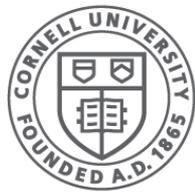


# Concurrent Programming

Robbert van Renesse



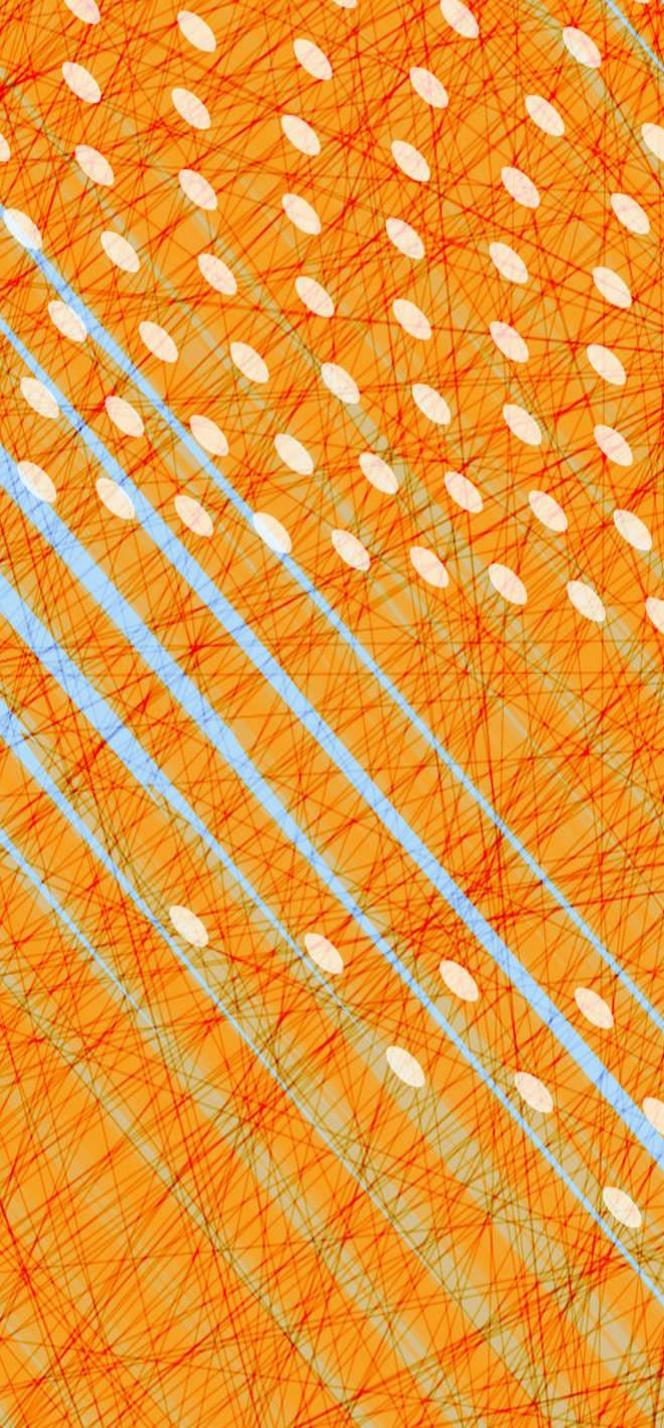
**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Concurrency Lectures Outline

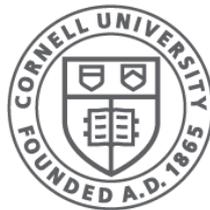
- What are the problems?
  - no determinism, no atomicity
- What is the solution?
  - some form of mutual exclusion
- How to specify concurrent problems?
  - atomic operations
- How to construct correct concurrent code?
  - behaviors
- How to test concurrent programs?
  - comparing behaviors

# Concurrency Lectures Outline

- How to build Concurrent Data Structures?
  - using locks
- How to wait for some condition?
  - using condition variables
- How to deal with deadlock?
  - prevention, avoidance, detection
- How to use barrier synchronization?
  - improve scalability
- How to make code interrupt-safe?
  - enabling/disabling interrupts



# The problems



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Concurrent Programming is Hard

## Why?

- Concurrent programs are *non-deterministic*
  - run them twice with same input, get two different answers
  - or worse, one time it works and the second time it fails
- Program statements are executed *non-atomically*
  - **x += 1** compiles to something like
    - **LOAD x**
    - **ADD 1**
    - **STORE x**
  - with concurrency, this leads to *non-deterministic interleavings*

# The problem with non-determinism

sequential

```
1 shared = True
2
3 def f(): assert shared
4 def g(): shared = False
5
6 f()
7 g()
```

concurrent

```
1 shared = True
2
3 def f(): assert shared
4 def g(): shared = False
5
6 spawn f()
7 spawn g()
```

*What will happen if you run each?*

# The problem with non-determinism

sequential

```
1 shared = True
2
3 def f(): assert shared
4 def g(): shared = False
5
6 f()
7 g()
```

#states: 2  
No issues

concurrent

```
1 shared = True
2
3 def f(): assert shared
4 def g(): shared = False
5
6 spawn f()
7 spawn g()
```

- Schedule thread T0: **init()**
  - Line 1: Initialize shared to True
  - **Thread terminated**
- Schedule thread T2: **g()**
  - Line 4: Set shared to False (was True)
  - **Thread terminated**
- Schedule thread T1: **f()**
  - Line 3: Harmony assertion failed

# The problem with non-atomicity

sequential

```
1 shared = 0
2
3 def f(): shared += 1
4
5 f()
6 f()
7
8 finally shared == 2
```

concurrent

```
1 shared = 0
2
3 def f(): shared += 1
4
5 spawn f()
6 spawn f()
7
8 finally shared == 2
```

*What will happen if you run each?*

# The problem with non-atomicity

sequential

```
1 shared = 0
2
3 def f(): shared += 1
4
5 f()
6 f()
7
8 finally shared == 2
```

#states: 2  
No issues

concurrent

```
1 shared = 0
2
3 def f(): shared += 1
4
5 spawn f()
6 spawn f()
7
8 finally shared == 2
```

Schedule thread T1: f()  
 Preempted in f()  
 about to store 1 into shared in line 3  
Schedule thread T2: f()  
 Line 3: Set shared to 1 (was 0)  
Schedule thread T1: f()  
 Line 3: Set shared to 1 (unchanged)  
Schedule thread T3: finally()  
 Line 8: Harmony assertion failed

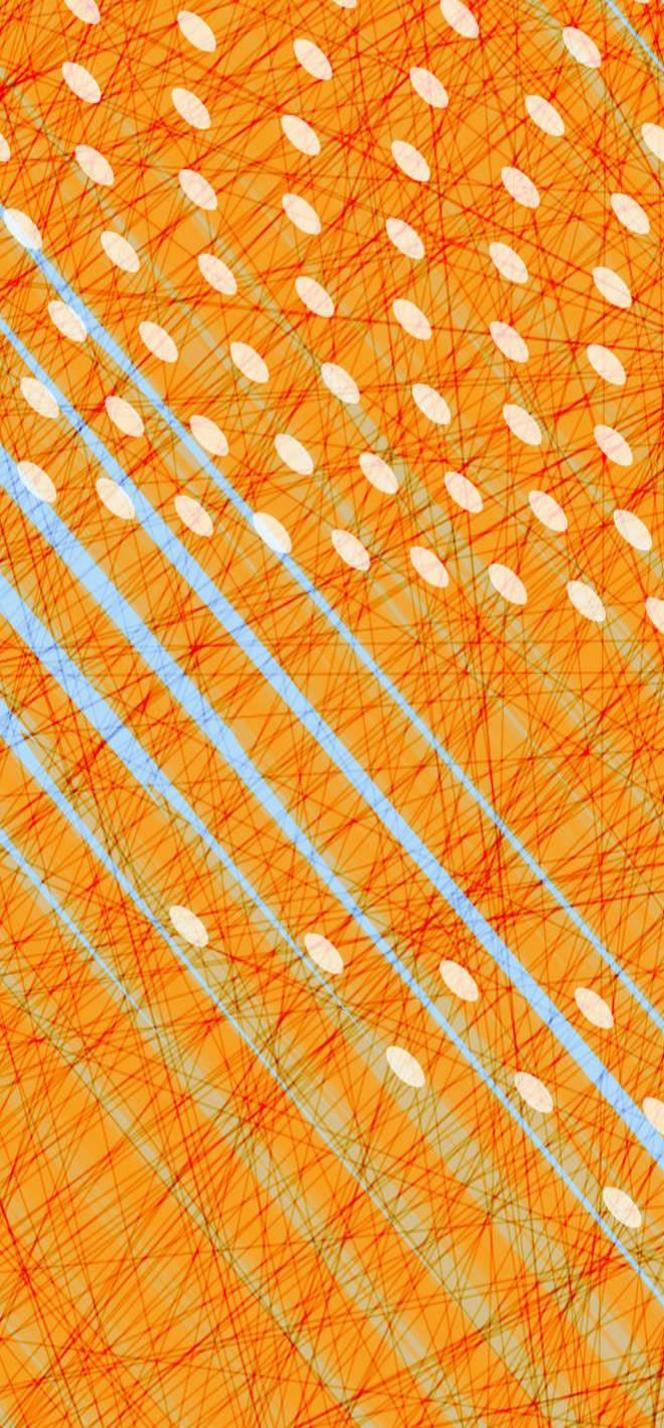
# Race Conditions

= *timing dependent error involving shared state*

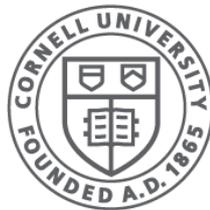
- A **schedule** is an interleaving of (i.e., total order on) the machine instructions executed by each thread
- Usually, many interleavings are possible
- A **race condition** occurs when at least one interleaving gives an undesirable result

# Race Conditions are Hard to Debug

- Number of possible interleavings is usually huge
- Bad interleavings, if they exist, may happen only rarely
  - Works 1000x  $\neq$  no race condition
- Timing dependent: small changes hide bugs
  - add print statement  $\rightarrow$  bug no longer seems to happen
- Harmony is designed to help identify such bugs
  - model checking!



# State Space and Model Checking



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Harmony Machine Code

```
def f():  
    shared += 1
```

compiler

1. **Frame f()**      Start a new stack frame
2. **Load *shared***      Push *shared* onto stack
3. **Push 1**      Push 1 onto stack
4. **2-ary +**      Add top two stack elements
5. **Store *shared***      Store top of stack into *shared*

# Harmony Virtual Machine *State*

Three parts:

1. code (never changes)
2. values of the shared variables
3. state of each of the running threads
  - PC and stack (aka *context*)

HVM state represents one vertex in a graph of states

# State space

→ thread 1 loads    .....▶ thread 1 stores  
→ thread 2 loads    .....▶ thread 2 stores

**Load** *shared*  
Push 1  
2-ary +  
**Store** *shared*

shared 0

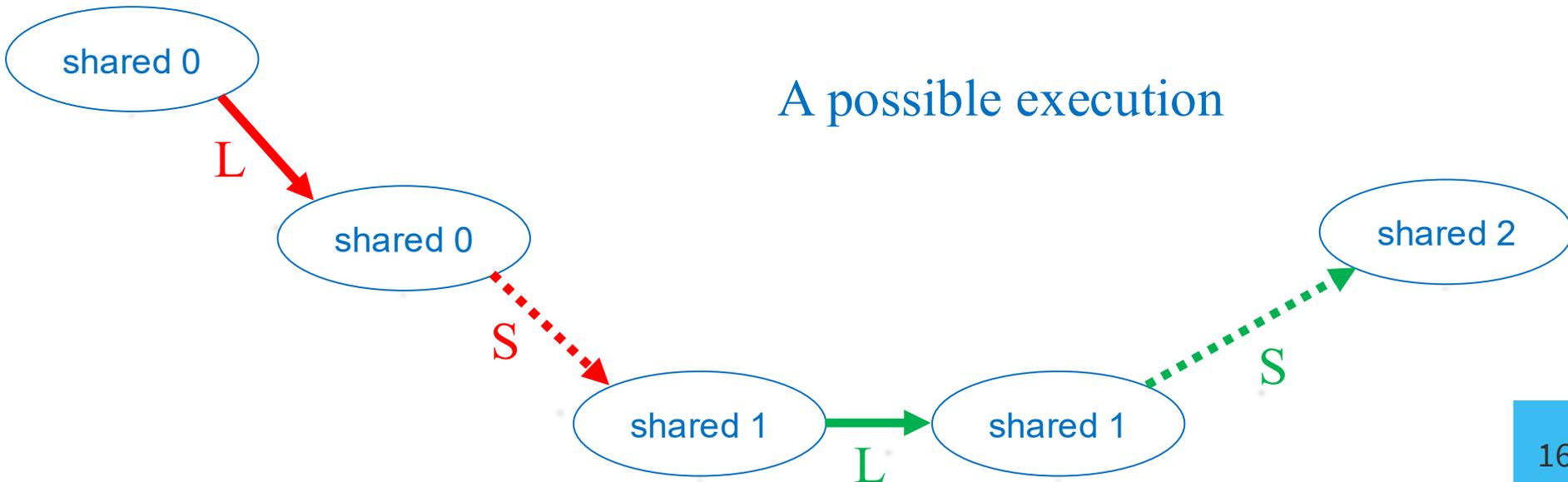
initial state

# State space

→ thread 1 loads    ····▶ thread 1 stores  
→ thread 2 loads    ····▶ thread 2 stores

**Load** *shared*  
Push 1  
2-ary +  
**Store** *shared*

A possible execution

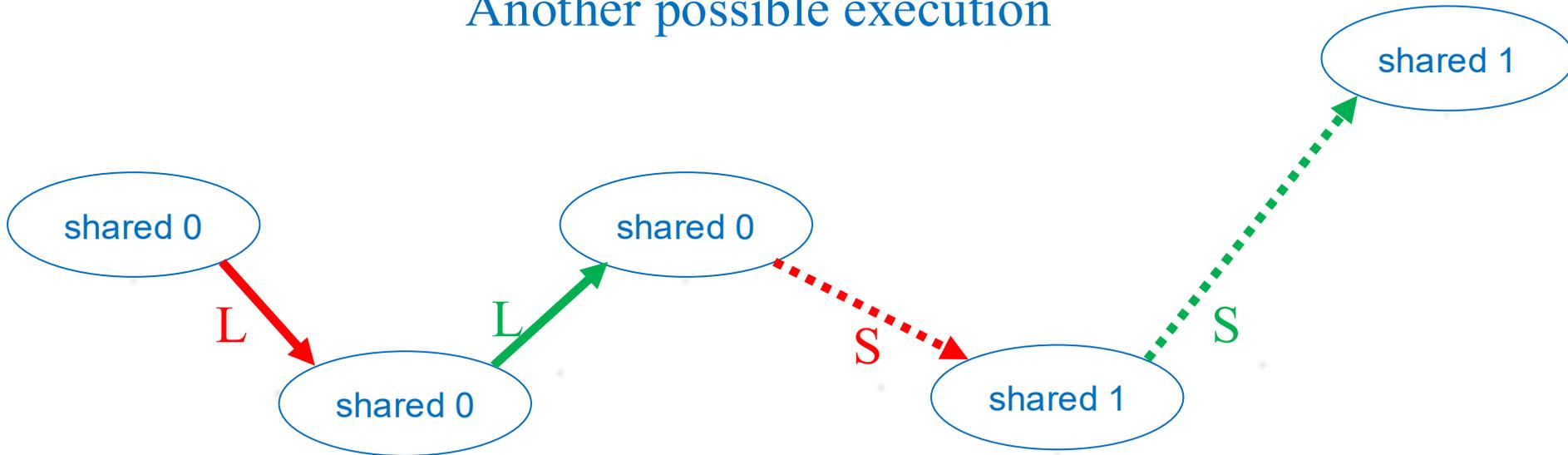


# State space

→ thread 1 loads    .....▶ thread 1 stores  
→ thread 2 loads    .....▶ thread 2 stores

**Load** *shared*  
Push 1  
2-ary +  
**Store** *shared*

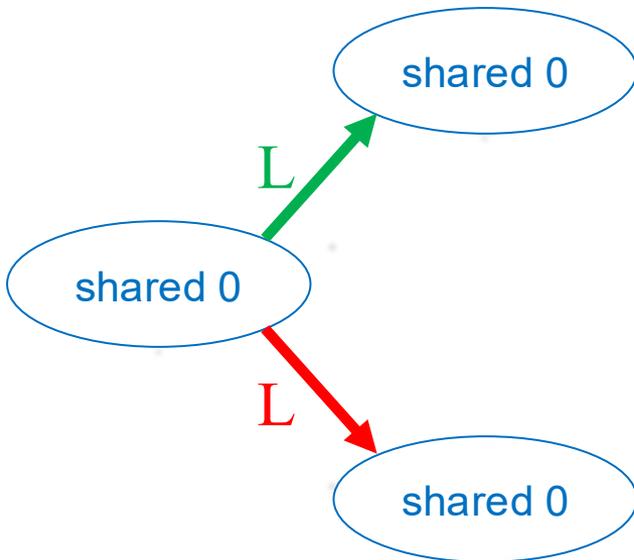
Another possible execution



# State space

→ thread 1 loads    ····▶ thread 1 stores  
→ thread 2 loads    ····▶ thread 2 stores

**Load** *shared*  
Push 1  
2-ary +  
**Store** *shared*

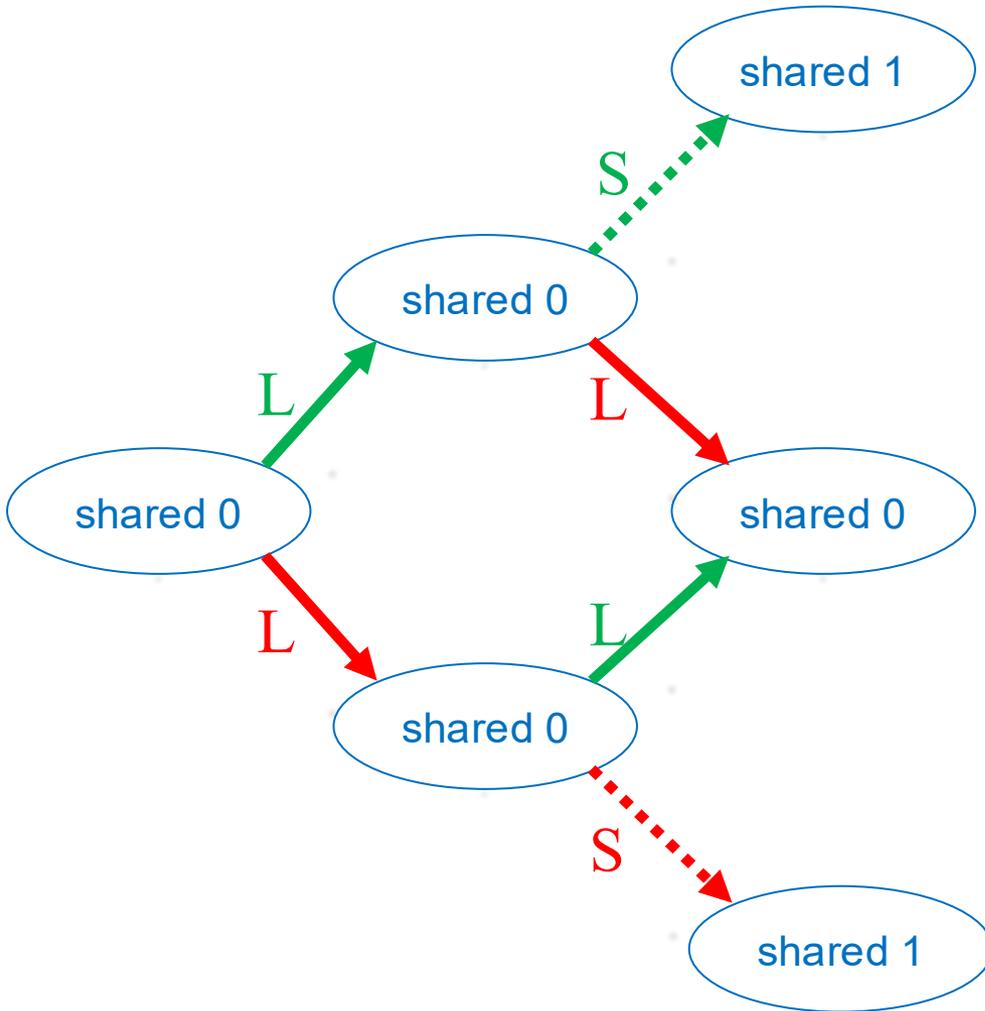


All possible states after one “step”

# State space

→ thread 1 loads    .....▶ thread 1 stores  
→ thread 2 loads    .....▶ thread 2 stores

**Load** *shared*  
Push 1  
2-ary +  
**Store** *shared*

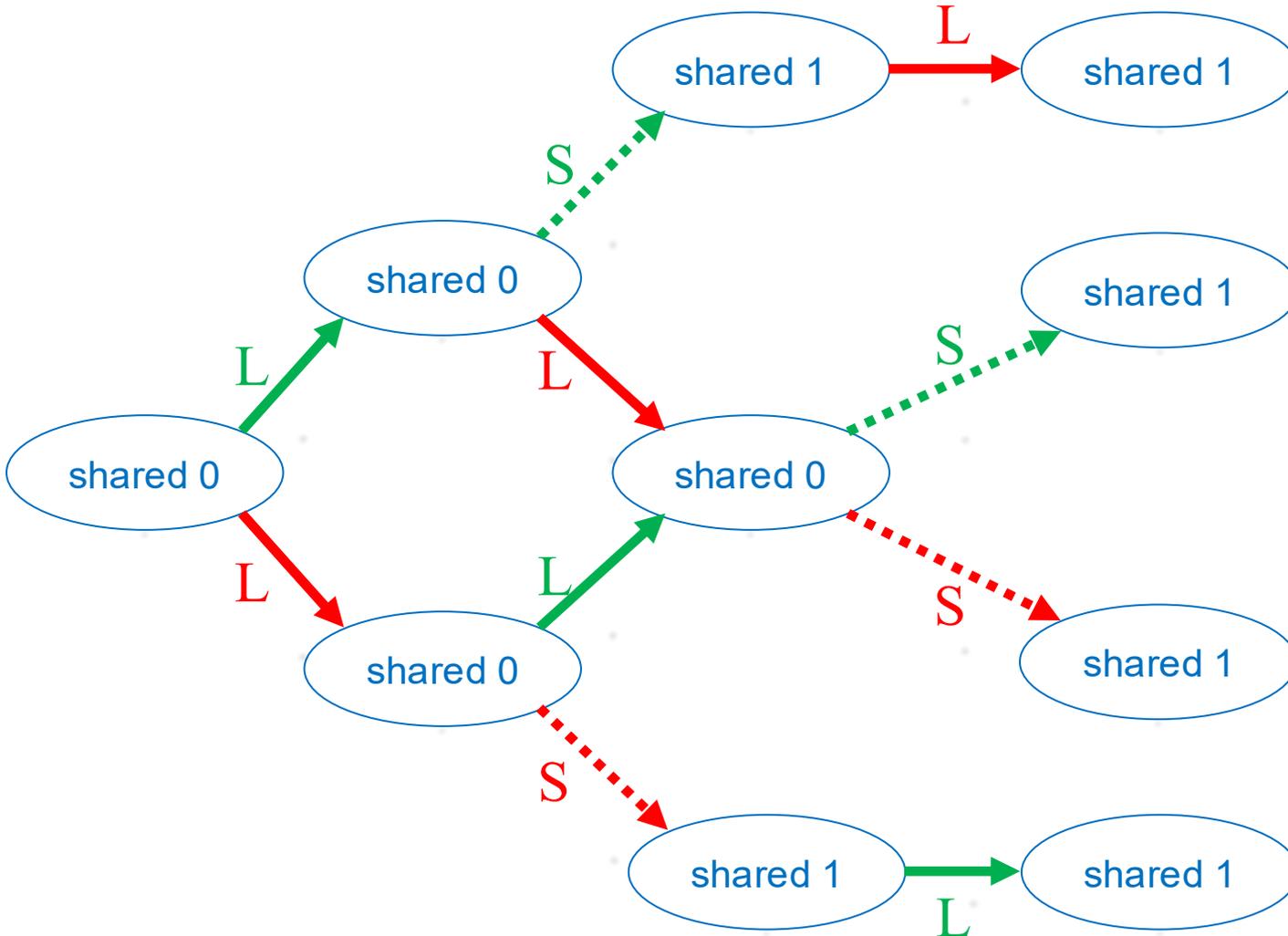


All possible states after two steps

# State space

**Load shared**  
Push 1  
2-ary +  
**Store shared**

→ thread 1 loads    ..... thread 1 stores  
→ thread 2 loads    ..... thread 2 stores

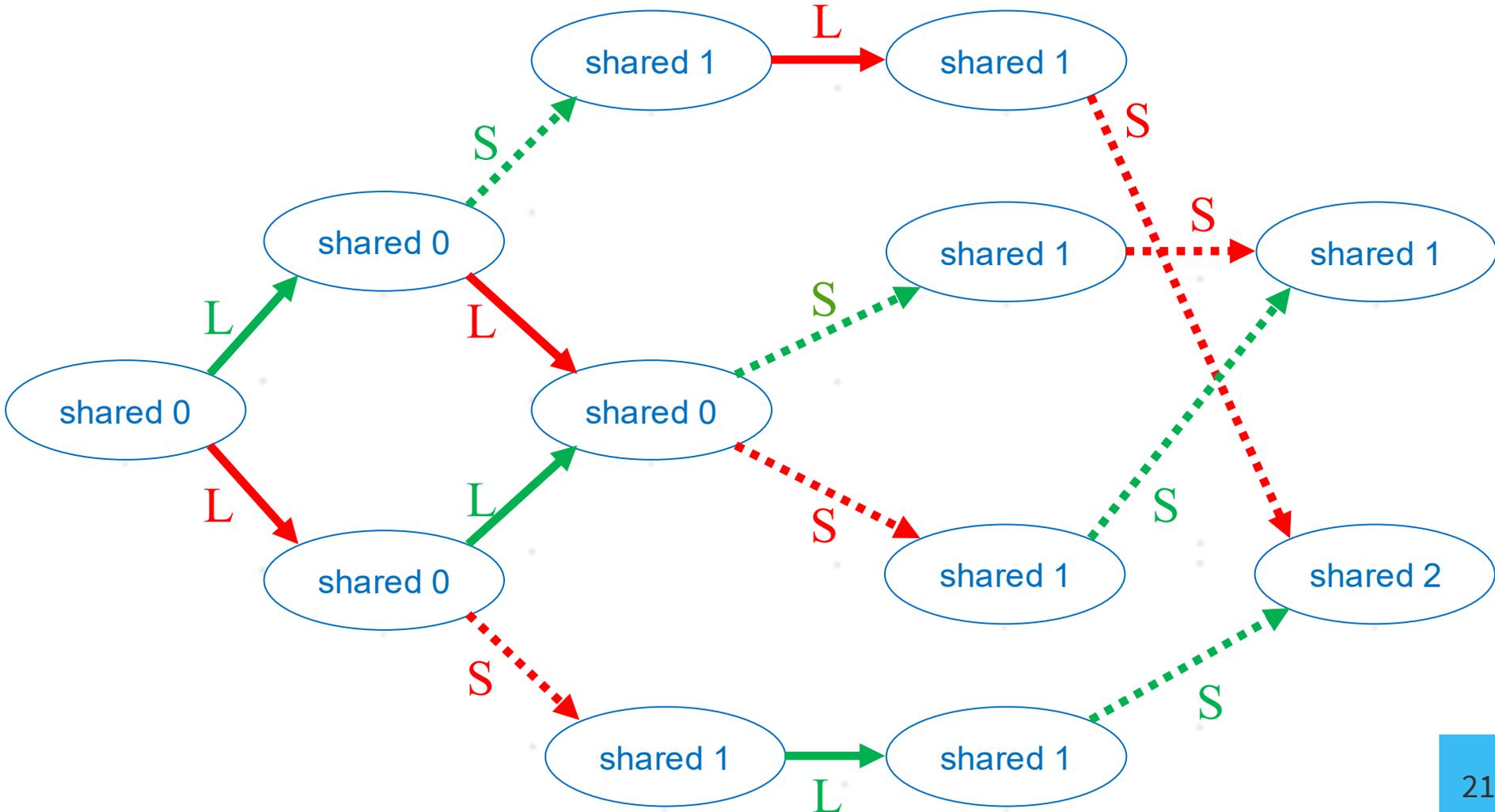


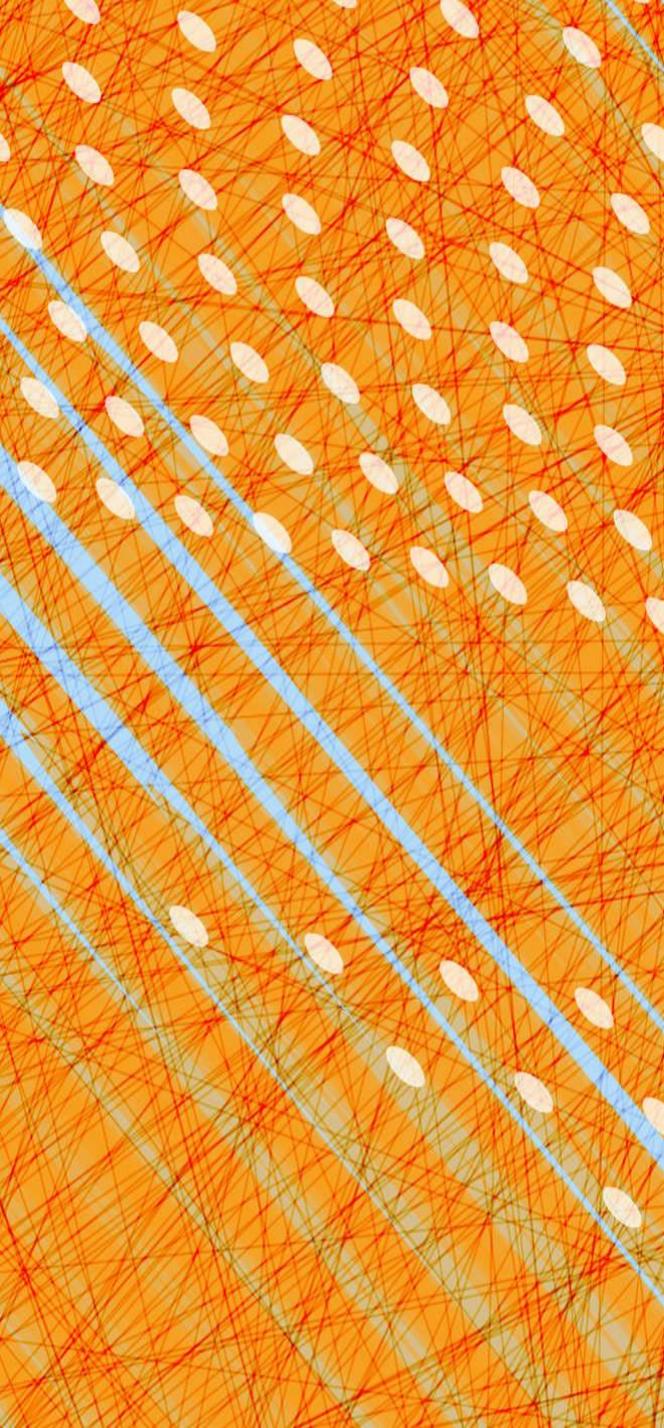
after three steps

# State space

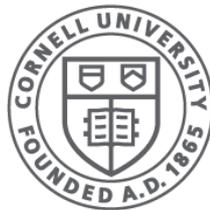
**Load shared**  
Push 1  
2-ary +  
**Store shared**

→ thread 1 loads    ..... thread 1 stores  
→ thread 2 loads    ..... thread 2 stores





# Harmony



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Harmony != Python

Harmony	Python
tries all possible executions	executes just one
<code>( ... ) == [ ... ] == ...</code>	<code>1 != [1] != (1)</code>
<code>1, == [1,] == (1,) != (1) == [1] == 1</code>	<code>[1,] == [1] != (1) == 1 != (1,)</code>
<code>f(1) == f 1 == f[1]</code>	<code>f 1</code> and <code>f[1]</code> are illegal (if <code>f</code> is method)
no <b>return</b> , <b>break</b> , <b>continue</b>	various flow control escapes
pointers	object-oriented
...	...

