Lecture #4: Bounded Buffers + Locks

getting many programs to communicate at once
operating systems enforce modularity on a single machine using virtualization. In order to enforce modularity and have an effective operating system, a few things need to happen:

1. Programs shouldn’t be able to refer to (and corrupt) each others’ memory. 
   - virtual memory

2. Programs should be able to communicate with each other.
   - assume they don’t need to (for today)

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

in order to enforce modularity + have an effective operating system, a few things need to happen

1. programs shouldn’t be able to refer to (and corrupt) each others’ **memory**

2. programs should be able to **communicate** with each other

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. Programs can send and receive messages via this buffer.
**bounded buffer**: a buffer that stores (up to) \( N \) messages. Programs can **send** and **receive** messages via this buffer.

// send a message by placing it in bb

```
send(bb, message):
    while True:
        if bb.in - bb.out < N:
            bb.buf[bb.in mod N] <- message
            bb.in <- bb.in + 1
    return
```
**bounded buffer**: a buffer that stores (up to) \(N\) messages. Programs can **send** and **receive** messages via this buffer.

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

**variables in use**
- \(bb\) = the bounded buffer
- \(message\) = the message we’re trying to send/receive
- \(bb.in\) = total number of messages sent via this buffer
- \(bb.out\) = total number of messages received via this buffer
**bounded buffer**: a buffer that stores (up to) \( N \) messages. Programs can **send** and **receive** messages via this buffer.

```plaintext
// send a message by placing it in bb
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 a message from bb
receive(bb):
    while True:
        if bb.out < bb.in:
            message <- bb.buf[bb.out mod N]
            bb.out <- bb.out + 1
        return message
```

**variables in use**
- \( bb \) = the bounded buffer
- \( message \) = the message we’re trying to send/receive
- \( bb.in \) = total number of messages sent via this buffer
- \( bb.out \) = total number of messages received via this buffer
**bounded buffer:** a buffer that stores (up to) $N$ messages. Programs can **send** and **receive** messages via this buffer.

// send a message by placing it in bb
send(bb, message):
    while True:
        if bb.in - bb.out < N:
            bb.in <- bb.in + 1
            bb.buf[bb.in mod N] <- message
        return

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

**variables in use**

- **bb** = the bounded buffer
- **message** = the message we’re trying to send/receive
- **bb.in** = total number of messages sent via this buffer
- **bb.out** = total number of messages received via this buffer
**bounded buffer**: a buffer that stores (up to) N messages. Programs can **send** and **receive** messages via this buffer.

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

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

**variables in use**
- `bb` = the bounded buffer
- `message` = the message we’re trying to send/receive
- `bb.in` = total number of messages sent via this buffer
- `bb.out` = total number of messages received via this buffer
bounded buffer: a buffer that stores (up to) \(N\) messages. programs can send and receive messages via this buffer

// send a message by placing it in bb
send(bb, message):
    while True:
        if bb.in - bb.out < N:
            bb.in <- bb.in + 1
            bb.buf[bb.in-1 mod N] <- message
        return

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

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
**bounded buffer**: a buffer that stores (up to) $N$ messages. Programs can **send** and **receive** messages via this buffer.

```python
// send a message by placing it in bb
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 a message from bb
receive(bb):
  while True:
    if bb.out < bb.in:
      message <- bb.buf[bb.out mod N]
      bb.out <- bb.out + 1
    return message
```

**variables in use**

- **bb** = the bounded buffer
- **message** = the message we’re trying to send/receive
- **bb.in** = total number of messages sent via this buffer
- **bb.out** = total number of messages received via this buffer
what happens when multiple programs try to send?

// send a message by placing it in bb
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

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
what happens when multiple programs try to send?

// send a message by placing it in bb
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

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
what happens when multiple programs try to send?

broccoli is trying to send message \( m_1 \)

magnus is trying to send message \( m_2 \)

// send a message by placing it in bb
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

variables in use

- \( bb \) = the bounded buffer
- \( message \) = the message we’re trying to send/receive
- \( bb.in \) = total number of messages sent via this buffer
- \( bb.out \) = total number of messages received via this buffer
what happens when multiple programs try to send?

