6.033 Spring 2015Lecture #5

- Bounded Buffers
- Concurrency
- Locks

Enforcing Modularity via Virtualization

in order to enforce modularity + build an effective operating system

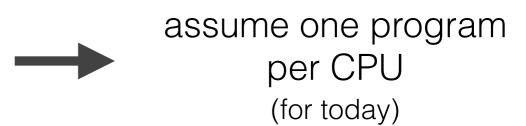
 programs shouldn't be able to refer to (and corrupt) each others' memory



programs should be able to communicate



3. programs should be able to **share a CPU** without one program halting the progress of the others



today's goal: implement bounded buffers so that programs can communicate

bounded buffer: a buffer that stores (up to) N messages

```
bounded buffer API:
    send(m)
    m <- receive()</pre>
```

```
send(bb, message):
   while True:
     if bb.in - bb.out < N:
        bb.buf[bb.in mod N] <- message</pre>
        bb.in <- bb.in + 1
        return
                              incorrect if we swap
                               these statements!
receive(bb):
   while True:
     if bb.out < bb.in:
        message <- bb.buf[bb.out mod N]</pre>
        bb.out <-bb.out + 1
        return message
```

Katrina LaCurts | lacurts@mit | 6.033 2015

locks: allow only one CPU to be inside a piece of code at a time

```
lock API:
   acquire(1)
   release(1)
```

example output:

```
101 102 103 1 2 3
101 102 1 0 2 3
1 102 103 0 2 3
1 2 3
```

correct!
empty spots in buffer
too few elements in buffer

```
int buf[6];
int in = 0;
                                       cpu_two()
                        cpu_one()
struct lock lck;
                           send(1);
                                         send(101);
send(int x)
                           send(2);
                                         send(102);
                           send(3);
                                         send(103);
  acquire(&lck);
  buf[in] = x;
  release(&lck);
  acquire(&lck);
  in = in + 1;
  release(&lck);
                                  example output:
                          correct! 101 102 103 1 2 3
                                  1 0 2 0 3 0
                                  101 1 0 2 0 3
               empty spots in buffer
                                  101 1 103 2 0 3
```

```
int buf[6];
int in = 0;
struct lock lck;
send(int x)
  acquire(&lck);
  buf[in] = x;
  in = in + 1;
  release(&lck);
```

```
cpu_two()
cpu_one()
  send(1);
                send(101);
  send(2);
                send(102);
  send(3);
                send(103);
```

example output:

```
correct! 101 1 102 2 103 3
       101 102 1 103 2 3
       1 101 2 102 3 103
       101 102 1 103 2 3
```

```
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</pre>
```

won't work! second sender could end up writing to full buffer

```
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</pre>
```

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

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

problem: poor performance

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

problem: inconsistent state

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

problem: deadlock

```
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)
```

could release dir1's lock here instead (nice job student who pointed that out!)

Implementing Locks

```
acquire(lock):
    while lock != 0:
        lock = 0
        lock = 1
```

problem: race condition
(need locks to implement locks!)

Implementing Locks

Bounded buffers

Bounded buffers allow programs to communicate, completing the second step of enforcing modularity on a single machine. They are tricky to implement due to **concurrency**.

Locks

Allow us to implement **atomic actions**. Determining the correct locking discipline is tough thanks to race conditions, deadlock, and performance.