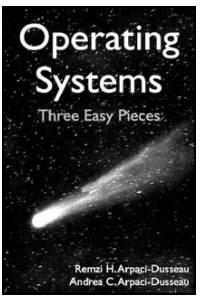
操作系统
Three Easy Pieces

Johnnie Xu　落 忧　译



武汉 · 常州

致读者

欢迎阅读本书！希望你们能像我们愉快地写作本书一样愉快地阅读本书。本书的名称——操作系统：three easy pieces，这个题目显然是向 Richard Feynman 撰写的伟大的有关物理学话题的讲义集致敬 [F96]。即使这本书肯定达不到这位著名物理学家高度，但也许它能够达到你理解操作系统（更一般的，系统）关键所在的需求。

这三个简单的部分是指本书围绕的三大主题：虚拟化，并发性，持久化。在讨论这些概念时，我们仅仅讨论操作系统所做的最重要的部分；但愿，你在这个过程中能寻找到乐趣。学习新事物多么的有趣，对吧？至少，我门觉得应当是的。

每个主题将分为多章来讲解，每一章会提出一个特定的问题以及讲解如何解决它。每章都比较短，但会尽可能去引用提出该思想的最原始的材料。我们写作本书的一个目的是使得历史的轨迹尽可能的清晰，因为我们认为这可以帮助学生更清晰地了解操作系统现在是什么样子的，过去是什么样子，将来会是什么样子。在此情况下，seeing how the sausage wasmade is nearly as important as understanding what the sausage is good for[1]。

有几个贯穿本书的 devices，这里有必要介绍一下。首先是 crux of the problem。任何在解决问题的时候，我会先阐述最重要的问题是什么；这个 device 会在正文中显眼的地方提出来，然后希望通过下文的技术、算法和思想来解决。

另外全书中还有很多加了背景色的 asides 和 tips。asides 偏向于讨论与正文相关但可能不太重要的知识点；tips 则倾向于可以应用在你的系统中的一般经验。为了方便查阅，在本书结尾的索引处列出了所有的 asides 和 tips 以及 crux。

纵贯本书，我们用”对话”这一古老的教学方式，作为呈现一些不同观点素材的方式【??】。这些对话用来引入各个主题概念，也用来偶尔回顾一下前面的内容。这些对话也是一个用幽默风格写作的机会。不知你们发现他们很有用或很幽默了没，好吧，这完全是另一码事儿。

在每个主要部分开始时，会先给出操作系统提供的抽象概念，接着的章节是提供该抽象概念所需要的机制、策略以及其他支持。这些抽象概念对于计算机科学各个方面都很重要，所以显然，它们对操作系统来说也是很必要的。

在这些章节中，我们会尽可能（实际上是所有例子）用真实可用的代码（而不是伪代码），你们可以自己编写并运行所有例子。在真实的系统上运行真实的代码是学习操作系统最好的方式，所以我们鼓励你们尽己所能的这么做。

在正文许多章节中，我们提供了少量的习题以确保你们可以理解正在学习的内容。许多习题是简单的模拟操作系统对应的部分。你们可以下载这些习题，运行并检测自己的学习状况。这些模拟器有一些特性：对于给定的不同的随机种子，你可以生成几乎无限的问题集；这些模拟器也可以告诉你如何解决这些问题。因此，你可以一测再测，直到比较好的理解了这些问题。

[1]提示：吃的！如果你是素食主义者，请远离。

这本书最重要的 addendum 是一系列的项目。在这些项目里，你可以学习真实系统的设计、实现和测试是如何进行的。所有的项目（包括上述的例子）是用 C 语言 [KR88] 编写的；C 是个简单而又强大的语言，绝大多数的操作系统都是以它为基础，因此很值得将他纳入你的语言工具集中。这些项目有两种形式可选（详见 the online appendix for ideas）。第一个是系统编程（systems programming）项目；这些项目对那些不熟悉 C 和 Unix，但又想学习如何进行底层编程的学生很适合。第二个是基于在 MIT 开发的真实的操作系统内核——xv6[CK+08]；这些项目对于那些已经有了一些 C 基础并想摸索进真正的 OS 里的学生很适合。在 Wisconsin，这个课程的实验以三种方式开展：全部系统编程，或全部 xv6 编程，或者两者皆有。

致教育工作者

如果你是希望使用本书的讲师或教授，随时欢迎。可能你已经注意到，本书是免费的，且可以从下面网页中获取：
http://www.ostep.org
你也可以从 lulu.com 网站购买纸质版。请在上述网页上查询。

本书的推荐引用格式（截止至目前）如下：
Operating Systems: Three Easy Pieces
Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau
Arpaci-Dusseau Books, Inc.
May, 2014 (Version 0.8)
http://www.ostep.org

这个课程分成一个学期 15 周比较好，这样可以覆盖书中的大部分话题，也能达到比较适中的深度。要是把本课程压缩到 10 周的话，可能需要每个部分都舍弃一些细节。还有一些关于虚拟机监控器的章节，通常压缩到虚拟化的结尾部分或者作为一个 aside 放在最后。

许多操作系统的书都会将并发性部分放在前面，本书不太一样，将其放在了虚拟化部分之后，这样在此之前可以对 CPU 和内存的虚拟化有一定理解。从我们 15 年教授此课程的经验来看，如果学生们没有理解地址空间是什么，进程是什么，或者上下文切换会在任意时间发生，他们都会有个困惑期——不知道并发性问题是如何引起的，为什么要试图解决这个问题。然而，一旦他们理解了虚拟化中的那些概念，当介绍线程和由线程引起的问题时会变得很简单，起码会简单些。

你可能注意到本书没有与之对应的幻灯片。主要原因是我们相信这个过时的教学方法：粉笔和黑板。因此，当我们在讲授这门课程时，我们脑子里带几个主要的思想和一些例子到课堂上，用板书的形式呈现给学生；讲义以及现场编写演示代码也是很有用的。在我们的经验里，使用太多的幻灯片会使得学生心思不在课堂上，因为他们知道反正资料就在这里，可以以后慢慢消化；板书的话会使得课堂像一个现场观看体验，因此会更有互动性、动态性，也会使学生们更好的享受你的课。

如果你想要一份我们上课的笔记的话，可以给我们发邮件。我们已经共享给了全世界很多老师。

最后一个请求：如果你使用网页上的免费章节，请仅仅链接到他们，而不是拷贝到本地。这会帮助我们追踪这些章节的使用情况（过去几年里已经有超过一百万的章节下载量），也可以确保学生们获得的是最新最好的版本。

致 学 生

如果你是正在阅读本书的学生，谢谢你！我们很荣幸能够为你提供一些材料帮助你求索关于操作系统的知识。我们俩都很怀念本科时候的一些教科书（如：Hennessy and Patterson [HP90]，计算机系统结构的一本经典书），当然也希望本书能够带给你一个美好回忆。

你也许注意到本书是免费的，在网上可以获取到。其主要原因是：教科书一般都比较贵。我们希望，本书能够成为免费书籍新浪潮的第一人，以帮助那些追求自己学业的人。无论他们来自世界的哪个地方或者他们愿意为一本书花多少钱。如若不然，这本免费书也聊胜于无。

可能的话，我们还希望给你们指出书中许多材料的原始出处：那些多年来塑造了操作系统领域的重要的论文和人物。思想不会凭空产生，他们来自于聪明又努力的人（包括许多图灵奖获得者[2]），因此我们应该尽可能的去纪念这些思想和人物。在纪念他们时，我们非常希望在写本书时能更好地理解已经发生的革命，而不是仿佛这些思想一直都存在似的 [K62]。再者，或许这些参考文献能够促进你自己去更深的挖掘；阅读著名的论文是这个领域最好的学习方法之一。

[2]图灵奖是计算机科学领域的最高奖；就像诺贝尔奖一样。

最后的话

Yeats【W. B. Yeats，威廉. 巴特勒. 叶慈】有句名言：”教育不是注满一桶水，而是点燃一把火。”他既是对的但同时也是错的[3]。你必须”注满那桶水”，这些笔记正好在此帮助你的学业；毕竟，当你去 Google 面试，他们问你一个关于如何使用信号量的刁钻问题，最好还是真的知道信号量是什么，对吧！

但是叶慈的重点明显是：教育真正的关键在于使你能对某样事务产生兴趣，去学习一些自己觉得更重要的东西，而不仅仅是某些课程中为了得高分而需消化的东西。Remzi 的父亲曾经说过：”走出课堂学习”。

我们撰写了这些笔记来激发你对操作系统的兴趣，从而去围绕这个话题自己做更多的阅读，去跟你的教授讨论这个领域正在进行的有意思的研究，甚至参与到这个研究中来。这是一个很伟大的领域，充满了精彩美妙思想，它们以深刻而重要的方式塑造了计算机历史。我们知道虽然这把火不能燃烧你们所有人，至少我们希望能够燃烧你们一部分人，甚至几个人。因为一旦这把火被点燃了，那就是你们真正可以做一些伟大事情的时候。教育过程的真正关键是：行动，去研究新的有趣的话题，去学习，去成长，还有最重要的是去寻找能够点燃你的事情。

JAndrea and Remzi
Married couple
Professors of Computer Science at the University of Wisconsin
Chief Lighters of Fires, hopefully[4]

[3] 如果他确实说了这句话；如许多名人名言一样，这句话的历史也不明

[4] 如果这个听起来我们如纵火犯一样在承认一些历史的话，那你可能抓错重点了。也许，如果这听起来俗气的话，那好，因为他就是这么俗气，但你门不得不为此原谅我们。

致谢

这一节包含了我们对那些帮助撰写本书的人的感谢。现在最重要的事情：你们的名字在这里找到！但是，你一定要帮忙。所以给我们发送一些反馈并帮助我调试本书吧。你可能会出名！或者至少你的名字会出现在本书中。

到目前为止，帮助过本书的人包括：Abhirami Senthilkumaran*, Adam Drescher* (WUSTL), Adam Eggum, Ahmed Fikri*, Ajaykrishna Raghavan, Akiel Khan, Alex Wyler, Anand Mundada, B. Brahmananda Reddy (Minnesota), Bala SubrahmanyamKambala, Benita Bose, BiswajitMazumder (Clemson), Bobby Jack, Bj ¨orn Lindberg, Brennan Payne, Brian Kroth, Cara Lauritzen, Charlotte Kissinger, Chien-Chung Shen (Delaware)*, Christoph Jaeger, Cody Hanson, Dan Soendergaard (U. Aarhus), David Hanle (Grinnell), Deepika Muthukumar, Dorian Arnold (NewMexico), DustinMetzler, Dustin Passofaro, Emily Jacobson, EmmettWitchel (Texas), Ernst Biersack (France), Finn Kuusisto*, Guilherme Baptista, Hamid Reza Ghasemi, Henry Abbey, Hrishikesh Amur, Huanchen Zhang*, Hugo Diaz, Jake Gillberg, James Perry (U.Michigan-Dearborn)*, Jan Reineke (Universit¨at des Saarlandes), Jay Lim, Jerod Weinman (Grinnell), Joel Sommers (Colgate), Jonathan Perry (MIT), Jun He, Karl Wallinger, Kartik Singhal, Kaushik Kannan, Kevin Liu*, Lei Tian (U.Nebraska-Lincoln), Leslie Schultz, LihaoWang, MarthaFerris,Masashi Kishikawa (Sony), Matt Reichoff, Matty Williams, Meng Huang, Mike Griepentrog, Ming Chen (Stonybrook),Mohammed Alali (Delaware),Murugan Kandaswamy, Natasha Eilbert, Nathan Dipiazza, Nathan Sullivan, Neeraj Badlani (N.C. State), Nelson Gomez, NghiaHuynh (Texas), Patricio Jara, Radford Smith, Riccardo-Mutschlechner, Ripudaman Singh, Ross Aiken, Ruslan Kiselev, Ryland Herrick, Samer AlKiswany, SandeepUmmadi (Minnesota), Satish Chebrolu (NetApp), Satyanarayana Shanmugam*, Seth Pollen, Sharad Punuganti, Shreevatsa R., Sivaraman Sivaraman*, Srinivasan Thirunarayanan*, Suriyhaprakhas BalaramSankari, SyJinCheah, Thomas Griebel, Tongxin Zheng, Tony Adkins, Torin Rudeen (Princeton), Tuo Wang, Varun Vats, Xiang Peng, Xu Di, Yue Zhuo (Texas A&M), Yufui Ren, Zef RosnBrick, Zuyu Zhang.

