Concurrency Primer*

Matt Kline and Ching-Chun (Jim) Huang

May 6, 2024

Abstract

System programmers are acquainted with tools such as mutexes, semaphores, and condition variables. However, the question remains: how do these tools work, and how do we write concurrent code in their absence? For example, when working in an embedded environment beneath the operating system, or when faced with hard time constraints that prohibit blocking. Furthermore, since the compiler and hardware often combine to transform code into an unanticipated order, how do multithreaded programs work? Concurrency is a complex and counterintuitive topic, but let us endeavor to explore its fundamental principles.

1 *Background*

Modern computers execute multiple instruction streams concurrently. On single-core systems, these streams alternate, sharing the CPU in brief time slices. Multi-core systems, however, allow several streams to run in parallel. These streams are known by various names such as processes, threads, tasks, interrupt service routines (ISR), among others, yet many of the same principles govern them all.

Despite the development of numerous sophisticated abstractions by computer scientists, these instruction streams—hereafter referred to as "*threads*" for simplicity—primarily interact through shared state. Proper functioning hinges on understanding the sequence in which threads read from and write to memory. Consider a simple scenario where thread A communicates an integer with other threads: it writes the integer to a variable and then sets a flag, signaling other threads to read the newly stored value. This operation could be conceptualized in code as follows:

```
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 must ensure that other threads only observe A's write to `v_ready` *after* A*'s* write to `v`. If another thread can "see" `v_ready` becoming true before observing `v` becoming 42, this simple scheme will not work correctly.

One might assume it is straightforward to ensure this order, yet the reality is often more complex. Initially, any optimizing compiler will restructure your code to enhance performance on its target hardware. The primary objective

*The original title was "What every systems programmer should know about concurrency".

is to maintain the operational effect within *the current thread*, allowing reads and writes to be rearranged to prevent pipeline stalls* or to optimize data locality.†

Variables may be allocated to the same memory location if their usage does not overlap. Furthermore, calculations might be performed speculatively ahead of a branch decision and subsequently discarded if the branch prediction proves incorrect.‡

Even without compiler alterations, we would face challenges because our hardware complicates matters further! Modern CPUs operate in a fashion far more complex than what traditional pipelined methods, like those depicted in Figure 1, suggest. They are equipped with multiple data paths tailored for various instruction types and schedulers that reorder and direct instructions through these paths.

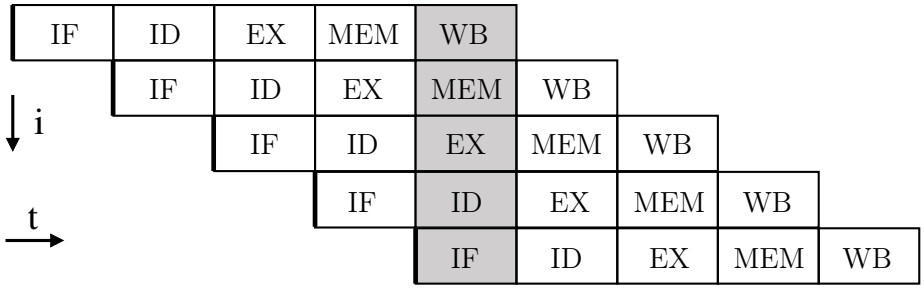