// send a message by placing it in bb
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

bb contains no messages
bb.in = 0
bb.out = 0
N is very large

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
what happens when multiple programs try to send?

variables in use

bb = the bounded buffer

message = the message we’re trying to send/receive

bb.in = total number of messages sent via this buffer

bb.out = total number of messages received via this buffer

Broccoli is trying to send message \( m_1 \)

```
// send a message by placing it in bb
-> 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

bb.buf is empty
bb.in = 0
bb.out = 0
N is very large
```

Magnus is trying to send message \( m_2 \)

Current line: 1
what happens when multiple programs try to send?

broccoli is trying to send message \( m_1 \)

```
// send a message by placing it in bb
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
```

magnus is trying to send message \( m_2 \)

current line: 2

current line: 1

variables in use

- \( bb \) = the bounded buffer
- \( message \) = the message we’re trying to send/receive
- \( bb.in \) = total number of messages sent via this buffer
- \( bb.out \) = total number of messages received via this buffer

\( N \) is very large

bb.buf is empty
bb.in = 0
bb.out = 0
what happens when multiple programs try to send?

broccoli is trying to send message $m_1$

```
// send a message by placing it in bb
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
```

current line: 3

variables in use

- $bb$ = the bounded buffer
- $message$ = the message we’re trying to send/receive
- $bb.in$ = total number of messages sent via this buffer
- $bb.out$ = total number of messages received via this buffer

magnus is trying to send message $m_2$

```
```
what happens when multiple programs try to send?

broccoli is trying to send message $m_1$

```
// send a message by placing it in bb
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
```

current line: 4

magnus is trying to send message $m_2$

```
bb.buf is empty
bb.in = \emptyset
bb.out = \emptyset
N is very large
```

current line: 1

variables in use

- $bb$ = the bounded buffer
- $message$ = the message we’re trying to send/receive
- $bb.in$ = total number of messages sent via this buffer
- $bb.out$ = total number of messages received via this buffer
what happens when multiple programs try to send?

variables in use

- \texttt{bb} = the bounded buffer
- \texttt{message} = the message we're trying to send/receive
- \texttt{bb.in} = total number of messages sent via this buffer
- \texttt{bb.out} = total number of messages received via this buffer


broccoli is trying to send message \( m_1 \)

```python
// send a message by placing it in \texttt{bb}
send(bb, message):
    while True:
        if bb.in - bb.out < N:
            bb.buf[bb.in mod N] <- message
            bb.in <- bb.in + 1
        return
```

\( \texttt{bb.buf[0]} = m_1 \)
\( \texttt{bb.in} = 0 \)
\( \texttt{bb.out} = 0 \)
\( N \) is very large

magnus is trying to send message \( m_2 \)

current line: 4
what happens when multiple programs try to send?

broccoli is trying to send message $m_1$

// send a message by placing it in bb
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

bb.buf[0] = $m_1$
bb.in = 0
bb.out = 0
N is very large

magnus is trying to send message $m_2$

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
what happens when multiple programs try to send?

// send a message by placing it in bb
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

bb.buf[0] = m₁
bb.in = 1
bb.out = 0
N is very large

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
what happens when multiple programs try to send?

broccoli is trying to send message $m_1$

magnus is trying to send message $m_2$

// send a message by placing it in bb
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

bb.buf[0] = $m_1$
bb.in = 1
bb.out = 0
N is very large

variables in use

$bb$ = the bounded buffer
$message$ = the message we’re trying to send/receive
$bb.in$ = total number of messages sent via this buffer
$bb.out$ = total number of messages received via this buffer
what happens when multiple programs try to send?

broccoli is trying to send message \( m_1 \)

magnus is trying to send message \( m_2 \)

// send a message by placing it in bb
1: \( \text{send}(bb, \text{message}) \):
2: \( \text{while True} \):
3: \( \text{if } bb.\text{in} - bb.\text{out} < N \):
4: \( bb.\text{buf}[bb.\text{in} \mod N] \leftarrow \text{message} \)
5: \( bb.\text{in} \leftarrow bb.\text{in} + 1 \)
6: \( \text{return} \)

\[ bb.\text{buf}[0] = m_1 \]
\[ bb.\text{in} = 1 \]
\[ bb.\text{out} = 0 \]
\( N \) is very large

variables in use

- \( bb \) = the bounded buffer
- \( \text{message} \) = the message we’re trying to send/receive
- \( bb.\text{in} \) = total number of messages sent via this buffer
- \( bb.\text{out} \) = total number of messages received via this buffer
what happens when multiple programs try to send?