# I/O in Harmony?

- Input:
  - **choose** expression
    - $x = \mathbf{choose}(\{ 1, 2, 3 \})$
    - allows Harmony to know all possible inputs
  - **const** expression
    - **const**  $x = 3$
    - can be overridden with “-c x=4” flag to harmony
  - Output:
    - **print**  $x + y$
    - **assert**  $x + y < 10, (x, y)$

# I/O in Harmony?

- Input:
  - **choose** expression
    - `x = choose({ 1, 2, 3 })`
    - allows Harmony to generate different inputs
  - **const** expression
    - `c = const 4`
    - can be overridden with “-c x=4” flag to harmony
  - Output:
    - **print** `x + y`
    - **assert** `x + y < 10, (x, y)`

No `open()`, `read()`, or  
or `input()` statements

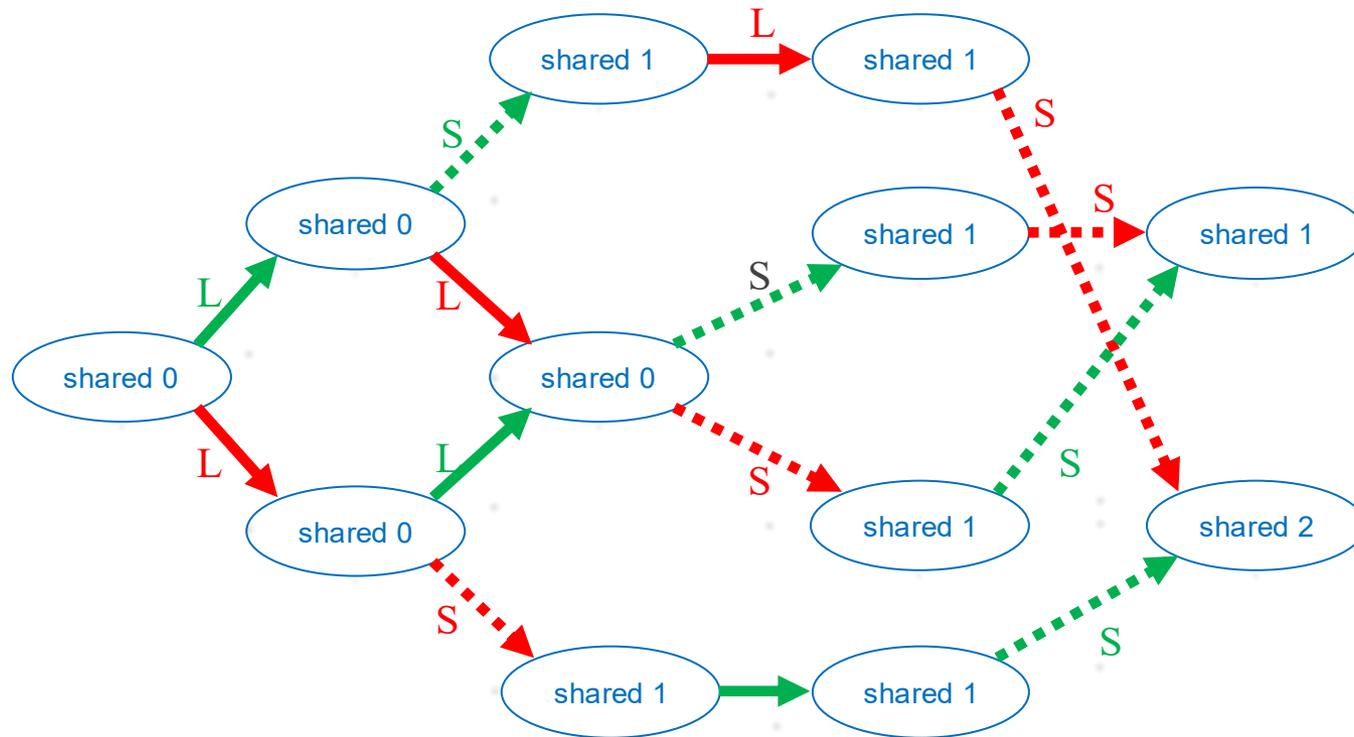
# Non-determinism in Harmony

Three sources:

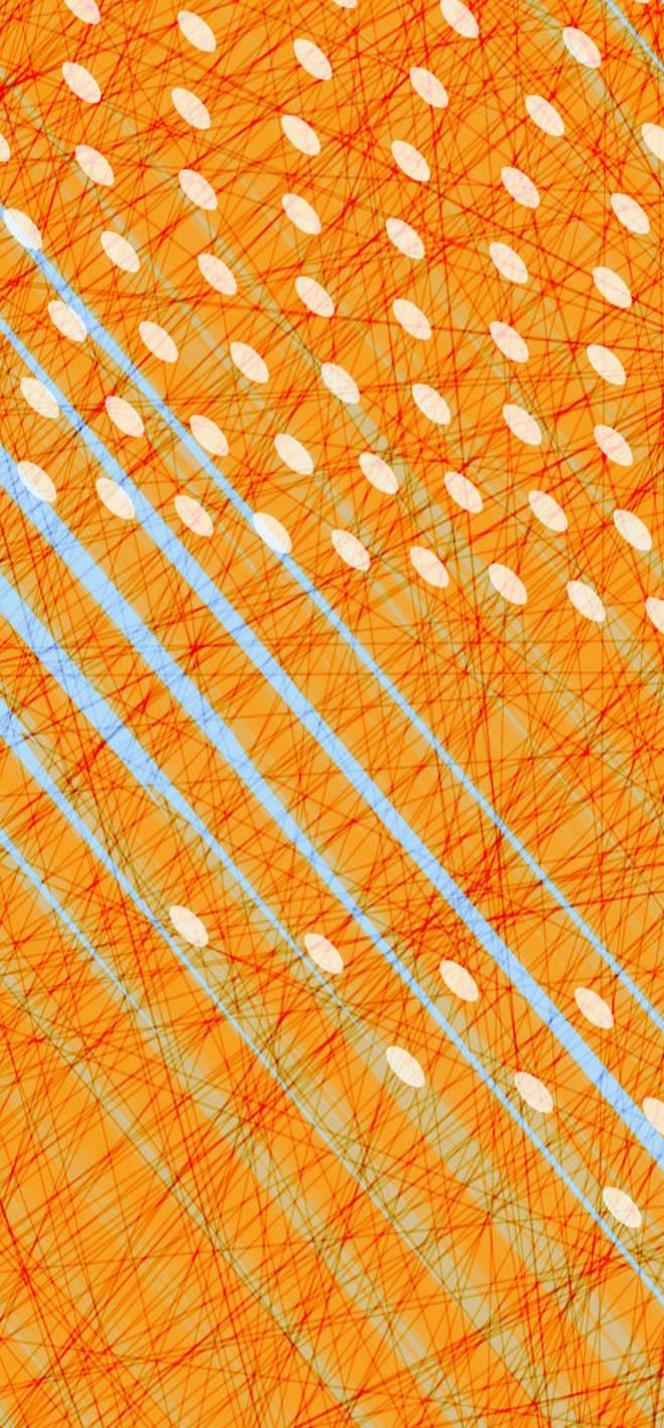
1. **choose** expressions
2. thread interleavings
3. interrupts



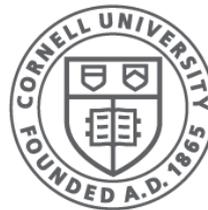
# Limitation: models must be finite!



- That is, there must be a finite number of states and edges.
- But models are allowed to have cycles.
- Executions are allowed to be unbounded!
- Harmony checks for *possibility* of termination.



# Critical Sections



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Back to our problem...

2 threads updating a shared variable

```
1 shared = 0
2
3 def f(): shared += 1
4
5 spawn f()
6 spawn f()
7
8 finally shared == 2
```

# Back to our problem...

2 threads updating a shared variable

```
1 shared = 0
2
3 def f(): shared += 1
4
5 spawn f()
6 spawn f()
7
8 finally shared == 2
```

“Critical Section”

# Back to our problem...

2 threads updating a shared variable

```
1 shared = 0
2
3 def f(): shared += 1
4
5 spawn f()
6 spawn f()
7
8 finally shared == 2
```

“Critical Section”

## Goals

**Mutual Exclusion:** 1 thread in a critical section at time

**Progress:** a thread can get in when there is no other thread

**Fairness:** equal chances of getting into CS

... in practice, fairness rarely guaranteed or needed

# Mutual Exclusion and Progress

Need both:

- either one is trivial to achieve by itself

# Specifying Critical Sections in Harmony

- How do we check mutual exclusion?
- How do we check progress?

# Specifying Critical Sections in Harmony

```
1 # number of threads in the critical section
2 in_cs = 0
3 invariant in_cs in { 0, 1 }
4
5 def thread():
6     while choose { False, True }:
7         # Enter critical section
8         atomically in_cs += 1
9
10        # Critical section is here
11        pass
12
13        # Exit critical section
14        atomically in_cs -= 1
15
16 spawn thread()
17 spawn thread()
```

- How do we check mutual exclusion?
- How do we check progress?

# Specifying Critical Sections in Harmony

```
1 # number of threads in the critical section
2 in_cs = 0
3 invariant in_cs in { 0, 1 }
4
5 def thread():
6     while choose { False, True }:
7         # Enter critical section
8         atomically in_cs += 1
9
10        # Critical section is here
11        pass
12
13        # Exit critical section
14        atomically in_cs -= 1
15
16 spawn thread()
17 spawn thread()
```

mutual exclusion

- How do we check mutual exclusion?
- How do we check progress?

# Specifying Critical Sections in Harmony

```
1 # number of threads in the critical section
2 in_cs = 0
3 invariant in_cs in { 0, 1 }
4
5 def thread():
6     while choose { False, True }:
7         # Enter critical section
8         atomically in_cs += 1
9
10        # Critical section is here
11        pass
12
13        # Exit critical section
14        atomically in_cs -= 1
15
16 spawn thread()
17 spawn thread()
```

mutual exclusion

do zero or more times

- How do we check mutual exclusion?
- How do we check progress?

# Specifying Critical Sections in Harmony

```
1 # number of threads in the critical section
2 in_cs = 0
3 invariant in_cs in { 0, 1 }
4
5 def thread():
6     while choose { False, True }:
7         # Enter critical section
8         atomically in_cs += 1
9
10        # Critical section is here
11        pass
12
13        # Exit critical section
14        atomically in_cs -= 1
15
16 spawn thread()
17 spawn thread()
```

mutual exclusion

do zero or more times

increment in\_cs

- How do we check mutual exclusion?
- How do we check progress?

# Specifying Critical Sections in Harmony

```
1 # number of threads in the critical section
2 in_cs = 0
3 invariant in_cs in { 0, 1 }
4
5 def thread():
6     while choose { False, True }:
7         # Enter critical section
8         atomically in_cs += 1
9
10        # Critical section is here
11        pass
12
13        # Exit critical section
14        atomically in_cs -= 1
15
16 spawn thread()
17 spawn thread()
```

mutual exclusion

do zero or more times

increment in\_cs

execute critical section

- How do we check mutual exclusion?
- How do we check progress?

# Specifying Critical Sections in Harmony

```
1 # number of threads in the critical section
2 in_cs = 0
3 invariant in_cs in { 0, 1 }
4
5 def thread():
6     while choose { False, True }:
7         # Enter critical section
8         atomically in_cs += 1
9
10        # Critical section is here
11        pass
12
13        # Exit critical section
14        atomically in_cs -= 1
15
16 spawn thread()
17 spawn thread()
```

mutual exclusion

do zero or more times

increment in\_cs

execute critical section

decrement in\_cs

- How do we check mutual exclusion?
- How do we check progress?

# Specifying Critical Sections in Harmony

```
1 # number of threads in the critical section
2 in_cs = 0
3 invariant in_cs in { 0, 1 }
4
5 def thread():
6     while choose { False, True }:
7         # Enter critical section
8         atomically in_cs += 1
9
10        # Critical section is here
11        pass
12
13        # Exit critical section
14        atomically in_cs -= 1
15
16 spawn thread()
17 spawn thread()
```

mutual exclusion

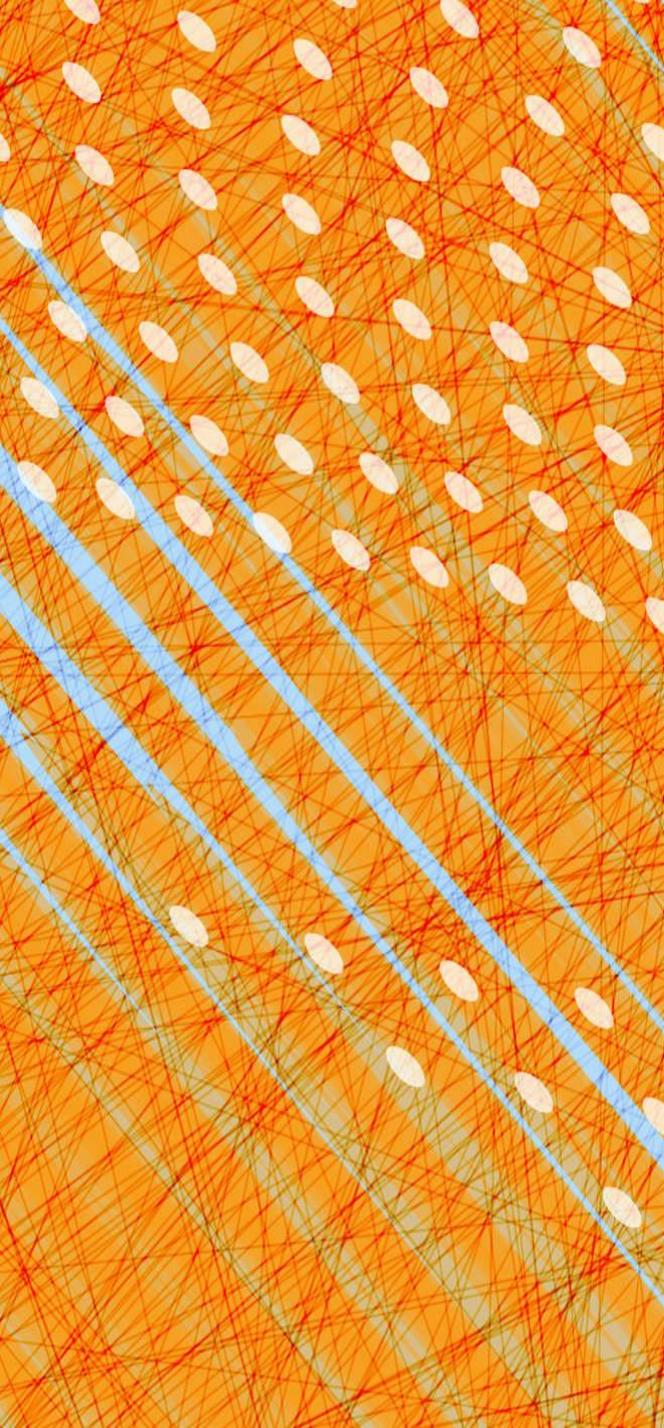
do zero or more times

increment in\_cs

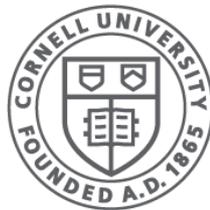
execute critical section

decrement in\_cs

**Progress:** Harmony checks that all thread *can* terminate



# Building a lock is hard



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# *Specification vs implementation*

- Spec is fine, but we'll need an implementation too
- Sounds like we need a *lock*
- The question is:

*How does one build a lock?*

# First attempt: a naïve lock

```
1  in_cs = 0
2  invariant in_cs in { 0, 1 }
3
4  lockTaken = False
5
6  def thread(self):
7      while choose({ False, True }):
8          # Enter critical section
9          await not lockTaken
10         lockTaken = True
11
12         atomically in_cs += 1
13         # Critical section
14         atomically in_cs -= 1
15
16         # Leave critical section
17         lockTaken = False
18
19  spawn thread(0)
20  spawn thread(1)
```

# First attempt: a naïve lock

```
1  in_cs = 0
2  invariant in_cs in { 0, 1 }
3
4  lockTaken = False
5
6  def thread(self):
7      while choose({ False, True }):
8          # Enter critical section
9          await not lockTaken
10         lockTaken = True
11
12         atomically in_cs += 1
13         # Critical section
14         atomically in_cs -= 1
15
16         # Leave critical section
17         lockTaken = False
18
19  spawn thread(0)
20  spawn thread(1)
```



wait till lock is free, then take it

# First attempt: a naïve lock

```
1 in_cs = 0
2 invariant in_cs in { 0, 1 }
3
4 lockTaken = False
5
6 def thread(self):
7     while choose({ False, True }):
8         # Enter critical section
9         await not lockTaken
10        lockTaken = True
11
12        atomically in_cs += 1
13        # Critical section
14        atomically in_cs -= 1
15
16        # Leave critical section
17        lockTaken = False
18
19 spawn thread(0)
20 spawn thread(1)
```

- Schedule thread T0: `init()`
  - Line 1: Initialize `in_cs` to 0
  - Line 4: Initialize `lockTaken` to False
  - **Thread terminated**
- Schedule thread T3: `thread(1)`
  - Line 7: Choose True
  - Preempted in `thread(1)` about to store True into `lockTaken` in line 10
- Schedule thread T2: `thread(0)`
  - Line 7: Choose True
  - Line 10: Set `lockTaken` to True (was False)
  - Line 12: Set `in_cs` to 1 (was 0)
  - Preempted in `thread(0)` about to execute atomic section in line 14
- Schedule thread T3: `thread(1)`
  - Line 10: Set `lockTaken` to True (unchanged)
  - Line 12: Set `in_cs` to 2 (was 1)
  - Preempted in `thread(1)` about to execute atomic section in line 14
- Schedule thread T1: `invariant()`
  - Line 2: Harmony assertion failed

# Second attempt: *flags*

```
1 in_cs = 0
2 invariant in_cs in { 0, 1 }
3
4 flags = [ False, False ]
5
6 def thread(self):
7     while choose({ False, True }):
8         # Enter critical section
9         flags[self] = True
10        await not flags[1 - self]
11
12        atomically in_cs += 1
13        # Critical section
14        atomically in_cs -= 1
15
16        # Leave critical section
17        flags[self] = False
18
19 spawn thread(0)
20 spawn thread(1)
```

# Second attempt: *flags*

```
1 in_cs = 0
2 invariant in_cs in { 0, 1 }
3
4 flags = [ False, False ]
5
6 def thread(self):
7     while choose({ False, True }):
8         # Enter critical section
9         flags[self] = True
10        await not flags[1 - self]
11
12        atomically in_cs += 1
13        # Critical section
14        atomically in_cs -= 1
15
16        # Leave critical section
17        flags[self] = False
18
19 spawn thread(0)
20 spawn thread(1)
```



show intent to enter critical section

# Second attempt: *flags*

```
1 in_cs = 0
2 invariant in_cs in { 0, 1 }
3
4 flags = [ False, False ]
5
6 def thread(self):
7     while choose({ False, True }):
8         # Enter critical section
9         flags[self] = True
10        await not flags[1 - self]
11
12        atomically in_cs += 1
13        # Critical section
14        atomically in_cs -= 1
15
16        # Leave critical section
17        flags[self] = False
18
19 spawn thread(0)
20 spawn thread(1)
```



show intent to enter critical section



wait until there's no one else

# Second attempt: *flags*

```
1 in_cs = 0
2 invariant in_cs in { 0, 1 }
3
4 flags = [ False, False ]
5
6 def thread(self):
7     while choose({ False, True }):
8         # Enter critical section
9         flags[self] = True
10        await not flags[1 - self]
11
12        atomically in_cs += 1
13        # Critical section
14        atomically in_cs -= 1
15
16        # Leave critical section
17        flags[self] = False
18
19 spawn thread(0)
20 spawn thread(1)
```

## Summary: some execution cannot terminate

Here is a summary of an execution that exhibits the issue:

- Schedule thread T0: **init()**
  - Line 1: Initialize in\_cs to 0
  - Line 4: Initialize flags to [ False, False ]
  - **Thread terminated**
- Schedule thread T1: thread(0)
  - Line 7: Choose True
  - Line 9: Set flags[0] to True (was False)
  - Preempted in thread(0) about to load variable flags[1] in line 10
- Schedule thread T2: thread(1)
  - Line 7: Choose True
  - Line 9: Set flags[1] to True (was False)
  - Preempted in thread(1) about to load variable flags[0] in line 10

**Final state** (all threads have terminated or are blocked):

- Threads:
  - T1: (blocked) thread(0)
    - about to load variable flags[1] in line 10
  - T2: (blocked) thread(1)
    - about to load variable flags[0] in line 10

# Third attempt: *turn* variable

```
1  in_cs = 0
2  invariant in_cs in { 0, 1 }
3
4  turn = 0
5
6  def thread(self):
7      while choose({ False, True }):
8          # Enter critical section
9          turn = 1 - self
10         await turn == self
11
12         atomically in_cs += 1
13         # Critical section
14         atomically in_cs -= 1
15
16         # Leave critical section
17
18     spawn thread(0)
19     spawn thread(1)
```

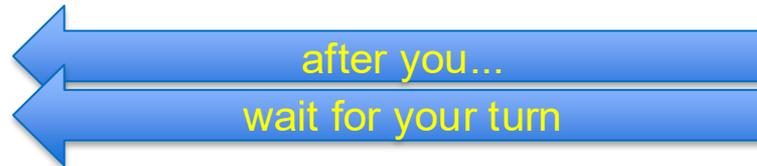
# Third attempt: *turn* variable

```
1 in_cs = 0
2 invariant in_cs in { 0, 1 }
3
4 turn = 0
5
6 def thread(self):
7     while choose({ False, True }):
8         # Enter critical section
9         turn = 1 - self
10        await turn == self
11
12        atomically in_cs += 1
13        # Critical section
14        atomically in_cs -= 1
15
16        # Leave critical section
17
18 spawn thread(0)
19 spawn thread(1)
```



# Third attempt: *turn* variable

```
1 in_cs = 0
2 invariant in_cs in { 0, 1 }
3
4 turn = 0
5
6 def thread(self):
7     while choose({ False, True }):
8         # Enter critical section
9         turn = 1 - self
10        await turn == self
11
12        atomically in_cs += 1
13        # Critical section
14        atomically in_cs -= 1
15
16        # Leave critical section
17
18 spawn thread(0)
19 spawn thread(1)
```



# Third attempt: *turn* variable

```
1 in_cs = 0
2 invariant in_cs in { 0, 1 }
3
4 turn = 0
5
6 def thread(self):
7     while choose({ False, True }):
8         # Enter critical section
9         turn = 1 - self
10        await turn == self
11
12        atomically in_cs += 1
13        # Critical section
14        atomically in_cs -= 1
15
16        # Leave critical section
17
18 spawn thread(0)
19 spawn thread(1)
```

## Summary: some execution cannot terminate

Here is a summary of an execution that exhibits the issue:

- Schedule thread T0: **init()**
  - Line 1: Initialize `in_cs` to 0
  - Line 4: Initialize `turn` to 0
  - **Thread terminated**
- Schedule thread T2: `thread(1)`
  - Line 7: Choose False
  - **Thread terminated**
- Schedule thread T1: `thread(0)`
  - Line 7: Choose True
  - Line 9: Set `turn` to 1 (was 0)
  - Preempted in `thread(0)` about to load variable `turn` in line 10

**Final state** (all threads have terminated or are blocked):

- Threads:
  - T1: (blocked) `thread(0)`
    - about to load variable `turn` in line 10
  - T2: (terminated) `thread(1)`

# Peterson's Algorithm: *flags & turn*

```
1  in_cs = 0
2  invariant in_cs in { 0, 1 }
3
4  sequential flags, turn
5  flags = [ False, False ]
6  turn = choose({0, 1})
7
8  def thread(self):
9      while choose({ False, True }):
10         # Enter critical section
11         flags[self] = True
12         turn = 1 - self
13         await (not flags[1 - self]) or (turn == self)
14
15         atomically in_cs += 1
16         # Critical section
17         atomically in_cs -= 1
18
19         # Leave critical section
20         flags[self] = False
21
22  spawn thread(0)
23  spawn thread(1)
```

# Peterson's Algorithm: *flags & turn*

```
1  in_cs = 0
2  invariant in_cs in { 0, 1 }
3
4  sequential flags, turn
5  flags = [ False, False ]
6  turn = choose({0, 1})
7
8  def thread(self):
9      while choose({ False, True }):
10         # Enter critical section
11         flags[self] = True
12         turn = 1 - self
13         await (not flags[1 - self]) or (turn == self)
14
15         atomically in_cs += 1
16         # Critical section
17         atomically in_cs -= 1
18
19         # Leave critical section
20         flags[self] = False
21
22  spawn thread(0)
23  spawn thread(1)
```



in critical section

# Peterson's Algorithm: *flags & turn*

```
1 in_cs = 0
2 invariant in_cs in { 0, 1 }
3
4 sequential flags, turn
5 flags = [ False, False ]
6 turn = choose({0, 1})
7
8 def thread(self):
9     while choose({ False, True }):
10        # Enter critical section
11        flags[self] = True
12        turn = 1 - self
13        await (not flags[1 - self]) or (turn == self)
14
15        atomically in_cs += 1
16        # Critical section
17        atomically in_cs -= 1
18
19        # Leave critical section
20        flags[self] = False
21
22 spawn thread(0)
23 spawn thread(1)
```

load and store instructions are atomic

in critical section

# Peterson's Algorithm: *flags & turn*

```
1 in_cs = 0
2 invariant in_cs in { 0, 1 }
3
4 sequential flags, turn
5 flags = [ False, False ]
6 turn = choose({0, 1})
7
8 def thread(self):
9     while choose({ False, True }):
10         # Enter critical section
11         flags[self] = True
12         turn = 1 - self
13         await (not flags[1 - self]) or (turn == self)
14
15         atomically in_cs += 1
16         # Critical section
17         atomically in_cs -= 1
18
19         # Leave critical section
20         flags[self] = False
21
22 spawn thread(0)
23 spawn thread(1)
```

load and store instructions are atomic

uses *flags* and *turn* variable (3 bits total)

in critical section

# Peterson's Algorithm: *flags & turn*

```
1 in_cs = 0
2 invariant in_cs in { 0, 1 }
3
4 sequential flags, turn
5 flags = [ False, False ]
6 turn = choose({0, 1})
7
8 def thread(self):
9     while choose({ False, True }):
10        # Enter critical section
11        flags[self] = True
12        turn = 1 - self
13        await (not flags[1 - self]) or (turn == self)
14
15        atomically in_cs += 1
16        # Critical section
17        atomically in_cs -= 1
18
19        # Leave critical section
20        flags[self] = False
21
22 spawn thread(0)
23 spawn thread(1)
```

load and store instructions are atomic

uses flags and turn variable (3 bits total)

first indicate intention to enter critical section

in critical section

no longer in critical section

# Peterson's Algorithm: *flags & turn*

```
1 in_cs = 0
2 invariant in_cs in { 0, 1 }
3
4 sequential flags, turn
5 flags = [ False, False ]
6 turn = choose({0, 1})
7
8 def thread(self):
9     while choose({ False, True }):
10        # Enter critical section
11        flags[self] = True
12        turn = 1 - self
13        await (not flags[1 - self]) or (turn == self)
14
15        atomically in_cs += 1
16        # Critical section
17        atomically in_cs -= 1
18
19        # Leave critical section
20        flags[self] = False
21
22 spawn thread(0)
23 spawn thread(1)
```

load and store instructions are atomic

uses flags and turn variable (3 bits total)

first indicate intention to enter critical section  
also give other thread a turn first

in critical section

no longer in critical section

# Peterson's Algorithm: *flags & turn*

```
1 in_cs = 0
2 invariant in_cs in { 0, 1 }
3
4 sequential flags, turn
5 flags = [ False, False ]
6 turn = choose({0, 1})
7
8 def thread(self):
9     while choose({ False, True }):
10        # Enter critical section
11        flags[self] = True
12        turn = 1 - self
13        await (not flags[1 - self]) or (turn == self)
14
15        atomically in_cs += 1
16        # Critical section
17        atomically in_cs -= 1
18
19        # Leave critical section
20        flags[self] = False
21
22 spawn thread(0)
23 spawn thread(1)
```

load and store instructions are atomic

uses flags and turn variable (3 bits total)

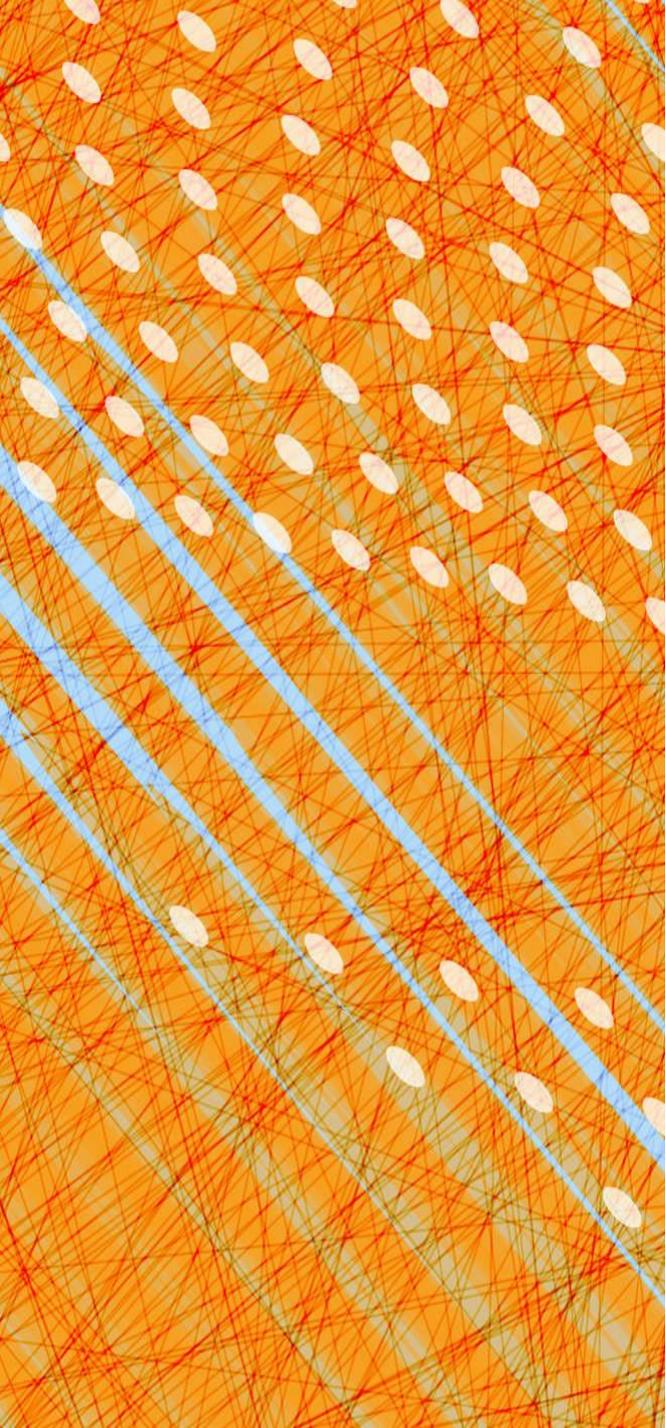
first indicate intention to enter critical section

also give other thread a turn first

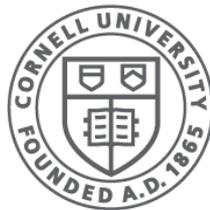
wait for one of either conditions

in critical section

no longer in critical section



# Proving a concurrent program correct



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

So, we proved Peterson's Algorithm correct by brute force, enumerating all possible executions. We now know *that* it works.

*But how does one prove it by deduction?  
so one understands *why* it works...*

# What and how?

- Need to show that, for any execution, all states reached satisfy mutual exclusion
  - in other words, mutual exclusion is *invariant*  
*invariant = predicate that holds in every reachable state*

# What is an invariant?

*A property that holds in all reachable states*

(and possibly in some unreachable states as well)

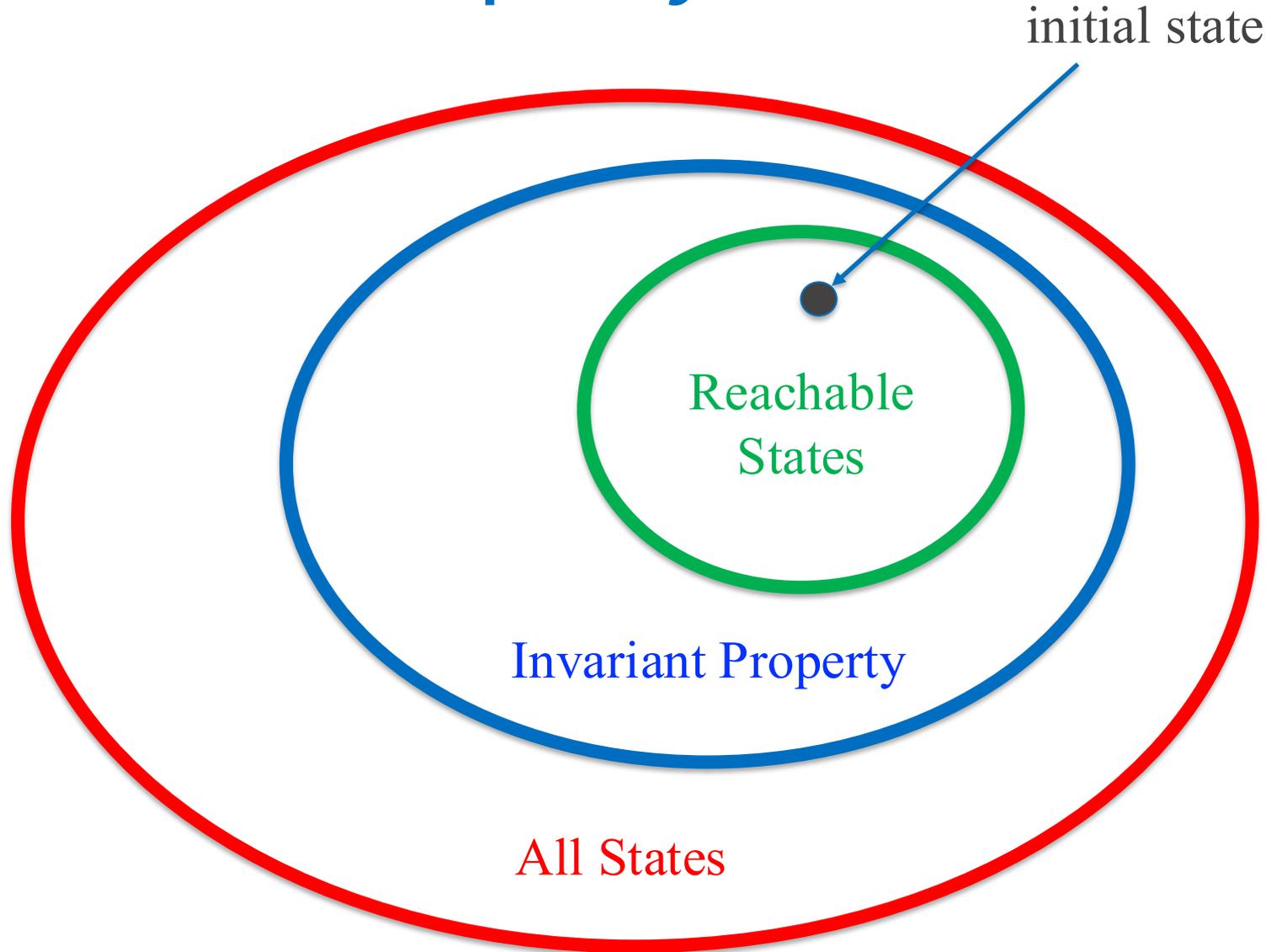
What is a property?

