What every systems programmer should know about lockless concurrency

Matt Kline

November 1, 2017

Abstract

Seasoned programmers are familiar with concurrency building blocks like mutexes, semaphores, and condition variables. But what makes them work? How do we write concurrent code when we can't use them, like when we're working below the operating system in an embedded environment, or when we can't block due to hard time constraints? And since your system transforms your code into things you didn't write, running in orders you never asked for, how do multithreaded programs work at all? Concurrency—especially on modern hardware—is a complicated and unintuitive topic, but let's try to cover some fundamentals.

Contents

1. *Background*

Modern computers run multiple sequences of instructions concurrently. We call them different names depending on the context—processes, threads, tasks, interrupt service routines, and so on—but many of the same principles apply across the board. On single-core machines, these sequences take turns, sharing the CPU. On multiprocessors, several can run in parallel, each on its own core.

While computer scientists have invented many useful abstractions, these instruction streams (let's call them *threads* from here on out for the sake of brevity) ultimately interact with one another by sharing bits of state. For this to work, we must be able to reason about the order of reads and writes that communicating threads make to memory. Consider a simple example where some thread A shares an integer with other threads. It writes the value to some variable, then sets a flag to instruct others to read whatever it just stored. As code, this might resemble:

```
int v;
bool v_ready = false;

void threadA()
{
  // Write the value
  // and set its ready flag.
  v = 42;
  v_ready = true;
}


void threadB()
{
  // Await a value change and read it.
  while (!v_ready) { /* spin */ }
  const int my_v = v;
  // Do something with my_v...
}
```

Our system must guarantee that other threads observe *A's* write to `v_ready` only *after A's* write to `v`. (If another thread can "see" `v_ready` change before it sees `v` change, our communication scheme can't work.)

This appears to be an incredibly simple guarantee to provide, but nothing is as it seems. For starters, any compiler worth its salt will happily modify and reorder your code to take better advantage of the hardware it runs on. So long as the resulting instructions run to the same effect *for the current thread,* reads and writes can be moved to avoid pipeline stalls* or to improve locality.† Variables can be assigned to the same memory location if they're never used in overlapping time frames. Instructions can be executed speculatively, before a branch is taken, then undone if the compiler guessed incorrectly.‡

Even if we used a compiler that didn't reorder our code, we'd still in trouble, since our hardware does it too! Modern CPU designs handle incoming instructions in a *much* more complicated fashion than traditional pipelined approaches like the one shown in Figure 1. They contain multiple data paths, each for different types of instructions, and schedulers which reorder and route instructions through these paths.

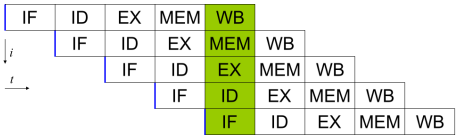

Figure 1: A traditional five-stage CPU pipeline with fetch, decode, execute, memory access, and write-back stages. Modern designs are much more complicated, often reordering instructions on the fly. Image courtesy of Wikipedia.

It's also easy to make naïve assumptions about how memory works. If we imagine a multiprocessor, we might think of something resembling Figure 2, where each core takes turns performing reads and writes to the system's memory.

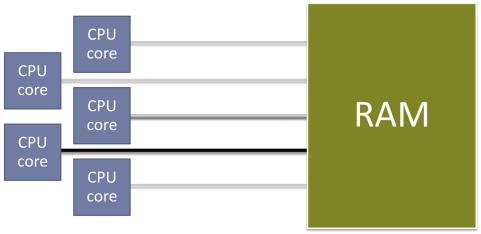

Figure 2: An idealized multiprocessor where cores take sequential turns accessing a single shared set of memory.

This is almost never the case. While processor speeds have increased exponentially in the past decades, RAM hasn't been able to keep up, creating an ever-widening gulf between the time it takes to run an instruction and the time needed to retrieve data from main memory. Hardware manufacturers have compensated by placing an increasing number of hierarchical caches directly on the CPU die. Each core also usually has a *store buffer* that handles pending writes while subsequent instructions are executed. Keeping this memory system *coherent*, so that writes made by one core are observable by others, even if those cores use different caches, is quite challenging.

