Introduction

We observed one of the fundamental challenges in concurrent programming in the introduction to concurrency: we wanted to execute a set of instructions atomically, but we couldn’t because of interrupts on a single processor (or several threads executing on different processors concurrently).

In this section, we will address this issue directly by introducing a concept known as a lock. Programmers use locks to annotate source code, placing them around key parts to ensure that any critical piece runs as if it were a single atomic instruction.

The Basic Concept of Locks

For example, consider the canonical update of a shared variable:

Other crucial parts, such as adding an element to a linked list or updating shared structures, are possible, but we'll stick to this simple example for now. You add code around the critical part to employ a lock.

Because a lock is a variable, you must define one to utilize it (such as mutex above). This lock variable (or "lock" for short) holds the lock's state at any time. So no thread has the lock, nor has it been acquired (or locked or held), so only one thread has the lock and is presumably in a vital portion. A lock data type could also record information about the thread that has the lock or a queue for ordering lock acquisition, but such information is concealed from the lock user.

Lock() and unlock() have simple semantics. If no other thread holds the lock (i.e., it is free), the thread will acquire it and enter the critical area; this thread is frequently referred to as the lock's owner. If another thread calls lock() on the same lock variable (mutex in this case), it will not return while the lock is held by another thread, preventing other threads from approaching the critical region.

When the lock owner uses unlock(), the lock becomes available (free). As long as no other threads are waiting for the lock (no other thread has called lock() and is stalled), the lock is set to free. There will be a critical part if there are waiting threads (stuck in lock()).

Programmers can use locks to control some aspects of scheduling. Generally, we see threads as entities generated by the programmer but scheduled by the OS. By establishing a lock on a section of code, the programmer can ensure that only one thread can ever be active within that code. Thus locks help manage the anarchy of typical OS scheduling.

Pthread Locks

The POSIX library calls a lock a mutex because it is used to provide mutual exclusion between threads. Thus, the POSIX threads code below does the same thing as above (we utilize our wrappers that check for errors upon lock and unlock):

If you look closely, you’ll see that the POSIX version passes a variable to lock and unlock. A fine-grained locking mechanism protects various data and data structures with different locks, allowing more threads to be in locked code at the same time (a more fine-grained approach).

Building a Lock

We’ll need some help from our old friend, the hardware, as well as our good pal, the OS, to develop a working lock. A number of different hardware primitives have been added to the instruction sets of various computer architectures over the years; while we won’t go over how these instructions are implemented (that’s the subject of a computer architecture class), we will go over how to use them to construct a mutual exclusion primitive like a lock. We’ll also look at how the operating system plays a role to round out the picture and help us design a robust locking library.

Evaluating Locks

Before we design any locks, we need to know what our objectives are, therefore we question how to assess the efficacy of a given lock implementation. We should create some important criteria to determine whether a lock works (and functions well). The first is if the lock accomplishes its fundamental goal of mutual exclusion. Second, does the lock operate to keep several threads out of a critical section?

Fairness is the second factor to consider. Does each thread that competes for the lock have a reasonable chance of obtaining it once it becomes available? Another way to look at it is to consider the worst-case scenario: does any thread vying for the lock starve while doing so, never acquiring it?

The third criterion is performance, namely the time overheads incurred as a result of employing the lock. There are a few distinct scenarios to think about here. One is when there is no contention; what is the overhead when a single thread takes and releases the lock? Another scenario is when numerous threads compete for a single CPU's lock; are there performance issues in this case? Finally, how does the lock perform when there are numerous CPUs involved, each with multiple threads vying for the lock? We can better understand the performance impact of alternative locking strategies by comparing these diverse circumstances, as explained below.

Controlling Interrupts

One of the earliest solutions to mutual exclusion for single-processor systems was to silence interrupts for essential areas. The code would be:

Assume we are using a single CPU. To ensure that the code inside a critical part is not interrupted, we switch off interrupts (using a special hardware command) before entering it. After that, we re-enable interrupts (by a hardware command), and the program continues as usual.

