Concurrent Programming with Harmony

Robbert van Renesse

Cornell University Founded A.D. 1865
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

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*

Harmony

- A new concurrent programming language
  - heavily based on Python syntax to reduce learning curve for many
- A new underlying virtual machine
  - it tries *all* possible executions of a program until it finds a problem, if any

(this is called "model checking")

The problem with non-determinism

sequential

```
shared = True

def f(): assert shared
def g(): shared = False

f()
g()
```

concurrent

```
shared = True

def f(): assert shared
def g(): shared = False

spawn f()
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

```
shared = 0

def f(): shared += 1

f()
f()

finally shared == 2
```

concurrent

```
shared = 0

def f(): shared += 1

spawn f()
spawn f()

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 ≠ no race condition
- Timing dependent: small changes hide bugs
  - add print statement → bug no longer seems to happen
- Harmony is designed to help identify such bugs
  - model checking!

State Space and Model Checking

Cornell University Founded A.D. 1865

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*)

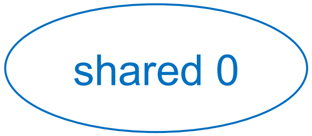
HVM state represents one vertex in a graph of states

State space

thread 1 loads · · · thread 1 stores

thread 2 loads · · · thread 2 stores

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

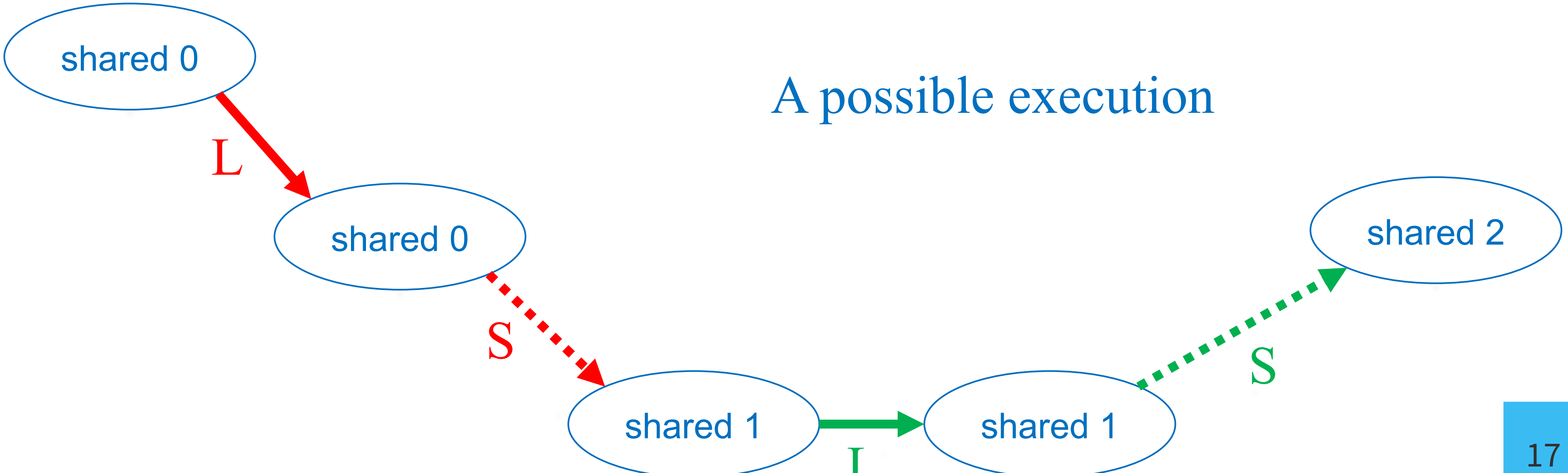


initial state

State space

- thread 1 loads
- thread 1 stores
- thread 2 loads
- thread 2 stores

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



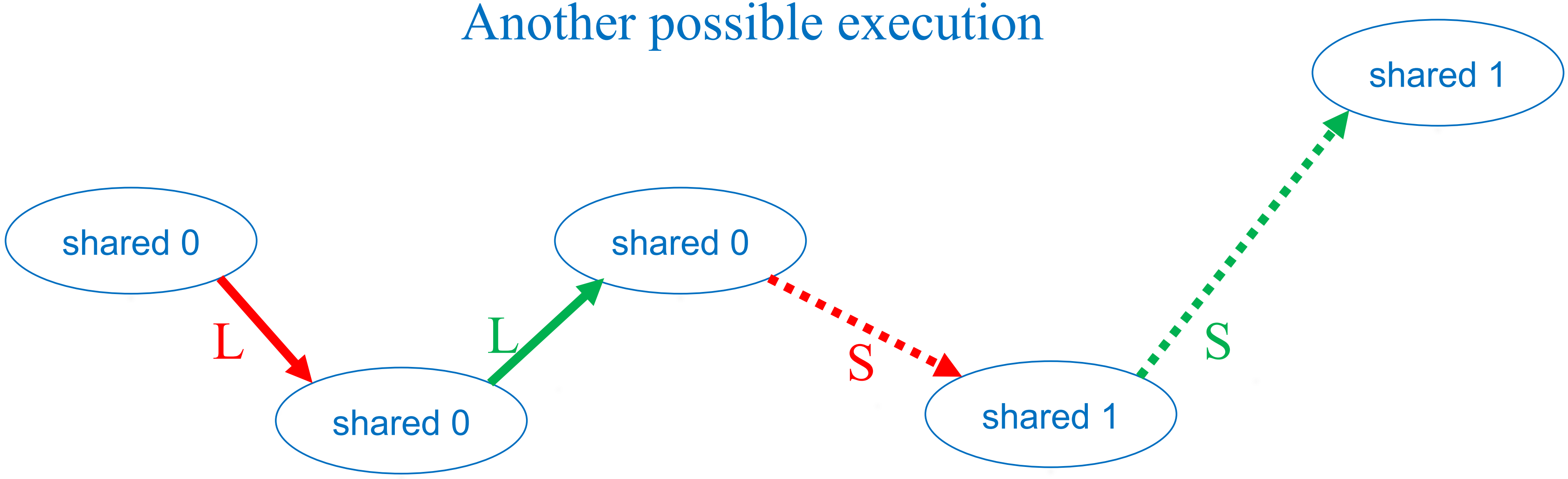
A possible execution

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

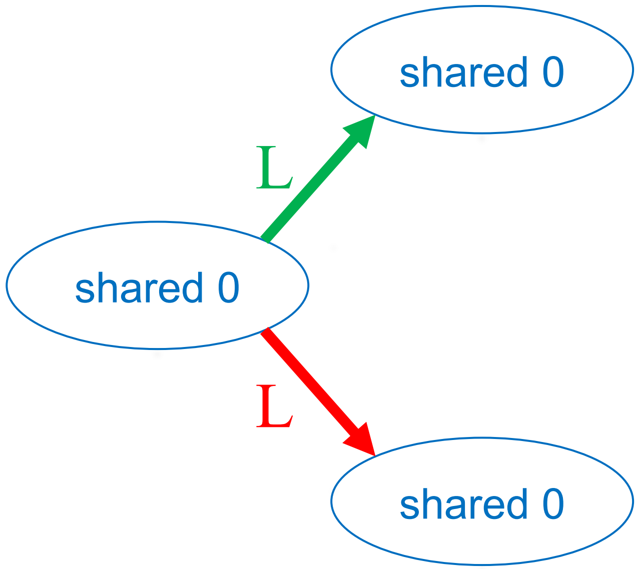


State space

thread 1 loads ⋯▶ thread 1 stores

thread 2 loads ⋯▶ thread 2 stores

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



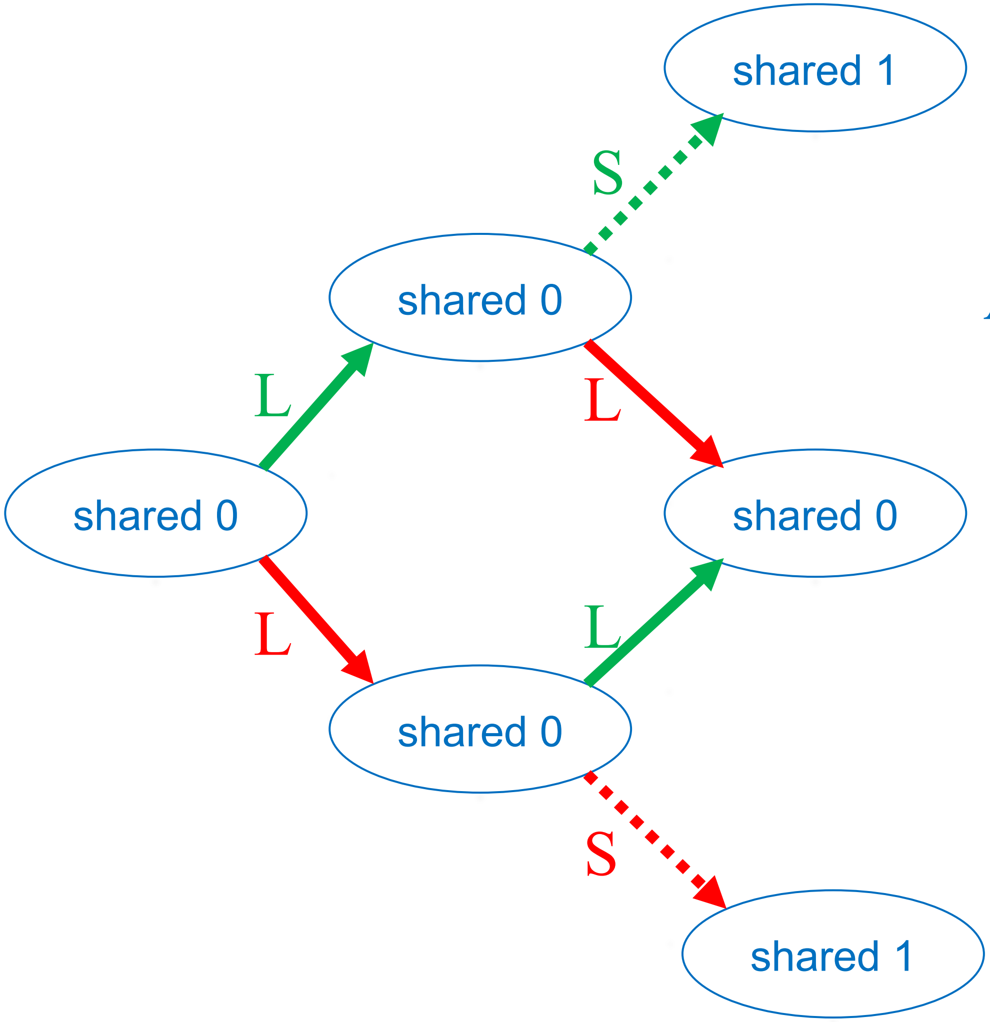
All possible states after one "step"

