，加锁互斥量要花时间，而且解开互斥量需要更多时间。就算你通过设计一个很少竞争的锁层次来增加并行，你的线程可能花费如此多的时间来加锁、开锁所有这些互斥量，而并没有做多少真正的工作。

加锁互斥量也影响内存子系统。除了加锁和开锁的时间开销，你可能因为额外的加锁操作而降低内存系统的效率。例如，锁住一个互斥量可能使所有处理器上的一块缓存无效。它可能停止某段物理地址范围上的所有总线活动。

因此要找出在哪儿你确实需要互斥量。例如，在前面一节中我建议为每个数据结构创建一个独立的互斥量。然而，如果两个数据结构通常被一起使用，或者如果

当其他线程正在使用第二个数据结构时，线程将很少需要使用第一个数据结构，则额外的互斥量可能降低程序的整体性能。

8.2.3 绝不要与缓存作对

现代计算机不会直接从主存读取数据。能够与计算机一样快的内存过于昂贵，使得（直接从主存读数据）不实际。相反，数据被内存管理单位取到一个很快的本地缓存数组中。当计算机写数据时，也是写到本地的缓存数组中。修改的数据可能很快地被写回主存，或者仅当需要时才被“刷新”回主存。

因此，如果在一个多处理机系统中，一个处理器需要读取其他处理器缓存区中的值时，必须有一些“缓存一致性”机制确保它能够发现正确的数据。更重要地，当一个处理器向某个位置写数据时，在缓存中拥有旧拷贝的所有其他处理器需要拷贝新数据，或记录旧数据无效。

计算机系统通常以比较大的 64 字节或 128 字节缓冲数据，这将通过优化到慢速主存的索引来改进效率。它也意味着，当同样的 64 字节或 128 字节块被多个处理器缓冲时，一个处理器写数据到块中的任何部分，缓冲该块的所有处理器必须扔掉整个块。

这对于高性能并行计算有严肃的含意。如果两个线程在同样的缓存块中存取不同的数据，没有线程可以利用它正在使用的处理器上的缓冲拷贝。每个线程需要从主存中添满一个新的缓存，这将降低程序的速度。

缓存行为可能变化很大，甚至在使用同样微处理器芯片的不同计算机系统上也是如此。不可能写出在所有系统上都最优的代码。然而，通过很小心地排列和分离由多个线程使用的性能-临界数据，你可以增加这种机会。

你可以通过判断系统的缓存特征来为一个特定的计算机系统优化代码（设计你的代码，使得任两个线程都不会在性能-临界的并行循环内向相同的缓存块写数据）。如果不是为某个特定的系统进行优化，你能够希望做到最好的事情就是确保每个线程有一个私有的、页边界的的数据段。不可能有哪个系统会使用像页面那样大的缓存块，因为一页包含了太多的各类数据来提供内存管理单位中的性能优势。

POSIX 多线程快速参考

本章是 POSIX.1c 标准的一个压缩版的参考手册。

9.1 POSIX 1003.1c-1995 选项

Pthreads 适用于多个领域。在高性能的计算程序中可以用它来支持多个循环的并行分解，实时应用程序可以用它支持并发的实时输入 / 输出，在数据库和网络服务程序中可以轻而易举地用它来支持客户的并发访问，商业或者软件开发程序在分时系统中可以使用其并行和并发的优点。

通过定义一套特征检测宏（见表 9.1），Pthreads 标准允许用户决定在系统中应用哪种可选功能。任何 Pthreads 的实现都会通过如下三种形式告诉用户支持哪种选项：

- 在 POSIX 文献中做出支持选项的正式声明，用户就可以使用这些帮助信息设计自己的特定应用。
- 在头文件<unistd.h>中定义编译时的符号常量，用户可以在很多 Pthread 系统中使用 #ifdef 或者 #ifndef 预处理条件语句，测试这些符号常量。
- 当使用 sysconf 符号调用 sysconf 函数时，返回正的非零值（对于特征检测宏通常无效，它们指定选项当前是否存在，如果不存在，就不支持相关的接口，你的代码就不会被链接，甚至不会被编译）。

例如，你会发现大部分的系统都不支持优先级调度，你或许选择不使用该选项。或者在提供该特性的系统上，你宁愿为互斥量使用优先级继承协议，你的代码应该避免在不支持该选项的系统上访问互斥性协议属性。

表 9.1 POSIX 1003.1c-1995 的选项

符号常量，sysconf 符号名称	描述
_POSIX_THREADS _SC_THREADS	支持线程（如果你的系统不支持线程，你就没有那么好的运气了）
_POSIX_THREAD_ATTR_STACKSIZE _SC_THREAD_ATTR_STACKSIZE	控制线程堆栈的大小

续表

符号常量, sysconf 符号名称	描述
_POSIX_THREAD_ATTR_STACKADDR _SC_THREAD_ATTR_STACKADDR	分配和控制线程堆栈
_POSIX_THREAD_PRIORITY_SCHEDULING _SC_THREAD_PRIORITY_SCHEDULING	支持实时调度
_POSIX_THREAD_PRIO_INHERIT _SC_THREAD_PRIO_INHERIT	支持优先级继承互斥量
_POSIX_THREAD_PRIO_PROTECT _SC_THREAD_PRIO_PROTECT	支持优先级 ceiling 互斥量
_POSIX_THREAD_PROCESS_SHARED _SC_THREAD_PROCESS_SHARED	支持进程间共享的互斥量和条件变量
_POSIX_THREAD_SAFE_FUNCTIONS _SC_THREAD_SAFE_FUNCTIONS	支持线程安全的特殊-r 库函数

9.2 POSIX 1003.1c-1995 限制

通过定义一套如表 9.2 所示的宏, Pthreads 多线程标准允许你决定可能影响应用的运行时系统限制, 例如, 你可以创建多少个线程。任何 Pthreads 多线程的实现都必须以下列三种方式告知它的限制:

- 在 POSIX 文献中做出正式声明。用户可以使用这些帮助信息设计自己的特定应用。
- 在头文件<limits.h>中定义编译时的符号常量。当限制满足最小需求时, 符号常量在头文件<limits.h>中可被忽略, 但是在编译时无法确定它的限制, 例如, 它可能依赖空闲的内存空间。用户可以使用#ifndef 或者#ifndef 预处理条件语句, 测试这些符号常量。
- 当使用 sysconf 符号调用 sysconf 函数时, 返回正的非零值。

文献中的系统大部分不支持超过 64 个线程, 你可以设计不超过 64 个线程的应用。或者如果超过 64 个线程, 可以依靠 PTHREAD_THREADS_MAX 符号常量, 或者在运行的时候调用 sysconf 函数来决定限制。

表 9.2 POSIX 1003.1c-1995 限制

运行时不变量, sysconf 符号名称	描述
PTHREAD_DESTRUCTOR_ITERATIONS _SC_THREAD_DESTRUCTOR_ITERATIONS	在终止时试图破坏一个线程的线程特定数据的最大值 (至少为 4)
PTHREAD_KEYS_MAX _SC_THREAD_KEYS_MAX	现存每个进程中线程特定的数据键的最大值 (至少为 128)

续表

运行时不变量, sysconf 符号名称	描 述
PTHREAD_STACK_MIN _SC_THREAD_STACK_MIN	支持的最小堆栈的大小
PTHREAD_THREADS_MAX _SC_THREAD_THREADS_MAX	支持的最大线程数量 (至少为 64)

9.3 POSIX 1003.1c-1995 接口

这些接口是按照函数类型进行排序的：线程、互斥量等。在每种类型里，这些接口都按照字母顺序排列，图 9.1 描述了接口的格式。

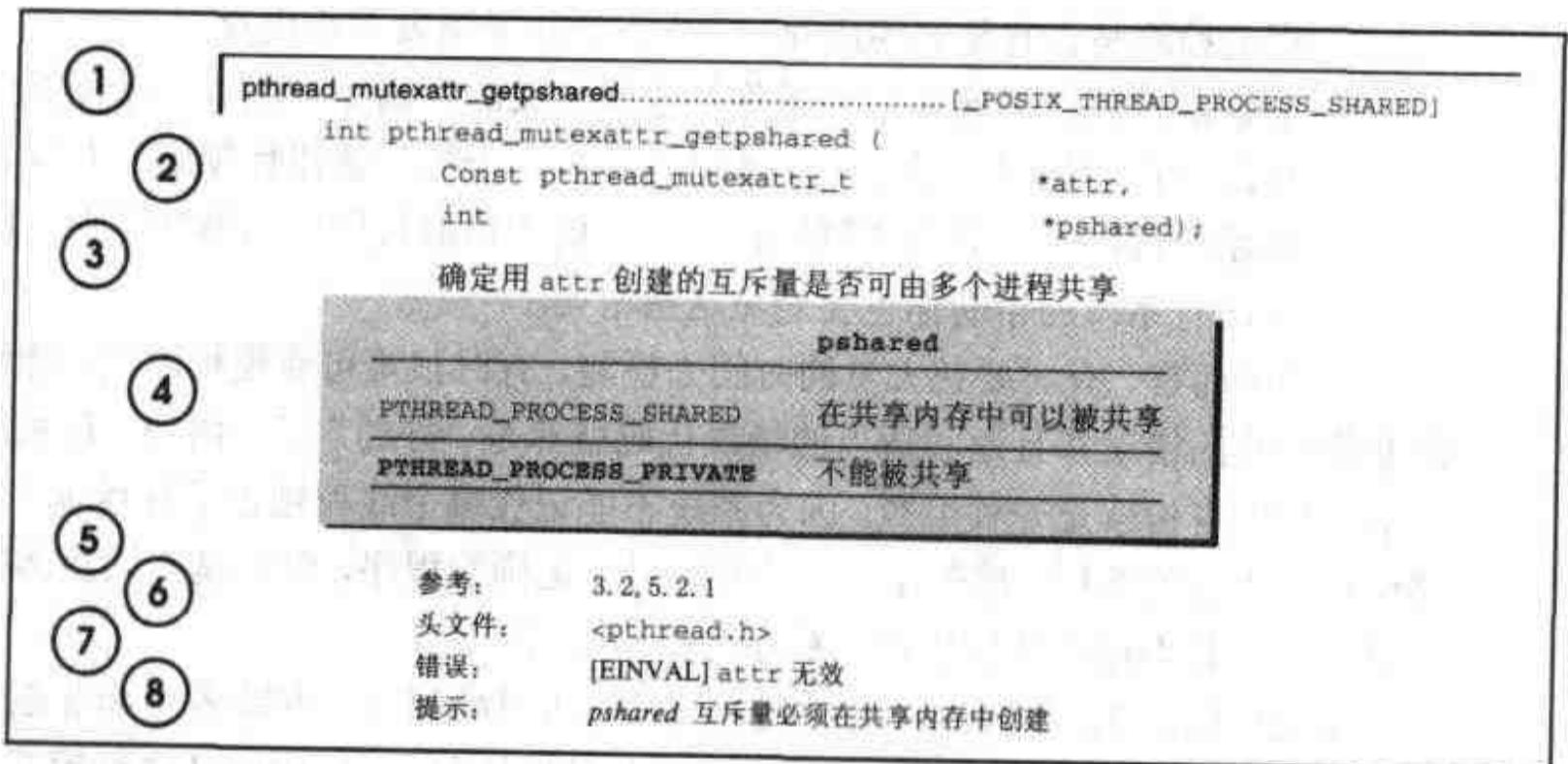


图 9.1 快速参考的格式

(1) 头部：显示了这个接口的名称，如果接口是 Pthreads 的可选部件，那么在该行的最后会显示特征检测宏的名称选项。例如，接口 `pthread_mutexattr_getpshared` 是 `_POSIX_THREAD_PROCESS_SHARED` 特征下的选项。

(2) 原型：显示了这个接口的全部 C 语言原型，描述了怎样用各种参数调用函数。

(3) 描述：给出了大纲摘要。例如，图中所示接口的目的是“确定用属性对象创建的互斥量能否在多进程间共享”。

(4) 表中描述了所有符号型参数可能的取值，如图中的 `pshared` 参数。参数的默认取值(新线程的状态或者新属性对象中某个属性的默认值)用黑体标识出。

(5) 参考：给出了本书论述该接口或相关接口的主要参考章节。

(6) 头文件：说明了用这种功能编码的头文件，你需要所有列在此处的头文

件。

(7) 错误：描述从接口返回的错误代号，因为 Pthreads 可以区分强制性错误检测和非强制性的错误检测，这些错误都会以黑体报告出来（具体细节见 9.3.1）。

(8) 提示：对该接口做个别的必需的简单评论。某些提示指出了使用接口常犯的错误；某些提示描述了设计者预期的接口用途，某些提示则指出了接口的基本限制；如在 `single pthread_mutexattr_getpshared` 函数中提示指出的那样。

9.3.1 错误检测和报告

POSIX 标准可以仔细地辨别出以下两种类型的错误。

1. 强制性错误（包括超出程序员控制范围的情况），这些错误必须总是由系统检测出并以特别的错误代码报告出来。如果由于进程缺乏足够的虚拟内存使你不能创建新的线程，那么系统一定要告知你。不可能要求在创建线程前检测是否有足够的内存——你无法知道需要多少内存。
2. 非强制性的错误通常是由于你的错误造成的。例如，你或许试图锁定一个还没有初始化的互斥量，或者试图释放一个被其他线程锁定的互斥量。某些系统可能不检测这些错误，但是它们在你的代码中仍然是错误，因此即使没有系统的帮助你也要避免这些错误。

就系统而言，有时要花大量的时间去检测，有时很难可靠地检测到；能够检测出非强制性的错误并且返回适当的错误代码就算是“好的”了。例如，让系统确定当前线程的身份就是一种浪费。因为系统不能记住哪个线程锁定了互斥量，也不能够保护未锁定的线程，这就是一个错误。对于正确的程序，降低基本同步运算是没有意义的，只不过在调试错误的程序时稍微容易一点。

