Software Solutions to Critical Sections

Hello, and welcome to this brief lesson on Software solutions for critical sections.

In an earlier lesson we learned about critical sections and how mutual exclusion can be realized using locks. In this first of two lesson we will investigate methods to implement locks. First, we will look at how this can be done in software, and in a following lesson we will explore how special CPU instructions can be used to efficiently implement locks as well.

We will set the stage by looking at a small number of naïve implementations of locks. This will help us better familiarize ourselves with mutual exclusion.

After a few false starts we will zoom in to a working algorithm, which was discovered in 1981 by Peterson.

We will then reason about a number of practical issues with this algorithm and how to address them. The first is that the algorithm is based on busy waiting, and the second is that the algorithm cannot directly be used in current CPU architectures.

At the end of this lesson you will have good working understanding of mutual exclusion, and you will understand some practical aspects of implementing software solutions to mutual exclusion.

Let's begin

Recall: Critical Sections & Locks

We remember from a previous lesson that critical sections are portions of code that must be executed in a mutually exclusive manner. We must therefore protect critical sections by appropriately bracketing them with code to enter and then to leave the critical section. This code, here marked in red, is responsible for enforcing mutual exclusion.

We also remember that this code can be easily implemented with the help of locks. If we had support for locks, for example in a class **Lock**, then one could associate a critical section with a lock. One could then enter the critical section by acquiring the lock and leave it by releasing the lock.

The question now is how to implement locks, for example, as a class with two functions, **lock** and **unlock**. In this lesson we will look at several software solutions for this.

Locks: Software Implementation 1

An intuitive, but overly naïve, solution would be to take turns.

The class **Lock** contains have a variable, called **turn**, which indicates which thread is allowed to acquire the lock next.

A thread acquires the lock by calling the function **lock()**.

In this function, the currently running thread keeps checking whether it is its turn to acquire the lock by comparing the value of the variable **turn** with its own thread index. It does this in a *while* loop, which gives rise to a behavior that is called *busy waiting*. The thread loops until some condition is true. In practice this is very problematic for a variety of reasons that we will discuss in detail later in this lesson.

How long does the thread loop? Well, until it is its turn. The function unlock releases the lock, and gives the turn to the next thread by incrementing the variable turn by one modulo the number of threads competing for the lock. We limit ourselves to two threads in this example. If turn was zero, it becomes one, and vice versa. If it was one, it now is zero again. We keep in mind that this solution in this form supports only two threads.

Let's see how this algorithm behaves, Thread 0 has the CPU, - and tries to acquire the lock.

It succeeds because the variable **turn** has value 0, and it is therefore Thread 0's turn. - Thread 0 enters the critical section. - After a while it gets preempted or blocked, and Thread 1 takes over. - Incidentally, it attempts to enter the critical section as well, - it keeps checking, but fails because it is still Thread 0's turn. (5) - At some point Thread 1 gets preempted, and Thread 0 takes over again. - It leaves the critical section and sets the **turn** variable to indicate that it is now Thread 1's turn. - Now the thread terminates. - and control goes back to Thread 1, - which returns to checking whether it is its turn. It is, and it enters and then leaves the critical section. Now it is Thread 0's turn again. (10) - After doing something else maybe, - Thread 1 attempts to enter the critical section again, - but now is locked forever, because Thread 0 terminated earlier, and won't be entering any critical section and upon leaving it set the turn back to Thread 1. - Thread 1 is blocked forever.

Clearly, this turn-based solution does not work.

Locks: Software Implementation 2

Let's look at a second solution.

This one has a local variable called **locked**, which indicates the state of the lock, that is whether the lock is locked or free.

In the function **lock()** the thread now, busy waits until the lock is free. Once the lock is free, the value of variable **locked** becomes false, and the thread drops through the while loop and immediately sets the state of the lock back to "locked".

The function unlock, simply sets the state back to free by setting the variable **locked** to false.

Let's illustrate what can go wrong with this solution.