State space

thread 1 loads · · · thread 1 stores

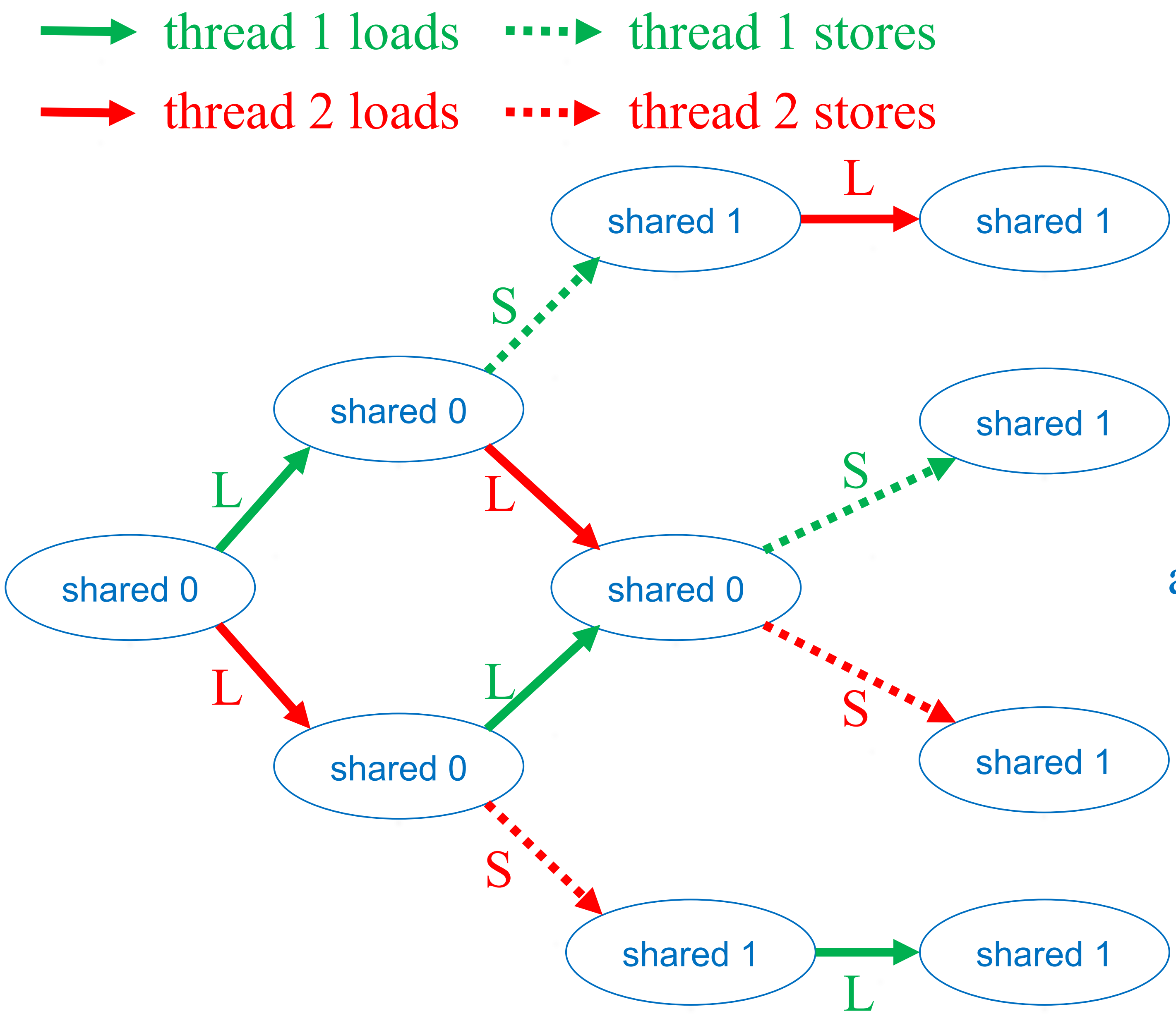
thread 2 loads · · · thread 2 stores

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



All possible states after two steps

State space

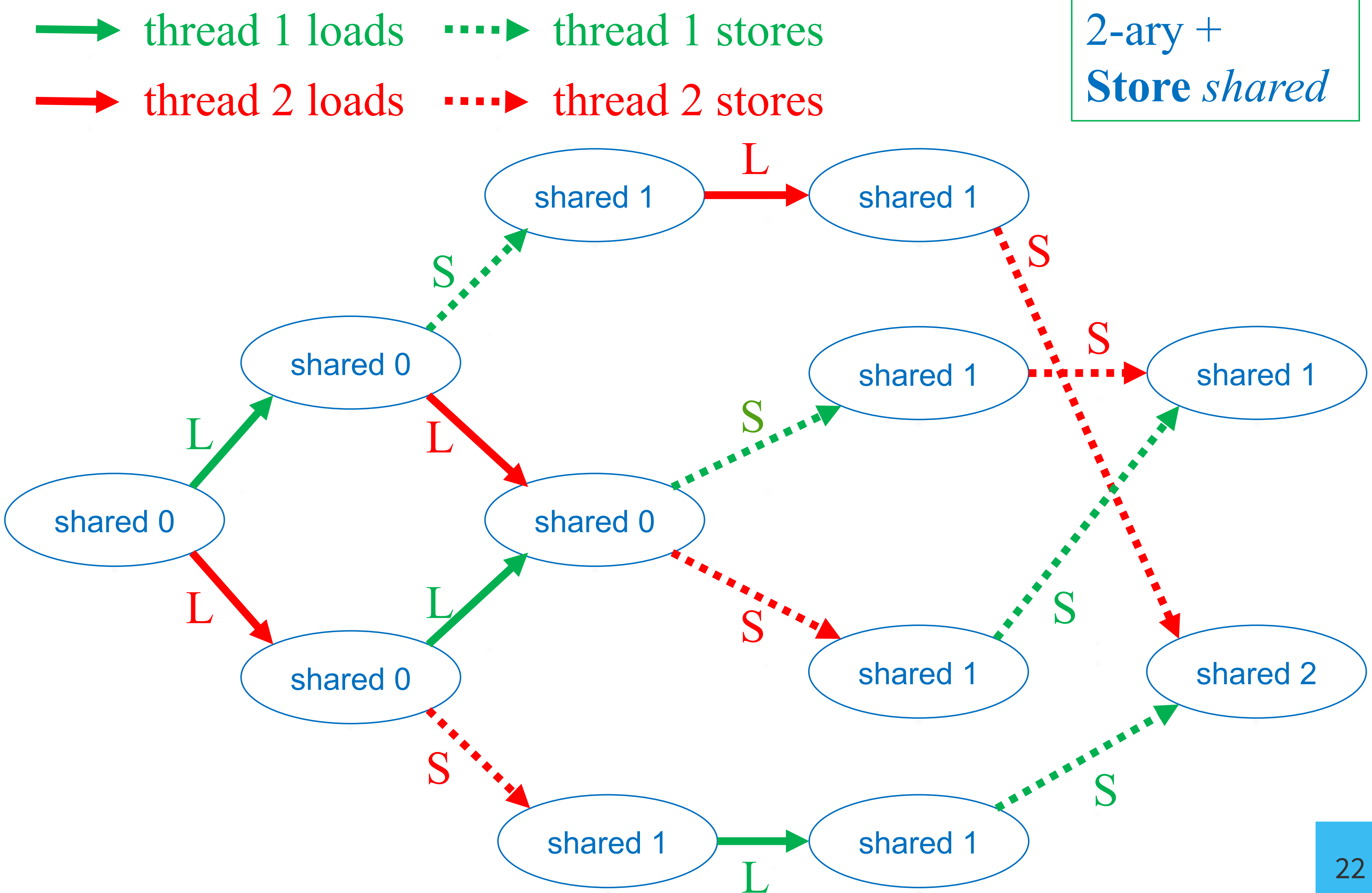


Load *shared*
Push 1
2-ary +
Store *shared*

after three steps

State space

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



Harmony

Harmony != Python

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

I/O in Harmony?

- Input:
  - **choose** expression
    - x = **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 Harm… …nputs
  - **cons**…
    - **c**…
    - ca… …en with “-c x=4” flag to harmony
  - Outp…t:
    - **print** x + y
    - **assert** x + y < 10, (x, y)

