6.1800 Spring 2025

Lecture #4: Bounded Buffers + Locks
getting many programs to communicate at once

Katrina LaCurts | lacurts@mit.edu | 6.1800 2025

6.1800 in the news

6.1800 in the news

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

Emergency Management
Massachusetts Institute of Technology

No Active Message

There is currently no major emergency or disruption.

For a campus map, visit https://whereis.mit.edu.

Quick Emergency Response Actions.

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 + 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** ┈┈┈┈▸ **virtual memory**

2. programs should be able to **communicate** with each other ┈┈┈┈▸ **bounded buffers**
(virtualize communication links)

3. programs should be able to **share a CPU** without one program halting the progress of the others ┈┈┈┈▸ assume one program per CPU
(for today)

today's goal: implement **bounded buffers** so that programs can communicate

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

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

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

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
```

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

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

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

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
```

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

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

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

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
```

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

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

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

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
```

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

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

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

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

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
```

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

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

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

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
```
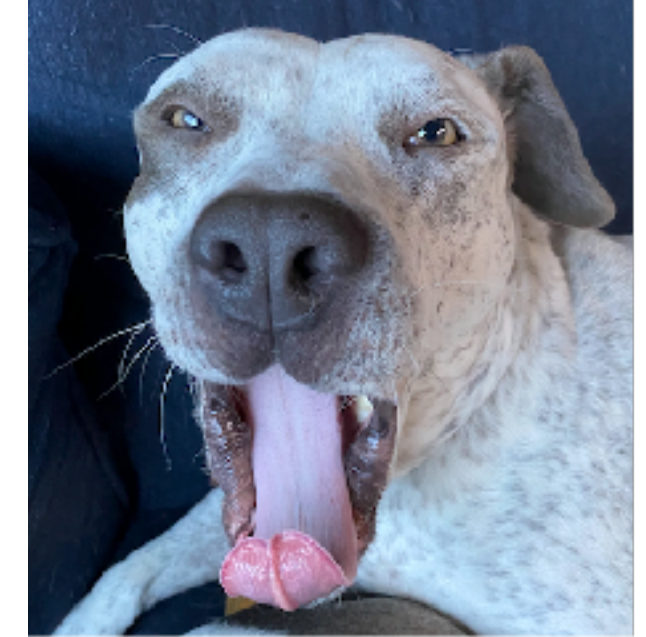
what happens when multiple programs try to send?

broccoli is trying to send message m₁

junebug is trying to send message m₂


```
// 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)
```

what happens when multiple programs try to send?

broccoli is trying to send message m_1

current line: 1

junebug is trying to send message m_2

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
```

```
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?

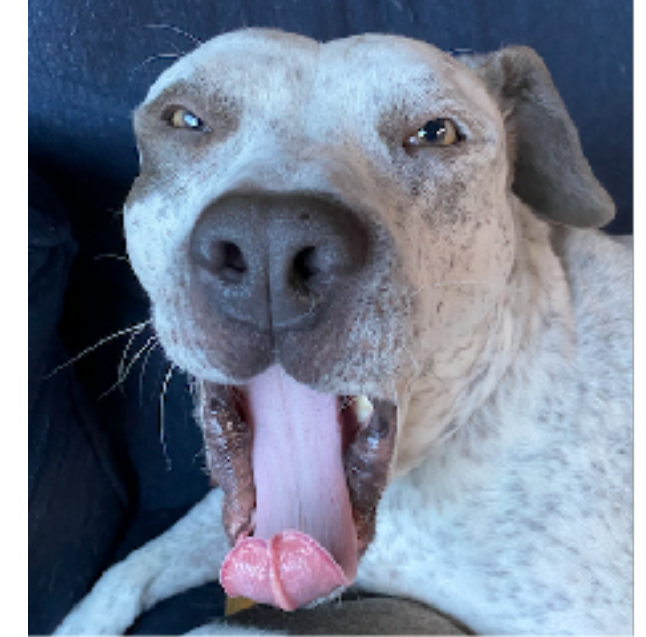**broccoli is trying to send message m₁**

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
```

junebug is trying to send message m₂

current line: 1

```
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?
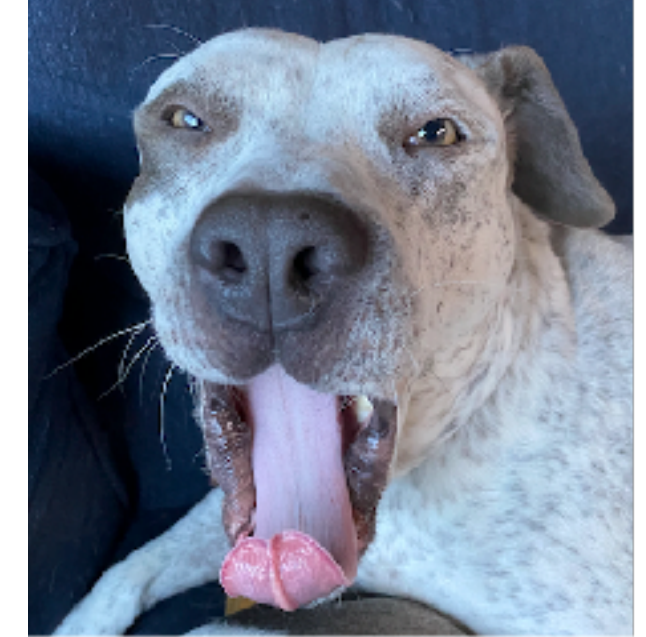
broccoli is trying to send message m$_1$

current line: 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
```

junebug is trying to send message m$_2$

current line: 1

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)
```