系统可以提供调试模式，在这种模式中，可以检测到一些或者全部非强制性的错误，例如 Digital UNIX 提供了 `error check` 互斥量和一个 `metered` 执行模式，跟踪互斥量的所有者，报告出相关的加锁与解锁互斥量造成的非强制性的错误。UNIX98 规范包含了 `error check` 互斥量，见 10.1.2 节，它们不久就会在大多数 UNIX 系统中使用。

9.3.2 使用 `void*` 类型

ANSI C 允许任何指针类型转化成 `void*` 类型，转化后，原值不变。但是，ANSI C 不允许所有的指针类型有相同的二进制表示法。例如，为了将其传递给线程起始例程，转化为 `void*` 的 `long*` 型必须当作 `long*` 型而不能作为 `char*` 型使用。另外，在 `pointer` 和 `integer` 之间转换的结果是通过“实现定义”得到的。大多数支持 UNIX 的系统允许将数值型置换成 `void*` 类型，或者允许将数值型置换成混合 `pointer` 类型——但是要注意，这样的代码并不适用于所有的系统。

某些其他的标准，如 POSIX.1b 实时系统标准，已经采用不同的方法解决了这

个问题（变量或者结构中的成员可以采用任何类型的值）。譬如 POSIX.1b 中的 `sigevent` 结构，它的一个成员 `sigev_value` 可以将值传进 `signal`——`catching` 函数。但是 `sigev_value` 没有定义为 `void*` 类型，而是由程序员指定适当的类型，`sigev_value` 成员是 `union sigval` 类型，该成员包含 `int` 和 `void*` 类型的成员。这种机制解决了 `integer` 和 `pointer` 之见置换的困难，排除了与 ANSI C 的冲突。

9.3.3 线程

线程提供一种并发能力，可以在一个进程中的同一时刻运行多个流。每个线程都有自己的硬件寄存器和堆栈。一个进程中的所有线程共享全部虚拟空间地址、所有文件描述、信号行为和其他的进程资源。

`pthread_attr_destroy`

```
int pthread_attr_destroy (
    pthread_attr_t *attr);
```

释放线程属性的对象，该对象不能再被使用。

参考: 2, 5.2.3

头文件: <pthread.h>

错误: [EINVAL] attr 无效。

提示: 不影响用 attr 创建线程。

`pthread_attr_getdetachstate`

```
int pthread_attr_getdetachstate (
    const pthread_attr_t *attr,
    int *detachstate);
```

判定用 attr 创建的线程是否是可分离的。

detachstate	
<code>PTHREAD_CREATE_JOINABLE</code>	线程 ID 有效，可以连接
<code>PTHREAD_CREATE_DETACHED</code>	线程 ID 无效，不能连接、取消或修改

参考: 2, 5.2.3

头文件: <pthread.h>

错误: [EINVAL] attr 无效。

提示: 不能联结或者取消可分离的线程。

pthread_attr_getstackaddr.....[_POSIX_THREAD_ATTR_STACKADDR]
 int pthread_attr_getstackaddr (
 const pthread_attr_t *attr,
 void **stackaddr);

判定用 attr 创建的线程运行的堆栈地址。

参考: 2, 5.2.3

头文件: <pthread.h>

错误: [EINVAL] attr 无效。

[ENOSYS] stackaddr 不支持。

提示: 每个堆栈地址仅能创建一个线程!

pthread_attr_getstacksize.....[_POSIX_THREAD_ATTR_STACKSIZE]
 int pthread_attr_getstacksize (
 const pthread_attr_t *attr,
 size_t *stacksize);

判定用 attr 创建的线程运行的堆栈大小。

参考: 2, 5.2.3

头文件: <pthread.h>

错误: [EINVAL] attr 无效。

[ENOSYS] stacksize 不支持。

提示: 用最新的属性对象查找默认的堆栈大小。

pthread_attr_init

int pthread_attr_init (
 pthread_attr_t *attr);

用默认属性初始化线程属性对象。

参考: 2, 5.2.3

头文件: <pthread.h>

错误: [ENOMEM] 对 attr 来说内存不足。

提示: 用来定义线程类型。

pthread_attr_setdetachstate

int pthread_attr_setdetachstate (
 pthread_attr_t *attr,
 int detachstate);

指定用 attr 创建的线程是否是可分离的。

detachstate	
PTHREAD_CREATE_JOINABLE	线程ID有效，可以连接
PTHREAD_CREATE_DETACHED	线程ID无效，不能连接、取消或者修改

参考: 2, 5.2.3

头文件: <pthread.h>

错误: [EINVAL] attr 无效。

[EINVAL] detachstate 无效。

提示: 不能连接或者取消可分离的线程。

pthread_attr_setstackaddr.....[_POSIX_THREAD_ATTR_STACKADDR]

```
int pthread_attr_setstackaddr (
    pthread_attr_t *attr,
    void           *stackaddr);
```

由 attr 创建的线程在以 stackaddr 起始的堆栈上运行时，该堆栈必须至少有 PTHREAD_STACK_MIN 个字节。

参考: 2, 5.2.3

头文件: <pthread.h>

错误: [EINVAL] attr 无效。

[ENOSYS] stackaddr 不支持。

提示: 每个堆栈地址只能创建一个线程，同时要注意堆栈队列。

pthread_attr_setstacksize.....[_POSIX_THREAD_ATTR_STACKSIZE]

```
int pthread_attr_setstacksize (
    pthread_attr_t *attr,
    size_t          stacksize);
```

由 attr 创建的线程在至少有 stacksize 个字节大小的堆栈上运行时，该堆栈必须至少有 PTHREAD_STACK_MIN 个字节。

参考: 2, 5.2.3

头文件: <pthread.h>

错误: [EINVAL] attr 或者 stacksize 无效。

[EINVAL] stacksize 太小或者太大。

[ENOSYS] stacksize 不支持。

提示：找到第一个默认值（`pthread_attr_getstacksize`），然后以乘法增加。如果线程所需超过默认值，选用合适的。

`pthread_create`

```
int pthread_create (
    pthread_t          *tid,
    const pthread_attr_t *attr,
    void             *(*start) (void *),
    void            *arg);
```

用来创建一个运行 `start` 函数的线程，即采用变量值 `arg` 异步调用该函数。变量 `attr` 设定了可选的创建属性，同时返回新的线程标识 `tid`。

参考：2, 5.2.3

头文件：<pthread.h>

错误：**[EINVAL]** attr 无效。

[EAGAIN] 资源紧张。

提示：线程所需资源必须已经初始化。

`pthread_detach`

```
int pthread_detach (
    pthread_t          thread);
```

分离线程。用来分离主线程，或者在不感兴趣的线程里创建了一个可连接的线程后，“改变了主意”。

参考：2, 5.2.3

头文件：<pthread.h>

错误：**[EINVAL]** 线程不是 joinable 可连接线程。

[ESRCH] 没有找到作为ID的线程。

提示：可分离的线程不能被连接或取消；终止时立刻释放存储区。

`pthread_equal`

```
int pthread_equal (
    pthread_t          t1,
    pthread_t          t2);
```

`t1` 等于 `t2` 时，返回非零值，否则返回零值。

参考：2, 5.2.3

头文件：<pthread.h>

提示： 比较 `pthread_self` 与储存的线程标识。

`pthread_exit`

```
int pthread_exit (  
    void *value_ptr);
```

用来终止主调线程，返回 `value_ptr` 值给任何连接的线程。

参考： 2, 5.2.3

头文件： <pthread.h>

提示： `value_ptr` 作为一个值对待，而不是值的地址。

`pthread_join`

```
int pthread_join (  
    pthread_t thread,  
    void **value_ptr);
```

用来等候线程终止，如果 `value_ptr` 非 Null，则返回线程退出值。也可以在成功编译时分离线程。

参考： 2, 5.2.3

头文件： <pthread.h>

错误： **[EINVAL]** 线程不是可连接的线程。

[ESRCH] 找不到ID线程。

[EDEADLK] 试图自己连接。

提示： 可分离的线程不能连接或取消。

`pthread_self`

```
pthread_t pthread_self (void);
```

返回主调线程的 ID。

参考： 2, 5.2.3

头文件： <pthread.h>

提示： 设置线程的调度参数。

`sched_yield`

```
int sched_yield (void);
```

在其他同等优先级的线程就绪之后，主调线程就绪，并且运行一个新的线程。允许同等优先级的协作线程更合理地共享处理器资源，特别是单处理器上的资源。本函数来源于 POSIX.1b（实时扩展），在头文件<sched.h>中声明。报错返回值为-1，同时在 `errno` 里存储一个错误代码。

参考: 2.5.2.3

头文件: <sched.h>

错误: [ENOSYS] sched_yield 不支持。

提示: 在给互斥量加锁前使用它, 从而在锁定互斥量时, 减少一次时间片的机会。

9.3.4 互斥量

互斥量提供了同步能力, 可以控制线程如何共享资源。使用互斥量可以避免在同一时刻多个线程修改共享的数据, 并且保证一个线程能够从一套资源中读到一致性数据, 其他线程可以修改这些资源(如内存)。

pthread_mutexattr_destroy

```
int pthread_mutexattr_destroy (
    pthread_mutexattr_t *attr);
```

释放互斥量属性对象, 该对象再也不能被使用。

参考: 3.2, 5.2.1

头文件: <pthread.h>

错误: [EINVAL] attr 无效。

提示: 不影响用 attr 创建互斥量。

pthread_mutexattr_getpshared [__POSIX_THREAD_PROCESS_SHARED]

```
int pthread_mutexattr_getpshared (
    const pthread_mutexattr_t *attr,
    int *Pshared);
```

判定用 attr 创建的互斥量能否被多进程共享。

pshared	
PTHREAD_PROCESS_SHARED	如果在共享内存内, 可以被共享
PTHREAD_PROCESS_PRIVATE	不能被共享

参考: 3.2, 5.2.1

头文件: <pthread.h>

错误: [EINVAL] attr 无效。

提示: 必须在共享内存内分配互斥量。

pthread_mutexattr_init

```
int pthread_mutexattr_init (
    pthread_mutexattr_t *attr);
```

用默认属性初始化互斥量属性对象。

参考: 3.2.5.2.1

头文件: <pthread.h>

错误: [ENOMEM] 对 attr 来说内存不足。

提示: 常用来定义互斥量类型。

pthread_mutexattr_setpshared.....[_POSIX_THREAD_PROCESS_SHARED]

```
int pthread_mutexattr_setpshared (
    pthread_mutexattr_t *attr,
    int pshared);
```

如果用进程共享的内存分配 pthread_mutex_t 变量，则进程间可以共享用 attr 创建的互斥量。

Pshared	
PTHREAD_PROCESS_SHARED	如果在共享内存内，可以被共享
PTHREAD_PROCESS_PRIVATE	不能被共享

参考: 3.2.5.2.1

头文件: <pthread.h>

错误: [EINVAL] attr 或者 detachstate 无效。

提示: 必须在共享内存内分配互斥量。

pthread_mutex_destroy

```
int pthread_mutex_destroy (
    pthread_mutex_t *mutex);
```

释放不用的 mutex。

参考: 3.2.5.2.1

头文件: <pthread.h>

错误: [EBUSY] 互斥量正在使用。

[EINVAL] 互斥量无效。

提示: 当没有其他的线程要上锁时，对互斥量解锁是最安全的。

pthread_mutex_init

```
int pthread_mutex_init (
    pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);
```

初始化互斥量。attr 参数设定可选的创建属性。

参考: 3.2.5.2.1

头文件: <pthread.h>

错误: [EAGAIN] 资源(除内存外的其他资源)不足。

[ENOMEM] 内存不足。

[EPERM] 无权执行此项操作。

[EBUSY] 互斥量已经初始化。

[EINVAL] attr无效。

提示: 如果可能, 采用静态的初始值代替。

pthread_mutex_lock

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

给互斥量加锁。如果当前该互斥量已经加锁, 主调线程就会阻塞, 直到互斥量解锁为止。作为回报, 线程就会占有这个互斥量, 直到调用 `pthread_mutex_unlock` 函数, 才会释放该互斥量。

参考: 3.2.5.2.1

头文件: <pthread.h>

错误: [EINVAL] 线程优先级超过互斥量优先级的最大限度。

[EINVAL] 互斥量无效。

[EDEADLK] 主调线程已经占有了互斥量。

提示: 在同一个线程内锁总是开的。

pthread_mutex_trylock

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

给互斥量加锁。如果当前互斥量已经上锁, 立刻返回 EBUSY, 否则, 主调线程成为该互斥量的拥有者, 并且一直到它解锁为止。

参考: 3.2.5.2.1

头文件: <pthread.h>

错误: [EINVAL] 线程优先级超过互斥量优先级的最大限度。

[EBUSY] 互斥量已经被锁定。

[EINVAL] 互斥量无效。

[EDEADLK] 主调线程已经占有这个 mutex。

提示: 在同一个线程里锁总是开的。

pthread_mutex_unlock

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

给互斥量解锁。该互斥量成为自由的。如果有些线程正在等候这个互斥量，其中的一个线程就会被唤醒（按照优先级顺序选择 SCHED_FIFO 调度策略和 SCHED_RR 策略的等待者，其他的则按照不定顺序选择。）。

参考: 3.2.5.2.1

头文件: <pthread.h>

错误: [EINVAL] 互斥量无效。

[EPERM] 主调线程已经占有该mutex。

提示: 在同一个线程里锁总是开的。

9.3.5 条件变量

条件变量提供通信功能，这种功能就是等待某些共享资源到达某个期望的状态，或者是在这些资源到达某种其他线程感兴趣的状态时，发出信号。条件变量与保护这种资源状态的互斥量紧密相关。

`pthread_condattr_destroy`

```
int pthread_condattr_destroy (
    pthread_condattr_t      *attr);
```