*A property is a set of states*

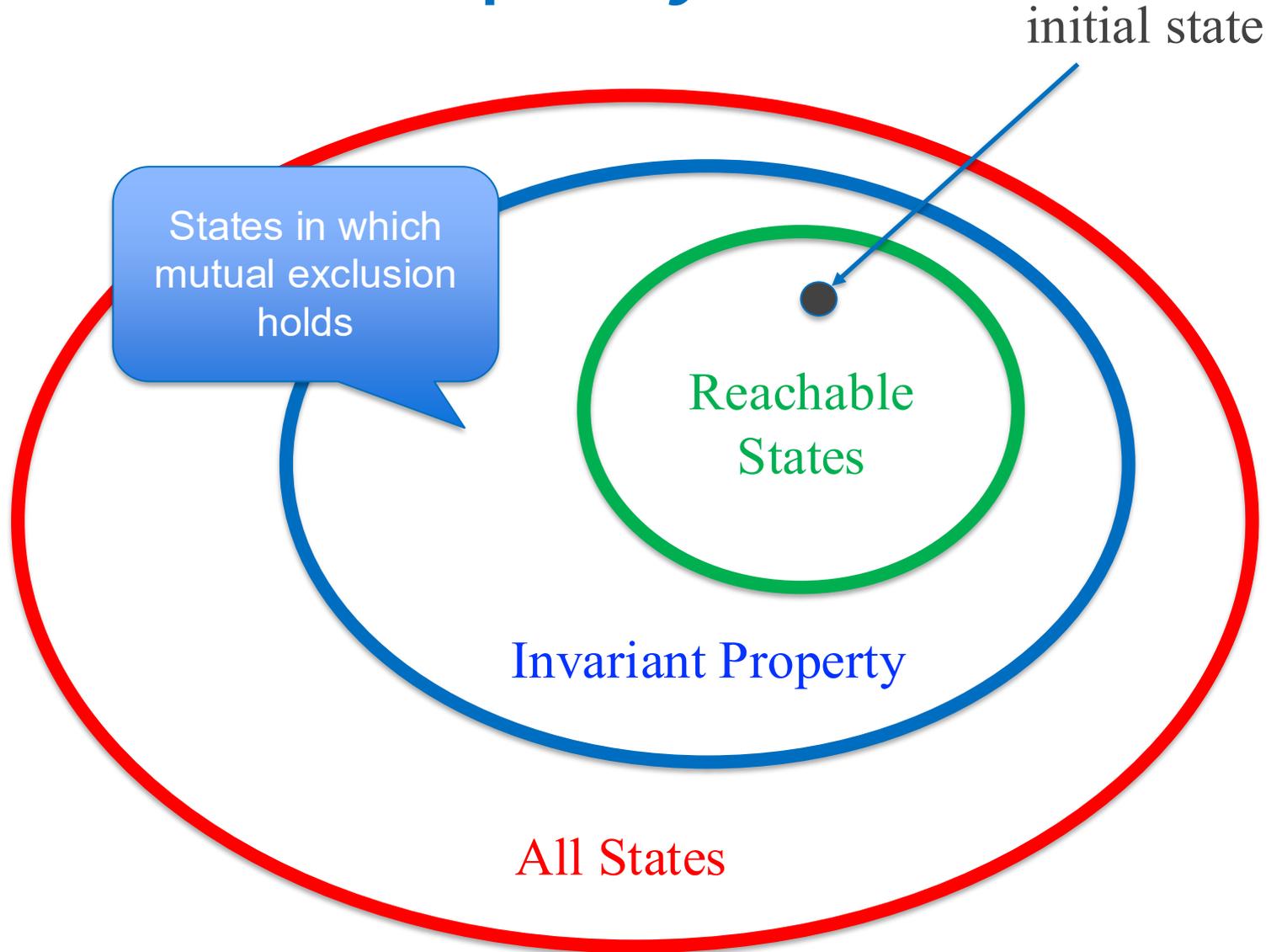
often succinctly described using a predicate

(all states that satisfy the predicate and no others)

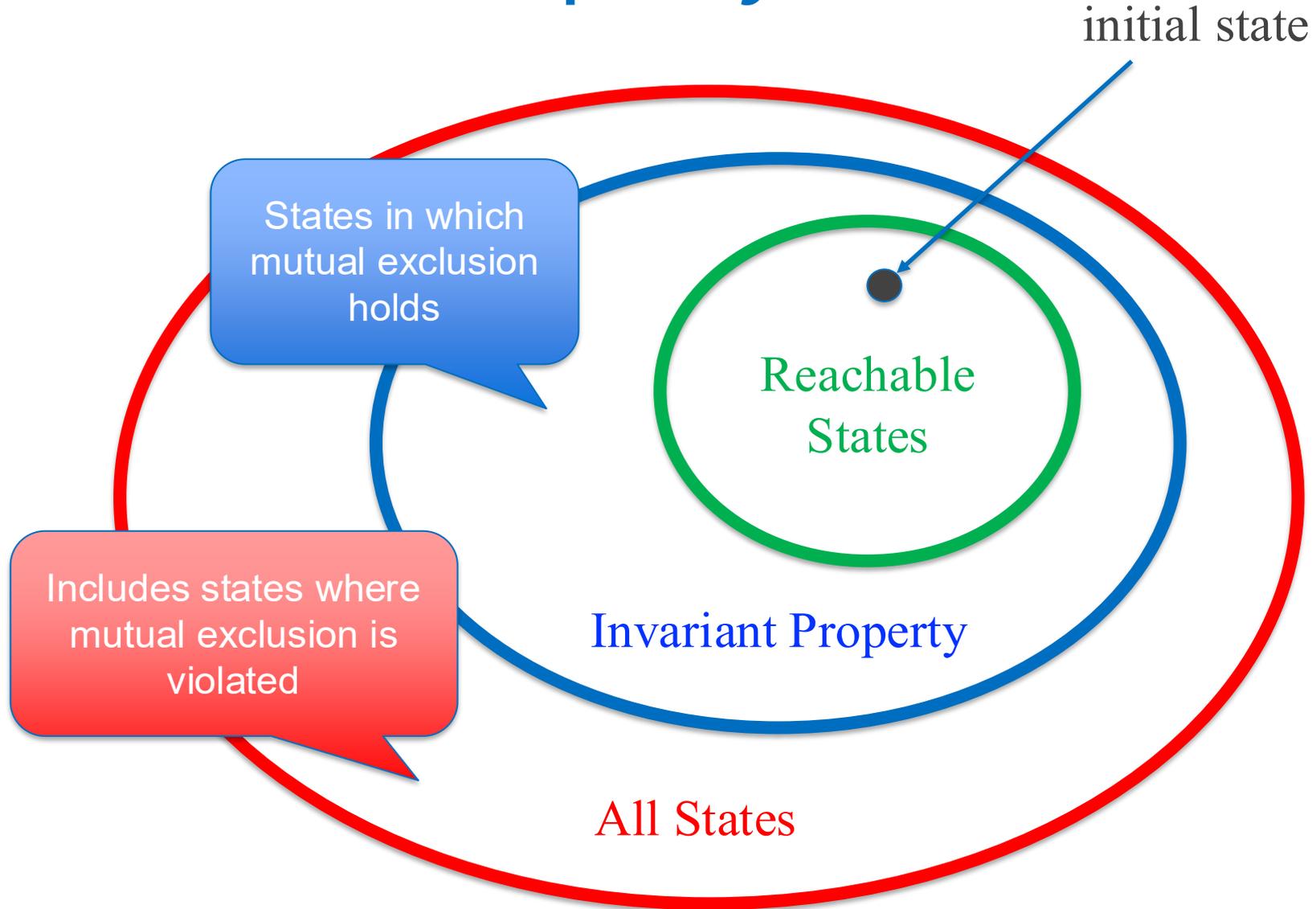
# Invariant Property



# Invariant Property



# Invariant Property



# How to prove an invariant?

- Need to show that, for any execution, all states reached satisfy the invariant
- Sounds similar to sorting:
  - Need to show that, for any list of numbers, the resulting list is ordered
- Let's try *proof by induction* on the length of an execution

# Proof by induction

You want to prove that some *Induction Hypothesis*  $IH(n)$  holds for any  $n$ :

- Base Case:

- show that  $IH(0)$  holds

- Induction Step:

- show that if  $IH(i)$  holds, then so does  $IH(i+1)$

# Proof by induction in our case

To show that some **IH** holds for an *execution* **E** of any number of *steps*:

- Base Case:

- show that **IH** holds in the initial state(s)

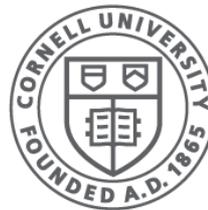
- Induction Step:

- show that if **IH** holds in a state produced by **E**, then for any possible next step **s**, **IH** also holds in the state produced by **E + [s]**

# Example

- Theorem: if **T** is in the critical section, then **flags[T] = True**
- **Base case**: true because initially T is not in the critical section and False implies anything
- **Induction**: easy to show (using Hoare logic) because flags[T] can only be changed by T itself

# Data Races



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Peterson's Reconsidered

- Assumes that LOAD and STORE instructions are *atomic*
- Not guaranteed on a real processor
- Also not guaranteed by C, Java, Python,

...

```
1 in_cs = 0
2 invariant in_cs in { 0, 1 }
3
4 sequential flags, turn ← Loads and Stores are atomic
5 flags = [ False, False ]
6 turn = choose({0, 1})
7
8 def thread(self):
9     while choose({ False, True }):
10        # Enter critical section
11        flags[self] = True
12        turn = 1 - self
13        await (not flags[1 - self]) or (turn == self)
14
15        atomically in_cs += 1
16        # Critical section
17        atomically in_cs -= 1
18
19        # Leave critical section
20        flags[self] = False
21
22 spawn thread(0)
23 spawn thread(1)
```

# For example

- CPU with 16-bit architecture
- 32-bit integer variable  $x$  stored in memory in two adjacent locations (aligned on word boundary)
- Initial value is 0
- Thread 1 writes FFFFFFFF to  $x$  (requires 2 STOREs)
- Thread 2 reads  $x$  (requires 2 LOADs)
- What are the possible values that thread 2 will read?

# For example

- CPU with 16-bit architecture
- 32-bit integer variable  $x$  stored in memory in two adjacent locations (aligned on word boundary)
- Initial value is 0
- Thread 1 writes FFFFFFFF to  $x$  (requires 2 STOREs)
- Thread 2 reads  $x$  (requires 2 LOADs)
- What are the possible values that thread 2 will read?
  - FFFFFFFF
  - 00000000
  - FFFF0000
  - 0000FFFF

# Other examples

- In Python, integers are arbitrary precision
  - that is, each integer variable is a complex data structure, and an operation may require multiple loads and stores
- Suppose your C compiler decides to pack multiple bits in a single word
  - E.g., `flags[0]`, `flags[1]`, and `turn`
  - Then setting a bit involves a load and a store

# Concurrent memory access

- Hardware may also cause problems in efforts to improve performance
  - buffering of writes
  - caching of reads
  - out-of-order execution
- Because of all these issues, programming languages will typically leave the outcome of concurrent operations to a variable *undefined*
  - if at least one of those operations is a store

# Data Race

- When two or more threads wish to access the same variable at the same time
- And at least one access is a STORE
- Then the semantics of the outcome is *undefined*
  - That is:
    - The value stored in the variable is undefined
    - The value loaded (if any) is undefined
    - Undefined means random (or worse, like a crash)

# Harmony “sequential” statement

- **sequential** *turn, flags* in Peterson’s
- ensures that loads/stores are atomic
- that is, concurrent operations appear to be executed sequentially
- This is called “*sequential consistency*”

## *For example*

- Shared variable  $x$  contains 3
- Thread A stores 4 into  $x$
- Thread B loads  $x$ 
  - With atomic load/store operations, B will read either 3 or 4
  - With normal operations, the value that B reads is undefined

# Sequential consistency

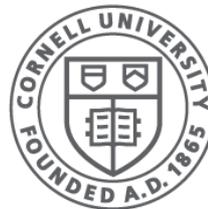
- Java has a similar notion:
  - **volatile int x;**
  - All accesses to volatile variables are sequentially consistent (but not whole program)
- **Not to be confused with the same keyword in C and C++ though...**
- Loading/storing volatile (sequentially consistent) variables is *more expensive* than loading/storing ordinary variables
  - because it restricts CPU and/or compiler optimizations
  - e.g., rules out caching

# Peterson's Reconsidered Again

- Peterson's algorithm is correct with atomic LOAD and STORE instructions
  - hardware supports such instructions, but they are very expensive
- Peterson's can be generalized to  $>2$  processes
  - even more STOREs and LOADs

*Too inefficient in practice*

# Specifying a lock



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Back to basics: specifying a lock

- What does a lock *do* exactly?
- What if we want more than one?

# Harmony interlude: *pointers*

- If  $x$  is a shared variable,  $?x$  is the address of  $x$
- If  $p$  is a variable and  $p$  contains  $?x$ , then we say that  $p$  is a *pointer* and it *points to*  $x$
- Finally,  $!p$  refers to the value of  $x$



Where?  
There!

# Specifying a lock

```
1  def Lock() returns result:  
2      result = False  
3  
4  def acquire(lk):  
5      atomically when not !lk:  
6          !lk = True  
7  
8  def release(lk):  
9      atomically:  
10         assert !lk  
11         !lk = False
```

# Specifying a lock

```
1 def Lock() returns result:  
2   result = False  
3  
4 def acquire(lk):  
5   atomically when not !lk:  
6     !lk = True  
7  
8 def release(lk):  
9   atomically:  
10    assert !lk  
11    !lk = False
```



returns initial value



acquires lock atomically once available



releases lock atomically

# Critical Section using a *lock*

```
1  from synch import Lock, acquire, release
2
3  shared = 0
4  thelock = Lock()
5
6  def f():
7      acquire(?thelock)
8      shared += 1
9      release(?thelock)
10
11  spawn f()
12  spawn f()
13
14  finally shared == 2
```

# Critical Section using a *lock*

```
1  from synch import Lock, acquire, release
2
3  shared = 0
4  thelock = Lock()
5
6  def f():
7      acquire(?thelock)
8      shared += 1
9      release(?thelock)
10
11  spawn f()
12  spawn f()
13
14  finally shared == 2
```



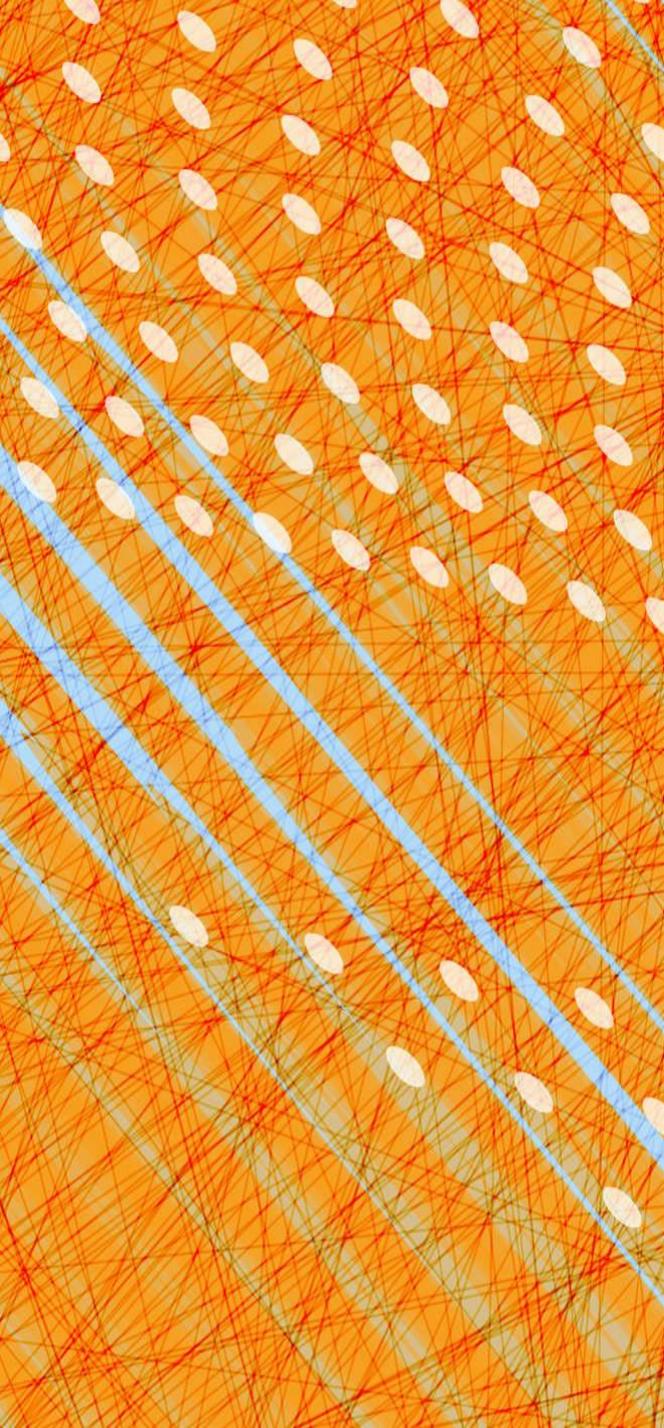
Contains Lock spec

# “Ghost” state

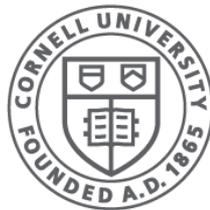
- We say that a lock is *held* or *owned* by a thread
  - implicit “ghost” state (not an actual variable)
  - nonetheless can be used for reasoning
- Two important invariants:
  1.  $T@CriticalSection \Rightarrow T$  holds the lock
  2. at most one thread can hold the lock

Together guarantee mutual exclusion

*Many systems do not keep track of who holds a particular lock, if anybody*



# Implementing a lock



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Implementing a lock

We saw that it is hard and inefficient to implement a lock with just LOAD and STORE instructions

# Enter *Interlock Instructions*

Machine instructions that do **multiple** shared memory accesses **atomically**

- e.g., **test\_and\_set s**
  - sets **s** to **True**
  - returns old value of **s**
- i.e., does the following:
  - LOAD r0, s      # load variable s into register r0
  - STORE s, 1      # store TRUE in variable s
- **Entire operation is *atomic***
  - other machine instructions cannot interleave

# Lock implementation (“spinlock”)

```
1 def test_and_set(s) returns result:  
2     atomically:  
3         result = !s  
4         !s = True  
5  
6 def atomic_store(p, v):  
7     atomically !p = v  
8  
9 def Lock() returns result:  
10     result = False  
11  
12 def acquire(lk):  
13     while test_and_set(lk):  
14         pass  
15  
16 def release(lk):  
17     atomic_store(lk, False)
```

*specification* of the CPU's  
**test\_and\_set** functionality

*specification* of the CPU's  
**atomic store** functionality

*lock implementation*

# Specification vs Implementation

```
1 def Lock() returns result:  
2     result = False  
3  
4 def acquire(lk):  
5     atomically when not !lk:  
6         !lk = True  
7  
8 def release(lk):  
9     atomically:  
10         assert !lk  
11         !lk = False
```

```
1 def test_and_set(s) returns result:  
2     atomically:  
3         result = !s  
4         !s = True  
5  
6 def atomic_store(p, v):  
7     atomically !p = v  
8  
9 def Lock() returns result:  
10     result = False  
11  
12 def acquire(lk):  
13     while test_and_set(lk):  
14         pass  
15  
16 def release(lk):  
17     atomic_store(lk, False)
```

**Specification:** describes *what an abstraction does*

**Implementation:** describes *how*



# Spinlocks and Time Sharing

- Spinlocks work well when threads on different cores need to synchronize
- But how about when it involves two threads time-shared on the same core:
  - when there is no pre-emption?
  - when there is pre-emption?

# Spinlocks and Time Sharing

- Spinlocks work well when threads on different cores need to synchronize
- But how about when it involves two threads time-shared on the same core:
  - when there is no pre-emption?
    - can cause all threads to get stuck while one is trying to obtain a spinlock
  - when there is pre-emption?

# Spinlocks and Time Sharing

- Spinlocks work well when threads on different cores need to synchronize
- But how about when it involves two threads time-shared on the same core:
  - when there is no pre-emption?
    - can cause all threads to get stuck while one is trying to obtain a spinlock
  - when there is pre-emption?
    - can cause delays and waste of CPU cycles while a thread is trying to obtain a spinlock

# Context switching in Harmony

- Harmony allows contexts to be saved and restored (i.e., **context switch**)

- ***r = stop p***

- stops the current thread and stores context in *!p*

- ***go (!p) r***

- adds a thread with the given context to the bag of threads. Thread resumes from **stop** expression, returning *r*

# Locks using **stop** and **go**

```
1 def Lock() returns result:
2     result = { .acquired: False, .suspended: [] }
3
4 def acquire(lk):
5     atomically:
6         if lk->acquired:
7             stop ?lk->suspended[ len lk->suspended ]
8             assert lk->acquired
9         else:
10            lk->acquired = True
11
12 def release(lk):
13     atomically:
14         assert lk->acquired
15         if lk->suspended == []:
16             lk->acquired = False
17         else:
18             go (lk->suspended[0]) ()
19             del lk->suspended[0]
```

.acquired: boolean  
.suspended: queue of contexts

# Locks using **stop** and **go**

```
1 def Lock() returns result:
2   result = { .acquired: False, .suspended: [] }
3
4 def acquire(lk):
5   atomically:
6     if lk->acquired:
7       stop ?lk->suspended[len lk->suspended]
8       assert lk->acquired
9     else:
10      lk->acquired = True
11
12 def release(lk):
13   atomically:
14     assert lk->acquired
15     if lk->suspended == []:
16       lk->acquired = False
17     else:
18       go (lk->suspended[0]) ()
19     del lk->suspended[0]
```

.acquired: boolean  
.suspended: queue of contexts

← put thread on wait queue

← resume first thread on wait queue

# Locks using **stop** and **go**

```
1 def Lock() returns result:  
2     result = { .acquired: False, .suspended: [] }  
3  
4 def acquire(lk):  
5     atomically:
```

Similar to a Linux “futex”: if there is no contention (hopefully the common case) acquire() and release() are cheap. If there is contention, they involve a context switch.

```
13         atomically:  
14             assert lk->acquired  
15             if lk->suspended == []:  
16                 lk->acquired = False  
17             else:  
18                 go (lk->suspended[0]) ()  
19                 del lk->suspended[0]
```

# Choosing modules in Harmony

- “synch” is the (default) module that has the specification of a lock
- “synchS” is the module that has the **stop/go** version of lock
- you can select which one you want:

`harmony -m synch=synchS x.hny`

- “synch” tends to be faster than “synchS”
  - smaller state graph

# Atomic Section $\neq$ Critical Section

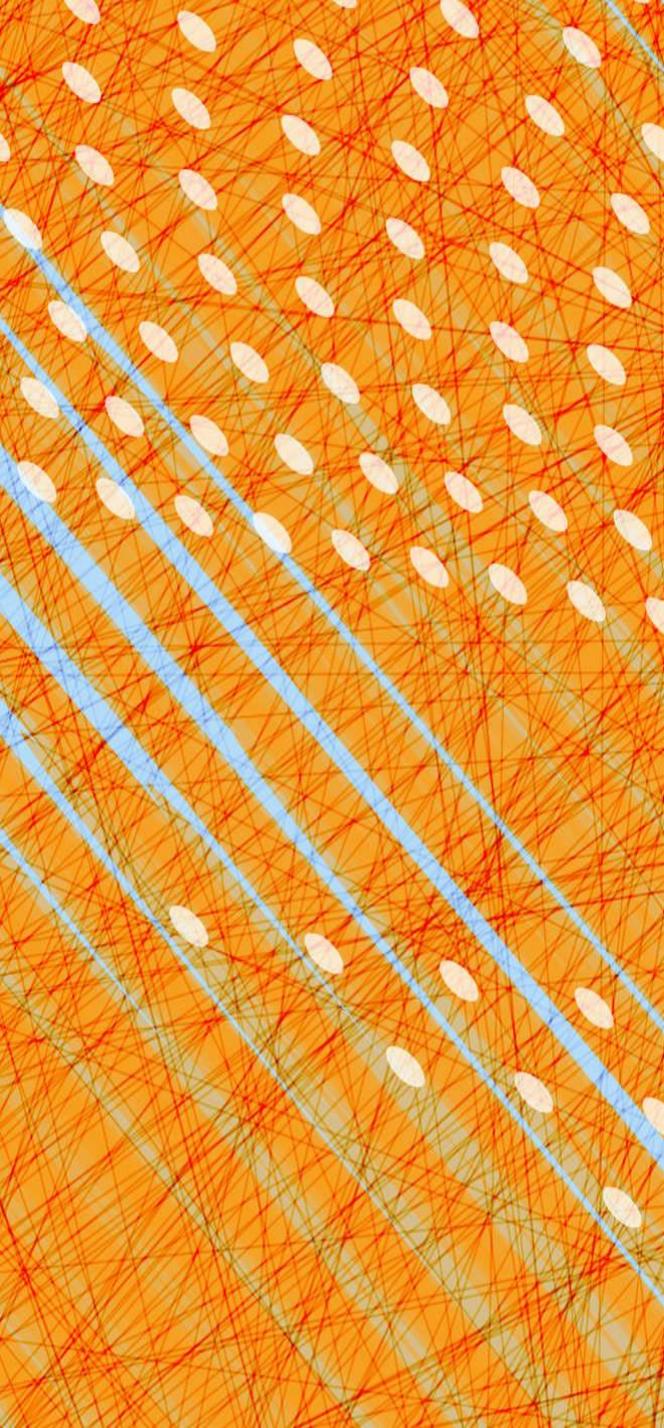
Atomic Section	Critical Section
only one thread can execute	multiple threads can execute concurrently, just not within a critical section
rare programming language paradigm	ubiquitous: <b>locks</b> available in many mainstream programming languages
good for <b>specifying interlock instructions</b>	good for <b>implementing concurrent data structures</b>

# Data Race $\neq$ Race Condition

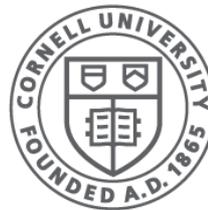
- A *Data Race* occurs when two threads try to access the same variable and at least one access is non-atomic and at least one access is an update.
  - The outcome of the operations is **undefined**
- A *Race Condition* occurs when the correctness of the program depends on ordering of variable access
  - Race Condition can happen without a Data Race

# Data Race $\neq$ Race Condition

- **Data Race:** Harmony can automatically detect these because Harmony enumerates all behaviors and fails if there is undefined behavior
- **Race Condition:** Harmony can only detect these if you tell Harmony what it is that you want using **assert**, **invariant**, or **finally**
  - or by explicitly enumerating the correct behaviors, as we'll see later



# Demo Time



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Harmony demo

Demo 1:  
data race

```
x = 0

def f():
    x = x + 1

def g():
    x = x + 1

spawn f()
spawn g()
```

Demo 2: no data race

```
x = 0

def atomic_load(p) returns v:
    atomically v = !p

def atomic_store(p, v):
    atomically !p = v

def f():
    atomic_store(?x, atomic_load(?x) + 1)

def g():
    atomic_store(?x, atomic_load(?x) + 1)

spawn f()
spawn g()
```

Demo 3: same  
semantics as  
Demo 2:

```
sequential x

x = 0

def f():
    x = x + 1

def g():
    x = x + 1

spawn f()
spawn g()
```

# Harmony demo

Demo 4: still a data race

```
x = 0

def atomic_load(p) returns v:
    atomically v = !p

def atomic_store(p, v):
    atomically !p = v

def f():
    atomic_store(?x, x + 1)

def g():
    atomic_store(?x, atomic_load(?x) + 1)

spawn f()
spawn g()
```

Demo 5: data race  
freedom does not imply  
no race conditions

```
sequential x
finally x == 2

x = 0

def f():
    x += 1

def g():
    x += 1

spawn f()
spawn g()
```

# Harmony demo

Demo 6: spec of  
what we want

```
finally x == 2

x = 0

def f():
    atomically x += 1

def g():
    atomically x += 1

spawn f()
spawn g()
```

Demo 7: implementation  
using critical section

```
from synch import Lock, acquire, release

finally x == 2

x = 0
thelock = Lock()

def f():
    acquire(?thelock)
    x += 1
    release(?thelock)

def g():
    acquire(?thelock)
    x += 1
    release(?thelock)

spawn f()
spawn g()
```

# Harmony demo

## Demo 8: broken implementation using two critical sections

```
from synch import Lock, acquire, release

finally x == 2

x = 0
thelock1 = Lock()
thelock2 = Lock()

def f():
    acquire(?thelock1)
    x += 1
    release(?thelock1)

def g():
    acquire(?thelock2)
    x += 1
    release(?thelock2)

spawn f()
spawn g()
```

Which of the following is a false statement?

 0

$3 < 4$  is an invariant

if  $3 > 4$ , then the moon is made out of green cheese

if the moon is made out of green cheese, then  $3 < 4$

if the moon is not made out of green cheese, then  $3 < 4$

None of the above

## Which of the following is a false statement?



$3 < 4$  is an invariant

0%

if  $3 > 4$ , then the moon is made out of green cheese

0%

if the moon is made out of green cheese, then  $3 < 4$

0%

if the moon is not made out of green cheese, then  $3 < 4$

0%

None of the above

0%

## Which of the following is a false statement?



$3 < 4$  is an invariant

0%

if  $3 > 4$ , then the moon is made out of green cheese

0%

if the moon is made out of green cheese, then  $3 < 4$

0%

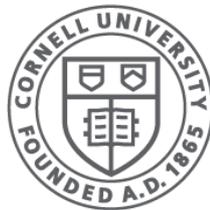
if the moon is not made out of green cheese, then  $3 < 4$

0%

None of the above

0%

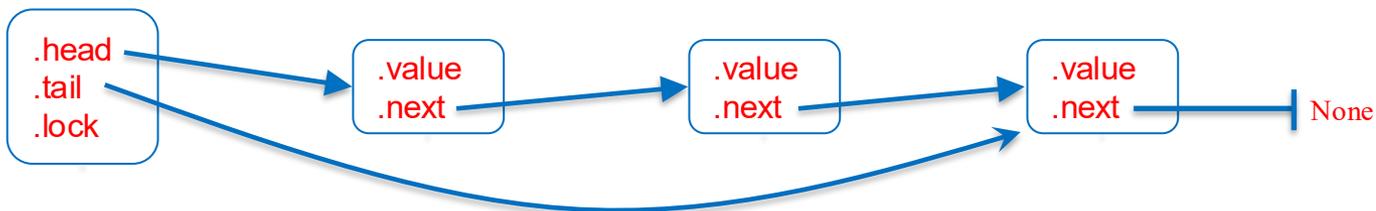
# Concurrent Data Structure Consistency



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Data Structure *consistency*

- Each data structure maintains some *consistency property*
  - e.g., in a linked list, there is a head, a tail, a list of nodes such that head points to first node, tail points to the last node, and each node points to the next one except the last, which points to **None**. However, if the list is empty, head and tail are both **None**.