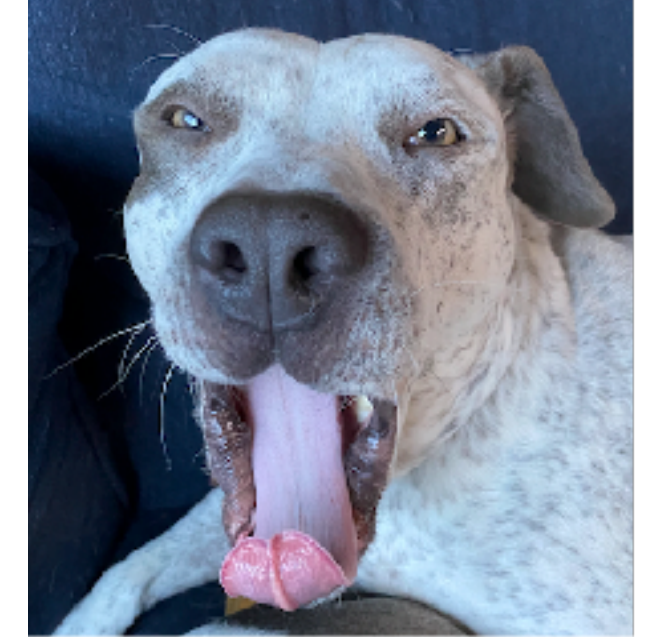# what happens when multiple programs try to send?

broccoli is trying to send message m₁

current line: 3

```
// 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 m₂

current line: 1

```
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)
```

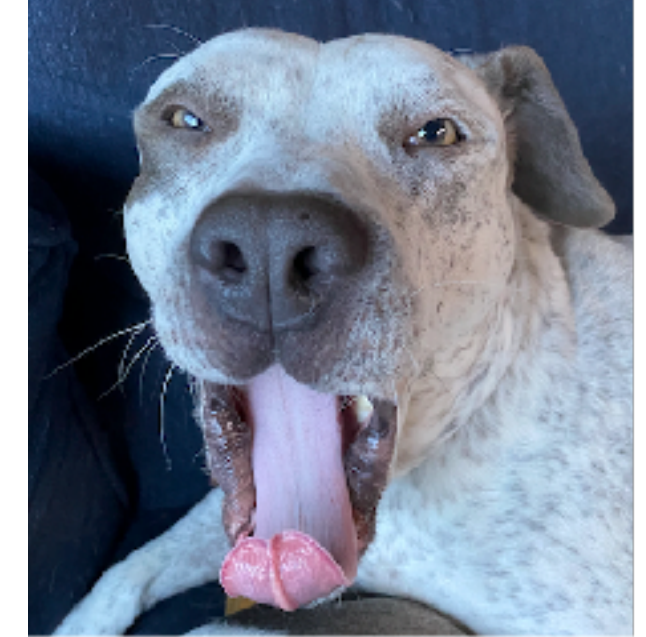# what happens when multiple programs try to send?

broccoli is trying to send message m_1

current line: 4

```
// 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

current line: 1

```
bb.in = 0
bb.out = 0
bb.buf = [ m₁ | | | | | ]
```

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

```
// 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

current line: 1

```
bb.in = 1
bb.out = 0
bb.buf = [ m₁ | | | | | ]
```

what happens when multiple programs try to send?

broccoli is trying to
send message m_1

current line: 6