释放条件变量属性对象，不能再使用该对象。

参考: 3.3.5.2.2

头文件: <pthread.h>

错误: [EINVAL] attr 无效。

提示: 不影响用 attr 创建条件变量。

`pthread_condattr_getpshared`.....[_POSIX_THREAD_PROCESS_SHARED]

```
int pthread_condattr_getpshared (
    const pthread_condattr_t   *attr,
    int                      *pshared);
```

判定多进程是否能够共享用 attr 创建的条件变量。

Pshared	
PTHREAD_PROCESS_SHARED	在共享内存中可以被共享
PTHREAD_PROCESS_PRIVATE	不能被共享

参考: 3.3.5.2.2

头文件: <pthread.h>

错误: [EINVAL] attr 无效。

提示: 必须在共享内存里分配 pshared 条件变量，同时该变量必须与 pshared 的互斥量一起使用。

pthread_condattr_init

```
int pthread_condattr_init (
    pthread_condattr_t *attr);
```

用默认的属性值初始化条件变量属性对象。

参考: 3.3.5.2.2

头文件: <pthread.h>

错误: [ENOMEM] 对 attr 来说内存不足。

提示: 用于定义条件变量类型。

pthread_condattr_setpshared.....[_POSIX_THREAD_PROCESS_SHARED]

```
int pthread_condattr_setpshared (
    pthread_condattr_t *attr,
    int pshared);
```

如果在进程共享的内存里分配 pthread_cond_t 变量，那么在进程间就可以共享用 attr 创建的条件变量。

pshared	
PTHREAD_PROCESS_SHARED	在共享内存中可以被共享
PTHREAD_PROCESS_PRIVATE	不能被共享

参考: 3.3.5.2.2

头文件: <pthread.h>

错误: [EINVAL] attr 或者 pshared 无效。

提示: 必须在共享内存里分配 pshared 条件变量，同时该变量必须与 pshared 的互斥量一起使用。

pthread_cond_destroy

```
int pthread_cond_destroy (
    pthread_cond_t *cond);
```

释放不需要的条件变量 cond。

参考: 3.3.5.2.2

头文件: <pthread.h>

错误: [EBUSY] cond 正在使用。

[EINVAL] cond 无效。

提示: 当没有其他的线程等候时, 从 cond 中唤醒是最安全的。

pthread_cond_init

```
int pthread_cond_init (
    pthread_cond_t          *cond,
    const pthread_condattr_t *attr);
```

初始化条件变量 cond。attr 参数设定可选的创建属性。

参考: 3.3.5.2.2

头部: <pthread.h>

错误: [ENOMEM] 资源不足。

[EINVAL] 内存不足。

[EBUSY] cond 已经初始化。

[EINVAL] attr 无效。

提示: 如果可能, 用静态的初始值代替。

pthread_cond_broadcast

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

广播条件变量 cond, 唤醒当前所有的等待者。

参考: 3.3.5.2.2

头部: <pthread.h>

错误: [EINVAL] cond 无效。

提示: 当许多等待者响应属性改变时或者任何一个等待的线程都不能响应时使用。

pthread_cond_signal

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

用信号通知条件变量 cond, 唤醒一个等待的线程。如果有些 SCHED_FIFO 或者 SCHED_RR 策略线程正在等待, 那么唤醒最高优先级的等待者, 否则唤醒不确定的等待者。

参考: 3.3.5.2.2

头部: <pthread.h>

错误: [EINVAL] cond 无效。

提示: 在一些等待者有能力响应, 但是其中只有一个需要响应的时候使用 (所有等待者优先级相同)。

`pthread_cond_timedwait`

```
int pthread_cond_timedwait (
    pthread_cond_t           *cond,
    pthread_mutex_t          *mutex,
    const struct timespec     *abstime);
```

用条件变量 cond 等待, 直到由一个信号或者广播, 或者绝对时间 abstime 到了, 才唤醒该线程。

参考: 3.3.5.2.2

头文件: <pthread.h>

错误: [ETIMEDOUT] 由 abstime 指定的时刻已经超过了。

[EINVAL] cond、mutex 或者 abstime 无效。

[EINVAL] 同时等待不同的互斥量。

[EINVAL] 互斥量没有被主调线程占有。

提示: 即使等待失败、超时或者被取消, 互斥量也总是在等待之前不加锁, 等待之后在 `pthread_cond_timedwait` 内部再加锁。

`pthread_cond_wait`

```
int pthread_cond_wait (
    pthread_cond_t           *cond,
    pthread_mutex_t          *mutex);
```

用条件变量 cond 等待, 直到由一个信号或者广播唤醒时为止。

参考: 3.3.5.2.2

头文件: <pthread.h>

错误: [EINVAL] cond 或者 mutex 无效。

[EINVAL] 同时等待不同的互斥量。

[EINVAL] 主调线程没有占有互斥量。

提示: 即使等待失败、超时或者被取消, 互斥量也总是在等待之前不加锁, 等待之后在 `pthread_cond_timedwait` 内部再加锁。

9.3.6 取消

取消提供了一种方法, 即当你不再需要一个线程完成正常的执行过程时, 温和地终止该线程。每个线程都能够控制“取消”如何影响自己, 或者是否影响自己,

并且因为取消而终止线程时，共享状态可以获得修复。

pthread_cancel

```
int pthread_cancel (            
                      pthread_t      thread);
```

请求取消 *thread* 线程。

参考: 5.3

头文件: <pthread.h>

错误: [ESRCH] 没有线程响应应该 *thread*。

提示: “取消”是异步的。如果需要，用 *pthread_join* 等候线程终止。

pthread_cleanup_pop

```
void pthread_cleanup_pop (int execute);
```

弹出大多数最近入栈的 cleanup handler (清除处理程序)。如果 *execute* 非零，就调用 cleanup handler。

参考: 5.3

头文件: <pthread.h>

提示: 设定 *execute* 非零，以免复制共同的 cleanup 代码。

pthread_cleanup_push

```
void pthread_cleanup_push (            
                           void          (*routine)(void *),  
                           void          *arg);
```

将 cleanup handler 推入线程的 cleanup handler 堆栈。当线程通过调用 *pthread_exit* 退出时，或者当线程作为取消请求时，或者当线程用非零 *execute* 参数调用 *pthread_cleanup_pop* 时，入栈的 cleanup handler 弹出，线程用 *arg* 参数调用它们。

参考: 5.3

头文件: <pthread.h>

提示: *pthread_cleanup_push* 和 *pthread_cleanup_pop* 在语法范围内必须成对。

pthread_setcancelstate

```
int pthread_setcancelstate (            
                           int           state,  
                           int           *oldstate);
```

原子地设置 state 为主调线程的取消状态， oldstate 返回以前的状态。

state,oldstate	
PTHREAD_CANCEL_ENABLE	取消被启用
PTHREAD_CANCEL_DISABLE	取消被禁用

参考： 5.3

头文件： <pthread.h>

错误： [EINVAL] state 无效。

提示： 用于在包含取消点的“原子”代码周围禁用取消。

pthread_setcanceltype

```
int pthread_setcanceltype (
    int             type,
    int             *oldtype);
```

原子地设置 type 为主调线程的取消类型，在已引用的区域通过 oldtype 返回以前的类型。

type,oldtype	
PTHREAD_CANCEL_DEFERRED	仅有延期的取消被允许
PTHREAD_CANCEL_ASYNCHRONOUS	允许异步取消

参考： 5.3

头文件： <pthread.h>

错误： [EINVAL] 类型无效。

提示： 注意：用于异步取消类型的大多数代码不安全。

pthread_testcancel

```
void pthread_testcancel (void);
```

用于在主调线程内创建一个延期的取消点。如果当前取消状态是 PTHREAD_CANCEL_DISABLE，调用该函数不受影响。

参考： 5.3

头文件： <pthread.h>

提示： 在计算密集型循环中用于轮循取消请求。

9.3.7 线程私有数据

线程私有数据提供了一种方法来声明在所有的线程里有相同“名称”（但是对于每个线程，值不同）的变量。在大多数方面，多线程程序里可以考虑使用线程私有数据，非线程化程序里可以考虑使用静态数据。例如，对某些函数，静态数据继续上下交叉的一系列调用时，环境关系通常就是线程的特定性关系（如果不是，互斥量就必须保护静态数据）。

pthread_getspecific

```
void *pthread_getspecific (
    pthread_key_t key);
```

在主调线程里返回当前的 key（键）值。如果在线程里没有为 key 设置值，则返回 NULL。

参考: 5.4, 7.2, 7.3.1

头文件: <pthread.h>

错误: 用无效 key 调用 pthread_getspecific, 所产生的结果未定义。
检测不出错误。

提示: 在 destructor 函数内调用 pthread_getspecific, 返回值为
NULL。应该使用 destructor 的参数。

pthread_key_create

```
int pthread_key_create (
    pthread_key_t *key,
    void (*destructor)(void *));
```

创建一个对所有线程可见的线程私有数据 key。所有现有线程和新线程的 key 为 NULL，直到使用 pthread_setspecific 进行设置。当任何带非 NULL key 的线程终止时，用该线程的 key 的当前值调用 destructor。

参考: 5.4, 7.2, 7.3.1

头文件: <pthread.h>

错误: **[EAGAIN]** 资源不足或者 PTHREAD_KEYS_MAX 溢出。
[ENOMEM] 内存不足，不能创建key。

提示: 每个 key (pthread_key_t 变量) 只能创建一次；使用互斥量或者 pthread_once 来创建。

pthread_key_delete

```
int pthread_key_delete (
    pthread_key_t key);
```

删除线程私有数据 key。该函数不能为任何线程改变线程私有数据 key 的值，也不能在任何线程内运行 key destructor，所以使用该函数要特别注意。

参考: 5.4

头文件: <pthread.h>

错误: [EINVAL] key 无效。

提示: 只能在知道所有线程有 NULL 值时使用。

pthread_setspecific

```
int pthread_setspecific (
    pthread_key_t      key,
    const void        *value);
```

在主调线程里为设定的 key 分配线程特定的 value。

参考: 5.4, 7.2, 7.3.1

头文件: <pthread.h>

错误: [ENOMEM] 内存不足。

[EINVAL] key 无效。

提示: 设置 value 为 NULL, 线程终止时就不会调用 key 的 destructor。

9.3.8 实时调度

实时调度在进程里为重要事件提供了可预测的响应时间。注意，这种“预测”并不意味着总是快的，在许多方面，实时调度可以利用这种“减慢执行”的限制。实时调度也遇到了同步的难题，例如优先级倒置（5.5.4 和 8.1.4 部分），Pthreads 提供了可选择的程序，用来解决部分难题。

pthread_attr_getinheritsched.....[_POSIX_THREAD_PRIORITY_SCHEDULING]

```
int pthread_attr_getinheritsched (
    const pthread_attr_t   *attr,
    int                   *inheritsched);
```

判定用 attr 创建的线程在运行时是否使用调度策略，是否使用创建线程或在属性对象中设定的那些参数。默认的 inheritsched 值是由具体的 pthread 实现系统决定的。

Inheritsched

PTHREAD_INHERIT_SCHED	使用创建线程的调度策略和参数
-----------------------	----------------

PTHREAD_EXPLICIT_SCHED	使用属性对象里的调度策略和参数
------------------------	-----------------

参考: 5.2.3, 5.5

头文件: <pthread.h>

错误: [ENOSYS] 不支持优先级调度。

[EINVAL] attr 无效。

pthread_attr_getschedparam.....[_POSIX_THREAD_PRIORITY_SCHEDULING]

```
int pthread_attr_getschedparam (
    const pthread_attr_t      *attr,
    struct sched_param        *param);
```

判定用 attr 创建的线程的调度策略。默认的 param 值是由具体的 Pthread 实现系统决定的。

参考: 5.2.3, 5.5

头文件: <pthread.h>

错误: [ENOSYS] 不支持优先级调度。

[EINVAL] attr 无效。

pthread_attr_getschedpolicy.....[_POSIX_THREAD_PRIORITY_SCHEDULING]

```
int pthread_attr_getschedpolicy (
    const pthread_attr_t      *attr,
    int                      *policy);
```

判定用 attr 创建的线程的调度策略。默认的 policy 值由具体的 Pthread 实现系统决定的。

Policy	
SCHED_FIFO	运行线程直到堵塞，当就绪时抢占低优先级的线程
SCHED_RR	遵循周期性时间片调度，其他同SCHED_FIFO
SCHED_OTHER	由实现定义（可能是 SCHED_FIFO、SCHED_RR 或者其他）

参考: 5.2.3, 5.5

头文件: <pthread.h>

错误: [ENOSYS] 不支持优先级调度。

[EINVAL] attr 无效。

pthread_attr_getscope.....[_POSIX_THREAD_PRIORITY_SCHEDULING]

```
int pthread_attr_getscope (
    const pthread_attr_t      *attr,
    int                      *contentionscope);
```

判定用 attr 创建的线程的竞争范围，默认值由实现系统决定。

Contentionscope	
PTHREAD_SCOPE_PROCESS	线程与其他线程在本进程里竞争处理器资源
PTHREAD_SCOPE_SYSTEM	线程与其他线程在所有的进程里竞争处理器资源

参考: 5.2.3, 5.5

头文件: <pthread.h>

错误: [ENOSYS] 不支持优先级调度。

[EINVAL] attr 无效。

提示: 实现必须支持上述的一种方案，但是不需要两种都支持。

pthread_attr_setinheritsched.....[_POSIX_THREAD_PRIORITY_SCHEDULING]

```
int pthread_attr_setinheritsched (
    pthread_attr_t      *attr,
    int                 inheritsched);
```

将 inheritsched 的属性由 PTHREAD_INHERIT_SCHED 转变为 PTHREAD_EXPLICIT_SCHED。默认值由系统实现决定。

Inheritsched	
PTHREAD_INHERIT_SCHED	使用创建线程的调度策略和参数
PTHREAD_EXPLICIT_SCHED	使用属性对象内的调度策略和参数

