A New Solution of Dijkstra's Concurrent Programming Problem

作者：*Leslie Lamport 1974*

原文：

https://www.microsoft.com/en-us/research/publication/new-solution-dijkstras-concurrent-programming-problem/

译者：phylips@bmy 2018-9-28

译文：

[序： 本文描述了用于实现互斥执行的“面包店算法”。Lamport 发明了很多并发算法，但是在他看来“面包店算法”就像是上帝的发明，他只是发现了它。与其他基于共享内存的同步算法一样，面包店算法也需要一个进程能够从内存中读取一个 word，而这个 word 可能正被另一个进程进行写入。(每个内存位置只有一个进程进行写入，因此不存在并发写入的情况)。与之前的算法，甚至是在它之后的那些算法相比，“面包店算法”的特别之处在于在并发读写的情况下无论读出来什么样的 value 值它都可以正确工作。比如在写操作要把 value 值从 0 改成 1 时，并发的读可能获取了一个 value 值为 7456，该算法依然可以工作。在最初设计该算法的时候，Lamport 并没有想着要设计一个满足这一属性的算法。他也是在写下该算法的正确性证明后，才意识到该证明并不依赖于并发读写的情况下读操作会读到什么样的值。

不知道有多少人意识到这个算法的卓越性，Lamport 在麻萨诸塞州计算机协会的前同事 Anatol Holt 肯定是其中之一。在 Lamport 向他讲述该算法的证明及其上述特殊性时，他被震惊了，甚至有些怀疑证明是不是有问题，虽然一时之间也无法指出证明的错误之处，但是他总感觉一定还有问题。在有些书里，对该算法进行了弱化，在书里增加关于读写原子性的假设，并把这个算法仍叫做“面包店算法”。Lamport 认为这是很糟糕的一件事，虽然简化版没有正确性问题，但是应该标明是简化版，而不能与原版一样叫做“面包店算法”。

“面包店算法”的重要性在于它在不依赖于任何更底层的互斥机制的情况下实现了互斥执行。而之前的互斥算法，都假设对于同一内存位置的读写是原子的，实际上是在假设针对该内存位置的读写操作是互斥执行的。对于这些算法来说，不能说他们真正解决了互斥执行问题。在“面包店算法”之前人们认为互斥执行问题是不可解的—要实现互斥执行就必须要基于更底层的互斥执行机制，Brinch

Hansen 在 1972 年的一篇论文中明确提出了这一点，虽然 Lamport 在 1974 年就提出了“面包店算法”，但是这一误解直到 1990 年还存在。为此 Lamport 专门写了一篇文章 The Mutual Exclusion Problem Has Been Solved 澄清这一问题：并不是所有的互斥执行算法都依赖于更底层的互斥执行机制，比如“面包店算法”就是其中之一。当然这里并不是说其他的算法没有意义，对于实际算法来说这是一个合理的假设，因为大部分的多处理器内存系统都提供了这一保证。但是对于其他没有这种提供这种内存访问保证的系统(比如异步多处理器系统)来说，这些算法就失效了，该怎么实现互斥执行呢？这时候就需要“面包店算法”了。

原始论文本身并没有直接指出“面包店算法”是一个“真正的”互斥执行算法。这可能是因为 Lamport 也是后来才意识到该算法这一特性的重要意义。这篇论文对于 Lamport 自己来说也非常重要，他后来关于并发方面的很多贡献和研究都是源于该问题。也正是从“面包店算法”开始，提出了进程内部变量的概念—这些变量可以由多个进程读取，但是只能由一个进程写。也是从这个时候开始，他意识到这种算法可以很方便地进行分布式实现—某些变量属于某个进程，其他进程通过向 owner 发送消息进行读取即可。因此，“面包店算法”可以算得上是 Lamport 分布式算法研究的起点。

本文包含了一个细微但是严重的错误。在一个附加说明里面，声明我们可以认为针对单个 bit 的读写是原子的。这就意味着要求与写并发的读操作只能读到两个可能值中的一个。如果读到老值，就认为读在写之前，否则就是读在写之后。直到后来在此文的相关工作中，Lamport 才意识到这个问题。

如上内容来自 Lamport 自己关于本文的一个最新说明。
]

导引

Knuth、deBruijn 和 Eisenberg & McGuire 针对 Dijkstra 提出并解决的并发编程问题都给出了他们自己的解。一个使用信号量机制的更简单的解也已经被实现。但是这些解在应用到一个实际的多计算机系统时都有缺点：单个单元的故障会导致整个系统瘫痪。我们提出了一种简单的解决方案，可以在任意独立单元故障的

情况下整个系统仍然可以运行。

考虑相互之间只能通过共享内存进行通信的 N 个异步计算机系统。每个计算机循环运行一段由两部分组成的程序代码：一部分是临界区，一部分是非临界区。经过 Knuth 扩展后的 Dijkstra 问题是要编写出满足如下条件的程序：

1. 任意时刻最多只有一个计算机处于临界区
2. 每个计算机最终一定能够进入它的临界区，除非它停止运行了
3. 任意计算机都可能停止在它的非临界区，我们甚至对它们的运行速度不能做任何假设

在此前 Knuth 他们提出的 4 个解决方案中，都需要所有的计算机读写同一个变量 k，如果包含该变量的内存单元出问题，整个系统就会瘫痪。而基于信号量的解决方案也需要依赖一个单一的硬件组件。

在我们的解决方案中假设有 N 个处理器，每个拥有自己的内存单元。一个处理器可能会读取其他处理器的内存内容，但是它只会对自己的内存进行写入。该算法有一个非常特别的属性：对于针对同一内存位置的并发读写来说，只需要写入操作可以成功完成，而读操作可以读取到任意值。