*Most CPU designs execute parts of several instructions in parallel to increase their clock speed (see Figure 1). When the result of an instruction is needed by another instruction in the pipeline, the CPU may need to suspend forward progress, or *stall,* until that result is ready.

†RAM is not read in single bytes, but in chunks called cache lines. If variables that are used together can be placed on the same cache line, several can be read or written at once. (We'll discuss caches and memory hierarchies shortly.)

‡This is especially common when using profile-guided optimization.

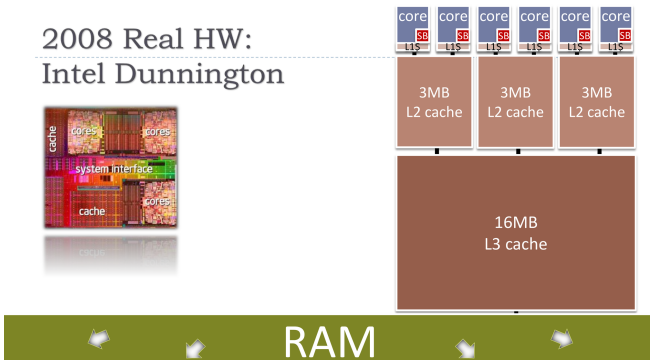

Figure 3: A common memory hierarchy for modern multiprocessors

The net effect of these complications is that there is no consistent concept of "now" in a multithreaded program, especially one running on a multiprocessor. Attaining some sense of order so that threads can communicate is a team effort of hardware manufacturers, compiler writers, language designers, and application developers. Let's explore what we can do, and what tools we will need.

2. *Enforcing law and order*

Creating order in our programs requires a different approach on each CPU architecture. Until alarmingly recently, systems languages like C and C++ offered no help here, so developers needed assembly to write lockless code.* Thankfully, the 2011 ISO standards of both languages introduced tools for inter-thread memory access. So long as the programmer uses them correctly, the compiler will prevent reorderings—both by the optimizer, and by hardware—that cause data races.†

Let's return to our example from before. For it to work as-desired, we need to use an *atomic type* for the "ready" flag:

```
int v = 0;
std::atomic_bool v_ready(false);

void threadA()
{
  v = 42;
  v_ready = true;
}

void threadB()
{
  while (!v_ready) { /* spin */ }
  const int my_v = v;
  // Do something with my_v...
}
```

The C and C++ standard libraries define a series of these types in `<stdatomic.h>` or `<atomic>`, respectively. They look and act just like the integer types they mirror (e.g., `bool` → `atomic_bool`, `int` → `atomic_int`, etc.), but the compiler ensures that other loads and stores aren't reordered around their reads and writes. By using an atomic Boolean, `v = 42` is now guaranteed to happen before `v_ready = true` in thread *A*, just as `my_v = v` must occur after reading `v_ready` in thread *B*.

Formally, these types provide a *single total modification order* such that, "[…] the result of any execution is the same as if the reads and writes occurred in some order, and the operations of each individual processor appear in this sequence in the order specified by its program." This model, defined by Leslie Lamport in 1979, is called *sequential consistency*. Informally, the important takeaway is that sequentially consistent reads and writes act as rendezvous points for threads. By ensuring that other reads and writes cannot move "past" them, we know that anything thread *A* did before writing to an atomic variable—such as assigning 42 to `v` before writing to `v_ready`—can be observed by another thread that reads the atomic variable.

3. *Atomicity*

Our focus so far on ordering sidestepped the other vital element of lockless programming: *atomicity*. Something is atomic if it cannot be divided into smaller parts. To see why lockless reads and writes must have this quality, let's see what problems we might encounter if they did not.

Consider a program with two threads. One thread processes some list of files and increments a counter each time it finishes working on one of them. The other thread handles the user interface, and will periodically read the counter to update a progress bar. If that counter is a 64-bit integer, we have a problem on 32-bit machines, since two loads or stores are needed to read or write the entire value. If we're having a particularly unlucky time, the first thread could be halfway through writing the counter when the second thread reads it, receiving an incorrect value. These unfortunate occasions are called *torn reads and writes*.