参考: 5.2.3, 5.5

头文件: <pthread.h>

错误: [ENOSYS] 不支持优先级调度。

[EINVAL] attr 或者 inheritsched 无效。

pthread_attr_setschedparam.....[_POSIX_THREAD_PRIORITY_SCHEDULING]

```
int pthread_attr_setschedparam (
    pthread_attr_t      *attr,
    const struct sched_param *param);
```

设定用 attr 创建的线程使用的调度参数。默认的 param 值由具体实现系统决定。

参考: 5.2.3, 5.5

头文件: <pthread.h>

错误: [ENOSYS] 不支持优先级调度。

[EINVAL] attr或者param 无效。

[ENOTSUP] param设置为被支持的值。

pthread_attr_setschedpolicy.....[_POSIX_THREAD_PRIORITY_SCHEDULING]

```
int pthread_attr_setschedpolicy (
    pthread_attr_t *attr,
    int             policy);
```

设定用 attr 创建的线程使用的调度策略。默认的 policy 值由具体的 Pthreads 实现系统决定。

Policy

SCHED_FIFO	运行线程直到堵塞，当就绪时抢占低优先级的线程
------------	------------------------

SCHED_RR	遵循周期性时间片调度，其他同 SCHED_FIFO
----------	---------------------------

SCHED_OTHER	由实现定义（可能是 SCHED_FIFO、SCHED_RR 或者其他）
-------------	-------------------------------------

参考: 5.2.3, 5.5

头文件: <pthread.h>

错误: [ENOSYS] 不支持优先级调度。

[EINVAL] attr或者policy 策略无效。

[ENOTSUP] param设为不支持的值。

pthread_attr_setscope.....[_POSIX_THREAD_PRIORITY_SCHEDULING]

```
int pthread_attr_setscope (
    pthread_attr_t *attr,
    int             contention_scope);
```

设定由 attr 创建的线程使用的竞争范围。默认值由实现定义。

contention_scope

PTHREAD_SCOPE_PROCESS	线程与其他线程在本进程里竞争处理器资源
-----------------------	---------------------

PTHREAD_SCOPE_SYSTEM	线程与其他线程在所有的进程里竞争处理器 资源
----------------------	---------------------------

参考: 5.2.3, 5.5

头文件: <pthread.h>

错误: [ENOSYS] 不支持优先级调度。

[EINVAL] attr或者contentionscope 无效。

[ENOTSUP] contentionscope设为不支持的值。

提示: 实现可以支持其中的一个或者两个, 但是不需要同时支持两个。

pthread_getschedparam.....[_POSIX_THREAD_PRIORITY_SCHEDULING]

```
int pthread_getschedparam (
    pthread_t           thread,
    int                 *policy
    struct sched_param  *param);
```

判定当前线程使用的调度策略和参数。

Policy

SCHED_FIFO	运行线程直到堵塞, 当就绪时抢占低优先级的线程
------------	-------------------------

SCHED_RR	遵循周期性时间片调度, 其他同SCHED_FIFO
----------	---------------------------

SCHED_OTHER	由实现定义(可能是 SCHED_FIFO、SCHED_RR或者其他)
-------------	------------------------------------

参考: 5.2.3, 5.5

头文件: <pthread.h>

错误: [ENOSYS] 不支持优先级调度。

[ESRCH] thread指的不是现存的线程。

提示: 如果可能, 尽量避免动态修改线程调度策略和参数。

pthread_mutex_getprioceiling.....[_POSIX_THREAD_PRIO_PROTECT]

```
int pthread_mutex_getprioceiling (
    const pthread_mutex_t  *mutex,
    int                  *prioceiling);
```

判定优先级的上限, 在这个限度内线程拥有互斥量时运行。

参考: 3.2, 5.2.1, 5.5.5

头文件: <pthread.h>

错误: [ENOSYS] 不支持优先级调度。

[EINVAL] mutex无效。

提示: 如果互斥量的创造者不能创建和控制所有能够加锁该互斥量的线程, 保护协议就不合适。

pthread_mutex_setprioceiling.....[_POSIX_THREAD_PRIO_PROTECT]

```
int pthread_mutex_getprioceiling (
    pthread_mutex_t      *mutex,
    int                  *prioceiling,
    int                  *old_ceiling);
```

设定优先级的上限，在这个限度内线程拥有互斥量时开始运行。返回互斥量以前优先级的上限。

参考: 3.2, 5.2.1, 5.5.5

头文件: <pthread.h>

错误: [ENOSYS] 不支持优先调度。

[EINVAL] mutex 无效，或者prioceiling溢出。

[EPERM] 无权设置prioceiling。

提示: 如果互斥量的创造者不能创建和控制所有能够加锁该互斥量的线程，保护协议就不合适。

pthread_mutexattr_getprioceiling.....[_POSIX_THREAD_PRIO_PROTECT]

```
int pthread_mutexattr_getprioceiling (
    const pthread_mutexattr_t *attr,
    int                      *prioceiling);
```

判定优先级的上限，在这个限度内线程拥有由 attr 创建的互斥量时开始运行。

参考: 3.2, 5.2.1, 5.5.5

头文件: <pthread.h>

错误: [ENOSYS] 不支持优先级调度。

[EINVAL] attr 无效。

提示: 如果互斥量的创造者不能创建和控制所有能够加锁该互斥量的线程，保护协议就不合适。

pthread_mutexattr_getprotocol.....[_POSIX_THREAD_PRIO_INHERIT_POSIX_THREAD_PRIO_PROTECT]

```
int pthread_mutexattr_getprotocol (
    const pthread_mutexattr_t *attr,
    int                      *protocol);
```

判定由 attr 创建的互斥量是否有优先级上限协议 (Protect) 保护协议、优先级继承协议 (inherit)，或者没有优先级的协议 (none)。

protocol	
PTHREAD_PRIO_NONE	无优先级继承协议
PTHREAD_PRIO_INHERIT	拥有互斥量的线程继承等待互斥量的最高优先级线程的优先级
PTHREAD_PRIO_PROTECT	拥有互斥量的线程继承互斥量的优先级上限

参考: 3.2, 5.2.1, 5.5.5

头文件: <pthread.h>

错误: [ENOSYS] 不支持优先级调度。

[EINVAL] attr 无效。

提示: 继承协议代价很高，并且如果互斥量的创造者不能创建和控制所有能够加锁该互斥量的线程，保护协议就不适用。

`pthread_mutexattr_setprioceiling`.....[_POSIX_THREAD_PRIO_PROTECT]

```
int pthread_mutexattr_setprioceiling (
    pthread_mutexattr_t *attr,
    int                  prioceiling);
```

设定优先级的上限，在这个限度内，线程拥有由 attr 创建的互斥量时开始运行。对于 SCHED_FIFO，prioceiling 值必须是一个有效的优先级参数。

参考: 3.2, 5.2.1, 5.5.5

头文件: <pthread.h>

错误: [ENOSYS] 不支持优先级调度。

[EINVAL] attr 或者 prioceiling 无效。

[EPERM] 无权设置 prioceiling。

提示: 如果互斥量的创造者不能创建和控制所有能够加锁该互斥量的线程，保护协议就不合适。

`pthread_mutexattr_setprotocol`.....[_POSIX_THREAD_PRIO_INHERIT_POSIX_THREAD_PRIO_PROTECT]

```
int pthread_mutexattr_setprotocol (
    pthread_mutexattr_t *attr,
    int                  protocol);
```

设定由 attr 创建的互斥量是否有优先级上限协议 (Protect)、优先级继承协议 (inherit)，或者没有优先级协议 (none)。

protocol	
PTHREAD_PRIO_NONE	无优先级继承协议
PTHREAD_PRIO_INHERIT	拥有互斥量的线程继承等待互斥量的最高优先级线程的优先级
PTHREAD_PRIO_PROTECT	拥有互斥量的线程继承互斥量的优先级上限

参考: 3.2, 5.2.1, 5.5.5

头文件: <pthread.h>

错误: [ENOSYS] 不支持优先级调度。

[EINVAL] attr或者protocol无效。

[ENOTSUP] 不支持protocol值。

提示: 继承协议的代价很高, 如果互斥量的创造者不能创建和控制所有能够给该互斥量加锁的线程, 保护协议就不适用。

pthread_setschedparam.....[_POSIX_THREAD_PRIORITY_SCHEDULING]

```
int pthread_setschedparam (
    pthread_t           thread,
    int                 policy
    const struct sched_param *param);
```

设定线程使用的调度策略和参数

Policy	
SCHED_FIFO	运行线程直到堵塞, 当就绪时抢占低优先级的线程
SCHED_RR	遵循周期性时间片调度, 其他同SCHED_FIFO
SCHED_OTHER	由实现定义(可能是 SCHED_FIFO、SCHED_RR或者其他)

参考: 5.5

头文件: <pthread.h>

错误: [ENOSYS] 不支持优先级调度。

[ESRCH] thread指的不是现存的线程。

[EINVAL] policy或者param无效。

[ENOTSUP]不支持policy或者param值。

[EPERM] 无权设置policy和param。

提示: 如果可能, 尽量避免动态修改线程调度策略和参数

sched_get_priority_max.....[_POSIX_PRIORITY_SCHEDULING]

```
int sched_get_priority_max (
    int policy);
```

返回特定调度 *policy* 许可的最大整数优先级。

policy

SCHED_FIFO	运行线程直到堵塞，当就绪时抢占低优先级的线程
------------	------------------------

SCHED_RR	遵循周期性时间片调度，其他同SCHED_FIFO
----------	--------------------------

SCHED_OTHER	由实现定义（可能是 SCHED_FIFO、 SCHED_RR或者其他）
-------------	-------------------------------------

参考: 5.5.2

头文件: <sched.h>

错误: [ENOSYS] 不支持优先级调度。

[EINVAL] *policy*无效。

提示: 优先级的最小值和最大值都是整数，可以计算出该范围内的大部分值。

sched_get_priority_min.....[_POSIX_PRIORITY_SCHEDULING]

```
int sched_get_priority_min (
    int policy);
```

返回特定调度 *policy* 许可的最小整数优先级。

policy

SCHED_FIFO	运行线程直到堵塞，当就绪时抢占低优先级的线程
------------	------------------------

SCHED_RR	遵循周期性时间片调度，其他同SCHED_FIFO
----------	--------------------------

SCHED_OTHER	由实现定义（可能是 SCHED_FIFO、 SCHED_RR或者其他）
-------------	-------------------------------------

参考: 5.5.2

头文件: <sched.h>

错误: [ENOSYS] 不支持优先级调度。

[EINVAL] *policy*无效。

提示: 优先级的最小值和最大值都是整数，可以计算出该范围内的大部分值。

9.3.9 分支处理程序

Pthreads 提供了一些新的函数以帮助新的线程环境与传统的基于进程的 UNIX 环境共存。如 fork (分支) 调用与该进程中的其他线程响应是异步的，拷贝全部的地址空间创建一个子进程，就会引发线程应用的困难。

`pthread_atfork`

```
int pthread_atfork (
    void (*prepare)(void),
    void (*parent)(void),
    void (*child)(void));
```

当进程创建了一个子进程时，定义要运行的“分支处理程序”。允许在子进程中保护同步对象和共享数据（否则很难控制）。

参考: 6.1.1

头文件: <unistd.h>^①

错误: [ENOMEM] 空间不足不能记录处理程序。

提示: 必须保护子进程所需的资源。

9.3.10 Stdio

Pthreads 提供了一些新的函数和一些旧函数的新版本，可以从多线程进程中安全地访问 ANSI C stdio。出于安全考虑，旧的访问 stdio 缓冲的 single-character 形式改变成给文件流加锁的方式，但这样会降低性能。可以通过改变旧的代码以代替人工给文件流加锁；在需要锁定的区域里，可以使用新的特性访问没有给文件流加锁的操作。

`flockfile`

```
void flockfile (
    FILE *file);
```

提高 stdio 文件流锁的数量，从而获得对文件流的专有访问。如果当前其他线程给该文件流加了锁，那么主调线程就会拥塞，直到文件流的锁计数为零时为止。如果主调线程已经拥有了该文件流锁，那么锁计数则增加一个同样的调用数值给 funlockfile，用以释放文件流锁。

尽管大多数 stdio 函数（如 printf 和 fgets）都是线程安全的，你或许发现调用 printf 的顺序也是非常重要的，例如，从一个线程调用它时，还需要其他线

① Digital UNIX 和 Solaris（错误地）将该定义放在〈pthread.h〉中。UNIX 98 有望改变这种情况。

程的帮助。当然了，也有少数 stdio 函数不是线程安全的，只能在调用者给文件流加锁时使用。

参考: 6.4.1

头文件: <stdio.h>

提示: 用来保护 stdio 操作的顺序。

ftrylockfile

```
int ftrylockfile (
    FILE *file);
```

如果文件流被其他线程锁定，返回一个非零值，否则，增加文件流锁计数并且返回零值。

参考: 6.4.1

头文件: <stdio.h>

提示: 用来保护 stdio 操作的顺序。

funlockfile

```
void funlockfile (
    FILE *file);
```

对以前 funlockfile 或 ftrylockfile 加锁的 stdio 文件流，减少锁计数。如果锁计数为 0，就打开锁，从而其他的线程可以重新给它加锁。

参考: 6.4.1

头文件: <stdio.h>

提示: 用来保护 stdio 操作的顺序。

getc_unlocked

```
int getc_unlocked (
    FILE *file);
```

从 stdio 流文件中返回一个单一的字符，而文件流不用加锁。这个操作仅仅在下面两种情况下使用：调用 flockfile 给文件流加了锁，或者预知当前没有任何线程要访问文件流。返回 EOF 作为读错误或者文件末端的标示。

