Hardware Solutions to Critical Sections

Hello, and welcome to this lesson on hardware-supported solutions for critical sections.

In an previous lesson we learned about how to implement locks to protect critical sections in software. Now we will explore together how CPU designers can support synchronization by providing appropriate hardware support.

A tried and true method to enforce mutual exclusion is by disabling interrupts. If interrupts are disabled, there is no way to preempt or otherwise interrupt the currently running thread, and mutual exclusion is ensured.

Designers realized early on that interrupts are a very blunt instrument for this, and so looked at other ways of how to support synchronization. Todays CPUs provide a variety of instructions to do just that. We will be looking at a subset, which includes *test-and-set*, *exchange*, *fetch-and-add*, and *compare-and-swap*. These are all called atomic instructions because they typically access or manipulate more than one memory location at once, and this sequence of accesses must not be interrupted.

We get an understanding of lock implementations based on atomic instructions by looking at how a *test-and-set*-based locks performs under stress on a multiprocessor system.

This will lead the way to a general idea of how to better implement these types of locks.

In this lesson we won't shy away from Assembly code, and by the end of this lesson you will be proficient with implementing synchronization primitives using hardware support available in modern CPUs.

Let's begin.

Recall: Critical Sections & locks

We previously encountered locks to protect critical sections of code. In order to enter a critical section, a thread has to acquire a lock, which it then gives up upon leaving the critical section.

Critical sections can therefore be realized by implementing locks.

We previously explored how to implement locks using software only. In this lesson we will learn a variety of ways to implement locks using hardware support, both for uniprocessor systems and for multicore and multiprocessor systems as well.

Simple Hardware Support: Disable Interrupts

Clearly the simplest way to enforce mutual exclusion is to disable interrupts.

By disabling interrupts, there is no way for the current thread of execution to be interrupted. Timers cannot force preemption; devices cannot trigger handlers, and so on. As a result, if the current thread disables all the interrupts, it is *de facto* in a critical section.

The obvious advantage of this approach is that it is very simple.

Disabling interrupts for critical sections has a number of problems, however.

First, disabling interrupts for any period of time may delay the time until the following interrupts can be handled. As a result, the *interrupt service* time increases. In some cases, interrupts can be also dropped, which can lead to all kind of problems.

In general, disabling interrupts can handle only one critical section. As a result, multiple non-related critical sections are handled as one, which may cause a lot of contention and blocking. Ways must be found to allow for independent critical sections to be controlled independently.

Finally, disabling interrupts to control access to critical sections does generally not work on multiprocessors.

Hardware Support: Atomic Operations

For all these reasons, processor designers have incorporated hardware support for synchronization in form of special CPU instructions. Because these instructions have to indivisibly perform multiple register and memory operations, they are called *atomic operations*.

Examples are *test-and-set*, *exchange*, *compare-and-swap*, and *load-link/store conditional*.

On a single core it is easy to ensure atomicity of these operations: the execution of this instruction cannot be interrupted. This is a bit more difficult on multicore or multiprocessor systems, where some thread on another processor can access or modify memory in a way that violates the atomicity of the operation. To avoid this, the bus is locked for the length of the operation. In this way, threads on other processors cannot access the memory, and can therefore not violate the atomicity of the atomic operation that operates on one or more memory locations.

Hardware Support: Test-and-Set (TAS)

Let's start exploring some of these atomic instructions. The simplest, and possibly the oldest one is the *test-and-set* instruction.

The *test-and-set* instruction is an uninterruptible instruction that mimics the a function that sets a given memory location to true and returns the value of the memory location before it was set to true.

Here is the pseudo-code of what this instruction does in hardware. One can think of it as storing the value of the memory location in a temporary variable, then setting the memory location to true, and finally returning the saved value.

This sequence of operations is executed by the hardware in an atomic fashion meaning it is not interruptible.

Locks with Test-and-Set

How do we build locks with test-and-set instructions? We follow the structure that we are familiar with from our exploration into software solutions for critical sections. We call this class **TASLock** (*test-and-set* lock), to indicate that we implement the lock with a *test-and-set* operation.

We store the state of the lock in a local variable **locked**, which indicates whether the lock is locked or free.

Releasing the lock is no magic. We simply set **locked** to false.

