Create account Log in

Article Talk Read Edit View history Search

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
Deutsch
Español
فارسی
Français
한국어
Հայերեն
Italiano
Magyar
Nederlands
日本語
Polski
Português
Русский
Српски / srpski
Українська
中文
Edit links

# Peterson's algorithm

From Wikipedia, the free encyclopedia

**Peterson's algorithm** (AKA Peterson's solution) is a concurrent programming algorithm for mutual exclusion that allows two processes to share a single-use resource without conflict, using only shared memory for communication. It was formulated by Gary L. Peterson in 1981.[1] While Peterson's original formulation worked with only two processes, the algorithm can be generalized for more than two,[2] as shown below.

**Contents** [hide]

## The algorithm [edit]

The algorithm uses two variables, *flag* and *turn*. A *flag[n]* value of *true* indicates that the process *n* wants to enter the critical section. Entrance to the critical section is granted for process P0 if P1 does not want to enter its critical section or if P1 has given priority to P0 by setting *turn* to 0.

```
bool flag[0]   = false;
bool flag[1]   = false;
int turn;
```

```
P0:      flag[0] = true;
P0_gate: turn = 1;
         while (flag[1] &&
turn == 1)
         {
             // busy wait
         }
         // critical section
         ...
         // end of critical
section
         flag[0] = false;
```

```
P1:      flag[1] = true;
P1_gate: turn = 0;
         while (flag[0] &&
turn == 0)
         {
             // busy wait
         }
         // critical section
         ...
         // end of critical
section
         flag[1] = false;
```

The algorithm does satisfy the three essential criteria to solve the critical section problem, provided that changes to the variables `turn`, `flag[0]`, and `flag[1]` propagate immediately and atomically. The while condition works even with preemption.[1]

The three criteria are mutual exclusion, progress, and bounded waiting.[3]

Since turn can take on one of two values, it can replaced by a single bit, meaning that the algorithms requires only three bits of memory.[4]:22

### Mutual exclusion [edit]

P0 and P1 can never be in the critical section at the same time: If P0 is in its critical section, then flag[0] is true. In addition, either flag[1] is false (meaning P1 has left its critical section), or turn is 0 (meaning P1 is just now trying to enter the critical section, but graciously waiting), or P1 is at label P1_gate (trying to enter its critical section, after setting flag[1] to true but before setting turn to 0 and busy waiting). So if both processes are in their critical sections then we conclude that the state must satisfy flag[0] and flag[1] and turn = 0 and turn = 1. No state can satisfy both turn = 0 and turn = 1, so there can be no state where both processes are in their critical sections. (This recounts an argument that is made rigorous in.[5])

### Progress [edit]

Progress is defined as the following: if no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in making the decision as to which process will enter its critical section next. This selection cannot be postponed indefinitely.[3] A process cannot immediately re-enter the critical section if the other process has set its flag to say that it would like to enter its critical section.

### Bounded waiting [edit]

Bounded waiting, or bounded bypass means that the number of times a process is bypassed by another process after it has indicated its desire to enter the critical section is bounded by a function of the number of processes in the system.[3][4]:11 In Peterson's algorithm, a process will not wait longer than one turn for entrance to the critical section: After giving priority to the other process, this process will run to completion and set its flag to 0, thereby allowing the other process to enter the critical section.

### Filter algorithm: Peterson's algorithm for more than two processes [edit]

The filter algorithm generalizes Peterson's algorithm to $N > 2$ processes.[6] Instead of a boolean flag, it requires an integer variable per process, stored in a single writer/multiple reader (SMWR) atomic register, and $N-1$ additional variables in similar registers. The registers can be represented in pseudocode as arrays:

```
level : array of N integers
last_to_enter : array of N−1 integers
```

The `level` variables take on values up to $N-1$, each representing a distinct "waiting room" before the critical section.[6] Processes advance from one room to the next, finishing in room $N-1$ which is the critical section. Specifically, to acquire a lock, process $i$ executes[4]:22

```
for ℓ from 0 to N−1 exclusive
    level[i] ← ℓ
    last_to_enter[ℓ] ← i
    while last_to_enter[ℓ] = i and there exists k ≠ i, such that level[k] ≥ ℓ
        wait
```

To release the lock upon exiting the critical section, process $i$ sets `level[i]` to −1.