This method's key benefit is its simplicity. It's not hard to figure out why this works. A thread may be assured that the code it executes will be executed without interruption.

Sadly, there are several drawbacks. First, we must trust that calling threads will not abuse privileges (like turning interrupts on and off) to make this method work. As you may be aware, we are in difficulty whenever we must trust an unknown program. A greedy software could call lock() at the start of its execution, monopolizing the processor, or an erroneous or malicious program could execute lock() and run into an eternal loop. In this instance, the OS never regains control, and the only option is to restart. Weakening interrupts needs too much faith in programs.

Second, it doesn't work with multiprocessors. For example, it cannot be confirmed if interrupts have been disabled or not if several threads are trying to enter the same critical section on different CPUs. As multiprocessors become more popular, our overall solution must improve.

Third, long-term interrupt disablement might result in lost interrupts, causing severe system issues. For example, consider what would happen if the CPU missed a disk device finishing a read request. How will the OS know to wake the waiting process?

Last but not least, this method is inefficient. Modern CPUs tend to run code that masks or unmasks interruptions slowly compared to typical instructions.

As a result, turning off interrupts is only used sparingly as a mutual exclusion mechanism. When accessing its own data structures, an operating system uses interrupt masking to ensure atomicity or avoid some messy interrupt handling circumstances. This usage makes sense because the OS trusts itself to perform privileged tasks anyway.

Just Using Loads/Stores

To establish a proper lock, we will need to rely on the CPU hardware and its instructions. First, define a simple lock using only a single flag variable. Using a single variable and accessing it via standard loads and stores is insufficient.

This first attempt (Figure 28.1) uses a primary variable (flag) to indicate if a thread has a lock. The first thread to enter the critical region will call lock(), which checks whether the flag is 1 (it isn't) and sets it to 1 if it currently holds the lock. After finishing the critical part, the thread runs unlock() to release the lock and clear the flag.

Calling unlock() and clearing the flag must be done while the first thread is in the crucial area. After that, the waiting thread exits the while loop, sets its own flag to 1, and enters the critical part.

Sadly, the code has two flaws: accuracy and performance. Once you become used to concurrent programming, the correctness issue becomes apparent. Assume the code interleaving in Figure 28.2.

We can rapidly generate a condition where both threads set the flag to 1 and enter the critical area using timely (untimely?) interrupts. Sadly, this is "bad" behavior — we have failed to supply the most basic requirement: mutual exclusion.

Performance issue. Sadly, this is how a thread waits for a lock that is already held: by repeatedly checking the value of flag. Waiting for another thread to release a lock wastes time. On a uniprocessor, the waiter thread cannot even run (until a context switch occurs)! So, as we develop more advanced systems, we should avoid this waste.

Building Working Spin Locks with Test-And-Set

System designers started incorporating locking hardware support because deactivating interrupts on several processors and utilizing basic loads and stores (as described above) doesn't work. Like the Burroughs B5000, early multiprocessor systems had this support; today, all systems do, even single CPU ones.

A test-and-set (or atomic exchange) instruction is the most direct hardware support. The test-and-set instruction is defined as follows in C code:

The test-and-set instruction performs this. It returns the old value and updates the old ptr. What matters is that this series of activities is atomic. It allows you to "test" the previous value (which is returned) while also placing a new value in the memory region; in this case, it creates a basic spin lock. Or better yet, figure it out!

Let's review why this lock works. For now, consider the case where a thread calls lock() and no other thread has the lock. When a thread does TestAndSet(flag, 1), the procedure returns the old value of flag, 0; the caller thread does not become stuck in the while loop and acquires the lock. Assuming the lock is now held, the thread will also atomically set the value to 1. After finishing its critical part, the thread invokes unlock() to reset the flag to 0.

The second scenario is when one thread already holds the lock (i.e., flag is 1). This thread will then call lock() and TestAndSet (flag, 1). This time, TestAndSet() will return 1 (since the lock is retained) while setting flag to 1. TestAndSet() will always return 1 if another thread has the lock; therefore, this thread will keep spinning until the lock is released. Finally, this thread will use TestAndSet() again upon receiving the lock, returning 0 while atomically updating the value to 1.

