经典互斥算法

郑思遥*

2013 年 1 月 4 日

摘要

本文用较为轻松的方式介绍了几个经典的互斥算法：Dekker 算法、Dijkstra 提出的算法、Peterson 算法和面包店算法，并简单地给出了每一个算法的正确性证明和相关的讨论。

1 互斥问题的定义

互斥（mutual exclusive）是我们现在进行多线程编程的一个基本工具。本文探寻分布式计算历史上的几个非常有名非常经典的互斥算法，尽管这些算法几乎是所有操作系统、分布式系统或多线程编程课本中必介绍的算法，可是由于这些算法由于性能问题已经被现代的算法或机制替代了，实际中不会有人使用这些算法。尽管如此，了解这些算法可以帮助我们理解同步领域的基本原理。此外，了解这些算法的正确性证明可还可以训练并发算法正确性推导的思维。

互斥问题最早是 Edsger W. Dijkstra[1] 在 1965 年的 ACM 通讯中正式提出的 [Dij65]。文中的问题描述如下：有 N 台计算机都在重复执行某项任务，这项任务中有一段称为 “临界区”，要求同一时刻这 N 台计算机中只能有一台在临界区中。计算机之间可以通过共享的存储空间协调访问临界区。写入和读取临界区的操作是不可分割的。解决方法必须满足以下特点：

1. N 台计算机是对称的，没有指定的优先级；
2. 对 N 台计算机的速度不做任何假设，甚至同一台计算机的速度也不一定是固定不变的；
3. 如果某台计算机在临界区之外停机了，那么不允许导致其他计算机阻塞；
4. 不允许发生 “After you”–“After you” 阻塞，也就是说，如果所有计算机都没有在临界区中，那么必须允许一个——只允许一个——计算机进入临界区。

这个问题看上去挺简单，其实不容易，因为每一台计算机每一次只能发出一次读写请求，按照现在的话说，就是在缺少 read-modify-write 这类原子操作的情况下要实现互斥是很难的。

在那个没有多核处理器或多处理器计算机的年代，一台 “计算机” 同时只能执行一个线程，所以原文中的 “计算机” 用现在的话来说就是一个处理器或多核处理器中的一个核心或一个硬件线程，“共享的存储空间” 可以是线程或进程间的共享内存，访问共享内存的方式可以是直接在总线上访问或通过互联网络访问或通过多核处理器内部的互联网络访问。为了统一术语，本文使用最常用的多线程编程使用的术语。

2 经典互斥算法

2.1 Dekker 算法

Dekker 算法是第一个正确地通过软件方法解决两线程互斥问题的算法，该算法最早由 Dijkstra 在 [Dij68a] 中描述，算法 1 就是 Dekker 算法。算法中的 *status*[1] 和 *status*[2] 分别表示两个线程的状态，状态可以取值 *competing* 和 *out*。如果状态取值 *competing*，表示当前线程处于正在竞争进入临界区的状态；如果状态取值为 *out*，则表示这个线程放弃尝试进入临界区或已经退出临界区。

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

[1] Edsger W. Dijkstra 是计算机发展历史上的重要人物，除了耳熟能详的计算最短路径的 Dijkstra 算法，他也是分布式计算领域的开创者，重要成果包括信号量和哲学家就餐问题。Dijkstra 卒于 2002 年，同年其 1974 年的论文 “Self-stabilizing systems in spite of distributed control” 被授予 ACM PODC（Symposium on Principles of Distributed Computing）最有影响力论文奖。为了纪念这位科学家，PODC 最有影响力论文奖在 2003 年更名为 Dijkstra 奖。

每一个线程都有一个私有变量 i 表示自己的 id，由于有两个线程，所以 i 的取值为 1 或者 2，因此 $other = 3 - i$ 取值为 2 或者 1，表示另一个线程的 id。*turn* 是一个共享变量，表示下一轮应该轮到谁进入临界区了。

算法 1 Dekker 算法

```
1:  status[i] = competing;
2:  while status[other] == competing do
3:      if turn == other then
4:          status[i] = out;
5:          wait until turn == i;
6:          status[i] = competing;
7:      end if
8:  end while
9:  临界区;
10: turn = other;
11: status[i] = out;
```

