Fairness in locking: the ticket lock

(show the ticket lock code)

Problem with our spin-locks from last time?

(1) they are wasteful --- but only a problem if you were planning on using the CPU for something else!

(2) they aren't *fair*; there's no way to guarantee a thread will *eventually* acquire a lock! This is called *starvation* and is a serious issue in concurrent programming. Let's solve (2) first, and then move on to (1).

Don't starve (together): a ticket lock

(project code)

In this lock system, our basic idea is to make it that threads acquire the lock in *roughly* the order that they enter a critical region. Last time, we talked about using an atomic fetch-and-add instruction to implement a spin-lock; that time, we used decrement to "release" the lock. But... what if we used *increment* to do that instead?

Basic idea: when a thead enters a critical region, it acquires a "ticket number" via an atomic increment. This is its "ticket number" and is the order in which the thread will (eventually) execute. It then spins on reading the current ticket value, waiting for its turn. Note: one thread will have gotten "0" for a ticket value, so it immediately takes its turn. When the turn is over, the exiting thread then *increments* the turn variable, et voila! Someone spinning on the turn sees it increment, and for one lucky thread that means its turn is now!

Let's kick the tires on this.

- What if no thread is waiting?
- Are we sure the initial thread will actually start, or are we off-by-one?
- Why do we need an *atomic* increment on unlock?

While we're here: note that we have included a new library, stdatomic.h, in this code. This "standard atomics" library is a new(ish) addition to C, and replaces the careful reasoning we needed to have before, about barriers and volatile

(redraw the communicating processes picture from last time, remind them about barriers)

write
barrier
c
c
m
c

You can use it! In x86 it mostly just casts things to volatile. (You can now answer why we needed the increment.

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!
  - note: we're going to do this live, so they can see how bad at C you are.
  - 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?
  - Re: the queue.. maybe just a fixed-size array to avoid building a queue live. We can always build in realloc if there are extra threads.
  - **do not move on** *before you get to setpark*
- More complex: the lock-management we've been talking about (but it's not fair! talk about fairness)
  - by which I mean lock groups in the kernel
- What is happening in Linux: futexes.
  - futex_wait(addr,exp) vs futex_wake(addr) (wait *unless* addr=exp. Interactive: why do we need this?)
  - 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?

Next: condition variables, reader/writer locks, and more joy

We're going to do this one live, but I don't have the pthreads library memorized, so here goes:

pthread_cond_t = PTHREAD_COND_INITIALIZER

pthread_cond_wait(cv, mutex) //remember: releases the lock.

pthread_cond_signal(cv)

The point here: if one thead needs to do something, and another thread needs to wait for the first thread to finish doing that thing, then we need a notification framework!!

Talk about the spurious-wakeup thing. Imagine with the class how we can re-use the basics from futex or park/unpark to implement condition variables.