特别感谢 Joe Meehean 教授（Lynchburg）对每一章的详细注释，Jerod Weinman 教授（Grinnell）以及他整个班级不可思议的手册，Chien-Chung Shen 教授（Delaware）宝贵细致的阅读和意见，以及 Adam Drescher（WUSTL）的仔细阅读和建议。以及所有在资料细化中极大帮助过这些作者的所有人。

同时，也感谢这些年修过 537 课程的数百名学生。特别的，感谢促使这些笔记第一次成为书面形式的 08 年秋季班（他们苦于没有任何可阅读的教材——都是有进取心的学生!），并通过赞扬他们使我们能坚持下来（包括那年课程评价里令人捧腹的评价”ZOMG！典故你应该写一本全新的教材！"）。

当然也亏欠了几个勇敢参与了 xv6 项目的实验课程的几个人的感谢，其中许多现在已经纳入饿了 537 主课程中。09 年春季班：Justin Cherniak, Patrick Deline, Matt Czech, Tony Gregerson, Michael Griepentrog, Tyler Harter, Ryan Kroiss, Eric Radzikowski, Wesley Reardan, Rajiv Vaidyanathan, and Christopher Waclawik。09 年秋季班：Nick Bearson, Aaron Brown, Alex Bird, David Capel, Keith Gould, Tom Grim, Jeffrey Hugo, Brandon Johnson, John Kjell, Boyan Li, James Loethen,Will McCardell, Ryan Szaroletta, Simon Tso, and Ben Yule.10 年春季班：Patrick Blesi, Aidan Dennis-Oehling, Paras Doshi, Jake Friedman, Benjamin Frisch, Evan Hanson,

Pikkili Hemanth, Michael Jeung, Alex Langenfeld, Scott Rick, Mike Treffert, Garret Staus, Brennan Wall, Hans Werner, Soo-Young Yang, and Carlos Griffin (almost).

尽管我们的研究生学生没有直接帮助本书的撰写，但是他们教会了我们很多关于这个系统的东西。他们在 Wisconsin 时，我们经常与他们交流，他们做了所有的实际的工作。并且通过汇报他们做的事情，每周我们也学到了很多新的东西。这个名单包括我们收集到的目前和以前已经发表过论文的学生，星号标记表示那些在我们指导下获得了博士学位的学生：Abhishek Rajimwale, Ao Ma, Brian Forney, Chris Dragga, Deepak Ramamurthi, Florentina Popovici*, Haryadi S. Gunawi*, James Nugent, John Bent*, Lanyue Lu, Lakshmi Bairavasundaram*, Laxman Visampalli, Leo Arulraj, Meenali Rungta,Muthian Sivathanu*, Nathan Burnett*, Nitin Agrawal*, Sriram Subramanian*, Stephen Todd Jones*, Suli Yang, Swaminathan Sundararaman*, Swetha Krishnan, Thanh Do, Thanumalayan S. Pillai, Timothy Denehy*, Tyler Harter, Venkat Venkataramani, Vijay Chidambaram, Vijayan Prabhakaran*, Yiying Zhang*, Yupu Zhang*, Zev Weiss.

最后还欠 Aaron Brown 一个感激之情，他多年前第一个带了这门课（09 年春），然后有带了 xv6 实验课（09 年秋），最后担任了本课程的研究生助教两年左右（10 年秋至 12 年春）。他的辛勤工作极大的促进了项目的进展（特别是基于 xv6 的），并因此帮助了 Wisconsin 的无数的本科生和研究生优化可学习体验。正如 Aaron 想说的（他一贯的简洁风格）：”Thx."

目　　录

Chapter 1

并发性简介

到目前为止，我们已经知道了操作系统基本执行部件的抽象概念的发展。我们已经了解如何『使用』单个物理 CPU 并将其抽象成多个虚拟 CPU，因此多个程序看似同时运行成为了可能【注：在单个单核 CPU 上，同一时刻实际上只有一个程序在运行】。我们也学习了如何为每个进程构建一个看似很大且私有的虚拟内存；地址空间的抽象概念让每一个程序都好像有了自己的内存，而实际上是操作系统暗地里跨物理内存复用了地址空间（有时是磁盘）。

本节，我们将为单个正在运行的进程引入一个新的概念：线程。不像先前一个程序内只有一个执行点的视角（只从一个 PC 寄存器取指令和执行指令），一个多线程程序不止一个执行点（多个 PC 寄存器，可以从每一个 PC 取指令和执行指令）。从另一个角度来思考这个问题，每一个线程很像一个独立的进程，但是与进程的区别是：进程内的各个线程共享相同的地址空间。所以各个线程可以访问相同的数据。

单个线程的状态与进程的状态的很类似，有一个程序计数器（PC）来跟踪程序从何处取指令。每个线程有自己的用于计算的寄存器集；因此，如果在单个处理器上运行着两个线程，当从一个线程（T1）切换到另一个线程（T2）时，就会发生一次上下文切换。线程的上下文切换跟进程的上下文切换很类似，因为需要保存 T1 的寄存器状态，并在 T2 运行之前加载 T2 的寄存器状态。对于进程来说，我们将状态保存到进程控制块（process control block，PCB）；对于线程来说，我们需要一个或多个线程控制块（thread control blocks，TCBs）来存储单个进程中的多个线程的状态。尽管如此，线程间的上下文切换比起进程来，有一个重要区别：地址空间依旧保持不变（没有必要切换正在使用的页表）。

线程与进程之间的另一个重要的区别是栈（stack）。在经典进程（现在可以称之为单线程进程，single-threaded process）地址空间的简单模型中，有一个单独的栈，通常位于地址空间的的底端（见下左图）。

然而，在多线程进程中，每个线程独立运行，当然会调用各种各样的 routines 来完成它正在做的任何工作。跟地址空间中只有一个栈不同，多线程进程中每个线程都有一个栈。假设一个多线程进程内有两个线程，其相应的地址空间与经典进程不一样（见上右图）。

在 Figure26.1 中，可以看到两个栈分布在进程的" 整个" 地址空间中。因此，

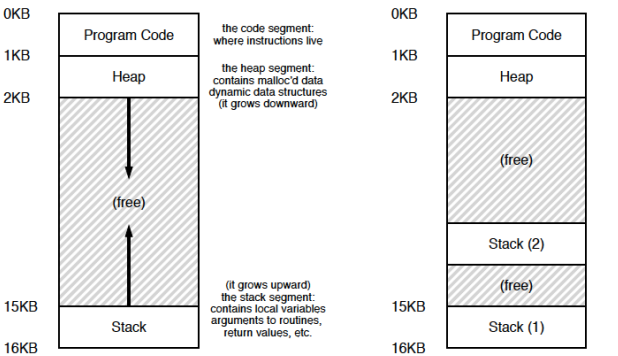


Figure 26.1: **A Single- and Multi-Threaded Address Space**

Figure 1.1: 图 26.1：单线程和多线程的地址空间

任何在栈中分配的变量、参数、返回值以及其他存放在栈中的数据将会存储在那个有时称作线程局部空间的地方，即相应线程的栈。

也许你也注意到了新的布局破坏了原来" 美观" 的地址空间布局。之前，堆栈可以各自独立增长而不出问题，除非超出了地址空间的范围。而现在的地址空间布局就不再有先前的"nice situation"。幸运的是，这样的空间布局通常是可行的，因为栈一般不需要特别大（程序大量使用递归时除外）。

1.1 例子：线程创建

假设我们想运行一个创建两个线程的程序，每个线程各自执行相互独立的任务，打印「A」或「B」。代码如 Figure 26.2 所示。

主程序创建两个线程，每个都执行函数 mythread()，但是传递不同的参数（字符串『A』或『B』）。一旦创建了线程，它可能会立即运行（取决于调度器的whims）；也可能会进入『就绪』态而非『运行』态，因此不立即执行。创建两个线程之后（T1 和 T2），主线程调用 pthread_join() 等待对应的线程结束。

我们来看一下这个小程序可能的执行顺序，在执行示意图中（表 26.1），时间从上向下依次增长，每一栏显示了什么时候运行不同的线程（主线程、线程 T1、或线程 T2）。

然而，注意这个顺序并不是唯一的执行顺序。实际上，对于一个给定的指令序列，它有不少的可能执行顺序，这取决于在特定的时刻调度器决定哪个线程能得到执行。比如，一旦创建了一个线程，它可能立即执行，如表 26.2 所示的执行顺序。

我们也可以看到『B』在『A』之前打印出来，这就是说调度器决定先执行线程 T2，即使线程 T1 更早创建；没有任何理由取假设先创建的线程就先运行。表 26.3 显示了这个执行顺序，线程 T2*******with Thread 2 getting to strut its stuff before Thread 1.

```
1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4
5  void *mythread(void *arg) {
6      printf("%s\n", (char *) arg);
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     pthread_t p1, p2;
13     int rc;
14     printf("main: begin\n");
15     rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
16     rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
17     // join waits for the threads to finish
18     rc = pthread_join(p1, NULL); assert(rc == 0);
19     rc = pthread_join(p2, NULL); assert(rc == 0);
20     printf("main: end\n");
21     return 0;
22 }
```

Figure 1.2: 简单的线程创建代码

As you might be able to see, one way to think about thread creation is that it is a bit like making a function call; however, instead of first executing the function and then returning to the caller, the system instead creates a new thread of execution for the routine that is being called, and it runs independently of the caller, perhaps before returning from the create, but perhaps much later.