参考: 6.4.2

头文件: <stdio.h>

提示: 用于代替旧的 getc 调用，以获得最快的访问。

getchar_unlocked

```
int getchar_unlocked (void);
```

从 stdio 流 stdin 中返回一个单一的字符，而文件流不用上锁。这个操作仅仅在下面两种情况下使用：调用 flockfile 给文件流加了锁，或者预知当前没有任何线程要访问文件流。返回 EOF 作为读错误或者文件末端的标示。

参考： 6.4.2

头文件： <stdio.h>

提示： 用于代替旧的 getchar 调用，以获得更快的访问。

putc_unlocked

```
int putc_unlocked (
    int             c,
    FILE           *file);
```

写入一个单一的字符 c 给 stdio 流文件，而不用给文件流上锁。这个操作仅仅在下面两种情况下使用：调用 flockfile 给文件流加了锁，或者预知当前没有任何线程要访问文件流。如果有错误发生，则返回这个字符或 EOF 值。

参考： 6.4.2

头文件： <stdio.h>

提示： 用于代替旧的 putc 调用，以获得更快的访问。

putchar_unlocked

```
int putchar_unlocked (
    int             c);
```

写一个单一的字符 c 给 stdio 流 stdout，而不用给文件流上锁。这个操作仅仅在下面两种情况下使用：调用 flockfile 给文件流加了锁，或者预知当前没有任何线程要访问文件流。如果有错误发生，则返回这个字符或 EOF 值。

参考： 6.4.2

头文件： <stdio.h>

提示： 用于代替旧的 putchar 调用，以获得更快的访问。

9.3.11 线程安全函数

线程安全函数改良了访问传统的 ANSI C 和 POSIX 的功能(该功能效率不高)，而保持接口不变。传统函数名后缀加_r 就是这些程序，例如 getlogin_r 代替了 getlogin。

getlogin_r

```
int getlogin_r (
    char           *name,
    size_t          namesize);
```

将与当前进程相关的用户名称写入 name 指向的缓冲区。该缓冲区字节长度为 namesize，并且有空间留给该名称和一个终止的空字符。登录名称的最大长度是 LOGIN_NAME_MAX。

参考: 6.5.1

头文件: <unistd.h>

readdir_r

```
int readdir_r (
    DIR          *dirp,
    struct dirent *entry,
    struct dirent **result);
```

在 dirp 指向的目录流的当前位置上，返回一个指向目录项的指针(result)。鉴于 readdir 保留了指向下一个目录项的指针，readdir_r 将下个目录项写入 entry 参数设定的缓冲区。

参考: 6.5.2

头文件: <sys/types.h>, <dirent.h>

错误: [EBADF] dirp 不是开放的目录流。

strtok_r

```
char *strtok_r (
    char        *s,
    const char  *sep,
    char        **lasts);
```

返回一个指向字符串内的下一个标记的指针。鉴于 strtok 用静态变量保留了当前字符串内的位置，strtok_r 使用调用者提供的 lasts 参数。

参考: 6.5.3

头文件: <string.h>

asctime_r

```
char *asctime_r (
    const struct tm  *tm,
    char            *buf);
```

将 tm 指向的结构里的 broken-down 时间转换成字符串，存储在 buf 指向的缓冲区中。由 buf 指向的缓冲区至少有 26 个字节。如果成功，函数返回指向缓冲区的指针，否则返回 NULL 值。

参考: 6.5.4

头文件: <time.h>

ctime_r

```
char *ctime_r (
    const time_t      *clock,
    char             *buf);
```

将 *clock* 指向的 calendar 时间转换成一个字符串, 表示本地时间, 存储在 *buf* 指向的缓冲区中。由 *buf* 指向的缓冲区至少有 26 个字节。如果成功, 函数返回指向缓冲区的指针, 否则返回 NULL 值。

参考: 6.5.4

头文件: <time.h>

gmtime_r

```
struct tm *gmtime_r (
    const time_t      *clock,
    struct tm        *result);
```

将 *clock* 指向的 calendar 时间转换成 broken-down 时间, 用以表示格林威治时间, 存储在由 *result* 指向的结构里。如果成功, 函数返回指向缓冲区的指针, 否则返回 NULL 值。

参考: 6.5.4

头文件: <time.h>

localtime_r

```
struct tm *localtime_r (
    const time_t      *clock,
    struct tm        *result);
```

将 *clock* 指向的 calendar 时间转换成 broken-down 时间, 用以表示本地时间, 存储在由 *result* 指向的结构里。如果成功, 函数返回指向缓冲区的指针, 否则返回 NULL 值。

参考: 6.5.4

头文件: <time.h>

rand_r

```
int rand_r (
    unsigned int      *seed);
```

从 0~RAND_MAX 范围的随机整数序列内返回下一个值。鉴于 rand 使用一个静态变量保留一系列调用间的前后关系，rand_r 用由调用者提供的 seed 指向该值。

参考: 6.5.5

头文件: <stdlib.h>

getgrgid_r

```
int getgrgid_r (
    gid_t                 gid,
    struct group          *group,
    char                  *buffer,
    size_t                bufsize,
    struct group          **result);
```

从用户组数据库中定位一个与 gid 参数匹配的项。存储在 buffer 指向的内存中，该内存有 bufsize 个字节，指向该项的指针存储在由 result 指向的地址中。最大的缓冲区大小通过调用带有 _SC_GETGR_R_SIZE_MAX 参数的 sysconf 确定。

参考: 6.5.6

头文件: <sys/types.h>, <grp.h>

错误: [ERANGE] 设定的缓冲区太小。

getgrnam_r

```
int getgrnam_r (
    const char            *name,
    struct group          *group,
    char                  *buffer,
    size_t                bufsize,
    struct group          **result);
```

从用户组数据库中定位一个与 name 参数匹配的用户组项，存储在 buffer 指向的内存中，该内存有 bufsize 个字节，指向该项的指针存储在由 result 指向的地址中。最大的缓冲区大小通过调用带有 _SC_GETGR_R_SIZE_MAX 参数的 sysconf 确定。

参考: 6.5.6

头文件: <sys/types.h>, <grp.h>

错误: [ERANGE] 设定的缓冲区太小。

getpwuid_r

```
int getpwuid_r (
    uid_t             uid,
    struct passwd     *pwd,
    char              *buffer,
    size_t            bufsize,
    struct passwd     **result);
```

从用户数据库中定位一个与 *uid* 参数匹配的用户项。该用户项存储在 *buffer* 指向的内存中，该内存有 *bufsize* 个字节，指向该项的指针存储在由 *result* 指向的地址中。最大的缓冲区大小通过调用带有 *_SC_GETGR_R_SIZE_MAX* 参数的 *sysconf* 确定。

参考: 6.5.6

头文件: <sys/types.h>, <pwd.h>

错误: [ERANGE] 指定的缓冲区太小。

getpwnam_r

```
int getpwnam_r (
    const char        *name,
    struct passwd     *pwd,
    char              *buffer,
    size_t            bufsize,
    struct passwd     **result);
```

在用户数据库中定位与 *name* 参数匹配的项。该用户项存储在 *buffer* 指向的内存中，该内存有 *bufsize* 个字节，指向该项的指针存储在由 *result* 指向的地址中。最大的缓冲区大小通过调用带有 *_SC_GETGR_R_SIZE_MAX* 参数的 *sysconf* 确定。

参考: 6.5.6

头文件: <sys/types.h>, <pwd.h>

错误: [ERANGE] 指定的缓冲区太小。

9.3.12 信号

Pthreads 提供了 POSIX 信号模型的扩展函数，用以支持多线程的进程。进程中的所有线程都共享同样的信号行为。每个线程都有自己待处理和阻塞的信号掩码。进程也有挂起信号的掩码，所以当所有的线程都有信号阻塞时，异同信号可以挂起。在多线程的进程中不定义 *sigprocmask* 的行为。

pthread_kill

```
int pthread_kill (
    pthread_t      thread,
    int            sig);
```

将 sig 信号请求传送给线程。如果 sig 为 0，则不送信号，但是执行错误检测。如果信号动作是终止、停止或者继续，则整个进程将受影响。

参考: 6.6.3

头文件: <signal.h>

错误: [ESRCH] 无线程间通信。

[EINVAL] 无效的 sig 信号值。

提示: 用于取消终止线程。

pthread_sigmask

```
int pthread_sigmask (
    int           how,
    const sigset_t *set,
    sigset_t       *oset);
```

用作在主调线程里控制信号掩码。

how

SIG_BLOCK	结果集是当前集和参数集的并集
-----------	----------------

SIG_UNBLOCK	结果集是当前集和参数集的差集
-------------	----------------

SIG_SETMASK	结果集是由参数集指向的集
-------------	--------------

参考: 6.6.2

头文件: <signal.h>

错误: [EINVAL] how 不是已定义值

提示: 除非信号在所有的线程里都被阻塞，否则总能将异步信号传输给这个进程。

sigtimedwait

```
int sigtimedwait (
    const sigset_t      *set,
    siginfo_t            *info,
    const struct timespec *timeout);
```

如果 set 里的信号挂起，就从挂起信号的集合中清除掉它，同时在 info 的 si-signo 成员中返回信号号码。信号起因将被储存在 si_code 成员内。如果有任何值列队等候选择信号，则返回在 si_value 成员内的第一个队列值。如果 set 里没有信号挂起，则悬挂主调程序直到一个或更多的信号挂起为止。如果由 timeout 制定的时间片超时，则 sigtimedwait 将返回 EAGAIN 错误。函数返回信号号码，错误时函数返回值为 -1，同时将 error 指向适当的错误码。

参考: 6.6.4

头部: <signal.h>

错误: [EINVAL] 包含一个无效的信号号码。

[EAGAIN] 时间片超时。

[ENOSYS] 不支持实时信号。

提示: 仅仅能传送异步的信号。在 set 中的所有信号在主调线程里必须屏蔽，并且在所有的线程里通常应该被屏蔽。

sigwait

```
int sigwait (
    const sigset_t *set,
    int             *sig);
```

如果 set 里的信号挂起，就从挂起信号的集合中清除掉它，同时返回由 sig 参数指向的信号号码，二者密不可分。如果 set 里的没有信号挂起，则悬挂主调程序直到一个或更多的信号挂起为止。

参考: 6.6.4

头文件: <signal.h>

错误: [EINVAL] 包含一个无效的信号号码。

提示: 仅仅能传送异步信号。set 内的所有信号在主调线程里必须被屏蔽，并且在所有的线程里通常应该被屏蔽。

sigwaitinfo

```
int sigwaitinfo (
    const sigset_t  *set,
    siginfo_t       *info);
```

如果 set 里的信号挂起，就从挂起信号的集合中清除掉它，同时在 info 的成员 si-signo 中返回信号号码。信号起因将被储存在 si_code 成员内。如果有任何值列队等候选择信号，则返回在 si_value 成员内的第一个队列值。如果 set 里的没有信号挂起，则悬挂主调程序直到一个或更多的信号挂起为止。函数返回信号号码，错误时函数返回值为 -1，同时将 error 指向适当的错误码。

参考: 6.6.4

头文件: <signal.h>

错误: [EINVAL] 包含一个无效的信号码。

[ENOSYS] 不支持实时信号。

提示: 仅仅能传送异步信号。在 set 中的所有信号在主调线程里必须被屏蔽，并且在所有的线程里通常应该被屏蔽。

9.3.13 信号灯

信号灯源于 POSIX.1b (POSIX 1003.1b 1993)，而不是 Pthreads。它们遵循旧的 UNIX 报错习惯，即发生错误时返回 -1，并将相应的错误码保存在 errno 变量里。所有的信号灯函数都需要包含头文件<semaphore.h>。

sem_destroy [_POSIX_SEMAPHORES]

```
int sem_destroy (
    sem_t *sem);
```

释放未命名的信号灯。

参考: 6.6.6

头文件: <semaphore.h>

错误: [ENOSYS] 不支持信号灯。

[EBUSY] 在 sem 上当前的线程或进程被阻塞。

sem_init [_POSIX_SEMAPHORES]

```
int sem_init (
    sem_t *sem,
    int pshared,
    unsigned int value);
```

初始化一个未命名的信号灯。信号灯计数器的初始值就是 value 的值。如果 pshared 变量有一个非零值，那么该信号灯就可以在进程间共享。如果 pshared 变量有一个零值，那么该信号灯就可以在同一个进程内的线程间共享。

参考: 6.6.6

头文件: <semaphore.h>

错误: [EINVAL] sem 不是一个有效的信号灯。

[ENOSPC] 所需资源耗尽。

[ENOSYS] 不支持信号灯。

[EPERM] 本进程无相应的权限。

提示: 使用值 1 表示锁定，值 0 表示等候。

sem_trywait.....[_POSIX_SEMAPHORES]

```
int sem_trywait (
    sem_t             *sem);
```

试图在一个信号灯上等待（或者试图锁定一个信号灯）。如果信号灯值大于零，则该值减 1。如果信号灯值为 0，则立刻返回 EAGAIN 错误。

参考: 6.6.6

头文件: <semaphore.h>

错误: [EAGAIN] 信号灯已经锁定。

[EINVAL] 超过SEM_VALUE_MAX的值。

[EINVAL] sem不是一个有效的信号灯。

[EINTR] 函数被一个信号打断。

[ENOSYS] 不支持信号灯。

[EDEADLK] 探测到死锁状态。

提示: 当信号灯初始化值为 1 时，是一个锁定操作；信号灯初始化值为 0 时，是一个等待操作。

sem_post.....[_POSIX_SEMAPHORES]

```
int sem_post (
    sem_t             *sem);
```

发送一个唤醒消息给信号灯。如果有等候的线程（或者进程），其中的一个被唤醒。如果没有，信号灯值加 1。

参考: 6.6.6

头文件: <semaphore.h>

错误: [EINVAL] sem 不是一个有效的信号灯。

[ENOSYS] 不支持信号灯。

提示: 可以用于信号处理函数内。

sem_wait.....[_POSIX_SEMAPHORES]

```
int sem_wait (
    sem_t             *sem);
```

