Lecture #4: Bounded Buffers + Locks

getting many programs to communicate at once
MIT Closed Tuesday February 13
Feb. 12, 2024, 8:01 p.m.

Due to the expected winter storm, MIT will be closed for all non-essential employees on Tuesday, February 13, from 7 a.m. to 11 p.m.

Employees who are able to work remotely, including those on a hybrid schedule, are generally expected to work their regularly scheduled hours. More detail can be found at Employment Policy Manual Section 5.6.7.
6.1800 in the news

why does emergency.mit.net exist when we have emergency.mit.edu?

---

Active Message

MIT Closed Tuesday February 13
Feb. 12, 2024, 8:01 p.m.

Due to the expected winter storm, MIT will be closed for all non-essential employees on Tuesday, February 13, from 7 a.m. to 11 p.m.

Employees who are able to work remotely, including those on a hybrid schedule, are generally expected to work their regularly scheduled hours. More detail can be found at Employment Policy Manual Section 5.6.7.
6.1800 in the news

why does emergency.mit.net exist when we have emergency.mit.edu?

“MIT owns the domain mit.net and is running the emergency notification service on http://emergency.mit.net/. It is replicated and will normally go to the same place as http://emergency.mit.edu/. Having it routed through a .net domain gives MIT additional recovery options in case something happens to the campus network or the registrar for .edu domains.”
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.

```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
```

**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
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

```python
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.

```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
- `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.

```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
```

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 \( m_1 \)

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

junebug is trying to send message \( m_2 \)

\[
\begin{align*}
\texttt{bb.in} &= 0 \\
\texttt{bb.out} &= 0 \\
\texttt{bb.buf} &= [ | | | | | ]
\end{align*}
\]

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

junebug is trying to send message $m_2$

```
broccoli is trying to send message $m_1$

```

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)

Katrina LaCurts | lacurts@mit.edu | 6.1800 2024
what happens when multiple programs try to send?

broccoli is trying to send message m₁

junebug is trying to send message m₂

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

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

bb.in = 0
bb.out = 0
bb.buf = [ | | | | | ]
```

Junebug 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 $m_1$

```
// 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
```

junebug is trying to send message $m_2$

```
bb.in = 0
bb.out = 0
bb.buf = [ | | | | ]
```

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
```

junebug is trying to send message \( m_2 \)

```
broccoli is trying to send message \( m_1 \)

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)
```

```
b.b.in = 0
bb.out = 0
bb.buf = [ m_1 | | | | ]
```

```
bb.in = 0
bb.out = 0
bb.buf = [ m_1 | | | | ]
```
what happens when multiple programs try to send?

broccoli is trying to send message $m_1$

junebug 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)

Current line: 5

Current line: 1

Current line: 5

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
```

Current line: 6

```
bb.in = 1
bb.out = 0
bb.buf = [ \( m_1 \) | | | | ]
```

 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)

Junebug 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

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)

junebug is trying to send message \( m_2 \)

complete

current line: 1

\( bb.in = 1 \)
\( bb.out = 0 \)
\( bb.buf = [ m_1 | | | | ] \)
what happens when multiple programs try to send?

broccoli is trying to send message $m_1$

![Broccoli](image1.png)

complete

junebug is trying to send message $m_2$

![Junebug](image2.png)

current line: 1

// send a message by placing it in bb

\[
\begin{align*}
1: & \quad \text{send}(\text{bb, message}): \\
2: & \quad \text{while True:} \\
3: & \quad \quad \text{if } \text{bb.in} - \text{bb.out} < N: \\
4: & \quad \quad \quad \text{bb.buf}[\text{bb.in mod } N] \leftarrow \text{message} \\
5: & \quad \quad \text{bb.in} \leftarrow \text{bb.in} + 1 \\
6: & \quad \text{return}
\end{align*}
\]

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

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
```

junebug is trying to send message \( m_2 \)

complete

```
bb.in = 1
bb.out = 0
bb.buf = [ \( m_1 \) | | | | ]
```

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 $m_1$

```
// 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
```

junebug is trying to send message $m_2$

```
bb.in = 1
bb.out = 0
bb.buf = [ 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 \( m_1 \)

junebug is trying to send message \( m_2 \)

complete

```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
```

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

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 \( m_1 \)

