6.033 - Bounded Buffers + Concurrency + Locks
Lecture 4
Katrina LaCurts, lacurts@mit.edu

0. 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
   - send(bb, message):

```
    while True: # Wait until it's okay to write
      if bb.in - bb.out < N:
        bb.buf[bb.in mod N] <- message
        bb.in <- bb.in + 1
        return
```

   - receive(bb):

```
    while True: # Wait until it's okay to read
      if bb.out < bb.in:
        message <- bb.buf[bb.out mod N]
        bb.out <- bb.out + 1
        return message
```

   - 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
  - Attempt 1 (using pseudocode): locks around every line
    - send(int x)

```
{
  acquire(&lck);
  buf[in] = x;
  release(&lck);
  acquire(&lck);
  in = in + 1;
  release(&lck);
}
```

    - Result: correct number of elements, but some slots have no messages (A and B write to same slot, and both increment)

  - Attempt 2:
    - send(int x)

```
{
  acquire(&lck);
  buf[in] = x;
  in = in + 1;
  release(&lck);
}
```

    - Correct: we want write and increment to be atomic (happen together)

  - Back to original code. Attempt 1:
    - send(bb, message):

```
while True:
  if bb.in - bb.out < N:
    acquire(bb.send_lock)
    bb.buf[bb.in mod N] <- message
    bb.in <- bb.in + 1
    release(bb.send_lock)
    return
```

    - No: concurrent senders will both think they can write, the first to acquire the lock might fill up the buffer (and so the second shouldn't write)

  - Attempt 2:
    - send(bb, message):

```
  acquire(bb.send_lock)
  while True:
    if bb.in - bb.out < N:
      bb.buf[bb.in mod N] <- message
```

```
        bb.in <- bb.in + 1
         release(bb.send_lock)
        return
```

- If the receiver is also trying to acquire send_lock, this attempt will prevent the receiver from ever receiving (so the sender will keep blocking when the buffer is full). If the receiver is using a different lock -- say receive_lock -- we will face issues with concurrently editing the same data structure.

- Attempt 3 (correct):
  - send(bb, message):

```
    acquire(bb.lock)
    while bb.in - bb.out = N:
      release(bb.lock) // repeatedly release and acquire, to allow
      acquire(bb.lock) // processes calling receive() to jump in
    bb.buf[bb.in mod N] <- message
    bb.in <- bb.in + 1
    release(bb.lock)
    return
```

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:
    - move(dir1, dir2, filename):

```
    unlink(dir1, filename)
    link(dir2, filename)
```

  - Coarse-grained locking:
    - move(dir1, dir2, filename):

```
    acquire(fs_lock)
    unlink(dir1, filename)
    link(dir2, filename)
    release(fs_lock)
```

    - Bad performance: can't move two different files between entirely different directories at the same time.

  - Fine-grained locking:
    - move(dir1, dir2, filename):

```
    acquire(dir1.lock)
    unlink(dir1, filename)
    release(dir1.lock)
    acquire(dir2.lock)
```

```
      link(dir2, filename)
      release(dir2.lock)
    - Better performance, but incorrect.  What if dir2 is renamed
      between release and acquire?
    - Bad because CPU sees inconsistent state

  - Fine-grained locking + holding both locks
    - move(dir1, dir2, filename):
        acquire(dir1.lock)
        acquire(dir2.lock)
        unlink(dir1, filename)
        link(dir2, filename)
        release(dir1.lock)
        release(dir2.lock)
    - Deadlock when A does move(M, N, file1.txt), B does move(N, M,
      file2.txt)

  - Fine-grained locking + solving deadlock
    - Heuristic: Look for all places where multiple locks are held,
      and ensure that locks are acquired in the same order
    - move(dir1, dir2, filename):
        if dir1.inum < dir2.inum:
          acquire(dir1.lock)
          acquire(dir2.lock)
        else:
          acquire(dir2.lock)
          acquire(dir1.lock)
        unlink(dir1, filename)
        link(dir2, filename)
        release(dir1.lock)
        release(dir2.lock)
    - Painful: requires global reasoning about all locks

  - Answer? start coarse-grained and refine

8. Implementing locks
  - Attempt 1:
    - acquire(lock):
        while lock != 0:
          do nothing
        lock = 1
    - release(lock):
        lock = 0
    - Race condition: both see lock = 0, set lock = 1, and execute
      code
  - Problem: need locks to implement locks
  - 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
```
- Atomic operations made possible by the controller that manages access to memory