ECE459: Programming for Performance Winter 2021

Lecture 15 — Memory Consistency

Patrick Lam and Jeff Zarnett *2020-10-16*

Memory Consistency, Memory Barriers, and Reordering

Previously, when atomics were introduced, we said to use sequential consistency without much detail and without discussing the other options. Now it's time to learn about it. We'll cover both instruction reordering by the CPU and reordering initiated by the compiler.

Compiler Reordering. When asked to compile code, the compiler does not take every statement that you provide and translate it into a (set of) machine language instruction(s). The compiler can change the order of certain events. The compiler will be aware of things like load-delay slots and can swap the order of instructions to make use of those slots more effectively. In the (silly) example on the left there might be a stall while we wait for `x` to be available before we can send it in to the `println!` macro; on the right we moved two unrelated instructions into the delay slots. So that feels like free performance!

```
let x = thing.y;
println!("x = {}", x);
z = z + 1;
a = b + c;
```

```
let x = thing.y;
z = z + 1;
a = b + c;
println!("x = {}", x);
```

We'll talk about other compiler optimizations soon, but we don't want to get away from the topic of reordering.

Hardware Reordering. In addition to the compiler reordering, the hardware can do some reordering of its own. A sequence of instructions is provided to the CPU, and it can decide it would rather do them in an order it finds more convenient. That is fairly straightforward.

There is another possibility we have to consider, and it is updates from other threads. When a thread is doing a check on a variable, such as a quit condition (exit the loop if `quit` is now true), how do we know if we have the most up-to-date value for `quit`? We know from the discussion of cache coherence that the cache will be updated via snooping, but we need a bit more reassurance that the value we're seeing is the latest one. How could we get the wrong order? If the read by thread A is reordered by the hardware so that it's after the write by thread B, then we'll see the "wrong" answer.

Different hardware provides different guarantees about what reorderings it won't do. Old 386 CPUs didn't do any; x86 usually won't (except where there are some specific violations of that; but ARM has weak ordering except where there are data dependencies [Pre12]. ARM is getting pretty popular, so we do have to care about hardware reorderings, unfortunately.

I have a plan, but it's a bad one. There are some reorderings where we are easily able to conclude that it is okay and safe to do, but not every reordering is. In an obvious case, if the lines of code are `z *= 2` and `z += 1` then neither the compiler nor hardware will reorder those because it knows that it would change the outcome and produce the wrong answer. There's a clear data dependency there, so the reordering won't happen. There are a couple of hardware architectures where that isn't respected, but we'll ignore them for now.

But what if there's no such clear dependency? Consider something like this pseudocode:

```
lock mutex for point
point.x = 42;
point.y = -42;
point.z = 0;
unlock mutex for point
```

```
lock mutex for point
point.x = 42;
point.y = -42;
unlock mutex for point
point.z = 0;
```

Wait a minute — that's not an okay reordering, because now an element of the point is being accessed outside of the critical section and we don't want that. It's a reordering, alright, in that the store of `point.z` has been moved to after the store of state of the mutex (unlock does, after all, change its state). What we need is a way to tell the compiler (and hardware) that this is not okay.

Sequential Consistency. In a sequential program, you expect things to happen in the order that you wrote them. So, consider this code, where variables are initialized to 0:

```
T1: x = 1; r1 = y;
T2: y = 1; r2 = x;
```

We would expect that we would always query the memory and get a state where some subset of these partially-ordered statements would have executed. This is the *sequentially consistent* memory model. A simple description: (1) each thread induces an *execution trace*; and (2) always, the program has executed some prefix of each thread's trace. Or, alternatively:

> "... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." — Leslie Lamport

It turns out that sequential consistency is expensive to implement. Think how much coordination is needed to get a few people to agree on where to go for lunch; now try to get a group of people to agree on what order things happened in. Right. Now imagine it's a disagreement between threads so they don't have the ability to negotiate. So most systems actually implement weaker memory models, such that both `r1` and `r2` might end up unchanged.

Allowing some reorderings could potentially significantly speed up the program! If left to its own devices, the compiler could reorder anything, but we need to tell it what is allowed and what is disallowed.

Memory Consistency Models

Rust uses the same memory consistency models as C++. The Rustonomicon (book of names of Rust[1]) says pretty directly that this is not because the model is easy to understand, but because it's the best attempt we have at modelling atomics because it is a very difficult subject. The idea behind the memory model is to have a good way of talking about the *causality* of the program. While causality definitely sounds like something Commander Data would talk about on the *Enterprise*, in this case it means establishing relationships between events such as "event A happens before event B".

