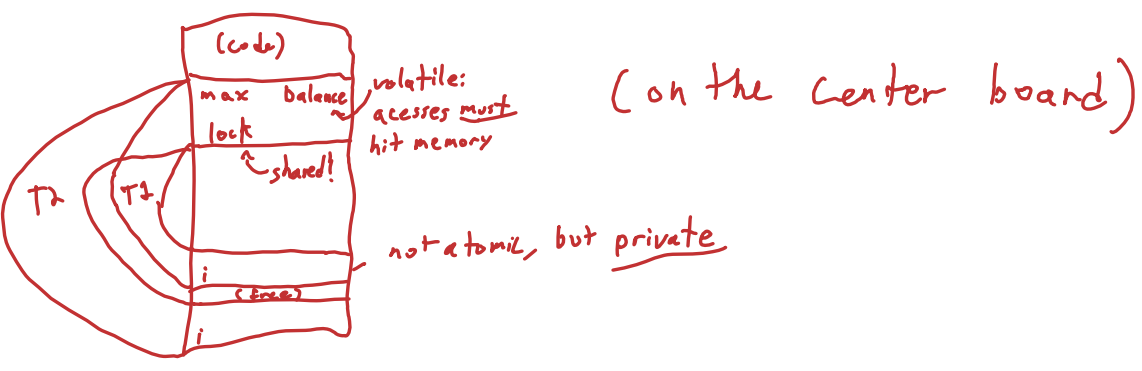
How locks work

Basic flow: from last time, we need to make sure that we know how the definitions work. So, write on board (left wing), main definitions:

- critical regions
- mutual exclusion
- atomic / atomicity
- race condition

These definitions are review from both 217-C and from Tuesday, so hopefully can be provided interactively.

Go back to the counter example, and once again show the critical regions (guard the increment). Talk about atomicity, and talk about what things are, or are not, atomic. Note: we will also want to draw out where the various variables are in the address space for this one:

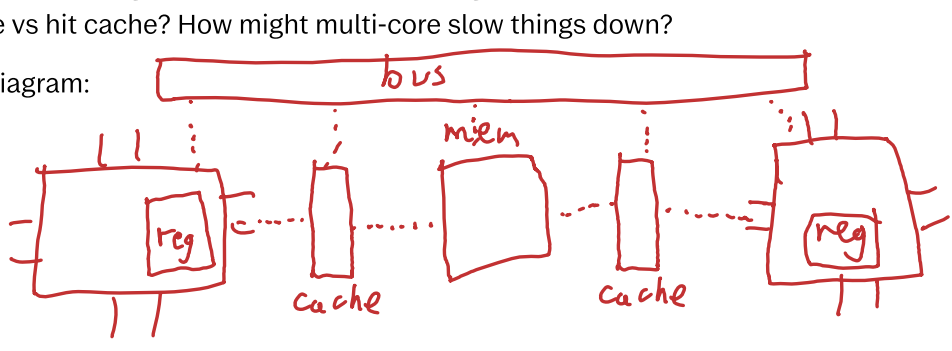


This can be interactive! Let's see if the class knows that individual instructions are atomic, by asking if there's an alternative way we could design hardware or an OS to prevent us from *needing* locks when doing this increment example. Also a good time to talk about privacy / private data never needs locks.

Walk through *exactly* what happens when a compiler interacts with the **volatile** keyword, and interacts with the reads/writes of various things in this example. In particular, focus on the compiler's choice to emit things in registers vs. memory. Talk about X86 and total store order (TSO), and why this matters for concurrency. We're getting into multicore now...

in particular (draw this on the board): we're going to have three things to consider: (1) does the compiler *emit the operation at all?* (2) does the compiler *put things in registers or memory?* (3) is the memory operation hitting cache, or hitting main storage? (4) in a multi-core setting, what happens when you hit storage vs hit cache? How might multi-core slow things down?

Draw diagram:



The pthreads lock API has to ensure locks define *atomic* regions. What are the considerations for this, in this diagram? (recall: *everything* that gets locked needs to be atomic).

How do we ensure that an individual write actually makes its way from one cache to the other? Answer: barriers. Hardware primtive that forces writes to become available everywhere. Can be implemented differently in different places, e.g. direct communication between caches vs. cache invalidations. No matter what, it is SLOOOW

(draw a log of writes to the cache, and then a "Barrier!", and then copy a bunch of stuff through the caches into memory)

In all of this, we're still using the counter example as our main thing. Keep it on the board, if you have space to do so.

Define: *consistency*. Given a write performed at a certain location, what are the *possible reads* that can be performed at other locations?

Talk about hadware differences: in X86 TSO, *simply emitting a write* ensures an amount of consistency on it, so you can get away with things being **volatile** there which you *can't* get away with in modern architectures (including the ones that most of us our running here).

Ok, now we know what's going on with this example. Let's talk about how to implement threads despite this

Last time: we talked about a *scheduler-first* policy for implementing threads. Draw same diagram on the board to remind the class.

This time: let's talk about a lot of different ways we can *combine OS and hardware help* to implement locks.

First: observe from before---if we just had an "atomic" fetch-and-add for this example, **and X86 TSO,** the *hardware itself* would have introduced atomicity! Powerful primitive. Note: even on hardware that *doesn't* talk about a total store order, there's usually an *explicitly atomic* fetch-and-add available to programmers. (So how might *that* have worked in hardware? We can imagine it, but (as an OS) we don't *really* have to know. Beyond the performance implications). It will fetch-and-add *and return the value that results*

Next: ignore the scheduler, and assume two cores running two threads actually-concurrently. Can we use the atomic-fetch-and-add to implement a lock? Genuinely, work through this one. With a partner. (sidebar: might want to try asking this for the other points of interaction earlier in this class). Note: you *can* do this, but you will need to invent a spin-lock to do so--hopefully folks can at least show you how to do the initial "decide who wins the race" part.

Note: we need to emphasize the *spin* part. What happens when someone *loses* the race for the lock??

Ok, there's actually different hardware primitives, test-and-set / compare-and-swap, that people use *in practice* to avoid the potential overflow and wasted writes that you'd get from this strategy.

On the board: here's how compare-and-swap works.

In code: here's the assembly for this in X86.

Note: these are all *spin locks;* they assume the threads are *always* scheduled, and so *always* need something to do. Is this bad? Not really *if there are at least as many cores as threads* (and you don't care about energy cost). (sidebar: the "mine crypto" lock. Not a real thing b/c hashing takes too long, but a funny thought).

Next: what can an OS provide *as an abstraction* to help with this?

- simple idea: just yield when you spin (not ideal---woken without purpose)
- SOLARIS idea: park / unpark(tid). *Actually write the thread implementation* using park/unpark, and hope they spot the race condition!
  - Recall: the thread library needs to have a queue for this, so maybe handwave how to lock the queue for management reasons? Or spinlock on the queue?
- More complex: the lock-management we've been talking about (but it's not fair! talk about fairness)
- What is happening in Linux: futexes.
  - futex_wait(addr,exp) vs futex_wake(addr)
  - this is very similar to the feature we described with the scheduler before, except there is an explicit queue now! Note: futex_wait takes an "expected" value for what's at the address---use it to solve the same problem as the "setpark" from Solaris (again, can interact for this!)
- Do we like any of these better than the others?

Note: we will need to *use precept time* to walk through data races and get students to think about them in more detail, probably.