As you also might be able to tell from this example, threads make life complicated: it is already hard to tell what will run when! Computers are hard enough to understand without concurrency. Unfortunately, with concurrency, it gets worse. Much worse.

main	Thread 1	Thread 2
starts running		
prints "main:begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
		runs
		prints "B"
		returns
prints "main:end"		

Table 1.1: 线程执行轨迹 (1)

main	Thread 1	Thread 2
starts running		
prints "main:begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main:end"		

Table 1.2: 线程执行轨迹 (2)

main	Thread 1	Thread 2
starts running		
prints "main:begin"		
creates Thread 1		
creates Thread 2		
	runs	
	prints "A"	
	returns	
waits for T1		
		runs
		prints "B"
		returns
waits for T2		
returns immediately; T2 is done		
prints "main:end"		

Table 1.3: 线程执行轨迹 (3)

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include "mythreads.h"
4
5  static volatile int counter = 0;
6
7  /* Simply adds 1 to counter repeatedly, in a loop. No, this is not how
8   * you would add 10,000,000 to a counter, but it shows the problem nicely. */
9
10 void *mythread(void *arg) {
11     printf("%s: begin\n", (char *) arg);
12     int i;
13     for (i = 0; i < 1e7; i++) {
14         counter = counter + 1;
15     }
16     printf("%s: done\n", (char *) arg);
17     return NULL;
18 }
19
20 /* Just launches two threads (pthread_create) and then waits for them (pthread_join)
21  */
22 int main(int argc, char *argv[]) {
23     pthread_t p1, p2;
24     printf("main: begin (counter = %d)\n", counter);
25     Pthread_create(&p1, NULL, mythread, "A");
26     Pthread_create(&p2, NULL, mythread, "B");
27
28     // join waits for the threads to finish
29     Pthread_join(p1, NULL);
30     Pthread_join(p2, NULL);
31     printf("main: done with both (counter = %d)\n", counter);
32     return 0;
33 }
```

1.2 为何更糟糕：共享数据

上一节简单的线程示例可以有效的解释线程是如何被创建，以及它们是如何按照不同的顺序运行，这个执行顺序决定于调度器决定如何运行它们。这个示例并没有展示线程间在访问共享数据时是如何交互的。

让我们设想一个简单的例子：有两个线程想要更新一个全局共享变量。我们将要研究的代码见 Figure 26.3。

有几个关于这段代码的注释。第一，如 Stevens 的建议【SR05】，我们封装了线程的 create 和 join 例程与其失败退出判断【***】，对于一个简单如此的程序，我们至少要关注发生的错误（如果发生的话），但是不做任何很【smart】的事儿（如：仅仅退出）。因,Pthread_create() 简单的调用 pthread_create() 并确定返回值为

0；如果不是，Pthread_create() 仅仅打印一个消息并退出。

第二，这个例子为工作线程仅用一个函数而不是两个独立的函数（即两个线程执行同一个函数，译者注），向线程传递一个参数（这里是字符串）让每一个线程在其消息之前打印不同的字母。

最后也是最重要的，我们可以看每个工作线程试图做的事：共享变量 counter 加 1，这个操作在循环中执行一千万次。因此，理想的最终结果是：20,000,000。

现在编译执行这个程序，看一下它的结果。有时候，everything works how we might expect：

```
prompt> gcc -o main main.c -Wall -pthread
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

不幸的是，当我们运行这段代码时，即使是在一个处理器上，我们也得不到那个预期的结果。有时，结果如下：

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

我们再试一次，看看是不是我们疯狂了。毕竟，如你所接受的教育，计算机并不认为会产生确定性结果。也许你的处理器欺骗了你？（等我喘口气）

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

不仅每一个都是错误的结果，而且每个结果还不一样！大大的疑问：为什么会发生这样的事儿呢？

技巧：了解并使用你的工具

你总是需要学习心得工具来帮助你编写、调试和理解计算机系统。这里我们使用一个小巧的工具：反汇编器（disassembler）。当你对一个可执行文件执行反汇编时，它会显示这个可执行文件是有哪些会变代码组成的。例如，如果你希望理解例子中更新 counter 的底层代码，运行 objdump（Linux）来它的汇编代码：

```
prompt> objdump -d main
```

1.3 核心问题：失控的调度

为了理解为何会发生这样的事儿，我们需要理解编译器为更新 counter 而生成的指令序列。在这个例子里，我们希望简简单单的将 counter 加 1。因此，做这一操作的指令序列也许应该如下（x86）：

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

这个例子假设变量 counter 分配的地址是 0x8049a1c。在这个三指令的序列里，x86 的 mov 指令首先取得该地址的内存值并将其放入寄存器 eax 中，然后，执行 add 操作，eax 寄存器的值加 1，最后，eax 的值写回原先的内存地址。

我们设想一下两个线程中的一个（线程 T1）进入这段代码，它对 counter 执行了加 1 操作。它首先加载 counter 的值（假设其初值是 50）进入寄存器 eax，因此对线程 T1 来说寄存器 eax 的值为 50。然后它对寄存器执行加 1 操作，此时 eax 的值是 51。现在，发生了件不幸的事儿：定时器中断到达；因此，操作系统保存当前运行线程的状态（PC，eax 等寄存器）至线程的 TCB。

现在发生了更糟糕的事儿：线程 T2 被调度执行，它进入同一段代码。它也执行第一条指令，获取 counter 的值并写入它的 eax 寄存器（注：每个线程在运行时都有自己私有的寄存器；这些寄存器是通过上下文切换代码保存、加载虚拟化出来）。counter 的值此时仍然是 50，因此线程 T2 的 eax 值为 50。假设线程 T2 继续执行接下来的两条指令，eax 加 1（eax=51），然后保存 eax 的内容至 counter（内存地址 0x8049a1c）。因此，全局变量 counter 此时的值是 51。

最后，又一次发生上下文切换，并且线程 T1 得到了执行。记得刚才仅仅执行过了 mov 和 add 指令，那么此时应当执行最后一条 mov 指令。刚才 eax 的值是 51，因此，最后执行 mov 指令并将值写入内存；counter 的又一次被写为 51。

简单的说，事儿是这样的：counter 加 1 的代码执行了两次，但是初值为 50 的 counter 现在只有 51。但是这个程序『正确的』结果应当是变量 counter 值为 52。

我们来看一下详细的执行路径以便更好的理解这个问题。对于这个例子，假设上述的代码加载到内存地址 100 处，如下图所示的指令序列（注意那些曾经优秀的精简指令集：x86 有变长指令；这里的 mov 指令占 5 字节的内存，add 指令仅占 3 字节）：

```
100 mov 0x8049a1c, %eax
105 add $0x1, %eax
108 mov %eax, 0x8049a1c
```

基于这些假设，上述发生的事儿如表 26.4 所示。假设 counter 起始值是 50，然后跟踪这个例子确保你可以理解正在发生什么。

				after instruction	
OS	**Thread 1**	**Thread 2**	**PC**	**%eax**	**count**
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	50
	mov %eax, 0x8049a1c		113	51	51

Table 1.4: 问题：up close and personal

上面已经示范的问题称作：竞争条件，其结果取决于这段代码的执行时机。有时候运气不好（例如：在执行的时候发生不合时宜的上下文切换），就会得到错误的结果。实际上，我们很可能每次都得到不同的值；因此，而不是确定性计算（曾经来源于计算机??），我们称之为不确定性，不知道输出的会是什么，且这些输出在交叉运行之间很可能会不同。

因为多线程执行这段代码会引起竞争条件，我们称这段代码为：临界区。一个临界区是一段访问共享变量（更一般的；共享资源）且不可被多于一个线程并发执行的代码段。

对于这段代码，我们需要的是称之为互斥量的东西。这个性质保证了如果有一个线程正在执行临界区代码，其他的线程都会被阻止进入临界区。

顺便提一下，实际上所有这些概念都是由 Edsger Dijkstra 创造出来的。他是这个领域的开拓者，并凭借这个工作和其他成果获得了图灵奖；详见他 1968 年的论文『Cooperating Sequential Processes』[D68] 中对此问题惊人清晰的描述。我们还会在本章中多次看见 Dijkstra。

1.4 原子性的愿望

解决这个问题的一个办法是更强大的指令，这些指令可以在单步之内做任何我们需要做的，因此避免了发生不合时宜的中断的可能性。例如，假如我们有一个像下面所示一样的超级指令，会怎样呢？

```
memory-add 0x8049a1c, $0x1
```

假设这条指令将一个值加到该内存地址，并硬件保证它是原子执行的；当这条指令执行后，它将按照预想的那样执行更新操作。它不会在指令执行中被中断，因为正是我们从硬件那儿得到的保证：当中断发生时，这条指令要么没有执行，要么已经执行结束；没有中间状态。硬件可以如此的美好，不是么？

在此文中，原子性意味着『作为一个单位』，有时称之为『全或无』。我们想原子地执行这三条指令序列：

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

如之前所说的，如果有一条单独指令能够完成这个操作，那么只需要发出指令即可完成。但是通常情况下，没有这样的指令。设想我们已经构建了一个并发的 B 树，此时想要更新它；难道要硬件支持『B 树的原子更新』指令么？恐怕不可能，至少在合理的指令集中是如此。

因此，相反的，由硬件提供少量有效的指令，我们可以借助这些指令构建一系列通用的同步原语。通过这些硬件同步原语与操作系统的支持，我们可以构造出以同步可控方式访问临界区的多线程代码，因此可以可靠的生成正确的结果，尽管存在并发执行的挑战。相当的棒，是不？

这是本节将要研究的问题。这是一个奇妙也很难的问题，应该会让你（有点）头疼。如果没有头疼，那就是你没懂！继续研究直到你头疼为止，那时候你就知道自己已经走向了正确的方向。到了那个时候，休息一会儿，我们可不想让你头疼的厉害。

症结所在：如何为同步提供支持

为了构建有效的同步原语，我恩需要硬件提供什么支持呢？又需要操作系统提供什么支持呢？我们怎么才能正确高效的构建这些原语呢？程序又如何使用他们以得到期望的结果呢？

关键并发术语
临界区，竞争条件，不确定性，互斥

这四个术语对于并发代码太核心了，以至于我认为非常值得把它们提出来说一下。详见 Dijkstra 早期的工作 [D65,D68]。

- 临界区，临界区是一段访问共享资源的代码段，共享资源一般是一个变量或数据结构
- 竞争条件，竞争条件发生在正在执行的多个线程几乎同时进入临界区；多个线程都试图更新共享数据结构，同时导致产生奇怪（可能非预期）的结果。
- 不确定性，一段不确定的程序有一个或多个竞争条件组成，一次一次执行的输出变化不一，视不同线程何时运行而定。因此结果是不确定的，通常我们指望着计算机系统。
- 互斥，为了避免这个这些问题，线程需要使用某种互斥原语；以此保证只有一个线程已经进入临界区，从而避免竞争得到确定性的结果。

1.5 另一个问题：等待其他线程

本章所设定的并发问题，线程间貌似只有一种交互形式，即访问共享变量以及临界区原子性的支持。事实证明，还会产生另一种常见的交互，即一个线程必须等待其他线程完成某些操作方能继续执行。例如，当一个进程执行磁盘 I/O 时而休眠时，这种交互就会产生；当磁盘 I/O 完成时，进程需要从休眠中唤醒以便继续执行。因此，在接下来的章节中，我们不仅仅会研究如何构建同步原语为原子性提供支持，还会研究相应的机制来为多线程程序中常见的休眠唤醒交互方式提供支持。要是现在这东西没有意义，那好！当你读到条件变量那章时就会觉得有意义了。如果那时觉得不太行，那你应该一遍再一遍的读那章直到觉得有意义。

1.6 小结：为什么在 OS 级

在 wrapping up 之前，你可能有一个疑问：为什么我们要在 OS 这一层研究这些？答案就一个词——历史；操作系统是第一个并发程序，许多技术都被创造用于 OS 中。后来，在多线程进程中，应用程序编写者也不得不考虑这些事儿。