# Consistency using locks

- Each data structure maintains some *consistency property*
  - e.g., in a linked list, there is a head, a tail, a list of nodes such that head points to first node, tail points to the last node, and each node points to the next one except the last, which points to **None**. However, if the list is empty, head and tail are both **None**.
- *You can assume the property holds right after obtaining the lock*
- *You must make sure the property holds again right before releasing the lock*

# Consistency using locks

- Each data structure maintains some *consistency property*
- *Invariant:*
  - lock not held  $\Rightarrow$  data structure consistent
- *Or equivalently:*
  - data structure inconsistent  $\Rightarrow$  lock held

# Building a concurrent queue

- `q = queue.Queue()`: initialize a new queue
- `queue.put(q, v)`: add `v` to the tail of queue `q`
- `v = queue.get(q)`: returns `None` if `q` is empty or `v` if `v` was at the head of the queue

# Specifying a ~~concurrent~~ queue

```
1  def Queue() returns empty:  
2      empty = []  
3  
4  def put(q, v):  
5      !q += [v,]  
6  
7  def get(q) returns next:  
8      if !q == []:  
9          next = None  
10     else:  
11         next = (!q)[0]  
12         del (!q)[0]
```

# Specifying a concurrent queue

```
1 def Queue() returns empty:
2     empty = []
3
4 def put(q, v):
5     !q += [v,]
6
7 def get(q) returns next:
8     if !q == []:
9         next = None
10    else:
11        next = (!q)[0]
12        del (!q)[0]
```

Sequential

```
1 def Queue() returns empty:
2     empty = []
3
4 def put(q, v):
5     atomically !q += [v,]
6
7 def get(q) returns next:
8     atomically:
9         if !q == []:
10            next = None
11        else:
12            next = (!q)[0]
13            del (!q)[0]
```

Concurrent

# Example of using a queue

```
1 import queue
2
3 def sender(q, v):
4     queue.put(q, v)
5
6 def receiver(q):
7     let v = queue.get(q):
8         assert v in { None, 1, 2 }
9
10 demoq = queue.Queue()
11 spawn sender(?demoq, 1)
12 spawn sender(?demoq, 2)
13 spawn receiver(?demoq)
14 spawn receiver(?demoq)
```

enqueue  $v$  onto  $!q$

dequeue and check

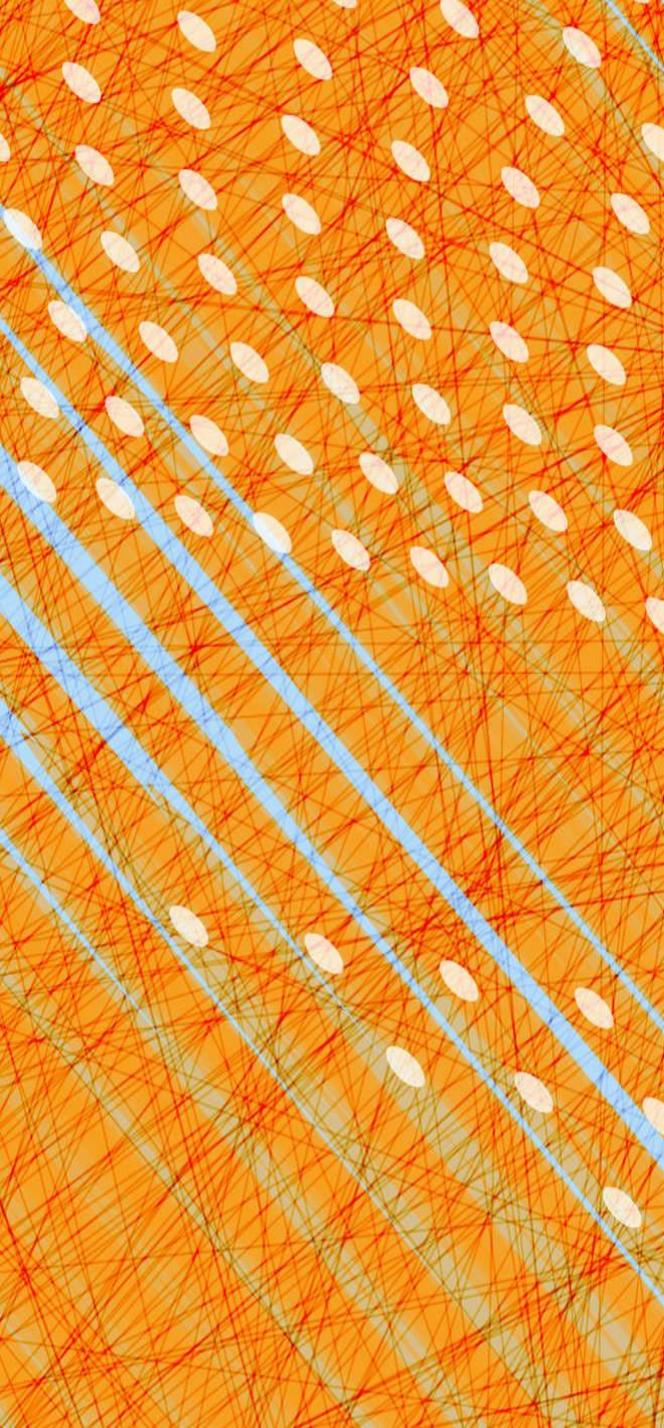
create queue

# Specifying a concurrent queue

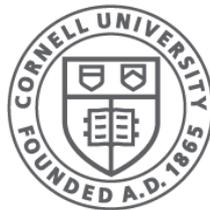
```
1 def Queue() returns empty:  
2     empty = []  
3  
4 def put(q, v):  
5     atomically !q += [v,]  
6  
7 def get(q) returns next:  
8     atomically:  
9         if !q == []:  
10            next = None  
11        else:  
12            next = (!q)[0]  
13            del (!q)[0]
```

Not a good implementation because

- operations are  $O(n)$
- code uses **atomically**  
compiler cannot generate code

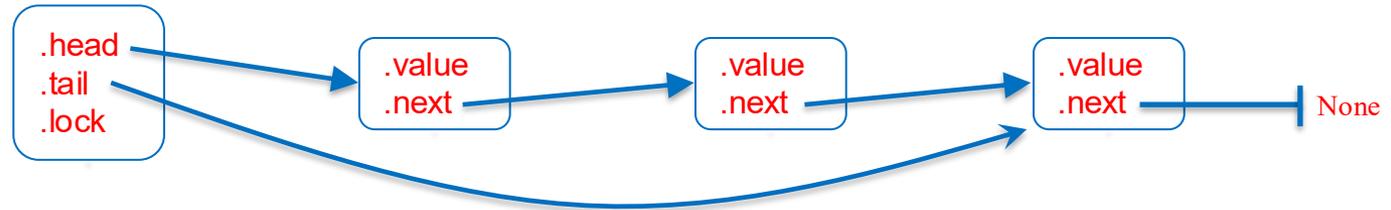


# Implementing a concurrent queue



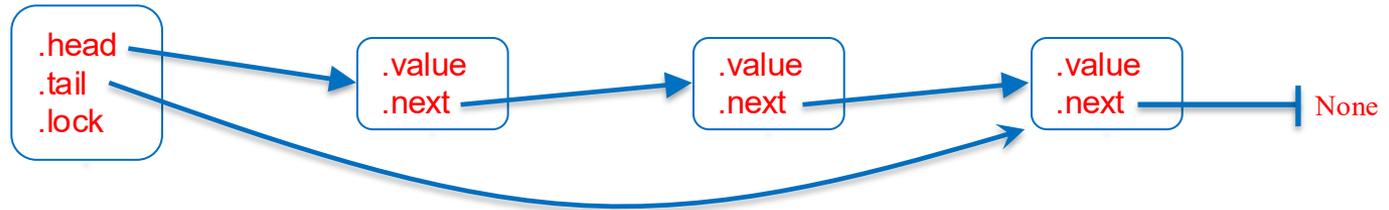
**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Queue implementation, v1



```
1  from synch import Lock, acquire, release
2  from alloc import malloc, free
3
4  def Queue() returns empty:
5      empty = { .head: None, .tail: None, .lock: Lock() }
6
7  def put(q, v):
8      let node = malloc({ .value: v, .next: None }):
9          acquire(?q->lock)
10         if q->tail == None:
11             q->tail = q->head = node
12         else:
13             q->tail->next = node
14             q->tail = node
15         release(?q->lock)
```

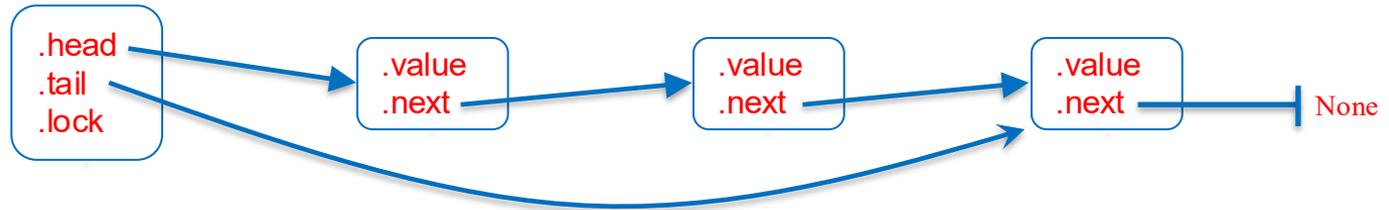
# Queue implementation, v1



```
1 from synch import Lock, acquire, release
2 from alloc import malloc, free
3
4 def Queue() returns empty:
5     empty = { .head: None, .tail: None, .lock: Lock() }
6
7 def put(q, v):
8     let node = malloc({ .value: v, .next: None }):
9         acquire(?q->lock)
10        if q->tail == None:
11            q->tail = q->head = node
12        else:
13            q->tail->next = node
14            q->tail = node
15        release(?q->lock)
```

*dynamic memory allocation*

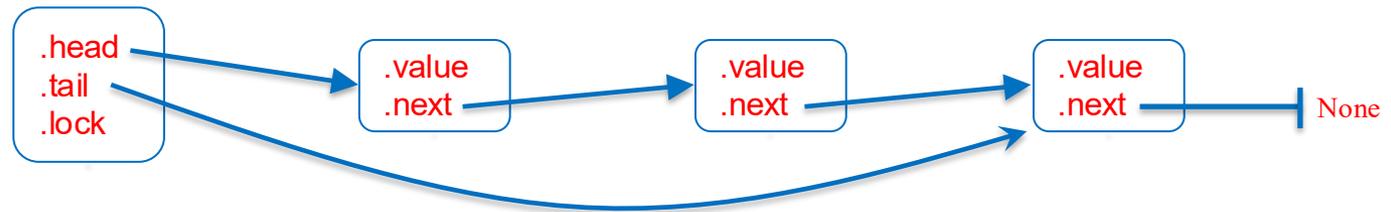
# Queue implementation, v1



```
1 from synch import Lock, acquire, release
2 from alloc import malloc, free
3
4 def Queue() returns empty:
5     empty = { .head: None, .tail: None, .lock: Lock() }
6
7 def put(q, v):
8     let node = malloc({ .value: v, .next: None }):
9         acquire(?q->lock)
10        if q->tail == None:
11            q->tail = q->head = node
12        else:
13            q->tail->next = node
14            q->tail = node
15        release(?q->lock)
```

empty queue

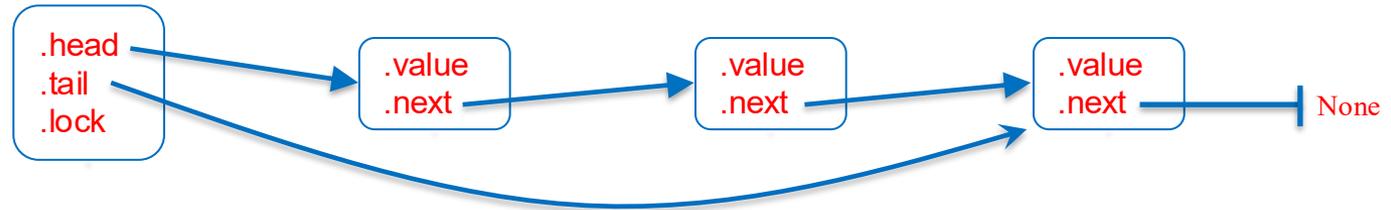
# Queue implementation, v1



```
1  from synch import Lock, acquire, release
2  from alloc import malloc, free
3
4  def Queue() returns empty:
5      empty = { .head: None, .tail: None, .lock: Lock() }
6
7  def put(q, v):
8      let node = malloc({ .value: v, .next: None }):
9          acquire(?q->lock)
10         if q->tail == None:
11             q->tail = q->head = node
12         else:
13             q->tail->next = node
14             q->tail = node
15         release(?q->lock)
```

*allocate node*

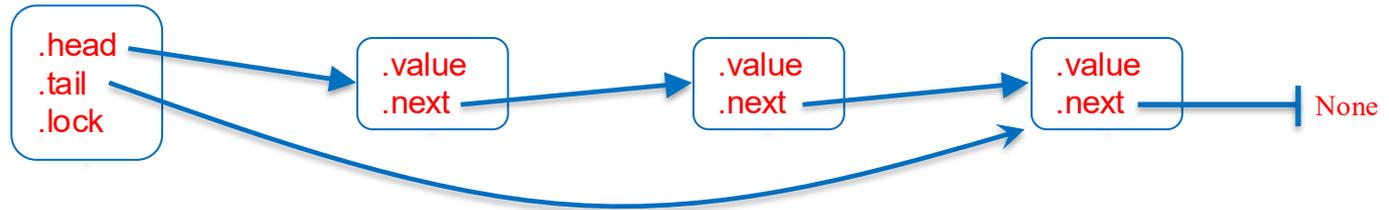
# Queue implementation, v1



```
1  from synch import Lock, acquire, release
2  from alloc import malloc, free
3
4  def Queue() returns empty:
5      empty = { .head: None, .tail: None, .lock: Lock() }
6
7  def put(q, v):
8      let node = malloc({ .value: v, .next: None }):
9          acquire(?q->lock)
10         if q->tail == None:
11             q->tail = q->head = node
12         else:
13             q->tail->next = node
14             q->tail = node
15         release(?q->lock)
```

grab lock

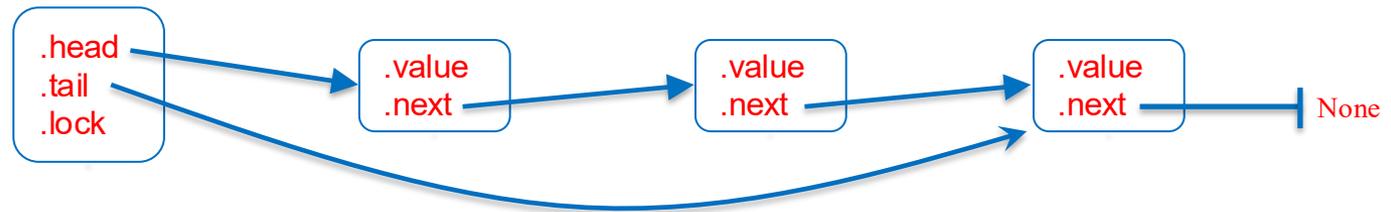
# Queue implementation, v1



```
1  from synch import Lock, acquire, release
2  from alloc import malloc, free
3
4  def Queue() returns empty:
5      empty = { .head: None, .tail: None, .lock: Lock() }
6
7  def put(q, v):
8      let node = malloc({ .value: v, .next: None }):
9          acquire(?q->lock)
10         if q->tail == None:
11             q->tail = q->head = node
12         else:
13             q->tail->next = node
14             q->tail = node
15         release(?q->lock)
```

*the hard stuff*

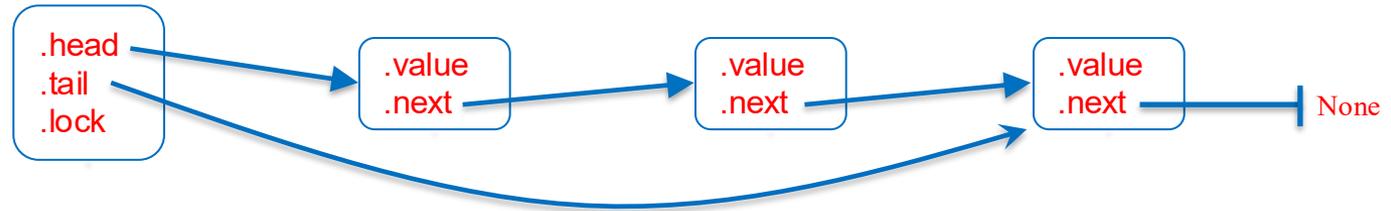
# Queue implementation, v1



```
1  from synch import Lock, acquire, release
2  from alloc import malloc, free
3
4  def Queue() returns empty:
5      empty = { .head: None, .tail: None, .lock: Lock() }
6
7  def put(q, v):
8      let node = malloc({ .value: v, .next: None }):
9          acquire(?q->lock)
10         if q->tail == None:
11             q->tail = q->head = node
12         else:
13             q->tail->next = node
14             q->tail = node
15         release(?q->lock)
```

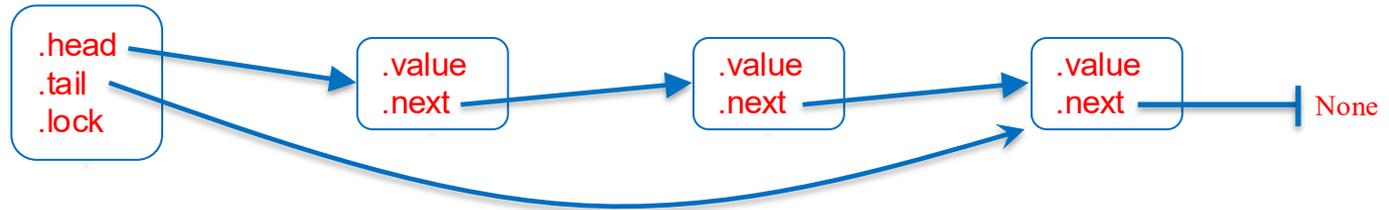
*release lock*

# Queue implementation, v1



```
17 def get(q) returns next:
18   acquire(?q->lock)
19   let node = q->head:
20     if node == None:
21       next = None
22     else:
23       next = node->value
24       q->head = node->next
25       if q->head == None:
26         q->tail = None
27       free(node)
28   release(?q->lock)
```

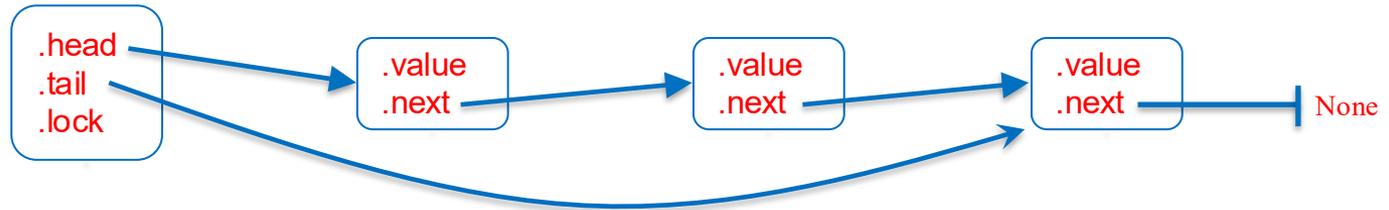
# Queue implementation, v1



```
17 def get(q) returns next:
18   acquire(?q->lock)
19   let node = q->head:
20     if node == None:
21       next = None
22     else:
23       next = node->value
24       q->head = node->next
25       if q->head == None:
26         q->tail = None
27       free(node)
28   release(?q->lock)
```

*the hard stuff*

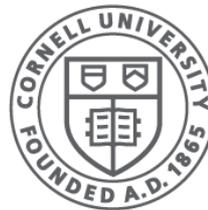
# Queue implementation, v1



```
17 def get(q) returns next:
18     acquire(?q->lock)
19     let node = q->head:
20         if node == None:
21             next = None
22         else:
23             next = node->value
24             q->head = node->next
25             if q->head == None:
26                 q->tail = None
27             free(node)
28     release(?q->lock)
```

*malloc'd memory must be explicitly released (cf. C)*

# Systematic Testing



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Systematic Testing

- Sequential case
  - try all “sequences” of 1 operation
    - put or get (in case of queue)
  - try all sequences of 2 operations
    - put+put, put+get, get+put, get+get, ...
  - try all sequences of 3 operations
  - ...
- How do you know if a sequence is correct?
  - compare “behaviors” of running test against implementation with running test against the sequential specification

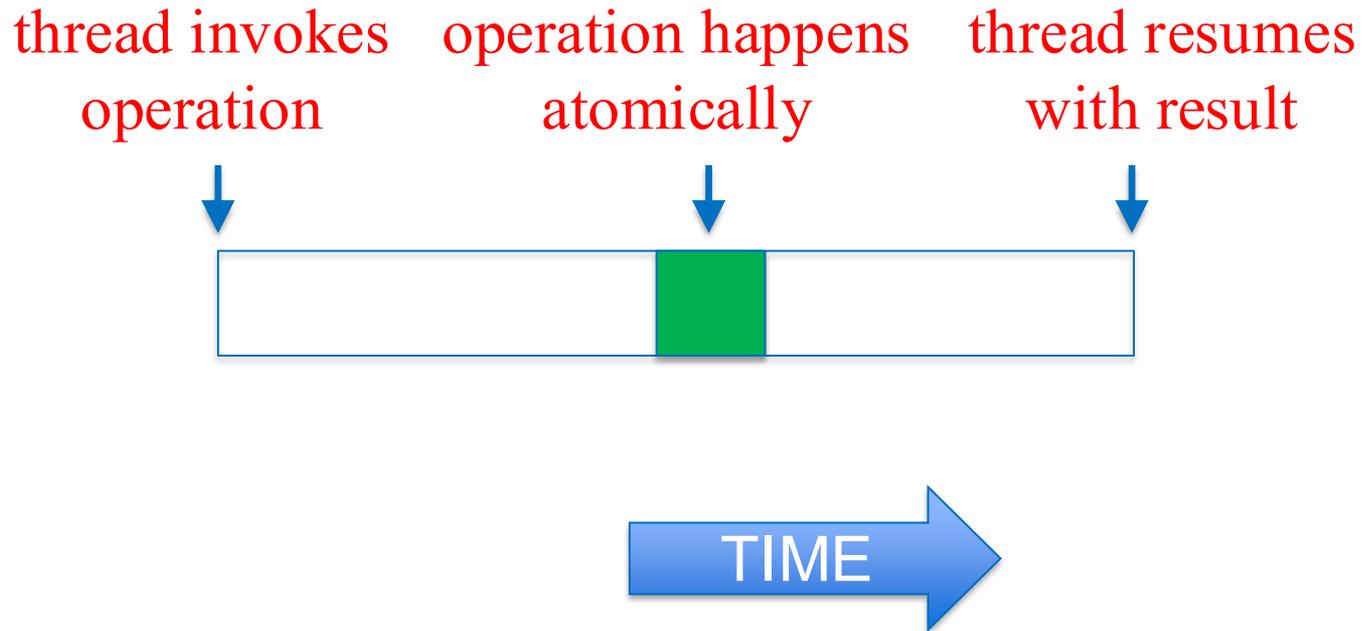
# Systematic Testing

- Concurrent case
  - try all “interleavings” of 1 operation
  - try all interleavings of 2 operations
  - try all interleavings of 3 operations
  - ...
- How do you know if an interleaving is correct?
  - compare “behaviors” of running test against concurrent implementation with running test against the concurrent specification

# How do we capture behaviors?

- And what is a behavior?

# Life of an atomic operation



# Concurrency and Overlap

Is the following a possible scenario?

1. customer X orders a burger
2. customer Y orders a burger (after X)
3. customer Y is served a burger
4. customer X is served a burger (after Y)

# Concurrency and Overlap

Is the following a possible scenario?

1. customer X orders a burger
2. customer Y orders a burger (after X)
3. customer Y is served a burger
4. customer X is served a burger (after Y)

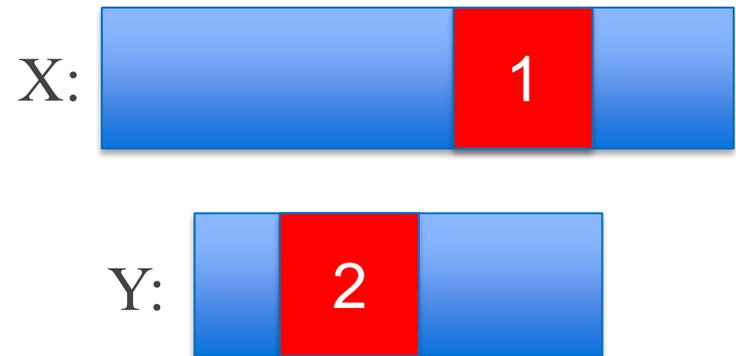
We've all seen this happen. **It's a matter of how things get scheduled!**

# Specification

- One operation: order a burger
  - result: a burger (at some later time)
- Semantics: the burger manifests itself atomically *sometime during the operation*
  - *Atomically: no two manifestations overlap*
- It's easier to specify something when you don't have to worry about overlap
  - i.e., you can simply give a sequential specification

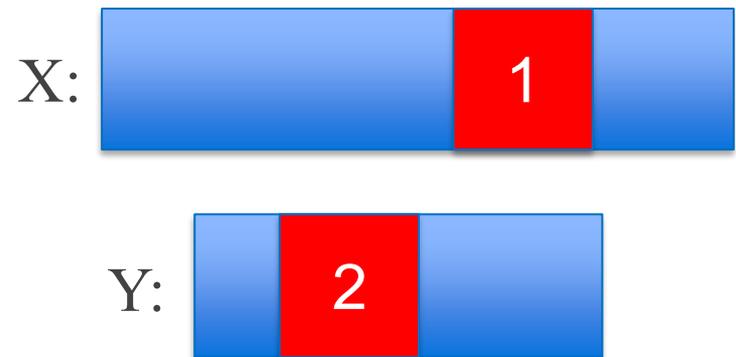
# Implementation?

- Suppose the diner has one small hot plate and two cooks
- Cooks use a lock for access to the hot plate
- Possible scenario:
  1. customer X orders burger, order ends up with cook 1
  2. customer Y orders burger, order ends up with cook 2
  3. cook 1 was busy with something else, so cook 2 grabs the lock first
  4. cook 2 cooks burger for Y
  5. cook 2 releases lock
  6. cook 1 grabs lock
  7. cook 1 cooks burger for X
  8. cook 1 releases lock
  9. customer Y receives burger
  10. customer X receives burger



# Implementation?

- Suppose the diner has one small hot plate and two cooks
- Cooks use a lock for access to the hot plate
- Possible scenario:
  1. customer X orders burger, order ends up with cook 1
  2. customer Y orders burger, order ends up with cook 2
  3. cook 1 was busy with something else, so cook 2 grabs the lock first
  4. cook 2 cooks burger for Y
  5. cook 2 releases lock
  6. cook 1 grabs lock
  7. cook 1 cooks burger for X
  8. cook 1 releases lock
  9. customer Y receives burger
  10. customer X receives burger



- *can't happen if Y orders burger after X receives burger*
- *but if operations overlap, any ordering can happen...*

# Correct Behaviors

queue initially empty

(1)

put(1)

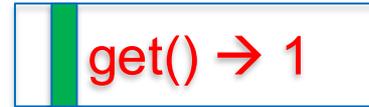
get() → ?



# Correct Behaviors

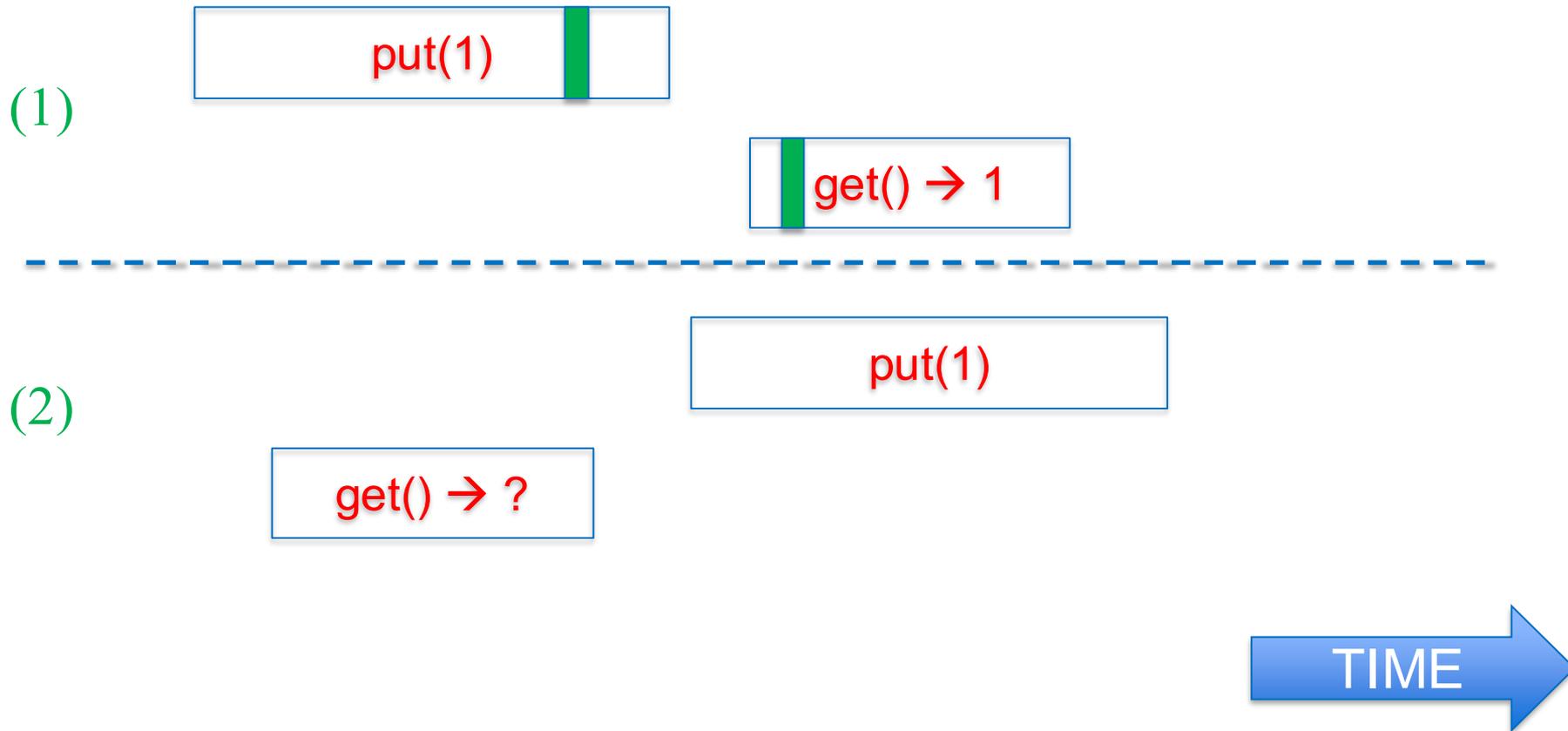
queue initially empty

(1)



