

Dijkstra 并发编程问题的新解法（面包店算法）

A New Solution of Dijkstra's Concurrent Programming Problem

Leslie Lamport, Massachusetts Computer Associates, Inc.

Concurrency: the Works of Leslie Lamport. 171–178. Oct. 2019. <https://doi.org/10.1145/3335772.3335782>
Originally published in *Comm. of the ACM*. 17(8), 453–455. Aug. 1974. <https://doi.org/10.1145/361082.361093>
<https://lamport.azurewebsites.net/pubs/bakery.pdf>

译者：Ying ZHANG. 2021-04.

<https://ying-zhang.gitee.io/time/1974-bakery-cn.pdf>
<https://ying-zhang.github.io/time/1974-bakery-cn.pdf>

提出了一个解决互斥问题的简单方法，该方案允许系统在任意单个组件失效的情况下继续运行。

关键词：临界区、并发编程、多进程、信号量

CR 类别：4.32

引言

对最初由 Dijkstra [4] 提出并解决的并发编程问题，Knuth [1]、deBruijn [2]、Eisenberg 和 McGuire [3] 也给出了解法。一个使用信号量的更简单的解法也已经实现 [5]。这些解法在真正的多计算机系统（而不是分时多处理器系统）中使用时有一个缺点：单个单元的故障会让整个系统停止。我们提出了一个简单的解法，它允许系统在任意单个组件失效的情况下继续运行。

算法

考虑仅通过共享内存相互通信的 N 台异步计算机。每台计算机都运行一个循环程序，有两个部分——**临界区**和**非临界区**。Dijkstra 问题（由 Knuth 扩展）是编写满足以下条件的程序：

1. 在任何时候，至多一台计算机可能处于临界区。
2. 每台计算机最终都必须能够进入其临界区（除非它停机了）。
3. 任意计算机都可能在其非临界区停机。

此外，不能对各计算机的运行速率做出任何假设。

[1–4] 的解法让所有 N 个处理器〔译注：processors，本文中处理器和计算机含义相同，可以理解为进程〕设置并检测（set and test）单个变量 k 的值。包含 k 的内存单元发生故障，将使系统停止运行。使用信号量也意味着依赖于单个硬件组件。

我们的解法假设有 N 个处理器，每个处理器都有自己的内存单元。一个处理器可以读取任意其它处理器的内存，但它只需要写入自己的内存。该算法有一个显著的特性：若对单个内存位置的读取和写入操作同时发生，则只需正确执行写入操作。读取操作可以返回**任意值**！

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by U.S. Army Research Office–Durham, under Contract No. DAHC04-70-C-0023. Author's address: Massachusetts Computer Associates, Inc., Lakeside Office Park, Wakefield, MA 01880.

处理器随时可能发生故障。我们假设当它故障时，它会立即转到非临界区并停止。之后可能会有一段时期，从它的内存中读取，返回的是任意值。最终，任何从其内存中的读取都必须返回零值（实践中，若计算机在指定的一段时期内未能响应读请求，则可能会被检测为故障了）。

与 [1-4] 的解法不同，我们的解法在以下意义上是先到先得的方法。当处理器想要进入其临界区时，它首先执行一个无循环的代码块——即具有固定数量执行步骤的代码块。之后它能确保进入其临界区，且先于任何其它稍后请求服务的处理器。

该算法非常简单。它在面包店很常用：顾客在进店时会得到一个号码。持有最小号码的成为下一个服务对象。在我们的算法中，每个处理器选择自己的号码。这些处理器被命名为 $1, \dots, N$ 。若两个处理器选择的号码相同，则名字较小的优先。

公共存储包括 **integer array** *choosing*[1 : *N*], *number*[1 : *N*]

choosing[*i*] 和 *number*[*i*] 在处理器 *i* 的内存中，且初始为 0。*number*[*i*] 的取值范围是无限的。这将在下面讨论。

以下是处理器 *i* 的程序。执行必须从非临界区开始。**maximum()** 函数的参数可以按任意顺序传入。有序整数对上的“小于”关系 $(a, b) < (c, d)$ 定义为 $a < c$ ，或 $a = c$ 且 $b < d$ 。

```

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

```

我们允许进程 *i* 随时故障，然后重新启动，进入其非临界区（且令 *choosing*[*i*] = *number*[*i*] = 0）。但是，若一个处理器不断故障并重启，则它可能会使系统死锁。

