The Art of Multiprocessor Programming 读书笔记 (更新至第 3 章)

郑思遥*

2013 年 10 月 15 日

摘要

这份笔记是我 2013 年下半年以来读 “The Art of Multiprocessor Programming” 这本书的读书笔记。目前有关共享内存并发同步相关的书籍并不多，但是学术文献却不少，跨越的时间范围也非常长，说明人们一直在做出努力。这本书是这个领域的好书，作为一本好书，它总结了这个领域自发展以来的大量重要成果，介绍了共享内存同步的基本理论，并介绍了大量并发算法和数据结构（主要是无锁算法），包括并发队列、栈、链表、计数器、排序网络、散列、跳表、优先队列等。更为重要的是，本书的作者之一 Maurice Herlihy 就是并发同步领域的泰斗级人物，本身提出了无等待同步的基本理论，还提出了不少重要的无锁算法，因此这本书的权威性和重要性毋庸置疑。为了加深自己对重要概念的理解，同时受到 Hawstein[1]的 “把《编程珠玑》读薄” 这篇博文[2]的启发和刘未鹏[3]《暗时间》的影响，我决定用自己的语言把这本书的重要内容复述一遍，其中也加入了自己的一些想法和理解，希望这份笔记能对这本书的其他读者有帮助，也希望能对所有对共享内存同步的朋友们有帮助。由于我才疏学浅缺乏经验，因此其中必定有不少谬误，望各路大牛批评指正多多交流。

目前整本书我已经读完一大半，这份笔记将持续更新，直到完全覆盖所有的 18 章。

目录

*zhengsyao@gmail.com, http://weibo.com/zhengsyao/

[1] http://weibo.com/hawstein

[2] http://hawstein.com/posts/make-thiner-programming-pearls.html

[3] http://weibo.com/pongba

形式化地描述共享内存同步中的互斥问题。介绍历史上一些经典的互斥算法，从最简单的两个会死锁的 2-线程互斥算法开始，然后介绍了不会死锁的 *Peterson* 算法，并且将 *Peterson* 算法泛化为支持 n 线程的过滤器锁（*Filter Lock*）算法。然后再进一步引入了支持公平性的面包店算法（*Bakery Lock*）。最后讨论了为什么这些经典算法现在不实用，因为缺乏强大的原子操作。

这一章介绍了并发数据结构基本理论。并发数据结构就是能够允许多个线程安全并发访问的数据结构以及相关的算法。并发数据结构的基本理论从两个方面探讨了数据结构在并发访问中是否安全：一个是正确性（*correctness*），定义了各种强度不同的一致性（*consistency*），在并发数据结构的设计中，要确保多线程并发访问能满足所需的一致性；另一个是进展（*progress*），也就是说多个线程并发访问数据结构的时候各个线程产生进展的情况。如果所有线程都不产生进展，则说明发生了死锁。这一部分还加上了对多核硬件的背景介绍，包括 *cache* 相关的问题和内存一致性的问题。

1 关于这本书

处理器设计的发展趋势是在单个处理器上集成越来越多的核心（core）。几年前个人计算机就开始流行双核处理器，现在个人计算机（笔记本电脑）上四核以上的处理器也不少见。服务器市场就更不用说，四核、八核的处理器已经是主流，而且通常还采用了超线程技术，一个核上跑两到四个硬件线程。在服务器上为了提升计算密度，普遍采用多处理器的架构，通过处理器的互联技术将多个处理器连接起来。多个处理器的多个核心共享内存，操作系统对这些核心进行统一的管理。应用程序开发者看到的就是一大堆逻辑处理器。

摩尔定律一直没有失效，只是处理器设计师不再把精力放在设计更复杂的核心上，而是鼓励应用开发更多地挖掘应用程序本身的并行性，通过多个核心的真正并行执行提升性能。在此之前的处理器设计中提升处理器执行性能的方法主要包括提升处理器主频、通过乱序执行提升处理器吞吐、通过多流水线技术提升指令集的并行度等方法。对于应用开发者来说，这些技术的采用可以“自动”地提升应用程序性能。应用开发者不需要对应用程序做任何优化，只需要换上新一代的处理器就可以获得更好的运行性能。然而随着单个核心越来越复杂，处理器功耗也越来越高，同时发热量也越来越大。主频的提升更是直接关系到功耗的提升。人们为功耗提升额外付出的电力和散热成本已经远超这些提升带来的性能提升，所以处理器设计的关注点转移到了多核处理器。设计师们开始关注如何在一块处理器上集成越来越多的核心，如何设计高效的网络连接这些核心，如何解决这些核心之间的 cache 一致性问题等等。

当核心数目很多的时候，我们就不叫这个处理器多核处理器了，而是取了另外一个名字，叫众核处理器（many-core processor）。我认为众核处理器和多核处理器相比不仅是核心数增多了的区别，更重要的是还要包括内部众多核心之间的互联技术（片上网络）。现在众核处理器已经不仅限于实验室的研究型产品了，有了一些商用的产品，例如 Intel 的 Xeon Phi 众核协处理器，包含 60 个左右的小 x86 核心，以及 Tilera 的 TILE 系列处理器，最顶级的产品包含了 72 个 MIPS 核心。

从应用开发的角度看，应用开发者可以将多处理器系统或众核处理器系统统统看作是多核处理器系统，因为操作系统将这些系统都当做共享内存式（shared-memory）的多处理器系统。Linux 在这些系统上运行的都是 SMP 版本的内核。

这本书的主要内容就是讨论如何在多核处理器上编写通过共享内存进行通信的程序（尽管这本书的标题依然是不够“潮流”的多处理器程序设计的艺术，但是基本的技术和理论和多核处理器是相同的）。

多处理器程序设计的挑战在于由于真正存在多个硬件执行单元（处理器或核心），所以多个线程会真正并行地执行，这些同时执行的线程之间会访问共享的数据，这些数据的访问需要正确地同步。而且每一个线程的执行在本质上是异步的，因为每一个线程的执行都有可能会被各种情形打断，例如 cache 缺失、缺页错误、中断和抢占等。这些执行的中断都是不可预期的，而且持续的时间也不同，例如 cache 缺失可能会产生不到 10 条指令的延迟，而缺页错误则会产生数百万条指令的延迟，如果线程被操作系统抢占了，那么延迟就更不好估计了。

这本书的第一部分是原理部分，讨论的是可计算性的问题，即异步并发环境中什么是可计算的。研究可计算性的意义在于：帮助识别哪些共享对象是无法通过任何并发算法实现的。有了这个理论基础的指导，我们就可以知道如何通过设计硬件机制和软件机制来解决并发的问题了。

在讨论可计算性问题的时候最重要的工具是讨论程序的正确性（correctness）。对于多处理器程序来说，正确性的讨论分为两个同等重要的方面：

- 安全性（safety）属性：就是说不能发生“坏事情”。尽管多线程的程序的线程会有多种多样的交错执行的情况，但是绝对不能破坏数据的完整性，绝对不能产生错误的结果。
- 活性（liveness）属性：表示“正确的事情”最终一定能发生。

本书第一部分理论部分的终极目标就是讨论如何推导并发程序的正确性。

本书的第二部分是实践部分，讨论现实世界中的各种并发数据结构和并发算法。

2 Chapter 1: Introduction

2.1 各种问题

本章概述了多线程编程使用共享内存作为信息交换的各种问题。首先用一个共享计数器的例子说明共享对象同步的重要性。为了引出这个共享计数器，书中编造了一个场景：在一台支持 10 个并发线程的并行计算机上找出 1 到 10^{10} 之间所有的素数。如果将这 10^{10} 个素数判定任务均摊到 10 个线程，每个线程处理 10^9 个显然是不合理的，因为素数在这些区间里面的分布是不均匀的，而且素数越大判定的时间需求也越大（暂且不考虑用什么判定算法），所以每一个线程的负载是非常不均衡的，任务最轻的线程完成任务之后只能干等任务重的线程完成任务，计算资源浪费。那么这就要引出共享计数器了：10 个线程共享一个计数器，每个线程每次领一个要判定的数，并且把计数器的值添加 1，这样下一个线程就可以领到下一个数，通过这种方式保证所有线程都保持忙碌。

接下来就关注这个共享计数器。下面的代码展示了这个计数器的功能，getAndIncrement() 方法的作用是领取一个数，然后增加计数器。但是这个计数器由于没有对共享变量进行保护，因此只支持单个线程。

```
1 public class Counter {
2     private long value; // counter starts at one
3     public Counter(int i) { // constructor initializes counter
4         value = i;
5     }
6     public long getAndIncrement() { // increment, returning prior value
7         return value++;
8     }
9 }
```

这个计数器的问题在于下面这行语句：

```
1 return value++;
```

这一行相当于下面这 3 行的操作：

```
1 long temp = value;
2 value = temp + 1;
3 return temp;
```