例如，设想这样的场景，有两个正在运行的进程。假如它们都调用 write() 来写一个文件，都想将数据附加到文件中（例如：添加数据到文件末尾，从而增加它的长度）。这样的话，两者都需要分配一个新的块，记录块的位置到文件的 inode 中，并改变文件大小以示一个更大的大小（我们将会在本书第三部分学到更多关于文件的内容）。因为中断可能随时都会发生，更新这些共享数据结构的代码就是临界区，因此，从最开始的中断介绍可知，操作系统设计者不得不担心操作系统怎么更新这些内部结构。一个不合时宜的中断引起了上述的所有问题。

不奇怪，页表，进程列表，文件系统结构等。实际上，所有内核数据结构都需要通过合适的同步原语来谨慎访问，以便正确工作。

技巧：使用原子操作

原子操作是构建计算机系统最强大的底层技术之一，从计算机体系结构到并发代码（本节正在研究的）、文件系统（后面将会研究的）、数据库管理系统，甚至分布式系统 [L+93]。使得一系列行为原子化背后的思想可以简单的缩成一个短语——全或无。你希望一起执行的行为要么全都发生了，要么全都没有发生，而没有中间状态。有时，将许多行为组织成一个原子行为称作：事务，这个思想在数据库和事务处理领域已经发展很成熟 [GR92]。在探索并发性的主题中，我们会使用同步原语把短指令序列转化成原子执行块。但是如我们将会见到的，原子性的思想远不止这些。例如，文件系统使用诸如日志或 copy-on-write 等技术来原子地转移它们的磁盘状态，使得在面对系统故障时能严格正确地执行。If that doesn’t make sense, don’t worry—it will, in some future chapter。

Chapter 2

插曲：线程 API

本章简要讲解线程 API 的主要内容。每个部分将会按照介绍如何使用这些 API 的顺序，在后续章节进行详细解释。更多的细节可以在许多书和在线资源里找到 [B97，B+96，K+96]。需要注意，由于有很多的例子，接下来锁和条件变量概念的章节讲起来会很慢；因此，本章用作一个引用会更好些。

关键：如何创建并控制线程

操作系统应该为线程创建和控制提供什么接口？应该如何设计这些接口，使之跟一般程序接口一样使用自如？

2.1 线程创建

编写多线程程序，首先要做的就是创建新线程；因此，需要存在某种线程创建接口。在 POSIX 中，很简单：

```
#include <pthread.h>
int pthread_create( pthread_t * thread, const pthread_attr_t * attr,
                    void * (*start_routine)(void*), void * arg);
```

这个声明貌似有点小复杂（特别是当你没有用过 C 语言的函数指针），其实这并不糟糕。这里有四个参数：thread，attr，start_routine，arg。第一个参数 thread 是一个指向 pthread_t 结构体的指针；我们将会用这个结构体与起对应的线程进行交互，因此我们需要将它传给 pthread_create() 函数初始化。

第二个参数 attr，用来指定一些这个线程可能需要的一些属性。比如设置栈大小，或者这个线程的调度优先级的信息等。可以单独调用 pthread_attr_init() 函数来初始化一个属性；欲知详情，请查阅对应手册。然而，许多情况下，默认属性就可以了，这样的话，给该参数传递一个 NULL 即可。

第三个参数是最复杂的，实际上它只是想知道：这个线程从哪个函数开始运行？在 C 语言中，称之为函数指针。这个函数指针告诉我们一下信息：函数名

（start_routine），传递给该函数一个 void * 类型的参数（如 start_routine 括号之后表示的），以及该函数一个类型为 void * 的返回值（例如：空指针）。

如果这个函数指针想要的不是空指针，而是一个整型参数，其声明如下：

```
int pthread_create(..., // 前两个参数一样
                   void * (*start_routine)(int), int arg);
```

如果函数指针希望传一个空指针参数，但是要返回一个整型值，则其声明如下：

```
int pthread_create(..., // 前两个参数一样
                   int (*start_routine)(void *), void * arg);
```

最后，第四个参数 arg 实际上就是传递个函数指针的参数。你可能会有疑问：为什么需要这些空指针呢？其实答案很简单：start_routine 函数有了一个空指针作为参数，就可以传递任意类型的参数给它了；有一个空指针的返回值就允许线程返回任意类型的结果。

```
1  #include <pthread.h>
2
3  typedef struct __myarg_t {
4      int a;
5      int b;
6  }myarg_t;
7
8  void *mythread(void *arg) {
9      myarg_t *m = (myarg_t *) arg;
10     printf("%d %d\n", m->a, m->b);
11     return NULL;
12 }
13
14 int main(int argc, char *argv[]) {
15     pthread_t p;
16     int rc;
17     myarg_t args;
18     args.a = 10;
19     args.b = 20;
20     rc = pthread_create(&p, NULL, mythread, &args);
21     ...
22 }
```

Figure 2.1: 创建线程

我们来看图 27.1 的例子。仅创建了一个传递了两个参数的线程，这两个参数组成自己定义的类型（myarg_t）。一旦创建后，这个线程就可以简单地将它的参数转化成想要的类型，并可按预期得到这些参数。

一旦创建了线程，就会存在一个新的存活的执行实体，与它自己的调用栈一起完成，跟程序中所有存在的线程一样运行在相同的地址空间。快乐之旅由此开始！

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <assert.h>
4  #include <stdlib.h>
5
6  typedef struct __myarg_t {
7      int a;
8      int b;
9  } myarg_t;
10 typedef struct __myret_t {
11     int x;
12     int y;
13 } myret_t;
14
15 void *mythread(void *arg) {
16     myarg_t *m = (myarg_t *) arg;
17     printf("%d %d\n", m->a, m->b);
18     myret_t *r = Malloc(sizeof(myret_t));
19     r->x = 1;
20     r->y = 2;
21     return (void *) r;
22 }
23
24 int main(int argc, char *argv[]) {
25     int rc;
26     pthread_t p;
27     myret_t *m;
28
29     myarg_t args;
30     args.a = 10;
31     args.b = 20;
32     Pthread_create(&p, NULL, mythread, &args);
33     Pthread_join(p, (void **) &m);
34     printf("returned %d %d\n", m->x, m->y);
35     return 0;
36 }
```

Figure 2.2: 等待线程完成

2.2 线程完成（completion）

之前的例子展示了如何创建一个线程。然而，如果想要等待一个线程的完成会发生什么呢？你需要做一些特别的事儿来等待线程完成；特别的，你需要调用的 pthread_join() 函数。

```
int pthread_join(pthread_t thread, void **value_ptr);
```

这个函数只有两个参数，第一个参数是 pthread_t 类型，用来指定要等待哪个线程。实际上就是在创建线程时传递给线程库的那个值；如果持有这个值，那么现在就可以用来等待那个线程的运行终止。

第二个参数是一个指向你想要取回的返回值的指针。因为这个函数可以返回任意值，故其定义成了一个空指针；由于 pthread_join 会改变传给它的参数，故应该传递该值的指针，而不是值本身。

我们来看另外一个例子（图 27.2）。在这段代码里，还是创建了一个线程，并通过 myarg_t 结构体传递了一对参数。返回值用到了 myret_t 类型。一旦这个线程结束运行，正等在 pthread_join() 的主线程就会返回，并且可以访问从线程返回的值，即 myret_t 里的内容。

这个例子有几个注意点。第一，大多数时候我们不需要做所有这些痛苦的 packing and unpacking of arguments 的事儿。例如，如果我们仅需创建一个无参的线程，传递一个 NULL 作为参数即可。类似的，如果我们不关心返回值的话，可以给 pthread_join() 传一个 NULL。第二，如果我们仅传一个值的话（如 int），就不需要 package 它作为一个参数。图 27.3 就是一个例子。这种情况下，由于我们不需要将参数和返回值 package 成结构体，就简单了很多。

```
1  void *mythread(void *arg) {
2      int m = (int) arg;
3      printf("%d\n", m);
4      return (void *) (arg + 1);
5  }
6
7  int main(int argc, char *argv[]) {
8      pthread_t p;
9      int rc, m;
10     Pthread_create(&p, NULL, mythread, (void *) 100);
11     Pthread_join(p, (void **) &m);
12     printf("returned %d\n", m);
13     return 0;
14 }
```

Figure 2.3: 等待线程完成

第三，需要非常注意返回值是如何从线程返回的。尤其，不要返回一个指向了线程调用栈中分配的值的指针。如果这么做了，你觉得会发生什么呢？（考虑一下）这里是一段问题代码的例子，改自图 27.2 中的例子。

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t r; // 分配在栈中: BAD!
    r.x = 1;
    r.y = 2;
    return (void *) &r;
```

```
}
```

在这种情况下，变量 r 是在 mythread 的栈中分配的。然而，当线程返回时，这个值会自动释放（这就是为什么栈使用起来这么简单）。因此，传回去的指针就是一个未分配的变量，可能会导致各种错误结果。当然，当你输出你觉得你返回了的值时，你很可能（不一定）会感到奇怪。可以自己试试看！【2. 幸运的是当你这样的代码时，编译器 gcc 可能会抱怨的，这也是一个注意编译器警告的原因】

最后，你可能注意到使用 pthread_create() 创建线程之后，紧接着就调用了 pthread_join()，这种创建线程的方式很奇怪。实际上，有一种更容易的方式来完成这样的任务，叫做过程调用。通常我们会创建不止一个线程并等待它完成，否则完全没有使用多线程的意义。

我们应该注意到并不是所有的多线程代码都使用 join 函数。例如，一个多线程 web 服务器可能创建多个工作线程（worker），然后使用主线程 accept 请求并将之传给工作线程（worker）。因此，这样长生存期的程序也许就不需要 join。然而，一个创建多个线程去并行执行特定任务并行程序很可能会用 join 来确保所有的工作都完成，才退出或进入下一阶段工作。

2.3 锁

除了线程创建（creation）和合并（join），也许，POSIX 线程库提供的最有用的一些列函数是通过锁（locks）来为临界区提供互斥量的函数。这对最基础的函数由下列两个函数提供：

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

这两个函数应该很容易理解和使用。当你实现的一段代码是临界区时，需要用锁来保护以使按照预期执行。你大概可以想象一下，代码应该如下：

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

这段代码的意思是这样的：如果在调用 pthread_mutex_lock() 时没有其他的线程持有这个锁，这个线程就取获取这个锁，并进入临界区。如果其他线程已经持有这个锁，则这个试图获取锁的线程会阻塞直到获取到锁（意味着那个持有锁的线程已经调用 unlock 释放了这个锁）。当然，有可能在一个给定的时间会有很多线程卡在线程获取函数；只有那个获取到锁的调用 unlock。【*****】

可惜，这段代码在两个地方有问题。第一个问题是：未有合适的初始化。所有的锁都必须被正确地初始化，以保证它们都有正确的初始值，以及在调用 lock 和 unlock 时能够正常工作。

在 POSIX 线程中,有两种初始化锁的方式。一种是用 PTHREAD_MUTEX_INITIALIZER 来初始化，如下：

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

这种方式将锁设置为默认值并设为可用。另一种动态方式是调用 pthread_mutex_init()，如下：

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

第一个参数是锁的地址，而第二个参数是一个可选的属性集。关于该属性集可以自己查阅；简单地传递 NULL 则是使用默认方式。上述的两种方式都可以，但我们通常使用动态的方法（后者）。注意，当锁使用结束时，相应的要调用 pthread_mutex_destroy()。可参见相应的手册获取这部分的所有细节。

