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 2
2. Enforcing law and order 3
3. Atomicity 3
4. Arbitrarily-sized "atomic" types 3
5. Read-modify-write 4
   5.1. Exchange 4
   5.2. Test and set 4
   5.3. Fetch and… 4
   5.4. Compare and swap 4
6. Atomic operations as building blocks 5
7. Sequential consistency on weakly-ordered hardware 5
8. Implementing atomic read-modify-write operations with LL/SC instructions . . 6
   8.1. Spurious LL/SC failures 6
9. Do we always need sequentially consistent operations? 6
10. Memory orderings 7
   10.1. Acquire and release 7
   10.2. Relaxed 8
   10.3. Acquire-Release 8
   10.4. Consume 9
   10.5. HC SVNT DRACONES 10
11. Hardware convergence 10
12. Cache effects and false sharing 10
13. If concurrency is the question, `volatile` is not the answer. 10
14. Atomic fusion 11
15. Takeaways 11
Additional Resources 12
Contributing 12
Colophon 12

Lock's mechanisms

- Mutex
- Semaphores
- Conditional Variables
- Spinlocks
- Consumer-Producer Problem

Hardware: * Cache Effects : v_ready is in Cache but not v
* Instruction Complexity: v_ready = True might translate to short and faster to execute.
* Store Buffer: if buffer has Space for vready == True, it'll go ahead and do it.
* Pipe line.

if CPU is stalled waiting for something related to v, but vready is already ready, it will prioritize vready

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.[‡]

Compiler, might think setting vread = True is more efficient to make this code faster, than writing 42 to variable v.

Compiler.

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.

Instruction Fetch
Instruction Decode
Execute
Memory Access
Write Back.

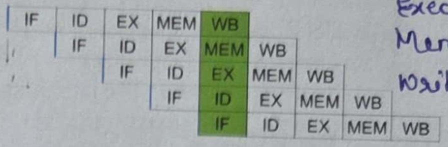

Figure 1: A traditional five-stage CPU pipeline with fetch, decode, execute, memory access, and write-back stages. Modern designs are much more complicated, often reordering instructions on the fly. Image courtesy of Wikipedia.

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.

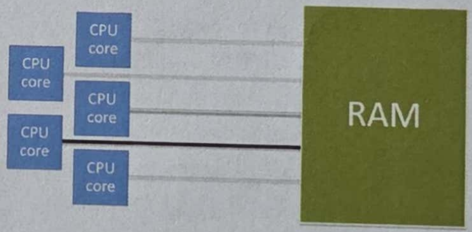

Figure 2: An idealized multi-core processor where cores take turns accessing a single shared set of memory.

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.

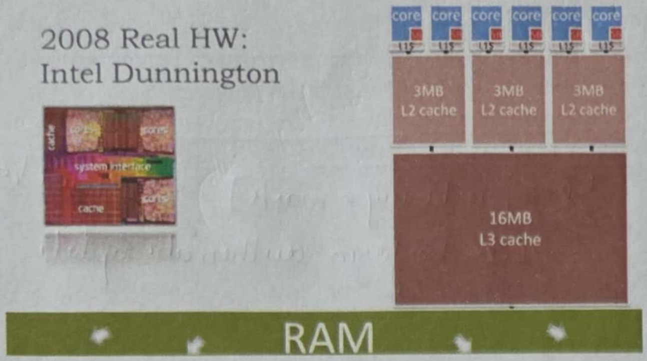

Figure 3: A common memory hierarchy for modern multi-core processors

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*.

a point where all thread meet

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.

```
int v = 0
_Atomic _Bool v_ready = 0;
void threadA() {              void threadB() {
    v = 42;                        while (!v_ready) { }
    v_ready = 1;                   int my_v = v;
}
```

NOTE: _Atomic ⇒ C11 (2011)
_Bool ⇒ C99 (1999)

T₁ starts writing new counter value (it first writes lower 32 bits then it goes ahead and write in upper 32 bits).

But T₂, read the counter value, before T₁ updates the upper 32 bit.

T₂, sees new lower 32 bit and old upper 32 bit, hence garbage value. This is called as Torn Reads & Writes.

```
#include <stdatomic.h>
atomic_int counter = ATOMIC_VAR_INIT(0)
void worker() { atomic_fetch_add(&counter, 1); }
```

```
void ui(){
    int procued = atomic_exchange(&counter, 0);
}
```

```
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* [Swap value automically]

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* (used to implement spinlocks)

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(); }
```

1. Read the current value
2. Sets it to TRUE
3. Return the previous Value.

Spinlok uses a flag to indicate wheather a resource is locked [TRUE] or free [FALSE].

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.

Spin [Busy - wait]

5.3. *Fetch and...*

Perform arithmatic update

Bitwise too.

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.

1. Worker thread loads [Counter=5], adds +1 [Counter=6]
2. UI thread sets [Counter=0] via exchange
3. Worker thread, writes 6, overwriting the UIthread's reset

5.4. *Compare and swap*

Enable conditional updates (ex. state transitions).

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.

For example, you might want to cancel a task only if its running, not if its idle. CAS lets you check the stat and update it automatically avoiding races where the state changes mid operation

*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.

```
#include <stdatomic.h>
typedef enum{
        IDLE;
        RUNNING,
        CANCELLED
}TaskState}

atomic_int ts = ATOMIC_VAR_INIT (IDLE);

void taskLoop(){
        atomic_store (&ts, RUNNING);
        while (atomic_load(&ts) == RUNNING){ }
}

bool cancel(){
        TaskState expected = RUNNING;
        return atomic_compare_exchange_strong (&ts,
                                                &expected,
                                                CANCELLED);
}
```

Explanation:

* taskLoop()

✓ automic_store (&ts, RUNNING) sets the state to RUNNI

✓ the loop continues while the state changes from RUNNING

* Cancel Function();

✓ If ts is RUNNING, sets it to CANCELLED and return True.

✓ If ts is not RUNNING, [eg: IDLE], update expected to current value and return False.

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 2
2. Enforcing law and order 3
3. Atomicity 3
4. Arbitrarily-sized "atomic" types 3
5. Read-modify-write 4
   5.1. Exchange 4
   5.2. Test and set 4
   5.3. Fetch and... 4
   5.4. Compare and swap 4
6. Atomic operations as building blocks 5
7. Sequential consistency on weakly-ordered hardware 5
8. Implementing atomic read-modify-write operations with LL/SC instructions . . 6
   8.1. Spurious LL/SC failures 6
9. Do we always need sequentially consistent operations? 6
10. Memory orderings 7
   10.1. Acquire and release 7
   10.2. Relaxed 8
   10.3. Acquire-Release 8
   10.4. Consume 9
   10.5. HC SVNT DRACONES 10
11. Hardware convergence 10
12. Cache effects and false sharing 10
13. If concurrency is the question, `volatile` is not the answer. 10
14. Atomic fusion 11
15. Takeaways 11
Additional Resources 12
Contributing 12
Colophon 12

- Mutex
- Semaphores
- Conditional Variables
- Spinlocks
- Consumer-Producer Problem

Locking mechanisms