Assuring that only one thread acquires the lock is achieved by making both tests atomic. That's how to make a mutual exclusion primitive work! You may now see why this form of lock is called a spin lock. It is the most straightforward lock to construct and merely rotates until the lock becomes available. However, a single CPU requires a preemptive scheduler (i.e., one that will interrupt a thread via a timer to run a different thread from time to time). Without preemption, a thread spinning on a CPU will never yield a spin lock.

Evaluating Spin Locks

We can now evaluate our basic spin lock's effectiveness along our previously defined axes. A lock's correctness is critical: does it provide mutual exclusion? Yes, the spin lock only permits one thread into the vital portion at a time. So we have a good lock.

Then there's fairness. Was a spin lock fair to the thread? Can you ensure a waiting thread reaches the critical section? Sadly, spin locks do not guarantee justice. A thread in dispute may spin indefinitely. Thus, simple spin locks are unfair and may cause starving.

Lastly, performance. How much does a spin lock cost? Consider a few alternative scenarios to understand this better. Imagine threads competing for the lock on a single processor; imagine threads distributed across multiple CPUs.

Consider the case where the thread holding the lock gets preempted within a vital part. The scheduler may then run every other thread (imagine N1), with each trying to acquire the lock. In this situation, each thread will spin for a time slice before releasing the CPU, wasting CPU cycles.

However, spin locks function well on several CPUs (if the number of threads roughly equals the number of CPUs). The idea is as follows: Consider Thread A on CPU 1 and Thread B on CPU 2. Thread B will spin if Thread A (CPU 1) acquires the lock first (on CPU 2). But, since the critical part is short, the lock becomes available and is quickly acquired by Thread B. Spinning to wait for a lock held by another CPU saves cycles and is effective.

Compare-And-Swap

The compare-and-swap instruction (as it is known on SPARC, for example), or compare-and-exchange instruction, is another hardware primitive that some systems provide (as it called on x86). Figure 28.4 shows the C pseudocode for this single instruction.

The core principle behind compare-and-swap is to check whether the value at the ptr-specified address is equal to expected, and if it is, update the memory location pointed to by ptr with the new value. If not, don't do anything. Instead, return the original value at that memory location in either scenario, allowing the code running compare-and-swap to determine if it was successful or not.

We can construct a lock in a manner similar to that of test-and-set using the compare-and-swap instruction. We could, for example, simply replace the lock() routine with the following:

The rest of the code is identical to the last test-and-set example. This code operates similarly; it just checks if the flag is 0 and, if it is, atomically swaps in a 1 to obtain the lock. Threads that attempt to obtain the lock while held will become stuck spinning until the lock is released.

The code sequence (from) can be beneficial if you want to see how to construct a C-callable x86-version of compare-and-swap.

Finally, compare-and-swap is a more robust instruction than test-and-set, as you may have seen. We'll take advantage of this capability in the future when we look at subjects like lock-free synchronization. However, if we merely use it to make a simple spin lock, it behaves just like the spin lock we looked at earlier.

Load-Linked and Store-Conditional

Other platforms two instructions that work together to produce important areas. Build locks and other concurrent structures on MIPS using load-linked and store-conditional instructions. Pseudocode for these instructions in C is shown in ARM and PowerPC share instructions.

By loading a value from memory, the load-linked instruction fills a register. Only a prior store to the address succeeds (and modifies the value saved at the location just load-linked from). Failure returns 0 and leaves the value at ptr untouched.

Consider a load-linked and store-conditional lock. Then look at the code below for a quick fix. Go! Figure 28.6 provides the solution.

The lock() function is the only one. First, a thread awaits the flag 0 (and thus indicates the lock is not held). Second, threads must acquire the lock before entering the crucial area.

The store-conditional may fail. Lock() returns 0 since the lock is not held. Next, a thread enters the load-linked instruction, receives a 0, and continues. After completing the load-linked, two threads will attempt the store-conditional. For this reason, the second thread will not be able to obtain the lock and will have to attempt again.

