



WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Español
فارسی
Français
Հայերեն
Italiano
日本語
Polski
中文

Edit links

[Create account](#) [Log in](#)

Article [Talk](#)

[Read](#) [Edit](#) [View history](#)

Lamport's bakery algorithm

From Wikipedia, the free encyclopedia
(Redirected from [Lamport's Bakery algorithm](#))



It has been suggested that *[Lamport's Distributed Mutual Exclusion Algorithm](#)* be **merged** into this article. ([Discuss](#)) *Proposed since October 2013.*



This article includes a [list of references](#), but **its sources remain unclear** because it has **insufficient inline citations**. Please help to [improve](#) this article by [introducing](#) more precise citations. *(December 2010)*

Lamport's bakery algorithm is a computer [algorithm](#) devised by computer scientist [Leslie Lamport](#), which is intended to improve the safety in the usage of shared resources among multiple [threads](#) by means of [mutual exclusion](#).

In [computer science](#), it is common for multiple threads to simultaneously access the same resources. [Data corruption](#) can occur if two or more threads try to write into the same [memory](#) location, or if one thread reads a memory location before another has finished writing into it. Lamport's bakery algorithm is one of many [mutual exclusion](#) algorithms designed to prevent [concurrent](#) threads entering [critical sections](#) of code concurrently to eliminate the risk of data corruption.

Contents [hide]

- 1 Algorithm
 - 1.1 Analogy
 - 1.2 Critical section
 - 1.3 Non-critical section
- 2 Implementation of the algorithm
 - 2.1 Definitions
 - 2.2 Code Examples
 - 2.2.1 Pseudocode
 - 2.2.2 PlusCal Code
 - 2.2.3 Java Code
 - 2.3 Discussion
- 3 See also
- 4 References
- 5 External links

Algorithm [\[edit\]](#)

Analogy [\[edit\]](#)

Lamport envisioned a bakery with a numbering machine at its entrance so each customer is given a unique number. Numbers increase by one as customers enter the store. A global counter displays the number of the customer that is currently being served. All other customers must wait in a queue until the baker finishes serving the current customer and the next number is displayed. When the customer is done shopping and has disposed of his or her number, the clerk increments the number, allowing the next customer to be served. That customer must draw another number from the numbering machine in order to shop again.

According to the analogy, the "customers" are threads, identified by the letter *i*, obtained from a global variable.

Due to the limitations of computer architecture, some parts of Lamport's [analogy](#) need slight modification. It is possible that more than one thread will get the same number when they request it; this cannot be avoided. Therefore, it is assumed that the thread identifier *i* is also a priority. A lower value of *i* means a higher priority and threads with higher priority will enter the [critical section](#) first.

Critical section [\[edit\]](#)

The critical section is that part of code that requires exclusive access to resources and may only be executed by one thread at a time. In the bakery analogy, it is when the customer trades with the baker and others must

wait.

When a thread wants to enter the critical section, it has to check whether now is its turn to do so. It should check the numbers of every other thread to make sure that it has the smallest one. In case another thread has the same number, the thread with the smallest i will enter the critical section first.

In [pseudocode](#) this comparison will be written in the form:

```
(a, b) < (c, d)
```

which is equivalent to:

```
(a < c) or ((a == c) and (b < d))
```

Once the thread ends its critical job, it gets rid of its number and enters the **non-critical section**.

Non-critical section [\[edit\]](#)

The non-critical section is the part of code that doesn't need exclusive access. It represents some thread-specific computation that doesn't interfere with other threads' resources and execution.

This part is analogous to actions that occur after shopping, such as putting change back into the wallet.

Implementation of the algorithm [\[edit\]](#)

Definitions [\[edit\]](#)

In Lamport's original paper, the *entering* variable is known as *choosing*, and the following conditions apply:

- Words choosing $[i]$ and number $[i]$ are in the memory of process i , and are initially zero.
- The range of values of number $[i]$ is unbounded.
- A process may fail at any time. We assume that when it fails, it immediately goes to its noncritical section and halts. There may then be a period when reading from its memory gives arbitrary values. Eventually, any read from its memory must give a value of zero.

Code Examples [\[edit\]](#)

Pseudocode [\[edit\]](#)

In this example, all threads execute the same "main" function, *Thread*. In real applications, different threads often have different "main" functions.

Note that as in the original paper, the thread checks itself before entering the critical section. Since the loop conditions will evaluate as *false*, this does not cause much delay.

```
0  // declaration and initial values of global variables
1  Entering: array [NUM_THREADS] of bool = {false};
2  Number: array [NUM_THREADS] of integer = {0};
3
4  lock(integer i) {
5      Entering[i] = true;
6      Number[i] = 1 + max(Number[1], ..., Number[NUM_THREADS]);
7      Entering[i] = false;
8      for (j = 0; j < NUM_THREADS; j++) {
9          // Wait until thread j receives its number:
10         while (Entering[j]) { /* nothing */ }
11         // Wait until all threads with smaller numbers or with the same
12         // number, but with higher priority, finish their work:
13         while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i))) { /*
nothing */ }
14     }
15 }
16
17 unlock(integer i) {
18     Number[i] = 0;
19 }
20
21 Thread(integer i) {
```

```

22     while (true) {
23         lock(i);
24         // The critical section goes here...
25         unlock(i);
26         // non-critical section...
27     }
28 }

```

PlusCal Code [\[edit\]](#)

We declare N to be the number of processes, and we assume that N is a natural number.

```

CONSTANT N
ASSUME N \in Nat