```
// 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

current line: 1

```
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

complete

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

junebug is trying to send message m_2

current line: 1

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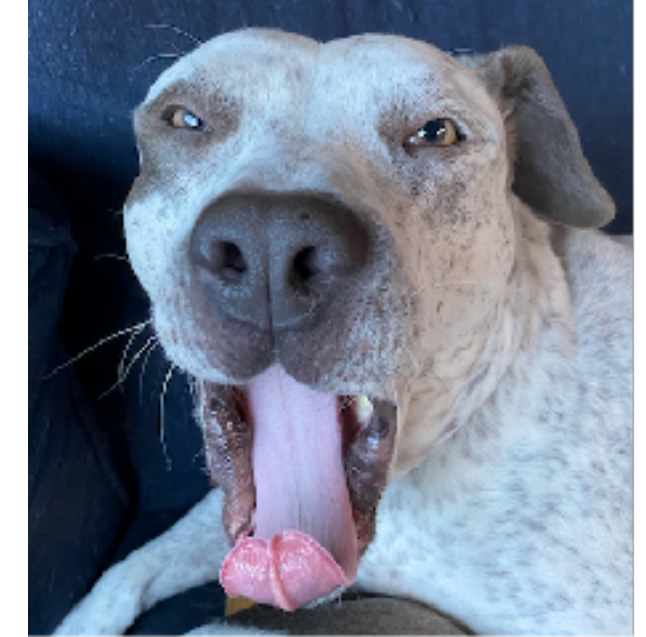
what happens when multiple programs try to send?

broccoli is trying to send message m_1

complete

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

junebug is trying to send message m_2

current line: 1

```
bb.in = 1
bb.out = 0
bb.buf = [ m1 | | | | | ]
```

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 m₁

complete

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

junebug is trying to send message m₂

current line: 2

```
bb.in = 1
bb.out = 0
bb.buf = [ m₁ |  |  |  |  |  ]
```

```
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?
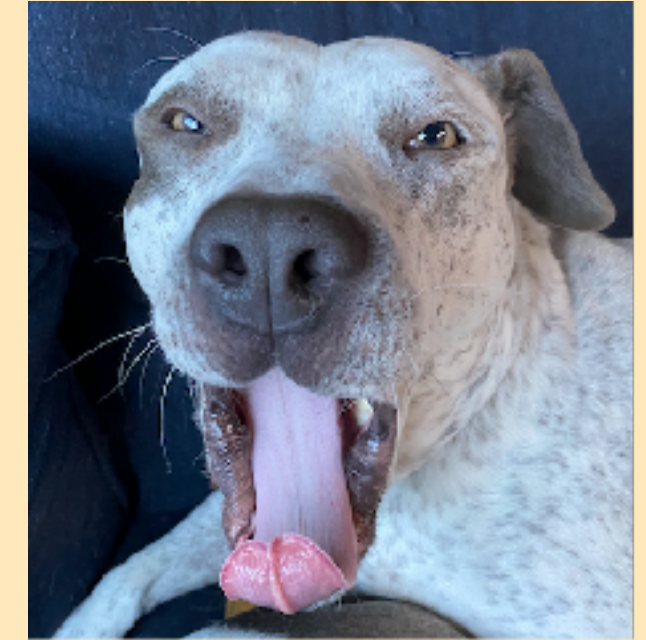
**broccoli is trying to
send message m₁**

complete

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

**junebug is trying to
send message m₂**

current line: 3

```
bb.in = 1
bb.out = 0
bb.buf = [ m₁ | | | | | ]
```

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
```

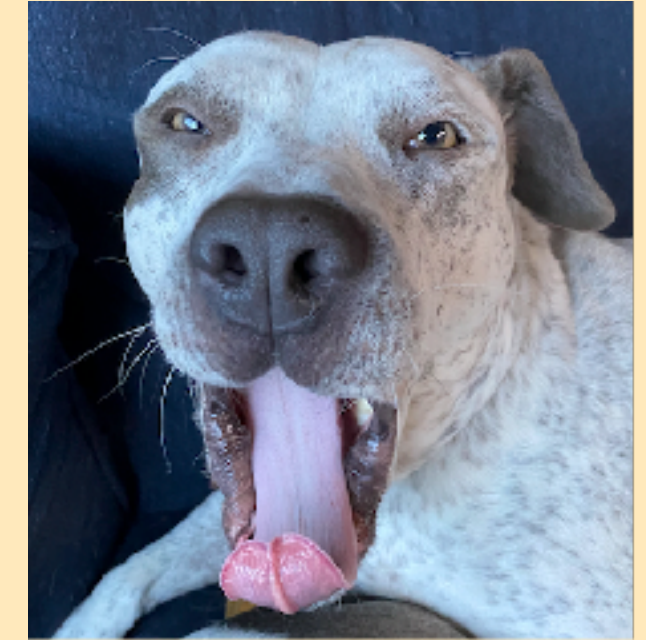# what happens when multiple programs try to send?

broccoli is trying to send message m_1

complete

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

junebug is trying to send message m_2

current line: 4

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?

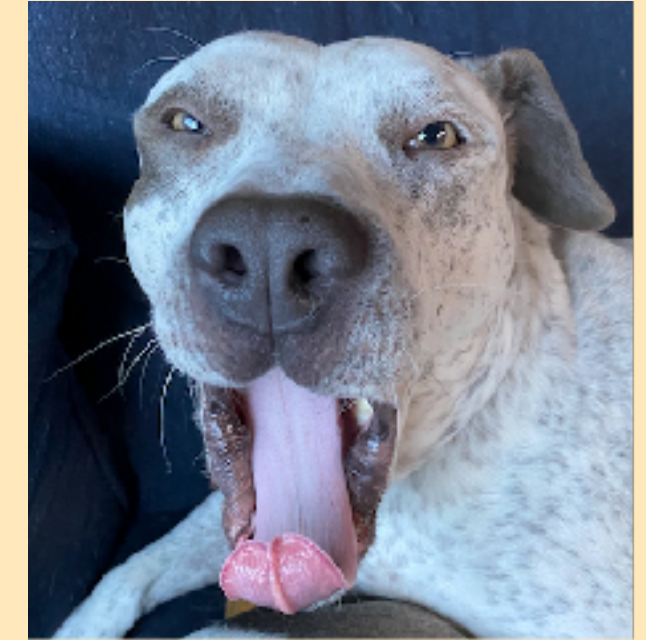**broccoli is trying to send message m_1**

complete

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

junebug is trying to send message m_2

current line: 5