For those who enjoy short-circuiting boolean conditionals, undergraduate student David Capel proposed a more compact form. Find out why it’s equal. It is shorter!

Fetch-And-Add

The fetch-and-add instruction atomically increments a value while returning the old value at a specific address. The fetch-and-add instruction in C is written as follows:

Mellor-Crummey and Scott implemented fetch-and-add to build a more engaging ticket lock. Figure 28.7 has the lock and unlock code.

Instead of a single value, this method combines a ticket and turn variable. To obtain a lock, a thread must first perform an atomic fetch-and-add on the ticket value, which is now considered this thread's "turn" (myturn). When (myturn == turn) for a thread, it is that thread's turn to enter the critical area. Unlocking is as simple as incrementing the turn so the next waiting thread (if any) can access the critical part.

Unlike our past attempts, this method guaranteed progress for all threads. So the thread will be scheduled later after a ticket is assigned (after the threads in front of it have passed through the critical section). Previously, a thread spinning on test-and-set might spin endlessly even as other threads acquired and released the lock.

Too Much Spinning

Our simple hardware-based locks are simple (just a few lines of code), and they work (you could even verify it by writing some code if you wanted to), which are two desirable qualities in any system or code. However, these solutions can be ineffective in other circumstances. Consider the case of two threads running on a single processor. Consider the case when one thread (thread 0) is in a vital area and hence has a lock in place, but is interrupted. The second thread (thread 1) now attempts to obtain the lock, but it is blocked. As a result, it starts to spin. And there's spin. It spins some more after that.

Finally, a timer interrupt occurs, thread 0 is rerun, releasing the lock,and thread 1 will be able to acquire the lock the next time it runs (say, the next time it runs). As a result, if a thread gets stuck spinning in a circumstance like this, it wastes an entire time slice checking a value that won't change! When N threads compete for a lock, the situation becomes even worse; N 1 time slices may be squandered in the same way, merely spinning and waiting for a single thread to release the lock. As a result, our next difficulty is to figure out how to create a lock that doesn't waste time spinning on the CPU.

Hardware support is insufficient to resolve the issue. We'll also require OS support! Let's have a look at how that might function.

Just Yield!

We achieved working locks and even (like with the ticket lock) fair lock acquisition. How to avoid threads spinning indefinitely waiting for the interrupted (lock-holding) thread to be rerun?
Our initial attempt is pleasant and straightforward: when you spin, provide the CPU to another thread. “Just yield, baby!”, said Al Davis. As shown in 28.8,

Assume a thread can invoke the operating system basic yield() to give up the CPU and let another thread run. A thread can be in one of three states: running, ready, or blocked. So the yielding thread de-schedule itself.

Consider the instance of two threads on one CPU; our yield-based strategy works well here. So, if a thread calls lock() and finds a lock held, it yields the CPU, allowing the other thread to continue and finish its essential part In this scenario, surrendering works wonderfully.

Consider now a situation where several threads (say 100) are competing for a lock. For example, if one thread gets the lock and is preempted before releasing it, the other 99 will run lock(), locate the lock, and yield the CPU. Assuming a round-robin scheduler, each of the 99 threads will follow this run-and-yield sequence before the lock thread runs again. A context transition can be expensive, and there is thus a lot of waste, even though it is better than spinning (which wastes 99 time slices).

Worse, we haven’t addressed the issue of hunger. Because of this, one thread may get stuck in a yield loop while others enter and escape the vital region. Clearly, we need a direct approach to this issue.

Using Queues: Sleeping Instead Of Spinning

Our earlier methods were flawed because they left too much to chance. So, if the scheduler chooses the wrong thread, it must either spin waiting for the lock or yield the CPU immediately (our second approach). In any case, waste is possible and starvation is unavoidable.

So, after the current holder releases the lock, we must explicitly restrict which thread receives it. We'll need greater OS support and a queue to track which threads are waiting for the lock.