由于处理器通常只能在寄存器中做计算，所以任何内存中的值都需要加载到寄存器中（也就是 LOAD 操作）做计算再存储回内存中（也就是 STORE 操作），因此 value++ 一句话的操作变成了 3 个操作：加载原来的值，计算原来的值 $+1$ 并，最后将递增后的值写回内存。这三条语句如果是一个线程执行，那么没有问题，如果是多个线程同时执行，那么就会有问题。例如线程 1 读到 value 的值为 1，假设在线程 1 将更新后的值写入内存之前，线程 2 进入，那么线程 2 读到的 value 值也为 1，这样两个线程就领到了

同一个值。这里的讨论有一个基本的假设，就是读/写操作（即加载/存储）操作是原子的（atomic）。

这个例子说明的问题就是在多线程执行的环境中，共享的对象需要正确地同步才能保证整体不出错。对于这种计数器的问题，现代的硬件都提供了特殊的读 -改 -写（read-modify-write，RMW）指令，这种指令可以在一条指令内原子地读一个值，对这个值进行某种修改，然后再写入这个值。如果是一段代码需要原子地执行，那么这个问题称为互斥（mutual exclusion）问题。

应用开发者可能不会需要自己写互斥算法，只要调用库就可以了，但是理解了互斥算法可以加深对并发计算的理解。

这一小节通过一个现实的故事（是 1984 年 Leslie Lamport 在第 3 届 PODC 大会上受邀演讲中编的故事）让大家感受了一下解决互斥问题的时候需要考虑的几个问题：

- 是不是真的互斥了？这是互斥算法最基本的要求。
- 会不会死锁（deadlock）？死锁说明系统中出现了互相依赖的关系，整个系统都停滞了，永远都不会产生进度了。这是互斥算法和分布式系统设计中必须要避免的问题。
- 会不会饿死（starvation）？多个线程要求一定的公平性，不要出现有的线程特别可怜怎么都得不到执行的情况。

故事是这样的，Alice 和 Bob 是邻居，同住在一个大院里。Alice 养了一只猫，Bob 养了一只狗。两人都需要遛自己的宠物，但是猫狗不相容，不能同时出现在院子里（即两个宠物必须互斥地访问院子这个资源）。Alice 和 Bob 商定通过一种协作协议（coordination protocol）保证两个不能同时出现在院子里。

由于院子很大，所以 Alice 要遛猫的时候不能直接查看院子里有没有狗，也不能跑到 Bob 家去看，也不能在院子里大喊。这些方式对应的都是不可靠的信道。所以 Alice 想了一招，Alice 和 Bob 都在对方的窗台上放置一些罐子，用绳子绑在罐子上，并且将绳子牵到自己家，这样就可以在自己家拉绳子把对方窗台上的罐子拉倒。这是一种可靠的信道，可以确保对方一定能看到自己发出的信号。就好像共享内存式的同步一样，一个线程写入共享变量之后，cache 一致性协议能确保其他线程一定能读到写入的值。

Alice 和 Bob 通过罐子发送信号的方式的问题在于只能将对方窗台上的罐子放倒，需要依赖对方将罐子重置，对方重置罐子有可能会有很大的延迟，所以会导致罐子用完。于是两人采用另外一种方法，将信号变量的所有修改权限都放在自己这边：采用对方能看到的旗帜，然后互相采用一种旗语来实现猫狗访问院子的同步（具体协议这里就略去了）。

这个故事引导我们思考一些问题：

- 互斥问题：猫狗不能同时出现在院子里；
- 避免死锁问题：如果两只宠物都想进入院子，那么一定要有一只最终能进去；
- 避免饿死问题：如果小猫或小狗想进入院子，那么这只小动物一定能得到机会进入院子；
- 等待问题：如果 Alice 或 Bob 有一人因故不管自己的小动物了，那么另一个人放小动物进入院子不应该受到影响。等待问题反映的是一种容错（*fault-tolerance*）问题。

Bob 和 Alice 的故事还可以引出另外两个多线程程序设计需要解决的问题：

- 生产者 -消费者（producer-consumer）问题：经过一番变故（结婚离婚）之后，Alice 和 Bob 变成这么一种关系，猫狗做好朋友了而且都归 Alice，Bob 要负责喂养猫狗，但是猫狗和 Bob 不能共存了，因此 Bob 要在院子里面放食物，然后 Alice 在 Bob 不在院子的情况下把两只宠物带出来；

- *读者 -写者（reader-writer）*问题：Alice 和 Bob 都太爱宠物了所以决定交流关于宠物的消息，于是 Bob 设立了一块公告板，Bob 在公告板上写内容，Alice 在自己家里读内容，这就是读者写者问题。读者写者问题要确保 Alice 能读到完整的句子。Alice 也不能整天坐在窗口傻等 Bob 写东西在公告板上。

2.2 Amdahl 定律

Amdahl 定律 [Amdahl, 1967]揭示了并行程序中串行部分比例对并行程序的加速比造成的影响。并行程序加速比定义为：

$$S = \frac{1}{1-p+\frac{p}{n}} \tag{1}$$

其中 p 表示并行部分的比例，n 表示可并行运行的计算单元数（例如处理器核心数）。

通过一些简单的计算可以告诉我们，串行部分对加速比的损伤是非常大的。例如，假设并行部分达到 90%，有 10 个处理器，那么只能得到 5 倍的加速比。这意味着我们必须努力缩小串行部分的比例。对于同步算法设计的意义就是：尽可能地缩小锁的粒度，甚至完全抛弃锁，采用无锁的算法。

3 Chapter 2: Mutual Exclusion

这一章介绍了共享内存同步中最常见的问题：互斥问题（mutual exclusion）。这一章首先介绍了一些形式化的描述方法。接下来介绍了一些经典的通过读写共享内存实现互斥的算法。尽管在实践中不可能再使用这些算法，因为现代的机器都有更高级的原子指令，但是了解这些算法的证明可以帮助理解同步领域中的算法正确性推导。

3.1 符号和概念

首先是时间的概念。在并发计算中，一般不关心绝对时间，而是关心相对时间，即事件发生的先后顺序。书中将线程建模为状态机（state machine），事件（event）导致状态机的状态发生变化。在这个模型中，事件是瞬时的（instantaneous），那么线程可以表示为 $a_0, a_1, ...$ 的序列。由于线程可以包含循环，所以事件 a_i 的第 j 次发生写为 a_i^j。如果事件 a 先于事件 b 发生，那么用 $a \to b$ 表示这个先序关系（precedence）。由于事件是瞬时发生的，所以先序关系 $\to$ 是所有事件的一个全序关系（total order）。

下面是区间（interval）的概念。如果两个事件 a_0 和 a_1 满足 $a_0 \to a_1$，那么 (a_0, a_1) 就称为一个区间。

两个区间：$I_A = (a_0, a_1)$，$I_B = (b_0, b_1)$，如果 $a_1 \to b_0$，那么区间 I_A 先于 I_B 发生，写为 $I_A \to I_B$。由于区间是一个时间段，所以两个区间可能有重合，有重合的区间就没有 $\to$ 关系了，所以先序关系 $\to$ 只能是所有区间集合的偏序关系（partial order）。

如果两个区间之间没有 $\to$ 关系，那么这两个区间就称为并发的（concurrent）。

以上介绍的这些形式化表示，基本上就是本书使用到的所有数学工具了。这种建模方法最早是由 Lamport 提出的 [Lamport, 1978]。引入符号的目的就是为了后面推导算法正确性的时候可以方便一些。可以说，这本书的 “理论” 部分实际上并没有复杂的理论，主要是逻辑推导，通过这些逻辑推导可以训练推导并发数据共享的思维方式。

如果数据结构会被多个线程修改，那么为了一个线程在修改数据结构的过程中不被其他线程打断，我们要将修改共享数据的代码（即危险的代码）放在临界区（critital section）中。线程进入临界区之前要尝试获得（acquire）一个访问临界区的锁（lock），即 `lock` 操作。如果当时已经有其

他线程获得了锁在访问临界区，那么这个线程就必须等待其他线程释放（release）锁，即 unlock 操作。这种机制确保了某一段代码一次只有一个线程可以访问，这种性质称为互斥（mutual exclusion）。

用上面的形式化语言描述互斥就是这样的：对于任意线程 A 和 B，以及任意整数 i 和 j，都有 $CS_A^k \to CS_B^j$ 或 $CS_B^j \to CS_A^k$ 的关系，也就是说任意一个线程的任意次执行临界区都不会和另一个线程任意次执行临界区重叠。