```
bb.in = 2
bb.out = 0
bb.buf = [ m₁ | m₂ |  |  |  |  ]
```

```
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 m₁
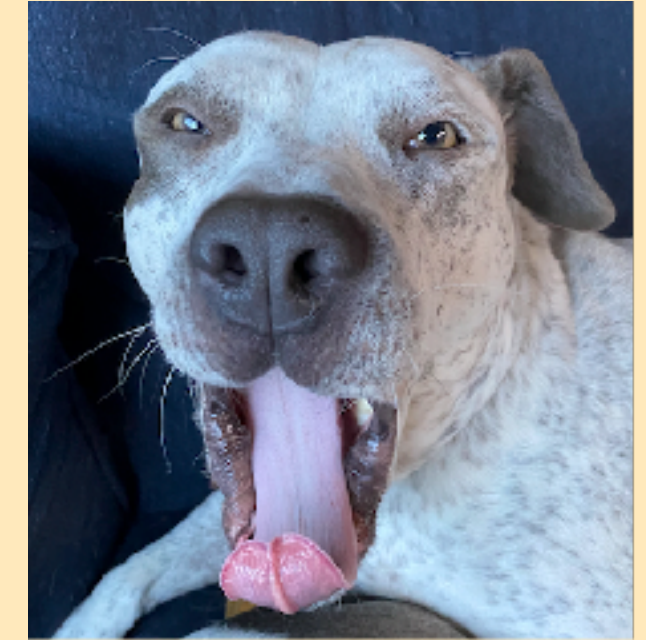

complete

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

junebug is trying to send message m₂

current line: 6

bb.in = 2
bb.out = 0
bb.buf = [m₁ | m₂ | | | |]

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
```

what happens when multiple programs try to send?

broccoli is trying to send message m_1

complete

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

junebug is trying to send message m_2

complete

```
bb.in = 2
bb.out = 0
bb.buf = [ m₁ | m₂ | | | | ]
```

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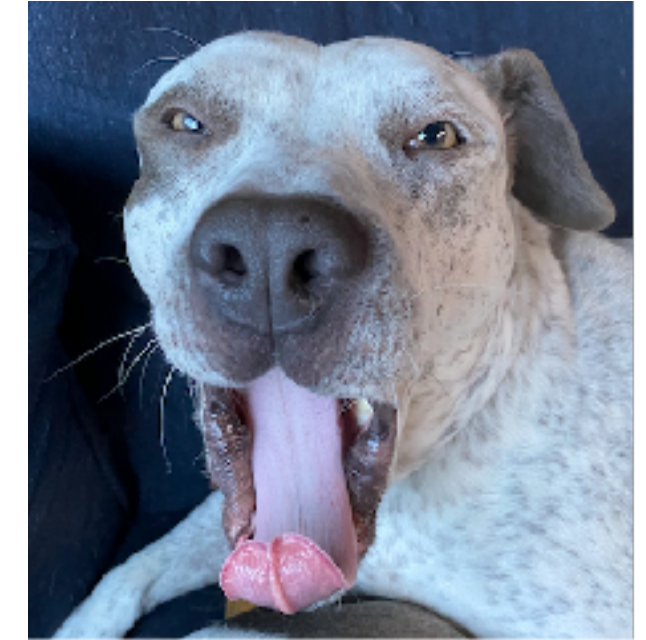
what happens when multiple programs try to send?

broccoli is trying to
send message m_1

current line: 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

current line: 1