以第 9 行表示的临界区为界，之前的操作是试图进入临界区之前要执行的操作，相当于互斥锁中的 lock 操作，之后的操作是执行完临界区之后要执行的操作，相当于 unlock 操作。线程在试图进入临界区的时候，首先将自己的状态标记为正在竞争进入临界区。然后通过第 2–8 行的 **while** 循环反复检查对方是否放弃或结束执行临界区。如果对方没有竞争，则直接进入临界区。如果对方正在竞争，而且下一轮还轮到对方的话，那么自己首先放弃，然后在第 5 行等待对方将 *turn* 让与自己，然后自己再宣布进入竞争状态。如果下一轮即将轮到自己，那么回到 **while** 循环等待对方完成临界区操作。

首先理解这个算法为什么能实现两个线程的互斥。对于一个试图进入临界区的线程来说，如果另一个线程没有这个企图，那么很简单直接进入临界区即可。如果两个线程同时在竞争进入临界区，那么根据第 3 行的条件判断，只有一个线程能够进入这个 **if** 语句，进入的这个线程将自己的状态设置为放弃，所以给另一个线程进入临界区的机会。另一个线程进入了临界区之后，会将 *turn* 设置为对方，所以放弃竞争临界区的线程又可以重新竞争临界区。从这个描述可以看出这个算法可以保证两个线程的互斥。

再来看这个算法是否会导致一个线程*饿死*。假设线程 1 和线程 2 同时进入临界区，并且不失一般性假设 $turn = 1$，那么两个线程进入 **while** 循环，其中线程 2 进入 **if** 语句。线程 2 将 *status*[2] 设置为 *out*，然后等待 *turn* 变成 2。此时线程 1 可以退出 **while** 循环。线程 1 退出 **while** 循环之后访问临界区，之后将 *turn* 设置为 2。*turn* 的下一次变化只能由线程 2 来完成，所以线程 2 最终总能发现 *turn* 变成 2 了，因此可以将自己的状态设置为 *competing*，然后继续在 **while** 循环中观察线程 1 的状态。线程 1 有两种可能，要么在退出临界区之后将状态设置为 *out*，要么重新尝试进入临界区，此时也会阻塞在第 5 行等待，因为状态也设置为了 *out*，所以线程 2 必然会退出 **while** 循环进入临界区。从以上描述可以看出，两个线程无论如何也不会饿死，所以 Dekker 算法也是不会死锁的互斥算法。

2.2 Dijkstra 提出的算法

Dijkstra 在 [Dij65] 文中正式提出了并发编程中的互斥问题并给出了一个解决方法。和 Dekker 算法不同的地方在于，Dijkstra 提出的算法支持 N 个线程同时竞争临界区，算法如算法2所示[2]。这个算法使用了 2 个共享的数组 $b[1:N]$ 和 $c[1:N]$，N 表示最大的线程数。这两个数组的 $b[i]$ 和 $c[i]$ 只能由线程 i 写入，但是可以由所有线程读取。还有一个共享整数变量 k，满足 $1 \le k \le N$，所有线程都可以原子地读写这个变量。i 是线程的私有变量，表示自己的 id。$b[1:N]$ 和 $c[1:N]$ 的初始值都为 **true**，k 的初始值可以是 $[1:N]$ 之间的任意值，具体值无所谓。

算法 2 Dijkstra 提出的算法

```
1:  b[i] = false;
2:  loop
3:      if k ≠ i then
4:          c[i] = true;
5:          if b[k] == true then k = i;
6:      else
7:          c[i] = false;
8:          if ∀j ≠ i, c[j] == true then break;
9:      end if
10: end loop
11: 临界区;
12: c[i] = true; b[i] = true;
```

要理解这个算法要从这个算法的整体结构入手。原文 [Dij65] 中列出的算法代码使用的是 ALGOL 60 语言编写的，结构和缩进都比较混乱，还使用了 Dijkstra 他自己