But how can we correctly acquire the lock?

The code for this is deceptively simple: we repeatedly call *test-and-set* lock with the variable **locked** as argument until the operation returns false. We remember from the description of *test-and-set* that the function sets the memory location to true and returns its previous value. Therefore, we fall through the while loop as soon as the previous value was false. The instruction did set the value to true however, so we now own the lock.

No other thread could have detected that the value was false and set it to true while we did this, because TAS is atomic. This simple piece of code therefore work correctly.

Locks with TAS: Assembly Implementation

While we described behavior of the *test-and-set* operation in pseudo-code, it is important to keep in mind that it is a CPU instruction. To drive this home, let's implement a lock in Assembly code using the *test-and-set* instruction.

The label **lock_acquire** is the start of the lock function.

We then issue the *test-and-set* instructions, which sets the value of variable **locked** to **true** and stores the previous value in register **reg**. We use the variable **locked** to store the state of the lock, similarly to how we implemented locks with *test-and-set* a minute ago.

We check whether the value in the register **reg** is zero, meaning **false**.

If **reg** is not zero, the result of that comparison is not zero, and then we jump back to the beginning and try again.

If **reg** is zero, the result of the comparison is zero, and we return from the function. We have acquired the lock.

The **lock_release** function is obvious. It simply sets the value of **locked** to zero and returns.

Analysis of TAS Locks

Let's have a brief look at the performance of this approach, in this case on a multiprocessor system. For this we plot the cost to acquire a lock against the number of threads competing for the lock. The baseline is a fictitious *ideal* lock, which has the same cost to acquire a lock independent of the number of threads. The curve is flat.

If we plot this curve for our *test-and-set*-locks, on the other hand, we notice that the cost increases dramatically as the number of threads increases. In order to understand why this is the case, we need to have a closer look at how *test-and-set* locks work.

For this we need to visualize a simple system.

We have a CPU, which runs a thread. The CPU is connected through a bus, to the main memory of the system. Let's assume that we have already created a *test-and-set* lock, and its **locked** state variable is stored in memory. It's value is 0.

Thread 0 now wants to acquire the lock and issues a first TAS instruction. We want to store a 1 in **locked** and remember what the previous value of **locked** was. (5)

At the beginning of the *test-and-set* operation the bus is blocked. The old value is copied to a local register and the value in memory set to 1. After this, the bus is unblocked.

The thread acquired the lock, which we mark here with a fat red frame. Until now no problem.

This changes when another thread joins on another CPU, however.

The other thread wants to enter the critical section as well, and so starts its own *test-and-set* while loop until it detects that the variable **locked** was false.

In every iteration, the bus is blocked, the old value copied into a local register, and a 1 pushed to memory.

This goes on for the entire duration of Thread 0 holding the lock.

Because the bus is repeatedly locked, it becomes difficult for threads on other CPUs to access the memory. This holds also for threads that want acquire the lock. They cannot access the bus and therefore cannot check the memory.

Note that caches don't help in this situation, because all read and writes must be to memory, not to the cached copies.

This situation becomes particularly bad when the thread that owns the lock wants to release it.

To do so it writes a zero into variable **locked**.

But the thread is no able to push the zero to memory because the bus is blocked all the time.

Both threads are unduly delayed because one thread cannot give up checking because the lock is busy while the other thread cannot give up the lock because it cannot go to memory.

This to a great extend explains why the *test-and-set* lock performs so poorly.

Improving Locks with Test-and-Set

So, how can it be improved? Let's look at its implementation again.

Locks with Test-and-Test-and-Set

And let's focus on the particular implementation of the function **lock**.

In this new implementation we keep trying, which we represent here as a forever loop, to do two things.

First, we wait until the variable **locked** is false. While it is true, we know for sure that somebody else has the lock. If it is false, we know that the lock either is available, or that it has been available very recently.

We therefore check whether we can get the lock. We do this with the *test-and-set* instruction. This sets variable locked to true and returns the value before we set it. If the instruction comes back with false, we know that we set the variable **locked** to true while it was false, and we own the lock; we can break out of the forever loop.

If the instruction comes back true, then somebody beat us to the lock, and we continue with the forever loop.

This form of lock is called a *TEST-and-test-and-set* lock, because we first test in a while loop and then *test-and-set*.