等候信号灯（或者锁定信号灯）。如果信号灯值大于 0，则该值减 1。如果信号灯值是 0，主调线程（或进程）被阻塞，一直等待信号灯值能够成功地降低或者有信号打断这一过程。

参考: 6.6.6

头文件: <semaphore.h>

错误: **[EINVAL]** sem 不是一个有效的信号灯。

[EINTR] 函数被一个信号打断。

[ENOSYS] 不支持信号灯。

[EDEADLK] 探测到死锁状态。

提示: 当信号灯初始化值为 1 时, 是一个锁定操作; 初始化值为 0 时, 是一个等待操作。

标准化过程展望

Pthreads 程序员受三个主要的标准化研究成果影响很大。X/Open 的 XSH5 是一个新的接口规范，包含 POSIX.1b、Pthreads 和一套附加的线程函数 POSIX.1j 草案标准提出增加 barrier、读 / 写锁、自旋锁 (spinlock)，并且改进了对条件变量的“相对时间”等待支持。POSIX.14 草案标准（一个“POSIX 标准子集”）指明了在多处理器环境下如何管理 Pthreads 的多种选项。

10.1 X/Open XSH5 [UNIX98]

互斥量类型属性：

```
int pthread_mutexattr_gettype (
    const pthread_mutexattr_t *attr, int *type);
int pthread_mutexattr_settype (
    pthread_mutexattr_t *attr, int type);
```

读/写锁：

```
int pthread_rwlock_init (pthread_relock_t *rwlock,
    const pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy (pthread_rwlock_t *rwlock);
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
int pthread_rwlock_rdlock (pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock (
    pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock (pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock (
    pthread_rwlock_t *rwlock);
int pthread_rwlockattr_init (
    pthread_rwlockattr_t *attr);
int pthread_rwlockattr_destroy (
    pthread_rwlockattr_t *attr);
int pthread_rwlockattr_getshared (
    const pthread_rwlockattr_t *attr, int *pshared);
int pthread_rwlockattr_setpshared (
    pthread_rwlockattr_t *attr, int pshared);
```

并行 I/O:

```
size_t pread (int fildes,
    void *buf, size_t nbyte, off_t offset);
size_t pwrite (int fildes,
    const void *buf, size_t nbyte, off_t offset);
```

其他:

```
int pthread_attr_getguardsize (
    const pthread_attr_t *attr, size_t *guardsize);
int pthread_attr_setguardsize (
    pthread_attr_t *attr, size_t guardsize);
int pthread_getxconcurrency ();
int pthread_setconcurrency (int new_level);
```

X/Open 是 The Open Group 的一部分，拥有 UNIX 商标并且发展了 UNIX 工业可移植性规范。X/Open 包括 XPG3、XPG4、UNIX93 和 UNIX95。UNIX95 也被称为“SPEC1170”或“Single UNIX Specification”。

X/Open 最近新出了 X/Open CAE 规范、系统接口和头文件、版本 5(即 XSH5)，它是新 UNIX98 旗下的一部分。XSH5 要求在 POSIX.1-1996 标准(包括 POSIX.1b 和 POSIX.1c 修正版)下运行。XSH5 也扩展了 POSIX。这一节论述了 XSH5 的扩展对多线程程序的特殊影响。你可以通过在<unistd.h> 定义一个取值为 500 或更高的_XOPEN_VERSION 标志，来识别系统对 XSH5 的兼容性。

然而 UNIX98 对多线程编程的最有价值的贡献却可能是发展了标准化的、可移植的测试系统。由于运行 Pthreads 时会出现一些复杂的问题，并且在标准的一些细小方面也是比较含糊的，所以这个测试系统要求所有的销售商都执行与 Pthreads 规格一致的 UNIX98 系统。

10.1.1 POSIX options for XSH5

在 Pthread 标准中 XSH5 需要一些选项的特征值。如果你的编码是建立在这些选项的基础上，它就可以在任何兼容 XSH5 的系统下运行。

- _POSIX_THREADS: 支持线程。
- _POSIX_THREAD_ATTR_STACKADDR: 支持 stackaddr 属性。
- _POSIX_THREAD_ATTR_STACKSIZE: 支持 stacksize 属性。
- _POSIX_THREAD_PROCESS_SHARED: 进程间可共享条件变量，互斥量、XSH5 的读/写锁。
- _POSIX_THREAD_SAFE_FUNCTIONS: 支持 Pthreads 的线程安全函数。

若干附加的 Pthreads 选项被捆绑在 XSH5 实时线程选项组。如果你的系统兼容并支持_XOPEN_REALTIME_THREADS 选项，那么也支持这些附加的选项。

- _POSIX_THREAD_PRIORITY_SCHEDULING: 支持实时优先级调度。

- `_POSIX_THREAD_PRIO_PROTECT`: 支持优先级 ceiling 互斥量。
- `_POSIX_THREAD_PRIO_INHERIT`: 支持优先级继承互斥量。

10.1.2 互斥量的类型

DCE 线程包提供了一个允许编程者规定所创建互斥量种类的扩展。DCE 线程提供快速 (fast)、可递归 (recursive) 和不可递归 (nonrecursive) 互斥量。XSH5 规约把属性的名字从“种类” (kind) 改成“类型” (type)，重新命名 *fast* 为 *default*、*nonrecursive* 为 *errorcheck*，并且增加了一种新类型 *normal* (表 10.1)。

normal 互斥量不能检查死锁错，也就是说如果一个线程锁定一个它所拥有的 *normal* 互斥量，它就会被挂起。*default* (默认) 互斥量类型与 DCE *fast* 互斥量^①一样，提供给定指令的检错。也就是说，*default* 可能映射另一标准类型或者可能就是彻底不同的其他类型。

表 10.1 XSH5 互斥量类型

互斥量类型	定义
<code>PTHREAD_MUTEX_NORMAL</code>	无特殊错误检查的基本互斥量，不能报出死锁错误
<code>PTHREAD_MUTEX_RECURSIVE</code>	允许任何线程递归地加锁该互斥量—必须开锁相等次数才能释放这个互斥量
<code>PTHREAD_MUTEX_ERRORCHECK</code>	发现并报告简单的应用错误——该错误是试图为没有被主调线程锁定（或根本没有上锁）的互斥量开锁，或者是试图重新给该线程已经拥有的互斥量加锁
<code>PTHREAD_MUTEX_DEFAULT</code>	默认的互斥量类型，采用非常松散的语义，允许自由更新和试验。可以映射为其他三种定义类型，或者完全别的东西

作为开发应用的人，只要你的编码不依赖于这样的实现，去检测（或失败检测）一些特殊的错误，你就可以交替地使用任何一个类型的互斥量。但是绝不要编写基于没有任何错误检测的实现的代码。例如，不能在一个线程中给一个互斥量加锁而在另一个线程中打开它，即使你确信不会报出这个错误——可以使用一个没有“所有权”语义的信号灯。

无论何种类型的互斥量都用 `pthread_mutex_init` 创建，用 `pthread_mutex_destroy` 释放，用 `pthread_mutex_lock`、`pthread_mutex_unlock` 和 `pthread_mutex_trylock` 操作。

normal 互斥量是最快的实现，但同时提供最少的错误检查。

recursive 互斥量主要用于在难于建立清晰的同步界限时转换旧编码。例如，当

^① DEC 线程实现的 *fast* 互斥量很像 XSH5 的 *normal* 互斥量，没有错误检测。不过，这不是规范的意图。

你调用一个函数时，必须加锁某个互斥量，而你调用的函数（或者它调用的某个函数）需要锁住同一个互斥量时，那么就很难建立一个清楚的同步界限了。我从来没有看到需要 recursive 互斥量去解决问题的情形，但是我曾看到过许多替换方案失效的例子。这频繁地导致编程者创建 recursive 互斥量，并且使单一的可用于各种情形的实现更有意义（但如果你不用 recursive 互斥量，你的编码可能运行得更好）。

errorcheck 互斥量被设计成一个调试工具，几乎没有（可用的）中断调试工具比它更好。使用 errorcheck 互斥量，必须重新编译代码来打开或者关闭调试功能。更有用的是，它有一个附加选项，强迫所有的互斥量记录调试数据。你可能想在最终的“产品”代码内使用 errorcheck 互斥量，当然，你也可能想尽早地检查出严重的错误，但你要意识到由于 errorcheck 互斥量的额外负载，它比 normal 互斥量要慢得多。

default 互斥量允许实现提供互斥量语义，卖方认为这对顾客最有用途。例如，设 errorcheck 互斥量为默认，可以改善系统线程调试环境。再如，销售商选择 normal 互斥量为默认，可以提高程序的运行速度。

`pthread_mutexattr_gettype`

```
int pthread_mutexattr_gettype (
    pthread_mutexattr_t *attr,
    int type);
```

判定用 attr 创建的互斥量类型。

	type
PTHREAD_MUTEX_DEFAULT	未设定的类型
PTHREAD_MUTEX_NORMAL	不带错误检查的基本互斥量类型
PTHREAD_MUTEX_RECURSIVE	可以给自己拥有的互斥量再加锁的类型
PTHREAD_MUTEX_ERRORCHECK	检查用法错误的类型

参考: 3.2, 5.2.1, 10.1.2

错误: [EINVAL] 无效的类型。

[EINVAL] attr 无效。

提示: *normal* 互斥量通常最快; *errorcheck* 互斥量用于调试; *recursive* 互斥量用于旧接口的线程安全。

`pthread_mutexattr_settype`

```
int pthread_mutexattr_settype (
    pthread_mutexattr_t *attr,
    int type);
```

设定用 attr 创建的互斥量的类型。

Type	
PTHREAD_MUTEX_DEFAULT	未设定的类型
PTHREAD_MUTEX_NORMAL	不带错误检查的基本互斥量类型
PTHREAD_MUTEX_RECURSIVE	可以给自己拥有的互斥量再加锁的类型
PTHREAD_MUTEX_ERRORCHECK	检查用法错误的类型

参考: 3.2, 5.2.1, 10.1.2

错误: [EINVAL] 无效的类型。

[EINVAL] attr 无效。

提示: *normal* 互斥量通常最快; *errorcheck* 互斥量用于调试; *recursive* 互斥量用于旧接口的线程安全。

10.1.3 设置并发级别

使用 Pthreads 实现在一些小的内核实体(见书的 5.6.3 部分)上调度用户线程, 当所有分配给这个进程的内核实体都忙的时候, 可能已经有用户的线程就绪了。某些实现(例如, 在内核里阻塞的用户线程给一个核实体加锁)直到完成阻塞的条件如 I/O 请求时才解锁。系统将为这个进程创建适当数量的核实体, 但是最终核实体池会耗尽。这个进程可以留下一些线程, 这些线程有能力为应用完成有用的工作, 但是无法调度它们。

`pthread_setconcurrency` 函数用来设定限制, 它允许应用请求更多的核实体。如果应用的设计者意识到, 内核里 15 个线程中的 10 个在某个时间阻塞了, 而且很重要的是其他 5 个线程能够继续工作, 于是这个应用就向内核申请了 15 个核实体; 如果 5 个中至少有 1 个继续工作, 但不是所有的线程都继续工作, 那么这个应用就向内核保守地申请 11 个核实体。或者如果所有的线程一会儿阻塞一次, 并且你知道在任何时候不可能有 6 个以上的线程同时阻塞, 那么这个应用可能就申请 7 个核实体。

`pthread_setconcurrency` 函数是一个提示, 实现可能忽略或者修改了它的说明。在任何与 UNIX98 一致的系统中, 都可以自由地使用该函数, 但是许多系统只是通过 `pthread_getconcurrency` 返回一个设定值。例如, 在 Digital UNIX 上, 不需要设定一个固定的并发级别, 原因是内核模式和用户模式调度程序互相协作, 保证了就绪的用户线程, 使其他在内核阻塞的线程无法阻止它的运行。

`pthread_getconcurrency`

```
int pthread_getconcurrency (void);
```

返回以前调用 `pthread_setconcurrency` 设置的值。如果以前没有对 `pthread_setconcurrency` 的调用，则返回 0 显示实现正在自动保存并发级别。

参考: 5.6.3, 10.1.3

错误: 无

提示: 并发级别是一个提示。它可能被一些实现忽略，并且将要被一个不需要确保并发的实现所忽略。

`pthread_setconcurrency`

```
int pthread_getconcurrency (int new_level);
```

允许应用通知线程的实现，告诉它所需的最小并发级别。该函数导致的实际的并发级别是不确定的。

参考: 5.6.3, 10.1.3

错误: **[EINVAL]** `new_level` 是负数。

[EAGAIN] `new_level` 系统资源不足。

提示: 并发级别是一个提示。它可能被一些实现忽略，并且将要被一个不需要确保并发的实现所忽略。

10.1.4 堆栈警戒尺寸

警戒尺寸来源于 DCE threads。大多数线程实现为线程堆栈增加了一个警戒 (guard) 区域，增加了一页或者更多的保护内存。保护页是一个安全的区域，它可以阻止线程内的堆栈溢出，避免破坏其他线程的堆栈。控制线程的警戒尺寸有两个理由：

1. 允许在堆栈上分配有大数组的应用或者函数库增加默认的警戒尺寸。例如，如果一个线程一次分配两页，则一个单独的警戒页对避免堆栈溢出提供很少的保护——线程可能越过保护页而破坏邻近的内存。
2. 创建大量的线程时，每个堆栈的额外的保护页可能会成为严重的负担。除了额外的保护页外，内核的内存管理器在临近页上保持对不同保护的跟踪，这可能消耗系统资源。因此，你应该要求系统相信你，并且避免给线程分配任何警戒页。可以通过申求 0 字节的警戒尺寸实现这种方法。

