What every systems programmer should know about concurrency

Matt Kline

April 28, 2020

Abstract

Systems programmers are familiar with tools like mutexes, semaphores, and condition variables. But how do they work? How do we write concurrent code when they're not available, 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 compiler and hardware conspire to turn your code into things you didn't write, running in orders you never asked for, how do multithreaded programs work at all? Concurrency is a complicated and unintuitive topic, but let's try to cover some fundamentals.

Contents

1. *Background*

Modern computers run many instruction streams concurrently. On single-core machines, they take turns, sharing the CPU in short slices of time. On multi-core machines, several can run in parallel. We call them many names—processes, threads, tasks, interrupt service routines, and more—but most of the same principles apply across the board.

While computer scientists have built lots of great abstractions, these instruction streams (let's call them all *threads* for the sake of brevity) ultimately interact by sharing bits of state. For this to work, we need to understand the order in which threads read and write to memory. Consider a simple example where thread *A* shares an integer with others. It writes the integer to some variable, then sets a flag to instruct other threads 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) { /* wait */ }
  const int my_v = v;
  // Do something with my_v...
}
```

We need to make sure that other threads only observe *A's* write to `v_ready` *after A's* write to `v`. (If another thread can "see" `v_ready` become true before it sees `v` become 42, this simple scheme won't work.)

You would think it's trivial to guarantee this order, but nothing is as it seems. For starters, any optimizing compiler will rewrite your code to run faster on the hardware it's targeting. 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 improve locality.† Variables can be assigned to the same memory location if they're never used at the same time. Calculations can be made speculatively, before a branch is taken, then ignored if the compiler guessed incorrectly.‡

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

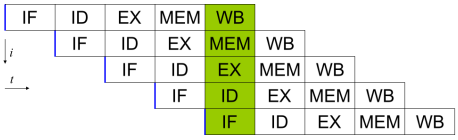

Figure 1: A traditional five-stage CPU pipeline with fetch, decode, execute, memory access, and write-back stages. Modern designs are much more complicated, often reordering instructions on the fly. Image courtesy of Wikipedia.

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

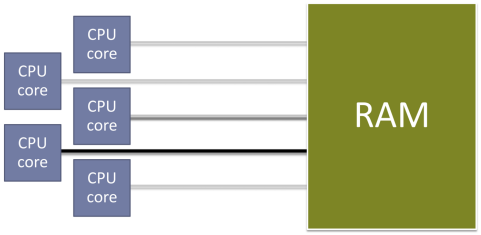

Figure 2: An idealized multi-core processor where cores take turns accessing a single shared set of memory.

But the world isn't so simple. While processor speeds have increased exponentially over 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 its data from memory. Hardware designers have compensated by placing a growing 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 throughput (see Figure 1). When the result of one instruction is needed by a subsequent 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, they will be read and written all at once. This usually provides a massive speedup, but as we'll see in §12, can bite us when a line must be shared between cores.

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

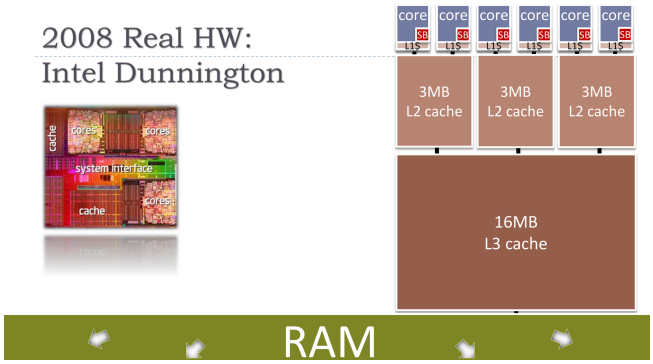

Figure 3: A common memory hierarchy for modern multi-core processors

All of these complications mean that there is no consistent concept of "now" in a multithreaded program, especially on a multi-core CPU. Creating some sense of order between threads is a team effort of the hardware, the compiler, the programming language, and your application. Let's explore what we can do, and what tools we will need.

2. *Enforcing law and order*

Creating order in multithreaded programs requires different approaches on each CPU architecture. For many years, systems languages like C and C++ had no notion of concurrency, forcing developers to use assembly or compiler extensions. This was finally fixed in 2011, when both languages' ISO standards added synchronization tools. So long as you use them correctly, the compiler will prevent any reorderings—both by its own optimizer, and by the CPU—that cause data races.*

Let's try our previous example again. For it to work, the "ready" flag needs to use an *atomic type*.

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

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

void threadB()
{
  while (!v_ready) { /* wait */ }
  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>` and `<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 variables' loads and stores aren't reordered around theirs.

Informally, we can think of atomic variables as rendezvous points for threads. By making `v_ready` atomic, `v = 42` is now guaranteed to happen before `v_ready = true` in thread *A*, just as `my_v = v` must happen after reading `v_ready` in thread *B*. Formally, atomic types establish a *single total modification order* where, "[…] 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*.

3. *Atomicity*

But order is only one of the vital ingredients for inter-thread communication. The other is what atomic types are named for: atomicity. Something is *atomic* if it cannot be divided into smaller parts. If threads don't use atomic reads and writes to share data, we're still in trouble.

Consider a program with two threads. One processes a list of files, incrementing a counter each time it finishes working on one. The other handles the user interface, periodically reading the counter to update a progress bar. If that counter is a 64-bit integer, we can't access it atomically on 32-bit machines, since we need two loads or stores to read or write the entire value. If we're particularly unlucky, the first thread could be halfway through writing the counter when the second thread reads it, receiving garbage. These unfortunate occasions are called *torn reads and writes*.

If reads and writes to the counter are atomic, however, our problem disappears. We can see that, compared to the difficulties of establishing the right order, atomicity is fairly straightforward: just make sure that any variables used for thread synchronization are no larger than the CPU word size.

4. *Arbitrarily-sized "atomic" types*

Along with `atomic_int` and friends, C++ provides the template `std::atomic<T>` for defining arbitrary atomic types. C, lacking a similar language feature but wanting to provide the same functionality, added an `_Atomic` keyword. If `T` is larger than the machine's word size, the compiler and the language runtime automatically surround the variable's reads and writes with locks. If you want to make sure this isn't happening,† you can check with:

*The ISO C11 standard lifted its 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++.

†…which is most of the time, since we're usually using atomic operations to avoid locks in the first place.

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

In most cases,* this information is known at compile time. Consequently, C++17 added `is_always_lock_free`:

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

5. *Read-modify-write*

Loads and stores are all well and good, but sometimes we need to read a value, modify it, and write it back as 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, the UI might want to show how many were processed per second. We could implement this by having the UI thread read the counter then zero it each second. But we could get the following race condition if reading and zeroing are separate steps:

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 atomically exchanges the current value with zero, 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 simple spinlock:

