

Lecture 7 — Use of Locks, Lock Convoys, Atomics

Jeff Zarnett

Appropriate Use of Locking

In previous courses you learned about locking and how it all works, then we did a quick recap of what you need to know about it. And perhaps you were given some guidance in the use of locks, but probably in earlier scenarios it was sufficient to just avoid all the bad stuff (data races, deadlock, starvation). That's important but is no longer enough. Now we need to use locking and other synchronization techniques appropriately.

I like to say that critical sections should be as large as they need to be but no larger. That is to say, if we have some shared data that needs to be protected by some mutual exclusion constructs, we need to consider carefully where to place the statements. They should be placed such that the critical section contains all of the shared accesses, both reads *and* writes, but also does contain any statements. The ones that don't need to be there are those that don't operate on shared data.

This can mean that a block of code or contents of a function need to be re-arranged to move some statements up or down so they are no longer in the critical section. Sometimes control flow or other very short statements might get swept into the critical section being created to make sure all goes as planned but those should be the exception rather than the rule.

Let's consider a short code example from the producer-consumer problem. We have some global variables below that will be initialized as appropriate. There is also a definition of the function that will consume the data.

```
sem_t spaces;
sem_t items;
int counter;
int* buffer;
int pindex = 0;
int cindex = 0;
int ctotol = 0;
pthread_mutex_t prod_mutex;
pthread_mutex_t con_mutex;

void consume( int to_consume );
```

And then here is our single-threaded code for the consumer:

```
void* consumer( void* arg ) {
    while( ctotol < MAX_ITEMS_CONSUMED ) {
        sem_wait( &items );
        consume( buffer[cindex] );
        buffer[cindex] = -1;
        cindex = (cindex + 1) % BUFFER_SIZE;
        ++ctotol;
        sem_post( &spaces );
    }
}
```

To this we need to add some mutual exclusion if we want to allow multiple consumers at the same time. I'll leave aside the case of only allowing one consumer by putting the lock and unlock statements outside the while loop since that defeats the purpose of having multiple threads altogether. One approach we could take is that which allows exactly one consumer to run at a time, as below. But what's wrong with this?

```
void* consumer( void* arg ) {
    while( ctotol < MAX_ITEMS_CONSUMED ) {
        pthread_mutex_lock( &con_mutex );
        sem_wait( &items );
```

```

    consume( buffer[cindex] );
    buffer[cindex] = -1;
    cindex = (cindex + 1) % BUFFER_SIZE;
    ++ctotal;
    sem_post( &spaces );
    pthread_mutex_lock( &con_mutex );
}
}

```

What I recommend is of course to analyze this function one statement at a time and look into which of these access global variables. We're not worried about statements like locking, wait, or post, but let's look at the rest and decide if they really belong. Can any statements be removed from the critical section?

At first glance it is probably not very obvious but the consume function takes a regular integer, any old integer, not a pointer of some sort. So we could. inside the critical section, read the value of the buffer at the current index into a temp variable. That temp variable then can be given to the consume function at any time... outside of the critical section. Everything else inside our lock and unlock statements seems to be shared data: operates on cindex or ctotal.

```

void* consumer( void* arg ) {
    while( ctotal < MAX_ITEMS_CONSUMED ) {
        pthread_mutex_lock( &con_mutex );
        sem_wait( &items );
        int temp = buffer[cindex];
        buffer[cindex] = -1;
        cindex = (cindex + 1) % BUFFER_SIZE;
        ++ctotal;
        sem_post( &spaces );
        pthread_mutex_lock( &con_mutex );
        consume( temp );
    }
}

```

Next question then. With nothing left to take away, is there something left to add? Yes! The condition of the while loop checks the value of ctotal and that is a read of shared data. Now we maybe have a problem. How do we get that inside the critical section? One idea we might have is to read the value of ctotal into a temporary variable and use that, but it might cause some headaches with the timing (the end of the loop might be mispredicted...). Instead what I'd recommend is to make the loop a while true loop and then have a test of the value to determine when we should break out of the loop. See the example below, remembering of course there is the potential pitfall of forgetting to unlock the mutex if we are going to the break statement:

```

void* consumer( void* arg ) {
    while( 1 ) {
        pthread_mutex_lock( &con_mutex );
        if ( ctotal == MAX_ITEMS_CONSUMED ) {
            pthread_mutex_unlock( &con_mutex );
            break;
        }
        sem_wait( &items );
        int temp = buffer[cindex];
        buffer[cindex] = -1;
        cindex = (cindex + 1) % BUFFER_SIZE;
        ++ctotal;
        pthread_mutex_unlock( &con_mutex );
        sem_post( &spaces );
        consume( temp );
    }
    pthread_exit( NULL );
}

```

At this stage we should (mostly) be happy with the conversion of the function to support multithreaded operation. This conversion isn't the only way, but there are others. Remember, though, that keeping the critical section as small as possible is important because it speeds up performance (reduces the serial portion of your program). But that's not the only reason. The lock is a resource, and contention for that resource is itself expensive.