Thread 0 is running, and attempts to acquire the lock. It first checks whether the lock is free. The lock is indeed free, but the thread gets preempted, and now Thread 1 takes over.

Incidentally, Thread 1 also wants to enter the critical section. It checks whether the lock's variable **locked** is still false to indicate that the lock is free. It is. Thread 1 now proceeds to acquire the lock by setting the variable **locked** to true to mark that the lock is locked, and it enters the critical section.

Inside the critical section Thread 1 may get blocked, and control goes back to Thread 0, which continues with its attempt to acquire the lock. And here this solution's problem starts to emerge. Thread 0 had checked the state of the lock earlier. Because it was free then it now continues with acquiring the lock. It is not aware that another thread has acquired the lock already. It sets variable **locked** to true, just as Thread 1 did earlier, and we now have two thread that own the lock.

This solution clearly does not work either.

It sometimes helps to think of this solution as a *check-and-then-set* solution. We check first, and then set. If the thread gets interrupted between the check and the set, the solution fails. Programmers that are new to multithreading get often tricked into using *check-and-then-set* approaches in their programs, which then results in all kinds of errors.

Locks: Software Implementation 3

Check-and-then-set did obviously not work. So, let's try the opposite, namely *set-and-then-check*.

For this we need an array of flags, which we call **busy**, one for each thread. To keep the implementation simple, we illustrate the solution for 2 threads only.

In the lock function, we first set the **busy** flag of the current thread to true, indicating that the current thread either intends to acquire the lock or that it owns the lock.

After we have set the flag, we check the other thread's flag to see whether the other thread intends to enter the lock or owns the lock outright. We do this by first determining the index of the other thread and then buy busy looping until the other threads flag is false. We release the lock, by clearing our flag.

Let's see how this "set-and-then-check" solution fares.

- Thread 0 is running and attempts to acquire the lock. It first sets its own flag to true.
- It then gets preempted, and Thread 1 takes over the CPU.
- Thread 1 also attempts to acquire the CPU. It sets the flag.
- Thread 1 now starts checking whether the lock is available, which it is not because Thread 0 set the flag earlier.
- Thread 1 keeps checking. (5)
- Until it gets preempted, and Thread 0 takes over again.
- Thread 0 now continues acquiring the lock, and starts checking whether the lock is available. Since Thread 1 set its flag earlier, Thread 0 fails to acquire the lock.
- At some point it gets preempted, and Thread 1 resumes.
- Nothing much has chanced since last time it ran, and so it keeps unsuccessfully checking for Thread 0's flag.
- As a result, we have a classical deadlock situation.

Apparently, *check-and-then-set* leads to mutual exclusion violations, and *set-and-then-check* to deadlocks.

Locks: Software Implementation 4

One software solution that works, at least conceptually, is Peterson's algorithm, which is an integration of the *set-and-then-check* approach with the turn-based approach that we explored at the very beginning.

In fact, the lock has a variable **turn**, and an array of flags, one for each thread. The function **lock** starts like the last, *set-and-then-check* solution by setting the flag of the current thread to true. It then determines the index of the other thread and sets the turn to the other thread. It then loops until the other thread does not own the lock and is not interested in the lock, or until it is our turn to acquire the lock. (5)

The lock is released, by setting the flag of the current thread to false.

We see that, fundamentally, this approach is no different than the *set-and-then-check* approach, except that it uses a turn variable to break the tie if both threads attempt to acquire the lock.

Peterson Algorithm: Considerations

Let's evaluate the Peterson's approach.

- First, It enforces mutual exclusion through the *set-before-check* approach.
- Second, it guarantees progress, meaning that threads wanting the acquire the lock are not prevented by threads who do not care about the lock, and the decision of who gets the lock does not take forever. This is because it breaks deadlocks by breaking ties using the turn-based approach.

Another benefit of this approach is that there is no starving. Other threads may not cut in front of a thread that is waiting to acquire a lock.

