**The Little Book of Semaphores (Second Edition)**

Allen B. Downey

Version 2.2.1

Permission is granted to copy, distribute and/or modify this document under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0) at <http://creativecommons.org/licenses/by-nc-sa/4.0>.

**前言**

大多数本科操作系统（Operating Systems）课本含有同步（synchronization）这一章节，通常展现一组原语（a set of primitives），如互斥mutexes，信号量semaphores，管程monitors，条件变量condition variables等，还有一些经典问题比如读者-写者，生产者-消费者。

当我在伯克利Berkeley上操作系统课程，和在Colby学院教学的时候，我有一种印象就是大多数学生都能够理解这些问题的解决方法，但是很少会有创造或者解决类似问题的学生。

学生并没有深入了解这些资料的其中一个原因就是操作系统课程需要更多的时间，更多的实践，要比上课时间还要多。同步只是其中一个在操作系统课程上有竞争空间的模块，我不确定我能不能说这是最重要的，但我认为这是一个最具挑战的，有趣的。

我写这本书的第一版的时候，目的是希望可以找到识别同步原语和模式，能用于单独理解和用于解决复杂的问题。这是一个挑战，因为同步代码没有组合好，随着同步组件的数量增加，交互也随之增加变得难以管理。

然而，我在看解决方案的时候找到了模式，并发现了至少有一些系统的方法来组装解决方案是正确的。

在Wellesley学院教操作系统的时候，我有机会来测试这种方法。我用了第一版的这本书和一本标准的教材，而且在课程期间教了同步的并发线程。每周我给学生书中的几页，会以谜题结束，有时会有提示。我告诉他们不要看提示除非真的是被难住了。

我也会给他们一些工具来测试他们的解决方案：一块小的磁白板，他们可以在上面写代码，和用一堆磁铁来展示线程执行的代买。

结果是令人激动的。给学生越多时间来吸收材料，学生越能展现出一种我之前从未见过的深入理解。最重要的是，他们大多数都能解决大部分的谜题。在一些情况下，他们还能重新发明经典的解决方案，在另外的情况下，他们能创造新的解决方法。

当我搬到Olin学院后，我采取下个步骤，和创建了半个课程，叫同步，包含了这本书和在x86汇编语言、POSIX，Python上实现的同步原语。

那些参加了课程的学生帮我找到第一版课本的错误，还有一些提出了比我更好的解决方法。在学期的末尾，我让他们写一个新的，最初的问题（最好是有解决方案的）。我把他们的贡献添加到第二版上。

从第一版出来后，Kenneth Reek发表了一片文章“Design Patterns for Semaphores”在ACM计算机专业兴趣小组科学教育上。他提出了一个问题，我把它当做Sushi Bar问题，和两种解决方法来展现这种模式，他称之为“Pass the baton”和“I’ll do it for you”。

一旦我开始领悟到这些模式后，我就开始把第一版的一些问题改写成我认为比较好的解决方案。

第二版的另外一个挑战就是语法了。当我写完第一版后，我学习了Python，不仅是一门非常棒的语言，同时也是非常好的伪代码语言（pseudocode language）。所以我从第一版的类C语法转化成和Python执行语言非常相近的语法。事实上，我已经写了一个模拟器，能执行本书中大多数的解决方案。

不熟悉Python的读者可能会发现得更明显。在我用Python特定特征的情况下，我会解析这些语法和他们的意思。我希望这些改动能这本书更具可读性。

这本书的页码可能会有点奇怪，一些的空白是我有意为之的。在每个谜题的后面，我

会留下足够的空间，提示出现在下一页纸，解决方案也是。当我在课堂上用这本书时，我一次只发几页，学生会把他们装在活页夹里。我的分页系统可以使得只包含问题，而没有给出提示和解决方案。有时我会把提示和问题折叠一起发给学生，由他们决定是否参考提示。如果你是单面印刷这本书，你可以忽略这些空白页，一样可以工作的。

这是一本免费的书，这意味着任何人都欢迎来阅读，复制，修改和重新发布，但是受许可证的限制。我希望人们可以发现这本书有用，但我更希望能继续帮助通过发送更正、建议和其他材料来开发他。感谢！

Allen B. Downey

Needham, MA

June 1, 2005

**第一章 介绍 Introduction**

**1.1 同步（Synchronization）**

在一般使用中，“同步”是指在同一时间发生两件事。在计算机系统中，同步会更一般：它指事件之间的关系--任意数量的事件，任何类型的关系（前，中，后）。

计算机程序员经常关心同步约束（**synchronization constraints**），它是与事件顺序相关的需求。举例包括：

序列化（**Serialization**）：事件A必须在事件B之前发生。

互斥现象（**Mutual exclusion**）：事件A和事件B不能同时发生。

在现实生活中，我们经常使用时钟（clock）来检查和执行同步约束。我们怎么知道A在B之前发生呢？如果我们都知道事件在什么时候发生，那么我们就能对比时间了。

在计算机系统中，我们经常需要在没有时钟的有利情况下来满足同步约束，或者因为没有通用时钟，或者是我们并不知道事件准确的发生时间。

这就是我们这本书的内容：利用软件技术来保证同步约束。

**1.2 运行模型（Execution model）**

为了能理解软件同步技术，你需要有一个计算机程序如何运行的模型。最简单的模型就是，计算机按顺序依次执行一条接着一条的指令。在这个模型里，同步是微不足道的；我们可以观察程序从而知道事件的顺序。如果语句A在语句B前，那么语句A会先执行。