```
std::atomic_flag af;

void lock()
{
  while (af.test_and_set()) { /* wait */ }
}

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

If we call `lock()` and the previous value is `false`, we are the first to acquire the lock, and 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 simple operation on it (addition, subtraction, bitwise AND, OR, XOR), and return its previous value—all as one atomic operation. You might have noticed in the exchange example that the worker thread's additions must also be atomic, or else we could get 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 executed atomically:

```
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 task that we might want to cancel. We'll give it three states: *idle*, *running*, and *cancelled*, and write a loop that exits when it is cancelled.

*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.

```
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 want to cancel the task if it's running, but do nothing if it's idle, we could CAS:

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

6. *Atomic operations as building blocks*

Atomic loads, stores, and RMW operations are the building blocks for every single concurrency tool. It's useful to split those tools into two camps: *blocking* and *lockless*.

Blocking synchronization methods are usually simpler to reason about, but they can make threads pause for arbitrary amounts of time. For example, consider a mutex, which forces threads to take turns accessing shared data. If some thread locks the mutex and another tries to do the same, the second thread must wait—or *block*—until the first thread releases the lock, however long that may be. Blocking mechanisms are also susceptible to *deadlock* and *livelock*—bugs where the entire system "gets stuck" due to threads waiting for each other.

In contrast, lockless synchronization methods ensure that the program is always making forward progress. These are *non-blocking* since no thread can cause another to wait indefinitely. Consider a program that streams audio, or an embedded system where a sensor triggers an interrupt service routine (ISR) when new data arrives. We want lock-free algorithms and data structures in these situations, since blocking could break them. (In the first case, the user's audio will begin to stutter if sound data isn't provided at the bitrate it is consumed. In the second, subsequent sensor inputs could be missed if the ISR does not complete as quickly as possible.)

It's important to point out that lockless algorithms are not somehow better or faster than blocking ones—they are just different tools designed for different jobs. We should also note that algorithms aren't automatically lock-free just because they only use atomic operations. Our primitive spinlock from §5.2 is still a blocking algorithm even though it doesn't use any OS-provided syscalls to put the blocked thread to sleep.*

Of course, there are situations where either blocking or lockless approaches would work.† Whenever performance is a concern, *profile!* Performance depends on many factors, ranging from the number of threads at play to the specifics of your CPU. And as always, consider the tradeoffs you make between complexity and performance—concurrency is a perilous art.

7. *Sequential consistency on weakly-ordered hardware*

Different hardware architectures provide different ordering guarantees. or *memory models*. For example, x64 is relatively *strongly-ordered*, and can be trusted to preserve some system-wide order of loads and stores in most cases. Other architectures like ARM are *weakly-ordered*, so you can't assume that loads and stores are executed in program order unless the CPU is given special instructions—called *memory barriers*—to not shuffle them around.