If reads and writes to shared data are atomic, however, our problem disappears. We can also see that, compared to the difficulties of establishing order, ensuring atomicity is fairly straightforward: make sure that variables used for thread synchronization are no larger than the architecture's word size.

*Calling this sort of work "lockless" programming is a bit of a misnomer, or at least isn't telling the whole story. While you can use these techniques to write non-blocking thread synchronization, they're the exact same approaches used to build locks and other blocking concurrency primitives.

†The ISO C11 standard lifted its lockless concurrency facilities, almost verbatim, from the C++11 standard. Everything you see here should be identical in both languages, barring some arguably cleaner syntax in C++.

4. *Arbitrarily-sized "atomic" types*

Along with `atomic_int` and friends, C++ provides the template `std::atomic<T>` for declaring arbitrary atomic types. C, lacking a similar language feature but wanting to provide the same functionality, added an `_Atomic` keyword. Running counter to what we just discussed, *any* type can be "atomic", even if it is larger than the target architecture's word size. In these cases, the compiler and the language runtime library automatically surround reads and writes to the variable with locks. For situations where this is unacceptable,* you can add an assertion:

```
std::atomic<Foo> bar;
ASSERT(bar.is_lock_free());
```

Though there a few rare exceptions,† the result of this check is almost always known at compile time. Consequently, the C++17 standard adds `is_always_lock_free`:

```
static_assert(
  std::atomic<Foo>::is_always_lock_free);
```

5. *Atomic read-modify-write operations*

Loads and stores are all well and good, but sometimes we need to read a value, modify it, and write it back in a single atomic step. There are a few common *read-modify-write* (RMW) operations. In C++, they're represented as member functions of `std::atomic<T>`. In C, they're freestanding functions.

5.1. *Exchange*

The simplest atomic RMW operation is an *exchange*: the current value is read and replaced with a new one. To see where this might be useful, let's tweak our example from §3. Instead of displaying the total number of processed files, we might want to show how many were processed each second. To do so, we'll have the UI thread zero the counter each time it is read. Even if these reads and writes are atomic, we could still run into the following race condition:

1. The UI thread reads the counter.
2. Before the UI thread has the chance to zero it, the worker thread increments it again.
3. The UI thread now zeroes the counter, and the previous increment is lost.

If the UI thread exchanges the current value of the counter with zero atomically, the race disappears.

5.2. *Test and set*

Test-and-set works on a Boolean value: we read it, set it to `true`, and provide the value it held beforehand. C and C++ offer a type dedicated to this purpose, called `atomic_flag`. We could use it to build a spinlock:

```
std::atomic_flag af;

void lock()
{
  while (af.test_and_set()) {
    // Spin with backoff
  }
}

void unlock() { af.clear(); }
```

If the previous value is `false`, we are the first to acquire the lock, and the caller can proceed with exclusive access to whatever the lock protects. If the previous value is `true`, someone else has acquired the lock and we must wait until they release it by clearing the flag.

5.3. *Fetch and…*

We can also read a value, perform some basic mathematical operation on it (addition, subtraction, bitwise AND, OR, XOR), and return its previous value. You might have noticed that in our exchange example, the worker thread's additions must also be atomic, or else we could run into a race where:

1. The worker thread loads the current counter value and adds one.
2. Before that thread can store the value back, the UI thread zeroes the counter.
3. The worker now performs its store, as if the counter was never cleared.

5.4. *Compare and swap*

Finally, we have *compare-and-swap* (CAS), sometimes called *compare-and-exchange*. It allows us to conditionally exchange a value *if* its previous value matches some expected one. In C and C++, CAS resembles the following, if it were all executed atomically:

*…which is quite often, since we're often using atomic operations to avoid locks in the first place.

†The language standards permit atomic types to be *sometimes* lock-free. This might be necessary for architectures that don't guarantee atomicity for unaligned reads and writes.

```
template <typename T>
bool atomic<T>::compare_exchange_strong(
    T& expected, T desired)
{
  if (*this == expected) {
    *this = desired;
    return true;
  }
  else {
    expected = *this;
    return false;
  }
}
```

You might be perplexed by the `_strong` suffix. Is there a "weak" CAS? Yes, but hold onto that thought—we'll talk about it in §8.1.