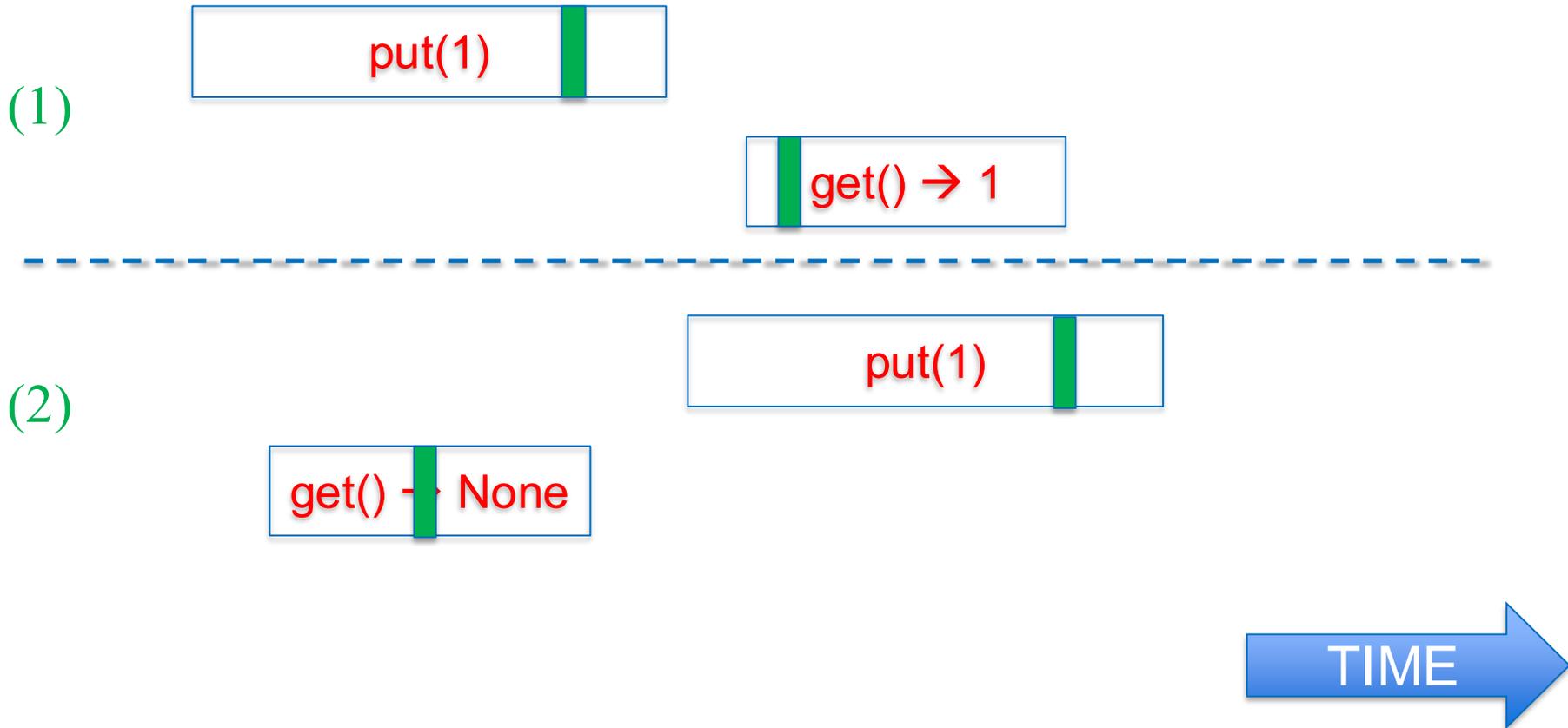
# Correct Behaviors

queue initially empty



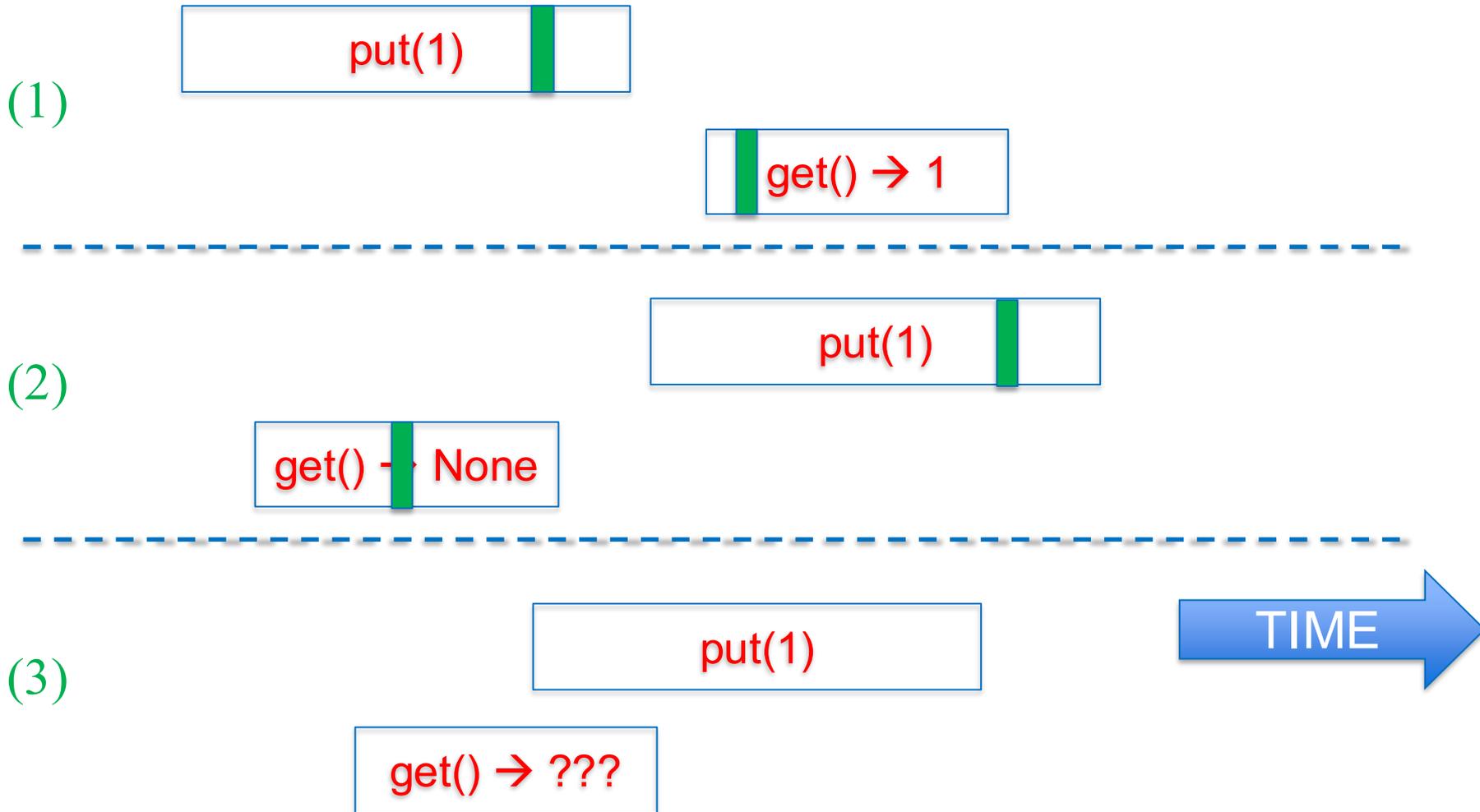
# Correct Behaviors

queue initially empty



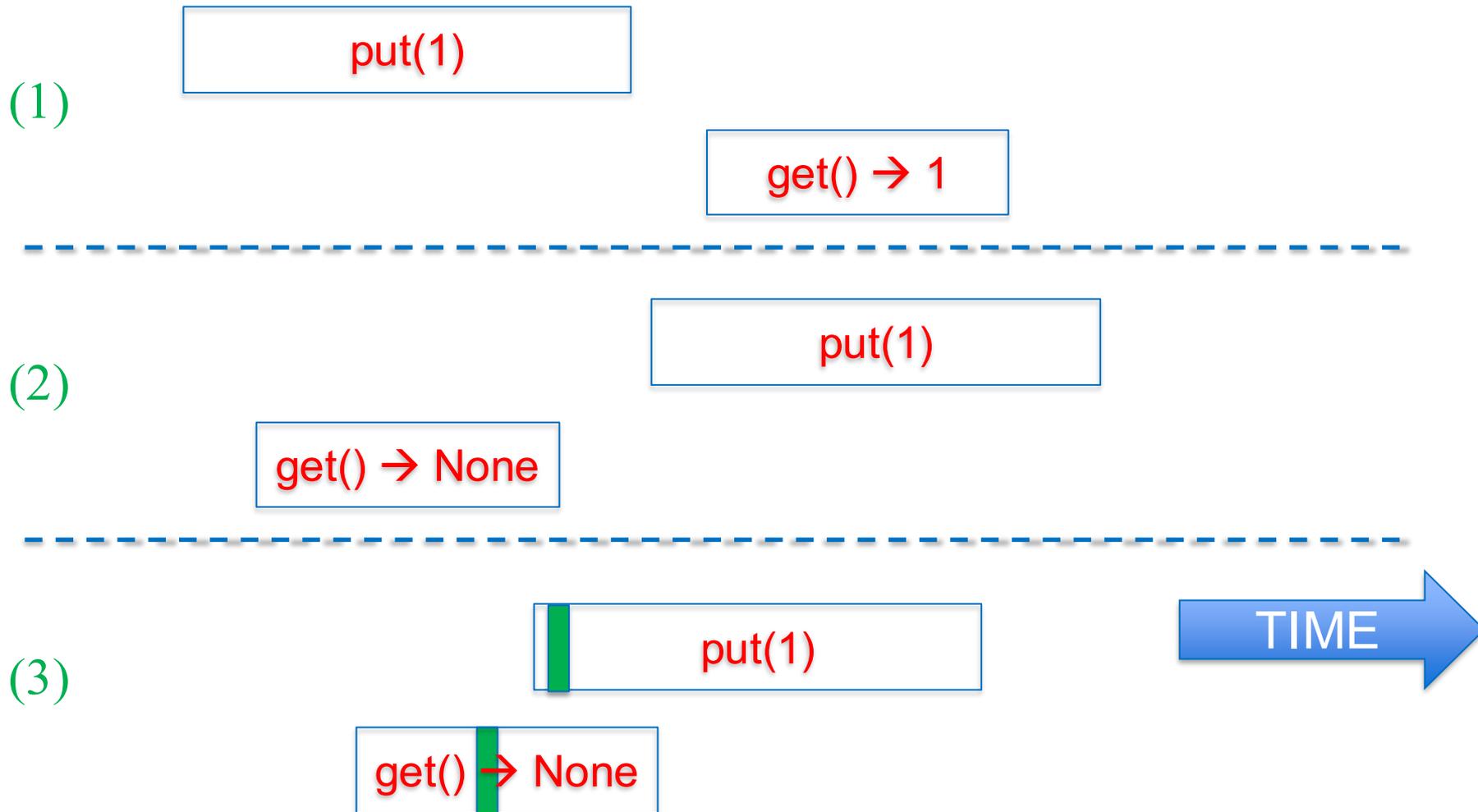
# Correct Behaviors

queue initially empty



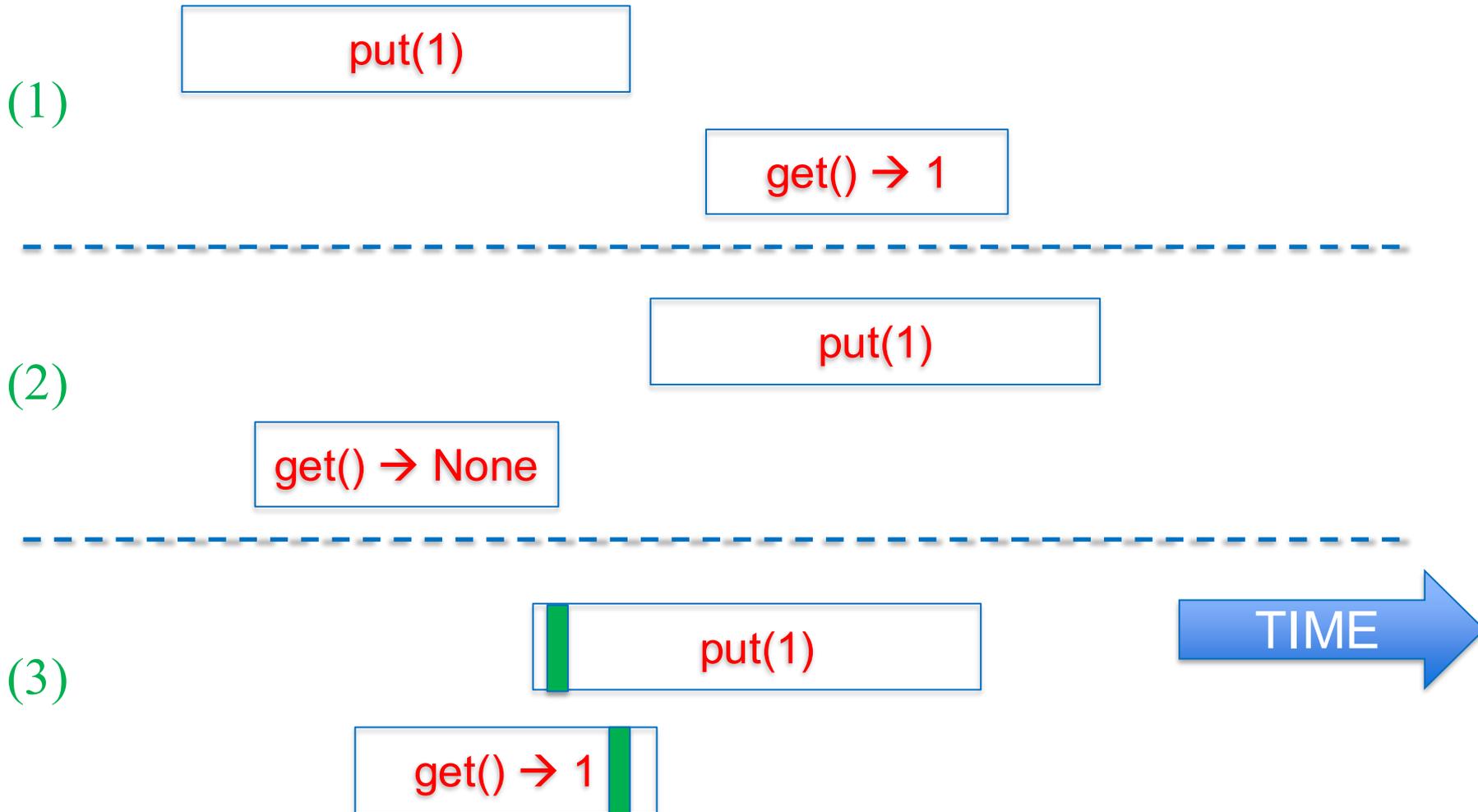
# Correct Behaviors

queue initially empty



# Correct Behaviors

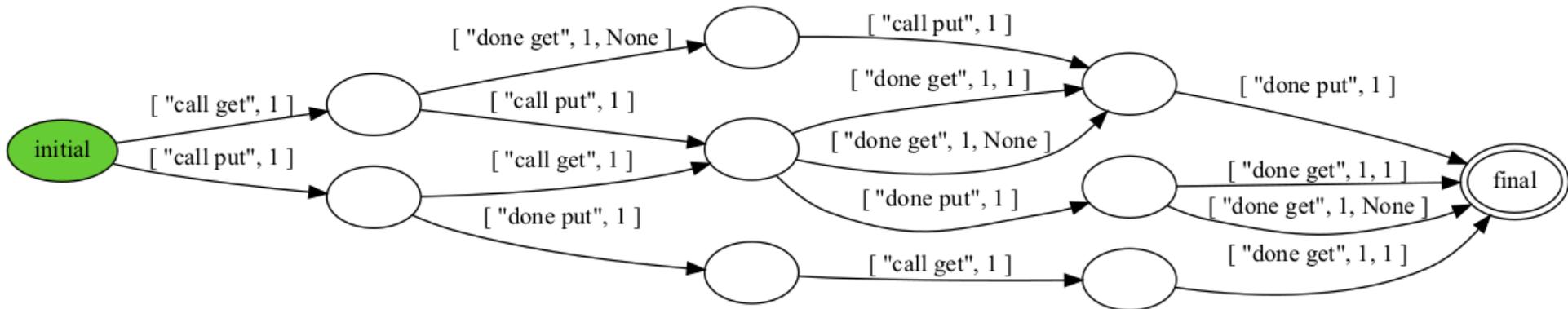
queue initially empty



# Concurrent queue test program

```
1 import queue
2
3 const N_PUT = 2
4 const N_GET = 2
5 q = queue.Queue()
6
7 def put_test(self):
8     print("call put", self)
9     queue.put(q, self)
10    print("done put", self)
11
12 def get_test(self):
13    print("call get", self)
14    let v = queue.get(q):
15        print("done get", self, v)
16
17 for i in {1..N_PUT}:
18    spawn put_test(i)
19 for i in {1..N_GET}:
20    spawn get_test(i)
```

# Correct behaviors (1 get, 1 put)



```
$ harmony -c N_GET=1 -c N_PUT=1 code/queue_btest2.hny
```

# Testing: comparing behaviors

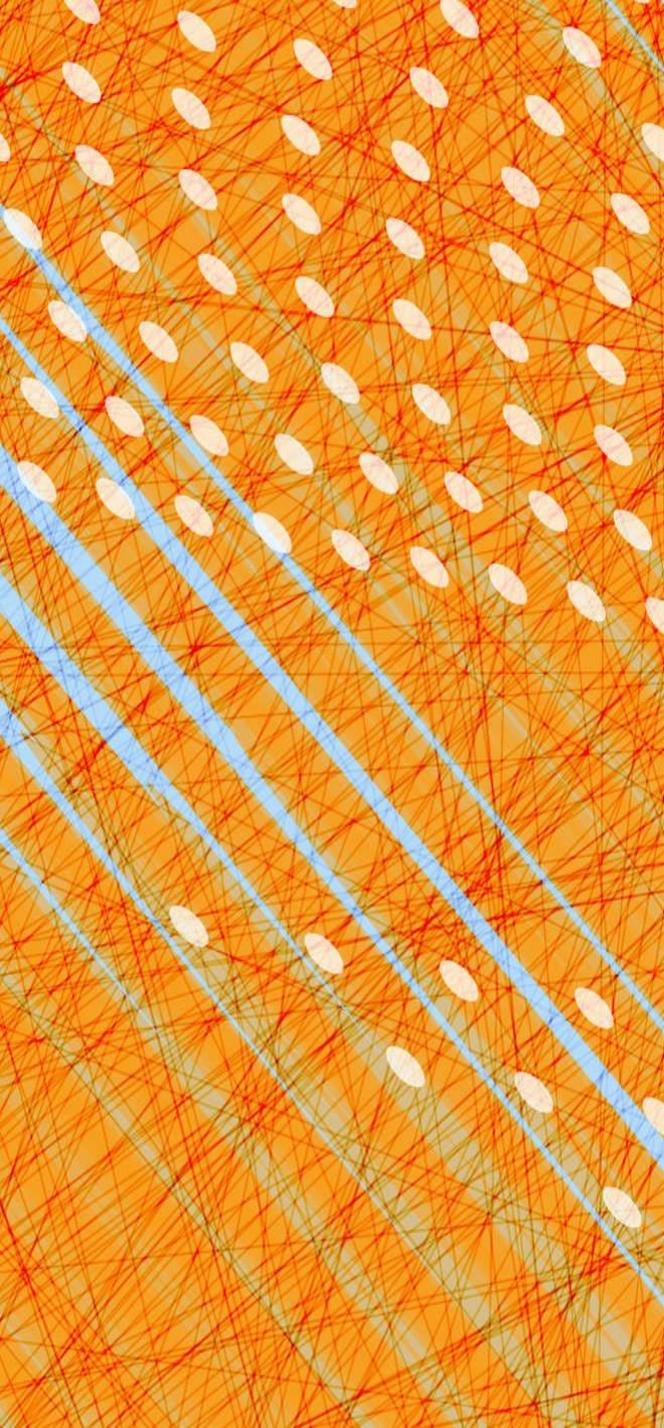
```
$ harmony -o queue.hfa code/queue_btest2.hny
```

```
$ harmony -B queue.hfa -m queue=queue_lock code/queue_btest2.hny
```

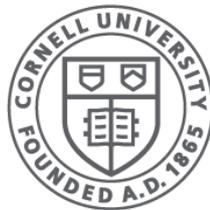
- The first command outputs the behavior of running the test program against the specification in file queue.hfa
- The second command runs the test program against the implementation and checks if its behavior matches that stored in queue.hfa

# Black Box Testing

- Not allowed to look under the covers
  - can't use *q->head*, etc.
- Only allowed to invoke the interface methods and observe behaviors
- Your job: try to find bad behaviors
  - compare against a *specification*
  - how would you test a clock? An ATM machine?
    - without looking inside
- In general testing cannot ensure correctness
  - only a correctness proof can
  - testing may or may not expose a bug
  - model checking helps expose bugs



# Fine-Grained Locking



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# How important are concurrent queues?

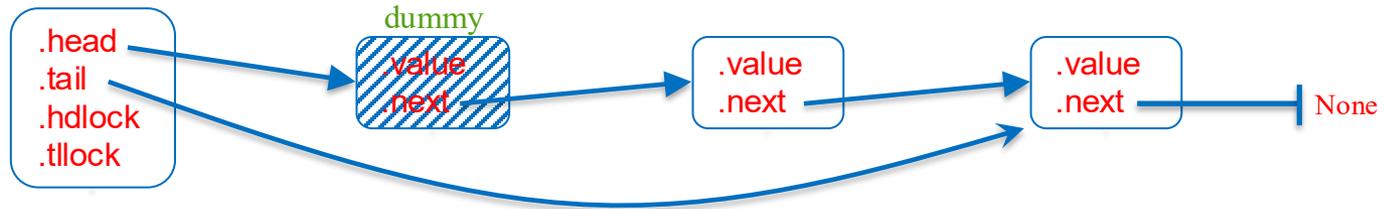
- Answer: **all important**
  - any resource that needs scheduling
    - CPU run queue
    - disk, network, printer waiting queue
    - lock waiting queue
  - inter-process communication
    - Posix pipes:
      - `cat file | tr a-z A-Z | grep RVR`
  - actor-based concurrency
  - ...

# How important are concurrent queues?

- Answer: **all important**
  - any resource that needs scheduling
    - CPU run queue
    - disk, network, printer waiting queue
    - lock waiting queue
  - inter-process communication
    - Posix pipes:
      - **cat file | tr a-z A-Z | grep RVR**
  - actor-based concurrency
  - ...

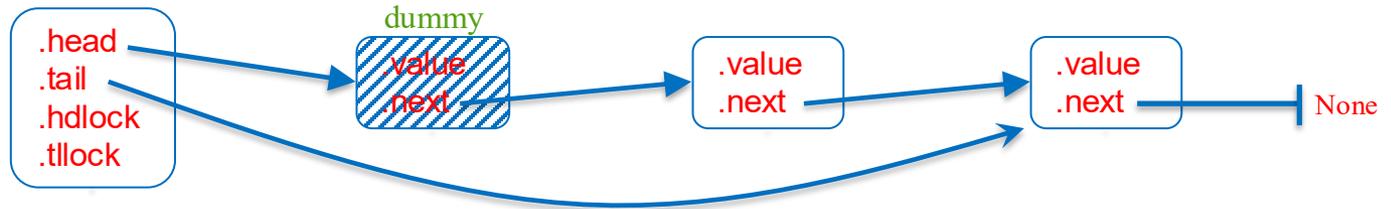
*Good performance is critical!*

# Concurrent queue v2: 2 locks



```
1 from synch import Lock, acquire, release, atomic_load, atomic_store
2 from alloc import malloc, free
3
4 def Queue() returns empty:
5     let dummy = malloc({ .value: (), .next: None }):
6         empty = { .head: dummy, .tail: dummy,
7                 .hdlock: Lock(), .tllock: Lock() }
8
9 def put(q, v):
10    let node = malloc({ .value: v, .next: None }):
11        acquire(?q->tllock)
12        atomic_store(?q->tail->next, node)
13        q->tail = node
14        release(?q->tllock)
```

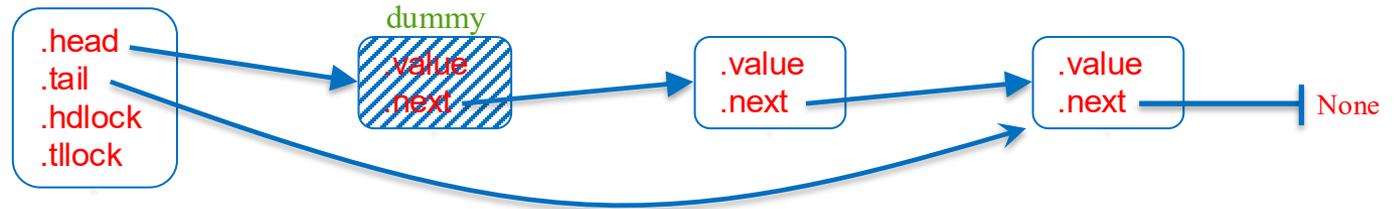
# Concurrent queue v2: 2 locks



```
1 from synch import Lock, acquire, release, atomic_load, atomic_store
2 from alloc import malloc, free
3
4 def Queue() returns empty:
5     let dummy = malloc({ .value: (), .next: None }):
6         empty = { .head: dummy, .tail: dummy,
7                 .hdlock: Lock(), .tllock: Lock() }
8
9 def put(q, v):
10    let node = malloc({ .value: v, .next: None }):
11        acquire(?q->tllock)
12        atomic_store(?q->tail->next, node)
13        q->tail = node
14        release(?q->tllock)
```

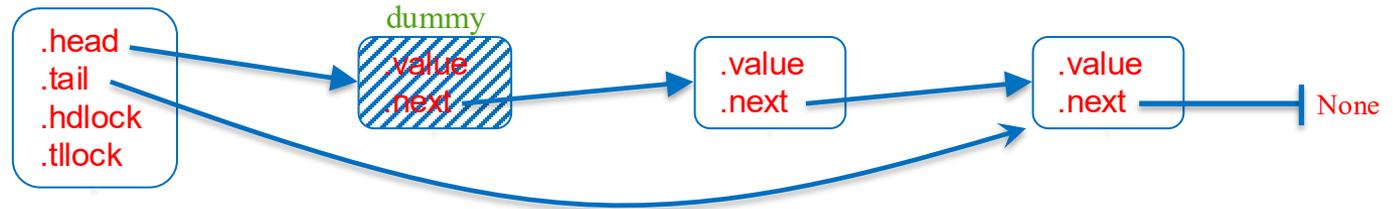
atomically `q->tail->next = node`

# Concurrent queue v2: 2 locks



```
16 def get(q) returns next:
17   acquire(?q->hdlock)
18   let dummy = q->head
19   let node = atomic_load(?dummy->next):
20     if node == None:
21       next = None
22       release(?q->hdlock)
23   else:
24     next = node->value
25     q->head = node
26     release(?q->hdlock)
27     free(dummy)
```

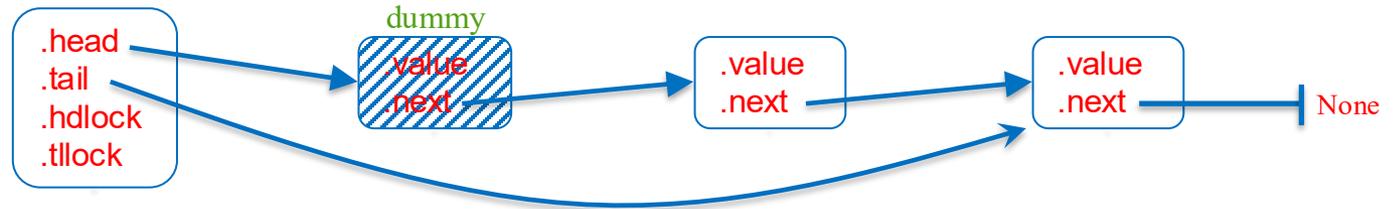
# Concurrent queue v2: 2 locks



```
16 def get(q) returns next:  
17   acquire(?q->hdlock)  
18   let dummy = q->head  
19   let node = atomic_load(?dummy->next):  
20     if node == None:  
21       next = None  
22       release(?q->hdlock)  
23     else:  
24       next = node->value  
25       q->head = node  
26       release(?q->hdlock)  
27       free(dummy)
```

No contention for concurrent  
enqueue and dequeue operations!  
→ more concurrency → faster

# Concurrent queue v2: 2 locks



```
16 def get(q) returns next:
17   acquire(?q->hdlock)
18   let dummy = q->head
19   let node = atomic_load(?dummy->next):
20     if node == None:
21       next = None
22       release(?q->hdlock)
23   else:
24     next = node->value
25     q->head = node
26     release(?q->hdlock)
27     free(dummy)
```

No contention for concurrent  
enqueue and dequeue operations!  
→ more concurrency → faster

Needs to avoid data race on  
*dummy* → *next* when queue is empty

# Global vs. Local Locks

- The two-lock queue is an example of a data structure with *finer-grained locking*
- A global lock is easy, but limits concurrency
- Fine-grained or local locking can improve concurrency, but tends to be trickier to get right

# Sorted Linked List with Lock per Node

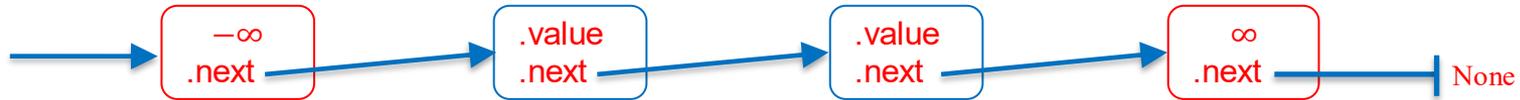


```
1 from synch import Lock, acquire, release
2 from alloc import malloc, free
3
4 def _node(v, n) returns node: # allocate and initialize
5     node = malloc({ .lock: Lock(), .value: v, .next: n })
6
7 def _find(lst, v) returns pair:
8     var before = lst
9     acquire(?before->lock)
10    var after = before->next
11    acquire(?after->lock)
12    while after->value < (0, v):
13        release(?before->lock)
14        before = after
15        after = before->next
16        acquire(?after->lock)
17    pair = (before, after)
18
19 def SetObject() returns object:
20    object = _node((-1, None), _node((1, None), None))
```

- $-\infty$  represented by  $(-1, \text{None})$
  - $v$  represented by  $(0, v)$
  - $\infty$  represented by  $(1, \text{None})$
- Note that  $\forall v: (-1, \text{None}) < (0, v) < (1, \text{None})$   
(lexicographical ordering)



# Sorted Linked List with Lock per Node



```
1 from synch import Lock, acquire, release
2 from alloc import malloc, free
3
4 def _node(v, n) returns node: # allocate and initialize
5     node = malloc({ .lock: Lock(), .value: v, .next: n })
6
7 def _find(lst, v) returns pair:
8     var before = lst
9     acquire(?before->lock)
10    var after = before->next
11    acquire(?after->lock)
12    while after->value < (0, v):
13        release(?before->lock)
14        before = after
15        after = before->next
16        acquire(?after->lock)
17    pair = (before, after)
18
19 def SetObject() returns object:
20    object = _node((-1, None), _node((1, None), None))
```

Helper routine to find and lock two consecutive nodes *before* and *after* such that *before* → value < v ≤ *after* → value

# Sorted Linked List with Lock per Node



```
1 from synch import Lock, acquire, release
2 from alloc import malloc, free
3
4 def _node(v, n) returns node: # allocate and initialize
5     node = malloc({ .lock: Lock(), .value: v, .next: n })
6
7 def _find(lst, v) returns pair:
8     var before = lst
9     acquire(?before->lock)
10    var after = before->next
11    acquire(?after->lock)
12    while after->value < (0, v):
13        release(?before->lock)
14        before = after
15        after = before->next
16        acquire(?after->lock)
17    pair = (before, after)
18
19 def SetObject() returns object:
20    object = _node((-1, None), _node((1, None), None))
```

Helper routine to find and lock two consecutive nodes *before* and *after* such that  $before \rightarrow value < v \leq after \rightarrow value$

*Hand-over hand locking*  
(good for data structures without cycles)

# Sorted Linked List with Lock per Node

```
def insert(lst, v):
  let before, after = _find(lst, v):
    if after->value != (0, v):
      before->next = _node((0, v), after)
    release(?after->lock)
    release(?before->lock)

def remove(lst, v):
  let before, after = _find(lst, v):
    if after->value == (0, v):
      before->next = after->next
      free(after)
    else:
      release(?after->lock)
    release(?before->lock)

def contains(lst, v) returns present:
  let before, after = _find(lst, v):
    present = after->value == (0, v)
  release(?after->lock)
  release(?before->lock)
```

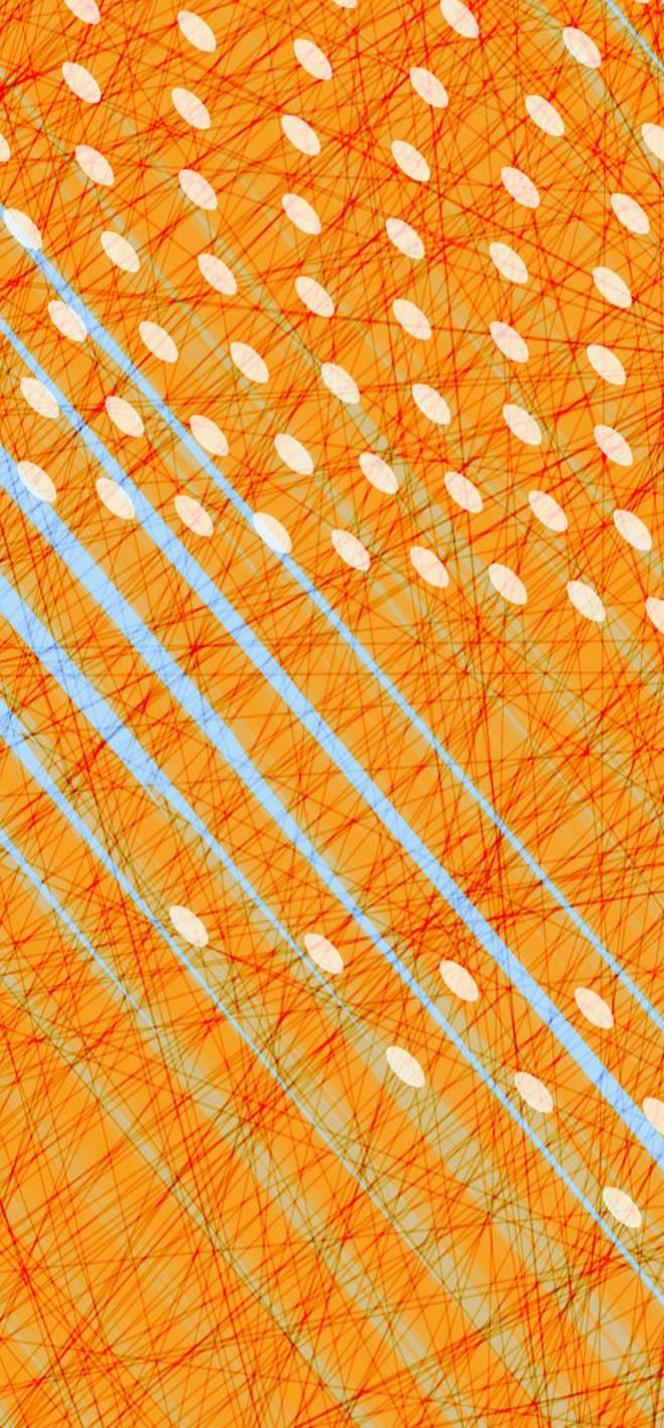
# Sorted Linked List with Lock per Node

```
def insert(lst, v):
  let before, after = _find(lst, v):
    if after->value != (0, v):
      before->next = _node((0, v), after)
    release(?after->lock)
    release(?before->lock)

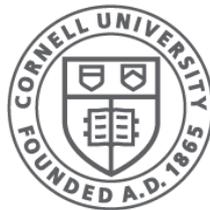
def remove(lst, v):
  let before, after = _find(lst, v):
    if after->value == (0, v):
      before->next = after->next
      free(after)
    else:
      release(?after->lock)
    release(?before->lock)

def contains(lst, v) returns present:
  let before, after = _find(lst, v):
    present = after->value == (0, v)
  release(?after->lock)
  release(?before->lock)
```

Multiple threads can access the list simultaneously, but they can't *overtake* one another



# Conditional Waiting



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Conditional Waiting

- Thus far we've shown how threads can wait for one another to avoid multiple threads in the critical section
- Sometimes there are other reasons:
  - Wait until queue is non-empty
  - Wait until there are no readers (or writer) in a reader/writer section
  - ...

