ECE459: Programming for Performance Winter 2021

Lecture 11 — Use of Locks, Reentrancy

Jeff Zarnett *2020-11-13*

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 in a way that reduces their impact on performance.

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 not contain any extraneous statements. The ones that don't need to be there are those that don't operate on shared data.

If you are rewriting code from sequential to parallel, 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, so the rule is not absolute. However, such statements should be there rarely and only if the alternatives are worse.

Let's consider a short code example from the producer-consumer problem. In the course repository's code directory, the full code is available the original and modified forms. It makes sense to look over the original before we discuss how to improve it. We'll look at the consumer code:

```
for _j in 0 .. NUM_THREADS {
    // create consumers
    let spaces = spaces.clone();
    let items = items.clone();
    let buffer = buffer.clone();
    threads.push(
        thread::spawn(move || {
            for _k in 0 .. ITEMS_PER_THREAD {
                let permit = block_on(items.acquire());
                let mut buf = buffer.lock().unwrap();
                let current_consume_space = buf.consumer_count;
                let next_consume_space = (current_consume_space + 1) % buf.buffer.len();
                let to_consume = *buf.buffer.get(current_consume_space).unwrap();
                buf.consumer_count = next_consume_space;
                spaces.add_permits(1);
                permit.forget();
                consume_item(to_consume);
            }
        })
    );
}
```

When we used locks in C (or similar), it was easier to identify what's in the critical section, because we had explicit lock and unlock statements. The explicit unlock statement, especially, made it much clearer where it ends. Now, we don't consider the critical section over until the `MutexGuard` (returned by `lock()`) goes out of scope. And that happens here at the end of the iteration of the loop.

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

In a practical sense, the critical section needs to enclose anything that references `buf` and that's most of the statements, save those three at the end: adding permits to spaces, forgetting our current permit, and consuming the item. Rust is good about not letting you access shared data in an uncontrolled way, so we can feel more certain that there's nothing left out of the critical section that should be in there.

How do we end the critical section? We need to make our acquisition of the mutex go out of scope. The easiest way to do that is to use manual scoping:

```
for _j in 0 .. NUM_THREADS {
    // create consumers
    let spaces = spaces.clone();
    let items = items.clone();
    let buffer = buffer.clone();
    threads.push(
        thread::spawn(move || {
            for _k in 0 .. ITEMS_PER_THREAD {
                let permit = block_on(items.acquire());
                let to_consume = {
                    let mut buf = buffer.lock().unwrap();
                    let current_consume_space = buf.consumer_count;
                    let next_consume_space = (current_consume_space + 1) % buf.buffer.len();
                    let to_consume = *buf.buffer.get(current_consume_space).unwrap();
                    buf.consumer_count = next_consume_space;
                    to_consume
                };
                spaces.add_permits(1);
                permit.forget();
                consume_item(to_consume);
            }
        })
    );
}
```

You'll notice that we return the value `to_consume` out of the block, because it's needed outside the block and would otherwise not be in scope when passed to the function that consumes it. The whole purpose of this is to get the value outside of the block. Because it's a simple type, we'll copy it, but a more complex type would just have ownership transferred, so there isn't a large performance penalty here.

The other approach to making the `MutexGuard` go out of scope is to actually call `drop()` on it, which is effective in telling the compiler that it is time for this value to die. Calling `drop()` moves ownership of the MutexGuard to the `drop` function where it will go out of scope and be removed. Convenient! But manual scoping is nice too.

Let's see if it works! I applied a similar change the producer code as we just discussed about the consumer. And for the purposes of the test, I added some thread sleeps to the original and modified program so it appears that consuming or producing an item actually takes meaningful work. As usual, benchmarks are created with `hyperfine -warmup 1 -m 5 "cargo run -release"`. The un-optimized program takes about 2.8 seconds to run and the optimized program takes about 1.1 seconds. Certainly worth doing.

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.

Locking Granularity

The producer-consumer example was a very specific instance of *lock granularity*: how much data is locked by a given lock. We have choices about the granularity of locking, and it is a trade-off (like always).

Coarse-grained locking is easier to write and harder to mess up, but it can significantly reduce opportunities for parallelism. *Fine-grained locking* requires more careful design, increases locking overhead and is more prone to bugs (deadlock etc). Locks' extents constitute their *granularity*. In coarse-grained locking, you lock large sections of your program with a big lock; in fine-grained locking, you divide the locks and protect smaller sections with multiple smaller locks.

We'll discuss three major concerns when using locks:

- overhead;
- contention; and
- deadlocks.

We aren't even talking about under-locking (i.e., remaining race conditions). We'll assume there are adequate locks and that data accesses are protected.

Lock Overhead. Using a lock isn't free. You pay:

- allocated memory for the locks;
- initialization and destruction time; and
- acquisition and release time.