Let's say we have some long-running piece of work that we might want to cancel from a UI thread. We'll give it three states: *idle*, *running*, and *cancelled*, and write a loop that exits when it is cancelled.

```
enum class TaskState : int8_t {
  Idle, Running, Cancelled
};

std::atomic<TaskState> ts;

void taskLoop()
{
  ts = TaskState::Running;
  while (ts == TaskState::Running) {
    // Do good work.
  }
}
```

If we only want to set `ts` to `Cancelled` when it's currently `Running`, but do nothing if it's already `Idle`, we could CAS:

```
bool cancel()
{
  auto expected = TaskState::Running;
  return ts.compare_exchange_strong(
    expected, TaskState::Cancelled);
}
```

6. *Lock-free does not mean "faster"*

Somewhere in this discussion, it's important to point out that lockless algorithms are not somehow "better" or "faster" than lock-based ones. They are just different tools. Well-written locks can be very efficient for sharing data between threads. On the other hand, if you find yourself in an interrupt service routine, or processing audio in real time, locks are not an option. Blocking in these cases would be disastrous, so a lock-free approach is your only hope.

Of course, there are situations where blocking and non-blocking approaches could both work.* If performance is a concern, *profile!* How well a given synchronization method will perform depends on a number of factors, ranging from the number of threads at play to the specifics of your CPU hardware. And as always, consider the tradeoffs you make between complexity and performance. Lockless programming is a perilous activity with a proud tradition of code that is ever so subtly wrong.

7. *Sequential consistency on weakly-ordered hardware*

As mentioned in §2, different hardware architectures provide different ordering guarantees, or *memory models*. For example, x64 is relatively *strongly-ordered*, and can usually be trusted to preserve some system-wide order of loads and stores. Other architectures like ARM are more *weakly-ordered,* so one should make few assumptions that loads and stores are in program order unless the CPU is given special instructions—called *memory barriers*—to not shuffle them around.

It's helpful to look at how atomic operations work in a weakly-ordered system, both to gain a better understanding of what's happening in hardware, and to see why the C and C++ models were designed as they were.† Let's examine ARM, since it's straightforward and widely-used. Consider the simplest atomic operations: loads and stores.

```
atomic_int foo;

int getFoo()
{
  return foo;
}
```

becomes

```
getFoo:
  ldr r3, <&foo>
  dmb
  ldr r0, [r3, #0]
  dmb
  bx lr
```

```
void setFoo(int i)
{
  foo = i;
}
```

becomes

```
setFoo:
  ldr r3, <&foo>
  dmb
  str r0, [r3, #0]
  dmb
  bx lr
```

After loading the address of our atomic variable into some scratch register (`r3`), the compiler sandwiches our load or store between memory barriers (`dmb`), then returns. These

*You may also hear of *wait-free* algorithms—they are a subset of lock-free ones which are guaranteed to complete in some bounded number of steps.

†It's worth noting that the concepts we discuss here aren't oddities specific to C and C++. Newer systems programming languages like D and Rust have converged on similar models.

barriers give us sequential consistency—the first ensures that previous reads and writes cannot be placed after our operation, and the second ensures that subsequent reads and writes cannot be placed before it.

8. *Implementing atomic read-modify-write operations with LL/SC instructions*

Like many other RISC* architectures, ARM lacks dedicated instructions for RMW operations. How do we make them atomic? Normal loads and stores won't do, since the processor can context switch to some other thread between any two instructions. We need something special: *load-link* and *store-conditional* (LL/SC). The two work in tandem: A load-link reads a value from a given address (just like any other load), but also instructs the processor to monitor that address. Store-conditional instructions write the given value *only if* no other stores were made to that address since the corresponding load-link. As an example, let's look at an atomic fetch and add:

On ARM,

```
void incFoo() { ++foo; }
```

compiles to:

```
incFoo:
  ldr r3, <&foo>
  dmb
loop:
  ldrex r2, [r3] // LL foo
  adds r2, r2, #1 // Increment
  strex r1, r2, [r3] // SC
  cmp r1, #0 // Check the SC result.
  bne loop // Loop if the SC failed.
  dmb
  bx lr
```

We LL the current value, add one, and immediately try to store it back with a SC. If that fails, another thread may have written a new value to `foo` since our LL, so we repeat the process. In this way, at least one thread is always making forward progress in atomically modifying `foo`, even if several are attempting to do so at once.†

8.1. *Spurious LL/SC failures*

As you might imagine, keeping track of load-linked addresses on a byte-addressable level can be infeasibly expensive in terms of CPU hardware. To reduce this cost, many processors monitor them at some coarser granularity, such as the cache line. This means that a SC can fail if it is preceded by a write to *any* address in the monitored block, not just the specific one that was load-linked.

This is troublesome when we want to compare and swap, and is the raison d'être for `compare_exchange_weak`. Unlike the `_strong` version, a weak CAS is allowed to fail spuriously, just like the underlying LL/SC mechanism. Consider some function that atomically multiplies a value:

```
void atomicMultiply(int by)
{
  int expected = foo;
  // Which CAS should we use?
  while (!foo.compare_exchange_?(
    expected, expected * by)) {
    // Empty loop.
    // (On failure, expected is updated with
    //  foo's most recent value.)
  }
}
```

If we use `compare_exchange_strong` here, the compiler must emit nested loops: an inner one to protect us from spurious SC failures, and an outer one which repeatedly loads and multiplies `foo` until no other thread has modified it. With `compare_exchange_weak`, the compiler is free to generate a single loop instead, since we don't care about the difference between spurious failures and "normal" ones caused by another thread modifying `foo`.

9. *Do we always need sequentially consistent operations?*

All of our examples so far have used sequentially consistent reads and writes to prevent memory accesses from being rearranged in ways that break our code. We've also seen how weakly-ordered architectures like ARM use a pair of memory barriers to provide this guarantee. As you might expect, these barriers can have a non-trivial impact on performance. After all, they inhibit optimizations that your compiler and hardware would otherwise make.

What if we could avoid some of this slowdown? Consider some simple case like the spinlock from §5.2. Between the `lock()` and `unlock()` calls, we have a *critical section* where we can safely modify shared state protected by the lock. Outside this critical section, we only read and write to things that aren't shared with other threads.

Reduced instruction set computer, in contrast to a *complex instruction set computer* (CISC) architecture like x64.

†…though generally, we want to avoid cases where multiple threads are vying for the same variable for any significant amount of time.

```
deepThought.calculate(); // non-shared

lock(); // Lock; critical section begins
sharedState.subject =
  "Life, the universe and everything";
sharedState.answer = 42;
unlock(); // Unlock; critical section ends

demolishEarth(vogons); // non-shared
```

It's vital that reads and writes to the shared memory we're protecting don't move outside the critical section. But the opposite isn't true—the compiler and hardware could move as much as they desire *into* the critical section without causing any trouble. We have no problem with the following if it is somehow faster:

```
lock(); // Lock; critical section begins
deepThought.calculate(); // non-shared
sharedState.subject =
  "Life, the universe and everything";
sharedState.answer = 42;
demolishEarth(vogons); // non-shared
unlock(); // Unlock; critical section ends
```

So, how do we tell the compiler as much?

10. *Memory orderings*

By default, all atomic operations—including loads, stores, and the various flavors of RMW—are sequentially consistent. But this is only one of several orderings that we can give them. We'll examine each of them in turn, but a full list, along with the enumerations that the C and C++ API uses, is:

- Sequentially Consistent (`memory_order_seq_cst`)
- Acquire (`memory_order_acquire`)
- Release (`memory_order_release`)
- Relaxed (`memory_order_relaxed`)
- Acquire-Release (`memory_order_acq_rel`)
- Consume (`memory_order_consume`)

To specify one of these orderings, you provide it as an optional argument that we've slyly failed to mention so far:*

```
void lock()
{
  while (af.test_and_set(
      memory_order_acquire)) {
    // Spin with backoff
  }
}

void unlock()
{
  af.clear(memory_order_release);
}
```