[2] 本文不把这个算法称为 Dijkstra 算法，因为 Dijkstra 算法通常指的是求图中最短路径的算法。

后来都反对 [Dij68b] 的 **go to** 语句。所以这里列出的算法对原文的算法结构进行了一些调整，使其更适合当代的阅读习惯。从算法的结构可以看出，在临界区之前的 lock 部分包含一个循环，循环内部是一个 **if** 判断语句，根据条件将循环体分割为两部分。假设 N 个线程都在竞争临界区，根据第 3 行的条件，只有一个线程能进入 **else** 部分，即第 7–8 行，这个线程也就是 k 表示的那个线程。这个线程判断是否除了自己的 $c[i]$ 之外其他所有线程的 $c[i]$ 都为 **true**，如果满足的话则退出循环进入临界区。这个条件显然最后是可以满足的，因为所有其他的线程都在循环执行 **if** 条件满足的那一部分，因此一定会将自己的 $c[i]$ 设置为 **true**。当 k 表示的这个线程完成了临界区之后，会将 $c[i]$ 和 $b[i]$ 都设置为 **true**。这时，除了线程 k 之外的所有其他的线程的第 5 行的 **if** 条件都会满足，因此执行 $k = i$，试图将 k 设置为自己。最终 k 总会表示某一个正在竞争的线程，这个线程进入 **else** 部分循环，而且此时 $b[k]$ == **false**，所以其他失败的线程依然在第 4–5 行循环执行，并且还会争相将自己的 $c[i]$ 设置为 **true**，让抢到了 k 的线程能够尽快进入临界区。所以可以看出，Dijkstra 提出的算法解决了 N 个线程之间的互斥问题。

这个算法中的采用的变量名初看上去不好理解，实际上对照算法 1 中使用的变量名，数组 $b[1 : N]$ 和 $c[1 : N]$ 共同表示了线程的状态，整数 k 实际上等同于 $turn$。

Dijkstra 提出的算法有一个问题就是不能防止线程饿死。如果有一个线程获得了 k，而且这个线程反复试图访问临界区，假设这个线程很快进入 lock 部分的代码，将 $b[k]$ 设置为 **false**，那么其他线程有可能永远都在循环中等待。这个问题很快被 Knuth 发现了 [Knu66]（对，就是那位大名鼎鼎的 Knuth），并且 “tried out over a dozen ways to solve this problem” 才找到一个他认为正确的解决方法，他的算法中一个线程最多等待 $2^{N-1} - 1$ 轮就可以进入临界区。而之后 de Bruijn 对 Knuth 的算法稍加改进，使其等待轮数降低到 $\frac{1}{2}N(N-1)$[dB67]。又过了几年，Eisenberg 进一步改进，使得算法等待轮数降低到 $N - 1$[EM72]。这三个改进算法本文就不讲解了，有兴趣的话可以参阅参考文献。

2.3 Peterson 算法

Peterson 提出的 2 线程互斥算法的结构非常简单，打破了 “2 进程互斥问题需要复杂的算法和复杂的证明” 的神话 [Pet81]。Peterson 在同一篇文章中还将算法扩展到 N 个线程的情况，结构同样很简单。我们先看 2 线程的 Peterson 算法，如算法 3 所示。算法中 q 是一个二元组，$q[1]$ 由线程 1 操作，$q[2]$ 由线程 2 操作，两个值的初始值都为 **false**。$turn$ 是两个线程共享的一个整型变量，初始值可以是 1 或 2。

算法 3 Peterson 算法（2 线程）

```
1: q[i] = true;
2: turn = i;
3: wait until not q[3 − i] or not turn == i;
4: 临界区;
5: q[i] = false;
```

在 lock 操作中，线程首先将自己的 $q[i]$ 设置为 **true**，并且试图将 $turn$ 设置为自己。第 3 行的等待循环判断两个条件，如果有一个为真就进入临界区。两个线程不可能同时通过第 3 行的判断条件，因为如果两个线程都到达这一行了，那么 $q[1]$ 和 $q[2]$ 都为 **true**，所以只能同时满足判断的后一个条件，而 $turn$ 不可能同时为两个值，所以两个线程只能先后通过这个条件。假设线程 1 通过了这个条件判断，进入了临界区。那么此时必满足 $q[2]$ == **false** 和 $turn$ == 2 中至少一个条件。如果满足前者，说明线程 2 已经退出临界区或还没有开始竞争临界区，如果线程 2 开始竞争进入临界区，那么必然无法通过第 3 行的判断条件。如果满足后者，说明线程 2 已经执行完成了第 2 行，此时由于线程 1 仍然在临界区，所以 $q[1]$ 为 **true**，因此线程 2 仍然无法通过第 3 行的判断条件。因此，这个算法可以保证两个线程互斥访问临界区。另外也可以看出，这两个线程都不会饿死。假设线程 1 在第 3 行的循环等待，线程 2 有可能出现 3 种情况。第一，线程 2 不竞争临界区，那么 $q[2]$ == **false**，线程 1 很快退出循环；第二，线程 2 也在第 3 行的循环等待，前面说过了，两个线程不可能同时在这一行循环等待；第三，线程 2 反复试图进入临界区，在这种情况下，线程 2 很快会将 $turn$ 设置为 2，所以线程 1 很快会退出循环。综上，采用 Peterson 算法的线程不会饿死。