It's helpful to see how atomic operations work in a weakly-ordered system, both to understand what's happening in hardware, and to see why the C and C++ concurrency models were designed as they were.‡ Let's examine ARM, since it's both popular and straightforward. Consider the simplest atomic operations: loads and stores. Given some `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
```

We load the address of our atomic variable into a scratch

*Putting a blocked thread to sleep is often an optimization, since the operating system's scheduler can run other threads on the CPU until the sleeping one is unblocked. Some concurrency libraries even offer hybrid locks which spin briefly, then sleep. (This avoids the cost of context switching away from the current thread if it is blocked for less than the spin length, but avoids wasting CPU time in a long-running loop.)

†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 specific to C and C++. Other systems programming languages like D and Rust have converged on similar models.

register (r3), sandwich our load or store between memory barriers (dmb), then return. The barriers give us sequential consistency—the first ensures that prior reads and writes can't be placed after our operation, and the second ensures that subsequent reads and writes can't be placed before it.

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

Like many other RISC* architectures, ARM lacks dedicated RMW instructions. And since the processor can context switch to another thread at any time, we can't build RMW ops from normal loads and stores. Instead, we need special instructions: *load-link* and *store-conditional* (LL/SC). The two work in tandem: A load-link reads a value from an address—like any other load—but also instructs the processor to monitor that address. Store-conditional writes the given value *only if* no other stores were made to that address since the corresponding load-link. Let's see them in action with 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 to foo since our LL, so we try again. 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, it would take too much CPU hardware to track load-linked addresses for every single byte on the machine. 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's preceded by a write to *any* address in the monitored block, not just the specific one that was load-linked.

This is especially troublesome for compare and swap, and is the raison d'être for compare_exchange_weak. To see why, consider a function that atomically multiplies a value, even though there's no atomic instruction to read-multiply-write in any common architecture.

```
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.)
  }
}
```

Many lockless algorithms use CAS loops like this to atomically update a variable when calculating its new value isn't atomic. They:

1. Read the variable.
2. Perform some (non-atomic) operation on its value.
3. CAS the new value with the previous one.
4. If the CAS failed, another thread beat us to the punch, so try again.

If we use compare_exchange_strong for this family of algorithms, the compiler must emit nested loops: an inner one to protect us from spurious SC failures, and an outer one which repeatedly performs our operation until no other thread has interrupted us. But unlike the _strong version, a weak CAS is allowed to fail spuriously, just like the LL/SC mechanism that implements it. So, with compare_exchange_weak, the compiler is free to generate a single loop, since we don't care about the difference between retries from spurious SC failures and retries caused by another thread modifying our variable.

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

All of our examples so far have been sequentially consistent to prevent reorderings that break our code. We've also seen how weakly-ordered architectures like ARM use memory barriers to create sequential consistency. But as you might expect, these barriers can have a noticeable 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 a simple case like the spinlock from §5.2. Between the lock()

*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.

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.

```
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 shared memory don't move outside the critical section. But the opposite isn't true! The compiler and hardware could move as much as they want *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, 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 pick an ordering, 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)) { /* wait */ }
}

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 them as "one-way" barriers: an acquire allows other reads and writes to move past it, but only in a *before* → *after* direction. A release works the opposite way, letting things 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)) {
    // wait
  }
  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. While this might seem rare, it's surprisingly common.

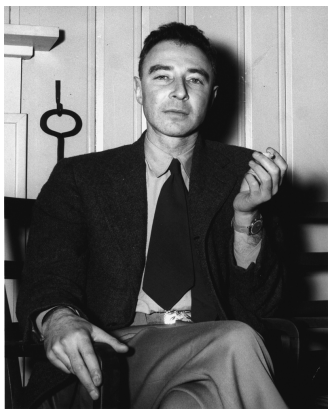

Figure 4: Relaxed atomic operations circa 1946

Recall our examples from §3 and §5 where some worker thread is incrementing a counter, which is then read by a UI thread. That counter could be incremented with `fetch_add(1, memory_order_relaxed)`, since 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` is only used to tell the worker to exit, and not to "announce" any new data.

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_weak(
    expected, expected * by,
    memory_order_release,
    memory_order_relaxed))
  { /* empty loop */ }
}
```

All of the loads can be relaxed—we don't need to enforce any order 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 access to the referenced object happens *before* the count reaches zero.
2. Deletion happens *after* the reference count reaches zero.*

Curious readers might be wondering about the difference between acquire-release and sequentially consistent operations. To quote Hans Boehm, 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 goes on to say,

> 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 often read by many threads. Maybe we're writing a kernel and we're tracking the peripherals plugged into the machine. This information will change *very* infrequently—only when someone plugs or unplugs something—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);
}
```

To further optimize readers, it would be great if loads could avoid a memory barrier on weakly-ordered systems. As it turns out, they 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 happen 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 in a tight loop?*
>
> *Does every microsecond count here?*

If the answer isn't yes to several of these, stick to to sequentially consistent operations. Otherwise, be sure to give your code extra review and testing.