有两种方式会把事情变得更复杂。一种是计算机是并行的（parallel），这意味着在同一时间会有多个处理器在处理。在那种情况下，是很难知道一条语句在这个处理器下是否会比另外一个处理器的语句先执行。

另外一种情况就是单个处理器正在跑着多个线程（multiple threads）。一个线程就是一个按顺序执行的指令序列。如果有多个线程，那么处理器就可能在一段时间内工作这个线程，然后切换到另外一个线程等等。

通常情况下，当每个线程运行时，程序员是无法控制的；操作系统（特别是调度器scheduler）会做出这些决定。因此，程序员不能指出语句在不同线程什么时候会执行。

同步的目的，对于并行模型和多线程模型是没有区别的。他们的问题都是一样的，在一个处理器里面（或者一个线程）我们知道指令执行的顺序，但是在多处理器（或者多线程）是很难说出他们的顺序的。

一个真实的例子可能会让你更加清楚。想象一下你和你的朋友Bob住在不同的城市，有一天，在晚餐时间，你开始想知道那天是谁先吃午饭，是你还是Bob，你会怎么知道？

显然你可以打电话给他问他什么时间吃的午饭。但是如果你是在你的时钟是11:59吃午饭，Bob在他的时钟是12:01吃的午饭呢？你可以确定是谁先吃的吗？除非你们两个都很小心保持精确的时钟，是吧？

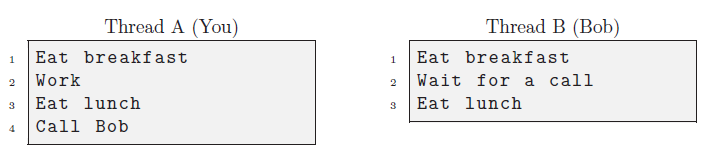
计算机系统也是面临这些问题，因为，即使它们的时钟通常很精确，但它们的精度总是有限的。此外，大多数的时候，系统并没有记录事情发生的时间。有太多的事情发生了，发生太快，不能记录下事件确切的时间。

谜题：假设Bob愿意遵循简单的指令，你是否有办法能保证明天你会比Bob先吃午饭？

（让Bob接到我的电话后才能吃午饭？）

**1.3 消息序列化（Serialization with messages）**

其中一个方法是命令Bot不能吃午饭直到我打电话给他，然后，保证你吃完午饭后在打电话给他。这个方法看似不重要，但是它所依赖的思想，消息传递（message passing），是各种同步问题的一种真实的解决方法。冒着明显的风险，考虑一下这个时间表。



第一列是你执行的动作，换句话说，就是线程的执行。第二列是Bob的执行线程。在一个线程内，我们可以说出事件的发生顺序。我们可以指出事件的顺序



其中a1 < a2 是指 a1 在 a2前执行。

通常，我们不能从不同的线程对比事件，比如我们不知道谁吃早餐先（a1 < b1 ?）

但是有消息传递（打电话）的话我们可以知道谁吃午餐先(a3 < b3)。假设Bob没有其他朋友，他不会吃午餐知道你通知他，所以 b2 < a4。结合所有关系，我们得到 b3 > b2 > a4 > a3 。这样就能证明你比Bob更早吃午饭。

在这种情况下，我们可以说你和Bob是顺序（sequentially）的吃午饭。因为我们知道事件的顺序，而你们吃早餐是有可能并发的（concurrently），因为我们不知道。

当我们讲并发事件，通常是说他们同一时间发生，或者同时的。作为速记，这很容易，你知道记住严格的定义：

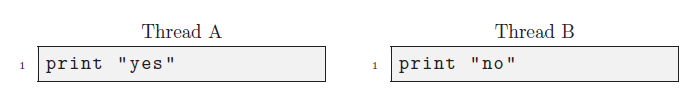
Two events are concurrent if we cannot tell by looking at the program

which will happen first.

有时，当程序结束后，我们可以说出哪个先发生，但同时是不能的。就算我们可以，也无法保证下次还能得到相同的结果。

**1.4 非确定性（Non-determinism）**

并发程序通常是不确定性的（non-deterministic），意味着不能通过查看程序来判断出当他执行时会发生什么。下面是一个简单的非确定性程序：



因为两个线程是并发的，执行的顺序由调度器决定。在任何程序运行期间，输出的结果有可能是“yes no”或者“no yes”。

非确定性是其中一样使得并发程序难以调试。一个程序可能连续正常工作1000次，但又可能在1001次运行就崩溃了，取决于调度器的调度特性。

这种类型的bug是基本很难在测试中发现的，只能通过仔细编程来避免。

**1.5 共享变量（Shared variables）**

大多数时候，大多数线程里的变量都是局部变量，意味着这些变量只属于自己的唯一线程，其他线程是不可以访问到的。除非是这样，这样就不存在同步的问题了，因为线程没有互动。

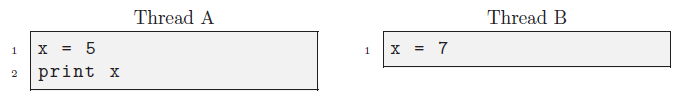
但通常有一些变量是在两个甚至是多个线程之间共享的（shared）。这是线程交互的其中一种方法。比如线程之间传递消息的一种方式就是一个线程读取另外一个线程写入的值。

如果线程没有同步（unsynchronized），我们不可能通过查看程序知道读者是否读取到写者所写的值 还是 已经存在的旧值。因此很多程序强制读者不能读取直到写者写入了值。这正是在1.3节中序列化的问题。

另外一个线程交互就是并发写入（两个或以上的写者）和并发更新（两个或多个执行读取然后更新）。下面两个小节讨论这些交互。共享变量的其他可能使用，如并发读取，是不会引发同步问题的。

