ECE459: Programming for Performance Winter 2018

Lecture 8 — Atomics, Lock-Free Datastructures

Patrick Lam and Jeff Zarnett

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):

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 really

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

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-Free Data Structures