在使用锁的时候，要注意防止死锁（deadlock），也就是说，如果有一些线程尝试获得一个锁，那么必须要有某个线程能成功地获得锁，否则整个系统就停滞了。

此外，还要防止饿死线程的现象（starvation）发生，所谓饿死，就是说有的线程永远都得不到锁进入临界区，换句话说，即所有尝试获得某个锁的线程最终都能成功获得锁。

注意，如果不会饿死，那么必然不会死锁，但是不死锁不表示一定不会有线程饿死。

注意，这里讨论的死锁是两个层面上的，一个是锁算法本身的死锁，另一个是系统的死锁。如果一个锁算法本身是不会产生死锁，那么不表示系统不会死锁。因为系统是否死锁涉及到多个锁的访问顺序问题。例如假设锁 ℓ_0 和锁 ℓ_1 本身的实现都是不会产生死锁的，但是两个线程 A 和 B 按照这样的顺序访问这两个锁：A 获得了 ℓ_0，B 获得了 ℓ_1，接下来，A 试图获得 ℓ_1，B 试图获得 ℓ_0，这样两个线程就会死锁了，因为两个线程互相等待对方释放锁。

下面开始讨论锁的具体实现。首先讨论 3 个经典的适用于两个线程的锁算法，然后再扩展到适用于 n 个线程的锁算法。要注意这些算法都是通过自旋实现等待的，由于这些算法发明的年代硬件上还没有支持 RMW（read-modify-write）操作的原子指令实现，所以和我们现在实用的自旋锁实现方式不同，一是结构复杂，二是会浪费存储空间。在具有 RMW 原子指令的现代硬件上自旋锁的实现在第 7 章讨论。本章第 2.8 节（对应本笔记的 3.5 小节）对经典算法采用的空间下界进行了讨论。

3.2 适用于 2 个线程的锁

本书讨论了 3 个适用于 2 个线程的锁，这些锁都是 Peterson 提出的 [Peterson, 1981]。前两个算法很简单，但是会引起死锁，第三个算法结合了前两个算法，解决了死锁问题。

算法假定两个线程的 id 分别为 0 和 1。通过调用 ThreadID.get() 获得当前线程的 id，那么用 1 减掉当前线程的 id 就得到另一个线程的 id。

第一个算法 LockOne 的代码如下所示：

```
1  class LockOne implements Lock {
2      private boolean[] flag = new boolean[2];
3      // thread-local index, 0 or 1
4      public void lock() {
5          int i = ThreadID.get();
6          int j = 1 - i;
7          flag[i] = true;
8          while (flag[j]) {} // wait
9      }
10     public void unlock() {
11         int i = ThreadID.get();
12         flag[i] = false;
13     }
14 }
```

LockOne 锁算法的实现很简单：线程首先表示自己想进入临界区，然后等待对方退出临界区。下面我们来证明这个锁是互斥的。为了方便证明，

首先定义下面两种事件的表示方法：

- $\text{write}_A(x = v)$：表示线程 A 将值 v 赋给变量 x 的事件。我们可以假定这种读写操作一定是原子的。
- $\text{read}_A(x == v)$：表示线程 A 从变量 x 读到了值 v 的事件。

定义好了符号之后下面是证明（原书中的证明过程有点突兀，但是我这里这个证明过程一定非常好懂☺）：

证明. 证明的过程采用反证法。我们先从代码中可以看到以下事件序列：

$$\text{write}_A(\texttt{flag}[A] = true) \to \text{read}_A(\texttt{flag}[B] == false) \to CS_A \tag{2}$$

$$\text{write}_B(\texttt{flag}[B] = true) \to \text{read}_B(\texttt{flag}[A] == false) \to CS_B \tag{3}$$

式 2 表示，如果线程 A 进入了临界区，那么肯定会有这么一个事件序列：首先 A 向 $\texttt{flag}[A]$ 写入了 $true$，然后进入第 8 行的循环等待，由于进入了临界区，所以必然读到了 $\texttt{flag}[B] == false$，要不然就退不出循环。式 3 表示的是线程 B 进入临界区时的事件序列，也是用类似的方式解释。

然后进行反证假设：假设线程 A 和线程 B 同时进入了临界区，也就是说式 2 中的 CS_A 和式 3 中的 CS_B 中有同一个时间点重合了，再往前推，必定有一个时刻 $\texttt{flag}[A]$ 和 $\texttt{flag}[B]$ 同时为 $false$。从式 3 可以看出，线程 B 向 $\texttt{flag}[B]$ 写入了 $true$ 之后，$\texttt{flag}[B]$ 的值就再也没有变过，所以线程 A 如果要读到 $\texttt{flag}[B]$ 为 $false$ 的话，那么这个事件必然在写入 $\texttt{flag}[B]$ 的事件之前，因此可以得到下面的事件序列：

$$\text{read}_A(\texttt{flag}[B] == false) \to \text{write}_B(\texttt{flag}[B] = true) \tag{4}$$

由于事件的 $\to$ 关系是全序关系，所以是具有传递性的，因此上面式 2、3 和 4 可以连起来，得到：

$$\begin{aligned}\text{write}_A(\texttt{flag}[A] = true) \to \text{read}_A(\texttt{flag}[B] == false) \to \\ \text{write}_B(\texttt{flag}[B] = true) \to \text{read}_B(\texttt{flag}[A] == false)\end{aligned} \tag{5}$$

从这个式子可以看出，线程 A 写了 $\texttt{flag}[A] = true$ 之后，中间没有对 $\texttt{flag}[A]$ 的修改事件，而后面线程 B 又读到了 $\texttt{flag}[A] = false$，所以产生了矛盾，证明这个锁算法是可以满足互斥要求的。 □

上面就是对于 LockOne 算法的证明。这个证明过程虽然看上去比较繁琐，但是证明的思路很重要，本书大部分算法的正确性证明都采用了这种思路。不过证明的过程不需要太严谨和形式化。

LockOne 算法的问题是会死锁。假设两个线程同时执行了第 7 行，那么 $\texttt{flag}[A]$ 和 $\texttt{flag}[B]$ 同时为 $true$，两个线程都会在第 8 行的循环永远等待下去。

下面是第二个适合 2 个线程的锁 LockTwo：

```
1 class LockTwo implements Lock {
2     private int victim;
3     public void lock() {
4         int i = ThreadID.get();
5         victim = i;           // let the other go first
6         while (victim == i) {} // wait
7     }
8     public void unlock() {}
9 }
```

这个算法和 LockOne 类似，只不过前一个算法采用的是激进的风格，先试图抢占资源，而这个算法则是保守的风格，先把资源让给对方。证明的方法也是采用反证法：

证明. 从代码可以看出：

$$\text{write}_A(\texttt{victim} = A) \to \text{read}_A(\texttt{victim} == B) \to CS_A \tag{6}$$

$$\text{write}_B(\texttt{victim} = B) \to \text{read}_B(\texttt{victim} == A) \to CS_B \tag{7}$$

假设同时进入了临界区，那么说明 victim 同时具有 A 和 B 的值，这是不可能的。 □

LockTwo 算法的问题在于，如果只有一个线程在跑，那么这个线程自己就死锁了，必须要另一个线程并发运行的时候才能获得锁。LockOne 和 LockTwo 两个锁都是有问题的，一个并发执行有问题，一个自己执行有问题，两者互补，因此就有了 Peterson 锁。

Peterson 锁的代码如下所示：

```
1  class Peterson implements Lock {
2      // thread-local index, 0 or 1
3      private boolean[] flag = new boolean[2];
4      private int victim;
5      public void lock() {
6          int i = ThreadID.get();
7          int j = 1 - i;
8          flag[i] = true;   // I'm interested
9          victim = i;       // you go first
10         while (flag[j] && victim == i) {}; // wait
11     }
12     public void unlock() {
13         int i = ThreadID.get();
14         flag[i] = false;  // I'm not interested
15     }
16 }
```

可以看出，线程在试图获得锁的时候，首先声明自己想要获得锁，然后主动将锁让给对方，接下来就是等待两个条件至少有一个发生变化。第 10 行的循环等待检查两个条件，当只有一个线程运行的时候，可以确保有一个条件为 $false$，所以只有自己运行的时候不会死锁等待。两个线程并发访问的时候，一定能保证有某一个线程的 $\texttt{victim} == i$ 的条件为 $false$，因此 Peterson 锁本身也一定不会死锁。下面依然通过反证法来证明 Peterson 锁满足互斥条件：

证明. 反证法，假设两个线程都进入了临界区。从代码中可以看出，两个线程进入临界区之前执行了以下事件序列：

$$\begin{aligned}\text{write}_A(\texttt{flag}[A] = true) \to \text{write}_A(\texttt{victim} = A) \to \\ \text{read}_A(\texttt{flag}[B]) \to \text{read}_A(\texttt{victim})\end{aligned} \tag{8}$$

$$\begin{aligned}\text{write}_B(\texttt{flag}[B] = true) \to \text{write}_B(\texttt{victim} = B) \to \\ \text{read}_B(\texttt{flag}[A]) \to \text{read}_B(\texttt{victim})\end{aligned} \tag{9}$$