**1.5.1 并发写（Concurrent writes）**

在下面的例子，x是共享变量可以被两个写者进行访问。



变量x会打印出什么呢？如果这些语句都有执行，那么x最终的值会是什么呢？它取决于这些语句的执行顺序，被称为 执行路径（execution path）。其中一种路径是 a1 < a2 < b1，这种情况下程序的输出是5，但是最终的值会是7。

问题：哪种执行路径会输出5而且最终值也是5？ (b1 < a1 < a2)

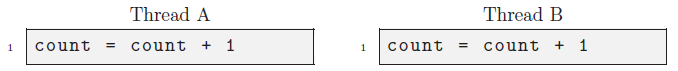
问题：哪种执行路径会输出7而且最终值也是7？ (a1 < b1 < a2)

问题：有没有一种执行路径输出7而最终值是5的呢？你可以提供吗？(没有，既然已经输出7了，那么肯定是 b1 < a2，而a1 < a2，所以只有两种可能，要么是 a1 < b1 < a2，要么是 b1 < a1 < a2，所以不可能出现题目这种情况。)

回答这种问题是并发编程（concurrent programming）中很重要的一部分：有哪些途径是可能的，哪些是有影响的。我们可以证明给出的（想要的）效果是否必须的或者不想要的效果是否不可能的吗？

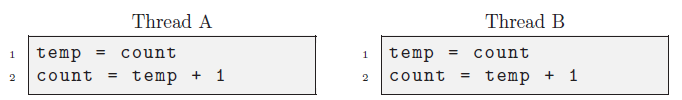
**1.5.2 并发更新（Concurrent updates）**

更新是一种读取变量的值，根据旧值计算出新值，并写入新值的操作。最常见的更新操作是增加（increment），其中新值等于旧值加一。下面的例子，显示一个共享变量 count ，被两个线程进行并发更新。



第一眼看上去，不会觉得有明显的同步问题。这里只有两个执行路径，而且他们产生的结果都相同。

问题是这些操作都会在执行前被翻译成机器语言，而且在机器语言里，这些更新操作都执行两个步骤，读取和写入。如果我们把代码用一个临时变量temp来重写的话，问题就显然易见多了。



现在考虑下面这种执行路径 a1 < b1 < b2 < a2。

假设x的初始值为0，那么最终值会是什么呢？因为两个线程都读取相同的初始值，他们写入相同的值。变量只会进行一次增加，这就会造成和程序员想的不一样了。

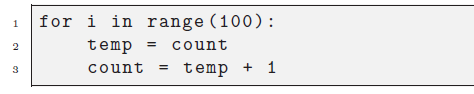
这种问题是很微妙的，在查看高级语言过程中它并不总是可能发现的，因为在单步执行的操作中他们有可能是被中断。事实上，一些计算机可以由硬件提供加法指令从而使得不会被中断。一个操作不能被中断可以称为 原子的（**atomic**）。

因此，如果我们不知道操作是否原子的，那我们怎么去写并发程序呢？其中一种可能是在每个硬件平台上搜集关于每个操作的具体信息。这种方法的缺点是显然易见的。

最常见的选择就是做出保守的假设所有更新操作和写入操作都不是原子的，使用同步约束来控制并发访问共享变量。

最常用的约束是使用互斥（mutual exclusion， mutex），我在1.1节提过的。互斥保证了同一时间只有一个线程可以访问共享变量，这样就消除了本节所提出的同步错误了。

问题：假设有100个线程并发运行下面的程序（如果你不熟悉Python的话，下面的for循环执行更新100次）



当所有线程执行完后，count的最大值会是什么，最小值呢？

（最大值：每个线程更新100次，100个线程总共更新 100\*100 次，所以等于10000；

最小值：所有线程去读同一个count值后再做更新操作得100，这个不对，）

**1.5.3 消息互斥（Mutual exclusion with messages）**

像序列化（串行 serialization），互斥也可以用于传递消息。比如，想象你和Bob一起远程进行监控核反应堆。大多数时候，你们两个都监控着警告灯，但是你们都可以休息去吃午餐。谁先去吃午餐都可以，但重要的是不能同时一起去吃午餐而留下无人看管的核反应堆。

问题：找出一个消息传递系统（电话）可以强制执行这些约束。假设没有时钟，你也不能预测午餐什么时候开始和持续多长。这个系统需要最小的消息数量是多少？（？）

**第二章 信号量 Semaphores**

在现实生活中，信号量（semaphore）是一种用于可视通信的信号系统，通常使用旗子，灯或者其他机械装置。在软件总，信号量是一种数据结构，用来解决各种同步问题。

信号量是由Edsger Dijkstra发明的，他是一个著名的计算机科学家。从最初的设计开始，有些细节已经改变了，但是基本思想是一样的。

**2.1 定义（Definition）**

信号量和integer整数相似，有3点不一样：

1. 当你创建信号量后，你可以初始化为任意整数值，但之后，只允许你执行增量（增加一）和减少（减少一）的操作。你不能读取信号量的当前值。
2. 当一个线程减少一次信号量，如果结果变成负数，线程会被阻塞不能执行知道其他线程线程增加信号量。
3. 当线程增加信号量，如果有其他线程正在等待，其中一个等待线程会被停止阻塞（唤醒）。

当我们说一个线程被自己阻塞（blocks itself）这意味会通知调度器这个线程不能执行下去。调度器将阻止线程运行直到让线程被解封的事件发生。在计算机科学传统的隐喻中，被解封（unblocking）通常叫做唤醒（waking）。