`pthread_attr_getguardsize`

```
int pthread_attr_getguardsize (
    const pthread_attr_t      *attr,
    size_t                      *guardsize);
```

判定堆栈的警戒区域尺寸，在该堆栈上，用 `attr` 创建的线程将要运行。

参考: 2, 5.2.3

错误: [EINVAL] attr 无效。

提示: 在地址空间里, 设定为 0 适合大部分堆栈, 或者线程在堆栈上分配大的缓冲区, 增加默认的 guardsize。

pthread_attr_setguardsize

```
int pthread_attr_setguardsize (
    pthread_attr_t *attr,
    size_t guardsize);
```

attr 创建的线程在堆栈上运行时, 使用 guardsize 字节保护, 以免堆栈溢出。可能将 guardsize 四舍五入为下一个 PAGESIZE 的倍数。设定 guardsize 为 0 值, 会造成用属性对象创建的线程运行时, 没有堆栈溢出保护。

参考: 2, 5.2.3

错误: [EINVAL] guardsize 或者 attr 无效。

提示: 在地址空间里, 设定为 0 适合大部分堆栈, 或者为需要在堆栈上分配大缓冲区的线程增加默认的 guardsize。

10.1.5 并行 I/O

许多高性能的系统, 如数据库引擎, 至少在某些部分使用线程, 并且通过其中的并行 I/O 以获取性能。不幸的是 Pthreads 不直接支持并行 I/O。即两个线程可以独立地执行文件 I/O 操作, 甚至是同一个文件这样做, 但是 POSIX 文件 I/O 模型在并行的级别上设置了一些限制。

一个瓶颈在于: 当前文件的位置是这个文件描述符的一个属性。为了从一个文件的特定区域读写数据, 线程必须调用 lseek, 搜寻文件里合适的字节偏移量, 然后再进行读写。如果不止一个线程在同一个时间里这样做, 第一个线程搜寻, 然后第二个线程在第一个线程执行读写操作之前将搜寻不同的地方。

X/Open 的 pread 和 pwrite 函数提供了一个解决方案: 进行一个查询, 同时读或写, 两者密不可分。线程能够并行发布 pread 或者 pwrite 操作。大体上, 系统不用给文件描述符加锁就可以完全并行处理那些 I/O 请求。

pread

```
size_t pread (
    int fildes,
    void *buf,
    size_t nbyte,
    off_t offset);
```

从偏移量 offset 处 (该偏移量存在于文件描述符 fildes 上打开的文件内) 读取 nbyte 字节数, 将结果放入 buf 内。文件描述符的当前偏移量不受影响, 允

许多个 `pread` 与/或 `pwrite` 操作并行发生。

参考: 无

错误: **[EINVAL]** `offset` 是负值。

[EOVERFLOW] 试图读取的值超过最大限度。

[ENXIO] 请求超出设计能力。

[ESPIPE] 是管道文件。

提示: 允许高性能的并行 I/O。

pwrite

```
size_t pwrite (
    int                 fildes,
    const void         *buf,
    size_t              nbytes,
    off_t               offset);
```

从 `buf` 处将 `nbyte` 字节数写入偏移量 `offset` (该偏移量存在于文件描述符 `fildes` 上打开的文件内) 处。文件描述符的当前偏移量不受影响, 允许多个 `pread` 与/或 `pwrite` 操作并行发生。

参考: 无

错误: **[EINVAL]** `offset` 是负值。

[ESPIPE] 是管道文件。

提示: 允许高性能的并行 I/O。

10.1.6 取消点

大多数 UNIX 系统都支持非 POSIX 的基本接口函数。例如, `select` 和 `poll` 接口函数应该是推迟的取消点。因为这些函数不在 POSIX.1 内, 所以 Pthreads 不要求这些函数存在取消点。

然而, `select` 和 `poll` 函数与其他许多函数都存在于 X/Open 内。XSH5 标准包括一个扩展的取消点列表, 这些取消点遮盖了 X/Open 的接口。

XSH5 中必须是取消点的附加函数:

<code>getmsg</code>	<code>pread</code>	<code>sigpause</code>
<code>getpmsg</code>	<code>putmsg</code>	<code>usleep</code>
<code>lockf</code>	<code>putpmsg</code>	<code>wait3</code>
<code>msgrcv</code>	<code>pwrite</code>	<code>waitid</code>
<code>msgsnd</code>	<code>readv</code>	<code>writev</code>
<code>poll</code>	<code>select</code>	

XSH5 中可能是取消点的附加函数:

catclose	fsetpos	popen
catgets	ftello	pututxline
catopen	ftw	putw
closelog	fwprintf	putwc
dbm_close	fwscanf	putchar
dbm_delete	getgrent	readdir_r
dbm_fetch	getpwent	seekdir
dbm_nextkey	getutxent	semop
dbm_open	getutxid	setgrent
dbm_store	getutxline	setpwent
dlclose	getw	setutxent
dlopen	getwc	syslog
endgrent	getwchar	ungetwc
endpwent	iconv_close	vfprintf
endutxent	iconv_open	vfwprintf
fgetwc	ioctl	vprintf
fgetwc	mkstemp	vwprintf
fputwc	nftw	wprintf
fputwc	openlog	wscanf
fseeko	pclose	

10.2 POSIX 1003.1j

条件变量等待时钟:

```
int pthread_condattr_getclock (
    const pthread_condattr_t *attr,
    clockid_t *clock_id);
int pthread_condattr_setclock (
    pthread_condattr_t *attr,
    clockid_t clock_id);
```

Barriers 变量:

```
int barrier_attr_init (barrier_attr_t *attr);
int barrier_attr_destroy (barrier_attr_t *attr);
int barrier_attr_getpshared (
    const barrier_attr_t *attr, int *pshared);
int barrier_attr_setpshared (
    barrier_attr_t *attr, int pshared);
int barrier_init (barrier_t *barrier,
    const barrier_attr_t *attr, int count);
int barrier_destroy (barrier_t *barrier);
int barrier_wait (barrier_t *barrier);
```

读 / 写锁变量:

```
int rwlock_attr_init (rwlock_attr_t *attr);
int rwlock_attr_destroy (rwlock_attr_t *attr);
int rwlock_attr_getpshared (
    const rwlock_attr_t *attr, int *pshared);
int rwlock_attr_setpshared (
    rwlock_attr_t *attr, int pshared);
int rwlock_init (
    rwlock_t *lock, const rwlock_attr_t *attr);
int rwlock_destroy (rwlock_t *lock);
int rwlock_wlock (rwlock_t *lock);
int rwlock_timedwlock (rwlock_t *lock,
    const struct timespec *timeout);
int rwlock_trywlock (rwlock_t *lock);
int rwlock_wlock (rwlock_t *lock);
int rwlock_timedwlock (rwlock_t *lock,
    const struct timespec *timeout);
int rwlock_trywlock (rwlock_t *lock);
int rwlock_unlock (rwlock_t *lock);
```

自旋锁变量:

```
int spin_init (spinlock_t *lock);
int spin_destroy (spinlock_t *lock);
int spin_lock (spinlock_t *lock);
int spin_trylock (spinlock_t *lock);
int spin_unlock (spinlock_t *lock);
int pthread_spin_init (pthread_spinlock_t *lock);
int pthread_spin_destroy (pthread_spinlock_t *lock);
int pthread_spin_lock (pthread_spinlock_t *lock);
int pthread_spin_trylock (pthread_spinlock_t *lock);
int pthread_spin_unlock (pthread_spinlock_t *lock);
```

Thread 异常中止返回的变量:

```
int pthread_abort (pthread_t thread);
```

同一个 POSIX 工作小组（他们曾经开发出 POSIX.1b 和 Pthreads）为实时和线程编程开发了一套新的扩展。与线程（或与本书）相关的大多数扩展是由 POSIX 1003.14 标准小组提出的，为适合多处理器系统，该小组专门研究调整了现存的 POSIX 标准。

POSIX.1j 增加了一些线程同步机制，这些机制在多处理器和线程编程中普遍使用，但是原始的 Pthreads 标准遗漏了它们。Barriers 和自旋锁（spinlocks）有良好的并行粒度，例如，系统里从程序循环中自动生成并行操作代码。共享数据的算法中读 / 写锁非常有用，它允许多线程同时地读数据，但是只能有一个线程可以更

新数据。

10.2.1 Barrier

barrier 是一种同步形式，普遍应用于循环的并行分解。它们几乎只在多处理器系统中使用。一个 barrier 就是一组相关线程的“集合点”，每个线程都要在这里等候其他的线程，直到所有的线程都到达这个 barrier。当最后一个线程在 barrier 上等候时，释放所有参与等待的线程。

见书 7.1.1 部分，该部分讲述了 barrier 的详细资料，并且用一个例子展示如何用标准的 Pthreads 同步实现一个 barrier（注意本例中的行为与 POSIX.1J 所提议的有些不同）。

10.2.2 读/写锁

读/写锁允许一个线程专有地锁定某些共享的数据，并对该数据进行写或修改操作，允许多个线程为了读访问同时锁定这个数据。UNIX98 规定的“读 / 写锁”与 POSIX.1j 规定的非常相似。虽然 X/Open 预计两个规范机理上是相同的，但是为了避免冲突，在正式批准前，POSIX 标准应该给“读 / 写锁”换名^①。

如果你的代码依赖于一个经常被引用、但很少更新的数据结构，你应该考虑使用读/写锁访问数据，而不是使用互斥量。大多数线程能够读数据而不需等候；只有进程中的某个线程修改数据时，它们才会阻塞（同样，如果任何线程正在读数据时，一个请求写数据的线程也会被阻塞）。

见书 7.1.2 部分，该部分讲述了读/写锁的详细资料，并且用一个例子展示如何用标准的 Pthreads 同步实现一个读/写锁（注意本例中的行为与 POSIX.1J 所提议的有些不同）。

10.2.3 自旋锁

自旋锁与互斥量非常相似。由于 POSIX 仅仅规定了资源级的 API，POSIX.1j 几乎没有讲过它们与互斥量的区别，所以存在大量关于在自旋锁接口上标准化是否更有意义的争论。基本看法是在所给的硬件体系结构上，自旋锁是最简单和最快的、合适的同步机制。在某些系统中，自旋锁可能就是一条“test and set（测试并设置）”指令，在其他的系统中，它可能是一条真正的“load locked, test, store conditional, memory barrier（锁定、测试、保存条件音量，记住 barrier）”的指令序列。

关键的区别是当另一个线程已经拥有了自旋锁时，一个试图给自旋锁加锁的线程不会被阻塞。即这个线程会“螺旋般”地再试着迅速加锁，直到它成功地给该自旋锁加上锁为止（这是“疑点”之一，在单处理器上有更好的阻塞方式，或者让出

^① POSIX 工作组正在考虑以下的可能性：采用 XSH5 的读写锁定义，抛弃原有的 POSIX.1j 命名。不过，最后的决定还没有形成。

剩余的时间片什么也不做，或者没有时间限制一直拖下去）。

自旋锁主要用于细粒度并行操作，当这个代码被设计为只在多处理器上运行时，仅需很少的指令、小心地调整，就可以拥有自旋锁，最终获取的性能要比与其他进程共享系统资源重要得多。更有效的是，作为线程间的“上下文开关”，自旋锁尽可能长时间地不被锁定。如果这种情况发生了，你可以通过阻塞，允许其他线程进行有用的工作而获得更高的性能。

POSIX.1j 包含两个自旋锁函数：一个是 `spin_lock`，它允许自旋锁在进程间同步；另一个是 `pthread_spin_lock`，它允许自旋锁在一个进程里的线程间同步。注意了，这与互斥量、条件变量、读 / 写锁模式不同，它们使用同样的函数，`pshared` 的属性规定同步对象的结果能否在进程间共享。

自旋锁预计会非常快，对任何可能的上限都不成问题。事实上，在大多数系统上 `spin_lock` 和 `pthread_spin_lock` 的实现不同，但是标准允许它们不同。

10.2.4 条件变量等待时钟

Pthreads 条件变量只支持绝对时间片。线程设定等待到“Jan 1 00:00:00 GMT 2001”的可能性远远超过等待 1 小时 10 分钟。因为条件变量等待面临唤醒的问题，你无法控制（或者很难控制）各种各样的唤醒原因。当你被提前唤醒，（早于 1 小时 10 分钟）时，很难测定还剩下多少时间。然而，当你从绝对等候中被提前唤醒时，你的目标时间仍然是“Jan 1 00:00:00 GMT 2001”（关于提前唤醒的原因见本书 3.3.2 部分）。

不管多么优秀的理论，“relative time（相对时间）”等候还是有用的。它的一个重要的优势在于绝对系统时间可以被外界改变。绝对系统时间可能被修改成为正确或者不正确的时钟，或者它来自服务器的网上最近时间，或者被出于其他原因的数据调整过。相对时间等待和绝对时间等候都保持正确的调节器，但是相对时间等待的表示与绝对时间等待不同。即当你想要等待“1 小时 10 分钟”时，对你而言，你所能做到的就是增加时间间隔给当前的时钟，一直到要求的时钟到了为止。即使系统时间变了，系统也无法调整这个绝对的溢出时间。

POSIX.1j 正忙于发表 POSIX 时间服务中的基本、普遍的“cleanup”部分。POSIX.1j（基于 POSIX.1b 建立，POSIX.1b 主要提出了实时时钟函数和 `CLOCK_REALTIME` 时钟）提出了新的系统时钟，该时钟称为 `CLOCK_MONOTONIC`，这种新的时钟不是传统意义上的“相对时间”，它永远不会减少，也永远不会被系统的日期和时间修改，它以一个恒定的速率增长。一个“relative time”等待无非是从 `CLOCK_MONOTONIC` 时钟上取到当前的绝对数值，增加一些固定的偏移量（1 小时 10 分钟的等待时间是 4200 秒），并且等待到等待的时钟值到达为止。