broccoli is trying to send
message $m_1$

// send a message by placing it in bb
-> 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

bb.buf[0] = $m_1$
bb.in = 1
bb.out = 0
N is very large

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer

magnus is trying to send
message $m_2$
what happens when multiple programs try to send?

broccoli is trying to send
message $m_1$

// send a message by placing it in bb
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

bb.buf[0] = $m_1$
bb.in = 1
bb.out = 0

N is very large

variables in use

bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer

magnus is trying to send
message $m_2$

current line: 2
what happens when multiple programs try to send?

// send a message by placing it in bb
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

bb.buf[0] = m₁
bb.in = 1
bb.out = 0
N is very large

variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
what happens when multiple programs try to send?

broccoli is trying to send message $m_1$

magnus is trying to send message $m_2$

```python
// send a message by placing it in bb
send(bb, message):

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

bb.buf[0] = m_1
bb.in = 1
bb.out = 0
N is very large
```

variables in use

- $bb$ = the bounded buffer
- $message$ = the message we’re trying to send/receive
- $bb.in$ = total number of messages sent via this buffer
- $bb.out$ = total number of messages received via this buffer
what happens when multiple programs try to send?

broccoli is trying to send message $m_1$

```
// send a message by placing it in bb
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
```

magnus is trying to send message $m_2$

current line: 4

variables in use

-bb = the bounded buffer
-message = the message we’re trying to send/receive
-bb.in = total number of messages sent via this buffer
-bb.out = total number of messages received via this buffer
what happens when multiple programs try to send?

broccoli is trying to send message $m_1$

```
// send a message by placing it in bb
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
```

magnus is trying to send message $m_2$

```
variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
```

broccoli is trying to send message $m_1$

```
bb.buf[0] = m_1
bb.buf[1] = m_2
bb.in = 1
bb.out = 0
N is very large
```
what happens when multiple programs try to send?

broccoli is trying to send message $m_1$

magnus is trying to send message $m_2$

// send a message by placing it in bb
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

variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer

magnus is trying to send message $m_2$

complete
what happens when multiple programs try to send?

broccoli is trying to send message \( m_1 \)

magnus is trying to send message \( m_2 \)

// send a message by placing it in bb
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

variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer

\( N \) is very large

\( \rightarrow \) current line: 6

bb.buf[0] = \( m_1 \)
bb.buf[1] = \( m_2 \)
bb.in = 2
bb.out = 0

broccoli is trying to send complete
what happens when multiple programs try to send?

broccoli is trying to send message $m_1$

magnus is trying to send message $m_2$

// send a message by placing it in bb
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

$bb.buf[0] = m_1$
$bb.buf[1] = m_2$
$bb.in = 2$
$bb.out = 0$

$N$ is very large

variables in use

$bb$ = the bounded buffer
$message$ = the message we’re trying to send/receive
$bb.in$ = total number of messages sent via this buffer
$bb.out$ = total number of messages received via this buffer
what happens when multiple programs try to send?

broccoli is trying to send message $m_1$

```
// send a message by placing it in bb
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
```

bb contains no messages
bb.in = 0
bb.out = 0
N is very large

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer

magnus is trying to send message $m_2$

current line: 1
what happens when multiple programs try to send?

broccoli is trying to send message $m_1$

// send a message by placing it in bb
- 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

bb contains no messages
bb.in = 0
bb.out = 0
N is very large

current line: 2

magnus is trying to send message $m_2$

current line: 1

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
what happens when multiple programs try to send?

broccoli is trying to send message \( m_1 \)

magnus is trying to send message \( m_2 \)

```python
// send a message by placing it in bb
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

bb contains no messages
bb.in = 0
bb.out = 0
N is very large
```

variables in use
\( bb \) = the bounded buffer
\( message \) = the message we’re trying to send/receive
\( bb.in \) = total number of messages sent via this buffer
\( bb.out \) = total number of messages received via this buffer
what happens when multiple programs try to send?