第二个问题是没有检测 lock 和 unlock 调用的错误码。就像你在 UNIX 系统中调用几乎所有的库函数一样，这些函数也可能失败！如果你的代码没有正确的检测错误码，那么错误就会默默的发生，这种情况下就有可能多个线程都进入了临界区。最起码，要用断言将其封装（如图 27.4）；更多的成熟（non-toy）的程序，在出错时不能简单的退出，当 lock 和 unlock 失败时，应当检测错误并采取一些适当措施。

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

Figure 2.4: 简单封装

lock 和 unlock 函数并不是 pthreads 仅有的锁相关函数。特别的，还有两个函数可能感兴趣函数：

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            struct timespec *abs_timeout);
```

这两个函数在获取锁时使用。trylock 函数会在锁已经被持有时返回失败；timedlock 函数会在得到了锁或者超时之后返回，无论那个先发生。因此，timedlock 函数在超时值为 0 时退化成 trylock 函数。一般讲，要避免使用这两个函数；然而，有少部分时候为了避免阻塞在获取锁上，这两函数还是很有用的，正如下一章我们将讲到的（如：当我们研究死锁时）。

2.4 条件变量

线程库的另一个主要组件就是条件变量，当然 POSIX 线程也是这样。当某种信号必须在信号间发生时，条件变量是很有用的，如一个线程需要等待其他线程完成某事才能继续执行。程序希望以这种方式交互时，会常用两个函数：

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

要使用条件变量，另外还需要有一个与之关联的锁。当调用这两个函数时，都要先持有对应的锁。

第一个函数，pthread_cond_wait()，会使调用线程休眠，等待其他线程唤醒它，通常是这个休眠的线程关心的某些事发生了改变。例如，典型的使用如下：

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;
Pthread_mutex_lock(&lock);
while (initialized == 0)
    Pthread_cond_wait(&init, &lock);
Pthread_mutex_unlock(&lock);
```

这段代码，在初始化了相关的锁和条件变量之后，线程检测变量 initialized 是否已经被设置为其他值而非 0。如果没有，线程就简单的调用 wait 函数休眠，直到其他线程唤醒它。

其他线程中唤醒这个线程的代码如下：

```
Pthread_mutex_lock(&lock);
initialized = 1;
Pthread_cond_signal(&init);
Pthread_mutex_unlock(&lock);
```

这段代码的顺序有几点要注意。首先，当发送信号时（同样在修改全局变量 initialized 时），总是要确保已经持有了锁。这保证了我们不会在代码中引入竞争条件。

第二，你也许注意到了 wait 函数用了一个锁作为其第二的参数，而 signal 函数仅仅只有一个条件变量参数。造成这点不同的原因是，wait 函数会将线程休眠，而 wait 函数需要在线程休眠前释放这个锁。设想一下如果不这么做：其他线程怎么获得这个锁并通知这个线程并唤醒呢？然而，在被唤醒之后，函数返回之前，pthread_cond_wait() 函数会重新获取这个锁，因此确保了任何时间，等待线程是运行在持有锁和释放锁之间的。

最后一个奇怪的是，等待线程在 while 循环里重复检测了这个条件，而不是一个简单的 if 判断语句。在后面研究条件变量的章节中会详细讨论这个问题，【这里简单的说】，一般的，用 while 循环是简单并安全的。虽然它重复检测这个条件（也许会增加一些负载），但是有一些 pthread 实现可能会虚假的唤醒等待线程；这种状况下，不重复检测的话，等待线程就会认为条件已经改变了，而实际上却没有改变。因此，将唤醒视作等待条件可能已经修改的提示而不是视作一个绝对的事实，这样会更安全一些。

注意到有时候，用一个简单的 flag 标志在两个线程间传递信号，而不用条件变量及其关联的锁，这是很诱人的。例如，我们可以将上面的等待代码改写成这样：

```
while (initialized == 0)
; // 空转
```

对应的发信号的代码像这样：

```
initialized = 1;
```

千万不要这么做，有以下几个原因。第一，许多情况下，它的性能很差（长时间空转只是在浪费 CPU 时间）。第二，它很容易出错。如最近的研究表明 [X+10]，像上述那样在线程间使用 flag 标志同步，是极其容易出错的；粗略的说，使用这种临时同步方法的代码，半数都是有 bug 的。别偷懒，乖乖的用条件变量，即使你觉得你可以避免这个问题。

2.5 编译运行

本章所有的示例代码都是比较容易上手并运行的。要编译这些代码的话，必须包含 pthread.h 头文件。在链接的时候，必须加上 -pthread 来显示的链接线程库。例如，要编一个简答的多线程程序，需要做的如下：

```
prompt> gcc -o main main.c -Wall -pthread
```

只要在 main.c 中包含了 pthreads 线程库的头文件，你就可以成功的编译并发程序了。这个程序能否如预期执行就是另一码事。

2.6 小结

本章介绍了线程库的基础内容：线程创建，通过锁创建互斥量，以及通过条件变量发信号和等待。除了耐心跟细心，不再需要其他东西就可以写出健壮高效的多线程程序。

我们以一些 tips 结束本章，这些 tips 可能在你编写多线程代码时很有用（详见下一页的 aside）。本章提到的 API 还有其他一些有意思的方面，如果你想获得更多信息，可以在 Linux 终端里输入 man -k pthread 来查看整个接口的一百多个 API。然而，本章讨论的基础应该可以让你构建出优秀（也希望正确且高性能）的多线程程序。多线程最困难的不是这些 API，而是你如何构建并发程序的复杂逻辑。继续阅读学习更多内容。

Aside：线程 API 指南

当你使用 POSIX 线程库构建一个多线程程序时，有一些细小但却很重要的注意点需要记住。他们是：

- **保持简单**。这一点高于一切，任何在线程间加锁或发信号的代码都要尽可能的简单。复杂的线程交互可能会导致 bug。
- **最小化线程交互**。保持线程间交互的方式最小。每一次交互都要仔细考虑，并通过尝试和正确的方法来构造（这些方法会在接下来的章节中学习）。
- **初始化锁和条件变量**。不这么做的话，你的代码就会很奇怪，有时候好好的，有时候就出错。
- **检查返回码**。当然，在你编写的任何 C 和 UNIX 程序，都应该检查每一个返回码，这一点在并发程序中也适用。不这么做的话，可能会导致奇怪的很难理解的行为，很可能会使你要么抓狂，要么扯头发【很可能会使你抓狂】。
- **注意如何传参给线程**，如何从线程返回值。特别的，传递一个在栈中分配的变量的引用的话，很有可能你就错了。
- **每个线程都有自己的栈**。跟前一点相关的，记住每个线程都有自己的栈空间。因此，如果某个线程在执行某个函数的时候，有一个局部分配的变量，这个变量本质上是该线程私有的；任何其他线程都没法轻易的访问它。要想在线程间共享数据，该变量应当在堆（heap）中或某个全局可访问的地方。
- **永远使用条件变量在线程间传信号**。使用简单的 flag 标志通常是很诱人的，但是不要这么做。
- **使用手册页**。特别的，Linux 的 pthread 手册页提供了非常有用的信息并讨论了许多本章提到的细微差别，而且更加详细。仔细阅读这些手册！

参考文献

[B97] “Programming with POSIX Threads”
David R. Butenhof
Addison-Wesley,May 1997
另一本关于线程的书。

[B+96] “PThreads Programming:
A POSIX Standard for Better Multiprocessing”
Dick Buttlar, Jacqueline Farrell, Bradford Nichols
O’Reilly, September 1996
O’Reilly 家的一本不错的书。O’Reilly 是一家非常优秀且很实用的一家出版社，我们的书架有很多这家公司的书，包括一些关于 Perl、Python 和 Javascript 的非常好的作品（尤其是 Crockford 的《Javascript: The Good Parts》）。

[K+96] “Programming With Threads”
Steve Kleiman, Devang Shah, Bart Smaalders
Prentice Hall, January 1996
本领域一本更好的书。值得收藏一本。

[X+10] “Ad Hoc Synchronization Considered Harmful”
Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, ZhiqiangMa
OSDI 2010, Vancouver, Canada
这篇文章展示了看似简单的同步代码是如何导致大量奇怪的 bug 的。用条件变量并正确的传递信号！

Chapter 3

锁

在并发性简介中，我们看到并发编程的一个基本问题：我们想要原子地执行一些列的指令，但是由于单处理器上中断的存在（或者多处理器上多线程并发执行），所以做不到。本章，我们通过提到过的锁来直接解决这个问题。程序员用锁标注源代码，将这些锁放在临界区的周围，因此确保这样的临界区像单条原子指令一样执行。

3.1 基本思想

作为示例，假设我们的临界区如下面的代码，一个典型的共享变量的更新：

```
balance = balance + 1;
```

当然，也有可能是其他的临界区，比如把一个元素添加到链表中或者其他更复杂的共享数据结构的更新，但是这里我们仅仅用这个简单的例子。为了使用锁，在临界区的周围添加一些代码，如下：

```
1 lock_t mutex; // some globally-allocated lock ' mutex'
2 ...
3 lock(&mutex);
4 balance = balance + 1;
5 unlock(&mutex);
```

锁只是一个变量，因此要使用锁的话，必须先声明一个某种类型的锁变量（比如上面的互斥量）。这个锁变量（简称锁）在任何时刻都持有这个锁的状态。它要么是可用的状态，因此没有线程持有这个锁；要么是已获取状态，因此恰有一个线程持有了这个锁，且可能进入了临界区。我们也可以在这个数据类型里存储一些其他信息，比如哪个线程持有了这个锁，或者等待获取锁的队列，但是这些信息是锁用户不可见的。

lock() 和 unlock() 函数的语义很简单。调用函数 lock() 尝试获取这个锁；如果没有其他线程持有这个锁，这个线程就会获得这个锁并进入临界区；那么这个线

程现在就是这个锁的拥有者。如果这时有其他线程对相同的锁变量（本例中的 mutex）调用 lock() 函数，直到它持有这个锁之后才会返回；通过这种方式，就可以防止其他线程在第一个持有锁的线程进入临界区的时候也进入临界区。

一旦锁的拥有者调用了 unlock()，锁就会又处于可获取状态（空闲）。如果没有其他线程等待这个锁（比如没有其他线程调用了 lock() 且阻塞在那里），那这个锁的状态就简单的转换称空闲态。如果有多个等待着的线程（阻塞在 lock() 函数），其中一个线程会注意到（或者被通知）锁的状态变化，然后获得锁并进入临界区。

锁为程序员提供了最起码的调度控制。一般讲，我将线程视作有程序员创建但由操作系统调度的实体，不管操作系统选择以何种方式实现。锁将一部分线程调度控制权交还给了程序员；通过在临界区周围放置锁，程序员可以保证不会超过一个线程能进入该临界区。因此，锁将传统操作系统调度的混乱转变得更加可控了。

3.2 线程锁

POSIX 库使用的锁的名字叫做 mutex，因为它用于在线程间提供互斥量（mutual exclusion），例如，如果一个线程已经进入临界区了，这个线程就会排斥其他线程进来，直到自己执行完临界区的代码。因此，当你看到下面的 POSIX 线程代码时，你应该能够理解他实际上做的就是上面说的那些（还是使用检测了 lock 和 unlock 错误的封装函数）：