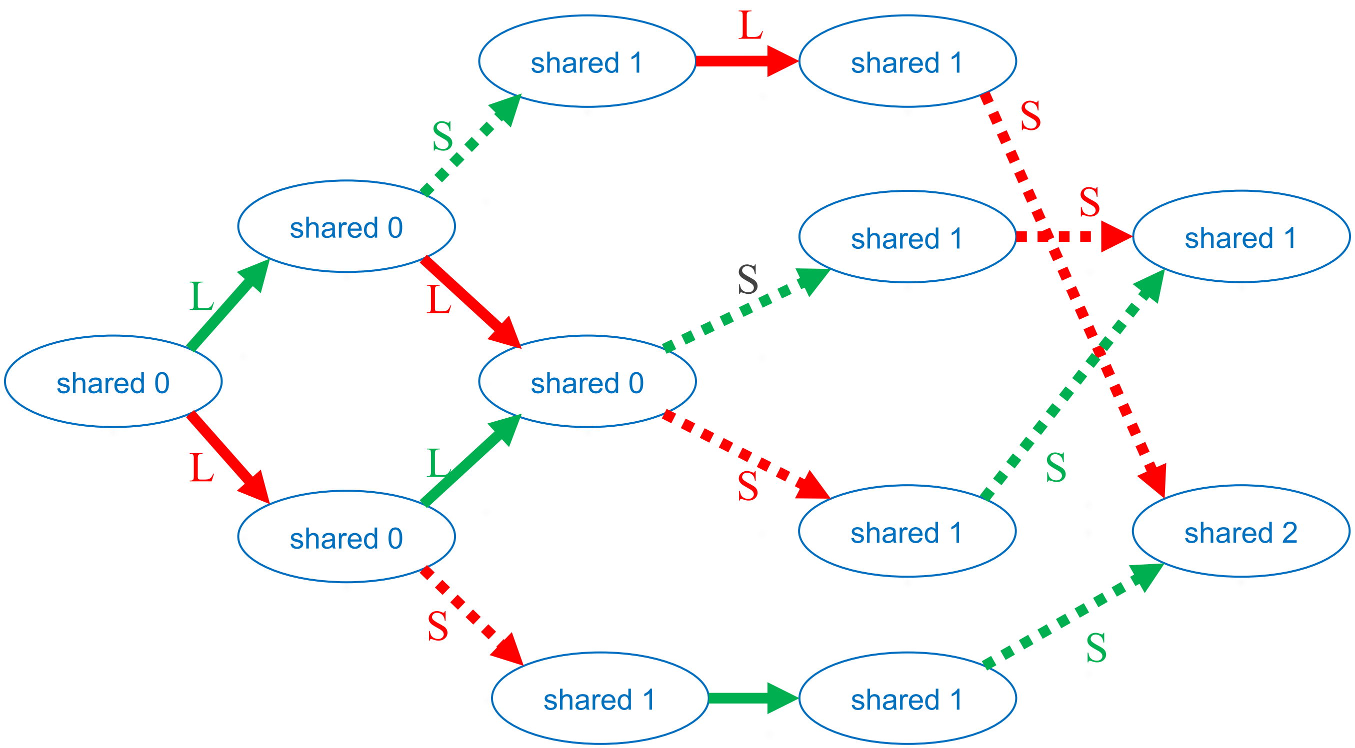
No open(), read(), or
or input() statements

Non-determinism in Harmony

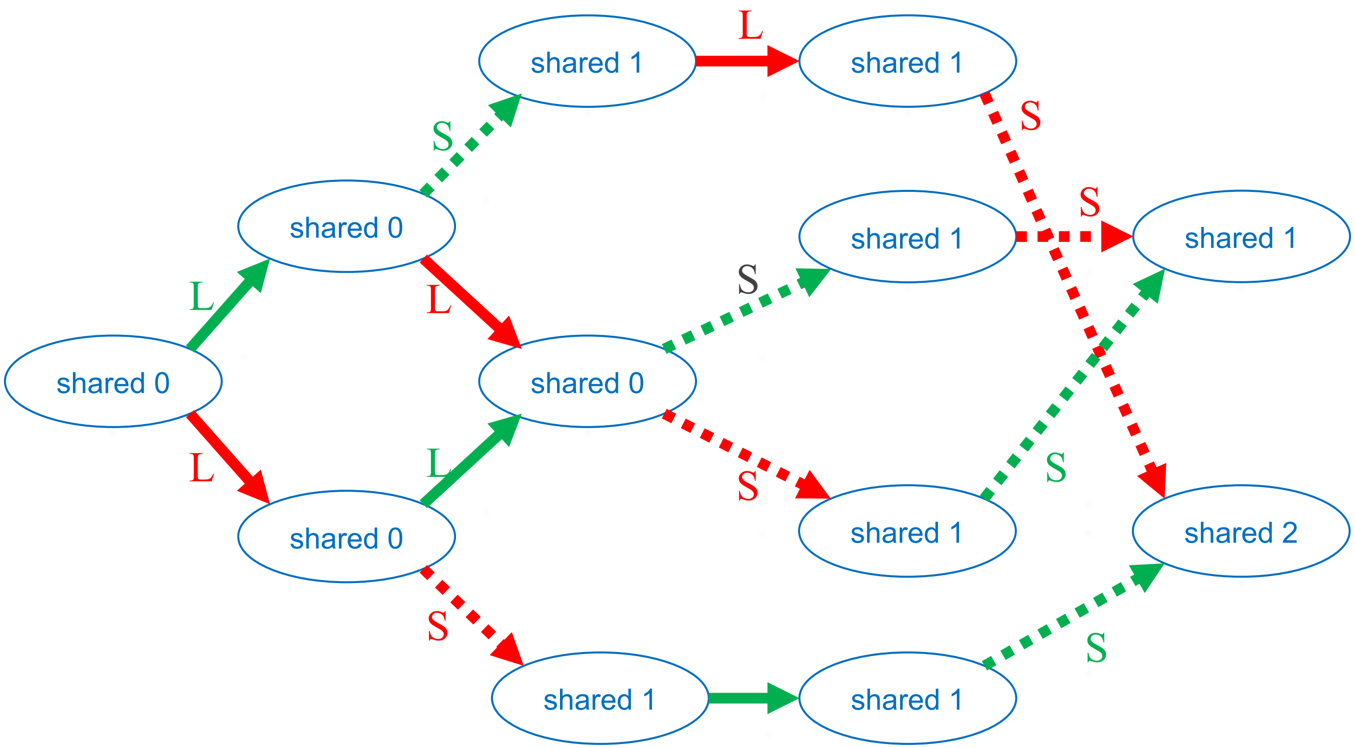
Three sources:

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

Limitation: models must be finite!