在上面的式子中，我们看到像 $\text{read}_A(\texttt{flag}[B])$ 这样的事件，这表示线程读了某个变量值，但是还不确定读到的是什么值。接下来，不失一般性，假设线程 A 最后写入 victim 变量，因此有下面的序列：

$$\text{write}_B(\texttt{victim} = B) \to \text{write}_A(\texttt{victim} = A) \tag{10}$$

由于有上面这个顺序，所以式 8 中的 $\text{read}_A(\texttt{victim})$ 事件可以填充为 $\text{read}_A(\texttt{victim} == A)$。根据代码第 10 行中的两个条件，由于要退出这个循环，所以只能是 $\texttt{flag}[B] == false$ 了。因此有下式：

$$\text{write}_A(\texttt{victim} = A) \to \text{read}_A(\texttt{flag}[B] == false) \tag{11}$$

结合上面所有的式子，根据 $\to$ 的传递性，可以得到下面的式子：

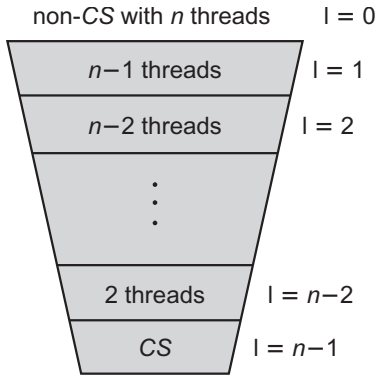
$$\begin{aligned}\text{write}_B(\texttt{flag}[B] = true) \to \text{write}_B(\texttt{victim} = B) \to \\ \text{write}_A(\texttt{victim} = A) \to \text{read}_A(\texttt{flag}[B] == false)\end{aligned} \tag{12}$$

这个式子的第一项和最后一项就发生了矛盾，读出的值和写入的值不一样，所以假设不成立，因此 Peterson 锁是可以满足互斥条件的。 □

Peterson 提出的 Peterson 锁是最简单优雅的 2 线程锁算法了。下面要讨论的是 2 线程的 Peterson 算法扩展到支持 n 个线程的实现。

3.3 适用于 n 个线程的锁

Filter 锁（过滤器锁）是对 2 线程的 Peterson 锁的简单泛化，思想很简单：Peterson 锁每次能从两个并发线程中选出一个线程进入临界区。那么过滤器锁顾名思义，对 n 个线程进行 $n-1$ 次过滤，每一轮（即）有一个线程被过滤出来等待，进行了 $n-1$ 轮之后，可以确保只有一个线程进入临界区，从而实现互斥。过滤过程是通过层次（level）实现的，就像下面这个图中所示的一样，每一个层次会有一个被踢出来的牺牲者。



下面是 Filter 锁的代码。注意代码的第 17 行的循环等待中，为了代码的简洁，while 的条件用了逻辑条件 $\exists k != me$，表示存在除了“我”之外存在其他线程满足后面的条件，实际通过代码实现的时候也是用一个循环遍历所有其他线程的 id。

```
1  class Filter implements Lock {
2      int[] level;
3      int[] victim;
4      public Filter(int n) {
5          level = new int[n];
6          victim = new int[n]; // use 1..n-1
7          for (int i = 0; i < n; i++) {
8              level[i] = 0;
9          }
10     }
11     public void lock() {
12         int me = ThreadID.get();
13         for (int i = 1; i < n; i++) { // attempt level i
14             level[me] = i;
15             victim[i] = me;
16             // spin while conflicts exist
17             while ((∃k != me) (level[k] >= i && victim[i] == me)) {};
18         }
19     }
20     public void unlock() {
21         int me = ThreadID.get();
22         level[me] = 0;
23     }
24 }
```

代码中使用 level[] 数组记录每一个线程所在的层次（第 14 行），用 victim[] 记录每一个层次阻塞的线程（第 15 行）。从第 17 行代码可以看出，每一层必然只有一个线程可以满足 $\texttt{victim}[i] == me$ 的条件，其他在这一层的线程都是不可能满足这个条件的，所以都会退出 while 循环进入下一层。

换个角度说，如果我不是当前层的受害者，那么我就能进入下一层，或者即使我是受害者，但是其他线程都还没有进入到我这一层或我后面的层次，那么我也能进入下一层，也就是说 while 中的第一个条件不满足。从

这个描述可以看出，过滤器锁本身是不会死锁的。

Filter 锁的互斥性证明不好直接证明，因此首先证明一个引理，即对于 $0 \leqslant j \leqslant n-1$，在第 j 层最多能进入 $n-j$ 个线程。有了这个引理之后就可以直接推出在第 $n-1$ 层最多能进入 $n-(n-1)=1$ 个线程，即满足 n 个线程的互斥。下面简单描述一下用数学归纳法证明这个引理的框架：

证明. 基础情形：$j=0$ 的时候，n 个线程还没有开始竞争，结论很明显成立。

归纳假设：假设在 $j-1$ 层，最多有 $n-j+1$ 个线程。采用反证法证明这个归纳假设。假设在第 j 层就有 $n-j+1$ 个线程。具体证明步骤和之前的证明差不多，也是从代码的顺序出发，然后找到矛盾。 □

此外，通过数学归纳法还可以证明 Filter 锁是不会饿死的。

3.4 公平性和面包店算法

不饿死的属性只能保证每一个请求锁的线程最终都能进入临界区，但是不保证这些线程进入临界区的顺序，也就是说无法保证每个线程从请求获得锁到真正获得锁的时间是不确定的。有时候，我们会要求获得锁的顺序服从先到先服务（first-come-first-served）的顺序。

之前的锁算法中，我们可以看出每一个 lock 方法的基本结构都是先设置一些变量，然后进入一个循环等待。那么将之前的设置定义为入口区（doorway）D_A，将后面的等待定义为等待区（waiting）W_A。入口区的步骤是一定的，而等待区的步骤是无限的。先到先服务的公平性可以定义为：对于任意线程 A 和 B 以及任意整数 j 和 k，如果有 $D_A^j \to D_B^k$，那么有 $CS_A^j \to CS_B^k$。

Lamport 认真观察生活，发现面包店的叫号系统能很好地维护先到先服务的顺序。就像我们银行的排队叫号系统一样，假设只有一个柜台提供服务，所有到达的客户按照先到先领号的顺序领号，按照顺序叫号自然能保持先到先服务的公平性，而且只有一个柜员能保证到达客户的互斥性。Lamport 的 Bakery 锁（面包店算法）就是这种思想。

这个算法的思想听上去很简单，但是在没有 RMW 这类原子操作的前提下，仔细想还是不容易实现的，因为取号这个操作本身是一个多步骤的操作，而且叫号的操作需要扫描所有已有的号，也是一个多步骤的操作。下面看一下 Lamport 给出的算法，当然，我们不能指望这个算法有多么高效，所以说这些算法只有理论意义和历史意义，而在今天不具有实践意义。

Bakery 锁算法中，每一个线程有两个状态变量，flag 表示线程是否请求锁，另一个是 label，用来保存线程取到的号。Bakery 锁算法的代码如下：

```
1  class Bakery implements Lock {
2     boolean[] flag;
3     Label[] label;
4     public Bakery (int n) {
5        flag = new boolean[n];
6        label = new Label[n];
7        for (int i = 0; i < n; i++) {
8           flag[i] = false; label[i] = 0;
9        }
10    }
11    public void lock() {
12       int i = ThreadID.get();
13       flag[i] = true;
14       label[i] = max(label[0], ...,label[n-1]) + 1;
15       while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
16    }
```

```
17     public void unlock() {
18         flag[ThreadID.get()] = false;
19     }
20 }
```

先解释一下这段代码中第 15 行使用的 $<<$ 关系，这个关系的定义是：

$$(\texttt{label}[i], i) << (\texttt{label}[j], j)$$
$$\text{当且仅当} \tag{13}$$
$$\texttt{label}[i] < \texttt{label}[j] \quad \text{或者} \quad \texttt{label}[i] = \texttt{label}[j] \text{ 且 } i < j$$

