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.

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)
6.033 Spring 2022
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

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.

// 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.

```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
```

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?

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

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 \( m_1 \)

```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 \( m_2 \)
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?

```
// 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?

// 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 \( m_2 \)

```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?

```
// 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
```

complete

ejolene 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
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: 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 $m_1$

```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: 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
$bb.buf$ = 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
- \( bb\text{-}in \) = total number of messages sent via this buffer
- \( bb\text{-}out \) = total number of messages received via this buffer
- \( bb\text{-}buf \) = the actual buffer for storing messages
- \( N \) = total number of messages \( bb\text{-}buf \) 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
```

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 $m_1$

// 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: 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

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?

broccoli is trying to send message \( m_1 \)

jolene is trying to send message \( m_2 \)

current line: 1

defined at current line: 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

defined at 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
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 $m_2$

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 \( m_1 \)

current line: 2

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

jolene is trying to send message \( m_2 \)

current line: 2

-> 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?

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: 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?

// 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: 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?

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

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
```

current line: 5

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

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:     if 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: 6

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?

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: 6

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 \( m_1 \)

variables in use

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

```
// 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 \( m_2 \)

complete
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 \( m_2 \)

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)
this implementation of send and receive only works with a single sender and receiver; it can introduce **race conditions** with multiple senders

```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
- `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
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
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
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
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
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

**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.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

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

if you are unsatisfied by the performance of this code, that’s okay; we’re going to revisit it

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 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

```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

```java
// 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)
```

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($lock$):

1. while $lock$ != 0: 
   1. do nothing 
2. $lock = 1$

### release($lock$):

1. $lock = 0$

**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

**acquire**(lock):
  do:
    r <- 1
    XCHG r, lock
    while r == 1

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

XCHG atomically swaps the value of r and 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
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

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

   → virtualize memory

   → bounded buffers
   (virtualize communication links)

   → 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)