These costs scale with the number of locks that you have.

Lock Contention. Most locking time is wasted waiting for the lock to become available. We can fix this by:

- making the locking regions smaller (more granular); or
- making more locks for independent sections.

Deadlocks. Finally, the more locks you have, the more you have to worry about deadlocks.

As you know, the key condition for a deadlock is waiting for a lock held by process X while holding a lock held by process X'. ($X = X'$ is allowed).

Okay, in a formal sense, the four conditions for deadlock are:

1. **Mutual Exclusion**: A resource belongs to, at most, one process at a time.
2. **Hold-and-Wait**: A process that is currently holding some resources may request additional resources and may be forced to wait for them.
3. **No Preemption**: A resource cannot be "taken" from the process that holds it; only the process currently holding that resource may release it.
4. **Circular-Wait**: A cycle in the resource allocation graph.

Consider, for instance, two processors trying to get two locks.

Thread 1	**Thread 2**
`Get Lock 1`	`Get Lock 2`
`Get Lock 2`	`Get Lock 1`
`Release Lock 2`	`Release Lock 1`
`Release Lock 1`	`Release Lock 2`

Processor 1 gets Lock 1, then Processor 2 gets Lock 2. Oops! They both wait for each other. (Deadlock!).

To avoid deadlocks, always be careful if your code **acquires a lock while holding one**. You have two choices: (1) ensure consistent ordering in acquiring locks; or (2) use trylock.

As an example of consistent ordering:

```
let mut thing1 = l1.lock().unwrap()           let mut thing1 = l1.lock().unwrap()
let mut thing2 = l2.lock().unwrap()           let mut thing2 = l2.lock().unwrap()
// protected code                             // protected code
// locks dropped when going out of scope      // locks dropped when going out of scope
```

This code will not deadlock: you can only get **l2** if you have **l1**. Of course, it's harder to ensure a consistent deadlock when lock identity is not statically visible. That is, if they don't always have the same names everywhere.

If we give a standard example from a textbook, we call the threads P and Q and they are attempting to acquire `a` and `b`. Thread Q requests `b` first and then `a`, while P does the reverse. The deadlock would not take place if both threads requested these two resources in the same order, whether `a` then `b` or `b` then `a`. Of course, when they have names like this, a natural ordering (alphabetical, or perhaps reverse alphabetical) is obvious.

We can certainly prove that consistent ordering does work and it's a proof by contradiction. If the set of all resources in the system is $R = \{R_0, R_1, R_2, ...R_m\}$, we assign to each resource R_k a unique integer value. Let us define this function as $f(R_i)$, that maps a resource to an integer value. This integer value is used to compare two resources: if a process has been assigned resource R_i, that process may request R_j only if $f(R_j) > f(R_i)$. Note that this is a strictly greater-than relationship; if the process needs more than one of R_i then the request for all of these must be made at once (in a single request). To get R_i when already in possession of a resource R_j where $f(R_j) > f(R_i)$, the process must release any resources R_k where $f(R_k) \geq f(R_i)$. If these two protocols are followed, then a circular-wait condition cannot hold [SGG13].

But I mentioned they might not have the same names everywhere. When locks travel with the data, this problem can arise. Consider the idea of a bank account and you want to transfer money from one to another. This will involve locking the sender account and locking the receiver account. And regardless of whether you say receiver first or sender first, there is the possibility of two concurrent transfers that mean we end up with a deadlock.

One thing that might work is making your structure somewhat different. Something like the account number is something that never changes, so you could leave that outside of the mutex and use that to determine an ordering, such as alphabetical ordering. That just means your struct is composed of the account number and the mutex surrounding another struct with the account data. Maybe a little weird, but it works.

Alternately, you can use trylock. Recall that Pthreads' `trylock` returns 0 if it gets the lock. But if it doesn't, your thread doesn't get blocked. Checking the return value is important, but at the very least, this code also won't deadlock: it will give up **l1** if it can't get **l2**.

```
loop {
    let mut m1 = l1.lock().unwrap();
    let m2 = l2.try_lock();
    if m2.is_ok() {
        *m1 += amount;
        *m2.unwrap() -= amount;
        break;
    } else {
        println!("try_lock␣failed");
        // Go around the loop again and try again
    }
}
```

(Incidentaly, using trylocks can also help you measure lock contention.)

This prevents the hold and wait condition, which was one of the four conditions. A process attempts to lock a group of resources at once, and if it does not get everything it needs, it releases the locks it received and tries again. Thus a process does not wait while holding resources.

Coarse-Grained Locking

One way of avoiding problems due to locking is to use few locks (perhaps just one!). This is *coarse-grained locking*. It does have a couple of advantages:

- it is easier to implement;
- with one lock, there is no chance of deadlocking; and
- it has the lowest memory usage and setup time possible.