我们来看 lock 方法。线程首先将自己标记为感兴趣，然后通过 max 操作获得当前最大标签，并加 1 设置为自己的标签。这个 max 操作显然不是原子操作，所以多个线程并发访问的时候可能会取到相等的号码，因此我们这里用 $<<$ 关系使得两个线程即使取到的号是一样的也能分出先后。在第 15 行的等待区中，线程不断地以任意顺序（顺序不重要）扫描其他所有的线程，如果发现只要存在线程正在尝试进入临界区并且比自己 $<<$，那么继续等待。很显然，扫描其他所有线程状态的操作也不是原子操作，但是这个操作不是原子操作也没关系，不会因为并发而影响正确性。因为如果要退出第 15 行的 while 循环，必须是扫描了一圈之后发现自己的标签不会小于所有请求锁的线程的标签，所以在扫描过程中，即使其他线程状态发生了变化，也只可能是标签值变得更大了，因为标签是单调递增的，所以不会破坏这个循环等待本身的语义。

有关这个算法不死锁性质、先到先服务的性质以及互斥的性质书上的证明很简单，这里就不证明了，至少从上面的 informal 描述可以看出 Bakery 锁算法就是按照这些性质要求设计的。

要注意书中几个证明的顺序：首先证明了不死锁，然后证明了先到先服务的公平性。只要同时满足这两个性质，那么这个算法必然是不会饿死的。

Bakery 锁算法中的 label 是单调递增的，而且在 unlock 方法中并没有重置 label，所以作为时间戳的 label 有溢出的可能。原书 2.7 节介绍了一些有界时间戳的数据结构和算法，这里暂且略去。

3.5 有关经典算法的空间下界

看了前面的 Bakery 算法之后，我们要想，这个算法既然这么简洁而且公平，为什么这个算法不实用呢？原因就在于时间效率和空间效率都低。如果需要服务于 n 个线程，那么空间复杂度就是 $O(n)$。而且在入口区需要扫描一遍所有线程的状态，在等待区每一次循环的时候也需要扫描一遍所有线程的状态。

那么在只提供共享读写内存的情况下，有没有更 “聪明” 算法呢？

答案是*没有*。形式化的定理是：任何互斥锁算法，在通过读写共享内存的条件下实现 n 个线程的不死锁（deadlock-free）互斥，必须至少使用 n 个不同的内存位置。本书 2.8 节就在证明为什么这种互斥锁的算法空间复杂度为 $O(n)$，这个下界是由 Burns 和 Lynch 给出的 [Burns and Lynch, 1993]，本书中的证明也来自这篇论文。

证明的过程并不重要，我们不需要把书上冗长的证明看得太明白。证明的基本思想是利用这样一个事实：对于不死锁的互斥，线程在试图进入临界区的时候必须能看到临界区的状态，也就是说要知道是不是有其他线程正在占有临界区。证明的过程分两种情况讨论。第一种情况是共享的内存位置只有单个写者的情况，也就是说一个位置只能被一个线程写入（Bakery 算法就是这种情况），那么很自然一个 n 个线程必须至少有 n 个内存位置才能让大家都看到临界区的状态；第二种情况是共享的内存位置允许多个写者写入的情况（例如 Peterson 算法中的 victim 变量），假设有一

个线程 A 正准备写入这个位置，这时线程 B 写入了这个位置，然后通过某种方式进入了临界区，B 还在临界区的时候，A 完成了写入，这时已经覆盖掉了 B 写入的状态，所以 A 也不知道临界区有没有别人在，也闯入了临界区，这样就不满足互斥了。证明的基本思想就是这样，但是实际的证明要复杂得多，有兴趣可以读 2.8 节或者参考原来的论文。这一种证明技术也被广泛应用于证明分布式计算中的各种下界。

在理解空间下界证明的时候，理解原书中的定义 2.8.1 特别重要，但是这个定义的原文比较绕口：

> **Definition 2.8.1** A `Lock` object state s is *inconsistent* in any global state where some thread is in the critical section, but the lock state is compatible with a global state in which no thread is in the critical section or is trying to enter the critical section.

那么这个定义实际上是定义了锁对象状态不一致（inconsistent）的意义：如果一个锁的状态处于某个线程正在临界区的全局状态，而同时这个锁状态和没有线程在临界区或准备进入临界区的全局状态兼容，那么这个锁对象的状态就是不一致的。也就是说，锁的状态必须要可靠，使得线程能通过锁的状态判定是否有线程正在临界区，如果判断不出来（即 “兼容” 了没有线程在临界区或准备进入临界区的全局状态），那么这个锁肯定不能实现无死锁的互斥。换句话说（也就是书中这个定义后面的引理 2.8.1 描述的），任何一个无死锁的锁算法都不可能进入不一致的状态。

尽管这个定理的证明并不重要，但是这个结论的意义却很重要：那就是光提供内存位置的原子读写操作是不够的，还需要更强大的原子操作。本书后面第 5 章和第 6 章就从理论上论证了各种原子操作的相对能力，通过这几章我们就可以知道为什么现代的硬件都提供了 CAS 或 LL/SC 之类的原子操作。

4 Chapter 3: Concurrent Objects

这一章介绍的是并发数据结构（由于书中使用的是 Java 语言作为演示语言，所以书中都叫做 “对象”，本笔记中在不会引起误会的情况下会混用对象和数据结构）的两个重要行为属性：

- 正确性（correctness）：正确性就是表示算法执行要正确。在单线程环境中，对正确性的要求就希望程序执行的结果符合程序中各条语句的顺序产生的结果，即程序最终结果要和程序的语句顺序保持一致（即程序顺序的一致性）。而并发环境中，由于多个线程是交错执行的，就好像讨论问题的维度发生了变化，所以对并发对象（具体说是针对并发对象操作的并发算法）的正确性定义就变得复杂了。多线程程序的正确性就不能像单线程程序那样简单地定义为程序顺序的一致性，而是可以定义为多种不同的一致性要求，学术界定义的一些并发一致性从弱到强的顺序分别为静态一致性（quiescent consistency）、顺序一致性（sequential consistency）和可线性化性（linearizability）等。一致性越弱则并发性能越好。应用程序采用哪种一致性取决于对正确性的要求。
- 进展性（progress）：也称为 liveness（活性），分为两类：阻塞的（blocking）和非阻塞的（non-blocking）。由于并发数据结构要由多个线程并发访问，所以线程在访问的时候很可能需要等待（比如说在一个循环中不断地判断某些状态）。非阻塞条件中最强的是无等待（wait-free）算法，这种算法能保证所有对并发对象进行操作的线程能在有限的步骤内完成操作。无锁（lock-free）条件是稍弱一点的条件，如果并发对象的某个操作是无锁的操作，那么可以保证有某些线程能在有限的步骤内完成操作。非阻塞条件中最弱的是无干扰（obstruction-free），表

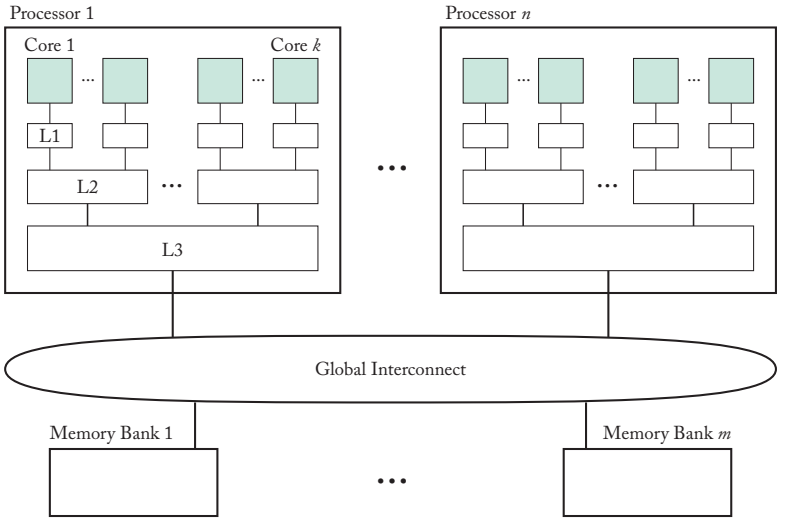
示如果其他线程都不动，那么在运行的那个线程没有其他线程的干扰，一定能在有限的步骤内完成操作。由于无等待的算法复杂度较高，所以满足无锁条件和无干扰条件的算法是在现在的高并发系统中使用最多的。本书中很多并发对象的操作给出的也都是无锁实现或无阻碍实现。

本笔记对第 3 章的总结并不严格遵守第 3 章的内容，试图以一种更简洁易懂且实用的方式介绍并发对象的正确性定义。为了内容的完整性，这份笔记在这一节还复习一些和多核硬件相关的知识，部分内容来自于 [Scott, 2013]。