Non-sequentially consistent loads and stores also use member functions of `std::atomic<>`:

```
int i = foo.load(memory_order_acquire);
```

Compare-and-swap operations are a bit odd in that they have *two* orderings: one for when the CAS succeeds, and one for when it fails:

```
while (!foo.compare_exchange_weak(
    expected, expected * by,
    memory_order_seq_cst, // On success
    memory_order_relaxed)) // On failure
    { /* empty loop */ }
```

With the syntax out of the way, let's look at what these orderings are and how we can use them. As it turns out, almost all of the examples we've seen so far don't actually need sequentially consistent operations.

10.1. *Acquire and release*

We've just seen acquire and release in action with the lock example from §9. You can think of these two as "one-way" barriers: the former allows other reads and writes to move past it in a *before* → *after* direction, and the latter works the opposite way, letting others move *after* → *before*. On ARM and other weakly-ordered architectures, this allows us to drop one of the memory barriers in each operation, such that

```
int acquireFoo()
{
  return foo.load(memory_order_acquire);
}

void releaseFoo(int i)
{
  foo.store(i, memory_order_release);
}
```

become:

*C, being C, defines separate functions for cases where you want to specify an ordering. `exchange()` becomes `exchange_explicit()`, a CAS becomes `compare_exchange_strong_explicit()`, and so on.

```
acquireFoo:          releaseFoo:
  ldr r3, <&foo>       ldr r3, <&foo>
  ldr r0, [r3, #0]     dmb
  dmb                  str r0, [r3, #0]
  bx lr                bx lr
```

Together, these provide *writer* → *reader* synchronization: if thread W stores a value with release semantics, and thread R loads that value with acquire semantics, then all writes made by W before its store-release are observable to R after its load-acquire. If this sounds familiar, it's exactly what we were trying to achieve in §1 and §2:

```
int v;
std::atomic_bool v_ready(false);

void threadA()
{
  v = 42;
  v_ready.store(true, memory_order_release);
}

void threadB()
{
  while (!v_ready.load(memory_order_acquire)) {
    // spin
  }
  assert(v == 42); // Must be true
}
```

10.2. *Relaxed*

Relaxed atomic operations are used when a variable will be shared between threads, but *no specific order* is required.

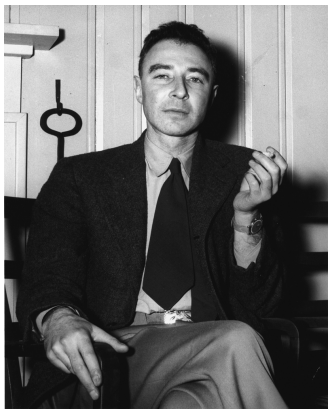

Figure 4: Relaxed atomic operations circa 1946

This might seem like a rare occurrence, but is surprisingly common. Recall our examples from §3 and §5 where some worker thread increments a counter which is read by a UI thread to show progress. That counter could be incremented with `atomic_fetch_add()` using `memory_order_relaxed`. All we need is atomicity—nothing is synchronized by the counter.

Relaxed reads and writes are also useful for sharing flags between threads. Consider some thread that loops until told to exit:

```
atomic_bool stop(false);

void worker()
{
  while (!stop.load(memory_order_relaxed)) {
    // Do good work.
  }
}

int main()
{
  launchWorker();
  // Wait some...
  stop = true; // seq_cst
  joinWorker();
}
```

We don't care if the contents of the loop are rearranged around the load. Nothing bad will happen so long as `stop` only indicates that the worker should exit and doesn't "announce" any new data to be read by the worker.

Finally, relaxed loads are commonly used with CAS loops. Return to our lock-free multiply:

```
void atomicMultiply(int by)
{
  int expected = foo.load(memory_order_relaxed);

  while (!foo.compare_exchange_?(
    expected, expected * by,
    memory_order_release,
    memory_order_relaxed))
  { /* empty loop */ }
}
```

All of the loads can be relaxed, as we don't need to enforce any sort of ordering until we've successfully modified our value. The initial load of `expected` isn't even strictly necessary—it just saves us a loop iteration if no other thread modifies `foo` before the CAS.

