ECE459: Programming for Performance Winter 2021

Lecture 12 — Lock Convoys, Atomics, Lock-Freedom

Patrick Lam and Jeff Zarnett *2022-02-16*

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.

Unfair Locks. 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, which is fair, is a good explanation of why can go wrong. In XP, if A unlocks a lock ℓ, and there is a thread

B waiting, then B gets the lock, it is no longer blocked, and when it wakes up, B already owns the lock. This is fair in the sense that there was no period during which the lock was available; therefore it could not be "stolen" by some other thread that happened to come along at the right (or perhaps wrong) time [**?**]. (Specifically, if the OS chooses who gets the lock among all the waiting threads randomly, then that's fair.)

Fairness is good, right? But 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. That whole time, it is holding onto the lock. No one else can acquire ℓ. Worse yet, a thread C might start processing, request ℓ, and then we have to context switch again. That is a lock convoy.

Unfair locks help with lock convoys by not giving the lock to B when A releases the lock. Instead, the lock is simply unowned. The scheduler chooses another thread to switch to after A. If it's B, then it gets the lock and continues. If it's instead some thread C which didn't want the lock initially, then C gets to run. If it doesn't request ℓ, then it just computes as normal. If C does request ℓ, it gets it. Maybe it'll release it before B gets its turn, thus enabling more throughput than the fair lock.

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.

Mitigating Lock Convoys Ourselves. 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, and 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. It requires a concept of yielding, of course, and it is fairly straightforward. The `yield_now` function just tells the OS scheduler that we are not able to do anything useful right now and we'd prefer to let another thread run instead. The Rust documentation points out that channels do this in the implementation of sending and receiving to and from channels. But, see the code below for a quick example.

```
let mut retries = 0;
let retries_limit = 10;
let counter = Mutex::new(0);

loop {
    if retries < retries_limit {
```

```
            let mut l = counter.try_lock();
            if l.is_ok() {
                *l.unwrap() = 1;
                break;
            } else {
                retries = retries + 1;
                thread::yield_now();
            }
        } else {
            *counter.lock().unwrap() = 1;
            break;
        }
    }
}
```

In short, we try to lock the mutex some number of times (up to a maximum of `retries_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.

You may have learned about condition variables earlier. Rust has them as well, the `std::sync::Condvar` type.

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

The general recommendation of the internet is to use `notify_all` in all situations. Counting on each thread to aways unconditionally wake up the next when it runs is sightly dangerous...

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

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

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. That means when you are asked about ordering in a method on an atomic type, it means `Ordering::SeqCst`. Later in the course we will revisit the idea of memory consistency and the different possible reorderings, but for now, just use sequential consistency and you won't get surprises.

So there are atomic types for integer types (signed and unsigned), boolean, size (signed and unsigned), and pointers. It's important to note that when interacting with the type, you cannot just assign or read the value; you're forced to use the `load` and `store` methods to be sure there's no confusion. Such types are safe to be passed between threads as well as being shared between them.

```
use std::sync::atomic::{AtomicBool, Ordering};

fn main() {
   let b = AtomicBool::new(false);
    b.store(true, Ordering::SeqCst);
    println!("{}", b.load(Ordering::SeqCst));
}
```

In addition, there are a few other methods to allow you to atomically complete the operations you normally need. For example, `fetch_add` is what you would use to atomically increase the variable's value. In C, `count++` is not atomic; in Rust we would use `count.fetch_add(1, Ordering::SeqCst)`.

The other atomic operations that we can breeze past are `fetch_sub` (fetch and subtract), `fetch_max` (fetch and return the max of the stored value and the provided argument), `fetch_min` (same as max but minimum), and the bitwise operations `and, nand, or, xor`.

Compare and Swap. This operation is also called **compare and exchange** (implemented by the `cmpxchg` instruction on x86). This is one of the more important atomic operations, because it combines the read, comparison, and write into a single operation. You'll see **cmpxchg** quite frequently in the Linux kernel code.

Here's a description of how a compare-and-swap operation works using C. This is obviously not how it is implemented, but explaining it using program code is more precise (and compact) than a lengthy English-language explanation. It is really implemented as an atomic hardware instruction and this all takes place uninterruptibly.

```
int compare_and_swap (int* reg, int oldval, int newval) {
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  return old_reg_val;
}
```

Afterwards, you can check if the CAS returned `oldval`. If it did, you know you changed it. If not, you should try again (maybe with some delay). If multiple threads are trying to do the compare-and-swap operation at the same time, only one will succeed.

The Rust equivalent for this is called `compare_and_swap` and it takes as parameters the expected old value, the desired new value, and the ordering. We'll see an example in just a moment. Rust does offer a simple `swap` on atomic types that doesn't do the comparison and just returns the old value, as well as two more advanced versions called `compare_exchange` and `compare_exchange_weak` that we won't talk about today.