One may wonder why double looping should be any faster than the simple while loop that we had before. This is due to caches, as the following illustration shows.

Analysis of TAS Locks

We have the same scenario as before, with Thread 0 holding the lock, and Thread 1 wanting to acquire the lock. Instead of repeatedly using the *test-and-set* instruction, which has to go to memory, Thread 1 simply loops reading the current value of the variable **locked**.

The first time, the read goes to memory, and a local copy is cached.

From now on, the read goes to the cache, thus putting no burden on the bus.

When now Thread 0 wants to release the lock by setting **locked** to 0, it can easily do this.

As a result, the value in memory becomes 0.

Thread 1 keeps reading from the cached copy, which is wrong.

This is not a problem, because the cache-coherency protocol kicks in, and the new version of the variable is pushed to CPU1's cache.

Now Thread 1 reads the correct value, zero, and stops looping.

It enters step 2 in its forever loop, where it checks the value of the **locked** variable with a *test-and-set* instruction.

This blocks the bus, and in this case returns zero. Nobody else acquired the lock in the meantime. Thread 1 has acquired the lock.

We see that this solution puts much less load on the bus, as most of the read operations hit the cache. This makes it very easy for the holder of the lock to give it up.

Locks with Test-and-Test-and-Set

If we were to measure the performance of *test-and-test-and-set* locks (here in green), we would, as expected, see that the critical-section latency would be significantly reduced compared to basic test-and-set locks. The performance of *test-and-test-and-set* locks would be much closer to that of ideal locks.

As we explore the use of other atomic instructions for synchronization, we will describe basic versions of the locks, similar to the *test-and-set* lock. Keep in mind that these version can be further improved by taking advantage of local caches, just as in the *test-and-test-and-set* lock.

Hardware Support: Exchange (XCHG)

One common instruction that is very similar to *test-and-set* is the *Exchange* (XCHG) instruction.

It is sometimes also called *Swap* instruction because it atomically exchanges or swaps the values of two different memory locations.

Here we have the pseudo-code description of what this instruction does. Again, it swaps the values stored in the two memory locations, in an uninterruptible manner. On a multiprocessor, the memory bus is blocked during the length of the operation in order to prevent threads on other processors to interfere with the exchange.

To keep things simple we have the function exchange boolean values, but typical CPUs define exchange instructions for all types of values. Also, on most CPUs the two locations don't have to be necessarily both in memory. The instruction may also atomically exchange a value in memory with another value in a register.

Exchanging two values in registers can also be done, but is not particularly interesting in our case.

Locks with Exchange

How do we use the Exchange instruction to implement locks, which we aptly name **XCHGLock**?

In principle the implementation is very similar to that of a *test-and-set* lock. We have a local variable **locked**, which tells the state of the lock, just as before.

The **unlock** function simply sets **locked** to false.

The lock function mimics the implementation of the *test-and-set* instruction with the exchange instruction.

For this we set a dummy variable to true and exchange the value of **locked** and **dummy** until **dummy** becomes false. Because **dummy** is the result of the exchange instruction it contains the value of variable **locked** before the exchange. If the value was false before the exchange, it is now true, and the lock is acquired.

Again, all we do is mimic the *test-and-set* instruction with the help of a dummy variable that we initialize to true.

Hardware Support: Fetch-and-Add (FAA)

Another popular instruction is the *fetch-and-add* instruction, which atomically increments the value stored in a memory location and returns the value before the increment.

This is the pseudo-code of the instruction. We see that we temporarily store the value, then increment the value in the memory location by one, and then return previously stored value. This is all done without interruptions.

Locks with Fetch-and-Add (Ticket-Lock Algorithm)

The *fetch-and-add* instruction can be used in an interesting way to implement locks, using what is called the *ticket-lock* algorithm. We call this lock the *fetch-and-add* lock.

The lock contains two counters, one called **ticketnumber**, which is used to order the thread who want to acquire the lock, and the other is the **turn**, which indicates who is to be served next.

To acquire the lock, we grab a ticket. We do this by calling *fetch-and-add* on the ticket number of the lock. This returns the ticket number before the increment and then increments the number. This is our turn.

Then we loop, waiting until the variable turn is the same as our turn.

We release the lock by incrementing the turn variable. Now it is somebody else's turn.