# Reader/writer lock

**Idea:** allow multiple read-only operations to execute concurrently

- Still no data races
- In many cases, reads are much more frequent than writes

→ **Either:**

- multiple readers, or
- a single writer

*thus not:*

- *a reader and a writer, nor*
- *multiple writers*

# Reader/Writer Lock Specification

```
1 def RWlock() returns lock:
2   lock = { .nreaders: 0, .nwriters: 0 }
3
4 def read_acquire(rw):
5   atomically when rw->nwriters == 0:
6     rw->nreaders += 1
7
8 def read_release(rw):
9   atomically rw->nreaders -= 1
10
11 def write_acquire(rw):
12   atomically when (rw->nreaders == 0) and (rw->nwriters == 0):
13     rw->nwriters = 1
14
15 def write_release(rw):
16   atomically rw->nwriters = 0
```

# Reader/Writer Lock Specification

```
1 def RWlock() returns lock:
2   lock = { .nreaders: 0, .nwriters: 0 }
3
4 def read_acquire(rw):
5   atomically when rw->nwriters == 0:
6     rw->nreaders += 1
7
8 def read_release(rw):
9   atomically rw->nreaders -= 1
10
11 def write_acquire(rw):
12   atomically when (rw->nreaders == 0) and (rw->nwriters == 0):
13     rw->nwriters = 1
14
15 def write_release(rw):
16   atomically rw->nwriters = 0
```

## Invariants:

- if  $n$  readers in the R/W critical section, then  $nreaders \geq n$
- if  $n$  writers in the R/W critical section, then  $nwriters \geq n$
- $(nreaders \geq 0 \wedge nwriters = 0) \vee (nreaders = 0 \wedge 0 \leq nwriters \leq 1)$

# R/W Locks: test for mutual exclusion

```
1 import rwlock
2
3 nreaders = nwriters = 0
4 invariant ((nreaders >= 0) and (nwriters == 0)) or
5           ((nreaders == 0) and (nwriters == 1))
6
7 const NOPS = 4
8
9 rw = rwlock.RWlock()
10
11 def thread():
12     while choose({ False, True }):
13         if choose({ "read", "write" }) == "read":
14             rwlock.read_acquire(?rw)
15             atomically nreaders += 1
16             atomically nreaders -= 1
17             rwlock.read_release(?rw)
18         else: # write
19             rwlock.write_acquire(?rw)
20             atomically nwriters += 1
21             atomically nwriters -= 1
22             rwlock.write_release(?rw)
23
24 for i in {1..NOPS}:
25     spawn thread()
```



*no writer, one or more readers*



*one writer, no readers*

# Cheating R/W lock implementation

```
1  import synch
2
3  def RWlock() returns lock:
4      lock = synch.Lock()
5
6  def read_acquire(rw):
7      synch.acquire(rw)
8
9  def read_release(rw):
10     synch.release(rw)
11
12 def write_acquire(rw):
13     synch.acquire(rw)
14
15 def write_release(rw):
16     synch.release(rw)
```

The *lock* protects the application's critical section

# Cheating R/W lock implementation

```
1  import synch
2
3  def RWLock() returns lock:
4      lock = synch.Lock()
5
6  def read_acquire(rw):
7      synch.acquire(rw)
8
9  def read_release(rw):
10     synch.release(rw)
11
12 def write_acquire(rw):
13     synch.acquire(rw)
14
15 def write_release(rw):
16     synch.release(rw)
```

The *lock* protects the application's critical section

Allows only one reader to get the lock at a time

Does *not* have the same behavior as the specification

- it is missing behaviors
- no bad behaviors though

# Busy Waiting Implementation

```
1 from synch import Lock, acquire, release
2
3 def RWlock() returns lock:
4     lock = { .lock: Lock(), .nreaders: 0, .nwriters: 0 }
5
6 def read_acquire(rw):
7     acquire(?rw->lock)
8     while rw->nwriters > 0:
9         release(?rw->lock)
10        acquire(?rw->lock)
11        rw->nreaders += 1
12        release(?rw->lock)
13
14 def read_release(rw):
15     acquire(?rw->lock)
16     rw->nreaders -= 1
17     release(?rw->lock)
18
19 def write_acquire(rw):
20     acquire(?rw->lock)
21     while rw->nreaders > 0 or rw->nwriters > 0:
22         release(?rw->lock)
23         acquire(?rw->lock)
24     rw->nwriters = 1
25     release(?rw->lock)
26
27 def write_release(rw):
28     acquire(?rw->lock)
29     rw->nwriters = 0
30     release(?rw->lock)
```

# Busy Waiting Implementation

```
1 from synch import Lock, acquire, release
2
3 def RWlock() returns lock:
4     lock = { .lock: Lock(), .nreaders: 0, .nwriters: 0 }
5
6 def read_acquire(rw):
7     acquire(?rw->lock)
8     while rw->nwriters > 0:
9         release(?rw->lock)
10        acquire(?rw->lock)
11        rw->nreaders += 1
12        release(?rw->lock)
13
14 def read_release(rw):
15     acquire(?rw->lock)
16     rw->nreaders -= 1
17     release(?rw->lock)
18
19 def write_acquire(rw):
20     acquire(?rw->lock)
21     while rw->nreaders > 0 or rw->nwriters > 0:
22         release(?rw->lock)
23         acquire(?rw->lock)
24     rw->nwriters = 1
25     release(?rw->lock)
26
27 def write_release(rw):
28     acquire(?rw->lock)
29     rw->nwriters = 0
30     release(?rw->lock)
```

The *lock* protects *nreaders* and *nwriters*, not the critical section of the application

# Busy Waiting Implementation

```
1 from synch import Lock, acquire, release
2
3 def RWlock() returns lock:
4     lock = { .lock: Lock(), .nreaders: 0, .nwriters: 0 }
5
6 def read_acquire(rw):
7     acquire(?rw->lock)
8     while rw->nwriters > 0:
9         release(?rw->lock)
10        acquire(?rw->lock)
11        rw->nreaders += 1
12        release(?rw->lock)
13
14 def read_release(rw):
15     acquire(?rw->lock)
16     rw->nreaders -= 1
17     release(?rw->lock)
18
19 def write_acquire(rw):
20     acquire(?rw->lock)
21     while rw->nreaders > 0 or rw->nwriters > 0:
22         release(?rw->lock)
23         acquire(?rw->lock)
24     rw->nwriters = 1
25     release(?rw->lock)
26
27 def write_release(rw):
28     acquire(?rw->lock)
29     rw->nwriters = 0
30     release(?rw->lock)
```

*waiting conditions*

# Busy Waiting Implementation

```
1 from synch import Lock, acquire, release
2
3 def RWlock() returns lock:
4     lock = { .lock: Lock(), .nreaders: 0, .nwriters: 0 }
5
6 def read_acquire(rw):
7     acquire(?rw->lock)
8     while rw->nwriters > 0:
9         release(?rw->lock)
10        acquire(?rw->lock)
11        rw->nreaders += 1
12        release(?rw->lock)
13
14 def read_release(rw):
15     acquire(?rw->lock)
16     rw->nreaders -= 1
17     release(?rw->lock)
18
19 def write_acquire(rw):
20     acquire(?rw->lock)
21     while rw->nreaders > 0 or rw->nwriters > 0:
22         release(?rw->lock)
23         acquire(?rw->lock)
24     rw->nwriters = 1
25     release(?rw->lock)
26
27 def write_release(rw):
28     acquire(?rw->lock)
29     rw->nwriters = 0
30     release(?rw->lock)
```

Good: has the same behaviors as the implementation

Bad: process is continuously scheduled to try to get the lock even if it's not available

*(Harmony complains about this as well)*

# Mesa Condition Variables

- A lock can have one or more *condition variables*
- A thread that holds the lock but wants to wait for some condition to hold can *temporarily* release the lock by *waiting* on some condition variable
- Associate a condition variable with each “waiting condition”
  - reader: no writer in the critical section
  - writer: no readers nor writers in the c.s.

# Mesa Condition Variables, cont'd

- When a thread that holds the lock notices that some waiting condition is satisfied it should *notify* the corresponding condition variable

# R/W lock with *Mesa* condition variables

```
1  from synch import *
2
3  def RWlock() returns lock:
4      lock = {
5          .nreaders: 0, .nwriters: 0, .mutex: Lock(),
6          .r_cond: Condition(), .w_cond: Condition()
7      }
```



*r\_cond*: used by readers to wait on  $nwriters == 0$

*w\_cond*: used by writers to wait on  $nreaders == 0 == nwriters$

# R/W Lock, reader part

```
9  def read_acquire(rw):
10     acquire(?rw->mutex)
11     while rw->nwriters > 0:
12         wait(?rw->r_cond, ?rw->mutex)
13     rw->nreaders += 1
14     release(?rw->mutex)
15
16  def read_release(rw):
17     acquire(?rw->mutex)
18     rw->nreaders -= 1
19     if rw->nreaders == 0:
20         notify(?rw->w_cond)
21     release(?rw->mutex)
```

# R/W Lock, reader part

```
9 def read_acquire(rw):
10     acquire(?rw->mutex)
11     while rw->nwriters > 0:
12         wait(?rw->r_cond, ?rw->mutex)
13         rw->nreaders += 1
14         release(?rw->mutex)
15
16 def read_release(rw):
17     acquire(?rw->mutex)
18     rw->nreaders -= 1
19     if rw->nreaders == 0:
20         notify(?rw->w_cond)
21     release(?rw->mutex)
```

} similar to  
busy waiting

# R/W Lock, reader part

```
9 def read_acquire(rw):
10     acquire(?rw->mutex)
11     while rw->nwriters > 0:
12         wait(?rw->r_cond, ?rw->mutex)
13         rw->nreaders += 1
14         release(?rw->mutex)
15
16 def read_release(rw):
17     acquire(?rw->mutex)
18     rw->nreaders -= 1
19     if rw->nreaders == 0:
20         notify(?rw->w_cond)
21     release(?rw->mutex)
```

} similar to  
busy waiting

} but need this

# R/W Lock, reader part

```
9 def read_acquire(rw):
10     acquire(?rw->mutex)
11     while rw->nwriters > 0:
12         wait(?rw->r_cond, ?rw->mutex)
13         rw->nreaders += 1
14     release(?rw->mutex)
15
16 def read_release(rw):
17     acquire(?rw->mutex)
18     rw->nreaders -= 1
19     if rw->nreaders == 0:
20         notify(?rw->w_cond)
21     release(?rw->mutex)
```

} similar to  
busy waiting

- Always use **while**
- Never just **if** (or nothing)
- **wait** without **while** is called a “naked wait”

} but need this

# R/W Lock, reader part

compare with busy waiting

```
def read_acquire(rw):
    acquire(?rw->lock)
    while rw->nwriters > 0:
        release(?rw->lock)
        acquire(?rw->lock)
    rw->nreaders += 1
    release(?rw->lock)

def read_release(rw):
    acquire(?rw->lock)
    rw->nreaders -= 1
    release(?rw->lock)
```

```
def read_acquire(rw):
    acquire(?rw->mutex)
    while rw->nwriters > 0:
        wait(?rw->r_cond, ?rw->mutex)
    rw->nreaders += 1
    release(?rw->mutex)

def read_release(rw):
    acquire(?rw->mutex)
    rw->nreaders -= 1
    if rw->nreaders == 0:
        notify(?rw->w_cond)
    release(?rw->mutex)
```

# R/W Lock, reader part

compare with busy waiting

```
def read_acquire(rw):
    acquire(?rw->lock)
    while rw->nwriters > 0:
        release(?rw->lock)
        acquire(?rw->lock)
    rw->nreaders += 1
    release(?rw->lock)

def read_release(rw):
    acquire(?rw->lock)
    rw->nreaders -= 1
    release(?rw->lock)
```

```
def read_acquire(rw):
    acquire(?rw->mutex)
    while rw->nwriters > 0:
        wait(?rw->r_cond, ?rw->mutex)
    rw->nreaders += 1
    release(?rw->mutex)

def read_release(rw):
    acquire(?rw->mutex)
    rw->nreaders -= 1
    if rw->nreaders == 0:
        notify(?rw->w_cond)
    release(?rw->mutex)
```

# R/W Lock, writer part

```
23 def write_acquire(rw):
24     acquire(?rw->mutex)
25     while rw->nreaders > 0 or rw->nwriters > 0:
26         wait(?rw->w_cond, ?rw->mutex)
27         rw->nwriters = 1
28         release(?rw->mutex)
29
30 def write_release(rw):
31     acquire(?rw->mutex)
32     rw->nwriters = 0
33     notify_all(?rw->r_cond)
34     notify(?rw->w_cond)
35     release(?rw->mutex)
```

} don't forget anybody!

# Condition Variable interface

- **wait(*cv*, *lock*)**
  - may only be called while holding *lock*
  - temporarily releases *lock*
    - but re-acquires it before resuming
  - if *cv* not notified, may block indefinitely
    - but wait() may resume "on its own"
- **notify(*cv*)**
  - no-op if nobody is waiting on *cv*
  - otherwise wakes up at least one thread waiting on *cv*
- **notify\_all(*cv*)**
  - wakes up all threads currently waiting on *cv*

# Busy Waiting or?

```
def test_and_set(s) returns oldvalue:  
    atomically:  
        oldvalue = !s  
        !s = True  
  
def atomic_store(p, v):  
    atomically !p = v  
  
def Lock() returns initvalue:  
    initvalue = False  
  
def acquire(lk):  
    while test_and_set(lk):  
        pass  
  
def release(lk):  
    atomic_store(lk, False)
```

```
def read_acquire(rw):  
    acquire(?rw->lock)  
    while rw->nwriters > 0:  
        release(?rw->lock)  
        acquire(?rw->lock)  
    rw->nreaders += 1  
    release(?rw->lock)  
  
def read_release(rw):  
    acquire(?rw->lock)  
    rw->nreaders -= 1  
    release(?rw->lock)
```

# Busy Waiting or?

```
def test_and_set(s) returns oldvalue:  
    atomically:  
        oldvalue = !s  
        !s = True  
  
def atomic_store(p, v):  
    atomically !p = v  
  
def Lock() returns initvalue:  
    initvalue = False  
  
def acquire(lk):  
    while test_and_set(lk)  
        pass  
  
def release(lk):  
    atomic_store(lk, False)
```

```
def read_acquire(rw):  
    acquire(?rw->lock)  
    while rw->nwriters > 0:  
        release(?rw->lock)  
        acquire(?rw->lock)  
    rw->nreaders += 1  
    release(?rw->lock)  
  
def read_release(rw):  
    acquire(?rw->lock)  
    rw->nreaders -= 1  
    release(?rw->lock)
```

# Busy Waiting or?

```
def test_and_set(s) returns oldvalue:  
    atomically:  
        oldvalue = !s  
        !s = True  
  
def atomic_store(p, v):  
    atomically !p = v  
  
def Lock() returns initvalue:  
    initvalue = False  
  
def acquire(lk):  
    while test_and_set(lk)  
        pass  
  
def release(lk):  
    atomic_store(lk, False)
```

State unchanged while condition does not hold. This thread only “observes” the state until condition holds

```
def read_acquire(rw):  
    acquire(?rw->lock)  
    while rw->nwriters > 0:  
        release(?rw->lock)  
        acquire(?rw->lock)  
        rw->nreaders += 1  
        release(?rw->lock)  
  
def read_release(rw):  
    acquire(?rw->lock)  
    rw->nreaders -= 1  
    release(?rw->lock)
```

State conditionally changes while condition does not hold. This thread actively changes the state until the condition hold

# Busy Waiting or?

```
def test_and_set(s) returns oldvalue:  
    atomically:  
        oldvalue = !s  
        !s = True  
  
def atomic_store(p, v):  
    atomically !p = v  
  
def Lock() returns initvalue:  
    initvalue = False  
  
def acquire(lk):  
    while test_and_set(lk)  
        pass  
  
def release(lk):  
    atomic_store(lk, False)
```

State unchanged while condition does not hold. This thread only “observes” the state until condition holds

```
def read_acquire(rw):  
    acquire(?rw->lock)  
    while rw->nwriters > 0:  
        release(?rw->lock)  
        acquire(?rw->lock)  
        rw->nreaders += 1  
        release(?rw->lock)  
  
def read_release(rw):  
    acquire(?rw->lock)  
    rw->nreaders -= 1  
    release(?rw->lock)
```

**Busy Waiting**

State conditionally changes while condition does not hold. This thread actively changes the state until the condition hold

# Why is busy waiting bad?

- Consider a timesharing setting
- Threads T1 and T2 take turns on the CPU
  - switch every 100 milliseconds
- Suppose T1 has a write lock and is running
- Now suppose a clock interrupt occurs, T2 starts running and tries to acquire a read lock
- **Non-busy-waiting acquisition:**
  - T2 is put on a waiting queue and T1 resumes immediately and runs until T1 releases the write lock
    - which puts T2 back on the run queue
- **Busy-waiting acquisition:**
  - T2 keeps running (wasting CPU) until the next clock interrupt
  - T1 and T2 switch back and forth every 100 ms until T1 releases the write lock

# Busy Waiting vs Condition Variables

Busy Waiting	Condition Variables
Use a lock and a loop	Use a lock <i>and a collection of condition variables</i> and a loop
Easy to write the code	Notifying is tricky
Easy to understand the code	Easy to understand the code
Progress property is easy	Progress requires careful consideration (both for correctness and efficiency)
Ok-ish for true multi-core, but bad for virtual threads	Good for both multi-core and virtual threading

# Just Say No to Busy Waiting

# Why no naked waits? (reason 1)

A **naked wait** is a `wait()` without **while** around it

By the time waiter gets the lock back, condition may no longer hold

- E.g., given three threads: W1, R2, W3
- W1 enters as a writer
- R2 waits as a reader
- W1 leaves, notifying R2
- W3 enters as a writer
- R2 wakes up
  - If R2 doesn't check condition again, R2 and W3 would both be in the critical section

# Why no naked waits? (reason 2)

- When notifying, be safe rather than sorry
  - it's better to notify too many threads than too few
  - in case of doubt, use `notify_all()` instead of just `notify()`
- Over-notifying can lead to some threads waking up when their condition is no longer satisfied

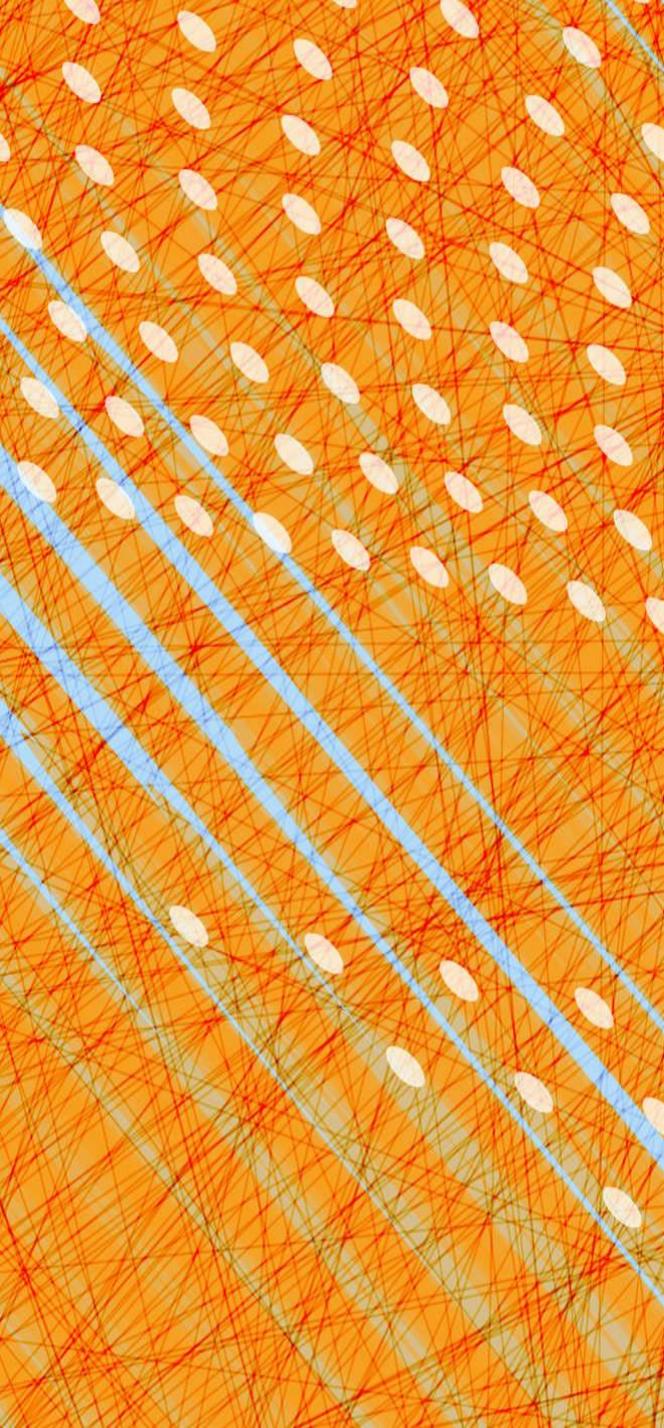
# Why no naked waits? (reason 3)

- Because you **should** use **while** around **wait**, many condition variable implementations allow “**spurious wakeups**”
  - wait() resumes even though condition variable was not notified
  - simplifies implementation of wait()

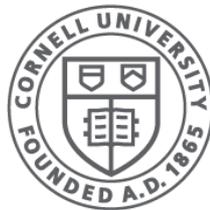
# Just Say No to Naked Waits

# Hints for reducing unneeded wakeups

- Use separate condition variables for each waiting condition
- Don't use **notify\_all** when **notify** suffices
  - but be safe rather than sorry
- You can use N calls to **notify** if you know at most N nodes can continue after a waiting condition holds



# Deadlock



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Deadlock example

```
1 from synch import Lock, acquire, release
2
3 def Account(balance) returns account:
4     account = { .lock: Lock(), .balance: balance }
5
6 accounts = [ Account(3), Account(7), Account(0) ]
7
8 def transfer(a1, a2, amount):
9     acquire(?accounts[a1].lock)
10    if amount <= accounts[a1].balance:
11        accounts[a1].balance -= amount
12        acquire(?accounts[a2].lock)
13        accounts[a2].balance += amount
14        release(?accounts[a2].lock)
15        release(?accounts[a1].lock)
16
17 spawn transfer(0, 1, 1)
18 spawn transfer(1, 0, 2)
```

What could go wrong?

# Harmony output

**Summary: some execution cannot terminate**

- **Schedule** thread T0: init()
  - Line 6:** Set accounts to [ { "balance": 3, "lock": False }, { "balance": 7, "lock": False } ]
- **Schedule** thread T1: transfer(0, 1, 1)
  - Line synch/36:** Set accounts[0]["lock"] to True (was False)
  - Line 11:** Set accounts[0]["balance"] to 2 (was 3)
  - Preempted in** transfer(0, 1, 1) --> acquire(?accounts[1]["lock"])
- **Schedule** thread T2: transfer(1, 0, 2)
  - Line synch/36:** Set accounts[1]["lock"] to True (was False)
  - Line 11:** Set accounts[1]["balance"] to 5 (was 7)
  - Preempted in** transfer(1, 0, 2) --> acquire(?accounts[0]["lock"])

**Final state (all threads have terminated or are blocked):**

**Threads:**

T1: (**blocked**) transfer(0, 1, 1) --> acquire(?accounts[1]["lock"])

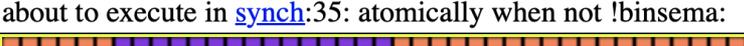
T2: (**blocked**) transfer(1, 0, 2) --> acquire(?accounts[0]["lock"])

**Variables:**

accounts: [ { "balance": 2, "lock": True }, { "balance": 5, "lock": True } ]

# Harmony HTML Output

**Issue: Non-terminating state**

Issue: Non-terminating state			Shared Variables	
Turn	Thread	Instructions Executed	PC	<i>accounts</i>
1	T0: <code>__init__()</code>	 terminated	1309	[ { "balance": 3, "lock": False }, { "balance": 7, "lock": False } ]
2	T1: <code>transfer(0, 1, 1)</code>	 about to execute in <a href="#">synch:35</a> : atomically when not !binsema:	949	[ { "balance": 2, "lock": True }, { "balance": 7, "lock": False } ]
3	T2: <code>transfer(1, 0, 2)</code>	 about to execute in <a href="#">synch:35</a> : atomically when not !binsema:	949	[ { "balance": 2, "lock": True }, { "balance": 5, "lock": True } ]

[synch:34](#) **def acquire(binsema):**

		Threads			
		ID	Status	Stack Trace	Stack Top
934	LoadVar binsema	T0	terminated	<code>__init__()</code>	
935	DelVar binsema				
936	Load				
937	StoreVar result	T1	blocked	<code>transfer(0, 1, 1)</code> <code>acquire(?accounts[1]["lock"])</code>	a1: 0, a2: 1, amount: 1 binsema: ?accounts[1]["lock"]
938	ReturnOp(result)				
939	Jump 1214	T2	blocked	<code>transfer(1, 0, 2)</code> <code>acquire(?accounts[0]["lock"])</code>	a1: 1, a2: 0, amount: 2 binsema: ?accounts[0]["lock"]
940	Frame held(binsema)				

# Deadlock vs Starvation

- **Starvation**: some processes can run in theory, but the **scheduler** continually selects other processes to run first. Tied to **fairness** in scheduling.
- **Deadlock**: no process can run because all are waiting for another process to change the state. The **scheduler** can't help you now.

# Deadlock vs Livelock

- **Livelock**: some processes continually change their state but don't make progress (like polite people trying to pass one another in a narrow hallway). The **scheduler** could fix this in theory.
- **Deadlock**: no process can run because all are waiting for another process to change the state. The **scheduler** can't help you now.

# System Model

- Collection of **resources** and **threads**
  - Examples of resources: I/O devices, GPUs, locks, buffers, slots in a buffer, ...
- **Exclusive access**
  - Only one thread can use a resource at a time
  - Protocol:
    1. Thread **acquires** resource
      - thread is blocked until resource is free
    2. Thread **holds** the resource
      - resource is allocated (not free) at this time
    3. Thread **releases** the resource

# Necessary Conditions for Deadlock

Edward Coffman 1971

## 1. Mutual Exclusion

- acquire() can block invoker until resource is free

## 2. Hold & wait

- A thread can be blocked while holding resources

## 3. No preemption

- Allocated resources cannot be reclaimed

## 4. Circular wait

- Let  $T_i \rightarrow T_j$  denote “ $T_i$  waits for  $T_j$  to release a resource”.
- Then  $\exists T_1, \dots, T_n : T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$

# Example: Mutual Exclusion

```
1 from synch import Lock, acquire, release
2
3 def Account(balance) returns account:
4     account = { .lock: Lock(), .balance: balance }
5
6 accounts = [ Account(3), Account(7), Account(0) ]
7
8 def transfer(a1, a2, amount):
9     acquire(?accounts[a1].lock)
10    if amount <= accounts[a1].balance:
11        accounts[a1].balance -= amount
12        acquire(?accounts[a2].lock)
13        accounts[a2].balance += amount
14        release(?accounts[a2].lock)
15        release(?accounts[a1].lock)
16
17 spawn transfer(0, 1, 1)
18 spawn transfer(1, 0, 2)
```



Mutual exclusion



Mutual exclusion

# Example: Hold & Wait

```
1  from synch import Lock, acquire, release
2
3  def Account(balance) returns account:
4      account = { .lock: Lock(), .balance: balance }
5
6  accounts = [ Account(3), Account(7), Account(0) ]
7
8  def transfer(a1, a2, amount):
9      acquire(?accounts[a1].lock)
10     if amount <= accounts[a1].balance:
11         accounts[a1].balance -= amount
12         acquire(?accounts[a2].lock)
13         accounts[a2].balance += amount
14         release(?accounts[a2].lock)
15         release(?accounts[a1].lock)
16
17  spawn transfer(0, 1, 1)
18  spawn transfer(1, 0, 2)
```



Thread holds a1.lock



Thread wants a2.lock

# Example: No Preemption

```
1  from synch import Lock, acquire, release
2
3  def Account(balance) returns account:
4      account = { .lock: Lock(), .balance: balance }
5
6  accounts = [ Account(3), Account(7), Account(0) ]
7
8  def transfer(a1, a2, amount):
9      acquire(?accounts[a1].lock)
10     if amount <= accounts[a1].balance:
11         accounts[a1].balance -= amount
12         acquire(?accounts[a2].lock)
13         accounts[a2].balance += amount
14         release(?accounts[a2].lock)
15         release(?accounts[a1].lock)
16
17  spawn transfer(0, 1, 1)
18  spawn transfer(1, 0, 2)
```



Only holder can release lock

# Example: Circular Wait

```
1  from synch import Lock, acquire, release
2
3  def Account(balance) returns account:
4      account = { .lock: Lock(), .balance: balance }
5
6  accounts = [ Account(3), Account(7), Account(0) ]
7
8  def transfer(a1, a2, amount):
9      acquire(?accounts[a1].lock)
10     if amount <= accounts[a1].balance:
11         accounts[a1].balance -= amount
12         acquire(?accounts[a2].lock)
13         accounts[a2].balance += amount
14         release(?accounts[a2].lock)
15     release(?accounts[a1].lock)
16
17  spawn transfer(0, 1, 1)
18  spawn transfer(1, 0, 2)
```



Circular wait conditions

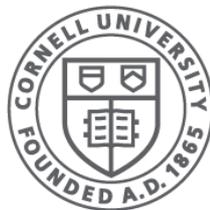
# Three ways to deal with deadlock

Prevention: **Programmer** ensures that at least one of the necessary conditions cannot hold

Avoidance: **Scheduler** avoids deadlock scenarios (e.g., by executing each thread to completion)

Detect and Recover: Allow deadlocks to happen. Detect them and recover in some way

# Deadlock Prevention



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Negate one of the following:

- 1. Mutual Exclusion**
- 2. Hold & wait**
- 3. No preemption**
- 4. Circular wait**

# 1. Negate Mutual Exclusion

- Make resources sharable without locks
  - Non-blocking concurrent data structures
    - See Harmony book for examples
- Have sufficient resources available so `acquire()` never blocks
  - bounded buffer: make sure it is large enough
    - Doesn't work for locks, as there is only one per critical section

## 2. Negate Hold & Wait

```
1  from synch import Lock, acquire, release
2
3  def Account(balance) returns account:
4      account = { .lock: Lock(), .balance: balance }
5
6  accounts = [ Account(3), Account(7) ]
7
8  def transfer(a1, a2, amount):
9      acquire(?accounts[a1].lock)
10     if amount <= accounts[a1].balance:
11         accounts[a1].balance -= amount
12         release(?accounts[a1].lock)
13         acquire(?accounts[a2].lock)
14         accounts[a2].balance += amount
15         release(?accounts[a2].lock)
16     else:
17         release(?accounts[a1].lock)
18
19  spawn transfer(0, 1, 1)
20  spawn transfer(1, 0, 2)
```



Release resource  
before acquiring another

# 2: Negate Hold & Wait, badly

```
1 from synch import Lock, acquire, release
2
3 def Account(balance) returns account:
4     account = { .lock: Lock(), .balance: balance }
5
6 accounts = [ Account(3), Account(7) ]
7
8 invariant all(a.balance >= 0 for a in accounts)
9
10 def transfer(a1, a2, amount):
11     acquire(?accounts[a1].lock)
12     var funds_available = amount <= accounts[a1].balance
13     release(?accounts[a1].lock)
14     if funds_available:
15         acquire(?accounts[a1].lock)
16         accounts[a1].balance -= amount
17         release(?accounts[a1].lock)
18         acquire(?accounts[a2].lock)
19         accounts[a2].balance += amount
20         release(?accounts[a2].lock)
21
22 spawn transfer(0, 1, 2)
23 spawn transfer(0, 1, 2)
```

} check if funds are available

} withdraw funds from a1

} deposit funds for a2

What could go wrong?

## 2. Negate Hold & Wait, alternate

```
1 def Lock() returns lock:
2   lock = False
3
4 def acquire2(lk1, lk2):
5   atomically when not (!lk1 or !lk2):
6     !lk1 = !lk2 = True
7
8 def release(lk):
9   atomically !lk = False
10
11 def Account(balance) returns account:
12   account = { .lock: Lock(), .balance: balance }
13
14 accounts = [ Account(3), Account(7) ]
15
16 def transfer(a1, a2, amount):
17   acquire2(?accounts[a1].lock, ?accounts[a2].lock)
18   if amount <= accounts[a1].balance:
19     accounts[a1].balance -= amount
20     accounts[a2].balance += amount
21   release(?accounts[a1].lock)
22   release(?accounts[a2].lock)
23
24 spawn transfer(0, 1, 1)
25 spawn transfer(1, 0, 2)
```



Spec: Acquire two locks



Acquire resources at  
the same time

# 3. Allow Preemption

- Time-multiplexing of resources
  - threads: context switching
  - memory: paging
- Database transactions
  - 2-phase locking + transaction abort and retry
- Not available for locks

# 4: Negate circular wait

- Define a **total order** on resources
- Rule: a thread cannot acquire a resource that is “**lower**” than a resource already held
- Either:
  - a thread is careful to acquire resources that it needs in order, **or**
  - a thread that wants to acquire a resource R must first release all resources that are lower than R

# Why does resource ordering work?

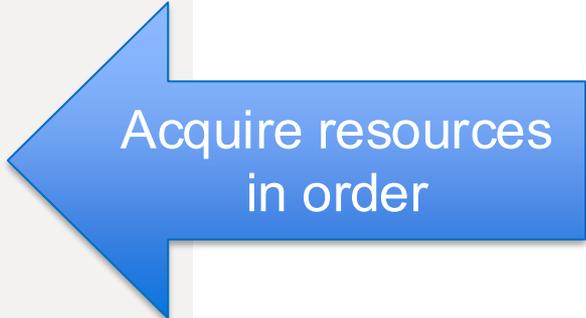
**Theorem:** Resource ordering prevents circular wait

## **Proof by contradiction:**

- Assume circular wait exists
- $\exists T_1, \dots, T_n : T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$
- $T_i$  holds  $R_i$
- $T_i$  requests  $R_j$  held by  $T_j$  ( $j = (i + 1) \bmod n$ )
- Resource ordering:  $R_1 < R_2, \dots, R_{n-1} < R_n, R_n < R_1$
- $R_1 < R_1$  (by *transitivity* of total order)
- Violates *irreflexivity* of total order

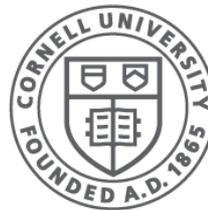
# 4: Negate circular wait

```
1  from synch import Lock, acquire, release
2
3  def Account(balance) returns account:
4      account = { .lock: Lock(), .balance: balance }
5
6  accounts = [ Account(3), Account(7) ]
7
8  def transfer(a1, a2, amount):
9      acquire(?accounts[min(a1, a2)].lock)
10     acquire(?accounts[max(a1, a2)].lock)
11     if amount <= accounts[a1].balance:
12         accounts[a1].balance -= amount
13         accounts[a2].balance += amount
14     release(?accounts[a1].lock)
15     release(?accounts[a2].lock)
16
17  spawn transfer(0, 1, 1)
18  spawn transfer(1, 0, 2)
```



Acquire resources  
in order

# Deadlock Avoidance



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Deadlock in traffic



How can these be avoided?

# Deadlock Avoidance

- Scheduler carefully schedules threads so deadlock cannot occur
- For example, it might allow only one thread to run at a time, to completion
  - This is extreme: no concurrency
  - Doesn't work with conditional waiting
- Better solutions typically require that the scheduler has some abstract knowledge of what the threads are trying to accomplish

# Safe States

- A *state* is an allocation of resources to threads
- The state changes each time a thread allocates or releases a resource
- A *safe state* is a state from which an execution exists that does not cause deadlock
- Notes:
  - the initial state is safe: threads can be scheduled one at a time and run to completion
  - an unsafe state is not necessarily deadlocked, but *deadlock is unavoidable eventually*
  - deadlock may be possible from a safe state, but it is *avoidable* through careful scheduling

# Deadlock Avoidance

- Scheduler should only allow safe states to happen in an execution
  - When a thread tries to **acquire()** a resource, the scheduler should **block** the thread, **if acquiring the resource leads to an unsafe state, until this is no longer the case**
  - **release()** is always ok

# Deadlock Avoidance

```
1  from synch import Lock, acquire, release
2
3  def Account(balance) returns account:
4      account = { .lock: Lock(), .balance: balance }
5
6  accounts = [ Account(3), Account(7), Account(0) ]
7
8  def transfer(a1, a2, amount):
9      acquire(?accounts[a1].lock)
10     if amount <= accounts[a1].balance:
11         accounts[a1].balance -= amount
12         acquire(?accounts[a2].lock)
13         accounts[a2].balance += amount
14         release(?accounts[a2].lock)
15         release(?accounts[a1].lock)
16
17  spawn transfer(0, 1, 1)
18  spawn transfer(1, 0, 2)
```

How?