broccoli is trying to send message $m_1$

```
// send a message by placing it in bb
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
```

current line: 3

magnus is trying to send message $m_2$

```
bb contains no messages
bb.in = 0
bb.out = 0
N is very large
```

current line: 2

variables in use

- $bb$ = the bounded buffer
- $message$ = the message we’re trying to send/receive
- $bb.in$ = total number of messages sent via this buffer
- $bb.out$ = total number of messages received via this buffer
what happens when multiple programs try to send?

// send a message by placing it in bb
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

bb contains no messages
bb.in = 0
bb.out = 0
N is very large

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
what happens when multiple programs try to send?

broccoli is trying to send message \( m_1 \)

```
// send a message by placing it in bb
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
```

current line: 4

magnus is trying to send message \( m_2 \)

```
bb contains no messages
bb.in = 0
bb.out = 0
N is very large
```

variables in use
- \( bb \) = the bounded buffer
- \( message \) = the message we’re trying to send/receive
- \( bb.in \) = total number of messages sent via this buffer
- \( bb.out \) = total number of messages received via this buffer
what happens when multiple programs try to send?

broccoli is trying to send message \( m_1 \)

```python
// send a message by placing it in bb
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
```

variables in use
- \( bb \) = the bounded buffer
- \( message \) = the message we're trying to send/receive
- \( bb.in \) = total number of messages sent via this buffer
- \( bb.out \) = total number of messages received via this buffer

magnus is trying to send message \( m_2 \)

current line: 4

broccoli contains no messages
- \( bb.in = 0 \)
- \( bb.out = 0 \)

\( N \) is very large
what happens when multiple programs try to send?

// send a message by placing it in bb
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

bb.buf[0] = m₁
bb.in = 0
bb.out = 0
N is very large

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
what happens when multiple programs try to send?

// send a message by placing it in bb
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

bb.buf[0] = m₁
bb.in = 0
bb.out = 0
N is very large

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
what happens when multiple programs try to send?

// send a message by placing it in bb
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

bb.buf[0] = m₂
bb.in = 0
bb.out = 0

N is very large

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer

broccoli is trying to send message \( m_1 \)

magnus is trying to send message \( m_2 \)
what happens when multiple programs try to send?

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer

broccoli is trying to send message \(m_1\)

magnus is trying to send message \(m_2\)

// send a message by placing it in bb
1: send(bb, message):
2: while True:
3: if bb.in - bb.out < N:
4: \(\text{bb.buf}[\text{bb.in mod N}] \leftarrow \text{message}\)
\(\Rightarrow\) 5: bb.in \leftarrow bb.in + 1
6: return

\(\text{bb.buf}[0] = m_2\)
\(\text{bb.in} = 0\)
\(\text{bb.out} = 0\)
\(N\) is very large
what happens when multiple programs try to send?

// send a message by placing it in bb
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

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer

broccoli is trying to send message \( m_1 \)
magnus is trying to send message \( m_2 \)

\( N \) is very large
what happens when multiple programs try to send?

broccoli is trying to send message \( m_1 \)

```
// send a message by placing it in bb
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

bb.buf[0] = m_2
bb.in = 1
bb.out = 0
```

variables in use

- \( bb \) = the bounded buffer
- \( message \) = the message we’re trying to send/receive
- \( bb.in \) = total number of messages sent via this buffer
- \( bb.out \) = total number of messages received via this buffer

magnus is trying to send message \( m_2 \)

current line: 6
current line: 5
what happens when multiple programs try to send?

// send a message by placing it in bb
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

bb.buf[0] = m2
bb.in = 2
bb.out = 0
N is very large

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer

broccoli is trying to send message m₁
magnus is trying to send message m₂
what happens when multiple programs try to send?

// send a message by placing it in bb
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

bb.buf[0] = m₂
bb.in = 2
bb.out = 0
N is very large

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
what happens when multiple programs try to send?