```
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)
```

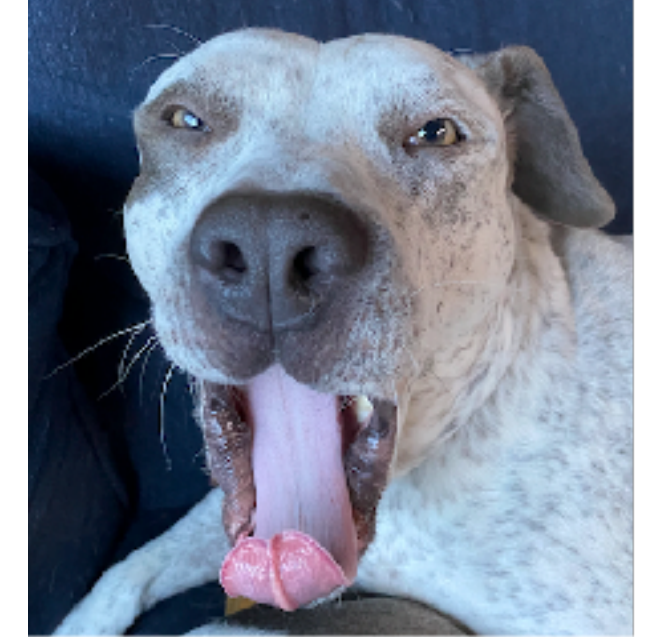# what happens when multiple programs try to send?

broccoli is trying to send message m₁

current line: 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
```

junebug is trying to send message m₂

current line: 1

```
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)
```

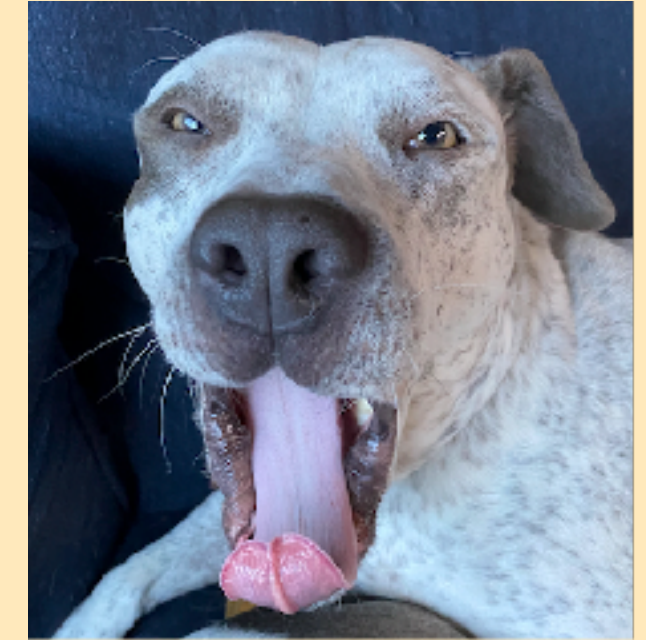# what happens when multiple programs try to send?

broccoli is trying to send message m₁

current line: 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
```

junebug is trying to send message m₂

current line: 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)
```

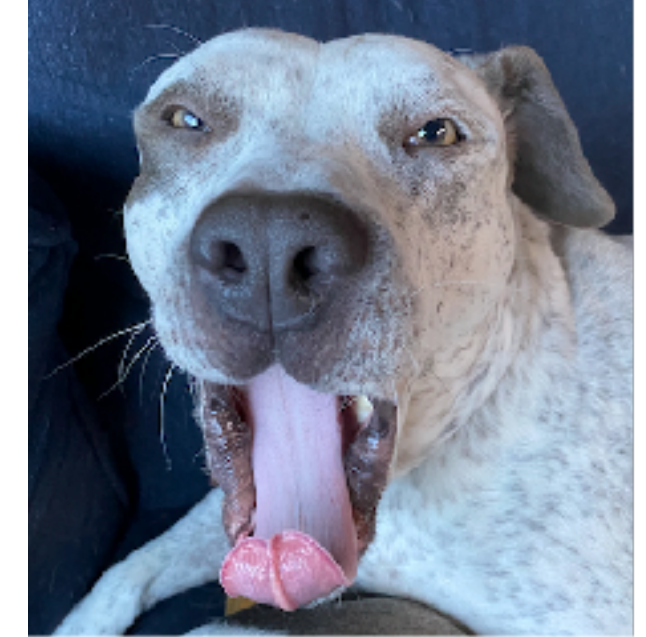# what happens when multiple programs try to send?

broccoli is trying to send message m₁

current line: 3

```
// 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 m₂

current line: 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

current line: 3

```
// 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

current line: 3

```
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)
```

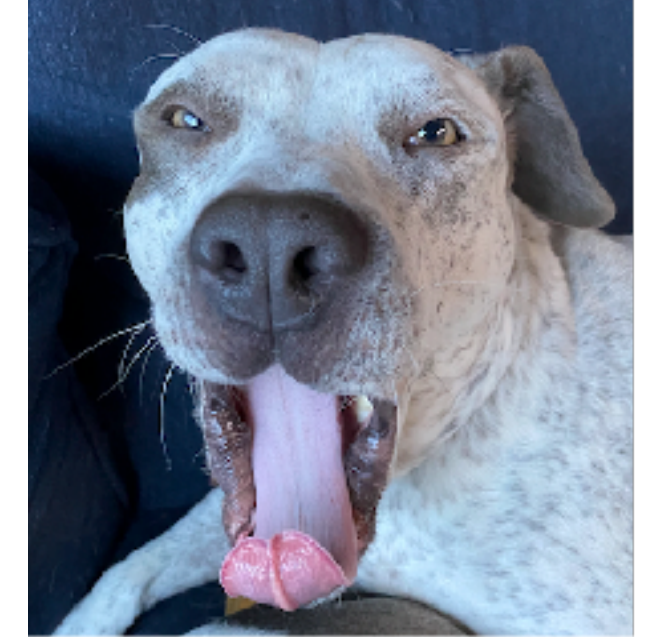# what happens when multiple programs try to send?

