

# Synchronization, Part 4: The Critical Section Problem

Alex Kizer edited this page on Mar 8 · 7 revisions

## What is the Critical Section Problem?

As already discussed in [Synchronization, Part 3: Working with Mutexes And Semaphores](#), there are critical parts of our code that can only be executed by one thread at a time. We describe this requirement as 'mutual exclusion'; only one thread (or process) may have access to the shared resource.

In multi-threaded programs we can wrap a critical section with mutex lock and unlock calls:

```
pthread_mutex_lock() - one thread allowed at a time! (others will have to wait he
... Do Critical Section stuff here!
pthread_mutex_unlock() - let other waiting threads continue
```

How would we implement these lock and unlock calls? Can we create an algorithm that assures mutual exclusion? An incorrect implementation is shown below,

```
pthread_mutex_lock(p_mutex_t *m)    { while(m->lock) {}; m->lock = 1;}
pthread_mutex_unlock(p_mutex_t *m)  { m->lock = 0; }
```

At first glance, the code appears to work; if one thread attempts to locks the mutex, a later thread must wait until the lock is cleared. However this implementation *does not satisfy Mutual Exclusion*. Let's take a close look at this 'implementation' from the point of view of two threads running around the same time. In the table below times runs from top to bottom-

Time	Thread 1	Thread 2
1	<code>while(lock) {}</code>	
2		<code>while(lock) {}</code>
3	lock = 1	lock = 1

Ooops! There is a race condition. In the unfortunate case both threads checked the lock and read a false value and so were able to continue.

## Candidate solutions to the critical section problem.

To simplify the discussion we consider only two threads. Note these arguments work for threads and processes and the classic CS literature discusses these problem in terms of

Pages 51

Home

#Example Markdown

#Informal Glossary

#Piazza: When And How to Ask For Help

C Programming, Part 1: Introduction

C Programming, Part 2: Text Input And Output

C Programming, Part 3: Common Gotchas

C Programming, Part 4: Debugging

Deadlock, Part 1: Resource Allocation Graph

Deadlock, Part 2: Deadlock Conditions

File System, Part 1: Introduction

File System, Part 2: Files are inodes (everything else is just data...)


File System, Part 3: Permissions


File System, Part 4: Working with directories


File System, Part 5: Virtual file systems


Show 36 more pages...


Clone this wiki locally


 Clone in Desktop











two processes that need exclusive access (i.e. mutual exclusion) to a critical section or shared resource.

Raising a flag represents a thread/process's intention to enter the critical section.

Remember that the psuedo-code outlined below is part of a larger program; the thread or process will typically need to enter the critical section many times during the lifetime of the process. So imagine each example as wrapped inside a loop where for a random amount of time the thread or process is working on something else.

Is there anything wrong with candidate solution described below?

```
// Candidate #1
wait until your flag is lowered
raise my flag
// Do Critical Section stuff
lower my flag
```

Answer: Candidate solution #1 also suffers a race condition i.e. it does not satisfy Mutual Exclusion because both threads/processes could read each other's flag value (=lowered) and continue.

This suggests we should raise the flag *before* checking the other thread's flag - which is candidate solution #2 below.

```
// Candidate #2
raise my flag
wait until your flag is lowered
// Do Critical Section stuff
lower my flag
```

Candidate #2 satisfies mutual exclusion - it is impossible for two threads to be inside the critical section at the same time. However this code suffers from deadlock! Suppose two threads wish to enter the critical section at the same time:

Time	Thread 1	Thread 2
1	raise flag	
2		raise flag
3	wait ...	wait ...

Ooops both threads / processes are now waiting for the other one to lower their flags. Neither one will enter the critical section as both are now stuck forever!

This suggests we should use a turn-based variable to try to resolve who should proceed.

## Turn-based solutions

The following candidate solution #3 uses a turn-based variable to politely allow one thread and then the other to continue

```
// Candidate #3
```

```
wait until my turn is myid
// Do Critical Section stuff
turn = yourid
```

Candidate #3 satisfies mutual exclusion (each thread or process gets exclusive access to the Critical Section), however both threads/processes must take a strict turn-based approach to using the critical section; i.e. they are forced into an alternating critical section access pattern. For example, if thread 1 wishes to read a hashtable every millisecond but another thread writes to a hashtable every second, then the reading thread would have to wait another 999ms before being able to read from the hashtable again. This 'solution' is not effective, because our threads should be able to make progress and enter the critical section if no other thread is currently in the critical section.