10.3. *Acquire-Release*

`memory_order_acq_rel` is used with atomic RMW operations that need to both load-acquire *and* store-release a value. A typical example involves thread-safe reference counting, like in C++'s `shared_ptr`:

```
atomic_int refCount;

void inc()
{
  refCount.fetch_add(1, memory_order_relaxed);
}


void dec()
{
  if (refCount.fetch_sub(1,
      memory_order_acq_rel) == 1) {
    // No more references.
    // Delete the data.
  }
}
```

Order doesn't matter when incrementing the reference count since no action is taken as a result. However, when we decrement, we must ensure that:

1. All reads and writes to the referenced object happen *before* the count reaches zero.
2. Deletion occurs *after* the reference count drops to zero.*

Curious readers might be wondering about the difference between acquire-release and sequentially consistent operations. To quote Hans Boehm, the chair of the ISO C++ Concurrency Study Group,

> The difference between `acq_rel` and `seq_cst` is generally whether the operation is required to participate in the single global order of sequentially consistent operations.

In other words, acquire-release provides order relative to the variable being load-acquired and store-released, whereas sequentially consistent operation provides some *global* order across the entire program. If the distinction still seems hazy, you're not alone. Boehm continues with,

> This has subtle and unintuitive effects. The [barriers] in the current standard may be the most experts-only construct we have in the language.

10.4. *Consume*

Last but not least, we have `memory_order_consume`. Consider a scenario where data is rarely changed, but frequently read by multiple threads. Perhaps it is a pointer in the kernel to information about peripherals plugged into the machine. This data will change *very* infrequently, so it makes sense to optimize reads as much as possible. Given what we know so far, the best we can do is:

```
std::atomic<PeripheralData*> peripherals;

// Writers:
PeripheralData* p = kAllocate(sizeof(*p));
populateWithNewDeviceData(p);
peripherals.store(p, memory_order_release);

// Readers:
PeripheralData* p =
    peripherals.load(memory_order_acquire);
if (p != nullptr) {
    doSomethingWith(p->keyboards);
}
```

Since we're trying to optimize readers as much as possible, it would be really nice if we could avoid a memory barrier on weakly-ordered systems. As it turns out, we usually can. Since the data we examine (`p->keyboards`) is *dependent* on the value of `p`, most platforms—even weakly-ordered ones—cannot reorder the initial load (`p = peripherals`) to take place after its use (`p->keyboards`).† So long as we convince the compiler not to make any similar speculations, we're in the clear.

This is what `memory_order_consume` is for. Change readers to:

```
PeripheralData* p =
    peripherals.load(memory_order_consume);
if (p != nullptr) {
    doSomethingWith(p->keyboards);
}
```

and an ARM compiler could emit:

```
  ldr r3, &peripherals
  ldr r3, [r3]
  // Look ma, no barrier!
  cbz r3, was_null // Check for null
  ldr r0, [r3, #4] // Load p->keyboards
  b doSomethingWith(Keyboards*)
was_null:
  ...
```

Sadly, the emphasis here is on *could*. Figuring out what constitutes a "dependency" between expressions isn't as trivial as one might hope,‡ so all compilers currently convert consume operations to acquires.

*This can be optimized even further by making the acquire barrier only occur conditionally, when the reference count is zero. Standalone barriers are outside the scope of this paper, since they're almost always pessimal compared to a combined load-acquire or store-release, but you can see an example here: http://www.boost.org/doc/libs/release/doc/html/atomic/usage_examples.html.

†Much to everybody's chagrin, this *isn't* the case on some extremely weakly-ordered architectures like DEC Alpha.

‡Even the experts in the ISO committee's concurrency study group, SG1, came away with different understandings. See N4036 for the gory details. Proposed solutions are explored in P0190R3 and P0462R1.

10.5. HC SVNT DRACONES

Non-sequentially consistent orderings have many subtleties, and a slight mistake can cause elusive Heisenbugs that only occur sometimes, on some platforms. Before reaching for them, ask yourself:

Am I using a well-known and understood pattern (such as the ones shown above)?