broccoli is trying to send message m₁

current line: 4

```
// 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 m₂

current line: 3

```
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?

current line: 4

current line: 4

```
// 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 = [ m1 | | | | | ]
```

```
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?
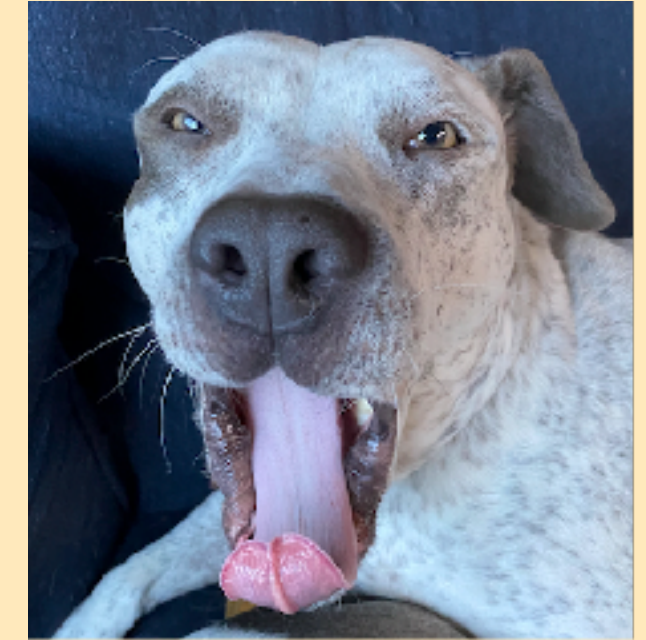
broccoli is trying to send message m_1

current line: 5

```
// 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

current line: 4

```
bb.in = 0
bb.out = 0
bb.buf = [ m₁ | | | | | ]
```

```
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)
```

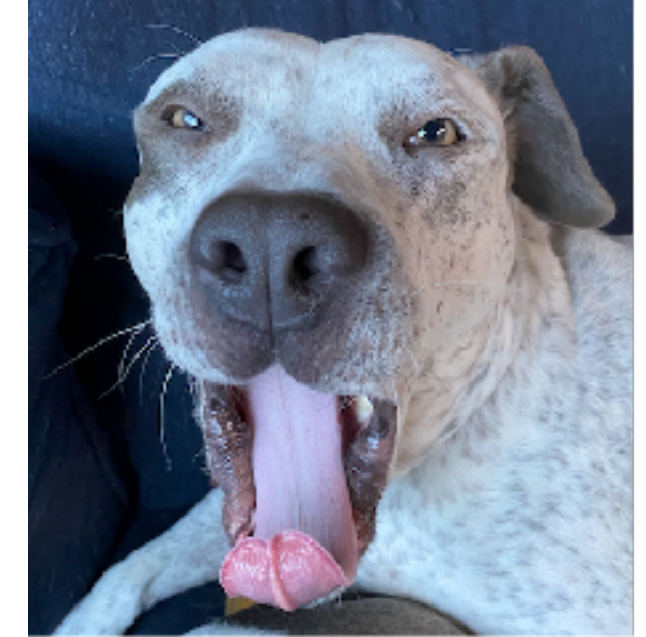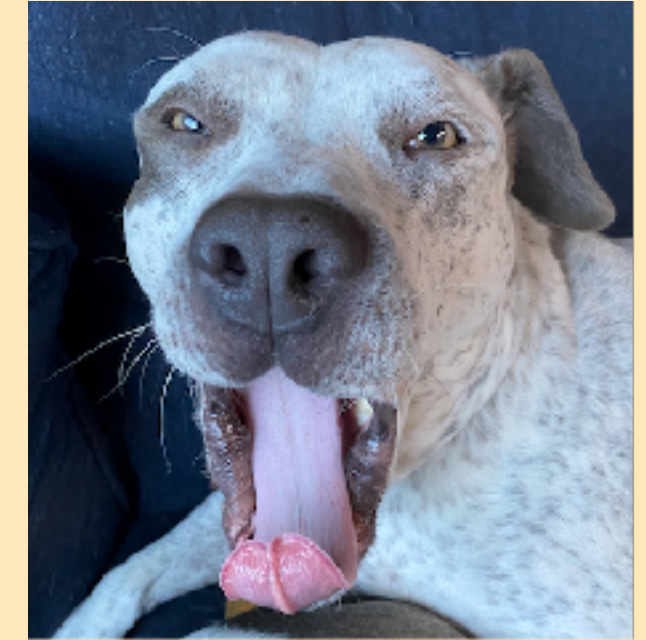# what happens when multiple programs try to send?

