6.033 Spring 2017
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
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
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!
1:    send(bb, message):
2:        while True:
3:            if bb.in - bb.out < N:
4:                bb.buf[bb.in mod N] <- message
5:                bb.in <- bb.in + 1
6:        return
**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:

<table>
<thead>
<tr>
<th>101</th>
<th>102</th>
<th>103</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>102</td>
<td>103</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>102</td>
<td>103</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

correct!
empty spots in buffer
too few elements in buffer
```c
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!

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>102</td>
<td>103</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>
empty spots in buffer

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>103</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

6.033 | spring 2017 | lacurts@mit.edu
```c
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
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
Filesistem 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
```python
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

\[
\text{move}(\text{dir1}, \text{dir2}, \text{filename}): \\
\text{if } \text{dir1.inum} < \text{dir2.inum}: \\
\quad \text{acquire(dir1.lock)} \\
\quad \text{acquire(dir2.lock)} \\
\text{else:} \\
\quad \text{acquire(dir2.lock)} \\
\quad \text{acquire(dir1.lock)} \\
\quad \text{unlink(dir1, filename)} \\
\quad \text{link(dir2, filename)} \\
\quad \text{release(dir1.lock)} \\
\quad \text{release(dir2.lock)}
\]

could release \text{dir1}’s lock here instead
Implementing Locks

acquire(\textit{lock}): 
\begin{align*}
\text{while } \text{lock} & \neq 0: \\
\text{do nothing} \\
\text{lock} &= 1
\end{align*}

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

\textbf{problem:} race condition \\
(need locks to implement locks!)
Implementing Locks

**acquire**(*lock*):

```plaintext
do:
    \( r \leftarrow 1 \)
    \text{XCHG } r, \text{ lock}
while \( r == 1 \)
```

**release**(*lock*):

```plaintext
\text{lock} = 0
```
• **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.