6.033 in the news

Matt Dovey @mattdoveywriter

FWIW, because Wordle works entirely in the client, you can just Save Page As to keep a local copy if you're worried about the NYT acquisition.

The list of future words is already in the code, it just picks the word on the basis of your computer's date. It's everything you need.

7:07 AM · Feb 1, 2022 · Twitter Web App

145 Retweets 17 Quote Tweets 403 Likes

we don’t always use a client/server setup — having a separate client and server, with a network between them, has both benefits and drawbacks (but this setup has quite a lot of benefits as our systems grow larger)
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 + 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

   virtual memory

   assume they don’t need to (for today)

   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.

```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
- `bb.buf` = the actual buffer for storing messages
- $N$ = total number of messages $bb.buf$ can hold (assume $N$ is large)
**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

// 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
- `bb.buf` = the actual buffer for storing messages
- `N` = total number of messages `bb.buf` can hold (assume `N` is large)
**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 \( bb \)
```python
def 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 \( bb \)
```python
def 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
- \( bb.buf \) = the actual buffer for storing messages
- \( N \) = total number of messages \( bb.buf \) can hold (assume \( N \) is large)
**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.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
- **bb.buf** = the actual buffer for storing messages
- **N** = total number of messages bb.buf can hold (assume N is large)
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

this code is **incorrect** if we swap these two lines!

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.buf = the actual buffer for storing messages
$N$ = total number of messages bb.buf can hold (assume $N$ is large)
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

// 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
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is 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
- `bb.buf` = the actual buffer for storing messages
- `N` = total number of messages `bb.buf` can hold (assume N is large)
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
```

jolene is trying to send message \( m_2 \)

broccoli is trying to send message \( m_1 \)

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
- \( bb.buf \) = the actual buffer for storing messages
- \( N \) = total number of messages \( bb.buf \) can hold (assume \( N \) is large)
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
```

jolene 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
- \(bb.buf\) = the actual buffer for storing messages
- \(N\) = total number of messages \(bb.buf\) can hold (assume \(N\) is 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
```

jolene 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
- \( bb.buf \) = the actual buffer for storing messages
- \( N \) = total number of messages \( bb.buf \) can hold (assume \( N \) is large)
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
- **bb.buf**: the actual buffer for storing messages
- **N**: total number of messages bb.buf can hold (assume N is large)

```
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
```
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
- `bb.buf` = the actual buffer for storing messages
- `N` = total number of messages `bb.buf` can hold (assume `N` is large)

broccoli is trying to send message \( m_1 \)

jolene 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
```

current line: 3

current line: 1
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
```

jolene is trying to send message \( m_2 \)

current line: 4

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
- \( bb.buf \) = the actual buffer for storing messages
- \( N \) = total number of messages \( bb.buf \) can hold (assume \( N \) is 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

broccoli is trying to send message \( m_1 \)

jolene 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
\( bb.buf \) = the actual buffer for storing messages
\( N \) = total number of messages \( bb.buf \) can hold (assume \( N \) is large)
what happens when multiple programs try to send?

broccoli is trying to send message \( m_1 \)

jolene 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

current line: 5

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.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is 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
```

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

current line: 5

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
- \( bb.buf \) = the actual buffer for storing messages
- \( N \) = total number of messages \( bb.buf \) can hold (assume \( N \) is large)
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
```

jolene 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
- $bb.buf$ = the actual buffer for storing messages
- $N$ = total number of messages $bb.buf$ can hold (assume $N$ is large)
what happens when multiple programs try to send?

broccoli is trying to send message \( m_1 \)

complete

```
// 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
```

jolene 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
- \( bb.buf \): the actual buffer for storing messages
- \( N \): total number of messages \( bb.buf \) can hold (assume \( N \) is 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
```

current line: 1

jolene is trying to send message $m_2$

complete

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
- $bb.buf$ = the actual buffer for storing messages
- $N$ = total number of messages $bb.buf$ can hold (assume $N$ is large)
what happens when multiple programs try to send?

broccoli is trying to send message $m_1$