增加条件变量的属性 `clock` 就可以达到目的。你可以在一个条件属性对象里用 `pthread_condattr_setclock` 函数设置 `clock` 属性，并且可以调用 `pthread_condattr_getclock` 函数请求当前的时钟值。假如大多数条件等候都

是有时间间隔的，那么默认值就是 CLOCK_MONOTONIC 变量。

尽管这种假设可能是错误的，它可能好像是一个与 Pthreads 性质相反的变化（某种程度上的确是这样），但 POSIX.1j 发现了时间条件等候函数碰到的难题，并且这个难题一直极为普遍地存在于 POSIX 标准的主要部分。时间通常定义得非常宽松，例如，一个时间条件等待并不需要精确地说出时间片变元是什么意思，只有在由 abstime 指定的绝对时间到达时（即系统时间等于或超过 abstime）才返回一个错误。这些语句目的清楚，但是没有明确的执行或使用的说明。根据有关 POSIX.1j 的基本原理，我们设想可以通过 clock_gettime (CLOCK_REALTIME, &now) 函数取得当前的时间。然而，POSIX 的“基本原理”并不比历史上的注释多多少，也不是正式标准的一部分。此外 clock_gettime 是可选的 POSIX.1b 子集_POSIX_TIMERS 的一部分，可以在许多支持线程的系统中存在。

POSIX.1j 试图让所有这些松散的限度合理化，至少让系统实现可能的 POSIX.1j 标准。当然，CLOCK_MONOTONIC 特性在它自己的选项下，加之与_POSIX_TIMERS 有关，因此不可能包治百病。缺少这样的选项，就没有时钟属性，没有办法去确信时间片行为——或者更加完全轻便的行为。

10.2.5 线程异常中断

pthread_abort 函数本质上是一个安全失败的取消函数。它用于需要使线程立即终止时。pthread_abort 危险的方面是该线程不运行清除处理程序或在这个线程之后没有任何其他的机会去清除。即，如果目标线程给一个互斥量加了锁，那么这个线程就会带着加锁的互斥量中止。因为不能从其他的线程中给互斥量开锁，这个应用必须准备完全放弃互斥量。进一步说，这就意味着任何等候这个被放弃的互斥量的线程会永远等下去，直到调用 pthread_abort 函数中止这些线程。

一般而言，真正的应用不可能从异常中断的线程中恢复过来，也永远不会使用 pthread_abort。然而，作为一种特定的应用类型，这种能力还是需要的。想像一下，一个实时的嵌入式控制系统不能关闭，并且在一些系统规则里要越过任何瞬时的错误，可靠地运行。一个线程可能遇到一个非常罕见的边界性条件错误，并且挂了起来，这个应用必须恢复。

在这样的系统里，由于实时的响应是非常关键的，所以所有的等候操作都使用时间片。一个线程检测到在合理的时间内某件事情没有发生，例如，一个航行的线程没有接到传感器的输入，它就会通知一个“错误管理器”，如果这个错误管理器不能判断出为什么警告传感器的线程没有响应，它就会试图去恢复。它可能尝试取消这个传感器线程，然后安全地关闭，但是如果在合理的时间内传感器线程不响应取消的信号，无论如何这个应用会继续下去。错误管理器于是会终止传感器线程，分析并且更正一些破坏的数据结构，如果需要还会创建一个新的传感器线程。

10.3 POSIX 1003.14

POSIX.14 是一个不同类型的标准，一个“POSIX 标准的子集”。不像 Pthreads 和 POSIX.1j，POSIX.14 没有给 POSIX 家族增加任何新的性能。但是作为替代，它试图给一些令人迷惑的选项提供一些命令，这些选项面对的是 POSIX 的实现者和用户。

对于多处理器硬件系统，POSIX.14 指定哪一种 POSIX 可选行为应该被认为是“必须的”。POSIX.14 也为不同的 POSIX 界限增加定义了最小值。POSIX.14 工作小组为额外的 POSIX 接口作出了建议，这些接口是基于基本的多重处理技术和 threading 成员的经验建立的。许多由 POSIX.14 发展的接口已经包含在 POSIX.1j 标准的草案中。

理论上，一旦 POSIX.14 成为一个标准，POSIX 实现的制作者可以要求与 POSIX.14 保持一致。并且那些希望发展多线程应用的人会发现寻找 POSIX.14 的一致性要比简单的 Pthreads 方便得多（因为当前使用 POSIX 标准子集的经验很少，剩下的就是看卖主和用户能否切实这样去做了）。

POSIX.14 工作组尽量从事于如下重要的问题：

- 为线程代码提供一种方法，这种方法用于判定活动的处理器数量。
- 为线程提供一种方法，这种方法用于在物理处理器上确定界限（bound）。
- 提供一个处理器管理（processor management）命令，用来控制系统中使用哪一个处理器。

虽然工作组意识到在所有的多处理器系统内这些性能普遍可用，但由于如下一些未解决的问题，从标准中撤销了它们。

- 如果你不能断定在任意时间有多少编码在运行，那么如何知道有多少处理器呢？别忘了，在你询问这件事的时候，信息也会改变。我们真正需要的是这样一个函数，它能够询问这样的问题“Would the current process benefit from creation of another thread?（创建另一个线程对当前的进程有益吗？）”，我们不知道如何回答这个问题。

或者说，为了让应用程序能回答这个问题，我们并不知道还需要提供的少信息才算足够（针对所有可能的体系结构）？

如何在一个多处理器体系中将线程绑定到一个处理器上？例如，在一个不对称的内存存取系统上，用一个统一的整数数组表示的处理器是令人误解和无用的，绑定一个线程给 processor 0 并且另一个相关的合作线程给 processor 1，这样在两个处理器上共享一个内存区域的效率还不如让它们通过一个相对较慢的通信接口高。

最后，一些标准机构（或者 POSIX）将会从事于这些问题的研究并且发展轻便的接口。致力于这些领域的人们可能会发现，他们需要限制这个标准在一个更小的范围内，而不是针对那些人们希望在其上使用线程的系统。

参考文献

[Anderson, 1991] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991.

一篇描述在操作系统中增加“两级调度”机制的论文。它是现代所有两级调度系统的鼻祖——大都要阅读它、参考它，并从它那儿获得灵感。

[Birrell, 1989] Andrew D. Birrell, *An Introduction to Programming with Threads*, SRC Research Report 35, Digital Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, January 1989. Available on Internet from <http://www.research.digital.com/SRC/publications/src-rr.html>

介绍了多线程编程的有关概念。尽管它特定于 Modula-2+ 和 SRC Taos 多线程操作系统，但是很多本质概念还是能很容易地在 Pthreads 标准中找到对应。

[Boykin, 1993] Joseph Boykin, David Kirschen, Alan Langerman, and Susan LoVerso, *Programming under Mach*, Addison-Wesley, Reading, MA, ISBN 0-201-52739-1, 1993.

[Custer, 1993] Helen Custer, *Inside Windows NT*, Microsoft Press, ISBN 1-55615-481-X, 1993.

[Digital, 1996] Digital Equipment Corporation, *Guide to DECthreads*, Digital Equipment Corporation, part number AA-Q2DPC-TK, 1996.

DEC 公司的 Pthreads 实现系统的参考手册。是旧的 CMA 和 DEC 线程 (POSIX 1003.4a 草案 4) 接口的附录 (将在 Digital UNIX 4.0 和 OpenVMS 7.0t 版本后删除)。

[Dijkstra, 1965] E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," *Communications of the ACM*, Vol. 8 (9), September 1956, pp. 569-570.

[Dijkstra, 1968a] E. W. Dijkstra, "Cooperating Sequential Processes," *Programming Languages*, edited by F. Genuys, Academic Press, New York, 1968, pp. 43-112.

[Dijkstra, 1968b] E. W. Dijkstra, "The Structure of the 'THE' -Multiprogramming

- System," *Communications of the ACM*, Vol. 11 (5), 1968, pp. 341-346.
- [Gallmeister, 1995]** Bill o. Gallmeister, *POSIX.4: Programming for the Real World*, O'Reilly, Sebastopol, CA, ISBN 1-56592-074-0, 1995.
- POSIX 1003.1b-1993 实时编程（基于标准的最后草案）。
- [Hoare, 1974]** C.A.R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, Vol. 17 (10), 1974, pp. 549-557.
- [IEEE, 1996]** 9945-1:1996 (ISO/IEC) [IEEE/ANSI Std 1003.1 1996 Edition] *Information Technology-Portable Operating System Interface (POSIX) -Part 1: System Application: Program Interface (API) [C Language]* (ANSI), IEEE Standards Press, ISBN 1-55937-573-6, 1996.
- POSIX C 语言编程接口，包括实时和线程。
- [Jones, 1991]** Michael B. Jones, "Bringing the C Libraries With Us into a Multi-Threaded Future," Winter 1991 *Usenix Conference Proceedings*, Dallas, TX, January 1991, pp. 81-91.
- [Kleiman, 1996]** Steve Kleiman, Devang Shah, and Bart Smaalders, *Programming with Threads*, Prentice Hall, Englewood Cliffs, NJ, ISBN 0-13-172389-8, 1996.
- 该书和本书的特点有些类似。例如，它们的作者都参与过 POSIX 标准工作组，并且都是各自公司的多线程实现的主要构建师。
- [Lea, 1997]** Doug Lea, *Concurrent Programming in Java™*, Addison-Wesley, Reading, MA, ISBN 0-201-69581-2, 1997.
- 从 Java 语言的角度给出了线程的不同视图，并提供了线程同步的独特构造。
- [Lewis, 1995]** Bil Lewis and Daniel J. Berg, *Threads Primer*, SunSoft Press, ISBN 0-13-443698-9, 1995.
- 对于多线程编程新手而言是一本很好的入门教材。第一版主要针对 Solaris 的 UI threads，同时也提供了 POSIX 线程的部分信息。
- [Lockhart, 1994]** Harold W. Lockhart, Jr., *OSF DCE, Guide to Developing Distributed Applications*, McGraw-Hill, New York, ISBN 0-07-911481-4, 1994.
- 关于 DCE 线程的一章描述了如何使用线程创建 DCE 应用程序。
- [McJones, 1989]** Paul F. McJones and Garret E. Swart, "Evolving the UNIX System Interface to Support Multithreaded Programs," SRC Research Report 21, Digital Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, September 1989. Available on Internet from <http://www.research.digital.com/>

SRC/publications/src-rr.html

关于 UNIX 系统对于多线程编程的调整报告。

[Schimmel, 1994] Curt Schimmel, *UNIX Systems for Modern Architectures*, Addison-Wesley, Reading, MA, ISBN 0-201-63338-8, 1994.

详细地描述了多处理器和共享内存系统的实现细节,如果本书的 3.4 节不能满足你对于这方面知识的渴望,就阅读该书吧。

因特网上的线程资源

*In the midst of the word he was trying to say,
In the midst of his laughter and glee,
He had softly and suddenly vanished away—
For the Snark was a Boojum, you see.
THE END*

—Lewis Carroll, *The Hunting of the Snark*

本节提供了一些进一步获得线程信息的网络资源。当然，由于网络总是在不断变化的，所以这些站点信息也不能永远正确。这就是信息时代的生活方式。

新闻组

comp.programming.threads

该新闻组包含了与线程相关的任何方面的广泛讨论。很多十分了解线程基本知识和 Pthreads 标准的各种实现的人们都是访问该新闻组的常客。这是一个很不错的问问题的地方，不管你是遇到了问题还是试图做什么事。请不要问关于屏保的问题！并且，如果你要问问题，请告诉我们你使用的系统硬件、操作系统及版本。

comp.unix.osf.osf1

这是一个特定于 Digital UNIX 系统的重要讨论组。当然，由于历史的原因导致了这个不直观的名字。如果要了解如何在 Digital UNIX 系统上使用线程，这是一个合适的地方。如

果问题不特定于 Digital UNIX 系统，则新闻组 comp.programming.threads 可能会更合适一些——因为它面向更广泛的线程专家和线程使用者。

Comp.unix.solaris

这是一个特定于 Solaris 系统的重要讨论组。如果要了解如何在 Solaris 系统上使用线程，这是一个合适的地方。如果问题不特定于 Solaris 系统，则新闻组 comp.programming.threads 可能会更合适一些——因为它面向更广泛的线程专家和线程使用者。

网站

<http://altavista.digital.com/>

Altavista 是由 DEC 公司开发的基于 Web 的多线程搜索引擎，也是一个帮你查找几乎任何事情的绝佳搜索引擎。它总是一个好的起点。

<http://www.aw.com/cp/butenhof/posix.html>

Addison-Wesley 的网页上包含了关于本书的信息，包括所有例程的源代码。

<http://www.best.com/~bos/threads-faq/>

该网页列出了 comp.programming.threads 新闻组上的常见问题 (FAQ)。在访问 comp.programming.threads 新闻组问问题之前，请先阅读该 FAQ，以避免再问那些先前已经被问了上百遍的问题。该网页上的信息同时也将定期地提交给新闻组。

<http://liinwww.ira.uka.de/bibliography/0s/threads.html>

由挪威的奥斯陆大学维护的线程相关术语的可搜索的参考书目。

<http://www.digital.com/>

DEC 公司的网站。该网站包含了许多关于 Digital UNIX 和 OpenVMS 操作系统的信息，包括线程和多处理器系统方面的信息。

<http://www.sun.com/>

SUN 公司网站，包括了许多关于 Solaris 操作系统的信息。你还能找到关于 Java 语言的信息，Java 语言通过将线程同步作为类方法是一个显式属性来支持线程的变种。

<http://www.sgi.com/>

Silicon Graphics 公司的网站，包含了 SGI 系统和 IRIX 操作系统的信息。

<http://www.netcom.com/~brownell/pthreads++.html>

该网页包含了试图“定义一种在 C++ 语言中使用线程的标准方式”的信息。