Figure 1: A traditional five-stage CPU pipeline with fetch, decode, execute, memory access, and write-back stages. Modern designs are much more complicated, often reordering instructions on the fly.

It is quite common to form oversimplified views about memory operations. Picturing a multi-core processor setup might lead us to envision a model similar to Figure 2, wherein each core alternately accesses and manipulates the system's memory.

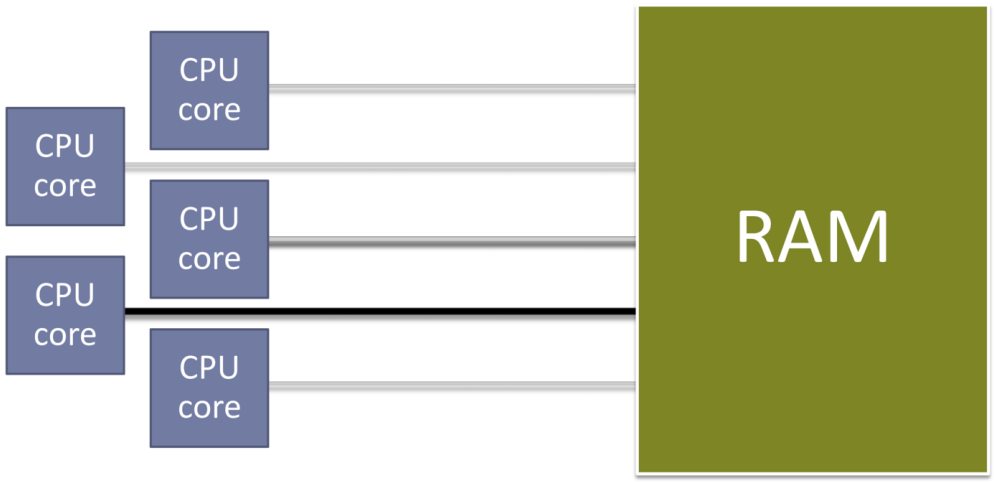

Figure 2: An idealized multi-core processor where cores take turns accessing a single shared set of memory.

The reality is far from straightforward. Although processor speeds have surged exponentially in recent decades, RAM has struggled to match pace, leading to a significant gap between the execution time of an instruction and the time required to fetch its data from memory. To mitigate this, hardware designers have incorporated increasingly complex hierarchical caches directly onto the CPU die. Additionally, each core often features a *store buffer* to manage pending writes while allowing further instructions to proceed. Ensuring this memory system remains *coherent*, thus allowing writes made by one core to be observable by others even when utilizing different caches, presents a significant challenge.

*Most CPU architectures execute segments of multiple instructions concurrently to improve throughput (refer to Figure 1). A stall, or suspension of forward progress, occurs when an instruction awaits the outcome of a preceding one in the pipeline until the necessary result becomes available.

†RAM accesses data not byte by byte, but in larger units known as *cache lines*. Grouping frequently used variables on the same cache line means they are processed together, significantly boosting performance. However, as discussed in §12, this strategy can lead to complications when cache lines are shared across cores.

‡Profile-guided optimization (PGO) often employs this strategy.

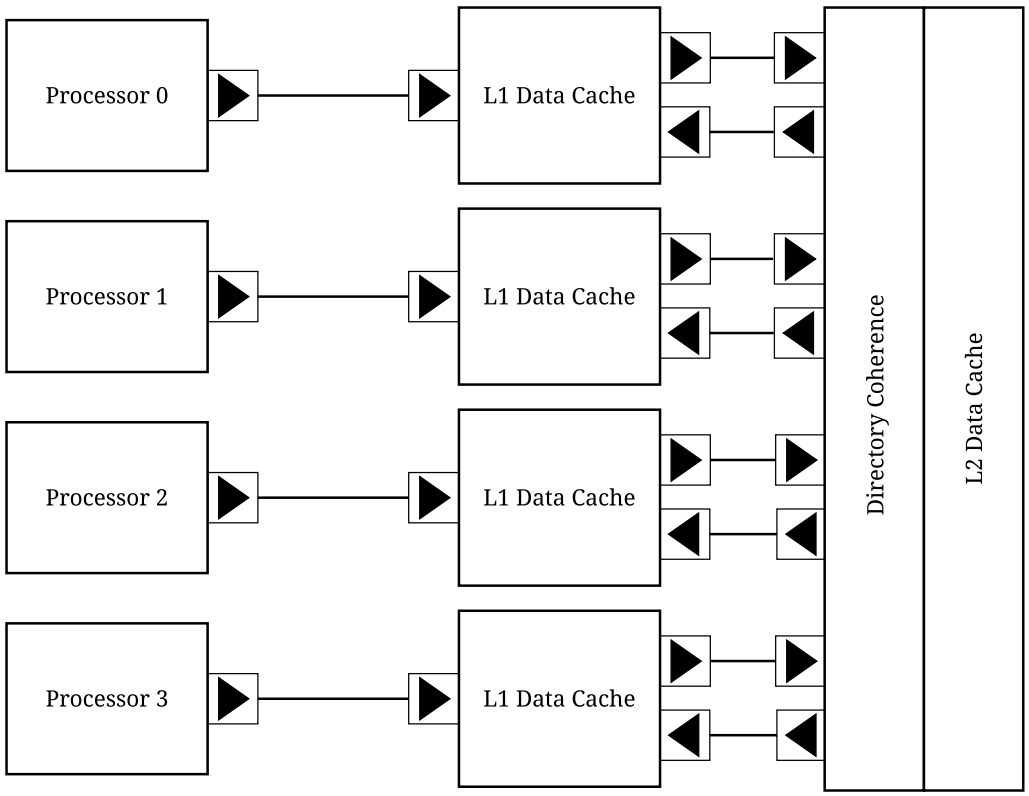

Figure 3: A common memory hierarchy for modern multi-core processors

The myriad complexities within multithreaded programs on multi-core CPUs lead to a lack of a uniform concept of "now". Establishing some semblance of order among threads requires a concerted effort involving the hardware, compiler, programming language, and your application. Let's delve into our options and the tools necessary for this endeavor.

2 *Enforcing law and order*

Establishing order in multithreaded programs varies across different CPU architectures. For years, systems languages like C and C++ lacked built-in concurrency mechanisms, compelling developers to rely on assembly or compiler-specific extensions. This gap was bridged in 2011 when the ISO standards for both languages introduced synchronization tools. Provided these tools are used correctly, the compiler ensures that neither its optimization processes nor the CPU will perform reorderings that could lead to data races.*

To ensure our earlier example functions as intended, the "ready" flag must utilize an *atomic type*.

