Previously
- We're on a quest to enforce modularity on a single machine
- Last time: virtualize memory to prevent programs from accessing each other's memory
- This time: virtualize communication links to allow programs to communicate
- Still assuming one program per CPU, and a correct kernel

1. Bounded Buffers
   - Allow programs to communicate
   - Another application of virtualization
   - Stores N messages, to deal with bursts
   - API: send(m), m <- receive()
   - Receivers and senders block if there are no messages (receiver) or no space (sender)
   - Concurrency causes problems in the implementation
   - Need to decide when it's okay to write, when it's okay to read, and where to write to/read from

2. Bounded buffers for single senders
   - Implementation is relatively short (see slides)
   - Can't swap the action and the increment; can cause reads of messages that don't exist

3. Bounded buffer for multiple senders
   - With two senders, different orders of executions will lead to unexpected output in the previous implementation (empty slots in the buffer, too few elements in the buffer)
   - Need locks

4. Locks
   - Allow only one CPU to be in a piece of code at a time
   - API: acquire(lock), release(lock)
     - *Not* acquire(variable I want to lock)
   - If two CPUs try to acquire the same lock at the same time, one will succeed and the other will block.

5. Bounded buffers with locks
   - Getting locks correct is tricky (see slides). Problems include:
     - Don't want to interrupt between Insert/read and increment of in/out
     - Need to hold a lock during the check for whether buffer has space/message, but don't want to block other programs from sending/receiving (if a sender blocks all receivers, we'll never make progress)
     - Even in correct solution, performance is a concern (more next
6. Atomic actions
   - How to decide what should make up an atomic action?
     - too much code in locks: performance suffers
     - too little code in locks: unexpected behavior
   - Think of locks as protecting an invariant. Don't release the lock when the invariant is false.

7. Example: Locks for file systems
   - Filesystem move:
     - Coarse-grained locking: Bad performance: can't move two different files between entirely different directories at the same time.
     - Fine-grained locking: Better performance, but harder to get correct
     - Issues arise if we're trying to move file1.txt from A to B, and file2.txt from B to A, for instance
   - Correct implementation is a little painful: requires global reasoning about all locks
   - Answer? start coarse-grained and refine

8. Implementing locks
   - Problem: need locks to implement locks (see slides)
   - Solution: hardware support (atomic instructions)
   - x86 example: XCHG
     - XCHG reg, addr
       temp <- mem[addr]
       mem[addr] <- reg
       reg <- temp
     - Now:
       - acquire(lock):
         do:
           r <= 1
           XCHG r, lock
         while r == 1

         How does this work? Suppose lock = 0 (i.e., no one else is holding lock). After the XCHG instruction, r=0 and lock=1, and the loop will terminate.

         On the other hand, suppose lock = 1 initially (i.e., someone else has lock). After the XCHG instruction, r=1 and lock=1, and the loop will not terminate.

         What's good here is that we've guaranteed that there is no interruption between setting lock=0 and r=1.
This is all made possible by a controller that manages access to memory.