```
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
```

complete

jolene 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
- `bb.buf` = the actual buffer for storing messages
- `N` = total number of messages `bb.buf` can hold (assume $N$ is 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
```

current line: 3

jolene 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
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is 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
```

current line: 4

jolene 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
- $bb.buf$ = the actual buffer for storing messages
- $N$ = total number of messages $bb.buf$ can hold (assume $N$ is 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

broccoli is trying to
send message \(m_1\)

jolene is trying to
send message \(m_2\)

complete

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.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
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
- `bb.buf` = the actual buffer for storing messages
- `N` = total number of messages `bb.buf` can hold (assume $N$ is large)

Jolene is trying to send message $m_2$

Current line: 6
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

broccoli is trying to send message $m_1$

jolene is trying to send message $m_2$

complete

complete

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.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
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
\( bb.buf \) = the actual buffer for storing messages
\( N \) = total number of messages \( bb.buf \) can hold (assume \( N \) is large)

broccoli is trying to send message \( m_1 \)

jolene 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: \( bb.buf[bb.in \mod N] \leftarrow message \)
5: \( bb.in \leftarrow bb.in + 1 \)
6: return
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
```

jolene 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
- \( bb.buf \) = the actual buffer for storing messages
- \( N \) = total number of messages \( bb.buf \) can hold (assume \( N \) is large)
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
- `bb.buf` = the actual buffer for storing messages
- `N` = total number of messages `bb.buf` can hold (assume `N` is large)

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: 2

jolene is trying to send message \( m_2 \)

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

broccoli is trying to send message $m_1$

jolene 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
```

current line: 3

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
- **bb.buf** = the actual buffer for storing messages
- **N** = total number of messages bb.buf can hold (assume N is large)
what happens when multiple programs try to send?

```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
- `bb.buf` = the actual buffer for storing messages
- `N` = total number of messages `bb.buf` can hold (assume `N` is large)
what happens when multiple programs try to send?

broccoli is trying to send message \( m_1 \)

jolene is trying to send message \( m_2 \)

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

current line: 4

current line: 3

variables in use
\begin{align*}
\textbf{bb} &= \text{the bounded buffer} \\
\textbf{message} &= \text{the message we’re trying to send/receive} \\
\textbf{bb.in} &= \text{total number of messages sent via this buffer} \\
\textbf{bb.out} &= \text{total number of messages received via this buffer} \\
\textbf{bb.buf} &= \text{the actual buffer for storing messages} \\
N &= \text{total number of messages bb.buf can hold (assume N is large)}
\end{align*}
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
- **bb.buf**: the actual buffer for storing messages
- **N**: total number of messages bb.buf can hold (assume N is 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
```

jolene is trying to send message $m_2$

current line: 5

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
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is 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

broccoli is trying to send message $m_1$

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
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)

jolene is trying to send message $m_2$
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

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.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is 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
```

current line: 6

jolene is trying to send message $m_2$

```
```

current line: 6

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.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is 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
```

complete

jolene is trying to send message \( m_2 \)

current line: 6

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.buf` = the actual buffer for storing messages
- \( N \) = total number of messages `bb.buf` can hold (assume \( N \) is large)
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
```

jolene 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
- `bb.buf` = the actual buffer for storing messages
- `N` = total number of messages `bb.buf` can hold (assume `N` is large)
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
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
// 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
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
// 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
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
// send a message by placing it in bb

```python
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
- `bb.buf` = the actual buffer for storing messages
- `N` = total number of messages `bb.buf` can hold (assume `N` is large)
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:
            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
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
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
1: send(bb, message):
2:    while True:
3:        if bb.in - bb.out < N:
4:            acquire(bb.lock)
5:            bb.buf[bb.in mod N] <- message
6:            bb.in <- bb.in + 1
7:            release(bb.lock)
8:        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.buf = the actual buffer for storing messages
- N = total number of messages bb.buf can hold (assume N is large)
- bb.lock = lock intended to protect the bounded buffer
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.