```
#include <stdatomic.h>
int v = 0;
atomic_bool v_ready = false;

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

int bv;
```

*The ISO C standard adopted its concurrency features, almost directly, from the C++ standard. Thus, the functionalities discussed should be the same in both languages, with some minor syntactical differences favoring C++ for clarity.

```
void *threadB()
{
    while(!v_ready) { /* wait */ }
    bv = v;
    /* Do something */
}
```

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 are not 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 can not be divided into smaller parts. If threads do not use atomic reads and writes to share data, we are still in trouble.

Consider a program with two threads. One thread processes a list of files, incrementing a counter each time it finishes working on one. The other thread handles the user interface, periodically reading the counter to update a progress bar. If that counter is a 64-bit integer, we can not access it atomically on 32-bit machines, since we need two loads or stores to read or write the entire value. If we are 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 is not happening,* you can check with:

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

In most cases,† this information is known at compile time. Consequently, C++ 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 are represented as member functions of `std::atomic<T>`. In C, they are freestanding functions.

*…which is most of the time, since we are usually 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 do not guarantee atomicity for unaligned reads and writes.

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:

```
atomic_flag af = ATOMIC_FLAG_INIT;

void lock()
{
    while (atomic_flag_test_and_set(&af)) { /* wait */ }
}

void unlock() { atomic_flag_clear(&af); }
```

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 a simple operation on it (such as addition, subtraction, or bitwise AND, OR, XOR) and return its previous value, all as part of a single atomic operation. You might recall from the exchange example that additions by the worker thread must be atomic to prevent races, 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;
```

```
    }
    expected = *this;
    return false;
}
```