正确性证明

为了证明算法的正确性，我们首先做出以下定义。当 *choosing*[*i*] = 1 时，称处理器 *i* 在门廊（doorway）。从它将 *choosing*[*i*] 置为 0 开始，直到故障或离开其临界区，它一直在面包店内。该算法的正确性是由下面的断言推导出来的。请注意，对同一内存位置，与写入并发的读取，这些证明没有对读到的值做出任何假设。

断言 1. 若处理器 *i* 和 *k* 都在面包店内，且在 *k* 进入门廊之前，*i* 就进入了面包店内，则 *number*[*i*] < *number*[*k*]。

证明. 根据假设，当 *k* 正在选择其 *number*[*k*] 的值时，*number*[*i*] 已经选好了当前值。因此，*k* 必须选择 *number*[*k*] ≥ 1 + *number*[*i*]。□

断言 2. 若处理器 *i* 在其临界区，处理器 *k* 在面包店内，且 *k* ≠ *i*，则 (*number*[*i*], *i*) < (*number*[*k*], *k*)。

证明. 由于 $choosing[k]$ 只有两个值——0 和非 0——我们可以假设, 从处理器 i 的角度来看, 读取或写入该处内存是瞬时完成的, 且不会同时读取和写入。例如, 如果 $choosing[k]$ 正在被从 0 改为 1, 同时它也被处理器 i 读取, 若读到 0, 则认为先读取; 否则认为是先写入。证明中定义的所有时刻都是处理器 i 的视角。

设 t_{L2} 是处理器 i 在最后一次执行 L2 时, 读取 $choosing[k]$ 的时刻 ($j = k$), 设 t_{L3} 是 i 开始最后一次执行 L3 的时刻 ($j = k$), 因此 $t_{L2} < t_{L3}$ 。当处理器 k 在选择 $number[k]$ 的当前值时, 设 t_e 是它进入门廊的时刻, t_w 是它完成写入 $number[k]$ 值的时刻, t_c 是它离开门廊的时刻, 那么 $t_e < t_w < t_c$ 。

由于 $choosing[k]$ 在 t_{L2} 时刻为 0, 我们有 (a) $t_{L2} < t_e$ 或 (b) $t_c < t_{L2}$ 。对情况 (a), 断言 1 意味着 $number[i] < number[k]$, 因此断言 2 成立。

对情况 (b), 我们有 $t_w < t_c < t_{L2} < t_{L3}$, 所以 $t_w < t_{L3}$ 。因此, 从时刻 t_{L3} 开始, 执行语句 L3 期间, 处理器 i 读取了 $number[k]$ 的当前值。对 $j = k$, 由于 i 没有再次执行 L3, 它肯定发现 $(number[i], i) < (number[k], k)$ 。因此, 断言 2 在在这种情况下也成立。□

断言 3. 假设只有有限数量的处理器可能故障。若没有处理器在其临界区, 且面包店里存在一个没有故障的处理器, 则某个处理器最终必然能进入其临界区。

证明. 假设没有处理器进入其临界区。那么将有某个时刻, 在此之后不再有处理器进入或离开面包店。在此时刻, 假设面包店内的所有处理器中, 处理器 i 有最小值 $(number[i], i)$ 。那么处理器 i 最终必然完成 for 循环并进入其临界区。这与假设矛盾。□

断言 2 意味着, 任意时刻最多有一个处理器可以处于其临界区。断言 1 和 2 证明处理器以先到先得的方式进入其临界区。因此, 除非整个系统死锁, 否则无法阻塞单个处理器。断言 3 意味着系统死锁只能因为: 处理器在其临界区中停机, 或处理器不断地故障和重启。后者可以按如下方式卡住系统: 若处理器 j 不断地故障和重启, 某个倒霉的处理器 i 总是读到 $choosing[j] = 1$, 并在 L2 处永远循环。

补充说明

如果面包店内始终至少有一个处理器, 那么 $number[i]$ 的值可以变得任意大。这个问题不能通过任何简单的在有限整数集回转的方案来解决。例如, 给定任意的值 r 和 s , 若 $N \geq 4$, 则可能同时有 $number[i] = r$ 和 $number[j] = s$, 对于某些 i 和 j 。

幸运的是, 对任何真实的应用, 实际考虑都会对 $number[i]$ 的值设置一个上限。例如, 若处理器以最多每毫秒一个的速率进入门廊, 则运行一年后, 我们将有 $number[i] < 2^{35}$ ——假设读取 $number[i]$ 永远不会返回大于已写入的值。

无界的 $number[i]$ 确实提出了一个有趣的理论问题: 是否可以为有限个处理器找到一个算法, 使得它们以先到先得的方式进入其临界区, 且任意处理器都不能写入其它处理器的内存? 答案不得而知¹。

