6.033 Spring 2016
Lecture #4

- Bounded Buffers
- Concurrency
- Locks
Enforcing Modularity via 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

2. 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:**

```plaintext
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

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

incorrect if we swap these statements!
**locks**: allow only one CPU to be inside a piece of code at a time

**lock API**:

- `acquire(l)`
- `release(l)`
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  
correct!
101 102 1 0 2 3  
empty spots in buffer
1 102 103 0 2 3  
too few elements in buffer
1 2 3
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);
}

example output:
correct! 101 102 103 1 2 3
1 0 2 0 3 0
empty spots in buffer
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.send_lock)
            bb.buf[bb.in % N] <- message
            bb.in <- bb.in + 1
            release(bb.send_lock)
    return

**problem:** 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

problem: deadlock if receive is trying to acquire same lock
(race conditions if receive is trying to acquire a different lock)
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
Filesystem move

```
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
**Filesystem move**

```plaintext
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(\texttt{dir1}, \texttt{dir2}, \texttt{filename}): 
\begin{verbatim}
acquire(\texttt{dir1.lock})
acquire(\texttt{dir2.lock})
unlink(\texttt{dir1, filename})
link(\texttt{dir2, filename})
release(\texttt{dir1.lock})
release(\texttt{dir2.lock})
\end{verbatim}

\textbf{problem:} deadlock
Filesystem move

```python
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(\text{lock}):
\begin{align*}
do: \\
&\text{\texttt{r} <- 1} \\
&\text{\texttt{XCHG r, lock}} \\
&\text{while \texttt{r == 1}}
\end{align*}

release(\text{lock}):
\begin{align*}
&\text{\texttt{lock} = 0}
\end{align*}
• **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.