下面看 N 线程的 Peterson 算法，如算法 4 所示。这个算法使用了两个共享的数组 $q[1 : N]$ 和 $turn[1 : N - 1]$，初始值分别为 0 和 1。i 和 j 是线程的私有变量，i 表示当前线程的 id，N 表示线程的总数。

根据算法的思想，N 线程的 Peterson 算法又称过滤器（filter）算法。从算法结构上看，每一个线程都需要执行一个 $N - 1$ 次迭代的循环，每一次循环中都要等待第 4 行

算法 4 Peterson 算法（N 线程）

```
1: for j = 1 to N − 1 do
2:     q[i] = j;
3:     turn[j] = i;
4:     wait until (∀k ≠ i, q[k] < j) or turn[j] ≠ i;
5: end for
6: 临界区;
7: q[i] = 0;
```

的一个条件满足才能进入下一次迭代。线程在每一次迭代中，首先将数组 q 中表示自己的那一个位置 $q[i]$ 设置为当前自己所在的迭代编号 j，然后试图将这一级迭代的 $turn[j]$ 设置为自己的线程 id。如果有 N 个线程并发竞争进入临界区，那么在大家的第一次迭代中有一个线程 “胜出”，“胜出” 的线程将 $turn[1]$ 设置为自己的 id。从第 4 行可以看出，只有 “胜出” 的这个线程两个条件都无法满足，所以被卡在了这一行，也就是说，“胜出” 的线程实际上是在这一轮竞争中被 “过滤” 的线程。从以上描述可以看出，$turn[j]$ 表示的意义就是第 j 轮被过滤的线程。以此类推下去，每一轮竞争都过滤一个线程，那么到第 $N-1$ 轮竞争之后，只有 1 个线程能通过，那么这个线程进入临界区。当最终这个线程进入临界区之后，将相应的 $q[i]$ 设置为 0，此时最后一个被过滤的线程在第 4 行等待的第一个条件满足，并且在剩余的迭代中每次都满足第一个条件，从而顺利完成循环进入临界区，将自己的 $q[i]$ 设置为 0，接着倒数第二个被过滤的线程进入满足等待条件完成循环进入临界区，以此类推，直到第一个被过滤的线程也进入临界区。

下面看一个例子，假设 4 个线程并发竞争进入临界区。数组 q 和 $turn$ 的初始值如下所示：

	1	2	3	4
q	0	0	0	0
$turn$	1	1	1	

4 个线程并发竞争进入临界区，每一个线程都进入第 1 轮竞争，将自己的 $q[i]$ 设置为 1，然后争抢着设置 $turn[1]$，不失一般性，假设线程 3 最后成功地设置了 $turn[1]$，数组 q 和 $turn$ 的值如下所示：

	1	2	3	4
q	1	1	1	1
$turn$	3	1	1	

根据第 4 行的等待条件，线程 3 在第 1 轮被过滤，在这一行循环等待。而其他线程都能满足后一个条件进入 **for** 循环的下一轮迭代。假设在第 2 轮的竞争中，线程 2 被过滤了，那么数组 q 和 $turn$ 的值如下所示：

	1	2	3	4
q	2	2	1	2
$turn$	3	2	1	

这时只有线程 1 和线程 4 能进入下一轮迭代。假设在第 3 轮的竞争中，线程 4 被过滤了，那么数组 q 和 $turn$ 的值如下所示：

	1	2	3	4
q	3	2	1	3
$turn$	3	2	4	

现在只剩下线程 1 了，线程 1 完成第 4 行之后也完成了 **for** 循环的所有迭代，直接进入临界区。执行完临界区之后，将 $q[1]$ 设置为 0，数组 q 的值如下所示：

