ECE459: Programming for Performance Winter 2018

Lecture 8 (v2) — Lock Convoys, Atomics, Lock-Freedom

Patrick Lam and Jeff Zarnett

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 [**?**]:

> 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 [**?**].

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 [**?**].

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 [**?**]) 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 does 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 [**?**]:

- 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 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 earlier 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 [**?**].

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.

The Lost Wakeup Problem. However! Waking up only one thread at a time has its own problems [1]. For instance, in the Java context, you can choose to wake up one waiting thread with either `notify()` or all waiting threads with `notifyAll()`. If you use `notify()`, then you can encounter the `lost wakeup` problem.

If all of your threads are identical, then you can use `notify()`. Otherwise, use `notifyAll()`, and brave the thundering herds. Or, in Java, use a `java.util.concurrent.locks.Condition`.

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 [**?**], but here's a quick summary from [**?**] (which is much more concise than the C++ Atomics listing):

[1] `https://stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again`

Value	Explanation
`memory_order_acquire`	Subsequent loads are not moved before the current load or any preceding loads.
`memory_order_release`	Preceding stores are not moved past the current store or any subsequent stores.
`memory_order_acq_rel`	Combine the acquire and release guarantees
`memory_order_consume`	A potentially weaker form of memory_order_acquire 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 memory_order_consume, subsequent operations that dereference this pointer won't be moved before it (yes, even that is not guaranteed on all platforms!).
`memory_order_relaxed`	All reordering are okay; only atomicity is required of this operation.
`memory_order_seq_cst`	Same as `memory_order_acq_rel`, 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 [**?**], 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'a 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 equivalent to (but faster than):

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

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`

Lock-Freedom

Let's suppose that we want to take this sort of thing up a level: we'd like to operate in a world in which there are no locks. Research has gone into the idea of lock-free data structures. If you have a map, like a HashMap, and it will be shared between threads, the normal thing would be to protect access to the map with a mutex (lock). But what if the data structure was written in such a way that we didn't have to do that? That would be a lock-free data structure.

It's unlikely that you want to use these sorts of things everywhere in your program. For a great many situations, the normal locking and unlocking behaviour is sufficient, provided one avoids the possibility of deadlock by, for example, enforcing lock ordering. We likely want to use it when we need to guarantee that progress is made, or when we really can't use locks (e.g., signal handler), or where a thread dying while holding a lock results in the whole system hanging.

Before we get too much farther though we should take a moment to review some definitions. I assume you know what blocking functions are (locking a mutex is one) and that you also have a pretty good idea by now of what is not (spinlock or trylock behaviour).

The definition of a non-blocking data structure is one where none of the operations can result in being blocked. In a language like Java there might be some concurrency-controlled data structures in which locking and unlocking is handled for you, but those can still be blocking. Lock-free data structures are always inherently non-blocking, but that does not go the other way: a spin lock or busy-waiting approach is not lock free, because if the thread holding the lock is suspended then everyone else is stuck [**?**].

A lock-free data structure doesn't use any locks (duh) but there's also some implication that this is also thread-safe; concurrent access must still result in the correct behaviour, so you can't make all your data structures lock-free ones by just deleting all the mutex code. Lock free also doesn't mean it's a free-for-all; there can be restrictions, like, for example, a queue that allows one thread to append to the end while another removes from the front, although two removals at the same time might cause a problem [**?**].

The actual definition of lock-free is that if any thread performing an operation gets suspended during the operation,

then other threads accessing the data structure are still able to complete their tasks [**?**]. This is distinct from the idea of waiting, though; an operation might still have to wait its turn or might get restarted if it was suspended and when it resumes things have somehow changed. If you learned about compare-and-swap then you might have some idea about this already: you try to do the compare-and-swap operation and if you find that someone changed it out from under you, you have to go back and try again. Unfortunately, going back to try again might mean that threads are frequently interrupting each other...

For this you might need wait-free data structures. This does not mean that nothing ever has to wait, but it does mean that each thread trying to perform some operation will complete it within a bounded number of steps regardless of what any other threads do [**?**]. This means that a compare-and-swap routine as above with infinite retries is not wait free, because a very unlucky thread could potentially take infinite tries before it completes its operations. The wait free data structures tend to be very complicated...

Let's consider some examples from [**?**]. First, a lock-free algorithm:

```
void stack_push(stack* s, node* n) {
    node* head;
    do
    {
        head = s->head;
        n->next = head;
    }
    while ( !atomic_compare_exchange(s->head, head, n) );
}
```

A particularly unlucky thread might spend literally forever spinning around the do-while loop as above, but that's okay because that thread's bad luck is someone else's good luck. At least some thread, somewhere, has succeeded in pushing to the stack, so the system is making progress (stuff is happening).

And here is a small wait-free algorithm:

```
void increment_reference_counter(rc_base* obj) {
    atomic_increment(obj->rc);
}

void decrement_reference_counter(rc_base* obj) {
    if (0 == atomic_decrement(obj->rc))
        delete obj;
}
```

Both operations will complete in a bounded number of steps and therefore there is no possibility that anything gets stuck or is forced to repeat itself forever.

The big question is: are lock-free programming techniques somehow better for performance? Well, they can be but they might not be either. Lock free algorithms are about ensuring there is forward progress in the system and not really specifically about speed. A particular algorithm implementation might be faster under lock-free algorithms. For example, if the compare and swap operation to replace a list head is faster than the mutex lock and unlock, you prefer the lock free algorithm. But often they are not. In fact, the lock free algorithms could be slower, in which case you use them because you must, not because it is particularly speedy.