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

也许你还注意到 POSIX 版本的锁会想 lock 和 unlock 传一个变量，是因为我们可能要使用不同的锁来保护不同的变量。这样做可以提高并发性：用不同的锁保护不通的数据和数据结构，这就允许更多的线程马上进入锁住的代码（细粒度的方式），而不是一个任何时候任何临界区都可用的大锁（粗粒度的锁策略）。

3.3 构造锁

到现在为止，从程序员的角度，你应该对锁是如何工作的有了一些理解。但是我们如何构造一个锁呢？硬件需要提供什么支持呢？操作系统又需要支持什么呢？这些问题就是本章剩下的内容要讲的。

Crux：如何构造锁

我们如何构造一个高效的锁？高效的锁以低开销提供了互斥量，同时也获得了一些其他性质，我们将在后面讨论。硬件需要提供什么支持呢？操作系统需要提供什么支持呢？

要构造一个可用的锁，需要我们的朋友——硬件和操作系统提供一些帮助。多年来，个中计算机体结构的指令集都添加了许多不同的硬件原语；虽然我们不需要学习这些指令是如何实现的（毕竟这是计算机体系结构课上的话题），但是我们还是要学习如何使用这些指令来构造一个像锁一样的互斥量原语。我们还会研究操作系统是如何参与完成这个工作的，并帮助我们构建一个精密的锁库。

3.4 评估锁

在构造锁之前，我们应该首先要明白我们的目标是什么，并因此问自己如何评价某个特定锁实现的的效率。要评估一个锁是否有用，应该建立一些基础标准。第一点是，锁能够实现基本的任务，即提供互斥量。基本地，这个锁有效么，能够阻止多线程进入同一个临界区么？

第二点是公平。一旦锁空闲了，是否每个竞争该锁的线程都被公平对待？从另一个方式看的话，就是测试更极端的情况：是否有竞争该锁的线程处于饥饿状态而一直得不到锁？

最后一点就是性能，尤其是用锁后的增加的时间开销。这里有几个不同的情况值得考虑一下。一个是没有竞争的情况；当只有一个线程在运行、获取、释放锁，锁开销有多少？另一个是多线程在单 CPU 上竞争同一个锁的情况，是否需要担心其性能？最后一点，当引入多 CPU，各个 CPU 上的多个线程竞争同一个锁的性能如何？通过对比这些不同的场景，我可以更好的理解使用各种锁技术对性能的影响。

3.5 中断控制

最早的互斥量的一个解决方案是为临界区关中断；这种方法是为单处理器系统设计的。其代码类似于下面：

```
void lock() {
     DisableInterrupts();
}
void unlock() {
     EnableInterrupts();
}
```

假设程序运行在一个单处理器系统上。通过在进入临界区前关中断（用某种特殊的硬件指令），我们可以确保进入了临界区的代码不会被中断，因此就可以好像原子一样的执行。临界区代码执行结束后重新开中断（还是用硬件指令），因此程序就可以像平常一样继续。

这个方法的好处就是简单。你当然不用想破了脑子去弄明白为什么这方法是可行的。没有了中断，线程就可以保证它执行的代码确实会执行，并且不会有其他线程会干扰它。

可是，负面影响有很多。首先，这个方法要求我们允许任何调用线程取执行特权操作（即中断的开和关），而且还要信任这个功能不会被滥用。正如你知的，任何时候我都要信任随意一个程序，很有可能会有麻烦的。麻烦会以多种方式发

生：一个贪心的程序可能在一开始的时候调用 lock()，因此而独占处理器；更糟糕的，一个不当的或者恶意的程序可能会调用 lock() 并进入无限循环。在后一种情况下，操作系统不能重新获得系统的控制权，只有一种办法：重启系统。用关中断作为一种通用的同步方案要求对应用程序太多的信任。

第二，这种方法在多处理器上行不通。如果多个线程运行在多个不同的 CPU 上，每个线程都试图进入相同的临界区，无论中断是否已经关了，线程还可以运行在其他的处理器上，并因此而进入了临界区。由于多处理器现在很普遍了，我们的通用方案必须要比这个方案好。

Aside：Dekker 和 Peterson 的算法

在二十世纪六十年代，Dijkstra 跟自己的朋友们提出了一个并发性问题，而且其中的一位叫做 Theodorus Jozef Dekker 的数学家提出了一个解决方案 [D68]，不像我们这个讨论的方案——用特殊的硬件指令，甚至是操作系统的支持，Dekker 的算法仅仅使用 load 和 store（假设它们彼此间是原子的，这在早期的硬件确实是的）。
Dekker 的方法后来被 Peterson 改进了 [P81]。这次也仅仅只用了 load 和 store，它的思想是确保两个线程从不会同时进入一个临界区。下面是 Peterson 的算法（对于两个线程）；如果能理解的话，可以看看。flag 和 turn 变量用来做什么呢？

```
int flag[2];
int turn;
void init() {
     flag[0] = flag[1] = 0; // 1->thread wants to grab lock
     turn = 0; // whose turn? (thread 0 or 1?)
}
void lock() {
     flag[self] = 1; // self: thread ID of caller
     turn = 1 - self; // make it other thread's turn
     while ((flag[1-self] == 1) && (turn == 1 - self))
          ; // spin-wait
}
void unlock() {
     flag[self] = 0; // simply undo your intent
}
```

由于某些原因，一段时间内，开发出不用特殊硬件指令的锁成了时尚，给出了很多问题以及相对应的理论类型。当然，当假设存在硬件支持时，就能更加容易地实现它，那么那些理论型的工作就没有任何用处了（确实这样的支持在多处理器的早期就已经有了）。进一步的，跟上面类似的算法在现代硬件上已经行不通了（因为弱内存一致性模型），因此使得他们比之前更加无用了。然而还有更多的研究被淹没在历史的长河里。

第三，也许最不重要的，这个方案的效率很低。与普通指令的执行相比，在现代 CPU 上中断的开关通常会慢一些。

由于这些原因，关中断只能在有限的情况下作为互斥量原语。例如，当操作系统在访问自己的数据结构时，会用中断的开关来保证原子性，至少可以避免发生繁杂的中断处理情况。这种用法是有意义的，因为信任问题是在 OS 内部，毕竟不管怎样还是会信任自己执行特权操作的。

3.6 test-and-set (原子交换)

由于关中断在多处理器下行不通，那么系统设计这就开始研究为锁提供硬件支持。最早的多处理系统已经有了相应的支持，比如上世纪六十年代早期的 Burroughs B5000。目前所有的系统都提供了这种类型的支持，即使是单 CPU 系统。

要理解的硬件支持最简单的部分就是 test-and-set 指令，也被称作原子交换。为了理解 test-and-set 是如何工作的，我们先试着不用它来构造一个简单的锁。在这个失败的尝试中，我们用一个简单的 flag 变量来表示该锁是否已被持有。

```
1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t *mutex) {
4      // 0 -> lock is available, 1 -> held
5      mutex->flag = 0;
6  }
7
8  void lock(lock_t *mutex) {
9      while (mutex->flag == 1) // TEST the flag
10         ; // spin-wait (do nothing)
11     mutex->flag = 1; // now SET it!
12 }
13
14 void unlock(lock_t *mutex) {
15     mutex->flag = 0;
16 }
```

Figure 3.1: 首次尝试：简单的 flag 标志

在第一次尝试中（图 28.1），思想很简单：用一个简单的变量来表示是否有线程已经拥有了该锁。第一个进入临界区的线程会调用 lock()，lock() 会测试 flag 是否等于 1 （），然后将 flag 置为 1 来表示这个线程已经持有该锁。当临界区代码执行结束，该线程会调用 unlock() 将 flag 清除，因此表示该所不再被这个线程持有。

如果在第一个线程在临界区里时，另一个线程恰好也调用了 lock()，这个线程就会简单的在 while 循环里自旋，等待持有该锁的线程调用 unlock() 清除 flag。一旦第一个线程清楚了 flag，在等待的线程就会跳出 while 循环并将 flag 置为 1，然后进入临界区。

可是，这个代码存在两个问题：一个是正确性，另一个是性能。只要你曾经思考过并发编程，那么正确性问题很简单。想象一下这个代码如表 28.1 那样交替执行（设 flag 初始值为 0）。

Thread 1	**Thread 2**
call lock()	
while(flag == 1)	
interrupt: switch to Thread 2	
	call lock()
	while(flag == 1)
	flag =1;
	interrupt: switch to Thread 1
flag = 1; //set flag to 1(too!)	

Table 3.1: 轨迹：无互斥量

Tip：think about concurrency as malicious scheduler （像恶意调度器一样思考并发性）

从这个简单的例子中，你也许感觉需要花点时间来理解这个方法的并发执行。你需要做的是，假设你是一个恶意调度器，总是在最不合时宜的时刻中断线程，以破坏他们为构造同步原语而做的微不足道的尝试。好一个奸诈的调度器！尽管恰好是表中的那个中断顺序的概率不大，但是它是可能的，这就是我们需要阐述的——特定的方法是行不通的。怀着恶意的去思考是有用的！（至少有时候是的）

正如你在上述例子中看到的，通过不及时的中断，我们可以很容易的就画出两个线程都将 flag 置为 1 且都进入了临界区的场景。这种行为就是内行说的”bad”，很明显这次尝试没能够实现最基本的要求：提供互斥量。

关于性能问题，我们会在稍后多讲一些。性能问题其实是线程等待获取已经被持有的锁的方式：它不停的检查 flag 的值，这个技术称作自旋等待（spin-waiting）。自旋等待为了等其他线程释放锁而浪费时间。这个时间浪费在单处理器上格外的高，等待线程等待的那个线程甚至无法运行（至少在上下文切换前是不能运行的）【此处是否要注一下】！因此，在开发出更有效的解决方案前，我们还是应该考虑一下如何避免这种浪费的方法。

3.7 构造一个可行的自旋锁

尽管上一个示例背后的思想是好的，但不靠硬件的某些支持来实现目标还是不可能。幸运的是，有些系统提供了基于这个思想创建简单的锁的指令。这个强有力的指令在不同的平台有不同的名字，在 SPARC 上，是 load/store 无符号字节指令（ldstub），在 x86 上是原子交换指令（xchg）；这些在不同平台上的指令功能都是一样的。他们一般都称作 test-and-set。我们将 test-and-set 指令的行为定义为如下代码片段：

```
1 int TestAndSet(int *ptr, int new) {
```

```
2      int old = *ptr; // fetch old value at ptr
3      *ptr = new; // store ' new' into ptr
4      return old; // return the old value
5 }
```

test-and-set 指令行为是这样的，它返回 ptr 指向的旧值，同时 ptr 的值更新为 new。当然，关键是这一些列的操作是以原子形式执行的。叫作”test and set”的是因为这条指令能够让你”test”旧值（即返回值），同时也将内存值”setting" 为新值；故而，这个略有些强大的指令已经足够用来构造自旋锁（spin lock）了，如图 28.2 所示。

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0 indicates that lock is available, 1 that it is held
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