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

```
shared = 0

def f(): shared += 1

spawn f()
spawn f()

finally shared == 2
```

Back to our problem…

2 threads updating a shared variable

```
shared = 0

def f(): shared += 1

spawn f()
spawn f()

finally shared == 2
```

"Critical Section"

Back to our problem…

2 threads updating a shared variable

```
shared = 0

def f(): shared += 1

spawn f()
spawn f()

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

```
def thread():
    while True:
        # Critical section is here
        pass

spawn thread()
spawn thread()
```

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

Specifying Critical Sections in Harmony

```
# number of threads in the critical section
in_cs = 0
invariant in_cs in { 0, 1 }

def thread():
    while choose { False, True }:
        # Enter critical section
        atomically in_cs += 1

        # Critical section is here
        pass

        # Exit critical section
        atomically in_cs -= 1

spawn thread()
spawn thread()
```

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

Specifying Critical Sections in Harmony

```
# number of threads in the critical section
in_cs = 0
invariant in_cs in { 0, 1 }

def thread():
    while choose { False, True }:
        # Enter critical section
        atomically in_cs += 1

        # Critical section is here
        pass

        # Exit critical section
        atomically in_cs -= 1

spawn thread()
spawn thread()
```

mutual exclusion

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

Specifying Critical Sections in Harmony

```
# number of threads in the critical section
in_cs = 0
invariant in_cs in { 0, 1 }

def thread():
    while choose { False, True }:
        # Enter critical section
        atomically in_cs += 1

        # Critical section is here
        pass

        # Exit critical section
        atomically in_cs -= 1

spawn thread()
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

```
# number of threads in the critical section
in_cs = 0
invariant in_cs in { 0, 1 }

def thread():
    while choose { False, True }:
        # Enter critical section
        atomically in_cs += 1

        # Critical section is here
        pass

        # Exit critical section
        atomically in_cs -= 1

spawn thread()
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

```
# number of threads in the critical section
in_cs = 0
invariant in_cs in { 0, 1 }

def thread():
    while choose { False, True }:
        # Enter critical section
        atomically in_cs += 1

        # Critical section is here
        pass

        # Exit critical section
        atomically in_cs -= 1

spawn thread()
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

```
# number of threads in the critical section
in_cs = 0
invariant in_cs in { 0, 1 }

def thread():
    while choose { False, True }:
        # Enter critical section
        atomically in_cs += 1

        # Critical section is here
        pass

        # Exit critical section
        atomically in_cs -= 1

spawn thread()
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

```
# number of threads in the critical section
in_cs = 0
invariant in_cs in { 0, 1 }

def thread():
    while choose { False, True }:
        # Enter critical section
        atomically in_cs += 1

        # Critical section is here
        pass

        # Exit critical section
        atomically in_cs -= 1

spawn thread()
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

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

```
in_cs = 0
invariant in_cs in { 0, 1 }

lockTaken = False

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        await not lockTaken
        lockTaken = True

        atomically in_cs += 1
        # Critical section
        atomically in_cs -= 1

        # Leave critical section
        lockTaken = False

spawn thread(0)
spawn thread(1)
```

First attempt: a naïve lock

```
in_cs = 0
invariant in_cs in { 0, 1 }

lockTaken = False

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        await not lockTaken
        lockTaken = True

        atomically in_cs += 1
        # Critical section
        atomically in_cs -= 1

        # Leave critical section
        lockTaken = False

spawn thread(0)
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*

```
in_cs = 0
invariant in_cs in { 0, 1 }

flags = [ False, False ]

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        await not flags[1 - self]

        atomically in_cs += 1
        # Critical section
        atomically in_cs -= 1

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

Second attempt: *flags*

```
in_cs = 0
invariant in_cs in { 0, 1 }

flags = [ False, False ]

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        await not flags[1 - self]

        atomically in_cs += 1
        # Critical section
        atomically in_cs -= 1

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

show intent to enter critical section

Second attempt: *flags*

```
in_cs = 0
invariant in_cs in { 0, 1 }

flags = [ False, False ]

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        await not flags[1 - self]

        atomically in_cs += 1
        # Critical section
        atomically in_cs -= 1

        # Leave critical section
        flags[self] = False

spawn thread(0)
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

```
in_cs = 0
invariant in_cs in { 0, 1 }

turn = 0

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        turn = 1 - self
        await turn == self

        atomically in_cs += 1
        # Critical section
        atomically in_cs -= 1

        # Leave critical section

spawn thread(0)
spawn thread(1)
```

Third attempt: *turn* variable

```
in_cs = 0
invariant in_cs in { 0, 1 }

turn = 0

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        turn = 1 - self
        await turn == self

        atomically in_cs += 1
        # Critical section
        atomically in_cs -= 1

        # Leave critical section

spawn thread(0)
spawn thread(1)
```

after you...