You will recall from the introduction to the subject of concurrency that we frequently sought the same thing in our program at a higher level, when we'd say that we can use a semaphore to ensure that one thing happens before another. The idea is the same, but our toolkit is a little bit different: it's the *memory barrier* or *fence*.

This type of barrier prevents reordering, or, equivalently, ensures that memory operations become visible in the right order. A memory barrier ensures that no access occurring after the barrier becomes visible to the system, or takes effect, until after all accesses before the barrier become visible.

[1] Not to be confused with the Necronomicon...

The x86 architecture defines the following types of memory barriers:

- `mfence`. All loads and stores before the barrier become visible before any loads and stores after the barrier become visible.
- `sfence`. All stores before the barrier become visible before all stores after the barrier become visible.
- `lfence`. All loads before the barrier become visible before all loads after the barrier become visible.

Note, however, that while an `sfence` makes the stores visible, another CPU will have to execute an `lfence` or `mfence` to read the stores in the right order.

Consider the example again:

```
                        f = 0

/* thread 1 */                      /* thread 2 */
while (f == 0) /* spin */;          x = 42;
// memory fence                     // memory fence
printf("%d", x);                    f = 1;
```

This now prevents reordering, and we get the expected result.

Memory fences are costly in performance. It makes sense when we think about it, since it (1) prevents re-orderings that would otherwise speed up the program; and (2) can force a thread to wait for another one. Sequential consistency will necessarily result in memory fences being generated to produce the correct results.

Other Orderings

The C++ standard includes a few other orderings that don't appear in this section because they aren't in Rust. But we'll cover Acquire-Release and Relaxed briefly. Neither comes with a recommendation to use it, but if you can prove that your use of it is correct, then you can do it. It may give a slight performance edge

Acquire means that accesses (reads or writes) after the acquire operation can't move to be before the acquire. Release means accesses before the release operation can't move to be after the release. They make a good team: by placing acquire at the start of a section and release after, anything in there is "trapped" and can't get out.

That makes them the perfect combination for a critical section: acquire prevents things from moving from inside the critical section to before the critical section; release prevents things from inside from moving to after the critical section. Nice!

Here's an example of acquire and release, as taken from the Rustonomicon's page about atomics (`https://doc.rust-lang.org/nomicon/atomics.html`. It's implementing a spinlock:

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

fn main() {
    let lock = Arc::new(AtomicBool::new(false)); // value answers "am I locked?"

    // ... distribute lock to threads somehow ...

    // Try to acquire the lock by setting it to true
    while lock.compare_and_swap(false, true, Ordering::Acquire) { }
    // broke out of the loop, so we successfully acquired the lock!

    // ... scary data accesses ...

    // ok we're done, release the lock
```

```
    lock.store(false, Ordering::Release);
}
```

The acquire and release semantics keep all the things that should be in the critical section inside it.

And then there is relaxed. Relaxed really does mean the compiler will take it easy, and all reorderings are possible. Even ones that you might not want! The Rustonomicon suggests one possible valid use for that scenario is a counter that simply adds and you aren't using the counter to synchronize any action. Something like atomically counting the requests to each resource might be suitable. You can report the counters as metrics and it's not super important that request 9591's increment of the counter occurs before that of request 9598. It's all the same in the end...

Matters, Order Does

There have been a few reminders to use sequential consistency because atomics are hard to reason about and it's easy to get it wrong. But does this happen in reality? Yes, and here's an example of it in Rust [O'C18].

The observed behaviour was an inconsistent state being reported by an assertion; when looking at the registers the registers contained garbage even though it was just after a read that should have loaded it in. That's hard to notice and difficult to debug as well, because running in debug mode might prevent the reordering in the first place

You can actually look at the fix applied to the lock-free queue at `https://github.com/crossbeam-rs/crossbeam/pull/98/files`. But the short summary is that the load of the `ready` property needs to have at least Acquire semantics and the store of it should have release. If we don't do that, we might attempt to park the thread early

References

[O'C18] Robert O'Callahan. Diagnosing a weak memory ordering bug, 2018. Online; accessed 2020-10-09. URL: `https://robert.ocallahan.org/2018/08/for-first-time-in-my-life-i-tracked.html`.

[Pre12] Jeff Preshing. Weak vs. strong memory models, 2012. Online; accessed 2020-10-09. URL: `https://preshing.com/20120930/weak-vs-strong-memory-models/`.