Figure 3.2: 用 test-and-set 构造简单的自旋锁

为确保理解了这个锁为什么可行，再来梳理一下。第一个情况，一个线程调用了 lock()，并且目前没有其他线程持有这个锁，因此 flag 此时值为 0。当线程调用了 TestAndSet(flag,1), 这个例程会返回 flag 的旧值——0；因此测试（testing）flag 值的线程不会在 while 循环中自旋并获得了锁。这个线程并会原子的置（set）flag 的值为 1。至此该线程就获得了这个锁。当这个线程执行完临界区的代码，再调用 unlock() 将 flag 的值重新置为 0。

第二种情况，假设已经有一个线程获得了锁（flag 值为 1）。这时候，当前线程调用 lock() 并执行 TestAndSet(flag,1)。这次，TestAndSet() 会返回旧值，即 1（因为此时锁已经被某个线程持有）并且还是将 flag 置为 1。只要锁被其他线程持有，TestAndSet() 就会不停地返回 1，因此这个线程就会自旋到锁被释放。当 flag 被某个线程置为 0，当前线程就会再次调用 TestAndSet()，此时 TestAndSet() 会返回 0，同时将 flag 置为 1 而获得该锁并进入临界区。

通过将 test（旧值）和 set（新值）改成一个原子操作，就可以确保只有一个线程获得这个锁。这就是构造一个可行的互斥量的方法。

你也许也明白了为何这种类型的锁也称作自旋锁。这是要构造的最简单的锁，仅简单的自旋到锁可用为止。为了在单处理器上正确工作，需要有一个可抢占的调度器（比如可通过定时器中断的调度器，以便可以运行其他线程）。不能抢占的话，自旋锁在单处理器上就没有太多的意义了，因为线程在 CPU 上自旋并不会自己结束。

3.8 评估自旋锁

基于上述的自旋锁，我们可以通过之前提到的指标评估一下它的效率如何。锁最重要的是正确性：是否能够支持互斥量？其答案还是明显是 yes：自旋锁一次只允许一个线程进入临界区。因此自旋锁是正确的。

另一个指标是公平性。自旋锁对等待线程的公平性如何呢？能够保证等待线程都会进入临界区么？不幸的是，这个答案不太好：自旋锁没有任何公平性保证。实际上，在竞争中，一个自旋线程会一直自旋下去。自旋锁是不公平的也会导致饥饿。

最后一个指标是性能。使用自旋锁的开销有多大？为了更仔细地分析性能，建议考虑几个不同的情况。第一个，多个线程在单个处理器上竞争锁；第二个，多个线程分散在多处理器上竞争锁。

在单 CPU 上，对于自旋锁，性能很糟糕；设想这样一种情况，已经持有锁的线程在临界区内是可抢占的。其他每个线程都会尝试获取这个锁，那么调度器可能执行所有其他线程（假设有 N-1 个线程，不包括当前线程）。这样的话，每个线程在放弃 CPU 之前都要在一个时间片内自旋，这是对 CPU 时间的浪费。

然而，在多 CPU 上，自旋锁的性能还可以（假设线程数与 CPU 数大致相当）。思路大致是这样的：假设线程 A 运行在 CPU 1 上，线程 B 运行在 CPU 2 上，同时竞争一个锁。如果线程 A（CPU 1）获得了锁，那么线程 B 试图获取锁的话，就会等待（在 CPU 2）。然而，临界区一般会比较短，因此锁会很快就转换称空闲状态，线程 B 就可以获得锁了。在另一个处理器上自旋等待锁并没有浪费太多的 CPU 时间，因此效率还可以。

3.9 compare-and-swap

有些系统提供的另一个硬件原语是 compare-and-swap 指令（SPARC），或者 compare-and-exchange（x86）。这条指令的 C 语言伪代码如图 28.3。

```
1  int CompareAndSwap(int *ptr, int expected, int new) {
2      int actual = *ptr;
3      if (actual == expected)
4          *ptr = new;
5      return actual;
6  }
```

Figure 3.3: Compare-and-swap

compare-and-swap 的基本思想是检测 ptr 指定地址的值是否与 expected 相等；如果相等，就将 ptr 指向的内存地址更新为 new 值。如果不等的话，什么都不做。最后都返回 actual 值，从而可以让调用 compare-and-swap 指令的代码知道成功与否。

通过 compare-and-swap 指令，我们可以用与 test-and-set 指令类似的方法构造一个锁。比如，仅仅将之前的 lock() 函数替换成这样：

```
1 void lock(lock_t *lock) {
2     while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3     ; // spin
4 }
```

其他的代码与上面的 test-and-set 例子一样。这段代码的运行过程跟之前的例子很类似；简单地检查 flag 是否是 0，如果是 0 的话，就原子地的 swap 新旧值（即将 flag 置为 1）并获得锁。试图获得已经被持有锁的线程会阻塞等待，直到锁被释放。

如果你想看看 C 可调用的 x86 版本的 compare-and-swap 的构造，下面这段代码也许有用 [S05]：

```
1  char CompareAndSwap(int *ptr, int old, int new) {
2      unsigned char ret;
3
4      // Note that sete sets a 'byte' not the word
5      __asm__ __volatile__ (
6          "␣lock\n"
7          "␣cmpxchgl␣%2,%1\n"
8          "␣sete␣%0\n"
9          : "=q" (ret), "=m" (*ptr)
10         : "r" (new), "m" (*ptr), "a" (old)
11         : "memory");
12         return ret;
13 }
```

最后，也许你已经感觉到，compare-and-swap 指令比 test-and-set 要更强大一些。我们将会在后面简要的研究 wait-free synchronization[H91] 时用到这个指令。然而，如果只是用它构造一个自旋锁的话，这与之前分析的自旋锁没有区别。

3.10 load-linked 和 store-conditional

有些平台支持一对合同构造临界区的指令。例如，在 MIPS 架构上 [H93]，load-linked 和 store-conditional 指令可以联合用来构造锁和其他并发结构。其 C 伪代码如图 28.4 所示。Alpha、PowerPC 和 ARM 上都有类似的指令 [W09]。

load-linked 指令的操作与典型的 load 指令很类似，简单的从内存中取值并将其放入寄存器中。不同之处在于 store-conditional，它仅在该地址没有 intermittent store 发生时成功（并更新相应地址的值，该地址只能是 load-linked 返回的）。如果成功的话，store-conditional 返回 1 并更新 ptr 指向的值为 value；如果失败的话，不更新 ptr 指向的值并返回 0。

```
1  int LoadLinked(int *ptr) {
2      return *ptr;
3  }
4
5  int StoreConditional(int *ptr, int value) {
6      if (no one has updated *ptr since the LoadLinked to this address) {
7          *ptr = value;
8          return 1; // success!
9      } else {
10         return 0; // failed to update
11     }
12 }
```

Figure 3.4: Load-linked 和 Store-conditional

你可以自我挑战一下，试着想想用 load-linked 和 store-conditional 指令如何构造一个锁。尝试之后，再看看下面的示例代码提供的一个简单解决方法。试试看，该方法就在图 28.5 中。

```
1  void lock(lock_t *lock) {
2      while (1) {
3          while (LoadLinked(&lock->flag) == 1)
4              ; // spin until it's zero
5          if (StoreConditional(&lock->flag, 1) == 1)
6              return; // if set-it-to-1 was a success: all done
7                      // otherwise: try it all over again
8      }
9  }
10
11 void unlock(lock_t *lock) {
12     lock->flag = 0;
13 }
```

Figure 3.5: 用 LL/SC 构造锁

lock 函数才是有趣的部分。首先，某个线程自旋等待 flag 标志置为 0（意味着锁处于空闲态）。一旦锁处于空闲态，这个线程就尝试用 store-conditional 指令获取锁；如果成功了，那么这个线程就原子地修改 flag 值为 1 并进入临界区。

要注意 store-conditional 指令可能会失败。某个线程调用了 lock() 并执行了 load-linked，得到返回值为 0，表示锁处于空闲态。此时，即在执行 store-conditional 之前，它被中断，另一个线程调用了 lock()，也执行了 load-linked 指令，也得到返回值 0。此时，两个线程各自都执行了 load-linked 指令，并且都将要执行 store-conditional 指令。这条指令的关键性质就是只有一个线程能成功的更新 flag 的值为 1 并获得锁；另一个线程执行 store-conditional 会失败（因为前一个线程

在 load-linked 和 store-conditional 之间更新了 flag 的值）并需要再次尝试获取锁。

TIP：更少的代码才是更好的代码（Lauer 定律）

程序员们在某些事的时候，都倾向于吹嘘自己写了许多代码。这么做从根本上就是错的。然而，需要炫耀的是对于某个任务用了多么少的代码。短小精悍的代码才是更好的选择；这样才有可能更容易的理解并少有 bug。当讨论到 Pilot 操作系统的构建时，Hugh Lauer 说道：”如果同一个人有两次机会，那他可以用一半的代码量来构造出一个同样优秀的系统”[L81]。我们称之为 Lauer 定律，非常值得记住它。所以，下次你在炫耀你在做某项任务时写了多少代码时，请再想想；更好的话，回去重写一遍，把代码写得尽可能的短小精悍。

在几年前的课堂上，一位研究生 David Cape 为那些喜欢用短路布尔条件（short-circuiting boolean conditionals）表达式的同学，提供了一个更简介的形式。如果能够弄明白为什么这两个形式是等价的话，那就看看吧。这确实更短！

```
1 void lock(lock_t *lock) {
2   while (LoadLinked(&lock->flag)||!StoreConditional(&lock->flag, 1))
3         ; // spin
4 }
```

3.11 fetch-and-add

最后一个硬件原语是 fetch-and-add 指令，它原子地将某一地址的值加 1.fetch-and-add 指令的 C 语言伪代码如下所示：

```
1 int FetchAndAdd(int *ptr) {
2     int old = *ptr;
3     *ptr = old + 1;
4     return old;
5 }
```

代码

在下面的例子中，我们将用 fetch-and-add 指令来构造一个更有意思的排队锁（ticket lock），由 Mellor-Crummey 和 Scott 提出 [MS91]。lock 和 unlock 代码如图 28.6 所示。

不像一个单独的值，这个方案里用了 ticket 和 turn 变量作为组合来构造锁。其基本操作很简单：当一个线程希望获得锁时，它先对 ticket 值做一次原子地 fetch-and-add 操作；此时这个值就作为这个线程的”turn”（myturn）。然后，全局共享变量 lock->turn 用来决定轮到了哪个线程；当对于某个线程 myturn 等于 turn 时，那就轮到了这个线程进入临界区。unlock 简单地将 turn 值加 1，由此下一个等待线程（如果存在的话）就可以进入临界区了。

注意这个方法相对于前面的几种方式的一个重要不同：它保证了所有线程的执行。一旦某个线程得到了他自己的 ticket 值，在将来的某一时刻肯定会被调度

```
1  typedef struct __lock_t {
2      int ticket;
3      int turn;
4  } lock_t;
5
6  void lock_init(lock_t *lock) {
7      lock->ticket = 0;
8      lock->turn = 0;
9  }
10
11 void lock(lock_t *lock) {
12     int myturn = FetchAndAdd(&lock->ticket);
13     while (lock->turn != myturn)
14         ; // spin
15 }
16
17 void unlock(lock_t *lock) {
18     FetchAndAdd(&lock->turn);
19 }
```

Figure 3.6: 排队锁 (Ticket Locks)

