

6.033 Spring 2018

Lecture #4

- **Bounded Buffers**
- **Concurrency**
- **Locks**

operating systems enforce modularity on a single machine using **virtualization**

in order to enforce modularity + build an effective operating system

1. programs shouldn't be able to refer to (and corrupt) each others' **memory** → **virtual memory**
2. programs should be able to **communicate** → assume that they don't need to
3. programs should be able to **share a CPU** without one program halting the progress of the others → assume one program per CPU

operating systems enforce modularity on a single machine using **virtualization**

in order to enforce modularity + build an effective operating system

1. programs shouldn't be able to refer to (and corrupt) each others' **memory** → **virtual memory**
2. programs should be able to **communicate** → **bounded buffers**
(virtualize communication links)
3. programs should be able to **share a CPU** without one program halting the progress of the others → assume one program per CPU
(for today)

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()`

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

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

lock API:

`acquire(1)`

`release(1)`

```

int buf[6];
int in = 0;
struct lock lck;

send(int x)
{
    buf[in%6] = x;
    in = in + 1;
}

```

```

cpu_one()
{
    send(1);
    send(2);
    send(3);
}

```

```

cpu_two()
{
    send(101);
    send(102);
    send(103);
}

```

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;
struct lock lck;

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

```

```

cpu_one()
{
    send(1);
    send(2);
    send(3);
}

```

```

cpu_two()
{
    send(101);
    send(102);
    send(103);
}

```

correct!

empty spots in buffer

example output:

101	102	103	1	2	3
1	0	2	0	3	0
101	1	0	2	0	3
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_one()
{
    send(1);
    send(2);
    send(3);
}

cpu_two()
{
    send(101);
    send(102);
    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.lock)  
      bb.buf[bb.in mod N] <- message  
      bb.in <- bb.in + 1  
      release(bb.lock)  
  return
```

problem: second sender could end up writing to full buffer

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

problem: deadlock if buffer is full
(receive needs to acquire `bb.lock` to make space in buffer)

```
send(bb, message):  
  acquire(bb.lock)  
  while bb.in - bb.out == N:  
    release(bb.lock)  
    acquire(bb.lock)  
  bb.buf[bb.in mod N] <- message  
  bb.in <- bb.in + 1  
  release(bb.lock)  
  return
```

← give up the lock to allow receivers to access the buffer

Filesystem move

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

Filesystem move

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

problem: poor performance

Filesystem move

```
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 is exposed


Filesystem move

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

problem: deadlock

Filesystem move

```
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

Implementing Locks

```
acquire(lock):  
    while lock != 0:  
        do nothing  
    lock = 1
```

```
release(lock):  
    lock = 0
```

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

Implementing Locks

```
acquire(lock):  
  do:  
    r <- 1  
    XCHG r, lock  
  while r == 1
```

```
release(lock):  
  lock = 0
```

- **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 issues.