	1	2	3	4
q	0	2	1	3

数组 $turn$ 的值已经不重要，因为线程 4 在剩下的 **for** 循环迭代中都可以满足第 4 行中前一个条件，从而顺利完成 **for** 循环剩下的所有迭代，进入临界区，最后将 $q[4]$ 设置为 0，如下所示：

	1	2	3	4
q	0	2	1	0

同样，线程 3 也能继续完成 **for** 循环剩下的所有迭代，完成临界区之后将 $q[3]$ 设置为 0，最后轮到线程 1 完成所有的迭代进入临界区。至此，4 个线程按照被过滤的相反顺序依次互斥地访问了临界区。

N 线程的 Peterson 算法是否会导致线程饿死呢？可以用数学归纳法证明该算法不会饿死 [HS08]，书中的证明似乎有些冗余，这里简化了证明。

这里对竞争的轮数采用反向的数学归纳法证明。如果 Peterson 算法不会导致线程饿死，那么意味着每一个竞争临界区的线程最终都会进入临界区。由于采用反向的数学归纳法，所以基础命题是：参与第 $N-1$ 轮竞争的线程最终能进入临界区。参与第 $N-1$ 轮竞争的线程只有一个，这个线程一定能进入临界区，所以基础命题为真。

有了基础命题之后，下面要进行归纳假设：假设到达第 $j+1$ 轮或更高轮竞争的线程最终能进入临界区。基于这个归纳假设，要证明：到达第 j 轮竞争的线程最终能够进入临界区。利用反证法，假设线程 A 卡在了第 j 轮竞

争，那么 $q[A] == j$，因此在第 j 轮之前被卡住的线程都不可能继续前进到下一轮，而是都卡在第 4 行的循环等待。由于竞争轮数大于 j 的线程最终都进入临界区并退出临界区了，所以线程 A 在第 4 行等待的第一个条件会满足，所以线程 A 会继续进入第 $j+1$ 轮竞争，产生了矛盾，即线程 A 不会卡在第 j 轮竞争，所以线程 A 肯定也能最终进入临界区。

后来有人提出质疑，说各种文献和教科书中说 Peterson 算法不会饿死是不对的，并给出了一个场景证明 Peterson 算法也是有可能导致线程饿死的 [Ala03]，不过我觉得 [Ala03] 中描述的场景并不能说明问题，有兴趣的读者可以参阅这篇文献。

2.4 面包店算法

面包店（bakery）算法是 Leslie Lamport [3] 针对 Dijkstra 的算法以及一系列针对 Dijkstra 原始算法的改进算法提出的改进 [Lam74]。这个算法的思想非常简单，基于面包店给顾客提供服务的一种常用方法，就是每次顾客进店的时候都取一个号，店家依次给剩下顾客中号码最小的那位提供服务。要是在现在，这个算法估计会叫做银行取号算法了。算法 5 列出了面包店算法。算法中使用了两个共享数组 $flag[1:N]$ 和 $number[1:N]$，$flag[i]$ 表示线程 i 是否正在竞争临界区，$number[i]$ 表示线程 i 取到的号。算法中使用了二元组小于的关系，这个关系定义如下：如果 $a < c$，或者如果 $a == c$ 且 $b < d$，那么 $(a,b) < (c,d)$。

线程开始竞争临界区的时候，首先将自己标记为正在竞争临界区，然后从所有已有的号码中挑出最大的号，取这个号的下一个号作为自己的号码。由于取号的过程是并发的，所以可能会有多个线程得到同样的号码。线程取号之后进入第 3 行循环等待，等待其他所有的线程要么没有在竞争临界区，要么自己取到的号比别

算法 5 面包店算法

```
1: flag[i] = true;
2: number[i] = 1 + max(number[1], ..., number[N]);
3: wait until ∀k ≠ i, flag[k] == false or (number[i], i) < (number[k], k);
4: 临界区;
5: flag[i] = false;
```