Lock Convoys

We'd like to avoid, if at all possible, a situation called a *lock convoy*. This happens when we have at least two threads that are contending for a lock of some sort. And it's sort of like a lock traffic jam. A more full and complex description from [Loh05]:

A lock convoy is a situation which occurs when two or more threads at the same priority frequently (several times per quantum) acquire a synchronization object, even if they only hold that object for a very short amount of time. It happens most often with critical sections, but can occur with mutexes, etc as well. For a while the threads may go along happily without contending over the object. But eventually some thread's quantum will expire while it holds the object, and then the problem begins. The expired thread (let's call it Thread A) stops running, and the next thread at that priority level begins. Soon that thread (let's call it Thread B) gets to a point where it needs to acquire the object. It blocks on the object. The kernel chooses the next thread in the priority-queue. If there are more threads at that priority which end up trying to acquire the object, they block on the object too. This continues until the kernel returns to Thread A which owns the object. That thread begins running again, and soon releases the object. Here are the two important points. First, once Thread A releases the object, the kernel chooses a thread that's blocked waiting for the object (probably Thread B), makes that thread the next owner of the object, and marks it as "runnable." Second, Thread A hasn't expired its quantum yet, so it continues running rather than switching to Thread B. Since the threads in this scenario acquire the synchronization object frequently, Thread A soon comes back to a point where it needs to acquire the object again. This time, however, Thread B owns it. So Thread A blocks on the object, and the kernel again chooses the next thread in the priority-queue to run. It eventually gets to Thread B, who does its work while owning the object, then releases the object. The next thread blocked on the object receives ownership, and this cycle continues endlessly until eventually the threads stop acquiring so often.

Why is it called a convoy? A convoy is when a grouping of vehicles, usually trucks or ships, travels all closely together. A freighter convoy, for example, might carry freight from one sea port to another. In this case, it means that the threads are all moving in a tight group. This is also sometimes called the "boxcar" problem: imagine that you have a train that is moving a bunch of boxcars along some railroad tracks. When the engine starts to pull, it moves the first car forward a tiny bit before it stops suddenly because of the car behind. Then the second car moves a bit, removing the slack between it and the next car. And so on and so on. The problem resembles this motion because each thread takes a small step forward before it stops and some other car then gets a turn during which it also moves forward a tiny bit before stopping. The same thing is happening to the threads and we spend all the CPU time on context switches rather than executing the actual code [Ost04].

This has a couple of side effects. Threads acquire the lock frequently and they are running for very short periods of time before blocking. But more than that, other, unrelated threads of the same priority get to run for an unusually large percentage of the (wall-clock) time. This can lead you to thinking that some other process is the real offender, taking up a large percentage of the CPU time. In reality, though, that's not the culprit. So it would not solve the problem if you terminate (or rewrite) what looks like offending process.

With that in mind, in Windows Vista and later versions, the problem is solved because locks are unfair. Unfair sounds bad but it is actually better to be unfair. Why? The Windows XP and earlier implementation of locks is a good explanation of why can go wrong. As you might imagine in a simple implementation of how locking works, if a lock l is unlocked by A and there is a thread B waiting, then the lock is modified so that it looks like B owns it, B is no longer blocked, and B already owns the lock when it wakes up. There was no period during which the lock was available and therefore it could not be "stolen" by some other thread that happened to come along at the right (or perhaps wrong) time [Duf06].

That doesn't sound like a bad thing until you realize that this means there is a period of time where the lock is held by B , but B is not running. In the best case scenario, after A releases the lock then there is a thread switch (the scheduler runs) and the context switch time is (in Windows, anyway, according to [Duf06] on the order of 4 000-10 000 cycles. That is a fairly long time but probably somewhat unavoidable. If, however, the system is busy

and B has to go to the back of the line it means that it might be a long time before B gets to run and that whole time, it is holding onto the lock. And really, at this point, would it be so bad to allow another thread C to sneak in there: if it wants the lock why should it not get it and release it before B gets its turn?

One of the ways in which one can then diagnose a lock convoy is to see a lock that has some nonzero number of waiting threads but nobody appears to own it. It just so happens that we're in the middle of a handover; some thread has signalled but the other thread has not yet woken up to run yet.