这就是定义的全部，但也有一些定义造成的后果需要你进行考虑。

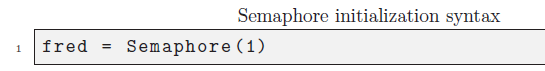
* 通常，我们不知道一个线程减少信号量后会是阻塞还是没有（在一些特定情况下，你可能知道会还是不会）。
* 在一个线程增加信号量和另一个线程被唤醒后，两个线程会继续并发运行。没有办法知道会是哪一个线程先运行。
* 当你发送信号量的时候（signal a semaphore），你不知道是否有其他线程正在等待，因此被阻塞的线程有可能是0个也有可能是1个。

最后，你可能需要考虑信号量的值是什么意思。如果值是正的（positive），这就表示线程可以进行减少操作而不不会阻塞。如果值是负的（negative），着就表示线程需要阻塞等待。如果值是0，这就意味着没有线程在等待，但如果有线程尝试减少，就会被阻塞。

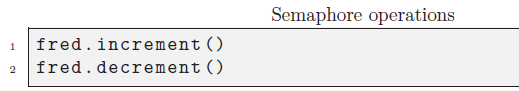
**2.2 语法（Syntax）**

在大多数编程环境中，信号量的实现已经成为操作系统的一种编程语言。不同系统的实现提供不同的功能，通常需要不同的语法。

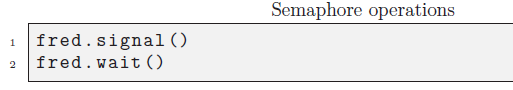
在这本书中，我会用一种简单的伪代码来展示信号量是怎样工作的。创建新的信号量和初始化的语法如下：

函数Semaphore是一个构造函数，它创建和返回新的信号量。信号量的初始值由构造函数的参数传递进去。

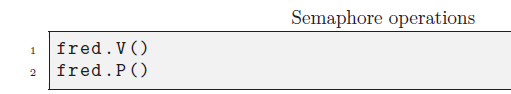
信号量的操作在不同的环境中以不同的名称进行，最常见的选择是



或



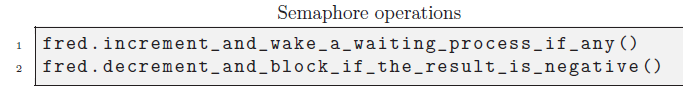
或



你可能会对有这么多名字感到惊讶，但这是有原因的。其中increment和decrement描述操作是做什么。Signal和wait描述它们通常用来做什么。P、V原名称是由Dijkstra提出的，他明智的意识到一个没有意义的名字比一个有误导性的名字更好。

我认为另一对具有误导性因为它是递增和递减，忽略了阻塞和唤醒的可能性，以及信号量与通常用的信号和等待没有关系。

如果你坚持使用有意义的名字，我建议使用这些：



我不认为世界很快会接受这两个名字。与此同时，我选择（或多或少）的使用signal和wait。

**2.3 为什么信号量 Why semaphores?**

看一下信号量的定义，不是很明显为什么它们这么有用。我们确实不需要使用信号量来解决同步问题，但使用它们也有一些好处：

* 信号量加强了刻意的约束，帮助程序员避免错误。
* 使用信号量的解决方案通常是整洁有序的，使得容易展示它们的正确性。
* 信号量可以在许多系统上高效的实现，因此使用信号量的解决方案更具有可移植性和通常更高效。

**第三章 基本同步模式 Basic synchronization patterns**

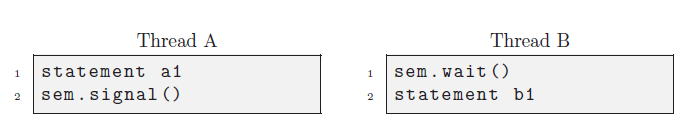
这一章提出一系列的基本同步问题，并给出一些使用信号量来解决问题的方法。这些问题包括序列化（serialization）和互斥（mutual exclusion），这些我们都看过了，此外还有其它的方法。

**3.1 信号 Signaling**

或许使用信号量最简单的就是发信号了singnaling，这意味着一个线程发送一个信号给另外一个线程，指示有某件事发生了。

发送信号可以保证一个线程中的一段代码比另外一个线程中的代码先运行，换句话说，它解决了同步的问题。

假设我们有一个信号量命名为sem，初始值为0，线程A和线程B都可以访问它。



单词statement（语句）表示任意的程序代码。为了是例子更具体，想象a1从文件读取一行，b1在屏幕显示一行。信号量在这个程序中确保线程A在线程B开始执行b1前执行完a1。

这是它如何工作的：如果线程B首先执行wait语句，会发现信号量初始值为0，然后被阻塞。接着线程A发送信号，线程B开始执行。

同样的，如果线程A先执行发送信号，信号量的值就会增加，当线程B到达wait语句，会立即执行。不管如何，a1和b1的执行顺序都会得到保证。

这种信号量的使用是基础是signal和wait，在这种情况下，名字是很方便记忆的。不幸的是，我们将会看到在其他情况下，这种名字是没那么有用的。

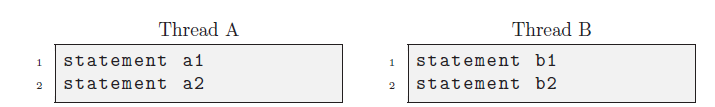
说道有意义的名字，sem并不是。在可能的情况下，给信号量一个能代表它是什么的名称是一个好主意。在这种情况下，名字 a1Done 可能比较好，因此 a1done.signal() 意味着 “signal a1 is done”，然后 a1done.wait() 意味着 “wait until a1 is done”。