该算法可以从两方面推广: (1) 在某些情况下, 允许两个处理器同时处于临界区; (2) 修改先到先得的属性, 使更高优先级的处理器先得到服务。这将在以后的文章介绍。

总结

我们的算法为互斥问题提供了一个新的, 简单的解法。由于它不依赖于任何形式的集中控制, 因此与以前的解法相比, 它对组件故障的较不敏感。

1973 年 9 月 收稿; 1974 年 1 月 修订

¹我们最近发现了这样的算法, 但是它相当复杂

参考文献

- [1] Knuth, D.E. Additional comments on a problem in concurrent programming control. *Comm. ACM* 9, 5 (May 1966), 321–322.
- [2] deBruijn, N.G. Additional comments on a problem in concurrent programming control. *Comm. ACM* 10, 3 (Mar. 1967), 137–138.
- [3] Eisenberg, M.A., and McGuire, M.R. Further comments on Dijkstra’s concurrent programming control problem. *Comm. ACM* 15, 11 (Nov. 1972), 999.
- [4] Dijkstra, E.W. Solution of a problem in concurrent programming control. *Comm. ACM* 8, 9 (Sept. 1965), 569.
- [5] Dijkstra, E.W. The structure of THE multiprogramming system. *Comm. ACM* 11, 5 (May 1968), 341–346.

我的文章（节选）

Leslie Lamport

<https://lamport.azurewebsites.net/pubs/pubs.html#bakery>

27. Time, Clocks and the Ordering of Events in a Distributed System

Communications of the ACM, 17, 8 (August 1974), 453–455.

本文介绍了实现互斥的面包店 (bakery) 算法。我发明了许多并发算法。我觉得不是我发明了面包店算法，而是我发现了它。与所有共享内存同步算法一样，面包店算法要求一个进程能够读取内存中的一个字 (word)，而另一个进程正在写入它（每个内存位置只由一个进程写入，因此不会发生并发写入）。

与以前的任何算法，以及几乎所有后续算法不同，与写入并发的读取，无论得到的是什么值，面包店算法都能工作。若写入将值从 0 变为 1，则并发读取可能获得值 7456（假设 7456 是一个可能位于该内存位置的值）。该算法仍然有效。我没有试图设计具有此属性的算法。我是在写了正确性证明，并注意到了该证明并不取决于跟写入并发的读取所返回的值之后，才发现面包店算法有这个属性。

我不知道有多少人意识到这个算法是多么的了不起。也许比任何人都更认识到这一点的人是 Anatol Holt，他是我在马萨诸塞州计算机协会的前同事。当我向他展示算法及其证明并指出其惊人的属性时，他感到震惊。他拒绝相信这是真的。他找不到我的证明有什么问题，但他确信一定有缺陷。那天晚上他离开时决心要找到它。我不知道他什么时候终于接受了算法的正确性。

有几本书收录了该算法的简化版，其中读取和写入是原子操作，并将这些版本称为“面包店算法”。我觉得这很可悲。发表简化版没什么错，只要叫它简化版就好了。

面包店算法的重要之处在于，它实现了互斥，而不依赖于任何较低级别的互斥。以前的互斥算法假设内存位置的读取和写入是原子操作，就等于假设了对该位置的互斥访问。因此，假设原子读取和写入的互斥算法已经假定了较低级别的互斥。这样的算法并不能说真正解决了互斥问题。在面包店算法之前，人们认为互斥问题是无解的——只能通过使用较低级别的互斥来实现。Brinch Hansen 在 1972 年的一篇论文中正是这样说的。许多人显然仍然相信它（参见 [91]）。

论文本身并没有声明它是一个“真正的”互斥算法。这表明我直到后来才意识到该算法的全部意义，但我不记得了。

在发现面包店算法后的几年里，我对并发的了解都来自对它的研究。[25]，[33] 和 [70] 等论文是该研究的直接结果。在面包店算法中，我还引入了进程所属变量的概念——也就是说，此类变量可以被多个进程读取，但只能由一个进程写入。我从一开始就意识到，这种算法有简单的分布式实现，其中变量驻留在所属进程中，其它进程通过向所属进程发送消息来读取变量。因此，面包店算法标志着我对分布式算法研究的开始。

这篇论文包含一个小但重要的错误。在脚注中，它声称我们可以将单个位的读取和写入视为原子操作。它认为，与写入并发的读取必须获得两个可能的值之一；如果它得到旧值，我们认为读在写之前，否则就在写之后。直到后来，随着最终在 [70] (*On Interprocess Communication - Part I: Basic Formalism, Part II: Algorithms*) 中描述的工作，我才意识到这种推理的谬误。