11. *Hardware convergence*

Those familiar with ARM may have noticed that all assembly shown here is for the seventh version of the architecture. Excitingly, the eighth generation offers massive improvements 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`. Hopefully, future CPU architectures will follow suit.

12. *Cache effects and false sharing*

As if all of this wasn't enough to keep rattling around in your head, modern hardware gives us one more wrinkle. Recall that memory is transferred between main RAM and the CPU in chunks called cache lines. These lines are also the smallest unit transferred *between* cores and their respective caches—if one core writes a value and another core reads it, the entire line containing that value must be transferred from the first core's cache(s) to the second core's in order to keep their "view" of memory coherent.

This can have a surprising performance impact. Consider a readers-writer lock, which avoids races by ensuring that shared data has one writer *or* any number of readers, but never both at the same time. At its core, it resembles the following:

```
struct RWLock {
  int readers;
  bool hasWriter; // Zero or one writers
};
```

Writers must block until `readers` reaches zero, but readers can take the lock with an atomic RMW operation whenever `hasWriter` is `false`.

Naïvely, it seems like this offers a huge performance win over exclusive locks (e.g., mutexes, spinlocks, etc.) for cases where we read the shared data more often than we write, but this fails to consider cache effects. If multiple readers—each running on a different core—simultaneously take the lock, its cache line will "ping-pong" between those cores' caches. Unless critical sections are very large, resolving this contention will likely take more time than the critical sections themselves,* even though the algorithm doesn't block.

This slowdown is even more insidious when it occurs between unrelated variables that happen to be placed on the same cache line. When designing concurrent data structures or algorithms, this *false sharing* must be taken into account. One way to avoid it is to pad atomic variables with a cache line of unshared data, but this is obviously a large space-time tradeoff.

13. *If concurrency is the question, `volatile` is not the answer.*

Before we go, we should lay a common misconception surrounding the `volatile` keyword to rest. Perhaps because of how it worked in older compilers and hardware, or due to its different meaning in languages like Java and C#,† some believe that the keyword is useful for building concurrency tools. Except for one specific case (see §14), this is false.

The purpose of `volatile` is to inform the compiler that a value can be changed by something besides the program we're executing. This is useful for memory-mapped I/O (MMIO), where hardware translates reads and writes to certain addresses into instructions for the devices connected to the CPU. (This is how most machines ultimately interact with the outside world.) `volatile` implies two guarantees:

1. The compiler will not elide loads and stores that seem "unnecessary". For example, if I have some function:

   ```
   void write(int* t)
   {
     *t = 2;
     *t = 42;
   }
   ```

   the compiler would normally optimize it to:

   ```
   void write(int* t) { *t = 42; }
   ```

   `*t = 2` is usually assumed to be a *dead store* that does nothing. But, if `t` points to some MMIO register, it's not safe to make this assumption—each write could have some effect on the hardware it's interacting with.

2. The compiler will not reorder `volatile` reads and writes with respect to other `volatile` ones for similar reasons.

*On some systems, a cache miss can cost more than two orders of magnitude than an atomic RMW operation. See Paul E. McKenney's talk from CppCon 2017 for more details.

†Unlike in C and C++, `volatile` *does* enforce ordering in those languages.

These rules don't give us the atomicity or order we need for safe inter-thread communication. Notice that the second guarantee only prevents `volatile` operations from being reordered in relation *to each other*—the compiler is still free to rearrange all other "normal" loads and stores around them. And even if we set that problem aside, `volatile` does not emit memory barriers on weakly-ordered hardware. The keyword only works as a synchronization mechanism if both your compiler *and* your hardware perform no reordering. Don't bet on that.

14. *Atomic fusion*

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);
}
```

If "fusing" reads or writes like this is unacceptable, 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.†

15. *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 to communicate 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 non-sequentially-consistent memory orderings.
- How *false sharing* can impact the performance of concurrent memory access.
- Why `volatile` is an inappropriate tool for inter-thread communication.
- How to prevent the compiler from fusing atomic operations in undesirable ways.

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

C++ atomics, from basic to advanced. What do they really do? by Fedor Pikus, a hour-long talk on this topic.

atomic<> Weapons: The C++11 Memory Model and Modern Hardware by Herb Sutter, a three-hour talk that provides a deeper dive. 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.

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.

Colophon

This guide was typeset using LuaLaTeX in Matthew Butterick's Equity, with code in Matthias Tellen's mononoki. The title is set in Neue Haas Grotesk, a Helvetica restoration by Christian Schwartz.