// send a message by placing it in bb
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

bb.buf[0] = m2
bb.in = 2
bb.out = 0

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer

magnus is trying to send message m2

broccoli is trying to send message m₁

current line: 6
what happens when multiple programs try to send?

broccoli is trying to send message $m_1$

magnus is trying to send message $m_2$

// send a message by placing it in bb
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

bb.buf[0] = $m_2$
bb.in = 2
bb.out = 0
N is very large

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
this implementation of send and receive only works with a single sender and receiver; it can introduce race conditions with multiple senders

// send a message by placing it in bb
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 a message from bb
receive(bb):
    while True:
        if bb.out < bb.in:
            message <- bb.buf[bb.out mod N]
            bb.out <- bb.out + 1
    return message

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
// send a message by placing it in bb

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

variables in use

bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
// send a message by placing it in bb

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

our earlier problem stemmed from the fact that a program could be interrupted after adding message to bb.buf, but before incrementing bb.in

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
// send a message by placing it in bb
send(bb, message):
    while True:
        if bb.in - bb.out < N:
            bb.buf[bb.in mod N] <- message
            bb.in <- bb.in + 1
        return

our earlier problem stemmed from the fact that a program could be interrupted after adding message to bb.buf, but before incrementing bb.in

(in fact, a program could be interrupted while incrementing bb.in; remember that bb.in <- bb.in + 1 is multiple lines in assembly)

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
locks: allow only one CPU to be inside a piece of code at a time.
programs can acquire and release a lock

// send a message by placing it in bb
send(bb, message):
    while True:
        if bb.in - bb.out < N:
            bb.buf[bb.in mod N] <- message
            bb.in <- bb.in + 1
        return

our earlier problem stemmed from the fact that a program could be interrupted after adding message to bb.buf, but before incrementing bb.in

(in fact, a program could be interrupted while incrementing bb.in; remember that bb.in <- bb.in + 1 is multiple lines in assembly)

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
**locks:** allow only one CPU to be inside a piece of code at a time. programs can **acquire** and **release** a lock

```
// send a message by placing it in bb
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
```

our earlier problem stemmed from the fact that a program could be interrupted after adding **message** to **bb.buf**, but before incrementing **bb.in**

variables in use
- **bb** = the bounded buffer
- **message** = the message we’re trying to send/receive
- **bb.in** = total number of messages sent via this buffer
- **bb.out** = total number of messages received via this buffer
- **bb.lock** = lock intended to protect the bounded buffer
locks: allow only one CPU to be inside a piece of code at a time. programs can acquire and release a lock

```python
// send a message by placing it in bb
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
```

our earlier problem stemmed from the fact that a program could be interrupted after adding message to bb.buf, but before incrementing bb.in

now, only one program can be “in” this section of the code at a time

variables in use
- `bb` = the bounded buffer
- `message` = the message we’re trying to send/receive
- `bb.in` = total number of messages sent via this buffer
- `bb.out` = total number of messages received via this buffer
- `bb.lock` = lock intended to protect the bounded buffer
**locks:** allow only one CPU to be inside a piece of code at a time. programs can **acquire** and **release** a lock

```python
// send a message by placing it in bb
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
```

our earlier problem stemmed from the fact that a program could be interrupted after adding **message** to **bb.buf**, but before incrementing **bb.in**

now, only one program can be “in” this section of the code at a time

**problem:** second sender could end up writing to full buffer

variables in use
- **bb** = the bounded buffer
- **message** = the message we’re trying to send/receive
- **bb.in** = total number of messages sent via this buffer
- **bb.out** = total number of messages received via this buffer
- **bb.lock** = lock intended to protect the bounded buffer
locks: allow only one CPU to be inside a piece of code at a time. programs can acquire and release a lock

```
// send a message by placing it in bb
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
```

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.lock = lock intended to protect the bounded buffer
locks: allow only one CPU to be inside a piece of code at a time. programs can **acquire** and **release** a lock

```plaintext
// send a message by placing it in bb
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
```

the previous problem stemmed from the fact that programs checked whether **bb.buf** had space before acquiring **bb.lock**

variables in use

**bb** = the bounded buffer
**message** = the message we’re trying to send/receive
**bb.in** = total number of messages sent via this buffer
**bb.out** = total number of messages received via this buffer
**bb.lock** = lock intended to protect the bounded buffer
locks: allow only one CPU to be inside a piece of code at a time. programs can acquire and release a lock

```
// send a message by placing it in bb
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

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

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.lock = lock intended to protect the bounded buffer
**locks:** allow only one CPU to be inside a piece of code at a time.

programs can **acquire** and **release** a lock