It also, however, has one big disadvantage in terms of programming for performance: your parallel program will quickly become sequential.

Example: Python (and other interpreters). Python puts a lock around the whole interpreter (known as the *global interpreter lock*). This is the main reason (most) scripting languages have poor parallel performance; Python's just an example.

Two major implications:

- The only performance benefit you'll see from threading is if one of the threads is waiting for I/O.
- But: any non-I/O-bound threaded program will be **slower** than the sequential version (plus, the interpreter will slow down your system).

You might think "this is stupid, who would ever do this?" Yet a lot of OS kernels do in fact have (or at least had) a "big kernel lock", including Linux and the Mach Microkernel. This lasted in Linux for quite a while, from the advent of SMP support up until sometime in 2011. As much as this ruins performance, correctness is more important. We don't have a class "programming for correctness" (software testing? Hah!) because correctness is kind of assumed. What we want to do here is speed up our program as much as we can while maintaining correctness...

Fine-Grained Locking

On the other end of the spectrum is *fine-grained locking*. The big advantage: it maximizes parallelization in your program.

However, it also comes with a number of disadvantages:

- if your program isn't very parallel, it'll be mostly wasted memory and setup time;
- plus, you're now prone to deadlocks; and
- fine-grained locking is generally more error-prone (be sure you grab the right lock!)

Examples. Databases may lock fields / records / tables. (fine-grained → coarse-grained).

You can also lock individual objects (but beware: sometimes you need transactional guarantees.)

Reentrancy

Recall from a bit earlier the idea of a side effect of a function call.

The trivial example of a non-reentrant C function:

```
int tmp;

void swap( int x, int y ) {
    tmp = y;
    y = x;
    x = tmp;
}
```

Why is this non-reentrant? Because there is a global variable `tmp` and it is changed on every invocation of the function. We can make the code reentrant by moving the declaration of `tmp` inside the function, which would mean that every invocation is independent of every other. And thus it would be thread safe, too.

Doing it wrong is highly discouraged by Rust as a language, because it doesn't want you to use global state (if it can help it) and it makes potential side effects pretty clear by requiring references to be annotated as mutable.

Remember that in things like interrupt subroutines (ISRs) having the code be reentrant is very important. Interrupts can get interrupted by higher priority interrupts and when that happens the ISR may simply be restarted (or we're going to break off handling what we're doing and call the same ISR in the middle of the current one). Either way, if the code is not reentrant we will run into problems. Rust's ownership capabilities make it difficult for you to modify something that you should not modify with signal handlers, like calling some function that is not reentrant.

Side effects are and sort of undesirable, but not necessarily bad. Printing to console is unavoidably making use of a side effect, but it's what we want. We don't want to call print reentrantly; interleaved print calls would result in jumbled output. Or alternatively, restarting the print routine might result in some doubled characters on the screen.

The notion of purity is related to side effects. A function is pure if it has no side effects and if its outputs depend solely on its inputs. (The use of the word pure shouldn't imply any sort of moral judgement on the code). Pure functions should also be implemented to be thread-safe and reentrant.

Functional Programming and Parallelization

Interestingly, functional programming languages (by which I do NOT mean procedural programming languages like C) such as Haskell and Scala and so on, lend themselves very nicely to being parallelized. Why? Because a purely functional program has no side effects and they are very easy to parallelize. If a function is impure, its functions signature will indicate so. Thus spake Joel[1]:

> *Without understanding functional programming, you can't invent MapReduce, the algorithm that makes Google so massively scalable. The terms Map and Reduce come from Lisp and functional programming. MapReduce is, in retrospect, obvious to anyone who remembers from their 6.001-equivalent programming class that purely functional programs have no side effects and are thus trivially parallelizable.* [Spo05]

This assumes of course that there is no data dependency between functions. Obviously, if we need a computation result, then we have to wait. But the key is to write your code like mathematical functions: $f(x, y, z) \to (a, b, c)$

Object oriented programming kind of gives us some bad habits in this regard: we tend to make a lot of `void` methods or those with no return type. In functional programming these don't really make sense, because if it's purely functional, then there are some inputs and some outputs. If a function returns nothing, what does it do?

[1] "Thus Spake Zarathustra" is a book by Nietzsche, and this was not a spelling error.

For the most part it can only have side effects which we would generally prefer to avoid if we can, if the goal is to parallelize things.

Rust does encourage the some things that help point you towards functional-style programming. For one thing, it discourages mutability of data, which points you more towards making the arguments to functions be either immutable references (and therefore not changed by the function they are passed to) or ownership transfer (meaning no concurrency). Internal mutability is a thing, but is somewhat discouraged.

References

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.

[Spo05] Joel Spolsky. The perils of JavaSchools, 2005. Online; accessed 8-December-2015. URL: `http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html`.