Implementing a Spinlock. You can use compare-and-swap to implement a spinlock. Remember that a spinlock is constantly trying to acquire the lock (here, represented by an atomic boolean) and only makes sense if the expected waiting time to acquire the lock is less than the time it would take for two thread switches.

```
use std::sync::atomic::{AtomicBool, Ordering, spin_loop_hint};
```

```
fn main() {
    let my_lock = AtomicBool::new(false);
    // ... Other stuff happens

    while my_lock.compare_and_swap(false, true, Ordering::SeqCst) == true {
        // The lock was 'true', someone else has the lock, so try again
        spin_loop_hint();
    }
    // Inside critical section
    my_lock.store(false, Ordering::SeqCst);
}
```

The call inside the loop to `spin_loop_hint` is just a nicety we can use to tell the CPU that it's okay to either switch to another thread in hyperthreading or to run in a lower-power mode if we are spinning[2]. Full CPU effort isn't needed for this, and it's nice if we can let the CPU know that.

ABA Problem Sometimes you'll read a location twice. If the value is the same both times, nothing has changed, right? *No.* This is an **ABA problem**.

The ABA problem is not any sort of acronym nor a reference to this [**?**]. It's a value that is A, then changed to B, then changed back to A. The ABA problem is a big mess for the designer of lock-free Compare-And-Swap routines. This sequence will give some example of how this might happen [**?**]:

1. P_1 reads A_i from location L_i.
2. P_k interrupts P_1; P_k stores the value B into L_i.
3. P_j stores the value A_i into L_i.
4. P_1 resumes; it executes a false positive CAS.

It's a "false positive" because P_1's compare-and-swap operation succeeds even though the value at L_i has been modified in the meantime. If this doesn't seem like a bad thing, consider this. If you have a data structure that will be accessed by multiple threads, you might be controlling access to it by the compare-and-swap routine. What should happen is the algorithm should keep trying until the data structure in question has not been modified by any other thread in the meantime. But with a false positive we get the impression that things didn't change, even though they really did.

You can combat this by "tagging": modify value with a nonce upon each write. You can also keep the value separately from the nonce; double compare and swap atomically swaps both value and nonce. Java collections do something resembling this. A collection has a modification count and every time the collection is modified in some way (element added, for example) the counter is increased. When an iterator is created to iterate over this collection, the iterator notes down the current value of the modification count. As it iterates over the collection, if the iterator sees that the collection's modification count is no longer the same as the value it has remembered, it will throw a `ConcurrentModificationException`.

Caveats. Obviously, the use of atomic types just ensures that a write or read (or read-modify-write operation) happens atomically; race conditions can still happen if threads are not properly coordinated.

Unfortunately, not every atomic operation is portable. Rust will try its best to give you the atomic types that you ask for. Sometimes emulation is required to make it happen, and an atomic type might be implemented with a larger type (e.g.,, `AtomicI8` will be implemented using a 4-byte type). Some platforms don't have it at all. So code that is focused on portability might have to be a bit careful.

[2] https://doc.rust-lang.org/std/sync/atomic/fn.spin_loop_hint.html

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. Since we just talked about compare-and-swap, 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 example from [**?**], with some modifications. We'll start with a lock-free stack.

```
use std::ptr::{self, null_mut};
use std::sync::atomic::{AtomicPtr, Ordering};

pub struct Stack<T> {
    head: AtomicPtr<Node<T>>,
}

struct Node<T> {
    data: T,
    next: *mut Node<T>,
}

impl<T> Stack<T> {
```

```
    pub fn new() -> Stack<T> {
        Stack {
            head: AtomicPtr::new(null_mut()),
        }
    }
}

impl<T> Stack<T> {
    pub fn push(&self, t: T) {
        // allocate the node, and immediately turn it into a *mut pointer
        let n = Box::into_raw(Box::new(Node {
            data: t,
            next: null_mut(),
        }));
        loop {
            // snapshot current head
            let head = self.head.load(Ordering::SeqCst);

            // update 'next' pointer with snapshot
            unsafe { (*n).next = head; }

            // if snapshot is still good, link in new node
            if self.head.compare_and_swap(head, n, Ordering::SeqCst) == head {
                break
            }
        }
    }
}
```

A particularly unlucky thread might spend literally forever spinning around the 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:

```
fn increment_counter(ctr: &AtomicI32) {
    ctr.fetch_add(1, Ordering::SeqCst);
}

fn decrement_counter(ctr: &AtomicI32) {
    let old = ctr.fetch_sub(1, Ordering::SeqCst);
    if old == 1 { // We just decremented from 1 to 0
        println!{ "All done." }
    }
}
```

Obviously, the print statement in the decrement counter is just a placeholder for something more useful. 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.