The `_strong` suffix may leave you wondering if there is a corresponding "weak" CAS. Indeed, there is. However, we will delve into that topic later 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.

```
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 is running, but do nothing if it is 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 is useful to split those tools into two camps: *blocking* and *lockless*.

Blocking synchronization methods are generally easier to understand, but they can cause threads to pause for unpredictable durations. Take a mutex as an example: it requires threads to access shared data sequentially. If a thread locks the mutex and another attempts to lock it too, the second thread must wait, or *block*, until the first one unlocks it, regardless of the wait time. Additionally, blocking mechanisms are prone to *deadlock* and *livelock*, issues that lead to the system becoming immobilized as threads perpetually wait on 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 is not 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.)

Lockless algorithms are not inherently superior or quicker than blocking ones; they serve different purposes with their own design philosophies. Additionally, the mere use of atomic operations does not render algorithms lock-free. For example, our basic spinlock discussed in §5.2 is still considered a blocking algorithm even though it eschews OS-specific syscalls for making the blocked thread sleep. Putting a blocked thread to sleep is often an optimization, allowing the operating system's scheduler to allocate CPU resources to active threads until the blocked one is revived. Some concurrency libraries even introduce hybrid locks that combine brief spinning with sleeping to balance CPU usage and context-switching overheads.

Both blocking and lockless approaches have their place in software development. When performance is a key consideration, it is crucial to profile your application. The performance impact varies with numerous factors, such as

thread count and CPU architecture specifics. Balancing complexity and performance is essential in concurrency, a domain fraught with challenges.

7 *Sequential consistency on weakly-ordered hardware*

Different hardware architectures offer distinct memory models or *memory models*. For instance, x64 architecture[*] is known to be *strongly-ordered*, generally ensuring a global sequence for loads and stores in most scenarios. Conversely, architectures like ARM are considered *weakly-ordered*, meaning one should not expect loads and stores to follow the program sequence without explicit instructions to the CPU. These instructions, known as *memory barriers*, are essential to prevent the reordering of these operations.
It is 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 is 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 register (`r3`), place our load or store operation between memory barriers (`dmb`), and then proceed. These barriers ensure sequential consistency: the first barrier guarantees that previous reads and writes are not reordered to follow our operation, and the second ensures that future reads and writes are not reordered to precede it.

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

Like many RISC[‡] architectures, ARM does not have dedicated RMW instructions. Given that the processor may switch contexts to another thread at any moment, constructing RMW operations from standard loads and stores is not feasible. Special instructions are required instead: *load-link* and *store-conditional* (LL/SC). These instructions are complementary: load-link performs a read operation from an address, similar to any load, but it also signals the processor to watch that address. Store-conditional executes a write operation only if no other writes have occurred at that address since its paired load-link. This mechanism is illustrated through an atomic fetch and add example.
On ARM,

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

compiles to:

[*]Also known as x86-64, x64 is a 64-bit extension of the x86 instruction set, officially unveiled in 1999. This extension heralded the introduction of two novel operation modes: 64-bit mode for leveraging the full potential of 64-bit processing and compatibility mode for maintaining support for 32-bit applications. Initially developed by AMD and publicly released in 2000, the x64 architecture has since been adopted by Intel and VIA, signaling a unified industry shift towards 64-bit computing. This wide adoption marked the effective obsolescence of the Intel Itanium architecture (IA-64), despite its initial design to supersede the x86 architecture.

[†]It is worth noting that the concepts we discuss here are not specific to C and C++. Other systems programming languages like D and Rust have converged on similar models.

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

```
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*

It is impractical for CPU hardware to track load-linked addresses for each byte within a system due to the immense resource requirements. To mitigate this, many processors monitor these operations at a broader scale, like the cache line level. Consequently, a SC operation may fail if any part of the monitored block is written to, not just the specific address that was load-linked.
This limitation poses a particular challenge for operations like compare and swap, highlighting the essential purpose of `compare_exchange_weak`. Consider, for example, the task of atomically multiplying a value without an architecture-specific atomic read-multiply-write instruction.

```
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 is not 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 do not care about the difference between retries from spurious SC failures and retries caused by another thread modifying our variable.

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

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 have 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()` 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 are not 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 is vital that reads and writes to shared memory do not move outside the critical section. But the opposite is not 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 various forms of RMW, are considered sequentially consistent. However, this is just one among many possible orderings. We will explore each of these orderings in detail. A comprehensive list, as well as the corresponding enumerations used by the C and C++ API, can be found here:

- 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 have slyly failed to mention so far:*

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

*In C, separate functions are defined for cases where specifying an ordering is necessary. `exchange()` becomes `exchange_explicit()`, a CAS becomes `compare_exchange_strong_explicit()`, and so on.

```
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 have seen so far do not actually need sequentially consistent operations.

