Students: this lecture involved looking at the details of a lot of code. Please see the slides for those implementations (yield(), wait(), notify(), yield_wait())

0. Intro
   - Today: get rid of assumption that we only have one program per CPU.
   - Sharing CPU is a problem because one program can block another

1. Threads
   - thread = virtual processor
   - need to capture program's state: value of all registers, all of its memory
   - Big question: when to suspend/resume a thread?

2. yield()
   - command to tell kernel that thread is waiting for an event
   - implementation does three things: suspends running thread, chooses new thread to run, resumes new thread
     - data structures: threads table, CPUs table, t_lock
     - suspending current thread: save stack pointer and page-table register
     - choosing a new thread: round-robin fashion until we hit a RUNNABLE thread (perhaps the one that just called yield)
     - resuming new thread: reload state
     - all of this happens as an atomic action

3. Condition variables
   - allow kernel to notify threads instead of having threads constantly make checks
   - "lost notify" problem
     - T1 has lock on buffer, finds it full, releases lock
     - Prior to T1 calling wait, T2 acquires lock, reads message, notifies waiting threads that the buffer is not full
     - ...but T1 is not yet waiting; it was interrupted before it could call wait
     - solution: API is wait(cv, lock), not wait(cv).
     - when a thread calls wait, it goes to sleep and releases the lock
   - wait implementation
     - requires a different version of yield() -- yield_wait() -- to prevent deadlock
     - yield_wait() releases and re-acquires t_lock in the middle, and must point to a special stack to prevent stack corruption.

4. Preemption
   - If a thread never calls yield or wait, it's okay; special hardware
will periodically generate an interrupt and forcibly call yield
- ..But what if this interrupt occurs while the CPU is running
  yield()? Deadlock.
- Solution: hardware mechanism to disable interrupts.

5. Thread Scheduling
- A lot more complex in practice than the while loop we use in
  lecture!
- Affects performance and fairness. Fairness is very complicated
  to define (we haven't even tried yet, in this class)
- There is no single best scheduling algorithm
  - We'll return to scheduling when we talk about network queues

6. Reflection/Summary
- we've enforced modularity on a single machine, assuming that the
  OS itself is indeed correct
- locks and threads are interesting: we needed them to get bounded
  buffers to work, but they bring up modularity issues. We had to
  reason globally about locks.
- to truly enforce modularity, we needed kernel and/or hardware
  support