6.033 Spring 2015

Lecture #5

- 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

   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**:

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

example output:
correct!

|   |   |
|---|---|---|---|---|---|---|
| 101| 102| 103| 1 | 2 | 3 |
| 1  | 0  | 2  | 0 | 3 | 0 |
| 101| 1  | 0  | 2 | 0 | 3 |
| 101| 1  | 103| 2 | 0 | 3 |

empty spots in buffer
```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.send_lock)
            bb.buf[bb.in mod N] <- message
            bb.in <- bb.in + 1
            release(bb.send_lock)
        return

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

\[
\text{move}(\text{dir1}, \text{dir2}, \text{filename}): \\
\text{acquire}(\text{fs\_lock}) \\
\text{unlink}(\text{dir1}, \text{filename}) \\
\text{link}(\text{dir2}, \text{filename}) \\
\text{release}(\text{fs\_lock})
\]

**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
Filesystem move

\[
\text{move}(\text{dir1, dir2, filename}): \\
\hspace{1cm} \text{acquire(dir1.lock)} \\
\hspace{1cm} \text{acquire(dir2.lock)} \\
\hspace{1cm} \text{unlink(dir1, filename)} \\
\hspace{1cm} \text{link(dir2, filename)} \\
\hspace{1cm} \text{release(dir1.lock)} \\
\hspace{1cm} \text{release(dir2.lock)}
\]

\textbf{problem:} deadlock
Filesystem move

```python
def 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!)

Katrina LaCurts | lacurts@mit | 6.033 2015
Implementing Locks

acquire(\texttt{lock}): 
\begin{verbatim}
while \texttt{lock} \neq 0:
  do nothing
\texttt{lock} = 1
\end{verbatim}

release(\texttt{lock}): 
\begin{verbatim}
\texttt{lock} = 0
\end{verbatim}

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

acquire(\texttt{lock}): 
\begin{align*}
\text{do:} & \\
& \text{\quad \texttt{r} \gets 1} \\
& \text{\quad \text{XCHG} \: \texttt{r}, \: \texttt{lock}} \\
& \text{while \: \texttt{r} \: \texttt{== 1}}
\end{align*}

release(\texttt{lock}): 
\begin{align*}
\text{\quad \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.