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**

**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？

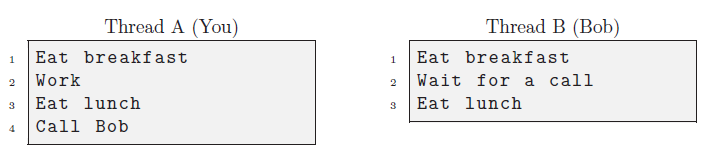
计算机系统也是面临这些问题，因为，即使它们的时钟通常很精确，但它们的精度总是有限的。此外，大多数的时候，系统并没有记录事情发生的时间。有太多的事情发生了，发生太快，不能记录下事件确切的时间。

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：

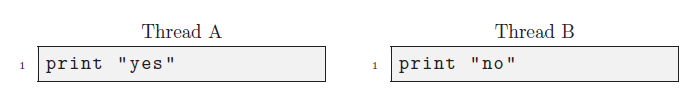
Two events are concurrent if we cannot tell by looking at the program

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）**