broccoli is trying to send message m₁

current line: 5

```
// 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 m₂

current line: 4

```
bb.in = 0
bb.out = 0
bb.buf = [ m₂ | | | | | ]
```

```
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?

current line: 5

```
// 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

```
bb.in = 1
bb.out = 0
bb.buf = [ m₂ | | | | | ]
```

```
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 m₁

current line: 6

```
// 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 m₂

current line: 5

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

```
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?

current line: 6
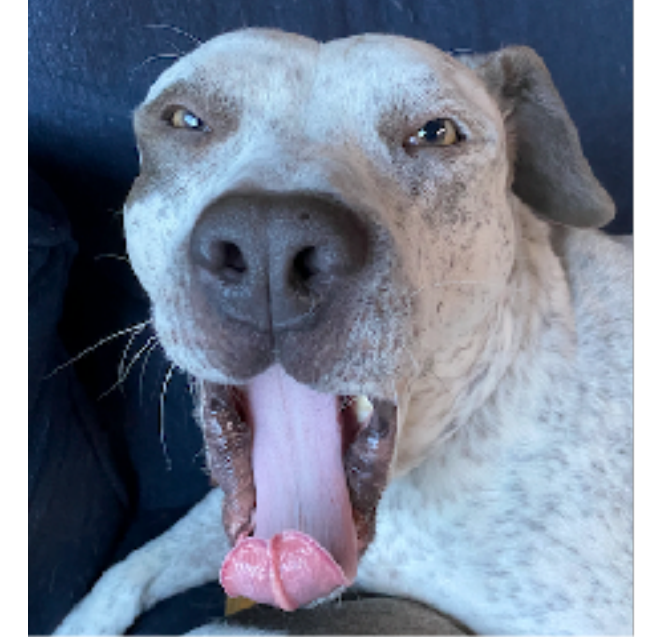
current line: 6

```
// 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 = 2
bb.out = 0
bb.buf = [ m₂ | | | | | ]
```

```
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 m₁

complete

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

junebug is trying to send message m₂

current line: 6

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

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
```
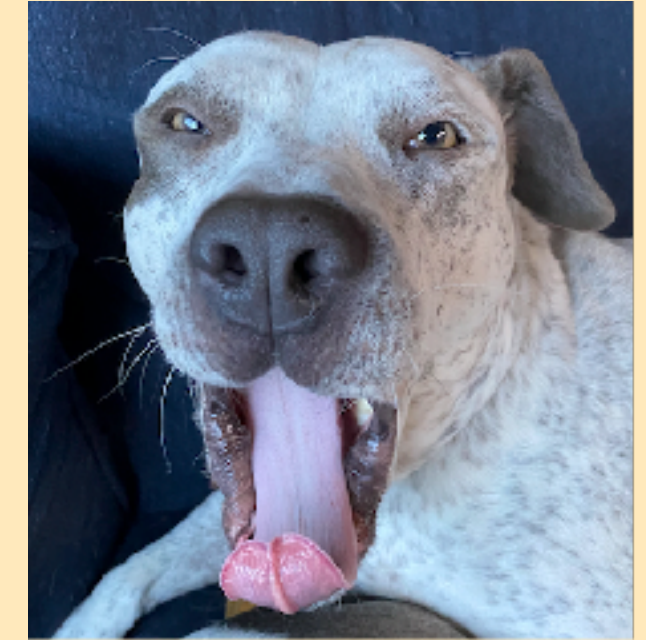
what happens when multiple programs try to send?

broccoli is trying to send message m_1

complete

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

junebug is trying to send message m_2

complete

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

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
```

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

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

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

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
```

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

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

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

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

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
```

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

```
// 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

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

the previous problem stemmed from the
fact that programs checked whether bb.buf

had space before acquiring bb.lock

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
bb.lock = lock intended to protect the bounded buffer
```

locks allow only one CPU to be inside a piece of code at a time.
programs can **acquire** and **release** a lock

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

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

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
bb.lock = lock intended to protect the bounded buffer
```

locks allow only one CPU to be inside a piece of code at a time.
programs can **acquire** and **release** a lock

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

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

```
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
```

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
```

problem: deadlock* if buffer is full

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
bb.lock = lock intended to protect the bounded buffer
```

locks allow only one CPU to be inside a piece of code at a time.
programs can **acquire** and **release** a lock

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

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

problem: deadlock* if buffer is full

*in 6.1800, we'll use "deadlock" to mean "two programs are waiting on each other, and neither can make progress until the other one does"

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
bb.lock = lock intended to protect the bounded buffer
```

locks allow only one CPU to be inside a piece of code at a time.
programs can **acquire** and **release** a lock