Third attempt: *turn* variable

```
in_cs = 0
invariant in_cs in { 0, 1 }

turn = 0

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        turn = 1 - self
        await turn == self

        atomically in_cs += 1
        # Critical section
        atomically in_cs -= 1

        # Leave critical section

spawn thread(0)
spawn thread(1)
```

after you...

wait for your turn

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*

```
in_cs = 0
invariant in_cs in { 0, 1 }

sequential flags, turn
flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        atomically in_cs += 1
        # Critical section
        atomically in_cs -= 1

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

Peterson's Algorithm: *flags & turn*

```
in_cs = 0
invariant in_cs in { 0, 1 }

sequential flags, turn
flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        atomically in_cs += 1
        # Critical section
        atomically in_cs -= 1

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

in critical section

Peterson's Algorithm: *flags & turn*

```
in_cs = 0
invariant in_cs in { 0, 1 }

sequential flags, turn
flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        atomically in_cs += 1
        # Critical section
        atomically in_cs -= 1

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

load and store instructions are atomic

in critical section

Peterson’s Algorithm: *flags & turn*

```
in_cs = 0
invariant in_cs in { 0, 1 }

sequential flags, turn
flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        atomically in_cs += 1
        # Critical section
        atomically in_cs -= 1

        # Leave critical section
        flags[self] = False

spawn thread(0)
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*

```
in_cs = 0
invariant in_cs in { 0, 1 }

sequential flags, turn
flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        atomically in_cs += 1
        # Critical section
        atomically in_cs -= 1

        # Leave critical section
        flags[self] = False

spawn thread(0)
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*

```
in_cs = 0
invariant in_cs in { 0, 1 }

sequential flags, turn
flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        atomically in_cs += 1
        # Critical section
        atomically in_cs -= 1