Are the operations occurring in a tight loop?

Does every microsecond count here?

If the answer isn't yes for at least one of these, default to sequentially consistent operations. Otherwise, be sure to give your code extra review and testing.

11. *Hardware convergence*

Those familiar with the platform may have noticed that all ARM assembly shown here is from the seventh version of the architecture. Excitingly, the current (eighth) generation offers a massive improvement for lockless code. Since most programming languages have converged on the memory model we've been exploring, ARMv8 processors offer dedicated load-acquire and store-release instructions, `lda` and `stl`. We can use them to implement everything we've discussed here without resorting to memory barriers. Hopefully, future CPU architectures will follow suit.

12. *Optimizations*

Finally, one should realize that while atomic operations do prevent certain optimizations, they aren't somehow immune to all of them. The optimizer can do fairly mundane things, such as replacing `foo.fetch_and(0)` with `foo = 0`, but it can also produce surprising results. Consider:

```
while (tmp = foo.load(memory_order_relaxed)) {
  doSomething(tmp);
}
```

Since relaxed loads provide no ordering guarantees, the compiler is free to unroll the loop as much as it pleases, perhaps into:

```
while (tmp = foo.load(memory_order_relaxed)) {
  doSomething(tmp);
  doSomething(tmp);
  doSomething(tmp);
  doSomething(tmp);
}
```

In some cases, "fusing" reads or writes like this is unacceptable, so we must prevent it with `volatile` casts or incantations like `asm volatile("" ::: "memory")`.* The Linux kernel provides `READ_ONCE()` and `WRITE_ONCE()` macros for this exact purpose.†

13. *Takeaways*

We've only scratched the surface here, but hopefully you now know:

- Why compilers and CPU hardware reorder loads and stores.
- Why we need special tools to prevent these reorderings for communications between threads.
- How we can guarantee *sequential consistency* in our programs.
- Atomic *read-modify-write* operations.
- How atomic operations can be implemented on weakly-ordered hardware, and what implications this can have for a language-level API.
- How we can *carefully* optimize lockless code using alternative memory orderings.
- How the compiler might optimize atomic operations even further, and what we can do to prevent certain undesirable optimizations.

To learn more, see the additional resources below, or examine lock-free data structures and algorithms, such as a *single-producer/single-consumer* (SP/SC) queue or *read-copy-update* (RCU).‡

Good luck and godspeed!

*See https://stackoverflow.com/a/14983432.
†See N4374 and the kernel's `compiler.h` for details.
‡See the Linux Weekly News article, *What is RCU, Fundamentally?* for an introduction.

Additional Resources

atomic<> Weapons: The C++11 Memory Model and Modern Hardware by Herb Sutter, a three-hour talk that provides a more thorough dive on the concepts discussed here. Also the source of figures 2 and 3.

Futexes are Tricky, a paper by Ulrich Drepper on how mutexes and other synchronization primitives can be built in Linux using atomic operations and syscalls.

Is Parallel Programming Hard, And, If So, What Can You Do About It?, by Paul E. McKenney, an *incredibly* comprehensive book covering parallel data structures and algorithms, transactional memory, cache coherence protocols, CPU architecture specifics, and more.

Memory Barriers: a Hardware View for Software Hackers, an older but much shorter piece by McKenney explaining how memory barriers are implemented in the Linux kernel on various architectures.

Preshing On Programming, a blog with many excellent articles on lockless concurrency.

No Sane Compiler Would Optimize Atomics, a discussion of how atomic operations are handled by current optimizers. Available as a writeup, N4455, and as a CppCon talk.

cppreference.com, an excellent reference for the C and C++ memory model and atomic API.

Matt Godbolt's Compiler Explorer, an online tool that provides live, color-coded disassembly using compilers and flags of your choosing. *Fantastic* for examining what compilers emit for various atomic operations on different architectures.

About This Document

Contributing

Contributions are welcome! Sources and history are available on Gitlab and Github. This paper is prepared in LaTeX—if you're not familiar with it, feel free to contact the author (via email, by opening an issue, etc.) in lieu of pull requests.

License