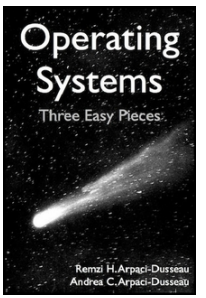
操作系统
Three Easy Pieces

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的写作本书一样的快乐的阅读本书。这本书的名字是——操作系统：three easy pieces，这个题目明显是向由 Richard Feynman 撰写的伟大的关于物理学话题讲义集致敬 [F96]。虽然这本书无疑达不到这位著名物理学家高度，也许它能够达到你理解操作系统（更一般的，系统）关键所在的要求。

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，你在这个过程中寻找到乐趣。学习新事物多么有趣，对吧？至少，我门觉得应当是这样。

每个主题将分为多章来讲解，每一章会呈现一个特定的问题以及如何去解决它。每章都比较短，会尽可能的去引用提出该思想的最原始材料。我们习作本书的一个目的是使得历史的路径尽可能的清晰，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，过去是什么样子的，将来会是什么样子的。在此情况下，seeing how the sausage wasmade is nearly as important as understanding what the sausage is good for【1. 提示：吃的！如果你是素食主义者，请远离。】。

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 会在正文中明显的提出来，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的经验。为了方便，在本书结尾的索引处列出了所有的 asides 和 tips 以及 crux。

贯穿本书，我们用”对话”这一古老的教学方式，作为以不同的观点来呈现一些素材的方式【??】。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。这些对话也是用更加幽默的风格写作的契机。你是否发现他们很有用或很幽默，好吧，这完全是另一码事儿。

在每个主要部分开始时，我会先给出操作系统提供的抽象概念，接着的章节是为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是很重要的，所以一点也不奇怪，对操作系统来说它们也是很必要的。

在这些章节中，我们试图尽可能（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：给定一个不通的随机种子，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。

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、实现和测试是如何工作的。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的，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，但又向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。

To Educators

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，且可以从下列网页中获取：http://www.ostep.org 你也可以从 lulu.com 网站购买纸质版。请从上述网页上查询。

引用本书的推荐格式（到目前为止）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，将其放在了对 CPU 和内存的虚拟化有了一定理解的虚拟化之后。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那个过时的教学方法：粉笔和黑板。因此，当我们在讲授这么课程时，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；讲义以及现场编写掩饰代码也是很有用的。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，因为他们知道反正材料就在这里，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了。

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（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。

To Students

如果你是正在阅读本书的学生，谢谢你！我们很荣幸能够给你提供一些材料帮助你求索关于操作系统的知识。我们俩都怀念起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你的一个美好回忆。

你也许注意到本书是免费的，在网上可以获取到。为此的主要原因是：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. 图灵奖是计算机科学领域的最高奖；就像诺贝尔奖一样。】），因此我们应该尽可能的去纪念这些思想和人物。在这样做时，我们非常希望能更好的理解已经发生的革命，而不是仿佛这些思想一直存在似的写本书 [K62]。再者，或许这些参考文献能够鼓励你自己去更深的挖掘；阅读著名的论文是我们这个领域最好的学习方法之一。

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, RiccardoMutschlechner, 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."

Johnnie Xu
****@****
2015 年 01 月

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

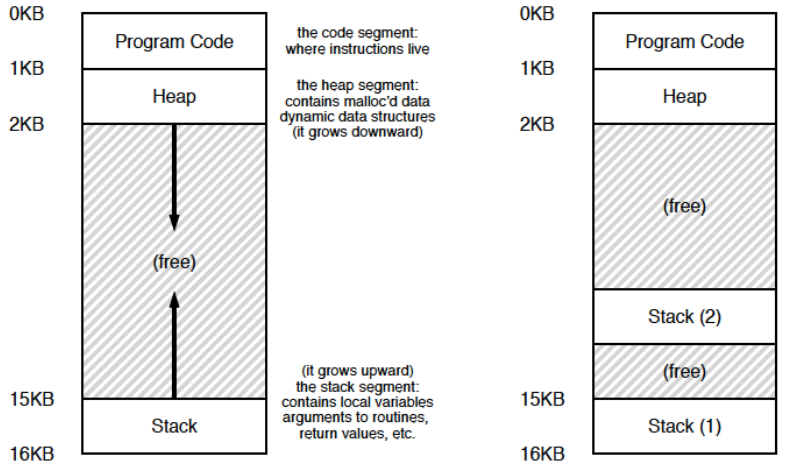


Figure 26.1: **A Single- and Multi-Threaded Address Space**

Figure 1.1: 图 26.1：单线程和多线程的地址空间

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，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。

```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

Figure 1.2: 简单的线程创建代码

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.

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**	**counter**
	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。

部分习题答案

习题一

Bibliography