**3.2 Sync.py**

TODO：没看明白

**3.3 Rendezvous**

谜题：



我们想要保证a1在b2之前执行，b1在a2之前执行。

Thread A：

Statement a1;

A1done.signal();

B1done.wait();

Statement a2;

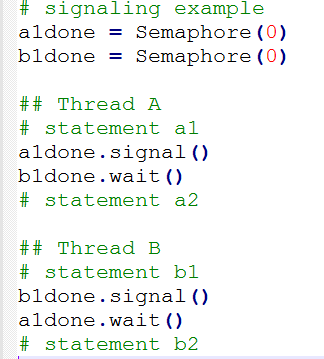
Thread B

Statement b1;

B1done.signal();

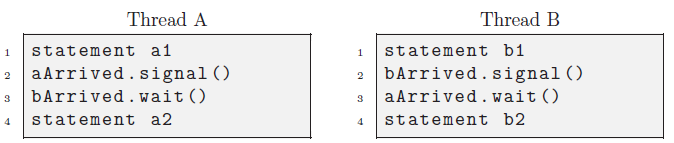
A1done.wait();

Statement b2;

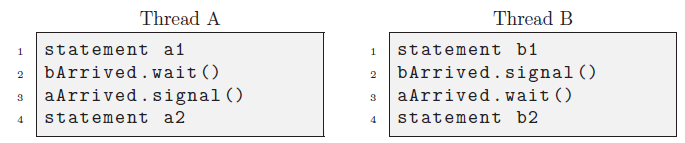


**3.3.2 Rendezvous 解决方法**

这个是我的解决方法：



你也可以尝试如下：



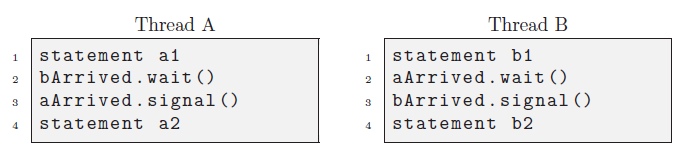
这种方法也可以工作，但是可能没有那么高效，因为他可能需要在A和B之间切换一次以上。

如果A到达先，它会等待B。当B到达后，它会唤醒A然后立即执行wait，这样会让它阻塞，等A运行到signal后，两个线程才会继续执行。

尝试思考其他可能的方式，并说服自己在所有情况下，两个线程都不能执行下去，直到两个线程都到达。

**3.3.3 死锁 Deadlock #1**

同样，在上一个问题，你可能会尝试下面的方式：



如果是，我希望你能快点拒绝（rejected），因为它有严重的问题。假设A先到达，它会则色等待wait。当B到达后，它也会阻塞，由于A不可能发送信号aArrived。在这种情况下，两个线程都不能执行，永远都不会。

这种情况叫做死锁（deadlock），显然的，它不是一种成功解决同步问题的方式。在这种情况下，错误是很显然易见的，但是通常死锁的可能性会更加难发现。我们会在后面看到更多的例子。

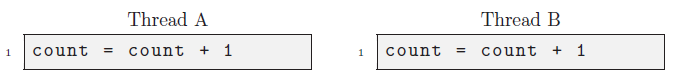
**3.4 互斥 Mutex**

第二种常用的信号量就是强制互斥。我们已经见过一种使用互斥的方式了，控制并发访问共享变量。互斥保证了在同一时间只有一个线程可以访问共享变量。

互斥（mutex）就像一个记号（token）从一个线程传递到另一个线程，只允许一个线程执行。比如，在《Lord of the Files》一组的孩子用贝壳当做互斥量。为了能说话，你需要拿着贝壳。只有一个孩子能握着贝壳，只有一个能说话。

同样的，一个线程如果需要访问共享变量，它需要先获取互斥量。当访问完后，释放互斥量。同一时间只能有一个线程获取互斥量。

问题：在下面的例子添加信号量强制互斥量来访问共享变量count。



Sem = Semaphore(1)

Thread A:

Sem.wait()

Count = count + 1

Sem.signal()

Thread B:

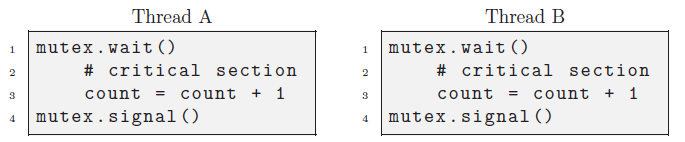
Sem.wait()

Count = count + 1

Sem.signal()

**3.4.2 互斥量解决方法**

这里是解决方法



由于mutex最开始为1，因此当第一个线程到达wait后会立即进行处理。当然，wait信号量的动作和相减一致，因此第二个线程到达后会等待直到对个线程发送信号。

我已经对更新操作进行了缩进，以显示它包含在互斥锁内。

在这个例子里，两个线程都运行同样的代码。这是有时被称为对称的（symmetric）解决方案。如果线程需要运行不同的代码，解决方案是不对称的（asymmetric）。对称解决方案比较容易生成。在这种情况下，互斥量解决方法能不修改就处理任意数量的并发线程。只要每个线程在执行更新前先等待锁（wait），更新后释放（signal），这样就不会有两个并发线程访问到count了。

需要被保护的代码段通常被称为临界区（critical section），我认为是因为防止并发访问是非常重要的。

在传统的计算机科学和混合隐喻，人们谈论互斥是有好几种方法的。在我们目前使用的隐喻中，互斥量是一个从一个线程传递到另一个线程的令牌（token）。

在另一个比喻中，我们把临界区想象为一个房间，同一时间只允许一个线程进入这个房间。在这种比喻下，mutex被称为锁，线程在进入前锁住互斥量，出来后解锁互斥量。不过，人们偶尔也会混合在一起谈论“获得”和“释放”锁，虽然没有多大意义。