```
// send a message by placing it in bb
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

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

**variables in use**

- `bb` = the bounded buffer
- `message` = the message we’re trying to send/receive
- `bb.in` = total number of messages sent via this buffer
- `bb.out` = total number of messages received via this buffer
- `bb.lock` = lock intended to protect the bounded buffer

**problem:** deadlock* if buffer is full
locks: allow only one CPU to be inside a piece of code at a time. programs can \textbf{acquire} and \textbf{release} a lock

// send a message by placing it in bb
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

// receive a message from bb
receive(bb):
    acquire(bb.lock)
    while True:
        if bb.out < bb.in:
            message <- bb.buf[bb.out mod N]
            bb.out <- bb.out + 1
        release(bb.lock)
    return message

variables in use
bb = the bounded buffer
message = the message we’re trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.lock = lock intended to protect the bounded buffer

problem: deadlock* if buffer is full

*in 6.033, we’ll use “deadlock” to mean “two programs are waiting on each other, and neither can make progress until the other one does”
locks: allow only one CPU to be inside a piece of code at a time. programs can **acquire** and **release** a lock

```
// send a message by placing it in bb
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
```

```
// receive a message from bb
receive(bb):
    acquire(bb.lock)
    while bb.out >= bb.in:
        release(bb.lock)
        acquire(bb.lock)
    message <- bb.buf[bb.out mod N]
    bb.out <- bb.out + 1
    release(bb.lock)
    return message
```

variables in use

- **bb** = the bounded buffer
- **message** = the message we’re trying to send/receive
- **bb.in** = total number of messages sent via this buffer
- **bb.out** = total number of messages received via this buffer
- **bb.lock** = lock intended to protect the bounded buffer
locks: allow only one CPU to be inside a piece of code at a time. programs can **acquire** and **release** a lock

```c
// send a message by placing it in bb
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

// receive a message from bb
receive(bb):
    acquire(bb.lock)
    while bb.out >= bb.in:
        release(bb.lock)
        acquire(bb.lock)
        message <- bb.buf[bb.out mod N]
        bb.out <- bb.out + 1
        release(bb.lock)
    return message
```

variables in use:
- bb = the bounded buffer
- message = the message we’re trying to send/receive
- bb.in = total number of messages sent via this buffer
- bb.out = total number of messages received via this buffer
- bb.lock = lock intended to protect the bounded buffer
locks: allow only one CPU to be inside a piece of code at a time. programs can **acquire** and **release** a lock

```c
// send a message by placing it in bb
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

// receive a message from bb
receive(bb):
acquire(bb.lock)
while bb.out >= bb.in:
    release(bb.lock)
acquire(bb.lock)
message <- bb.buf[bb.out mod N]
bb.out <- bb.out + 1
release(bb.lock)
return message
```

variables in use

- **bb** = the bounded buffer
- **message** = the message we’re trying to send/receive
- **bb.in** = total number of messages sent via this buffer
- **bb.out** = total number of messages received via this buffer
- **bb.lock** = lock intended to protect the bounded buffer

if you are unsatisfied by the performance of this code, *that’s okay*; we’re going to revisit it
locks: create **atomic actions**. deciding what actions should be atomic, while balancing **performance**, is a challenge
locks: create **atomic actions**. deciding what actions should be atomic, while balancing **performance**, is a challenge

```plaintext
// move a file from one directory to another
move(dir1, dir2, filename):
    unlink(dir1, filename)
    link(dir2, filename)
```
locks: create **atomic actions**. deciding what actions should be atomic, while balancing **performance**, is a challenge

```c
// move a file from one directory to another
move(dir1, dir2, filename):
    unlink(dir1, filename)
    link(dir2, filename)
```