10.1 *Acquire and release*

We have just examined the acquire and release operations in the context of the lock example from §9. You can think of them as "one-way" barriers: an acquire operation permits other reads and writes to move past it, but only in a *before* → *after* direction. A release works the opposite manner, allowing actions to move in an *after* → *before* direction. On ARM and other weakly-ordered architectures, this enables us to eliminate 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);
}
```

```
        acquireFoo:                          releaseFoo:
          ldr r3, <&foo>                       ldr r3, <&foo>
become:   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 is 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 useful for variables shared between threads where *no specific order* of operations is needed. Although it may seem like a niche requirement, such scenarios are quite common.
Refer back to our discussions on §3 and §5 operations, where a worker thread increments a counter that a UI thread then reads. In this case, the counter can be incremented using `fetch_add(1, memory_order_relaxed)` because the only requirement is atomicity; the counter itself does not coordinate synchronization.
Relaxed operations are also beneficial for managing flags shared between threads. For example, a thread might continuously run until it receives a signal 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 do not 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 as we do not need to enforce any order until we have successfully modified our value. The initial load of `expected` is not strictly necessary but can help avoid an extra loop iteration if `foo` remains unmodified by other threads before the CAS operation.

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 does not 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 are 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 introduce `memory_order_consume`. Imagine a situation where data changes rarely but is frequently read by many threads. For example, in a kernel tracking peripherals connected to a machine, updates to this information occur very infrequently—only when a device is plugged in or removed. In such cases, it is logical to prioritize read optimization as much as possible. Based on our current understanding, the most effective strategy 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);
}
```

*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 are almost always pessimal compared to a combined load-acquire or store-release.

To further enhance optimization for readers, bypassing a memory barrier on weakly-ordered systems for loads would be ideal. Fortunately, this is often achievable. The data being accessed (`p->keyboards`) relies on the value of `p`, leading most platforms, including those with weak ordering, to maintain the sequence of the initial load (`p = peripherals`) and its subsequent use (`p->keyboards`). However, it is notable that on some particularly weakly-ordered architectures, like DEC Alpha, this reordering can occur, much to the frustration of developers. Ensuring the compiler avoids any similar reordering is crucial, and `memory_order_consume` is designed for this purpose. 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 is not as trivial as one might hope,* so all compilers currently convert consume operations to acquires.

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 is not 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 have 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*

Given all the complexities to consider, modern hardware adds another layer to the puzzle. Remember, memory moves between main RAM and the CPU in segments known as cache lines. These lines also represent the smallest unit of data transferred between cores and their caches. When one core writes a value and another reads it, the entire cache line containing that value must be transferred from the first core's cache(s) to the second, ensuring a coherent "view" of memory across cores.

*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.

This dynamic can significantly affect performance. Take a readers-writer lock, for example, which prevents data races by allowing either a single writer or multiple readers access to shared data but not simultaneously. At its most basic, this concept can be summarized as follows:

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

Writers must wait until the `readers` count drops to zero, while readers can acquire the lock through an atomic RMW operation if `hasWriter` is `false`.
At first glance, this approach might seem significantly more efficient than exclusive locking mechanisms (e.g., mutexes or spinlocks) in scenarios where shared data is read more frequently than written. However, this perspective overlooks the impact of cache coherence. If multiple readers on different cores attempt to acquire the lock simultaneously, the cache line containing the lock will constantly be transferred among the caches of those cores. Unless the critical sections are considerably lengthy, the time spent managing this cache line movement could exceed the time spent within the critical sections themselves,* despite the algorithm's non-blocking nature.
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 are 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 often considered a *dead store*, seemingly performing no function. However, when `t` is directed at an MMIO register, this assumption becomes unsafe. In such cases, each write operation could potentially influence the behavior of the associated hardware.

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

These rules fall short of providing the atomicity and order required for safe communication between threads. It is important to note that the second rule only prevents `volatile` operations from being reordered in relation to one

*This situation underlines how some systems may experience a cache miss that is substantially more costly than an atomic RMW operation, as discussed in Paul E. McKenney's talk from CppCon 2017 for a deeper exploration.

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

another. The compiler remains at liberty to reorganize all other "normal" loads and stores around them. Furthermore, even setting this issue aside, `volatile` does not generate memory barriers on hardware with weak ordering. The effectiveness of the keyword as a synchronization tool hinges on both the compiler and the hardware avoiding any reordering, which is not a reliable expectation.

14 *Atomic fusion*

Finally, one should realize that while atomic operations do prevent certain optimizations, they are not 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 have 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 `rwonce.h` for details.
‡See the LWN 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++ 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 are available on GitHub. This paper is prepared in LaTeX.