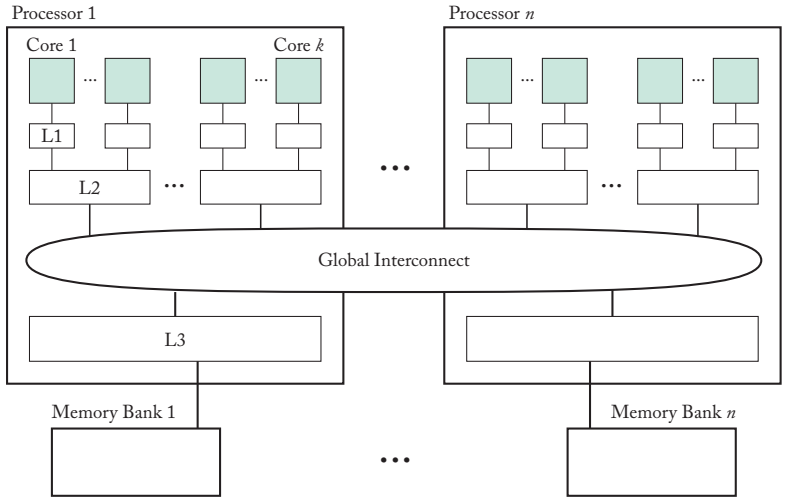
下面首先复习一下多核硬件相关的知识，主要是多核硬件内存访问的一致性问题，然后从单线程程序的顺序一致性出发，讨论多线程并发程序的正确性问题，最后讨论并发算法的活性。

4.1 多核硬件基础知识

下面两个图取自 [Scott, 2013]，这两个图从内存访问的角度展示了典型的多核平台架构。图中展示了处理器（对应 socket）、处理器核心（core）、各级高速缓存（cache）以及内存之间的连接关系。第一个图展示的是均匀内存访问（uniform memory access，UMA）的结构，在 UMA 系统中，所有处理器核心到内存的距离都是一样的，因为处理器连接在全局互联网络中，核心对内存的访问被互联网络分发到各个内存控制器。



第二个图展示的是非均匀内存访问（nonuniform memory access，NUMA）的结构。在 NUMA 系统中，内存控制器直接连接到处理器，处理器 i 上的核心如果要访问处理器 j 连接的内存，那么需要通过全局互联网络将内存访问的请求发送给处理器 j，处理器 j 再将内存访问的结果发送回处理器 i，所以处理器核心访问不同内存的距离是不一样的。从图中可以看出，NUMA 架构的 L2 缓存是通过互联网络直接连起来的。有关 NUMA 架构的基础知识，以及 Linux 操作系统对 NUMA 架构的处理和优化，可以参考近期 ACM Queue 杂志的一篇文章 [Lameter, 2013]。



在应用程序开发中，由于多核硬件架构的特殊性，需要特别关注 *cache 相关的问题*和*内存一致性相关的问题*，下面对这两方面的问题进行阐述。

cache 是处理器核心和内存之间的高速缓存，可能有好几层，距离处理器核心越近的 cache 速度越快，容量越小。处理器核心要访问数据的时候首先要在离自己最近的 cache 中查找要访问的数据是否存在，如果不存在，则向下一层 cache 查找，如果 cache 中都没有数据（称为没有命中），则需要发起一次内存访问请求。如果命中了，则直接从 cache 中取数据到寄存器进行操作。由于不同层次之间 cache 速度的差异是数量级上的差异，因此每次 cache 缺失都会造成延迟，处理器核心需要停下来等待数据到位。虽然处理器设计中通过复杂的流水线技术和乱序执行技术（out-of-order，后面我们会讨论乱序执行技术带来的内存一致性的问题）实现了指令集的并行（instruction level parallel，ILP），能够在一定程度上缓解访问内存带来的延迟，但是随着处理器内部集成的核心越来越多，单个核心的设计也越来越简单，所以在众核处理器中这种功耗高的复杂流水线技术以及乱序执行技术可能会被简化甚至取消（例如 60 核的 Intel Xeon Phi 协处理器），因此在编写应用中关键部分的时候应该特别注意 cache 的命中率，一些常见的 profile 工具也会给出各级 cache 利用率的数据。

cache 中的数据是以 cache 线（line）来组织的，一条 cache 线一般是 32 到 512 字节，现在貌似 64 字节的 cache 线比较常见，cache 线是 cache 操作的基本单位。cache 线和内存中的地址有映射关系。如果需要加载某个内存地址的数据到 cache，不论请求的数据大小是多少，都会把整条 cache 线大小的数据加载到 cache 中。在编程的时候，有关 cache 需要注意的问题可以参见这篇博文 “Gallery of Processor Cache Effects”[4]，作者用具体示例的方式演示了 7 个需要注意的问题，这是一篇非常好的科普文，读了这篇文章就可以了解 cache 对于软件开发的意义。

我们再看一下在多核环境下的 cache。从前面的图中可以看出，每一个处理器核心都有一个私有的 L1 cache，然后再往下走 L2 cache 一般是几个核心共享的，如果还有再深层次的 cache，则由更多的核心共享，到最底下物理内存由所有处理器核心共享。可以看出，在靠近处理器核心层次的 cache 是分区的，甚至被核心独享的。那么这就引出 cache 一致性（cache coherence）的问题了。假设核心 1 访问某个内存位置，那么在核心 1 的 L1 中就会有一份拷贝（注意是整条 cache 线的拷贝），如果核心 2 也访问了，那么在核心 2 的 L1 中也会有一份拷贝。如果两个核心都是读这个位置，那就相安无事。如果核心 2 修改了这个内存位置，那么必须通过某种方式通知核心 1 的 L1，这样 L1 下次再读这份数据的时候就不会读到旧数据。在多核处理器中，保持多份 cache 拷贝一致性的机制称为 cache 一致性协议。cache 一致性协议的具体设计和实现已经超出了这份笔记的范围，强烈建议参考文献 [Sorin et al., 2011]。在 Intel 的处理器手册中也可以找到 Intel 处理器使用的 cache 一致性协议。

[4]原文地址：http://igoro.com/archive/gallery-of-processor-cache-effects/，酷壳上的中文翻译：http://coolshell.cn/articles/10249.html

由于 cache 一致性协议，所以在多核多线程程序设计中要特别注意 “Gallery of Processor Cache Effects” 文中提到的*伪共享*（false sharing）问题，由于 cache 是按照 cache 线组织数据的，在同一条 cache 线内的一个字节被修改了都会使得整条 cache 线失效，因此频繁读写的数据最好放在独立的 cache 线中，表现在数据结构中就是通过 padding 实现 cache 对齐（alignment）。如果在多线程的应用程序中要使用每一个线程都有自己一份独立拷贝的数据结构，那么这个数据结构也应该 cache 对齐。

下面讨论内存一致性（memory consistency）的问题。

所谓的内存一致性就是说程序在执行的过程中，内存状态变化的顺序和程序语句的顺序是完全一致的。然而在现代硬件中，为了优化写入内存的性能，处理器的写请求一般不会立即到达内存，而是会被放在一个叫做*写缓冲区*（write buffer）的硬件队列中，然后在恰当的时机批量送入内存。很明显这种批次（batch）操作可以减少处理器到内存的流量，提升整体性能。这种优化带来的后果就是内存的实际写入顺序可能会和程序顺序不一致，而且由于写操作被打包了，所以穿插在写操作中的读操作会被重排到打包的写操作之外。除了硬件对读写操作的重排之外，编译器在优化的过程中有可能也会对读写操作进行重排。换句话说，硬件为了优化内存访问的性能（减少或隐藏延迟、减少总线或片上网络流量等），现代硬件会采用较为松弛的（relaxed）内存一致性。

在单核单线程的系统中，硬件可以保证程序执行的结果符合程序顺序的预期，例如首先设置了一个变量，然后后面一条语句再根据这个变量的值做一些操作，完全不用担心这个变量的值还没有设置上，也就是说某一条语句看到的变量的值一定是这条语句之前最后一条写入这个变量的语句写入的值。

然而在多核系统中，特别是通过共享内存进行线程间通信的多线程程序中，这种松弛的内存会产生严重的问题，那就是尽管在线程本身看来执行的顺序好像和程序语句指定的顺序一样，但是在其他线程看来，这个顺序就无法保证了。比如说下面的例子：

```
1 // initially x = f = 0
2
3 // thread1:
4 x = foo();
5 f = 1;
6
7 // thread2:
8 while (f == 0) {} // spin wait
9 y = 1/x;
```