**variables in use**
- `dir1` = the directory to move the file from
- `dir2` = the directory to move the file to
- `filename` = the absolute path of the file
locks: create **atomic actions**. deciding what actions should be atomic, while balancing **performance**, is a challenge.

```python
// move a file from one directory to another
move(dir1, dir2, filename):
    acquire(fs_lock)
    unlink(dir1, filename)
    link(dir2, filename)
    release(fs_lock)
```

**variables in use**
- `dir1` = the directory to move the file from
- `dir2` = the directory to move the file to
- `filename` = the absolute path of the file
- `fs_lock` = a global lock held whenever a program interacts with the filesystem
locks: create **atomic actions**. deciding what actions should be atomic, while balancing **performance**, is a challenge

```c
// move a file from one directory to another
move(dir1, dir2, filename):
    acquire(fs_lock)
    unlink(dir1, filename)
    link(dir2, filename)
    release(fs_lock)
```

**problem:** poor performance

**variables in use**
- `dir1` = the directory to move the file from
- `dir2` = the directory to move the file to
- `filename` = the absolute path of the file
- `fs_lock` = a global lock held whenever a program interacts with the filesystem
locks: create atomic actions. deciding what actions should be atomic, while balancing performance, is a challenge

// move a file from one directory to another
move(dir1, dir2, filename):
    acquire(dir1.lock)
    unlink(dir1, filename)
    release(dir1.lock)
    acquire(dir2.lock)
    link(dir2, filename)
    release(dir2.lock)

variables in use

- **dir1** = the directory to move the file from
- **dir2** = the directory to move the file to
- **filename** = the absolute path of the file
- **dir1.lock**, **dir2.lock** = directory-specific locks
locks: create **atomic actions**. deciding what actions should be atomic, while balancing **performance**, is a challenge

```python
// move a file from one directory to another
move(dir1, dir2, filename):
    acquire(dir1.lock)
    unlink(dir1, filename)
    release(dir1.lock)
    acquire(dir2.lock)
    link(dir2, filename)
    release(dir2.lock)

problem: exposes inconsistent state
```

variables in use

- `dir1` = the directory to move the file from
- `dir2` = the directory to move the file to
- `filename` = the absolute path of the file
- `dir1.lock`, `dir2.lock` = directory-specific locks
locks: create **atomic actions**. deciding what actions should be atomic, while balancing **performance**, is a challenge

```python
// move a file from one directory to another
move(dir1, dir2, filename):
    acquire(dir1.lock)
    acquire(dir2.lock)
    unlink(dir1, filename)
    link(dir2, filename)
    release(dir1.lock)
    release(dir2.lock)
```

**variables in use**
- `dir1` = the directory to move the file from
- `dir2` = the directory to move the file to
- `filename` = the absolute path of the file
- `dir1.lock, dir2.lock` = directory-specific locks
locks: create **atomic actions**. deciding what actions should be atomic, while balancing **performance**, is a challenge

```plaintext
// move a file from one directory to another
move(dir1, dir2, filename):
    acquire(dir1.lock)
    acquire(dir2.lock)
    unlink(dir1, filename)
    link(dir2, filename)
    release(dir1.lock)
    release(dir2.lock)
```

**problem:** deadlock

variables in use
- `dir1` = the directory to move the file from
- `dir2` = the directory to move the file to
- `filename` = the absolute path of the file
- `dir1.lock`, `dir2.lock` = directory-specific locks
locks: create **atomic actions**. deciding what actions should be atomic, while balancing **performance**, is a challenge

```python
// move a file from one directory to another
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)
```

variables in use
- `dir1` = the directory to move the file from
- `dir2` = the directory to move the file to
- `filename` = the absolute path of the file
- `dir1.lock`, `dir2.lock` = directory-specific locks
- `dir1.inum`, `dir2.inum` = i-numbers for each directory
locks: create **atomic actions**. deciding what actions should be atomic, while balancing **performance**, is a challenge