For simplicity, we will use two Solaris calls: park() to put a calling thread to sleep and unpark(threadID) to wake it. They can be combined to form a lock that sleeps a caller if it attempts to acquire a held lock and wakes it when the lock is released. Examine the code in Figure 28.9 to see how such primitives might be used.

This sample does a few fascinating things. We first combine the old test-and-set concept with a lock waiter queue to improve lock efficiency. Second, we employ a queue to prevent lock hunger.

The guard is employed as a spin-lock around the flag and queue manipulations (Figure 28.9, page 16). As a result, if a thread is interrupted while getting or releasing a lock, other threads will spin-wait for it to resume. But the time spent spinning is restricted (just a few instructions inside the lock and unlock code, instead of the user-defined critical section). So this may be reasonable.

If a thread cannot acquire the lock (it is already held), we add ourselves to a queue (by calling the gettid() function to get the current thread's thread ID), set guard to 0, and yield the CPU. So, what if the guard lock was released after the park(), not before it? Hint: it's bad.

You may also notice that the flag is not reset when another thread is started. So what? It's not an error, but a need! When a thread is woken up, it appears to be returning from park(), but it does not keep the guard and hence cannot set the flag to 1. So we just pass the lock from thread to thread, releasing it to the next thread acquiring it, without setting the flag to 0.

In the solution, just before the call to park, there is a race condition (). A thread may be about to park, expecting it should sleep until the lock is released. However, switching to another thread (say, one holding the lock) at that point could cause issues, if that thread later relinquished the lock. The first thread's park would then slumber indefinitely (possibly), a phenomenon known as the wakeup/waiting race.

Solaris adds a third system call: setpark (). A thread can indicate it is about to park by invoking this function. If it is interrupted and another thread calls unpark before park, the subsequent park does not sleep. The code change inside lock() is minor:

Another option is to put the guard in the kernel. In that instance, the kernel could take steps to atomically release the lock and de-queue the thread.

Different OS, Different Support

We’ve seen one form of OS support for building a more efficient lock in a thread library. These features vary between OSes.
For example, Linux has a futex like Solaris but with additional in-kernel capabilities. Each futex has its own physical memory location and in-kernel queue. Callers can also use futex calls (explained below) to sleep or wake up.

There are two calls available. The call to futex wait(address, expected) puts the caller thread to sleep. If it’s not equal, the call ends. The futex wake(address) method wakes one thread on the queue. Figure 28.10 shows these calls in a Linux mutex (page 19).

I found this code excerpt from lowlevellock.h in the nptl library intriguing for several reasons. In the first place, it employs a single integer to track both the lock’s status (high bit) and the number of waiters on the lock (all the other bits). The lock is held if it is negative (because the high bit is set, and that bit determines the integer sign).

Second, the code snippet explains how to optimize for the normal case, where just one thread acquires and releases a lock (the atomic bit test-and-set to lock and an atomic add to release the lock).

Try to figure out how this “real-world” lock works. Do it and become a Linux locking master, or at least someone who follows instructions from a book.

Two-Phase Locks

Finally, the Linux approach resembles a more traditional strategy employed occasionally since the 1960s, possibly as early as Dahm Locks. Two-phase locks are currently the standard. A two-phase lock recognizes that spinning can help when releasing the lock. So, in the first phase, the lock spins, attempting to get the lock.

If the lock is not gained during the first spin phase, the caller is put to sleep and awakened when the lock becomes free. A generalization could spin in a loop for a specified amount of time before employing futex support to sleep.

Two-phase locks are another example of combining two good ideas to make a superior one. Of course, it depends on numerous factors, including hardware, thread count, and workload. Making a single general-purpose lock suited for all use cases is always a challenge.

Summary

The preceding approach demonstrates how modern locks are constructed: hardware support (in the form of a more robust instruction) combined with operating system support (e.g., park() and unpark() primitives on Solaris or futex on Linux). Of course, the specifics vary, and the code used to execute such locking is often fine-tuned. If you want additional information, read the Solaris or Linux code bases. Also also David et al.'s study, which compares locking techniques on current multiprocessors.