这段代码展示了两个线程，变量 x 和标志 f 的初始值都为 0，线程 1 先设置 x 的值，然后设置标志 f，意图让线程 2 退出自旋等待，然后用 x 作为分母计算 y 的值。这段代码看上去没有问题，但是在实际中线程 2 可能会出现 “除 0” 的错误，因为线程 1 的写入 x 和写入 f 的顺序可能会重排，导致线程 2“看到” 标志 f 变为 1 的时候 x 还仍然为 0。而对于线程 1 来说，由于两条语句并没有依赖关系，所以顺序重排不会有正确性的问题，还能提升性能。“Memory Reordering Caught in the Act” 这篇博文[5]“活捉”了一个重排在现实中出现的现象。Lamport 定义了*顺序一致性*（sequential consistency）[Lamport, 1979]来规定线程在其他线程看上去的一致性，现代的硬件一般都提供了显式的屏障指令允许程序员在像上面的例子的情况下强制采用顺序一致性。通常在硬件手册上可以找到重排的具体规则和强制顺序一致性的屏障（barrier 或 fence）指令，但是为了编写可靠且跨平台的代码，*强烈建议不要*依赖于具体平台提供的具体的重排规则，这里推荐本人的一篇博文 “Erlang 运行时提供的原子操作 API”[6]，这篇文章总结了

[5] http://preshing.com/20120515/memory-reordering-caught-in-the-act/

[6] http://www.cnblogs.com/zhengsyao/archive/2012/11/02/Erlang_atomics_op_and_memory_barrier.html

Intel 平台上的重排规则和屏障指令，还介绍了 Erlang 运行时中对于各种硬件差异性的跨平台处理。要注意，屏障指令由于强制取消了硬件的重排优化，所以会影响性能，所以在编写多线程程序的时候除非真地必要，否则不要使用屏障指令。

一致性（coherence）和一致性（consistency），这里我们看到两个概念的中文都叫一致性（后面第 6 章还有一个一致性：consensus，到时再表）。那么这两个 “一致性” 有什么区别呢？我想，既然这两个词都是从英文文献中而来，那么可以尝试探寻一下这两个词的本源，也许从中可以看出区别来。根据 “New Oxford American Dictionary 3rd edition” 中提供的拉丁语词源，形容词性的 coherency 源自 cohærent，表示 “sticking together”，也就是 “绑定在一起”，很形象地表示了一个对象的值在同一存储层次中的多个实体中（例如多个 L1 cache）的副本绑定在一起的场景，一个副本发生了变化，其他副本也要通过一致性协议同时发生变化。而形容词性的 consistent 源自 consistent，表示 “standing firm or still, existing”，表示的是一种稳定性，也就是逻辑上的合理性，比如内存一致性中实际结果和程序员预期的一致性。

这只是我的理解，open for debate :)

4.2 并发算法的正确性条件

并发的正确性条件就是判定并发程序是否正确需要满足的条件。串行程序的正确性条件很好定义，对于串行对象的方法，只需要针对每一个方法，定义好进入方法前的条件，以及方法执行后期待得到的结果条件即可。每一个方法在对对象进行操作的时候都不会被打断，我们可以完全不用理会方法执行过程中间对象的状态，在方法调用之间（前一个方法返回之后到后一个方法调用之前的时间段内）对象的状态是固定不变而且是可推导的。

然而对于被多个线程共享的并发数据结构来说，事情就没这么简单了。在这种情况下，并发对象可能会被多个线程运行的同一个方法或不同方法并发操作，这些方法在时间维度上互相交叠（也就是之前定义的偏序关系区间），并发对象甚至没有 “在两个方法调用之间” 的状态。

为了探讨如何定义并发程序的正确性，书中首先举了一个极端的例子，这个例子实现了一个基于数组的并发队列。为什么说这个例子很极端呢？是因为这个例子在实现 `enq` 和 `deq` 操作的时候，使用的算法就是普通队列使用的算法，但是为了支持并发，在这两个方法的开头和结尾都分别添加了加锁和解锁操作。很显然，我们可以认为这个算法是正确的，因为一系列的 `enq` 和 `deq` 操作都被一个锁给串行化了，每个方法调用被强行分隔开。这种并发固然正确，但是实际上等价于串行之行，没有实现真正的并行化，因此通过这种粗粒度的锁实现并发的意义不大。

具有实际意义的高并发算法必须使用细粒度的锁甚至不使用锁，这样才能提升并行度。因此下面就探讨各种强度不同的一致性（consistency）条件，和之前讨论的内存一致性一样，越强的一致性条件对并发度的影响越大，因此性能损失也越大。并发应用程序在设计的时候可以根据自身的需求选择适当的一致性条件。

4.2.1 静态一致性条件

前面说并发对象可能总在被一些方法操作。那么假设有这么一种状态，那就是所有方法的操作都完成了，没有任何方法在操作一个并发对象的状态称为静态（quiescence 状态，表示 “安静” 了，注意不是 static，static 表示的是静止不变的意思）。如果一些方法调用被安静状态分隔开，但是能够区分出安静前后方法调用的结果，那么这种条件称为静态一致性条件。

比如说线程 A 和 B 并发 enq 了一个 FIFO 队列 x 和 y，那么 x 和 y 在队列中的顺序我们可能没法区分。然后队列进入静态状态。接下来，线程 C 将 z enq 了这个队列，我们能区分出 z 一定在 x 和 y 后面，这就是静态一致性。满足静态一致性的时候，当对象进入静态状态时，这个状态等同于之前已经执行的方法以某个顺序执行的结果。

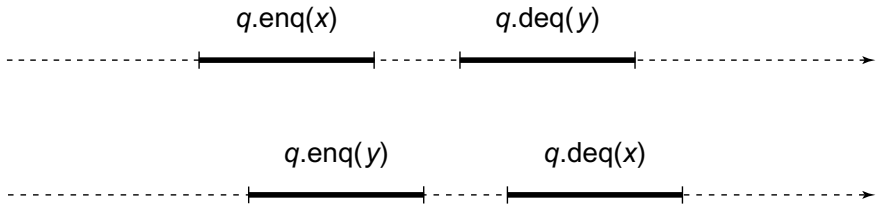
共享计数器也是一种满足静态一致性的并发对象。

由于静态一致性对顺序的要求最弱，所以可以说静态一致性对并发性没有任何影响。

静态一致性是可组合的（compositional）。可组合的意思是说，如果系统中的每一个对象都满足某个正确性条件，那么整个系统也满足同样的正确性条件。可组合性也是考察并发对象的一个重要因素，因为大型系统的构建必然会采用模块化的方法。

4.2.2 顺序一致性

有了之前对内存一致性的讨论，这里就好理解并发对象顺序一致性的定义了。类似内存一致性，如果针对某个对象的操作从任何一个线程的角度看来，都和这个线程的程序顺序一致，那么就说明这个执行过程是具有顺序一致性（sequential consistency）的。比如说下面这个图展示了线程 A 和线程 B 的时间线，上面是线程 A，下面是线程 B：

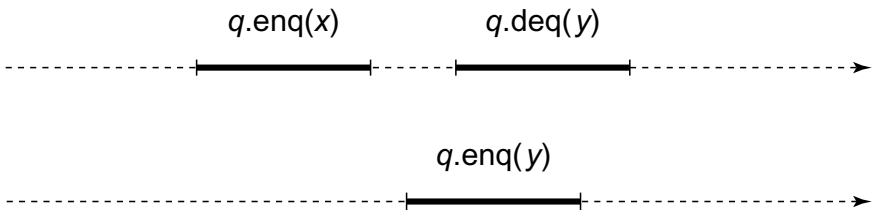


对于这样的执行历史，有两种可能的执行顺序都满足顺序一致性：

1. A enq(x) $\rightarrow$ B enq(y) $\rightarrow$ B deq(x) $\rightarrow$ A deq(y)
2. B enq(y) $\rightarrow$ A enq(x) $\rightarrow$ A deq(y) $\rightarrow$ B deq(x)

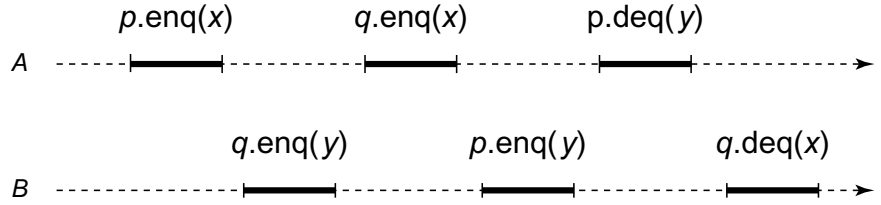
单独看 A 和 B，这两种执行结果和两个线程的程序顺序都是一致的，因此这两种执行都是符合顺序一致性的。

再看一个符合顺序一致性的例子：还是两个线程：线程 A 和线程 B 的时间线，上面是线程 A，下面是线程 B：



这个例子初看上去比较匪夷所思，明明 A 先 enq(x)，B 再 enq(y)，怎么 A 就 deq(y) 了？看上去不符合 FIFO 的语义。但这个例子确实是顺序一致性，因为 A 的两个操作在程序顺序中没有相关性，所以顺序一致性可以自由地重排这些操作。