```
// send a message by placing it in bb
send(bb, message):
    acquire(bb.lock)
    while bb.in - bb.out >= N:
        release(bb.lock)
        acquire(bb.lock)
    bb.buf[bb.in mod N] <- message
    bb.in <- bb.in + 1
    release(bb.lock)
    return
```

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

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
bb.lock = lock intended to protect the bounded buffer
```

locks allow only one CPU to be inside a piece of code at a time.
programs can **acquire** and **release** a lock

```
// 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
```

give up the lock to allow other
programs to access the buffer

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

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
bb.lock = lock intended to protect the bounded buffer
```

locks allow only one CPU to be inside a piece of code at a time.
programs can **acquire** and **release** a lock

```
// send a message by placing it in bb
send(bb, message):
    acquire(bb.lock)
    while bb.in - bb.out >= N:
        release(bb.lock)
        acquire(bb.lock)
    bb.buf[bb.in mod N] <- message
    bb.in <- bb.in + 1
    release(bb.lock)
    return
```

give up the lock to allow other
programs to access the buffer

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

```
variables in use
bb = the bounded buffer
message = the message we're trying to send/receive
bb.in = total number of messages sent via this buffer
bb.out = total number of messages received via this buffer
bb.buf = the actual buffer for storing messages
N = total number of messages bb.buf can hold (assume N is large)
bb.lock = lock intended to protect the bounded buffer
```

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

locks create **atomic actions**. deciding what actions should be atomic, while balancing **performance**, is a challenge

locks create **atomic actions**. deciding what actions should be atomic, while balancing **performance**, is a challenge

```
// 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

```
// 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

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

problem: poor performance

variables in use
dir1 = the directory to move the file from
dir2 = the directory to move the file to
filename = the absolute path of the file
fs_lock = a global lock held whenever a program interacts with the filesystem

locks create **atomic actions**. deciding what actions should be atomic, while balancing **performance**, is a challenge

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

```
variables in use
dir1 = the directory to move the file from
dir2 = the directory to move the file to
filename = the absolute path of the file
dir1.lock, dir2.lock = directory-specific locks
```

locks create **atomic actions**. deciding what actions should be atomic, while balancing **performance**, is a challenge

```
// 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

```
// 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

```
// 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

Katrina LaCurts | lacurts@mit.edu | 6.1800 2025

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)
```

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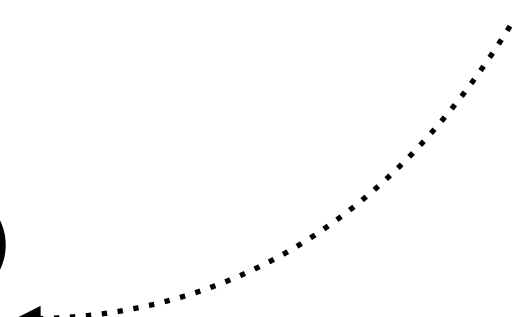
locks create **atomic actions**. deciding what actions should be atomic, while balancing **performance**, is a challenge

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

could release **dir1**.lock
here instead

```
variables in use
dir1 = the directory to move the file from
dir2 = the directory to move the file to
filename = the absolute path of the file
dir1.lock, dir2.lock = directory-specific locks
dir1.inum, dir2.inum = i-numbers for each directory
```

to believe that all of this works, we should understand the implementations of **acquire** and **release**

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

acquire(lock):

release(lock):

to believe that all of this works, we should understand the implementations of **acquire** and **release**

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

```
acquire(lock):
```

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

to believe that all of this works, we should understand the implementations of **acquire** and **release**

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

```
acquire(lock):
```

```
release(lock):
    lock = 0
```
lock is released; no program holds it

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

another program holds **lock**; it can't be acquired

```
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

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

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

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

```
variables in use
lock = the lock being acquired/released
```
Katrina LaCurts | lacurts@mit.edu | 6.1800 2025

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

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

```
variables in use
lock = the lock being acquired/released
```

```
// send a message by placing it in bb
send(bb, message):
    acquire(bb.lock)
    while bb.in - bb.out >= N:
        release(bb.lock)
        acquire(bb.lock)
    bb.buf[bb.in mod N] <- message
    bb.in <- bb.in + 1
    release(bb.lock)
    return
```

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

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

…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, Fall 2023

operating systems enforce modularity on a single machine

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

1. programs shouldn't be able to refer to (and corrupt) each others' **memory**   ┄┄┄▶ virtualize **memory**

2. programs should be able to **communicate** with each other   ┄┄┄▶ **bounded buffers**
   (virtualize communication links)

3. programs should be able to **share a CPU** without one program halting the progress of the others   ┄┄┄▶ assume one program per CPU
   (for today)

bounded buffers allow programs to communicate, completing the second step of enforcing modularity on a single machine. dealing with **concurrency** opens up a number of new challenges

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

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

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