Changing the locks to be unfair does risk starvation, although one can imagine that it is fairly unlikely given that a particular thread would have to be very low priority and very unlucky. Windows goes give a thread priority boost, temporarily, after it gets unblocked to see to it that the unblocked thread does actually get a chance to run.

Although it can be nice to be able to give away such a problem to the OS developers and say "please solve this, thanks", that might not be realistic and we might have to find a way to work around it. We'll consider four solutions from [Loh05]:

- Sleep
- Share
- Cache
- Trylock

We could make the threads that are NOT in the lock convoy call a `sleep()` system call fairly regularly to give other threads a chance to run. This solution is lame, though, because we're changing the threads that are not the offenders and it just band-aids the situation so the convoy does not totally trash performance. Still, we are doing a lot of thread switches, which themselves are expensive as outlined above.

The next idea is sharing: can we use a reader-writer lock to allow much more concurrency than we would get if everything used exclusive locking? If there will be a lot of writes then there's limited benefit to this speedup, but if reads are the majority of operations then it is worth doing. We can also try to find a way to break a critical section up into two or more smaller ones, if that can be done without any undesirable side effects or race conditions.

The next idea has to do with changing when (and how) you need the data. If you shrink the critical section to just pull a copy of the shared data and operate on the shared data, then it reduces the amount of time that the lock is held and therefore speeds up operations. But you saw the first part of the discussion about critical section sizes, right? So you did that already...?

The last solution suggested is to use try-lock primitives: try to acquire the lock, and if you fail, yield the CPU to some other thread and try again. See the code below:

```
int retries = 0;
while(pthread_mutex_trylock( &lock ) != 0 ) { /* 0 indicates lock acquired */
    if ( retries < SPIN_LIMIT ) {
        retries++;
        sleep(0);
        continue;
    }
    pthread_mutex_lock( &lock );
    break;
}
```

In short, we try to lock the mutex some number of times (up to a maximum of `SPIN_LIMIT`), releasing the CPU each time if we don't get it, and if we do get it then we can continue. If we reach the limit then we just give up and enter the queue (regular lock statement) so we will wait at that point. You can perhaps think of this as being like waiting for the coffee machine at the office in the early morning. If you go to the coffee machine and find there is a line, you will maybe decide to do something else, and try again in a couple minutes. If you've already tried the come-back-later approach and there is still a line for the coffee machine you might as well get in line.

Why does this work? It looks like polling for the critical section. The limit on the number of tries helps in case the critical section belongs to a low priority thread and we need the current thread to be blocked so the low priority thread can run. Under this scheme, if *A* is going to release the critical section, *B* does not immediately become the owner and *A* may keep running and *A* might even get the critical section again before *B* tries again to acquire the lock (and may succeed). Even if the spin limit is as low as 2, this means two threads can recover from contention without creating a convoy [Loh05].

The Thundering Herd Problem. The lock convoy has some similarities with a different problem called the *thundering herd problem*. In the thundering herd problem, some condition is fulfilled (e.g., broadcast on a condition variable) and it triggers a large number of threads to wake up and try to take some action. It is likely they can't all proceed, so some will get blocked and then awoken again all at once in the future. In this case it would be better to wake up one thread at a time instead of all of them.

Atomics

What if we could find a way to get rid of locks and waiting altogether? That would avoid the lock convoy problem as well as any potential for deadlock, starvation, et cetera. In previous courses, you have learned about test-and-set operations and possibly compare-and-swap and those are atomic operations supported through hardware. They are uninterruptible and therefore will either completely succeed or not run at all. Is there a way that we could use those sorts of indivisible operations? Yes!

Atomics are a lower-overhead alternative to locks as long as you're doing suitable operations. Remember that what we wanted sometimes with locks and mutexes and all that is that operations are indivisible: an update to a variable doesn't get interfered with by another update. Remember the key idea is: an *atomic operation* is indivisible. Other threads see state before or after the operation; nothing in between.

We are only going to talk about atomics with sequential consistency. If you use the default `std::memory_order`, that's what you get. What do I mean by that? Well, in the header file `atomic` (C++11 here) there is an enumeration of memory orders and I am suggesting that using the default is pretty nice, compared to the alternative which may or may not be a Lovecraftian Horror to understand (or prove correctness). If you'd like to know about all the options, take a look at [C++15], but here's a quick summary from [Mil08a] (which is much more concise than the C++ Atomics listing):

Value	Explanation
<code>memory_order_acquire</code>	Subsequent loads are not moved before the current load or any preceding loads.
<code>memory_order_release</code>	Preceding stores are not moved past the current store or any subsequent stores.
<code>memory_order_acq_rel</code>	Combine the acquire and release guarantees
<code>memory_order_consume</code>	A potentially weaker form of <code>memory_order_acquire</code> that enforces ordering of the current load before other operations that are data-dependent on it (for instance, when a load of a pointer is marked <code>memory_order_consume</code> , subsequent operations that dereference this pointer won't be moved before it (yes, even that is not guaranteed on all platforms!)).
<code>memory_order_relaxed</code>	All reordering are okay; only atomicity is required of this operation.
<code>memory_order_seq_cst</code>	Same as <code>memory_order_acq_rel</code> , plus a single total order exists in which all threads observe all modifications in the same order.