Hardware Support: Compare-and-Swap (CAS)

Another popular instruction is the *compare-and-swap*, often also called *compare-and-exchange*.

Its operation is a bit tricky. It takes two values, called **old** and **new**, and a memory location. It then atomically compares the value in the location with **old**. If the two values are equal, the instruction assigns the value of **new** to the location and returns true. Otherwise it returns false.

This may become a bit clearer when we illustrate the operation of *compare-and-swap* in pseudo-code.

If the value of **old** is the same as the value stored in the location, then we store the value **new** into the location and return true. Otherwise we return false. This all happens atomically, of course.

This is a very strange instruction indeed. We will se how it can be used to implement locks. We will see in a later lesson how it can be used to implement so-called lock-free data structures as well.

Locks with Compare-and-Swap

Let's look at how we implement locks using *compare-and-swap*. We call this lock *compare-and-swap Lock*.

The structure is again similar to the *test-and-set* lock. The lock has a local variable **locked**, which reflects the state of the lock. The lock is released by setting **locked** to false, of course.

We acquire the lock by repeatedly invoking the *compare-and-swap* instruction in this fashion.

Let's look at it a bit closer: The *compare-and-swap* instruction compares the value in **locked** with the **old** value, which is false. If the two match, that is, the value of **locked** is false, then *compare-and-swap* assigns the value **new** – which is true - to **locked**, and then returns true. The lock is acquired.

If **old** and **locked** don't match, then **locked** is true, and *compare-and-swap* returns false. We continue with the while loop.

Compare-and-Swap on the x86 (simplified!)

How does this work on a real CPU, on the x86 for example.

The *compare-and-swap* is called *compare-and-exchange* on the Intel, and the notation is generally a bit different. We call the location **destination**, and the **new** value **source**. How does this work?

There is a hidden argument, the **accumulator**, which is the A, AX, or EAX register, depending on the size of the operands in the operations. The accumulator contains the **old** value. If the **old** value is equal to the value in the destination, then the source value is assigned to the destination, and a one is stored in the zero flag to indicate success.

If the accumulator and the destination don't match, the destination value is stored in the accumulator, which is a step that is typical to the x86, and a zero is stored in the zero flag to indicate failure.

The prefix "lock" is used to tell the CPU to lock the bus on a multiprocessor.

Locks with CAS: Assembly Implementation

Let's implement a *compare-and-swap* lock in Assembler using the x86 compare-and-exchange instruction.

Let the label **lock_acquire** be the entry point to acquire the lock.

First we store a 1 in register EDX.

The label **lock_retry** is used to loop.

We now store a 0 into the accumulator, which in this case is register EAX.

We now invoke the *compare-and-exchange* instruction by passing **locked** as destination, and the EDX register as the source. Since EAX contains 0, we are *de facto* calling the *compare-and-swap* instruction with 0 as **old**, 1 as **new**, and **locked** as the location.

If the "compare-and-exchange" instruction failed, the zero flag is 0, and we branch back to **lock_retry**.

If the *compare-and-exchange* succeeded, we return.

The lock is released by setting **locked** to 0.

Conclusion: Hardware Solutions to Critical Sections

After this good portion of Assembly code we have come to the end of this lesson on hardware solutions to critical sections.

We have learned that disabling interrupts can be used to enforce critical sections. In older or very simple systems, this is still used because of its simplicity, but interrupts have a number of disadvantages, among other that they don't work on multiprocessors, so they are not as popular anymore.

Instead, CPU designers provide a variety of atomic instructions to support synchronization. We learned about *test-and-set*, *Exchange*, *fetch-and-add*, *compare-and-swap*. We will encounter other instructions, such as *load-link/store conditional* later.

We have learned how to build locks using these instructions.

We were reminded that these are very expensive instructions because they have to go to memory instead of the cache and because they block the bus in multiprocessor systems.

We learned that all basic lock implementations can be easily improved by taking advantage of the cash, possibly by at the expense of additional loop operations. The performance improvements are significant.

We hope that you enjoyed this lesson on hardware supported synchronization. After this lesson you will be in a good position to effectively make use of sophisticated atomic instructions in support of synchronization on uniprocessor or multicore or multiprocessor systems

Thank you for watching.