执行（一旦前面的那些线程执行完临界区并释放锁）。在先前的方案中，并没有这一保证；比如，某个自旋在 test-and-set 的线程可能会一直自旋下去，即使其他的线程获得、释放锁。

3.12 小结：如此多的自旋

这些的基于硬件的锁都很简单（只有寥寥数行代码）并且也是可行的（你可以写些代码来证明），这对于任何系统或代码来说都是很好的两个性质。然而，在某些情况下，这些方法都非常的低效。设想，你在单个处理器上运行两个线程。其中一个线程（线程 0）正在临界区中，因此锁是被其持有的；它被不幸地中断了。另一个线程（线程 1）此时试图获得锁，但是发现它被另一个线程持有了，那么它就开始等啊等啊等，直到最后定时中断到达。线程 0 再次运行，释放锁，线程 1 下次运行的时候就不用再等那么久了，它会马上获得锁。因此任何一个发生线程 1 这种情况的线程，都会浪费一整个时间片做无用功来检测一个根本不会改变的值！如果是 N 个线程竞争同一个锁的话，情况会更糟糕；N-1 个时间片会以类似的方式被浪费——简单的等待单个线程释放锁。那么，我们的下一个问题：

Crux：如何避免自旋

怎么开发出一个不会浪费时间，不在 CPU 上无谓自旋的锁？

仅有硬件支持不能解决这个问题，还需要操作系统的支持！下面就来弄明白

怎么实现这样的锁。

3.13 简单方法：放手吧，孩子！

硬件支持已经让我们走了很远：可行的锁，甚至是获取锁的公平性（ticket lock）。然而，我们还有个问题：在临界区中发生了上下文切换时需要做什么，多个线程开始无休止的自旋等待被中断线程（扔持有锁）在此执行？

第一次尝试一个简单但友好的方法：当线程将要自旋时，换作放弃 CPU 给其他线程，或者如 Al Davis 说：”放手吧，孩子！”[D91]。图 28.7 呈现了这个方法。

```
1  void  init () {
2      flag  = 0;
3  }
4
5  void  lock() {
6      while  (TestAndSet(&flag, 1) == 1)
7          yield () ; // give up the CPU
8  }
9
10 void unlock() {
11     flag  = 0;
12 }
```

Figure 3.7: Lock with test-and-set and yield

这个方法中，假设有一个操作系统原语 yield()，可以在需要放弃 CPU 让其他线程执行时调用它。线程可以处于三个状态（运行、就绪和阻塞）中的一个；yield 是一个将调用者从运行态转换称就绪态的简单系统调用，从而推进其他线程执行。因此，yield 过程实质上是对自己取消调度（deschedule）。

考虑这么一个简单的例子，有两个线程运行在一个 CPU 上，我们的这个基于 yield 的方法就相当的好。如果线程碰巧调用 lock()，发现这个锁被其他线程持有了，它就简单地放弃 CPU，那么另一个线程就可以运行并完成临界区的任务。在这个简单的例子里，这个 yield 的方法可以工作的很好。

我们考虑这样的场景，有许多线程（假设 100 个）不停地竞争同一个锁。此时，如果其中一个线程获得了锁并在释放它之前抢占着；其他 99 个线程就会各自调用 lock()，发现锁不处于空闲态就放弃 CPU。假设系统采用某种 round-robin 调度器，在持有锁的线程在此得到执行前，这 99 个线程各自都会执行一次 run-and-yield 流程。尽管比前面讲的自旋方法（会浪费 99 个时间片来自旋）好，但是这个方法开销还是很大；上下文切换的开销还是很显著的，因此会有大量的浪费。

糟糕的是，我们还是没有解决饥饿的问题。如果其他线程不听的进入、退出临界区，某个线程还是有可能会不停的放弃 CPU。显然我们需要一个直接解决这个问题的方法。

3.14 队列：休眠而非自旋

先前的那些方法存在的真正问题是它们将太多的东西交给了运气。由调度器决定下次执行哪个线程；如果调度器做了一个不好的选择，那么该线程要么自旋等待锁（第一种方法），要么立即放弃 CPU（第二种方法）。不管哪种方法，都有浪费 CPU 和引起饥饿的潜在可能性。

因此我们在当前持有者释放了锁之后，必须明确地对哪个线程获得锁采取一些控制。要这么做，需要操作系统提供一些支持，也就是用一个队列来跟踪哪些线程在等待获取这个锁。

简单起见，我们用 Solaris 提供的两个系统调用：park()，使调用线程进入休眠，unpark(threadID) 唤醒 threadID 指定的某个线程。这两个例程可以联合用来构造一个锁，将试图获取非空闲态锁的线程转换至休眠，当锁被释放后就唤醒这个线程。来看一下图 28.8 所示的代码以理解使用这个原语的一种方式。

在本例中，将做几个有意思的事情。首先，我们将原来的 test-and-set 思想和锁等待队列结合来构造一个更高效的锁。第二，我们用一个队列来帮助控制哪个线程下次获得锁，从而避免饥饿。

首先，你也许会注意这些 guard 是如何使用的，比如锁使用的 flag 前后的自旋锁和队列操作。这个方法并没有完全避免自旋等待；线程在获取或释放锁时坑你会被中断，从而引发其他线程需要自旋等待这个线程在此执行。然而，自旋花费的时间是相当有限的（仅仅是 lock 和 unlock 代码中的几条指令。而不是用户定义的临界区），因此这个方法还是合理的。

其次，你也许注意到 lock() 中，当某个线程不能获取锁（已被持有）时，会小心的将它添加到队列中（通过调用 gettid() 获取当前线程的线程 ID），将 guard 置为 0 然后放弃 CPU。读者可能有疑问：如果将 guard 锁的释放放在 park() 之后而不是前面，会发生什么呢？提示：something bad。

你也许还注意到一个有趣是事实：当另一个线程被唤醒后，flag 的值并没有被置回 0。为什么会这样呢？当然这不是一个错误，而是有其必要性的！当一个线程被唤醒，他似乎是从 park() 中返回的；然而，它此时并没有持有 guard，因此不可能将 flag 置为 1。从而，我们只需将锁直接从释放锁的线程传给下一个获取了锁的线程；flag 在这之间没有被置为 0。

最后，你应该注意到这个解决方案里竞争条件仅仅在 lock 调用的前面（一直到 park()）。只需一个错误的实际，某个线程正要执行 park()，假定它要休眠到锁不再被持有。此时切换至另一个线程（即持有锁的线程）会导致问题，比如，如果该线程释放了锁。第一个线程紧接着的 park() 就会永远休眠。这个问题有时被称作 wakeup/waiting race；要避免它，还需要一些其他帮助。

Solaris 通过添加第三个系统调用:setpark()。通过吊用这个例程，线程就可以表明它将要进入 park 了。如果刚好发生了中断，且其他线程在真正调用 park 之前调用了 unpark，那么接下来的 park 就会立即返回而不是休眠。lock() 里的代码修改很少：

```
1          queue_add(m->q, gettid());
2          setpark(); // new code
3          m->guard = 0;
```

另一个不同的解决方法，可以将 guard 传递进内核。这样的话，内核就可以采取预防措施来原子地释放锁并 dequeue 运行线程。

```
1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; // acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; // acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock (for next thread!)
33     m->guard = 0;
34 }
```

Figure 3.8: Lock with Queues、test-and-set、yield and wakeup

3.15 不同操作系统，不同支持

目前，就操作系统在线程库中为构造更高效的锁而提供的支持，已经了解过了其中的一种形式。其他的操作系统也提供了类似的支持，细节都不一样。

例如，Linux 提供的 futex，与 Solaris 提供的接口类似，但是 futex 提供了更多内核功能。特别的，每个 futex 可以与某个特定的物理内存位置关联；与之关联的内存位置是内核队列。线程可以调用 futex 来休眠或者唤醒。

特别地，还有两个系统调用。futex_wait(address, expected) 调用在 address 处的值与 expected 相等时，使调用线程休眠。如果不相等的话，该系统调用会立即

返回。futex_wake(address) 调用唤醒等待队列中的某个线程。它们在 Linux 中的使用方法见图 28.9。

```
1  void mutex_lock (int *mutex) {
2      int v;
3      /* Bit 31 was clear, we got the mutex (this is the fastpath) */
4      if (atomic_bit_test_set (mutex, 31) == 0)
5          return;
6      atomic_increment (mutex);
7      while (1) {
8          if (atomic_bit_test_set (mutex, 31) == 0) {
9              atomic_decrement (mutex);
10             return;
11         }
12         /* We have to wait now. First make sure the futex value
13             we are monitoring is truly negative (i.e. locked). */
14         v = *mutex;
15         if (v >= 0)
16             continue;
17         futex_wait (mutex, v);
18     }
19 }
20
21 void mutex_unlock (int *mutex) {
22     /* Adding 0x80000000 to the counter results in 0 if and only if
23         there are not other interested threads */
24     if (atomic_add_zero (mutex, 0x80000000))
25         return;
26
27     /* There are other threads waiting for this mutex, wake one of them up. */
28     futex_wake (mutex);
```

Figure 3.9: 基于 Linux 的 Futex 锁

摘自 NPTL 库（GNU libc 库的一部分）中 lowlevellock.h 的代码片段 [L09] 非常有意思。基本地，它用单个 integer 跟踪锁是否被持有（integer 的最高位 bit）以及这个锁有多少个等待者（剩余的其他 bits）。因此，如果锁是负的，即它被某个线程持有了（因为最高位被置 1 了，且此位决定 integer 的符号）。这些代码如此有趣还因为它们展示了在没有竞争的正常情况下如何优化：在仅有一个线程获取、释放锁时，只有很少的开销（lock 的 atomic bit test-and-set 和释放锁的原子加操作）。如果能够弄明白其余这些”real world”锁是如何工作的，那就去看一看吧。

3.16 两段 (two-phase) 锁

最后一个 note：Linux 的锁还有一些老方法的味道，老方法已经断断续续的使用了很多年了，最早至少可以追溯到上世纪六十年代的 Dahm 锁 [M82], 现在被称

作两段锁（two-phase lock）。两段锁实现自旋会很用，特别在锁即将被释放时。在第一阶段，锁自旋一段时间，希望可以获得锁。

然而，如果在第一阶段的自旋中没有获得锁，就进入第二阶段，此阶段调用者进入休眠直到锁变为空闲态后才被唤醒。前面的 Linux 的锁就是这种锁的一种形式，但是他仅仅自旋一次；这种锁的一个原则就是在调用 futex 进入休眠前可以在循环中自旋固定次数。

两段锁还是混合（hybrid）方法的另一个实例，这种方法将两个优秀的思想结合以实现一个更好的方法。当然，这种方法是否真的更好还取决于很多其他因素，包括硬件环境、线程数和其他负载细节。通常，构造一个适用于所有可能情况的通用锁，是很具有挑战性的。

3.17 小结

上述的方法讲解了当前真实的锁是如何构造的：一些硬件支持（那些强大的指令）加上一些操作系统支持（Solaris 的 unpark() 原语或者 Linux 的 futex）。当然，各自的细节都不尽相同，实现这些锁的代码都是非常优秀的。如果想要了解更多细节的话，可以去看 Solaris 或 Linux 的代码库；它们都非常吸引人 [L09,S09]。也可以看看 David 等人对比现代微架构上各种锁策略的优秀的成果 [D+13]。

部分习题答案

习题一

Bibliography