The C++11 standard includes both strong and weak atomics. The weak ones are the ones where you get to specify the the memory ordering of load and store operating in a way that is not sequentially consistent. But we care about the standard, sequentially consistent kind of operation. *Don't* use relaxed atomics unless you're an expert! Basically, a value that is seen from a memory load may come from the past or from the future (it's all relative, of course). If you want to dig into the details about an example, I recommend [Mil08b], which goes into the details of just how difficult it is to prove correctness. If that doesn't talk you out of it, I'm not sure what will.

Atomic Flags. The simplest form of C++11 atomic is the `atomic_flag`. Not surprisingly, this represents a boolean flag. You can clear the flag and test-and-set it.

```
#include <atomic>

atomic_flag f = ATOMIC_FLAG_INIT;
int foo() {
    f.clear();
    if (f.test_and_set()) {
        // was true
    }
}
```

This returns the previous value. There is no assignment (=) operator for `atomic_flag`s. Although I guess in C++ you could define one if you wanted. This is kind of a dangerous thing about C++. If in C you see a line of code like `z = x + y`; you can have a pretty good idea about what it does and you can infer that there's some sort of natural meaning to the `+` operator there, like addition or concatenation. In C++, however, this same line of code tells you nothing, unless you know (1) the type of `x`, (2) the type of `y`, and (3) how the `+` operator is defined on those two operands *in that order*. But I'm digressing.

More general C++ atomics. Boolean flags are nice, but we want more. C++11 supports arbitrary types as atomic. Here's an example declaration:

```
#include <atomic>
atomic<int> x;
```

The C++11 library implements atomics using lock-free operations for small types and using mutexes for large types. The general types of operations that you can do with atomics are three: reads, writes, and RMW (read-modify-write) operations. C++ has syntax to make these all transparent.

```
// atomic reads and writes
#include <atomic>
#include <iostream>

std::atomic<int> ai;
int i;

int main() {
    ai = 4;
    i = ai;
    ai = i;
    std::cout << i;
}
```

If you want, you can also use `i = ai.load()` and `ai.store(i)`.

As for RMW operations, consider `ai++`. This is really

```
tmp = ai.read();
tmp++;
ai.write(tmp);
```

But, hardware can do that atomically. It can also do other RMWs: `+-`, `&=`, etc, `compare-and-swap`.

More info on C++11 atomics:

<http://preshing.com/20130618/atomic-vs-non-atomic-operations/>

We talked about C++11 atomics. Is there a pthread equivalent? Nope, not really.

OS X has atomics via OS calls:

<https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Multithreading/ThreadSafety/ThreadSafety.html>

The Linux kernel provides a number of atomic operations (but that doesn't really make them portable). Reference: <http://stackoverflow.com/questions/1130018/unix-portable-atomic-operations>

References

- [C++15] C++ Reference. `std::memory_order`, 2015. Online; accessed 6-December-2015. URL: http://en.cppreference.com/w/cpp/atomic/memory_order.
- [Duf06] Joe Duffy. Anti-convoy locks in Windows Server 2003 SP1 and Windows Vista, 2006. Online; accessed 5-December-2017. URL: <http://joeduffyblog.com/2006/12/14/anticonvoy-locks-in-windows-server-2003-sp1-and-windows-vista/>.
- [Loh05] Sue Loh. Lock Convoys and How to Recognize Them, 2005. Online; accessed 3-December-2017. URL: <https://blogs.msdn.microsoft.com/sloh/2005/05/27/lock-convoys-and-how-to-recognize-them/>.
- [Mil08a] Bartosz Milewski. C++ atomics and memory ordering, 2008. Online; accessed 6-December-2015. URL: <http://bartoszmilewski.com/2008/12/01/c-atomics-and-memory-ordering/>.
- [Mil08b] Bartosz Milewski. The inscrutable c++ memory model, 2008. Online; accessed 6-December-2015. URL: <http://bartoszmilewski.com/2008/12/23/the-inscrutable-c-memory-model/>.
- [Ost04] Larry Osterman. So you need a worker thread pool..., 2004. Online; accessed 4-December-2017. URL: <https://blogs.msdn.microsoft.com/larryosterman/2004/03/29/so-you-need-a-worker-thread-pool/>.