一个处理器可能在任意时刻失败。我们假设在它失败时，它会立即进入非临界区并停止运行。此后再从它的内存单元里面读取，一段时间内可能会读到任意值。最终，任意针对它的内存单元的读取只能读到 0。(在实际实践中，某个计算机的失败可以通过特定时间长度内针对它的读请求无法被响应来检测出来)。

与前面的算法不同，我们的算法采用了先来先服务的模式。在某个计算机要进入临界区之前，它要先执行一个 loop-free 的代码段—具有固定执行步数。然后会确保它在其他处理器发出服务请求之前进入临界区。

该算法非常简单。它基于在面包店里面通常采用的一种服务模式，在该模式中进入面包店的顾客会首先分配一个编号。最小编号的持有者就是下一个要服务的对象。在我们的算法中，每个处理器选择它们自己的编号。处理器本身还会通过 1...N 进行命名。如果两个处理器具有相同的编号，就再比较它们自己的名称。

共享存储模块中包含如下变量：
integer array choosing [1 :N], number [1 :N]

choosing[i]和 number[i]位于处理器 i 的内存中，初始化值为 0。number[i]的取值范围没有上限。

下面是处理器 i 的程序代码。它必须从非临界区开始执行。Maximum 函数的参数可以以任意顺序读取，其对应的“<”关系运算，是通过两个值(number[i]和 processor i 的名称)进行比较。

```
begin integer j;
  L1: choosing [i] := 1;
      number[i] := 1 + maximum (number[1], . . . , number[N]);
      choosing[i] := 0;
      for j = 1 step 1 until N do
        begin
          L2: if choosing[j] ≠ 0 then goto L2;
          L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
                  i) then goto L3;
        end;
      critical section;
      number[i] := 0;
      noncritical section;
      goto L1;
end
```

processor i 可以在任意时刻失败，并从非临界区重启(重启时要确保 number[i]和 choosing[i]的值为 0)。但是如果有进程不断地失败重启，还是会导致系统死锁。

正确性证明

为了证明算法的正确性，我们首先做如下定义。当 choosing[i]=1 的时候，我们认为 processor i 进入了面包店的门道排队。从 choosing[i]被从 1 重置为 0 开始，直到它失败或者进入非临界区，在此期间我们认为 processor i 进入了面包店。算法正确性可以通过如下几个断言推断出来。可以看到这些证明并不依赖针对同一内存位置的读写并发的情况下会读到什么样的 value。

断言 1.如果处理器 i 和 k 都在面包店里面，并且 i 进入面包店是在 k 进入门道排队之前，那么 number[i]<number[k]。

证明：根据假设，在处理器 k 选择自己的 number[k]的当前 value 时，处理器 i 的 number[i]值已经赋值完毕。因此 number[k] >= 1+number[i]。

断言 2.如果处理器 i 进入了它的临界区，处理器 k 进入了面包店，并且 i != k。那么(number[i], i) < (number[k], k)。

证明：因为 choosing[k]只有两个值—0 或者非 0—我们可以认为在处理器 i 的角度看来，针对该该值的读写是瞬时完成的，不存在并发读写的情况。比如，在 choosing[k]的值正在从 0 改成 1 的时候，处理器 i 刚好触发了读操作，如果读到了 0，我们就认为读发生在写之前，如果读到了 1 就认为读发生在写之后。{!此处实际上就是假设了针对单个 bit 的读写是原子的，正如作者在后来的说明中最后提到的，这个假设是不必要的}。下面证明中使用的时间都是从处理器 i 的视图角度看到的。

令 TL2 代表处理器 i 最后一次执行 j=k 时 L2 指令时读取 choosing[k]时的时间，同时令 TL3 代表处理器 i 最后一次执行 j=k 时 L3 指令的开始时间，很明显 TL2 < TL3。在处理器 k 选择它的 number[k]当前值时，令 Te 代表它进入门道时的时间，Tw 代表它完成 number[k]的 value 写入的时间，Tc 代表它离开门道的时间，显然 Te < Tw < Tc。

在 TL2 时，choosing[k]等于 0。因此要么 TL2 < Te，要么 TL2>Tc。如果 TL2<Te，根据断言 1，number[i] < number[k]，因此这种情况下断言 2 成立。

如果 TL2>Tc，我们有 Tw < Tc < TL2 < TL3，因此 Tw < TL3。因此在 L3 开始执行的时候，可以读到 number[k]的当前值。同时因为这是处理器 i 最后一次执行 j=k 下的 L3，因此它此时一定认为(number[i], i) < (number[k], k)，否则它也不会进入临界区。因此在这种情况下，断言 2 也成立。

断言 3.假设只有一定数量的处理器故障会发生。如果没有处理器处于临界区，并

且有一个处理器处于面包店并且没有故障的话，最终该处理器一定会进入临界区。

证明：假设没有人进入临界区。那么肯定存在一个时间点，在这个时间点之后不会有人进入或离开面包店。此时，假设处理器 i 具有所有面包店内的处理器中最小的(number[i], i)，那么处理器 i 最终一定会结束 for 循环进入临界区。

断言 2 意味着任意时刻最多只有一个处理器可以进入它的临界区。断言 1 和断言 2 可以证明处理器是以先到先服务的方式进入临界区的。因此单个处理器不会被 block 住除非整个系统发生了死锁。断言 3 意味着只有处理器停在临界区内，或者是发生无限制的进程故障和重入时才会导致系统死锁。对于后者来说是以如下方式阻碍系统运行的，处理器 j 持续的失败重启，对于处理器 i 来说，极端情况下它可能总是发现 choosing[j]=1，然后一直被卡在 L2 上。