```

We define P to be the set $\{1, 2, \dots, N\}$ of processes.

```

P == 1..N

```

The variables num and flag are declared as global.

```

--algorithm AtomicBakery {
variable num = [i \in P |-> 0], flag = [i \in P |-> FALSE];

```

The following defines LL(j, i) to be true iff $\langle\langle \text{num}[j], j \rangle\rangle$ is less than or equal to $\langle\langle \text{num}[i], i \rangle\rangle$ in the usual [lexicographical ordering](#).

```

define { LL(j, i) == \/\ num[j] < num[i]
                \/\ /\ num[i] = num[j]
                /\ j =< i
}

```

For each element in P there is a process with local variables unread, max and nxt. Steps between consecutive labels p1, ..., p7, cs are considered atomic. The statement with (x \in S) { body } sets id to a nondeterministically chosen element of the set S and then executes body. A step containing the statement await expr can be executed only when the value of expr is TRUE.

```

process (p \in P)
  variables unread \in SUBSET P,
             max \in Nat,
             nxt \in P;
{
p1: while (TRUE) {
    unread := P \ {self} ;
    max := 0;
    flag[self] := TRUE;
p2:  while (unread # {}) {
        with (i \in unread) { unread := unread \ {i};
                             if (num[i] > max) { max := num[i]; }
        }
    };
p3:  num[self] := max + 1;
p4:  flag[self] := FALSE;
    unread := P \ {self} ;
p5:  while (unread # {}) {
        with (i \in unread) { nxt := i ; };
        await ~ flag[nxt];
p6:  await \/\ num[nxt] = 0
        \/\ LL(self, nxt) ;
        unread := unread \ {nxt};
    } ;
cs:  skip ; /* the critical section;
p7:  num[self] := 0;
}}

```

```
}
```

Java Code [\[edit\]](#)

```
1  int[] ticket = new int[threads]; // ticket for threads in line, n - number of
   threads
2  // Java initializes each element of 'ticket' to 0
3
4  boolean[] entering = new boolean[threads]; // true when thread entering in line
5  // Java initializes each element of 'entering' to false
6
7  public void lock(int pid) // thread ID
8  {
9      entering[pid] = true;
10     int max = 0;
11     for (int n : ticket) { if (n > max) { max = n; } } // find max in the array
12     ticket[pid] = 1 + max;
13     entering[pid] = false;
14     for (int i = 0; i < ticket.length; ++i)
15     {
16         if (i != pid)
17         {
18             while (entering[i]) { Thread.yield(); } // wait while other thread
   picks a ticket
19             while (ticket[i] != 0 && ( ticket[pid] > ticket[i] ||
20                 (ticket[pid] == ticket[i] && pid > i)))
21                 { Thread.yield(); }
22         }
23     }
24     // The critical section goes here...
25 }
26
27 public void unlock(int pid)
28 {
29     ticket[pid] = 0;
30 }
```

Discussion [\[edit\]](#)

Each thread only writes its own storage, only reads are shared. It is remarkable that this algorithm is not built on top of some lower level "atomic" operation, e.g. [compare-and-swap](#). The original proof shows that for overlapping reads and writes to the same storage cell only the write must be correct. The read operation can return an arbitrary number. Therefore this algorithm can be used to implement mutual exclusion on memory that lacks synchronisation primitives, e.g., a simple SCSI disk shared between two computers.

The necessity of the variable *Entering* might not be obvious as there is no 'lock' around lines 7 to 13. However, suppose the variable was removed and two processes computed the same `Number[i]`. If the higher-priority process was preempted before setting `Number[i]`, the low-priority process will see that the other process has a number of zero, and enter the critical section; later, the high-priority process will ignore equal `Number[i]` for lower-priority processes, and also enter the critical section. As a result, two processes can enter the critical section at the same time. The bakery algorithm uses the *Entering* variable to make the assignment on line 6 look like it was atomic; process *i* will never see a number equal to zero for a process *j* that is going to pick the same number as *i*.

When implementing the pseudo code in a single process system or under [cooperative multitasking](#), it is better to replace the "do nothing" sections with code that notifies the operating system to immediately switch to the next thread. This primitive is often referred to as `yield`.




Lamport's bakery algorithm assumes a sequential consistency memory model. Few, if any, languages or multi-core processors implement such a memory model. Therefore correct implementation of the algorithm typically requires inserting fences to inhibit reordering.^[1]

See also [\[edit\]](#)




- [Dekker's algorithm](#)
- [Eisenberg & McGuire algorithm](#)

- [Peterson's algorithm](#)
- [Szymanski's algorithm](#)
- [Semaphores](#)

References [\[edit\]](#)

1. [^] Chinmay Narayan, Shibashis Guha, S.Arun-Kumar [Inferring Fences in a Concurrent Program Using SC proof of Correctness](#) 
- [Original Paper](#) 
- On his [publications page](#) , Lamport has added some remarks regarding the algorithm.

External links [\[edit\]](#)

- [Wallace Variation of Bakery Algorithm](#)  which overcomes limitations of Javascript language
- [Lamport's Bakery Algorithm](#) 
- [Another JavaScript implementation](#)  by a.in.the.k

Categories: [Concurrency control algorithms](#)

This page was last modified on 2 June 2015, at 01:28.

Text is available under the [Creative Commons Attribution-ShareAlike License](#); additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

[Privacy policy](#) [About Wikipedia](#) [Disclaimers](#) [Contact Wikipedia](#) [Developers](#) [Mobile view](#)