顺序一致性的缺点在于：不具有可组合性。还是用一个例子来说明，如下图所示：



这个例子包含了两个并发 FIFO 队列 p 和 q，q 和前面那个例子中的 p 是一样的，p 则是对称的。很明显，p 和 q 都满足顺序一致性，但是这两个对象组合在一起，就不可能重组出一个和程序顺序一致的执行顺序了。像前

一个例子一样分别看 p 和 q，根据 deq 返回的内容，p 和 q 的 enq 顺序一定是：

1. B:p.enq(y) → A:p.enq(x)
2. A:q.enq(x) → B:q.enq(y)

而从程序顺序，我们可以得到：

1. A:p.enq(x) → A:q.enq(x)
2. B:q.enq(y) → B:p.enq(y)

这两条顺序和前面两条顺序整合在一起会形成环，所以产生了矛盾，因此 p 和 q 两个满足顺序一致性的对象结合在一起就怎么也无法满足了。

下面再给出 [Scott, 2013]中的一个例子，这个例子使用的是一个编造出来的并发对象：

```
1 // initially L is free and A[i] = 0 ∀i ∈ 𝒯
2 void put(int v):
3     L.acquire()
4     for i ∈ 𝒯
5         A[i] = v
6     L.release()
7
8 int get():
9     return A[self]
```

这其实是段伪代码。代码中的 $\mathcal{T}$ 表示所有线程 id 的集合。每一个线程都有一个私有的值，放在数组 A 中。put 方法在锁的保护下，逐个设置所有线程的私有值，由于上锁了，所以这个过程不会被中断。get 方法返回当前线程的这个私有值。可以看出，如果 get 和 put 并发执行，get 不一定能返回 put 设置的值，因为 put 有可能还没有轮到自己。

下面来看两个这样的并发对象组合的情况，看看这个两个对象组合了是否还满足顺序一致性。系统中有 4 个线程：

1. $T1$ 调用 X.put(1)
2. $T2$ 调用 Y.put(1)
3. $T3$ 调用 X.get() 和 Y.get()
4. $T4$ 调用 X.get() 和 Y.get()

这 4 个线程的一个执行顺序如下表所示：

	共享对象的值							
	$T1$		$T2$		$T3$		$T4$	
	$X[1]$	$Y[1]$	$X[2]$	$Y[2]$	$X[3]$	$Y[3]$	$X[4]$	$Y[4]$
初始	0	0	0	0	0	0	0	0
$T1$ 开始 X.put(1)	1	0	1	0	0	0	0	0
$T2$ 开始 Y.put(1)	1	1	1	1	0	1	0	0
$T3$: X.get() 返回 0	1	1	1	1	0	1	0	0
$T3$: Y.get() 返回 1	1	1	1	1	0	1	0	0
$T1$ 完成 X.put(1)	1	1	1	1	1	1	1	0
$T4$: X.get() 返回 1	1	1	1	1	1	1	1	0
$T4$: Y.get() 返回 0	1	1	1	1	1	1	1	0
$T2$ 完成 Y.put(1)	1	1	1	1	1	1	1	1

从 $T3$ 的角度看，Y 先变为 0，而从 $T4$ 的角度看，X 先变为 0，因此破坏了一致性。为了解决这个一致性问题，还需要有一种机制能让其他线程看到更新线程完成更新的那个时间点。

4.2.3 可线性化一致性

为了解决顺序一致性的不可组合的问题，Herlihy 和 Wing 提出了一种更强的一致性条件：可线性化性（linearizability）[Herlihy and Wing, 1990]。

从前一小节对顺序一致性的描述可以看出，我们对方法调用生效的时间点并不确定。可线性化性将限制增强为：每一个方法的调用必须在方法调用开始和返回之间的某一个时间点瞬间生效。自可线性化性提出的 20 多年以来，一直都是并发对象的标准一致性标准。在本书后面介绍的各种无锁算法的分析中，我们可以看到分析中几乎都要找到算法的线性化点（linearization point）。

线性化点就是方法生效的那个时间点。对于基于锁的实现来说，临界区可以当做是线性化点。对于无锁算法来说，线性化点的确定则较为复杂，通常是某一个让别的方法调用可以 “看到” 当前方法起作用的点，而且在不同条件下同一个方法的线性化点可能不同。

4.3 并发算法的活性（liveness）条件

讨论完了以上和一致性相关的正确性标准之后，我们再从时间的维度讨论并发算法，我们希望并发算法能不断产生进展（make progress）。并发算法的活性条件（也称为进展条件）分为两类：一类是阻塞（blocking）算法，另一类是非阻塞（nonblocking）算法。

- 阻塞算法：使用这种算法的时候，如果某一个线程发生意外的延迟，那么其他所有线程都无法产生进展。在多核处理器上发生意外延迟是非常常见的，意外延迟包括 cache 缺失、页错误、操作系统抢占等。基于锁的并发算法本质上就是阻塞的，如果持有锁的线程发生了意外的延迟，那么其他所有线程都得等着。基于锁的并发算法的临界区如果不小心出现了无限循环，那么其他等待锁的线程也进入无限等待。
- 非阻塞算法：这种算法保证方法的调用一定能够结束。非阻塞算法包括了几种不同强度的条件：
  - 无等待（wait free，最强）：保证每一个方法的调用都可以在有限的步骤内完成操作返回；
  - 无锁（lock free）：保证某一个线程调用方法的时候可以在有限的步骤内完成操作返回；
  - 无干扰（obstruction free，最弱）：保证某一个线程在调用方法的时候，可以在所有其他线程都不执行（干扰）的情况下在有限的步骤内完成操作返回。

设计并发算法的时候要在算法效率和进展条件之间做平衡。强度越强的非阻塞算法固然能最大限度提升并行度，因为所有线程都能产生进度，但是算法本身的复杂度也更高，在第 6 章会讨论 Herlihy 最早提出的通用 wait-free 算法 [Herlihy, 1991]，使用这种通用的算法能将任何数据结构修改为 wait-free 的并发数据结构，但是算法效率极低，因为虽然说可以确保在有限的步骤内完成，但是这个常量可能非常大。Kogan 等人近几年的工作提出了快速的 wait-free 算法构造方法 [Kogan and Petrank, 2012]，能够极大地减少时间开销，但是空间复杂度依然不低。因此在现实中使用最多的还是无锁算法和无干扰算法，在本书后面介绍的各种实用算法中我们就能看出这一点。

这一章有关并发算法正确性的内容比较理论化，就是抽象。因此学习这一章的最大感觉就是，很多概念都有似曾相识的感觉，不自觉地想往自己已有的知识上套。本书后面在讨论各种并发算法的时候，都会分析算法满足的具体正确性条件，因此训练这种思维能够帮助我们编写更健壮的并发程序。

5 Chapter 4: Foundations of Shared Memory

To be done...

6 Chapter 5: The Relative Power of Primitive Synchronization Operations
Chapter 6: Universality of Consensus

To be done...

7 Chapter 7: Spin Locks and Contention

To be done...

8 Chapter 8: Monitors and Blocking Synchronization

To be done...

9 Chapter 9: Linked Lists: The Role of Locking

To be done...

10 Chapter 10: Concurrent Queues and the ABA Problem

To be done...

11 Chapter 11: Concurrent Stacks and Elimination

To be done...

12 Chapter 12: Counting, Sorting, and Distributed Coordination

To be done...

13 Chapter 13: Concurrent Hashing and Natural Parallelism

To be done...

14 Chapter 14: Skiplists and Balanced Search

To be done...

15 Chapter 15: Priority Queues

To be done...

16 Chapter 16: Futures, Scheduling, and Work Distribution

To be done...

17 Chapter 17: Barriers

To be done...

18 Chapter 18: Transactional Memory

To be done...

19 更新日志

- 2013 年 10 月 14 日：更新至第 3 章。

参考文献

[Amdahl, 1967] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA. ACM. 6

[Burns and Lynch, 1993] Burns, J. and Lynch, N. (1993). Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171 – 184. 12

[Herlihy, 1991] Herlihy, M. (1991). Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13:124–149. 20

[Herlihy and Wing, 1990] Herlihy, M. P. and Wing, J. M. (1990). Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492. 19

[Kogan and Petrank, 2012] Kogan, A. and Petrank, E. (2012). A methodology for creating fast wait-free data structures. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 141–150, New York, NY, USA. ACM. 20

[Lameter, 2013] Lameter, C. (2013). Numa (non-uniform memory access): An overview. *Queue*, 11(7):40:40–40:51. 14

[Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565. 6

[Lamport, 1979] Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, C-28(9):690–691. 16

[Peterson, 1981] Peterson, J. L. (1981). Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116. 7

[Scott, 2013] Scott, M. L. (2013). *Shared-Memory Synchronization*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers. 14, 19

[Sorin et al., 2011] Sorin, D. J., Hill, M. D., and Wood, D. A. (2011). *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers. 15