这两种隐喻都有潜在的用处，但也有误导性。当你做下一个问题时，尝试用这两种方式思考，看哪种方式让你得到解决方案。

**3.5 Multiplex**

问题：扩展之前的解决方法，使得允许多个线程在同一时间进入临界区，但是它强制并发线程数的上限。换句话说，不超过n个线程在同一时间进入临界区。

这种模式称为 multiplex（复用）。在现实中，复用问题出现在繁华的夜总会上，那里每次只允许容纳建筑最多数量的人，为了维持安全和创造排他性的幻觉。

在这种地方，保镖通常强制执行同步约束，通过追踪里面的人数，当房间达到人数容量后禁止他人进入。当有一个离开后才允许另一个人进入。

用信号量来执行这种约束听起来可能很困难，但其实是很微不足道的。

Mutex = Semaphores(n)

Thread 1 ... N:

Mutex.wait()

# critical section

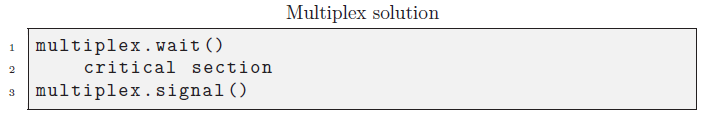
Mutex.signal()

**3.5.1 Multiplex Solution**

允许多个线程进入临界区，只需要初始化信号量为n，n是允许的线程最大数量。

在任何时候，信号量的值表示允许可以额外进入的线程数量。如果值为0，下一个线层就会被阻塞直到里面线程退出和发信号。当所有线程退出后，信号量的值恢复为n。

由于解决方法是对称的（symmetric），非常方便展示只有一份的代码。但你需要想象多个并发的线程在跑着多份复制的代码。



如果临界区被占了而且多个线程到达，会发生什么。当然，我们希望所有到达的线程会等待。这个方案就是这样做的。每次到达的线程进入到队列，信号量减少，因此信号量（负值）表示了在等待队列中的线程数。

当一个线程离开时，它发信号给信号量，增加它的值和允许其中一个等待的线程去执行。

在想一次隐喻，在这种情况下，我发现把信号量想象为一系列的令牌（token）比想象成锁更有帮助。当每个线程调用wait时，它捡起其中一个令牌；当它调用signal是，它放下令牌。只有线程握有令牌时才允许进入房间。如果当线程来时没有令牌了，它需要等待直到另一个线程释放令牌。

在现实生活中，票务窗口有时也使用类似的系统。他们向客户发放令牌（有时是扑克筹码）。每个令牌允许持有者购买一张票。

**3.6 Barrier**

在思考一下之前3.3章节的Rendezvous问题。其中我们展示的方法有一个限制，就是不能工作在两个以上的线程。

问题：扩展rendezvous解决方法，每个线程应该运行下面的代码：



同步要求是当所有线程都执行rendezvous后，才能开始执行critical point。

你可以假设有n个线程，这个n值存储在一个变量n中，所有线程都能访问这个变量。

当第一个n-1线程到达后它们需要阻塞直到第n个线程到达，这时所有线程都可以执行。

Mutex = Semaphore(1)

Barrier = Semaphore(0)

Count = 0

Thread:

# rendezvous

Mutex.wait()

Count += 1

Mutex.signal()

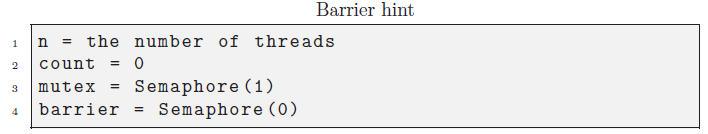
if count == n:

Barrier.signal()

Barrier.wait()

# critical point

**3.6.1 Barrier hint**

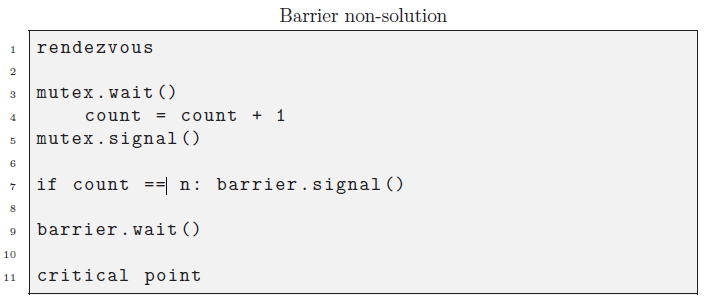


Count跟踪有多少个线程到达。Mutex保护互斥访问count，所以线程可以安全增加count值。

Barrier会锁住（等于0或负数），直到所有线程到达后，才释放锁（1或者以上）。

**3.6.2 Barrier non-solution**

首先我会展示一个不太正确的方法。因为对于解释错误的方案和理解哪里出了什么问题是非常有帮助的。



由于count有mutex来保护，计算所有执行的线程。首先的n-1线程当到达barrier由于锁，会等待。直到第n个线程到达，barrier会解锁。

问题：这个方法有什么问题？

**3.6.3 死锁 Deadlock #2**

问题是死锁。

作为一个例子，想一下当n=5时，4个线程正在等待barrier。信号量的值为在队列中的线程数，是-4。

当第5个线程给barrier发送信号，其中一个等待的线程允许执行，信号量增长到-3。

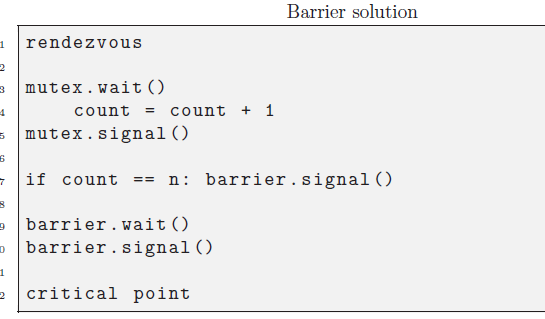
但是之后再也没有给信号量发送信号了，没有一个线程可以通过barrier。这是第二个死锁的例子。