的正在竞争临界区的线程取到的号都小。确定自己编号是最小的之后退出循环等待进入临界区。执行完临界区之后将自己标记为没有在竞争临界区。采用二元组比较 $(number[i],i) < (number[k],k)$ 是为了处理两个线程取号相同的情况，如果取号相同，则选择其中 id 较小的那个线程。

面包店算法能满足互斥，证明如下：假设不满足，那么假设线程 A 和 B 同时在临界区，可以假设 $(number[A],A) < (number[B],B)$，即线程 A 先进入临界区。根据第 3 行的条件，B 要么看到 $flag[A] == 0$，要么 $(number[B],B) < (number[A],A)$。而后者是不可能满足的，因为 A 先进入临界区。而由于 B 看到了 $flag[A] == 0$，说明 A 还没有领号，此时 B 已经领号了，那么 A 不可能比 B 更早进入临界区，所以和假设矛盾，证明该算法能满足互斥。

面包店算法也不会死锁，因为一定有线程的 $(number[i],i)$ 二元组最小，所以总能有线程进入临界区。

和前面提到的几个互斥算法不同的地方在于，面包店算法是公平的（fair）。所谓公平，就是说算法能满足先到先服务（first-come-first-served，FCFS）的性质。先到先服务的定义如下：将竞争临界区的算法（即 lock 算法）分为固定步数的区域和循环等待的区域，前者称为 doorway；如果先执行完 doorway 的线程能够先进入临界区，那么这个算法是先到先服务的算法。

后执行 doorway 的线程取到的号一定至少比先执行完 doorway 的线程取到的号大 1，所以后到的线程必须等待先到的线程执行完临界区才能满足第 3 行的条件进入临界区，因此面包店算法是先到先服务的公平算法。

面包店算法还有一个很重要的特点，这个特点就和 Lamport 分布式系统的专长有关了。根据 Lamport 的论文 [Lam74]，这个算法设计的目的是为了解决多计算机系统中有一个机器故障就会导致整个系统死锁的问题。这个算法要求，如果同一个内存位置同时发生读和写操作，那

[3] Leslie Lamport 是分布式计算领域的大牛，为分布式计算奠定了很多理论基础。Lamport 在 1978 年发表的论文 “Time, Clocks and the Ordering of Events in a Distributed System” 因为奠定了分布式系统中时间同步理论而在 2000 年获得了 ACM PODC（Symposium on Principles of Distributed Computing）最有影响力奖（这个奖项在 2003 年更名为 Dijkstra 奖）。Lamport 在 1979 年发表的论文 “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs” 中定义了并发编程中 “顺序一致性” 的概念。Lamport 在 1982 年的论文 “ The Byzantine Generals Problem” 中提出了了拜占庭将军问题的解决方法，这是分布式系统容错理论中的重要问题。Lamport 1989 年提出的用于解决不可靠计算机网络中一致性问题的 Paxos 算法因为后来用于 Google 的 Bigtable 而更显其影响力。此外，Lamport 就是 LaTeX 中的 “La”。

么只需要写操作正确执行，而读操作可以返回任意值。在分布式系统中，一个处理器随时都有可能发生故障。假设如果处理器发生了故障，那么这个处理器执行的程序会立即进入非临界区并且停止运行。在从发生故障到停机的时间段内，从这个处理器的内存访问值的时候可能会得到任意值。最终，完全停机之后从这个处理器得到的值应该等于 0。也就是说停机的处理器相当于没有在竞争处理器，所以其他正常运行的线程能够最终进入临界区。

参考文献

[Ala03] K. Alagarsamy. Some myths about famous mutual exclusion algorithms. *SIGACT News*, 34(3):94–103, September 2003.

[dB67] N. G. de Bruijn. Additional comments on a problem in concurrent programming control. *Commun. ACM*, 10:137–138, March 1967.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8:569–, September 1965.

[Dij68a] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[Dij68b] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, March 1968.

[EM72] Murray A. Eisenberg and Michael R. McGuire. Further comments on dijkstra's concurrent programming control problem. *Commun. ACM*, 15:999–, November 1972.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[Knu66] Donald E. Knuth. Additional comments on a problem in concurrent programming control. *Commun. ACM*, 9:321–322, May 1966.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17:453–455, August 1974.

[Pet81] James L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.