# Lecture 15 — Memory Consistency

Patrick Lam & Jeff Zarnett
patrick.lam@uwaterloo.ca, jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

November 23, 2020

When we introduced atomics, we said to always use sequential consistency.



"Stay out of trouble." - Robocop, 2043.

There are other options, but we need to discuss reordering.

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.

```
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.

In addition to the compiler reordering, the hardware can do some reordering of its own.

There is another possibility we have to consider, and it is updates from other threads.

We need a bit more reassurance that the value we're seeing is the latest one...

Different hardware provides different guarantees about what reorderings it won't do.

ARM is getting pretty popular, so we do have to care about hardware reorderings, unfortunately.

In an obvious case, if the lines of code are   z  *=  2   and z  +=  1 then neither the compiler nor hardware will reorder those.

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.

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

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

What we need is a way to tell the compiler (and hardware) that this is not okay.

Sequential program: statements execute in order.

Your expectation for concurrency: sequential consistency.

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

- each thread induces an *execution trace*.
- always: program has executed some prefix of each thread's trace.

*"... 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*

In brief:

1. for each thread: in-order execution;
2. interleave the threads' executions.

But unfortunately, threads have their own view of the world.

Compilers and processors may reorder non-interfering memory operations.

$$T1 : x = 1; r1 = y;$$

If two statements are independent:

- OK to execute them in either order.
- (equivalently: publish their results to other threads).

Reordering is a major compiler tactic to produce speedup.

Rust uses the same memory consistency models as C++.

It's the best attempt we have at modelling atomics because it is a very difficult subject

We need way of talking about the causality of the program...
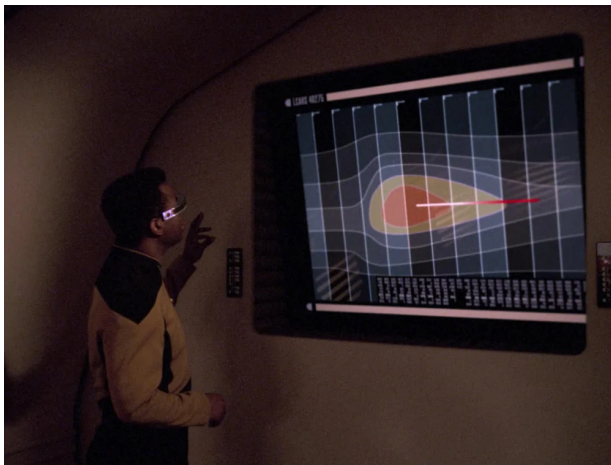
Establishing relationships between events such as "event A happens before event B".

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.

The x86 architecture defines the following types of memory barriers:

- `mfence`
- `sfence`
- `lfence`

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.

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 and Release 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.

```rust
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);
}
```

Relaxed really does mean the compiler will take it easy, and all reorderings are possible.

A counter that simply adds and you aren't using the counter to synchronize any action.

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.

Consider an inconsistent state in a lock-free-queue structure...

We can actually look at the fix applied in the code.