```
// send a message by placing it in bb
1: send(bb, message):
2:    while True:
3:        if bb.in - bb.out < N:
4:            acquire(bb.lock)
5:            bb.buf[bb.in mod N] <- message
6:            bb.in <- bb.in + 1
7:            release(bb.lock)
8:    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.buf` = the actual buffer for storing messages
- `N` = total number of messages `bb.buf` can hold (assume `N` is large)
- `bb.lock` = lock intended to protect the bounded buffer
// send a message by placing it in the bounded buffer
1: `send(bb, message)`:  
2: while True:  
3: if `bb.in - bb.out < N`:  
4: `acquire(bb.lock)`  
5: `bb.buf[bb.in mod N] <- message`  
6: `bb.in <- bb.in + 1`  
7: `release(bb.lock)`  
8: 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.

question: suppose the buffer has room for exactly one more message. program A and program B each call `send`. what might happen?

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.buf` = the actual buffer for storing messages
- `N` = total number of messages `bb.buf` can hold (assume `N` is large)
- `bb.lock` = lock intended to protect the bounded buffer
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.buf` = the actual buffer for storing messages
- `N` = total number of messages `bb.buf` can hold (assume `N` is large)
- `bb.lock` = lock intended to protect the bounded buffer

problem: second sender could end up writing to full 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 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.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
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.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
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
```

problem: deadlock* if buffer is full

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.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
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

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.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
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 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.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
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
```

```c
// 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.buf` = the actual buffer for storing messages
- `N` = total number of messages `bb.buf` can hold (assume `N` is large)
- `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.

```javascript
// 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

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

```plaintext
// 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

```c
// 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(dir2.lock)
    release(dir1.lock)
```
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)
```

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 **acquire** and **release**

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

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

**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

```plaintext
acquire(lock):
lock = 0

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}):  
\begin{align*}
\text{while } \texttt{lock } &\neq 0: \\
\text{do nothing}
\end{align*}

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

\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 \texttt{acquire} and \texttt{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(} \texttt{lock} \textbf{)}: & \quad \text{while } \texttt{lock} \neq 0: \quad \texttt{do nothing} \\
\textbf{release(} \texttt{lock} \textbf{)}: & \quad \texttt{lock} = 0
\end{align*}

another program holds \texttt{lock}; it can’t be acquired

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

\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\( (\text{lock}):\)
\[
\text{while } \text{lock} \neq 0:\n\quad \text{do nothing}
\]
\[
\text{lock} = 1
\]

release\( (\text{lock}):\)
\[
\text{lock} = 0
\]

another program holds \text{lock}; it can’t be acquired

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

variables in use
\text{lock} = the lock being acquired/released

Katrina LaCurts | lacurts@mit.edu | 6.033 2022
to believe that all of this works, we should understand the implementations of \texttt{acquire} and \texttt{release}

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

<table>
<thead>
<tr>
<th>acquire(\texttt{lock}):</th>
<th>release(\texttt{lock}):</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{while lock} \neq 0:</td>
<td>\texttt{lock} = 0</td>
</tr>
<tr>
<td>\texttt{do nothing}</td>
<td>\texttt{lock} is released; no program holds it</td>
</tr>
</tbody>
</table>

\texttt{lock} = 1

\textbf{problem}: race condition

(need locks to implement locks!)

\textbf{variables in use}

\texttt{lock} = the lock being acquired/released
to believe that all of this works, we should understand the implementations of \texttt{acquire} and \texttt{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}(\texttt{lock}) & : \\
\texttt{while} \ \texttt{lock} \neq 0 : \\
\texttt{do nothing} \\
\texttt{lock} & = 1
\end{align*}

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

another program holds \texttt{lock}; it can’t be acquired

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

\textit{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
    lock = 1
release(lock):
    lock = 0
```

another program holds lock; it can’t be acquired

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

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

\[ \text{acquire}(\text{lock}): \]
\[ \text{do:} \]
\[ r \leftarrow 1 \]
\[ \text{XCHG } r, \text{ lock} \]
\[ \text{while } r \equiv 1 \]

\[ \text{release}(\text{lock}): \]
\[ \text{lock } = 0 \]

XCHG atomically swaps the value of \( r \) and \( \text{lock} \); it cannot be interrupted in the middle of this action

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

variables in use
\( \text{lock} \) = the lock being acquired/released
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 the next 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
   virtualize memory

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

notice that we have choices about how apply locks (e.g., fine-grained, coarse-grained). those choices impact the performance and simplicity of our systems, which in turn impacts users, developers, and beyond

(and right now, performance and simplicity appear to be at odds)