// move a file from one directory to another
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.lock here instead

variables in use
- dir1 = the directory to move the file from
- dir2 = the directory to move the file to
- filename = the absolute path of the file
- dir1.lock, dir2.lock = directory-specific locks
- dir1.inum, dir2.inum = i-numbers for each directory
to believe that all of this works, we should understand the implementations of acquire and release

we can treat a lock as a flag that is true (1) when the lock is held and false (0) otherwise

acquire(lock):

release(lock):

variables in use
lock = the lock being acquired/released
to believe that all of this works, we should understand the implementations of \textbf{acquire} and \textbf{release}

we can treat a lock as a flag that is true (1) when the lock is held and false (0) otherwise

\begin{align*}
\textbf{acquire(} \text{lock} \text{)}: \quad \textbf{release(} \text{lock} \text{)}: \\
\text{lock} = 0
\end{align*}
to believe that all of this works, we should understand the implementations of **acquire** and **release**

we can treat a lock as a flag that is true (1) when the lock is held and false (0) otherwise

**acquire**(*lock*):

**release**(*lock*):

`lock = 0`

*lock* is released; no program holds it

---

**variables in use**

*lock* = the lock being acquired/released
to believe that all of this works, we should understand the implementations of acquire and release

we can treat a lock as a flag that is true (1) when the lock is held and false (0) otherwise

acquire(\texttt{lock}): 
while \texttt{lock} \neq 0:
do nothing

release(\texttt{lock}):
\texttt{lock} = 0

\texttt{lock} is released; no program holds it

variables in use
\texttt{lock} = the lock being acquired/released
to believe that all of this works, we should understand the implementations of acquire and release

we can treat a lock as a flag that is true (1) when the lock is held and false (0) otherwise

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

release(lock):
    lock = 0
```

variables in use

`lock` = the lock being acquired/released

another program holds lock; it can’t be acquired

lock is released; no program holds it
to believe that all of this works, we should understand the implementations of **acquire** and **release**

we can treat a lock as a flag that is true (1) when the lock is held and false (0) otherwise

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

release(lock):
    lock = 0

lock is released; no program holds it
```

**variables in use**

lock = the lock being acquired/released
to believe that all of this works, we should understand the implementations of **acquire** and **release**

we can treat a lock as a flag that is true (1) when the lock is held and false (0) otherwise

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

**release**(lock):
  lock = 0

another program holds lock; it can't be acquired

lock is released; no program holds it

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

**variables in use**
lock = the lock being acquired/released
to believe that all of this works, we should understand the implementations of acquire and release

we can treat a lock as a flag that is true (1) when the lock is held and false (0) otherwise

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

\[
\begin{align*}
\text{release}(\text{lock}): \\
\text{lock} = 0
\end{align*}
\]

another program holds lock; it can’t be acquired

\[
\begin{align*}
\text{lock} \text{ is released; no program holds it}
\end{align*}
\]
to believe that all of this works, we should understand the implementations of acquire and release

we can treat a lock as a flag that is true (1) when the lock is held and false (0) otherwise

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

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

implementing locks requires hardware support — namely an atomic exchange operation. much like how the MMU needs the physical address of page tables, and DNS clients need to know the IP address of a root server

\text{variables in use}
\[
\text{lock} = \text{the lock being acquired/released}
\]
// send a message by placing it in bb
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

// receive a message from bb
receive(bb):
    acquire(bb.lock)
    while bb.out >= bb.in:
        release(bb.lock)
        acquire(bb.lock)
    message <- bb.buf[bb.out mod N]
    bb.out <- bb.out + 1
    release(bb.lock)
    return message

lingering **performance issue**: this is a *lot* of releasing and acquiring, especially if the buffer remains full (or empty) for some time. we will address this in Wednesday’s lecture.

there is also something a bit unsatisfying about locks, in that we often need a global understanding of how they’re used; we’ll also come back to that later in 6.033
operating systems enforce modularity on a single machine

in order to enforce modularity + have an effective operating system, a few things need to happen

1. programs shouldn’t be able to refer to (and corrupt) each others’ memory

2. programs should be able to communicate with each other

3. programs should be able to share a CPU without one program halting the progress of the others
bounded buffers allow programs to communicate, completing the second step of enforcing modularity on a single machine. dealing with concurrency opens up a number of new challenges

locks allow us to implement atomic actions. determining the correct locking discipline can be tough thanks to race conditions, deadlock, and performance issues