That this algorithm achieves mutual exclusion can be proven as follows. Process $i$ exits the inner loop when there is either no process with a higher level then `level[i]`, so the next waiting room is free; or, when `i ≠ last_to_enter[ℓ]`, so another process joined its waiting room. At level zero, then, even if all $N$ processes were to enter waiting room zero at the same time, no more than $N-1$ will proceed to the next room, the final one finding itself the last to enter the room. Similarly, at the next level, $N-2$ will proceed, etc., until at the final level, only one process is allowed to leave the waiting room and enter the critical section, giving mutual exclusion.[4]:22–24

The algorithm can also be shown to be starvation-free, meaning that all processes that enter the loop eventually exit it (assuming they don't stay in the critical section indefinitely). The proof proceeds by induction from $N-1$ downward. A process at $N-1$ is in the critical section, and by assumption will exit it. At all lower levels $\ell$, it is impossible for a process $i$ to wait forever, since either another process $j$ will enter the waiting room, setting `last_to_enter[ℓ] ← j` and "liberating" $i$; or this never happens, but then all processes $j$ that are also in the waiting rooms must be at higher levels and by the inductive hypothesis, they will eventually finish the loop and reset their levels, so that for all $k \neq i$, `level[k] < ℓ` and $i$ again exits the loop.[4]:24–25

Starvation freedom is in fact the highest liveness guarantee that the algorithm gives; unlike the two-process Peterson algorithm, the filter algorithm does not guarantee bounded waiting.[4]:25–26

## Note [edit]

When working at the hardware level, Peterson's algorithm is typically not needed to achieve atomic access.

Some processors have special instructions, like test-and-set or compare-and-swap, that, by locking the memory bus, can be used to provide mutual exclusion in SMP systems.

Most modern CPUs reorder memory accesses to improve execution efficiency (see memory ordering for types of reordering allowed). Such processors invariably give some way to force ordering in a stream of memory accesses, typically through a memory barrier instruction. Implementation of Peterson's and related algorithms on processors which reorder memory accesses generally requires use of such operations to work correctly to keep sequential operations from happening in an incorrect order. Note that reordering of memory accesses can happen even on processors that don't reorder instructions (such as the PowerPC processor in the Xbox 360).[*citation needed*]

Most such CPUs also have some sort of guaranteed atomic operation, such as XCHG on x86 processors and load-link/store-conditional on Alpha, MIPS, PowerPC, and other architectures. These instructions are intended to provide a way to build synchronization primitives more efficiently than can be done with pure shared memory approaches.

## Footnotes [edit]

1. ^ *a* *b* G. L. Peterson: "Myths About the Mutual Exclusion Problem", *Information Processing Letters* 12(3) 1981, 115–116
2. ^ As discussed in *Operating Systems Review*, January 1990 ("Proof of a Mutual Exclusion Algorithm", M Hofri).
3. ^ *a* *b* *c* Silberschatz. Operating Systems Concepts: Seventh Edition. John Wiley and Sons, 2005., Pages 194
4. ^ *a* *b* *c* *d* *e* *f* Raynal, Michel (2012). *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer Science & Business Media. ISBN 3642320279.
5. ^ F. B. Schneider. On Concurrent Programming, Springer Verlag, 1997, Pages 185–196
6. ^ *a* *b* Herlihy, Maurice; Shavit, Nir (2012). *The Art of Multiprocessor Programming*. Elsevier. p. 28–31. ISBN 9780123977953.

## See also [edit]

- Dekker's algorithm
- Eisenberg & McGuire algorithm
- Lamport's bakery algorithm
- Szymanski's algorithm
- Semaphores

## External links [edit]

- http://lxr.free-electrons.com/source/arch/arm/mach-tegra/sleep-tegra20.S Petterson's algorithm implementation

Categories: Concurrency control algorithms