问题：这个代码会一直创建死锁吗？你可以找到一个不会引起死锁的执行路径吗？

问题：解决这个问题。

**3.6.4 Barrier解决方法**

最终，这是一个可以工作的barrier：



唯一的改变就是在等待barrier后发送信号。现在每个线程通过，发送信号给信号量，因此下一个线程可以通过。

这种模式，wait和signal连接不断，经常发生，以至于它有一个名字，称作 旋转门（turnstile），因为它一次只允许一个线程通过，它可以被锁住禁止所有线程。

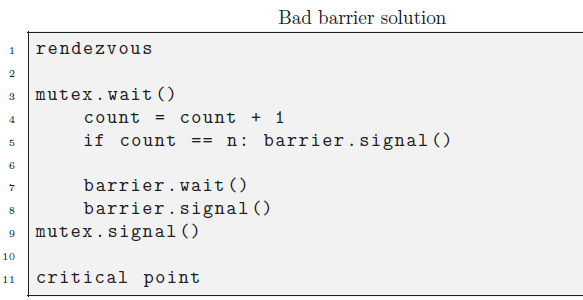
在它的初始状态（0），旋转门是锁住的。第n个线程解锁它，然后所有n个线程通过它。

如果在mutex之外读取count的值可能是危险的。在这种情况下是没有问题的，但通常来说不是一个好主意。我们会在几页后移除它，但在这时，你可能需要考虑这个问题：第n个线程后，旋转门的状态是什么？有没有一种可能barrier发送信号超过一次？

（首先两个信号量mutex和barrier，mutex初始为1，barrier初始为0。Mutex是保护count的，这个我理解。假设n为5,5个线层，前面四个线程执行到barrier.wait()后，需要等待，barrier的信号量值为-4，当第5个线程通过后，count=5，然后barrier.signal()发信号，此时barrier信号量为-3。因为加1了，所以唤醒其中一个等待的线程，比如第一个线程，然后它接着执行，发送信号，barrier信号量变为-2；接着唤醒第二个线程，发送信号，barrier信号量变为-1；接着唤醒第三个线程，发送信号，barrier变为0；接着唤醒第4个线程，发送信号，信号量变为1；然后第5个线程执行barrier.wait()，信号量减为0。继续执行，发送信号，barrier变为1。所以第n个线程后，旋转门的状态是打开的，即1。）

**3.6.5 死锁 Deadlock #3**

由于一次只能有一个线程通过互斥量，和一次只有一个线程通过旋转门，可能足够理由把旋转门放到互斥量里面，像这样：



这会变成一个坏主意因此会引起死锁。

想象一下第一个线程进入mutex后，然后到达旋转门处被阻塞。由于互斥量被锁住了，没有其他线程可以进入，因此条件标量，count == n 永远不会变真，没有人会解锁旋转门。

在这种情况下，死锁是相当的明显，但它展示了一种非常普遍的死锁原因：得到互斥锁后在信号量处被阻塞（blocking on a semaphore while holding a mutex）。

**3.7 Reusable barrier**

通常一组协作的线程会在循环中执行一系列的步骤，每步后会在barrier(栅栏???)进行同步。在这个应用里，我们需要一个可以重复使用的barrier，它会在所有线程通过后锁住自己。

问题：重写barrier解决方法，在所有线程通过后，旋转门会再次锁上。

Mutex = semaphore(1)

Barrier = semaphore(0)

Mutex.wait()

Count = count + 1

Mutex.signal()

If count == n:

Barrier.signal()

Barrier.wait()

Barroer.signal()

# ...

Mutex.wait()

Count = count - 1

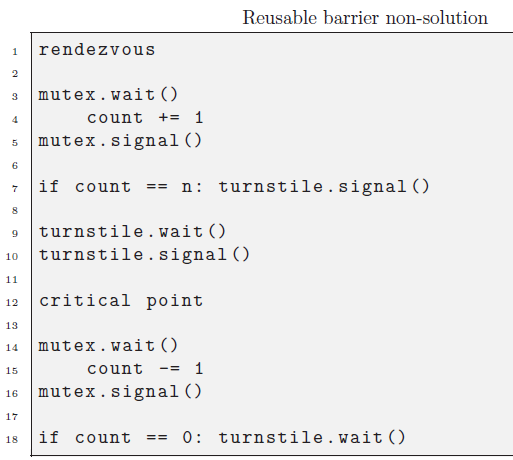
Mutex.signal()

If count == 0:

Barrier.wait()