This algorithm has a number of problems, chief among them that it works for 2 threads only. There are other algorithms that are specifically designed for more than two threads: Filter algorithm and the Baker's algorithm.

Next, it relies on busy waiting.

Eliminating Busy Loop in Peterson's Algorithm

Let's go back to the implementation of the lock.

And let's focus on the busy waiting loop.

Busy waiting is problematic for a number of reasons, chief among them of course that it burns CPU bandwidth. Depending on what happens in the loop, it may burn memory bandwidth as well. Busy waiting is sometimes used in a controlled fashion in multiprocessor systems.

In single-processor systems busy waiting is particularly dumb, because the thread burns CPU time to wait for something that cannot possibly change until after the thread gives up the CPU and some other thread can take over and change the state of the system.

Why not think this through and give up the CPU immediately instead of burning up CPU time waiting until the scheduler preempts our thread?

In this way, there is no busy waiting. When the lock is busy, the thread gives up the CPU and tries again when it gets scheduled to run again.

Peterson Algorithm: Considerations

Another problem that the Peterson's algorithm shares with all other software based approaches, is that it relies on the memory operations being executed in the same way as they are specified by the algorithm.

This is often not the case in real systems. Compilers tend to re-arrange memory load and store operations to increase performance, and modern CPUs often execute memory operations out of order to prevent memory stalls. Multilevel caches further complicate the picture.

Peterson Algorithm and Memory Oder

How does this affect a software-based synchronization algorithm like Peterson?

Let's focus on the lock function.

For this, let's stretch it a bit, so that we can better distinguish the two parts of the code. Let's keep the *set-and-then-check* mantra in mind.

We first *set* by updating the flag and the turn.

We later *check* by reading the flag of the other thread and the turn variable.

In a multithreaded setting it is not guaranteed that we read the most recent values in the *check* part of the code. Maybe the compiler detects that the busy flag and the turn variable are not changed by computation in the loop, and therefore do not need to be reloaded. Or maybe the compiler simply loads the value of the busy flag ahead of the first loop iteration.

Similarly, the CPU may speculatively load memory locations ahead of their use. One can imaging other scenarios as well. This all of course becomes much more complicated in multiprocessor systems.

Fortunately, there are ways to force the compiler and the CPU to load the most current version of the memory location. This is done through what is called *memory fences*. After the memory fence we are sure that we read whatever has been written last to the memory location.

In the case of gcc compiler, the memory fence operation is called **sync__synchronize**, whose name indicates that we are synchronizing the memory.

We will see in a later lesson that CPUs have specific instructions to support thread synchronization in hardware. If we try to do it in software on modern CPUs we have to make judicious use of memory fences to ensure that our code is correct.

Conclusion - Software Solutions to Critical Sections

We have come to the conclusion of this brief lesson on Software Solutions for critical sections.

- We have formulated the critical section problem as a lock-implementation problem.
- We then looked at a sequence of software solutions to implement locks that all their problems.
- After a few failed attempts we looked at the Peterson's algorithm, which allows to correctly implement locks for a two-thread scenario.
- We looked at two practicality related problems of the Peterson's approach, which are typical of software solutions in general.

First was the problem of the algorithm busy waiting. This had to be eliminated in order to conserve CPU bandwidth. Fortunately, this was easy. Whenever a thread finds the lock busy, it gives up the CPU, and tries again when it gets to run again. In this fashion it does not unnecessarily burn CPU time. . The second problem is the need to rely on in-order memory operations. If the compiler or the CPU issue read and write operations out of order to optimize performance, this affects the correctness of the algorithm. Fortunately, one can – through so-called memory fences – force the compiler and the CPU to make sure that the next memory references return the most recent value of the memory locations.

We genuinely hope that you enjoyed this brief lesson on Software solutions for critical sections. In this lesson we have been exposed to several problems in multithreaded programming, and we are now in a good position to proceed to explore hardware supported synchronization.

Thank you for watching.