# Deadlock Avoidance

```
1 from synch import Lock, acquire, release
2
3 def Account(balance) returns account:
4     account = { .lock: Lock(), .balance: balance }
5
6 accounts = [ Account(3), Account(7), Account(0) ]
7
8 def transfer(a1, a2, amount):
9     acquire(?accounts[a1].lock)
10    if amount <= accounts[a1].balance:
11        accounts[a1].balance -= amount
12        acquire(?accounts[a2].lock)
13        accounts[a2].balance += amount
14        release(?accounts[a2].lock)
15        release(?accounts[a1].lock)
16
17 spawn transfer(0, 1, 1)
18 spawn transfer(1, 0, 2)
```

For example, don't schedule two threads  $\text{transfer}(a1, a2)$  and  $\text{transfer}(a3, a4)$  at the same time unless  $\{a1, a2\} \cap \{a3, a4\} = \emptyset$

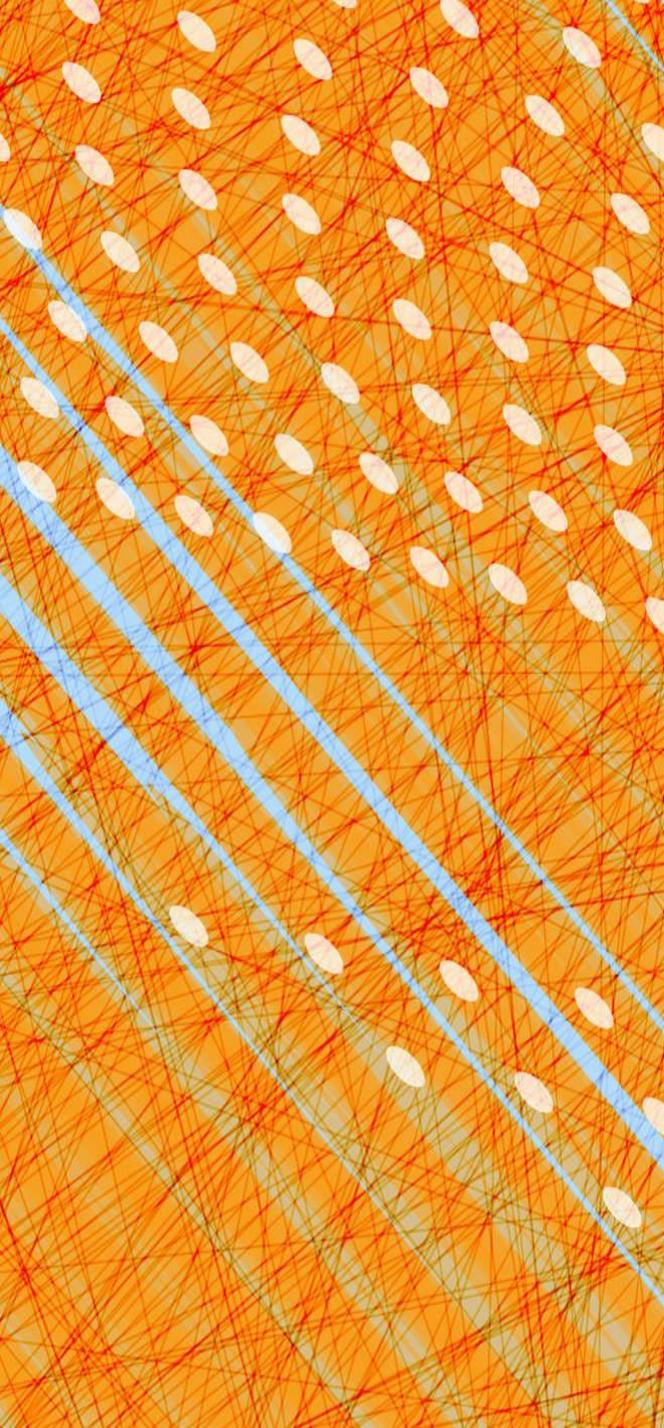
# Avoidance specified in Harmony

```
1  from synch import Lock, acquire, release
2
3  def Account(balance) returns account:
4      account = { .lock: Lock(), .balance: balance }
5
6  active = {}
7  accounts = [ Account(3), Account(7) ]
8
9  def transfer(a1, a2, amount):
10     atomically when ({ a1, a2 } & active) == {}:
11         active |= { a1, a2 }
12
13         acquire(?accounts[a1].lock)
14         if amount <= accounts[a1].balance:
15             accounts[a1].balance -= amount
16             acquire(?accounts[a2].lock)
17             accounts[a2].balance += amount
18             release(?accounts[a2].lock)
19         release(?accounts[a1].lock)
20
21     atomically:
22         active -= { a1, a2 }
```

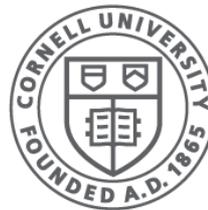
keep track of which  
accounts are active

enforce no intersection  
with active transfers

update scheduler state



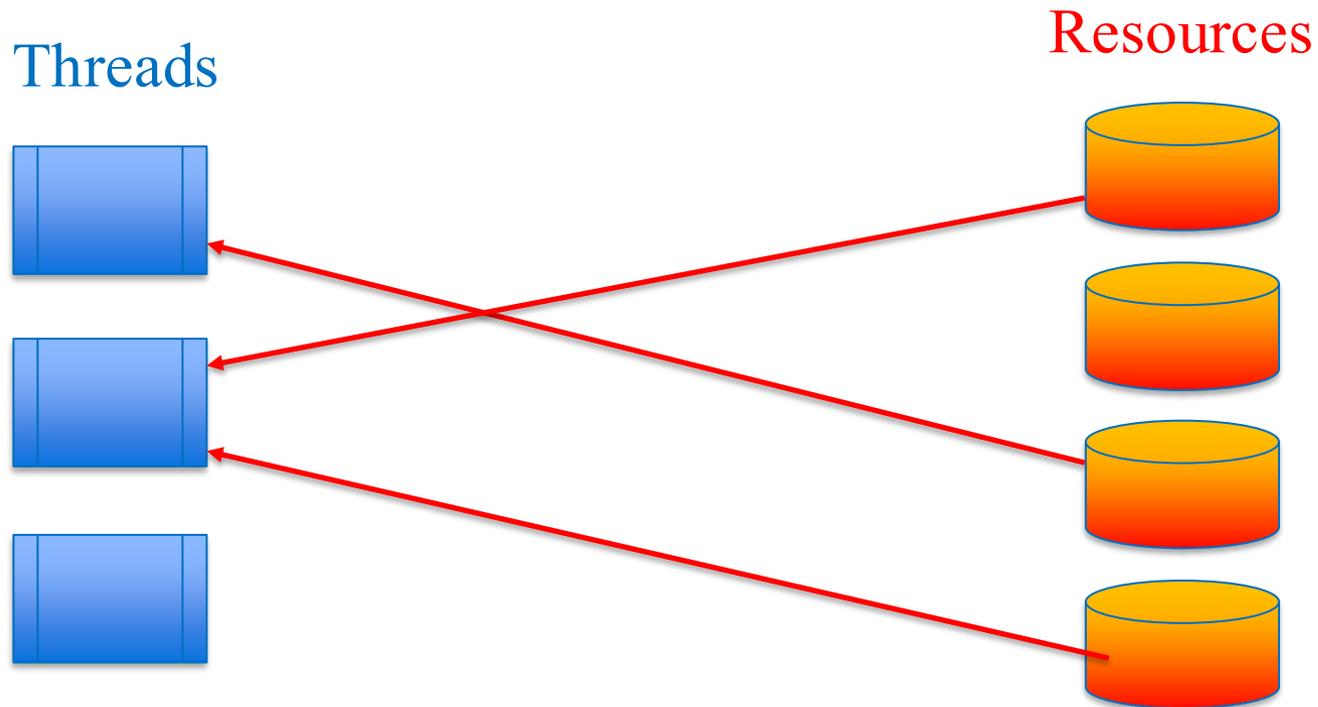
# Deadlock Detection and Recovery



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

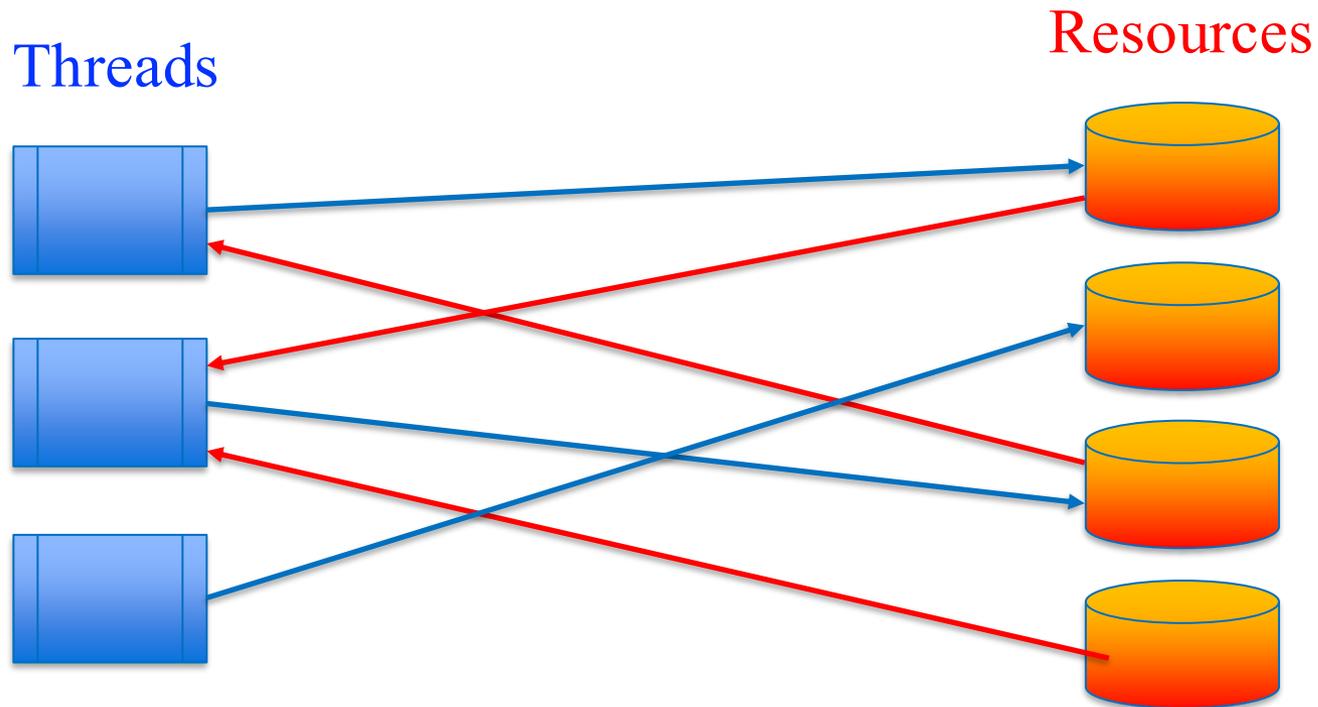
# Deadlock Detection

- Keep track of allocation of **resources** to threads



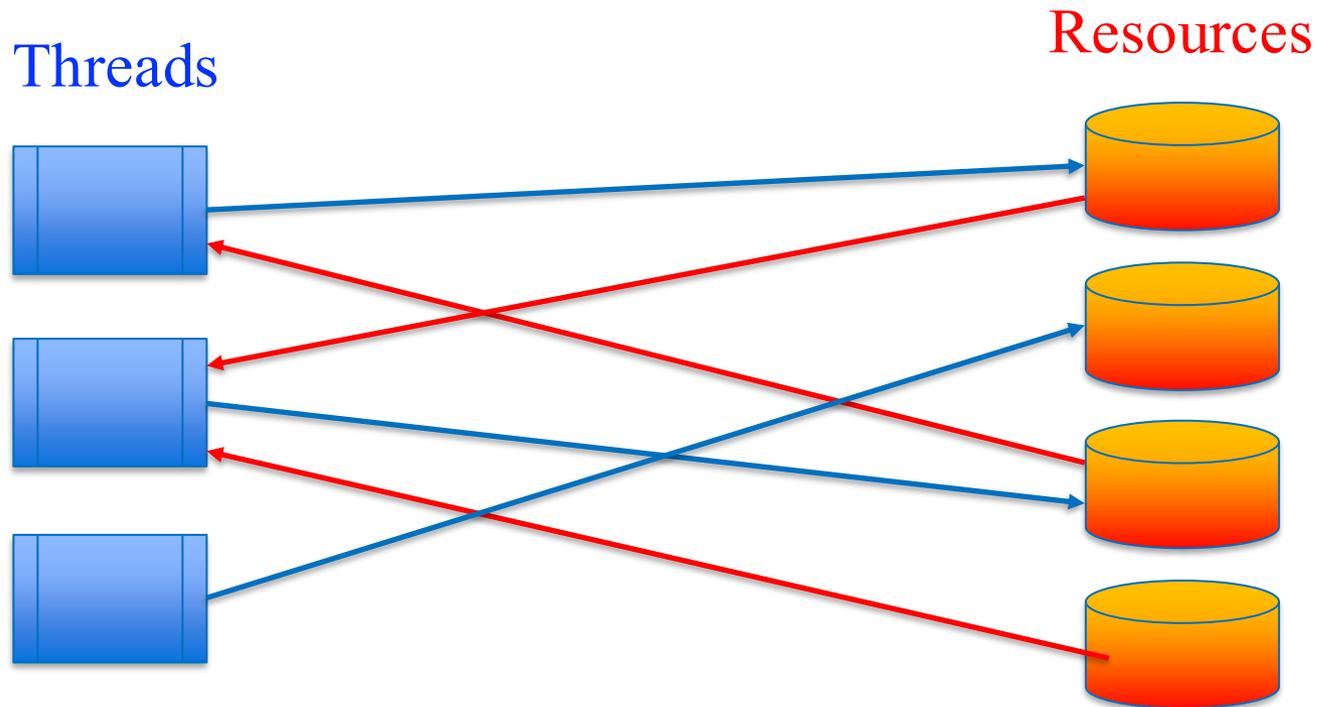
# Deadlock Detection

- Keep track of allocation of **resources** to threads
- Keep track of which **threads** are trying to acquire which **resource**



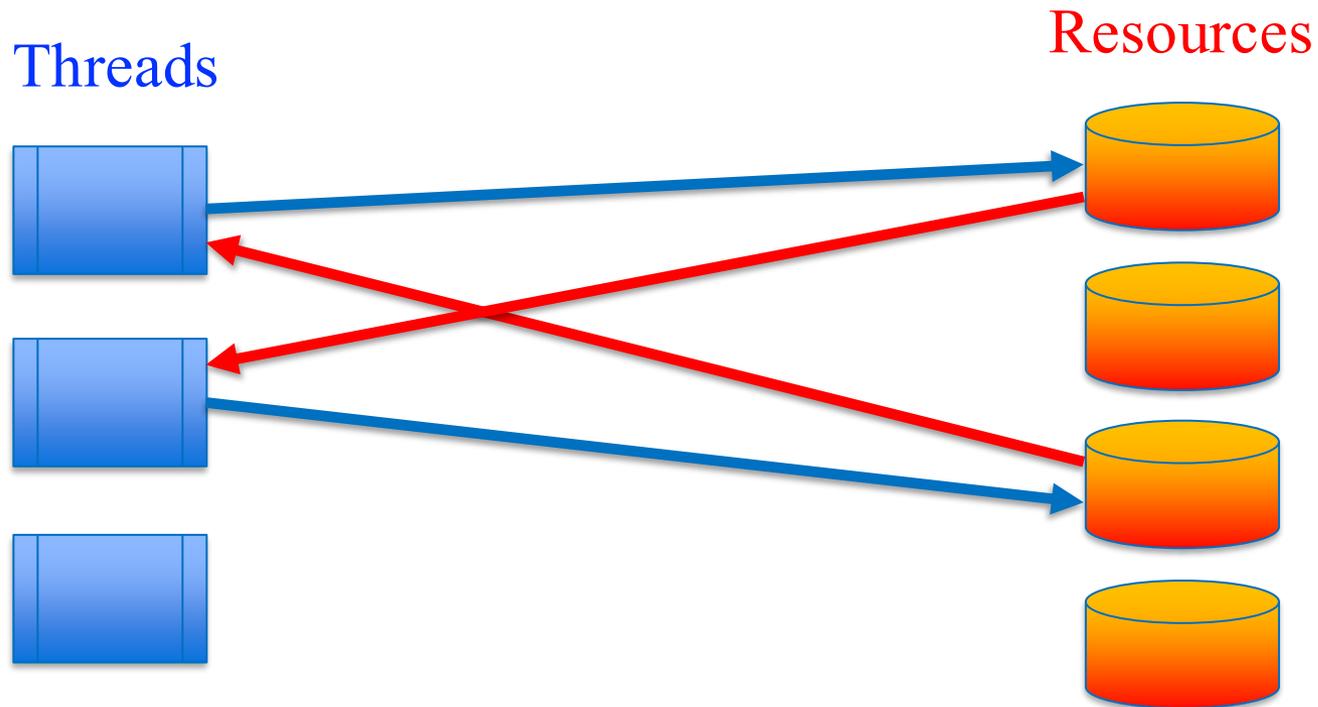
# Deadlock Detection

- Known as the **Resource Allocation Graph**
- Deadlock  $\equiv$  cycle in the graph



# Deadlock Detection

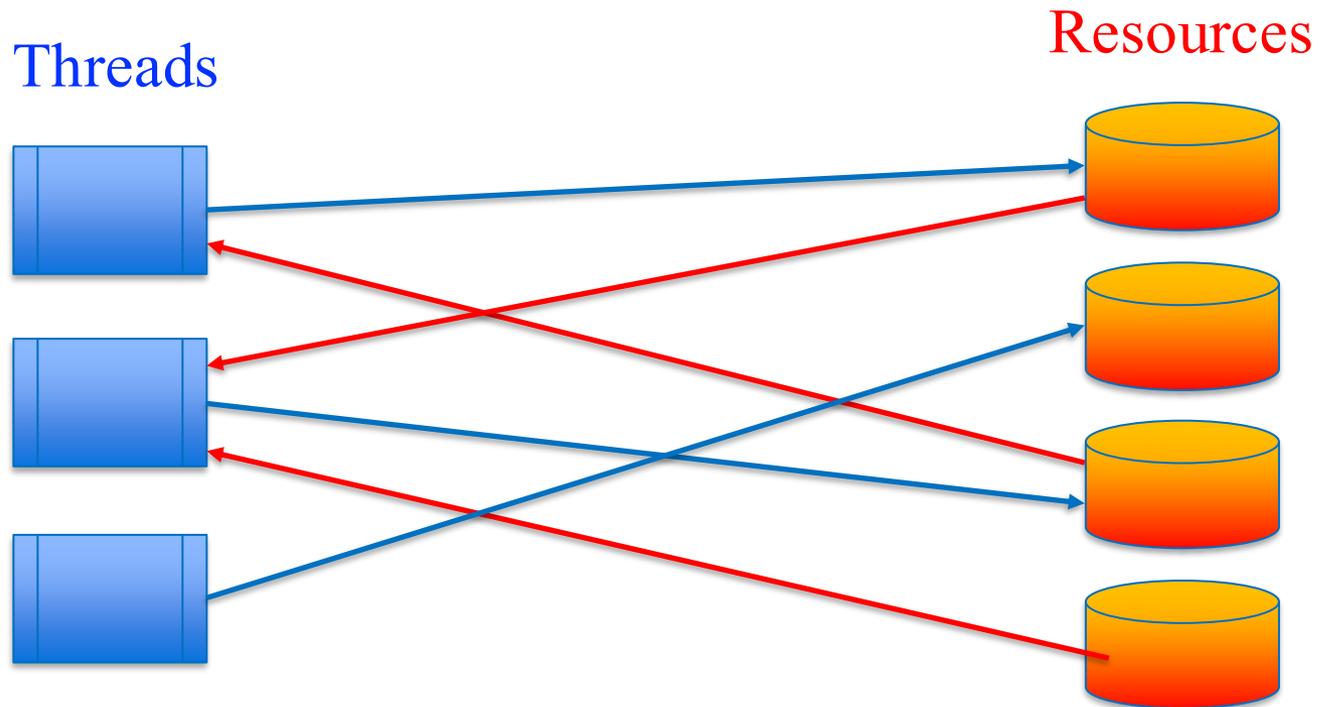
- Known as the **Resource Allocation Graph**
- Deadlock  $\equiv$  cycle in the graph



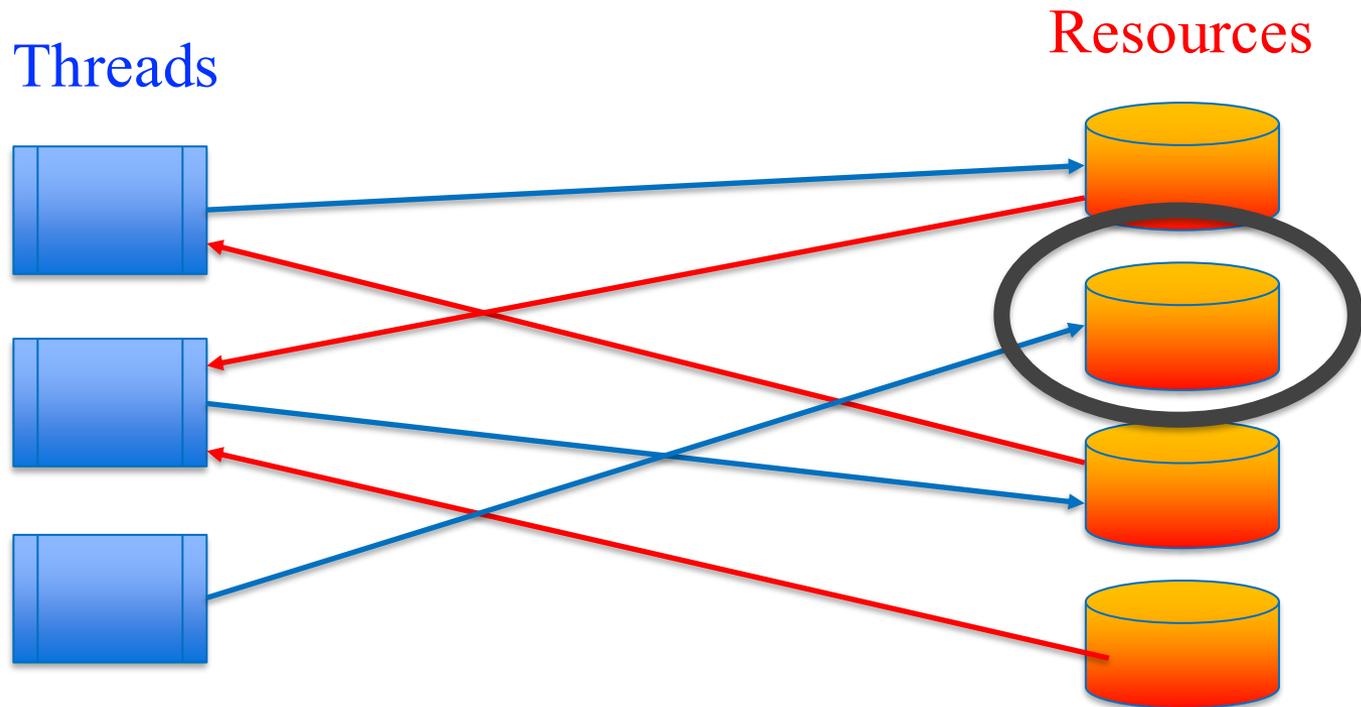
# Finding Cycles

- Graph Reduction Algorithm:
  - While there are nodes with no outgoing edges
    - select one such node
    - remove node and its incoming edges
  - If the resulting graph empty (no nodes), then no cycles
  - No cycles  $\implies$  No deadlock

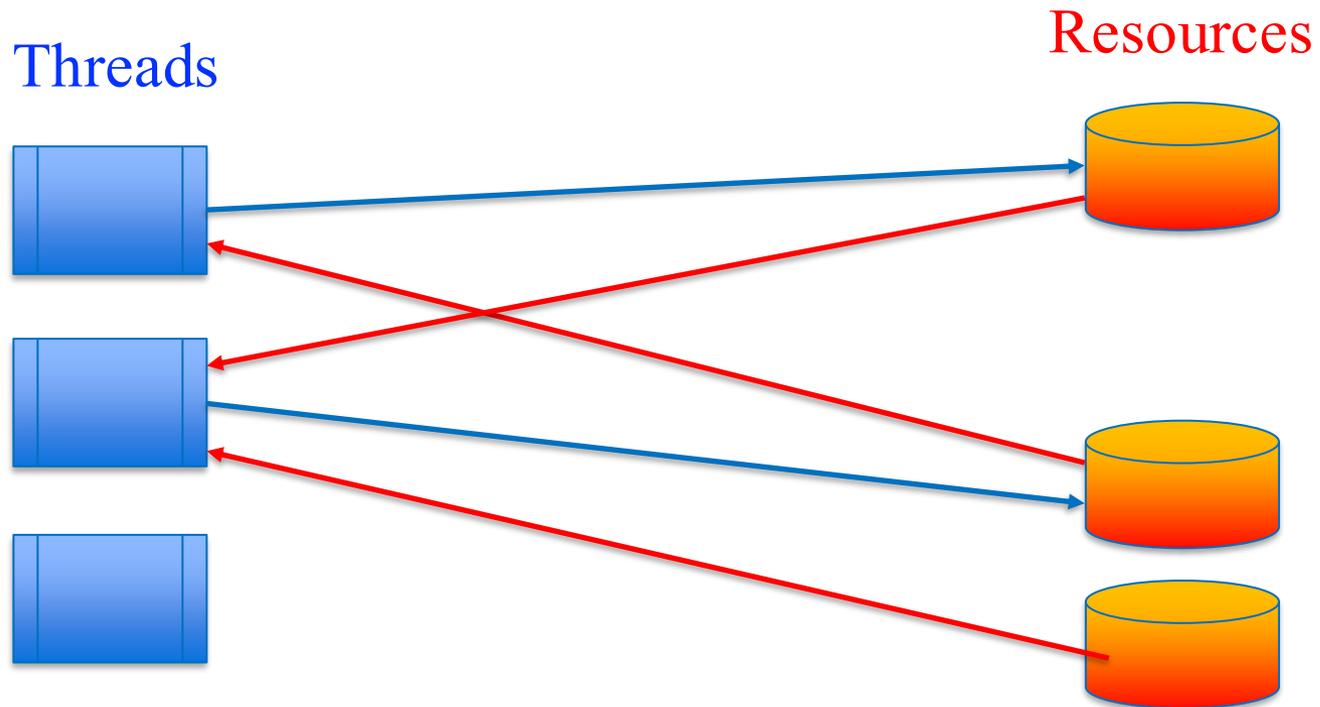
# Deadlock Detection



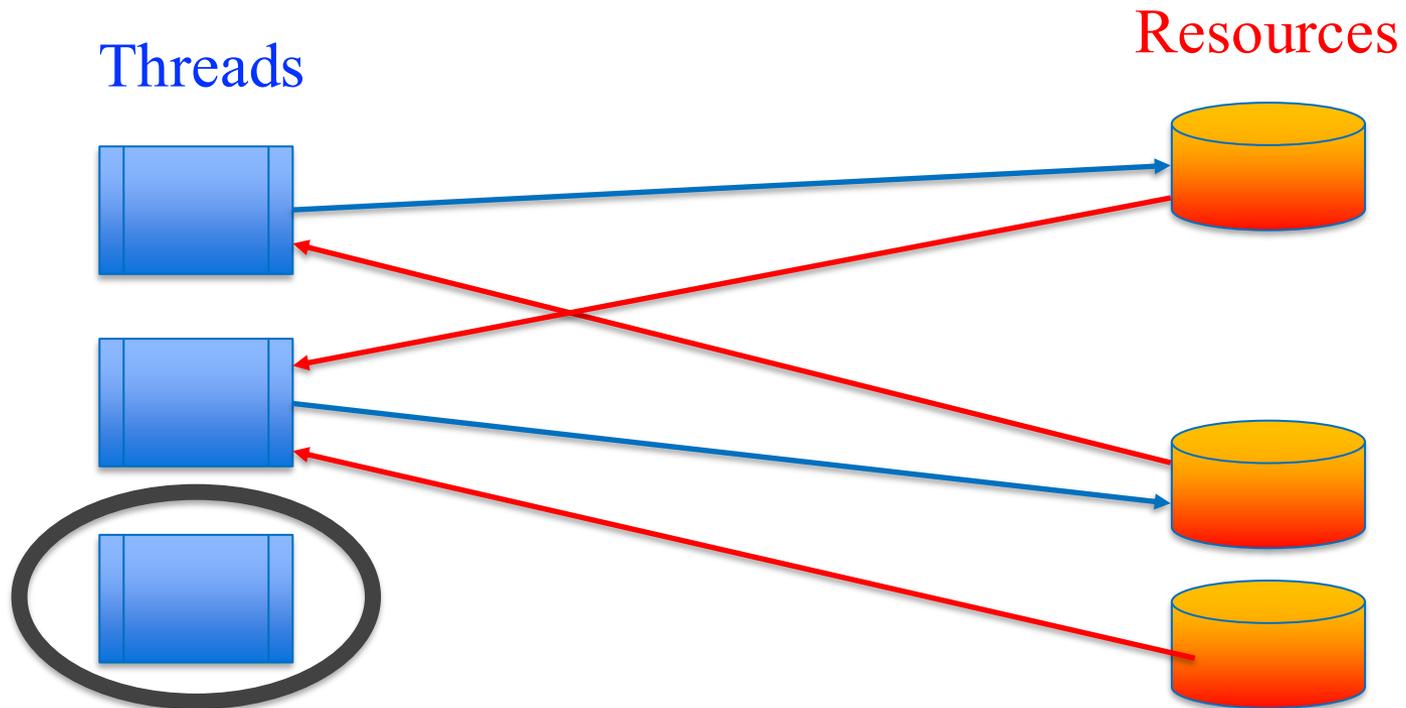
# Deadlock Detection



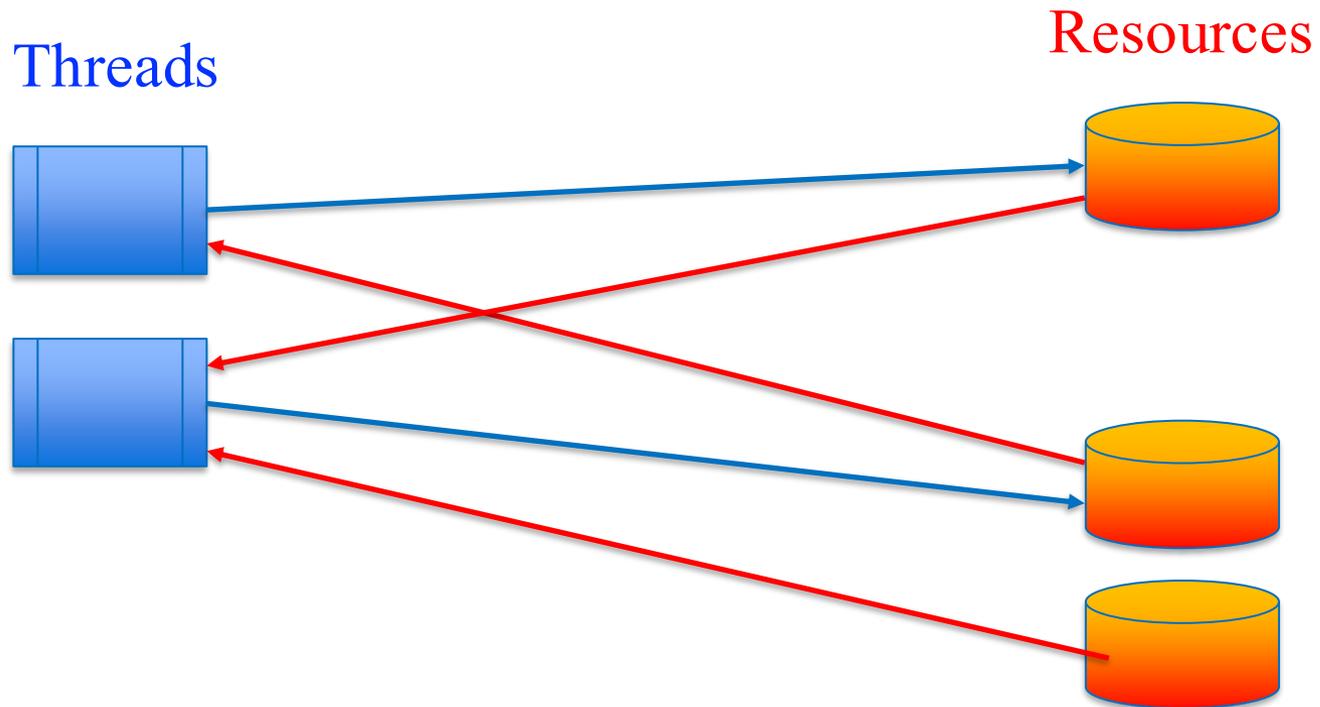
# Deadlock Detection



# Deadlock Detection

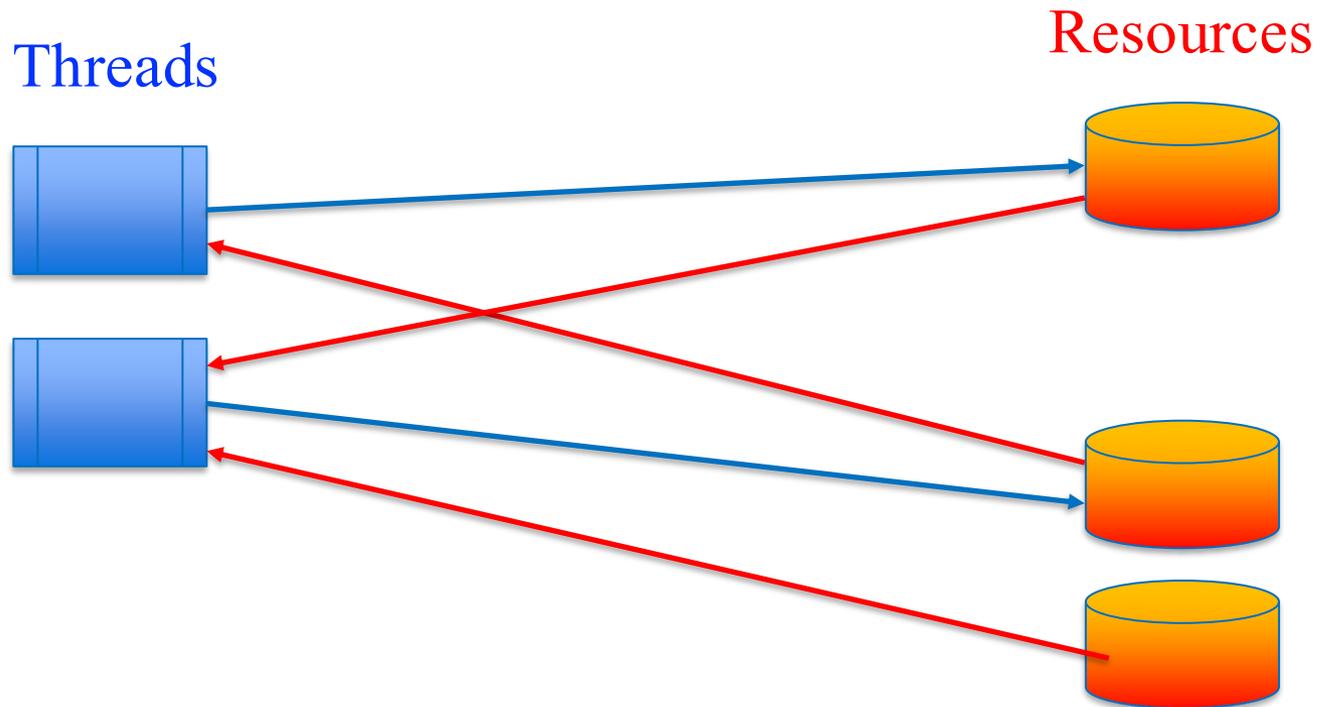


# Deadlock Detection



# Deadlock Detection

No more nodes can be removed, but graph is non-empty →  
*cycle is present*



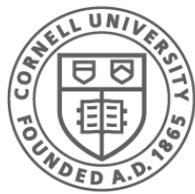
# Deadlock Detection

- Deadlock detection is expensive
- When to run graph reduction?
  - When a resource request cannot be granted?
  - When a thread has been blocked for a certain amount of time?
  - Periodically?