```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
```

junebug 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)

complete

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

junebug is trying to send message \(m_2\)

complete

\[
\begin{align*}
\text{bb.in} &= 2 \\
\text{bb.out} &= 0 \\
\text{bb.buf} &= [ m_1 | m_2 | | | ]
\end{align*}
\]

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
```

junebug is trying to send message $m_2$

```
bb.in = 2
bb.out = 0
bb.buf = [ m_1 | 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

junebug is trying to send message \( m_2 \)

```
bb.in = 0
bb.out = 0
bb.buf = [ | | | | | ]
```

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)

Katrina LaCurts | lacurts@mit.edu | 6.1800 2024
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

```

```
bb.in = 0
bb.out = 0
bb.buf = [ | | | | | ]
```

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)

junebug 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
```

junebug is trying to send message $m_2$

```
bb.in = 0
bb.out = 0
bb.buf = [ | | | | ]
```

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 \( bb.\text{in} - bb.\text{out} < N \):
  4: \( bb.\text{buf}[bb.\text{in} \mod N] \leftarrow \text{message} \)
  5: \( bb.\text{in} \leftarrow bb.\text{in} + 1 \)
  6: return
```

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

junebug is trying to send message \( m_2 \)

```
bb.\text{in} = 0
bb.\text{out} = 0
bb.\text{buf} = [ | | | | | ]
```

current line: 3

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

broccoli is trying to send message \( m_1 \)

junebug is trying to send message \( m_2 \)

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

bb.in = 0
bb.out = 0
bb.buf = [ | | | | | ]

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
```

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)

junebug is trying to send message $m_2$

```
bb.in = 0
bb.out = 0
bb.buf = [ | | | | | ]
```
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: 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)

junebug is trying to send message \( m_2 \)

\( bb.in = 0 \)
\( bb.out = 0 \)
\( bb.buf = [ m_1 | | | | ] \)

current line: 4
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
```

current line: 5

junebug is trying to send message $m_2$

```
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

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 \( m_1 \)

current line: 5

```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)

junebug is trying to send message \( m_2 \)

current line: 4

\( bb.in = 0 \)
\( bb.out = 0 \)
\( bb.buf = [ m_2 | | | | ] \)
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
```

junebug is trying to send message $m_2$

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

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
```

junebug 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
```

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)

junebug is trying to send message $m_2$

```
broccoli is trying to send message $m_1$

```

```
junebug 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)

```

```
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)

```

```
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)

```

```
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)

```

```
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)

```

```
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)

```

```
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)

```

```
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

junebug is trying to send message \( m_2 \)

```
bb.in = 2
bb.out = 0
bb.buf = [ 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

complete

junebug is trying to send message $m_2$

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

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

```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)
// 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
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.

```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
```

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
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

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
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.

```
// 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

```python
// 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.

```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
```

```c
// 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 (assumed N is large)
- **bb.lock** = lock intended to protect the bounded buffer

**question:** suppose the buffer is full. Program A calls send, and program B calls receive. What might happen?
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
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 6.1800, 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

```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.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.

```plaintext
// 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
```

```plaintext
// 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
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):
    unlink(dir1, filename)
    link(dir2, filename)
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):
    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.

```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)
```

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(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)
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)
```

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

```c
// 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
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

\[
\text{acquire(}\text{lock}\text{):} \quad \text{release(}\text{lock}\text{):}
\]

**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.

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

variables in use
\text{lock} = \text{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

\[
\text{acquire(}\text{lock}\text{):}
\]

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

\[
\text{lock} = 0
\]

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

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

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

*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

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

release(lock):
    lock = 0
```

another program holds lock; it can’t be acquired

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

another program holds lock; it can’t be acquired

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

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

release(lock):
    lock = 0

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

- **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

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

```
XCHG atomically swaps the value of r and lock; it cannot be interrupted in the middle of this action

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

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

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
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 unsatisfying about locks, in that we often need a global understanding of how they’re used; we’ll come back to that later in 6.1800.

```plaintext
// 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
```
…Since these filesystems may contain millions or hundreds of millions of files, most of which are inspected exactly once and found not to have changed, it generates a lot of "garbage" in kernel memory which must eventually be reclaimed. The kernel only actively collects this garbage, which it does by means of a pseudo-LRU queue, when it runs into a configured limit. **There is a broad-scope mutex which protects this queue, and one of the issues is that it is held too long while the garbage-collector is running, which causes any process on the system that needs to open a file -- including the NFS server process -- to block.**

- email from Garrett Wollman in CSAIL last fall
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

---

1. virtualize **memory**

2. bounded buffers
   - (virtualize communication links)

3. 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)