**3.7.1 Reusable barrier non-solution #1**

再次，我们会开始简单的企图解决方法，然后慢慢改进：



注意在旋转门之后的代码和之前的代码很相似。再次的我们需要互斥量来保护访问共享变量count。可惜的是，这个代码不太正确。

问题：有什么问题？

**3.7.2 Reusable barrier problem #1**

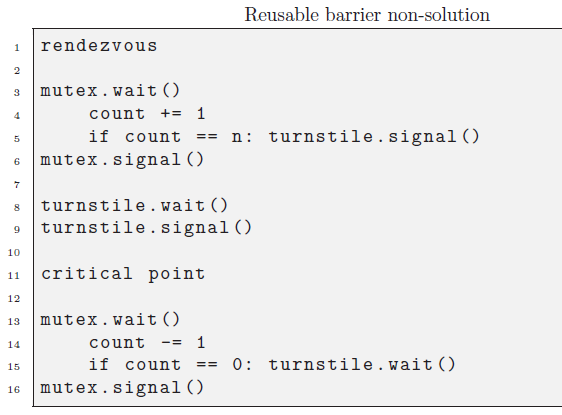
上面的代码在第7行有个问题点。如果第n-1个线程在这里被中断了，然后第n个线程通过互斥量过来了，两个线程都会发现 count == n，然后两个线程都会发送信号给旋转门。事实上，甚至有可能所有的线程都发送信号给旋转门。

相似的，在第18行代码出，有可能会有很多线程在等待wait，这样会引起死锁。

问题：解决这个问题。

**3.7.3 Reusable barrier non-solution #2**

这个尝试修复上次的错误，但还有一个细小的问题存在。



在两种情况下，检查都在mutex里面，因此线程不会在检查前和改变counter后被打断。

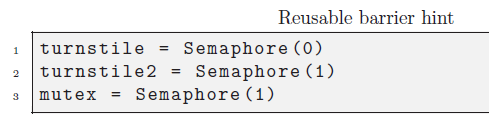
可悲的是这份代码还是不正确。记住这个barrier会在循环里面，因此，在执行完最后一行代码后，每个线程都会走回到rendezvous。

问题：定位和解决问题。

**3.7.4 Reusable barrier hint**

正如目前所写的那样，这个代码允许早完成的线程通过第二个mutex，然后在循环和通过第一个mutex和旋转门，比其他线程更快的领先一圈。

为了解决这个问题，我们用两个旋转门。



最初那个是锁住的而第二个是打开的。当所有线程到达后，我们关上第二个打开第一个。当所有线程到达第二个后我们在锁住一个，这样会安全使所有线程循环到开始，然后打开第二个。

Mutex.wait()

Count += 1

If count == n:

Turnstile2.wait()

Turnstile.signal()

Mutex.signal()

Turnstile.wait()

Turnstile.signal()

Mutex.wait()

Count -= 1

If count == 0:

Turnstile.wait()

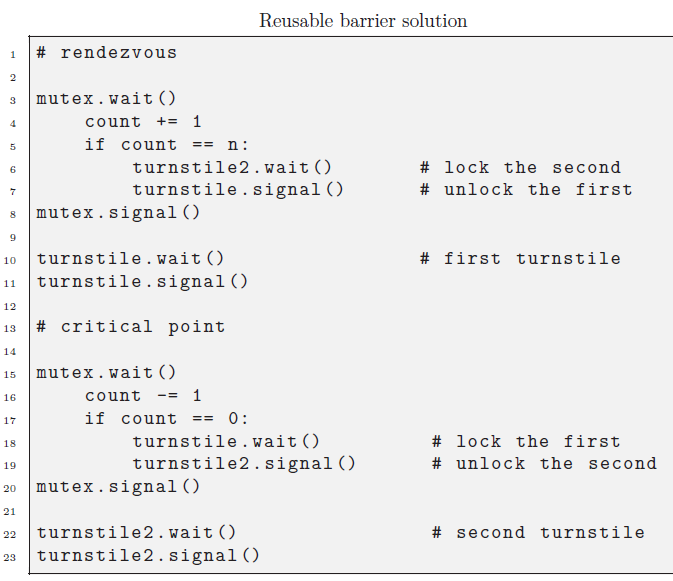
Turnstile2.signal()

Mutex.signal()

Turnstile2.wait()

Turnstile2.signal()

**3.7.5 Reusable barrier solution**



这种方法有时叫做 two-phase barrier 因为它强制所有线程等待两遍：第一次让所有线程到达，第二次让所有线程执行临界区（critical section）。

不幸的是，这个解决方法是大多数典型的非平凡（non-trivial）同步代码：这是很难确定这个方法是否正确的。通常存在一种微妙的方法通过程序特定的路径可以导致错误。

会让这个问题变得更糟的是，测试这个方法的实现不会有太多的帮助。错误可能会极少发生因为产生特定的执行路径可能需要极其不幸的组合情况下。这种错误几乎是不可能通过传统的方法来重现和调试的。

唯一的选择是非常仔细的检查代码和“证明”他是正确的。我把“证明”用双引号表示是因为并不是说，你需要写下正式的证明（尽管有狂热者鼓励这种行为）。

我心里想的那种证明比较随便。我们可以使用关于代码的结构，以及我们开发的习惯用语，然后演示一些关于程序中的中级声明（intermediate-level claims）。比如：

1.只有第n个线程可以锁住和解锁旋转门。

2.在线程可以解锁第一个旋转门前，需要先把第二个旋转门锁上，反之亦然；因此一个线程是不可能比其他线程先通过另一个旋转门。

通过找到合适的语句来断言（assert）和证明，你有时会找到简洁的方法来说服你自己（或者疑惑的同事）你的代码是没问题的。

**3.7.6 Preloaded turnstile**

一件关于旋转门的好事就是它是一个万能通用的组件，你可以在各种解决方案中使用。但它的一个缺点是强制线程有序的执行通过，这样可能会使得更过的上下文切换。

在reusable barrier解决方法里，我们可以简化方法，如果