# Deadlock Recovery Strategies

- Blue screen and reboot
  - Can lose data / results of long computations
- Deny a request to remove cycle
  - Programmer responsible for exception
- Kill processes until cycle is gone
  - Can lose data / results of long computations
  - Select processes that have been running shortest amount of time
- Use transactions to access resources
  - Abort and retry transaction if deadlock exists
  - Requires roll-back or versioning of state

# Actors



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

[Robbert van Renesse]

# Actor Model

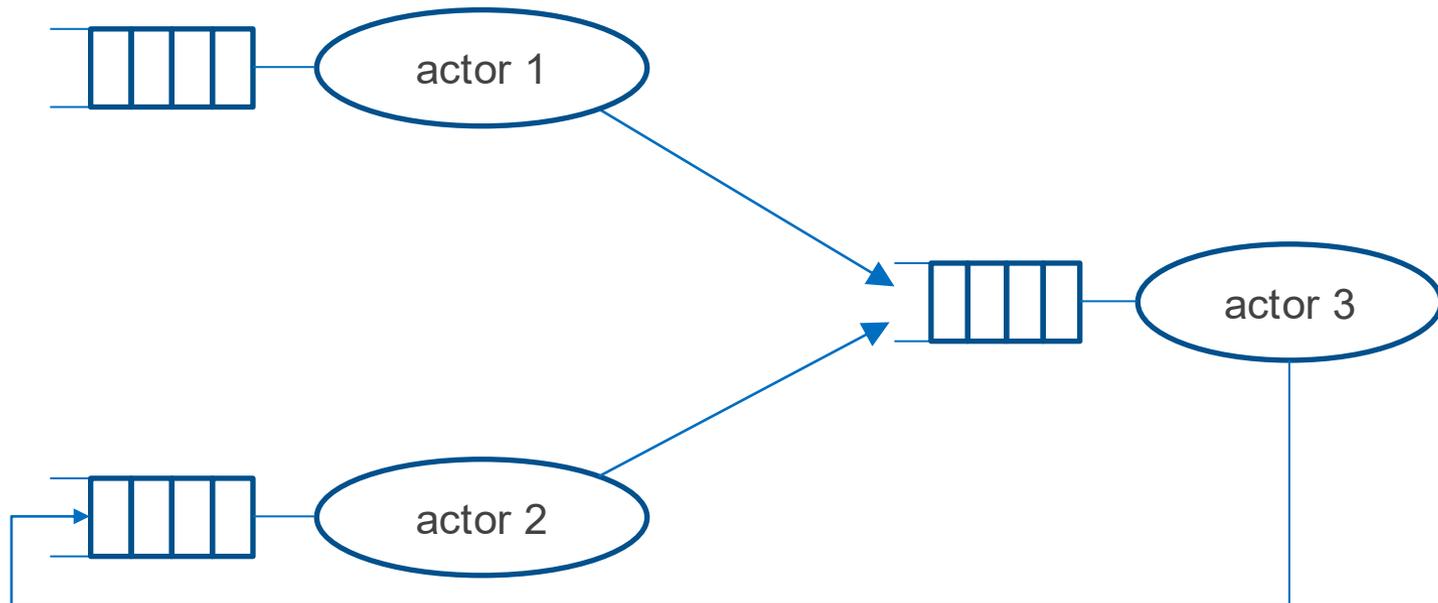
- An *actor* is a type of process
- Each actor has an incoming *message queue*
- **No other shared state**
- Actors communicate by “message passing”
  - placing messages on message queues
- Supports modular concurrent programs
- *Actors and message queues are abstractions*

# Mutual Exclusion with Actors

- Data structure owned by a “server actor”
- Client actors can send request messages to the server and receive response messages if necessary
- Server actor awaits requests on its queue and executes one request at a time



- Mutual Exclusion (one request at a time)
- Progress (requests eventually get to the head of the queue)
- Fairness (requests are handled in FCFS order)

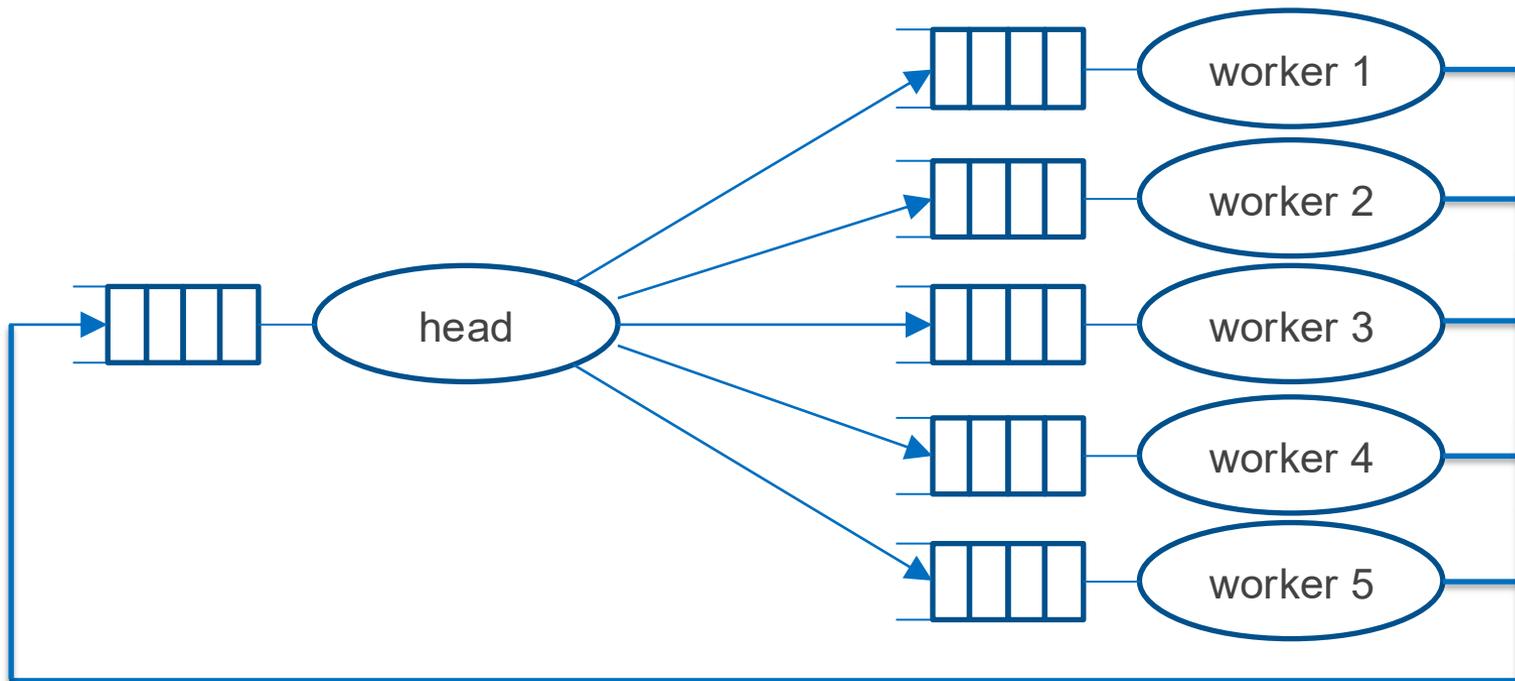


# Conditional Critical Sections with Actors

- An actor can “wait” for a condition by waiting for a specific message
- An actor can “notify” another actor by sending it a message

# Parallel processing with Actors

- Organize program with a Manager Actor and a collection of Worker Actors
- Manager Actor sends work requests to the Worker Actors
- Worker Actors send completion requests to the Manager Actor



# Parallel processing example

```
1 from synch import *
2
3 ranges = { (2,10), (11,20), (21,30) }
4 queues = { r:Queue() for r in ranges }
5 mainq = Queue()
6
7 def isPrime(v) returns prime:
8     prime = True
9     var d = 2
10    while prime and (d < v):
11        if (v % d) == 0:
12            prime = False
13            d += 1
14
15 def worker(q):
16     while True:
17         let rq, (start, finish) = get(q):
18             for p in { start .. finish }:
19                 if isPrime(p):
20                     put(rq, p)
21
22 def main(rq, workers):
23     for r:q in workers:
24         put(q, (rq, r))
25     while True:
26         print get(rq)
27
28 for r in ranges:
29     spawn eternal worker(?queues[r])
30 spawn eternal main(?mainq, { r:?queues[r] for r in ranges })
```

# Pipeline Parallelism with Actors

- Organize program as a chain of actors
- For example, REST/HTTP server
  - Network receive actor → HTTP parser actor → REST request actor → Application actor → REST response actor → HTTP response actor → Network send actor



automatic flow control (when actors run at different rates)

- with bounded buffer queues

# Pipelining Example

```
1  from synch import *
2
3  const MAX = 10
4
5  def isPrime(v) returns prime:
6      prime = True
7      var d = 2
8      while prime and (d < v):
9          if (v % d) == 0:
10             prime = False
11             d += 1
12
13  q1 = q2 = q3 = Queue()
14
15  def actor0():
16      for v in {2..MAX}:
17          put(?q1, v)
```

```
19  def actor1():
20      while True:
21          let v = get(?q1):
22              put(?q2, (2 ** v) - 1)
23
24  def actor2():
25      while True:
26          let v = get(?q2):
27              if isPrime(v):
28                  put(?q3, v)
29
30  def actor3():
31      while True:
32          let v = get(?q3):
33              print(v)
34
35  spawn actor0()
36  spawn eternal actor1()
37  spawn eternal actor2()
38  spawn eternal actor3()
```

Find Mersenne primes

# Support for actors in programming languages

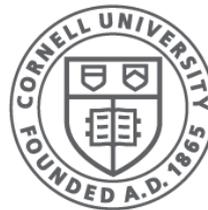
- Native support in languages such as Scala and Erlang
- "blocking queues" in Python, Harmony, Java
- Actor support libraries for Java, C, ...

Actors also nicely generalize to distributed systems!

# Actor disadvantages?

- Doesn't work well for “fine-grained” synchronization
  - overhead of message passing much higher than lock/unlock
- Sending/receiving messages just to access a data structure leads to significant extra code

# Barrier Synchronization



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Barrier Synchronization: the opposite of mutual exclusion...

- Set of processes run in **rounds**
- Must all complete a round before starting the next
- Popular in simulation, HPC, graph processing, model checking...
  - Lock-based synchronization reduces opportunities for parallelism
  - Barrier Synchronization supports scalable parallelism

# Barrier abstraction

- **Barrier(N)**: barrier for N threads
- **bwait()**: start the next round



# Example: dot product

```
1  import barrier
2
3  const NWORKERS = 2
4
5  vec1 = [ 1, 2, 3, 4 ]
6  vec2 = [ 5, 6, 7, 8 ]
7  barr = barrier.Barrier(NWORKERS)
8  output = [ 0, ] * NWORKERS
9
10 def split(self, v) returns x:
11     x = (self * len(v)) / NWORKERS
12
13 def dotproduct(self, v1, v2):
14     assert len(v1) == len(v2)
15     var total = 0
16     for i in { split(self, v1) .. split(self + 1, v1) - 1 }:
17         total += v1[i] * v2[i]
18     output[self] = total
19     barrier.bwait(?barr)
20     print sum(output)
21
22 for i in { 0 .. NWORKERS - 1 }:
23     spawn dotproduct(i, vec1, vec2)
```

# Test program for barriers

```
1 import barrier
2
3 const NTHREADS = 3
4 const NROUNDS = 4
5
6 barr = barrier.Barrier(NTHREADS)
7 before = after = [0,] * NTHREADS
8
9 invariant min(before) >= max(after)
10
11 def thread(self):
12     for _ in { 1 .. NROUNDS }:
13         before[self] += 1
14         barrier.bwait(?barr)
15         after[self] += 1
16
17 for i in { 0 .. NTHREADS - 1 }:
18     spawn thread(i)
```



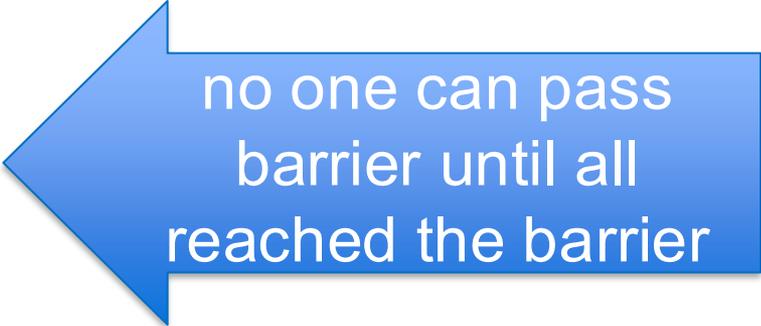
work done before barrier



work done after barrier

# Test program for barriers

```
1  import barrier
2
3  const NTHREADS = 3
4  const NROUNDS = 4
5
6  barr = barrier.Barrier(NTHREADS)
7  before = after = [0,] * NTHREADS
8
9  invariant min(before) >= max(after)
10
11 def thread(self):
12     for _ in { 1 .. NROUNDS }:
13         before[self] += 1
14         barrier.bwait(?barr)
15         after[self] += 1
16
17 for i in { 0 .. NTHREADS - 1 }:
18     spawn thread(i)
```



no one can pass  
barrier until all  
reached the barrier



work done before barrier



work done after barrier

# Barrier Specification, Attempt 1

```
1 def Barrier(required) returns barrier:  
2   barrier = { .required: required, .n: 0 }  
3  
4 def bwait(b):  
5   atomically b->n += 1  
6   atomically await b->n == b->required
```

## State:

- *required*: #threads
- *n*: #threads that have reached the barrier

# Barrier Specification, Attempt 1

```
1 def Barrier(required) returns barrier:  
2   barrier = { .required: required, .n: 0 }  
3  
4 def bwait(b):  
5   atomically b->n += 1  
6   atomically await b->n == b->required
```



## State:

- *required*: #threads
- *n*: #threads that have reached the barrier



# Barrier Specification, Attempt 1

```
1 def Barrier(required) returns barrier:  
2   barrier = { .required: required, .n: 0 }  
3  
4 def bwait(b):  
5   atomically b->n += 1  
6   atomically await b->n == b->required
```

turnstile

State:

- *required*: #threads
- *n*: #threads that have reached the barrier

waiting area



# Barrier Specification, Attempt 1

```
1 def Barrier(required) returns barrier:  
2   barrier = { .required: required, .n: 0 }  
3  
4 def bwait(b):  
5   atomically b->n += 1  
6   atomically await b->n == b->required
```

## State:

- *required*: #threads
- *n*: #threads that have reached the barrier

Only works one round

# Barrier Specification, Attempt 2

```
1 def Barrier(required) returns barrier:
2   barrier = { .required: required, .n: 0 }
3
4 def bwait(b):
5   atomically:
6     b->n += 1
7     if b->n == b->required:
8       b->n = 0
9   atomically await b->n == 0
```

# Barrier Specification, Attempt 2

```
1 def Barrier(required) returns barrier:
2   barrier = { .required: required, .n: 0 }
3
4 def bwait(b):
5   atomically:
6     b->n += 1
7     if b->n == b->required:
8       b->n = 0
9   atomically await b->n == 0
```

**Broken!**  
(if used more than once)

# Barrier Specification, Attempt 3

```
1 def Barrier(required) returns barrier:
2   barrier = { .required: required, .n: [0, 0] }
3
4 def turnstile(b, i):
5   atomically:
6     b->n[i] += 1
7     if b->n[i] == b->required:
8       b->n[1 - i] = 0
9   atomically await b->n[i] == b->required
10
11 def bwait(b):
12   turnstile(b, 0)
13   turnstile(b, 1)
```



# Barrier Specification, Attempt 3

```
1 def Barrier(required) returns barrier:  
2   barrier = { .required: required, .n: [0, 0] }  
3  
4 def turnstile(b, i):  
5   atomically:  
6     b->n[i] += 1  
7     if b->n[i] == b->required:  
8       b->n[1 - i] = 0  
9   atomically await b->n[i] == b->required  
10  
11 def bwait(b):  
12   turnstile(b, 0)  
13   turnstile(b, 1)
```



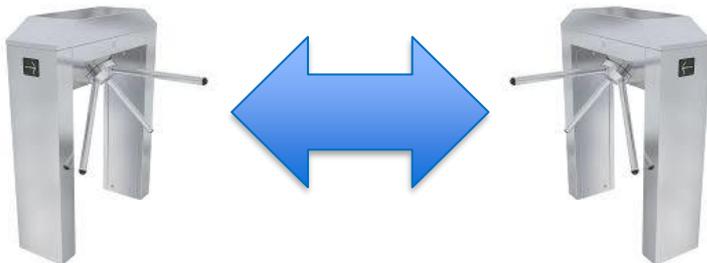
**Works, but double  
waiting is inefficient**

# Barrier Specification, final version

```
1 def Barrier(required) returns barrier:  
2   barrier = { .required: required, .n: 0, .color: 0 }  
3  
4 def bwait(b):  
5   var color = None  
6   atomically:  
7     color = b->color  
8     b->n += 1  
9     if b->n == b->required:  
10      b->color ^= 1  
11      b->n = 0  
12   atomically await b->color != color
```

## State:

- *required*: #threads
- *n*: #threads that have reached the barrier
- *color*: allows re-use of barrier. Flipped each round



# Barrier Implementation

```
1  from synch import *
2
3  def Barrier(required) returns barrier:
4      barrier = {
5          .mutex: Lock(), .cond: Condition(),
6          .required: required, .n: 0, .color: 0
7      }
8
9  def bwait(b):
10     acquire(?b->mutex)
11     b->n += 1
12     if b->n == b->required:
13         b->color ^= 1
14         b->n = 0
15         notify_all(?b->cond)
16     else:
17         let color = b->color:
18             while b->color == color:
19                 wait(?b->cond, ?b->mutex)
20     release(?b->mutex)
```

# Advanced Barrier Synchronization

- Given is a **resource** of **finite capacity**
  - Bus with N seats, say
- Resource must be used at full capacity
  - Bus won't go until it is full
- Resource must be completely emptied before it can be re-used
  - Everybody must get off at destination before anybody can get back on the bus

# Advanced Barrier Synchronization

- Given is a **resource** of **finite capacity**
  - Bus with N seats, say
- Resource must be used at full capacity
  - Bus won't go until it is full
- Resource must be **correctly emptied** before it can be used
  - Passengers must get off at destination
  - Before anybody can get back on the bus

**Typical Exam Question!**

# Interface

- *enter(resource)*
  - must wait if resource is in use or if resource has not yet been fully unloaded
  - after that, must wait until resource is full
- *exit(resource)*
  - any time

# Rounds and Phases

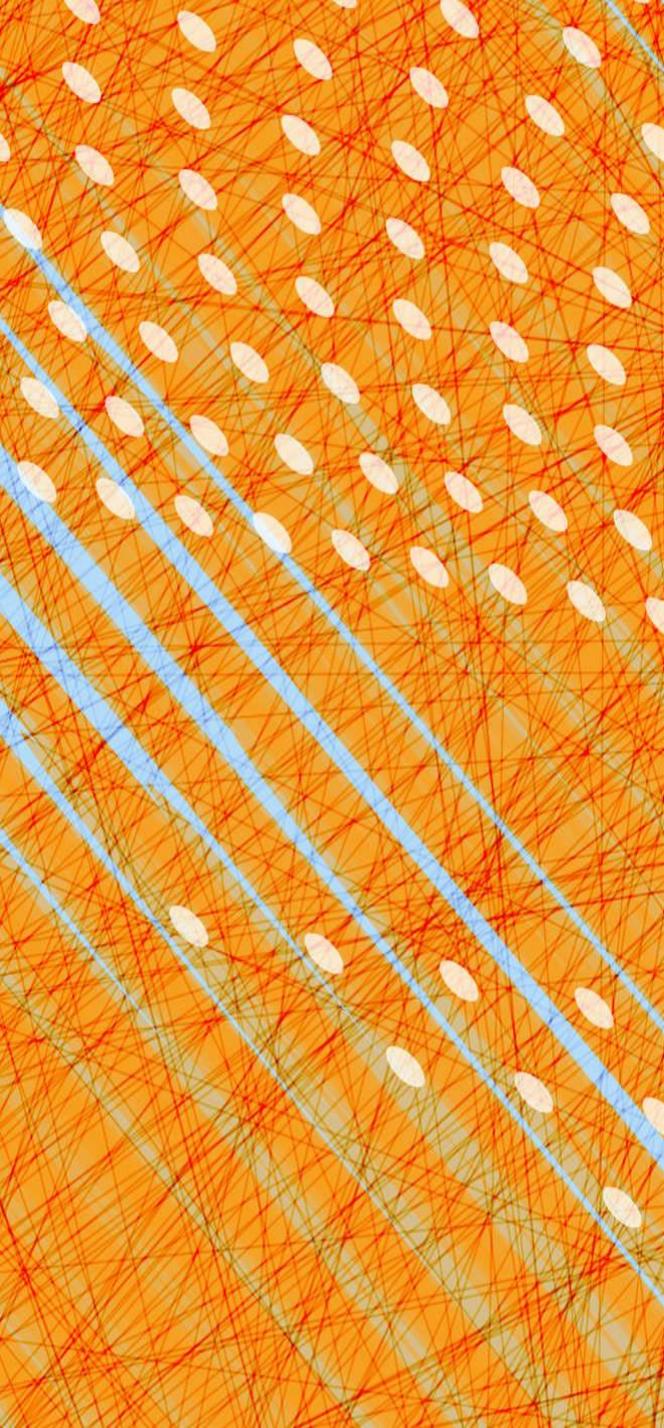
- Round: each time the resource gets used
- Three phases in each round:
  1. Resource is loaded
  2. Resource is used
  3. Resource is unloaded
- Two waiting conditions:
  - Wait until resource is fully unloaded
    - Before starting to load the resource
  - Wait until resource is fully loaded
    - Before starting to use the resource

# Rollercoaster

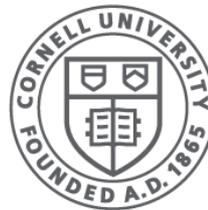


JOE MCBRIDE / GETTY IMAGES

```
1 from synch import *
2
3 def RollerCoaster(nseats): result = {
4     .mutex: Lock(), .nseats: nseats, .entered: 0, .left: nseats,
5     .empty: Condition(), .full: Condition()
6 }
7
8 def enter(b):
9     acquire(?b->mutex)
10    while b->entered == b->nseats: # wait for car to empty out
11        wait(?b->empty, ?b->mutex)
12    b->entered += 1
13    if b->entered != b->nseats: # wait for car to fill up
14        while b->entered < b->nseats:
15            wait(?b->full, ?b->mutex)
16    else: # car is ready to go
17        b->left = 0
18        notify_all(?b->full) # wake up others waiting in car
19    release(?b->mutex)
20
21 def exit(b):
22    acquire(?b->mutex)
23    b->left += 1
24    if b->left == b->nseats: # car is empty
25        b->entered = 0
26        notify_all(?b->empty) # wake up riders wanting to go
27    release(?b->mutex)
```



# Interrupt Safety



**Cornell CIS**  
COMPUTING AND INFORMATION SCIENCE

# Interrupt handling

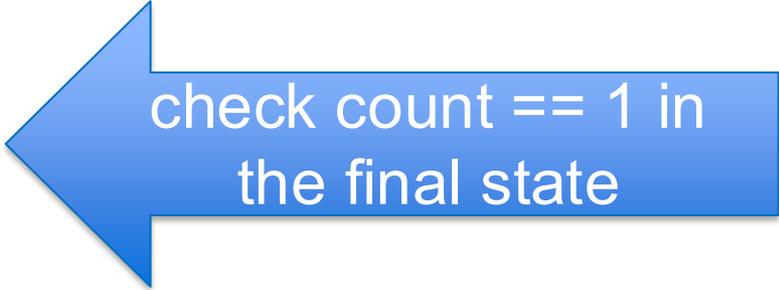
- When executing in user space, a device interrupt is invisible to the user process
  - State of user process is unaffected by the device interrupt and its subsequent handling
  - This is because contexts are switched back and forth
  - So, the user space context is *exactly restored* to the state it was in before the interrupt

# Interrupt handling

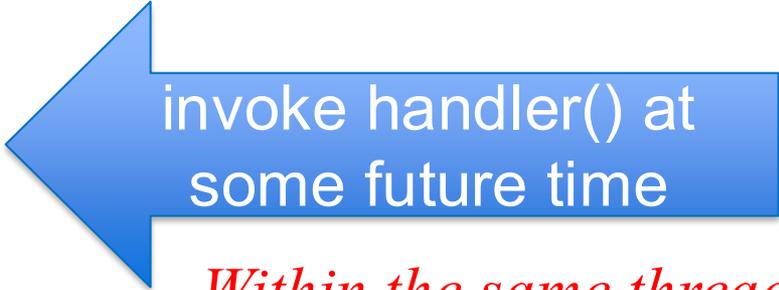
- However, there are also “in-context” interrupts:
    - kernel code can be interrupted
    - user code can handle “signals”
- *Potential for race conditions*

# “Traps” in Harmony

```
1 | count = 0
2 | done = False
3 |
4 | finally count == 1
5 |
6 | def handler():
7 |     count += 1
8 |     done = True
9 |
10 | def main():
11 |     trap handler()
12 |     await done
13 |
14 | spawn main()
```



check count == 1 in  
the final state



invoke handler() at  
some future time

*Within the same thread!  
(trap ≠ spawn)*

# But what now?

```
1 | count = 0
2 | done = False
3 |
4 | finally count == 2
5 |
6 | def handler():
7 |     count += 1
8 |     done = True
9 |
10 | def main():
11 |     trap handler()
12 |     count += 1
13 |     await done
14 |
15 | spawn main()
```

# But what now?

```
1  count = 0
2  done = False
3
4  finally count == 2
5
6  def handler():
7      count += 1
8      done = True
9
10 def main():
11     trap handler()
12     count += 1
13     await done
14
15 spawn main()
```

## Summary: something went wrong in an execution

- Schedule thread To: **init()**
  - Line 1: Initialize count to 0
  - Line 2: Initialize done to False
  - **Thread terminated**
- Schedule thread T1: main()
  - Line 12: Interrupted: jump to interrupt handler first
  - Line 12: Interrupts disabled
  - Line 7: Set count to 1 (was 0)
  - Line 8: Set done to True (was False)
  - Line 6: Interrupts enabled
  - Line 12: Set count to 1 (unchanged)
  - **Thread terminated**
- Schedule thread T2: finally()
  - Line 4: Harmony assertion failed

# Locks to the rescue?

```
1  from synch import Lock, acquire, release
2
3  countlock = Lock()
4  count = 0
5  done = False
6
7  finally count == 2
8
9  def handler():
10     acquire(?countlock)
11     count += 1
12     release(?countlock)
13     done = True
14
15  def main():
16     trap handler()
17     acquire(?countlock)
18     count += 1
19     release(?countlock)
20     await done
21
22  spawn main()
```

# Locks to the rescue?

```
1 from synch import Lock, acquire, release
2
3 countlock = Lock()
4 count = 0
5 done = False
6
7 finally count == 2
8
9 def handler():
10     acquire(?countlock)
11     count += 1
12     release(?countlock)
13     done = True
14
15 def main():
16     trap handler()
17     acquire(?countlock)
18     count += 1
19     release(?countlock)
20     await done
21
22 spawn main()
```

## Summary: some execution cannot terminate

- Schedule thread T0: `init()`
  - Line 3: Initialize `countlock` to `False`
  - Line 4: Initialize `count` to `0`
  - Line 5: Initialize `done` to `False`
- Schedule thread T1: `main()`
  - Line `synch/36`: Set `countlock` to `True` (was `False`)
  - Line 18: Set `count` to `1` (was `0`)
  - Line `synch/39`: Interrupted: jump to interrupt handler first
  - Line `synch/39`: Interrupts disabled
  - Preempted in `main()` --> `release(?countlock)` --> `handler()` --> `acquire(?countlock)` about to execute atomic section in line `synch/35`

## Final state (all threads have terminated or are blocked):

- Threads:
  - T1: (blocked interrupts-disabled) `main()` --> `release(?countlock)` --> `handler()` --> `acquire(?countlock)`
    - about to execute atomic section in line `synch/35`

# Enabling/disabling interrupts

```
1 count = 0
2 done = False
3
4 finally count == 2
5
6 def handler():
7     count += 1
8     done = True
9
10 def main():
11     trap handler()
12     setintlevel(True)
13     count += 1
14     setintlevel(False)
15     await done
16
17 spawn main()
```

← disable interrupts

← enable interrupts

# Interrupt-Safe Methods

```
1  count = 0
2  done = False
3
4  finally count == 2
5
6  def increment():
7      let prior = setintlevel(True):
8          count += 1
9          setintlevel(prior)
10
11 def handler():
12     increment()
13     done = True
14
15 def main():
16     trap handler()
17     increment()
18     await done
19
20 spawn main()
```



disable interrupts



restore old interrupt level

# Interrupt-safe *AND* Thread-safe?

```
1  from synch import Lock, acquire, release
2
3  count = 0
4  countlock = Lock()
5  done = [ False, False ]
6
7  finally count == 4
8
9  def increment():
10     let prior = setintlevel(True):
11         acquire(?countlock)
12         count += 1
13         release(?countlock)
14         setintlevel(prior)
15
16  def handler(self):
17     increment()
18     done[self] = True
19
20  def thread(self):
21     trap handler(self)
22     increment()
23     await done[self]
24
25  spawn thread(0)
26  spawn thread(1)
```

# Interrupt-safe AND Thread-safe?

```
1 from synch import Lock, acquire, release
2
3 count = 0
4 countlock = Lock()
5 done = [ False, False ]
6
7 finally count == 4
8
9 def increment():
10     let prior = setintlevel(True):
11         acquire(?countlock)
12         count += 1
13         release(?countlock)
14         setintlevel(prior)
15
16 def handler(self):
17     increment()
18     done[self] = True
19
20 def thread(self):
21     trap handler(self)
22     increment()
23     await done[self]
24
25 spawn thread(0)
26 spawn thread(1)
```



*wait for own interrupt*

# Interrupt-safe AND Thread-safe?

```
1 from synch import Lock, acquire, release
2
3 count = 0
4 countlock = Lock()
5 done = [ False, False ]
6
7 finally count == 4
8
9 def increment():
10     let prior = setintlevel(True):
11         acquire(?countlock)
12         count += 1
13         release(?countlock)
14         setintlevel(prior)
15
16 def handler(self):
17     increment()
18     done[self] = True
19
20 def thread(self):
21     trap handler(self)
22     increment()
23     await done[self]
24
25 spawn thread(0)
26 spawn thread(1)
```

*first disable interrupts*

*wait for own interrupt*

# Interrupt-safe AND Thread-safe?

```
1 from synch import Lock, acquire, release
2
3 count = 0
4 countlock = Lock()
5 done = [ False, False ]
6
7 finally count == 4
8
9 def increment():
10     let prior = setintlevel(True):
11         acquire(?countlock)
12         count += 1
13         release(?countlock)
14         setintlevel(prior)
15
16 def handler(self):
17     increment()
18     done[self] = True
19
20 def thread(self):
21     trap handler(self)
22     increment()
23     await done[self]
24
25 spawn thread(0)
26 spawn thread(1)
```

*first disable interrupts*

*then acquire a lock*

*wait for own interrupt*

# Interrupt-safe AND Thread-safe?

```
1 from synch import Lock, acquire, release
2
3 count = 0
4 countlock = Lock()
5 done = [ False, False ]
6
7 finally count == 4
8
9 def increment():
10     let prior = setintlevel(True):
11         acquire(?countlock)
12         count += 1
13         release(?countlock)
14         setintlevel(prior)
15
16 def handler(self):
17     increment()
18     done[self] = True
19
20 def thread(self):
21     trap handler(self)
22     increment()
23     await done[self]
24
25 spawn thread(0)
26 spawn thread(1)
```

why 4?

first disable interrupts

then acquire a lock

wait for own interrupt

# Warning: very few C functions are interrupt-safe

- pure system calls are interrupt-safe
  - e.g. `read()`, `write()`, etc.
- functions that do not use global data are interrupt-safe
  - e.g. `strlen()`, `strcpy()`, etc.
- `malloc()` and `free()` are *not* interrupt-safe
- `printf()` is *not* interrupt-safe
- *However, all these functions are thread-safe*