## Desired properties for solutions to the Critical Section Problem?

There are three main desirable properties that we desire in a solution the critical section problem

- Mutual Exclusion - the thread/process gets exclusive access; others must wait until it exits the critical section.
- Bounded Wait - if the thread/process has to wait, then it should only have to wait for a finite, amount of time (infinite waiting times are not allowed!). The exact definition of bounded wait is that there is an upper (non-infinite) bound on the number of times any other process can enter its critical section before the given process enters.
- Progress - if no thread/process is inside the critical section, then the thread/process should be able to proceed (make progress) without having to wait.

With these ideas in mind let's examine another candidate solution that uses a turn-based flag only if two threads both required access at the same time.

## Turn and Flag solutions

Is the following a correct solution to CSP?

```
\\ Candidate #4
raise my flag
if your flag is raised, wait until my turn
// Do Critical Section stuff
turn = yourid
lower my flag
```

One instructor and another CS faculty member initially thought so! However, analyzing these solutions is tricky. Even peer-reviewed papers on this specific subject contain incorrect solutions! At first glance it appears to satisfy Mutual Exclusion, Bounded Wait and Progress: The turn-based flag is only used in the event of a tie (so Progress and Bounded Wait is allowed) and mutual exclusion appears to be satisfied. However.... Perhaps you can find a counter-example?

Candidate #4 fails because a thread does not wait until the other thread lowers their flag. After some thought (or inspiration) the following scenario can be created to demonstrate

how Mutual Exclusion is not satisfied.

Imagine the first thread runs this code twice (so the the turn flag now points to the second thread). While the first thread is still inside the Critical Section, the second thread arrives. The second thread can immediately continue into the Critical Section!

Time	Turn	Thread #1	Thread #2
1	2	raise my flag	
2	2	if your flag is raised, wait until my turn	raise my flag
3	2	// Do Critical Section stuff	if your flag is raised, wait until my turn(TRUE!)
4	2	// Do Critical Section stuff	// Do Critical Section stuff - OOPS

## What is Peterson's solution?

Peterson published his novel and surprisingly simple solution in a 2 page paper in 1981. A version of his algorithm is shown below that uses a shared variable `turn` :

```
\\ Candidate #5
raise my flag
turn = myid
wait until your flag is lowered or turn is yourid
// Do Critical Section stuff
lower my flag
```

This solution satisfies Mutual Exclusion, Bounded Wait and Progress. If thread #2 has set turn to 2 and is currently inside the critical section. Thread #1 arrives, *sets the turn back to 1* and now waits until thread 2 lowers the flag.

Link to Peterson's original article pdf: [G. L. Peterson: "Myths About the Mutual Exclusion Problem", Information Processing Letters 12\(3\) 1981, 115–116](#)

## Was Peterson's solution the first solution?

No. Todo; Discuss Dekkers Algorithm

```
raise my flag
while(your flag is raised) :
    if it's your turn to win :
        lower my flag
        wait while your turn
        raise my flag
// Do Critical Section stuff
set your turn to win
lower my flag
```

## Can I implement Peterson's algorithm in C or assembler?

Yes - and it is used today in production for specific mobile processors: Peterson's algorithm is used to implement low-level Linux Kernel locks for the Tegra mobile processor (a system-on-chip ARM process and GPU core by Nvidia)

<https://android.googlesource.com/kernel/tegra.git/+/android-tegra-3.10/arch/arm/mach-tegra/sleep.S#58>

However in general, CPUs and C compilers can re-order CPU instructions or use CPU-core-specific local cache values that are stale if another core updates the shared variables. Thus a simple pseudo-code to C implementation is too naive for most platforms.

# How do we implement Critical Section Problem on hardware?

Good question. Next lecture...

Legal and Licensing information: Unless otherwise specified, submitted content to the wiki must be original work (including text, java code, and media) and you provide this material under a [Creative Commons License](#). If you are not the copyright holder, please give proper attribution and credit to existing content and ensure that you have license to include the materials.