        # Leave critical section
        flags[self] = False

spawn thread(0)
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*

```
in_cs = 0
invariant in_cs in { 0, 1 }

sequential flags, turn
flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        atomically in_cs += 1
        # Critical section
        atomically in_cs -= 1

        # Leave critical section
        flags[self] = False

spawn thread(0)
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)

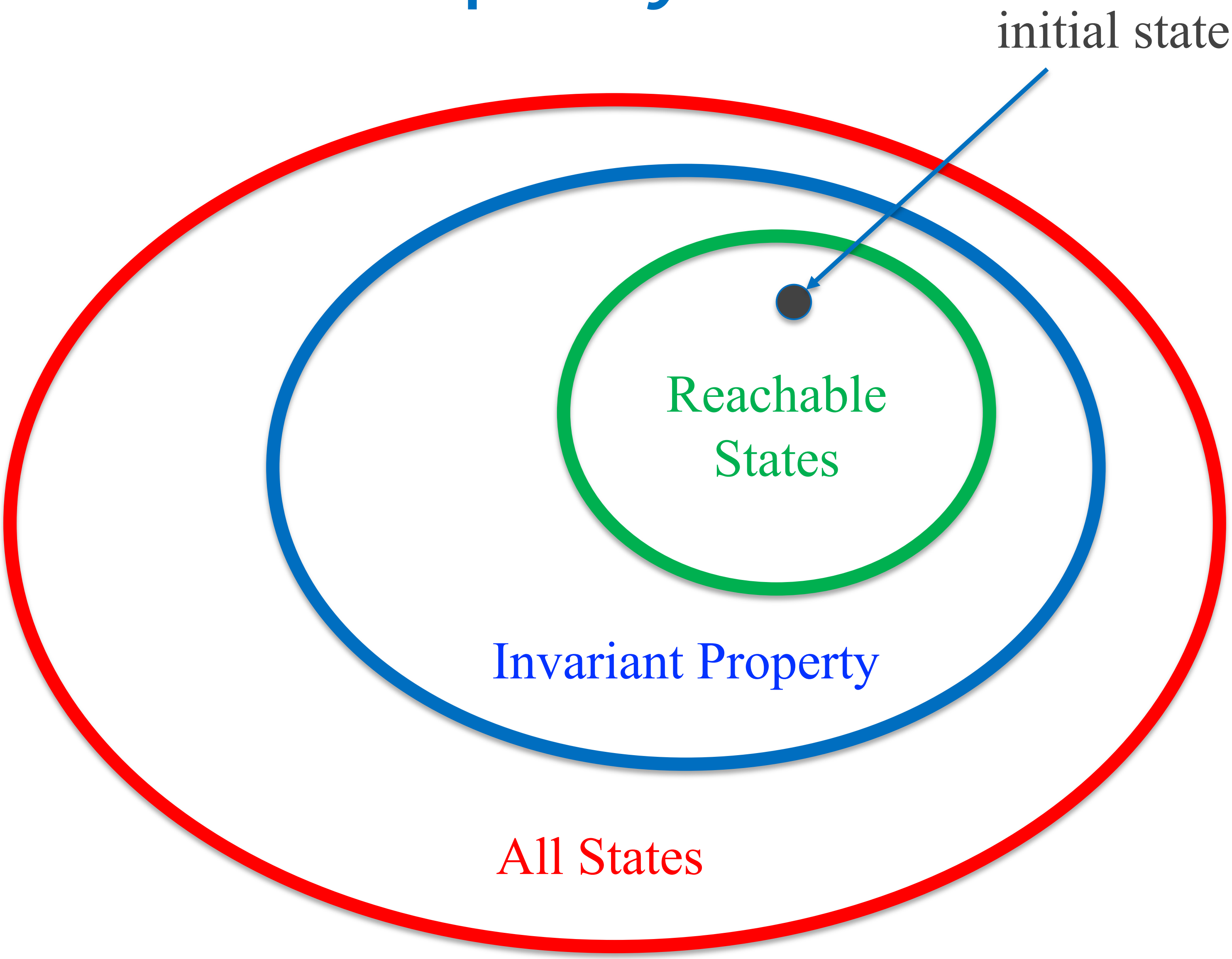
What is a property?

A property is a set of states

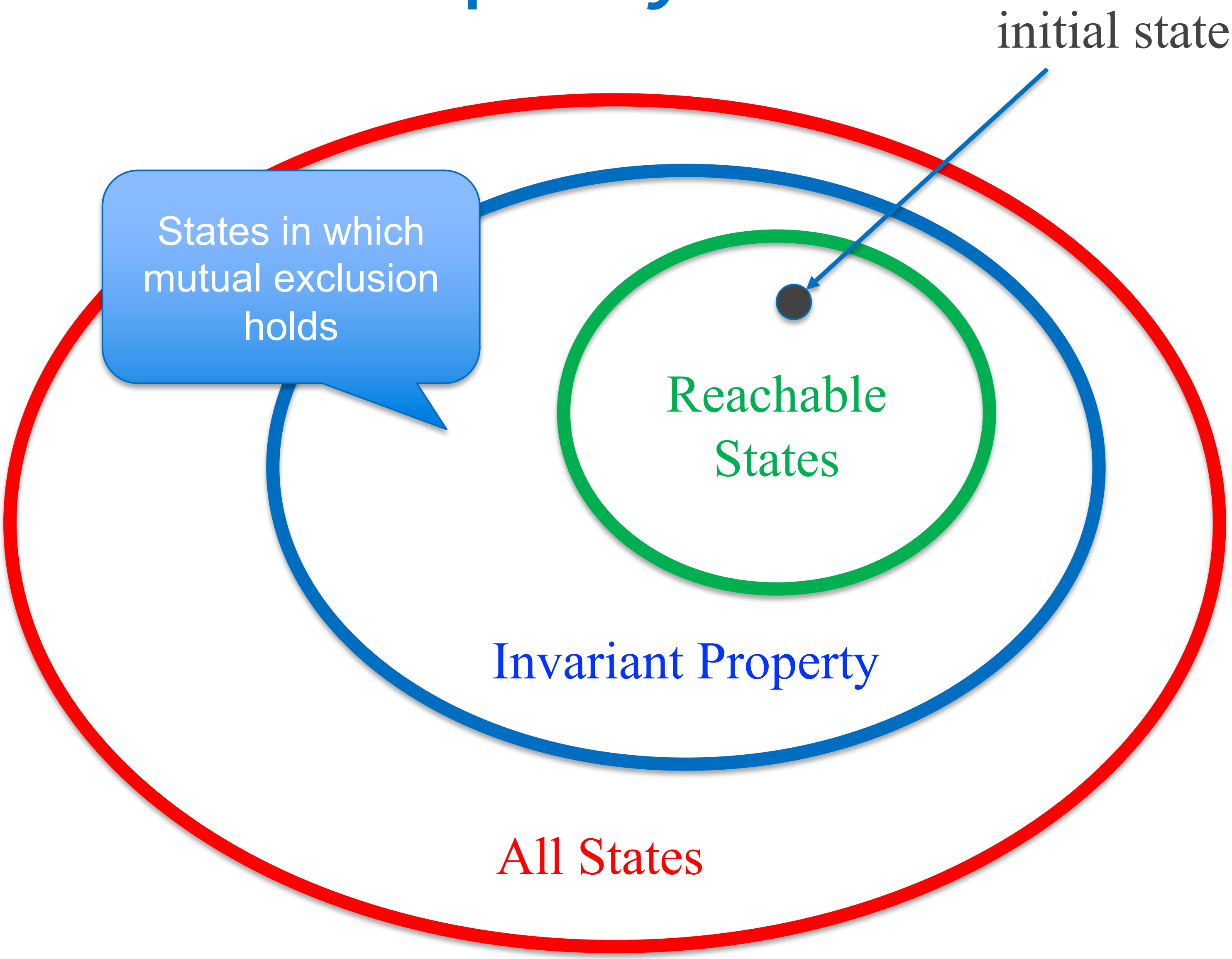
often succinctly described using a predicate

(all states that satisfy the predicate and no others)

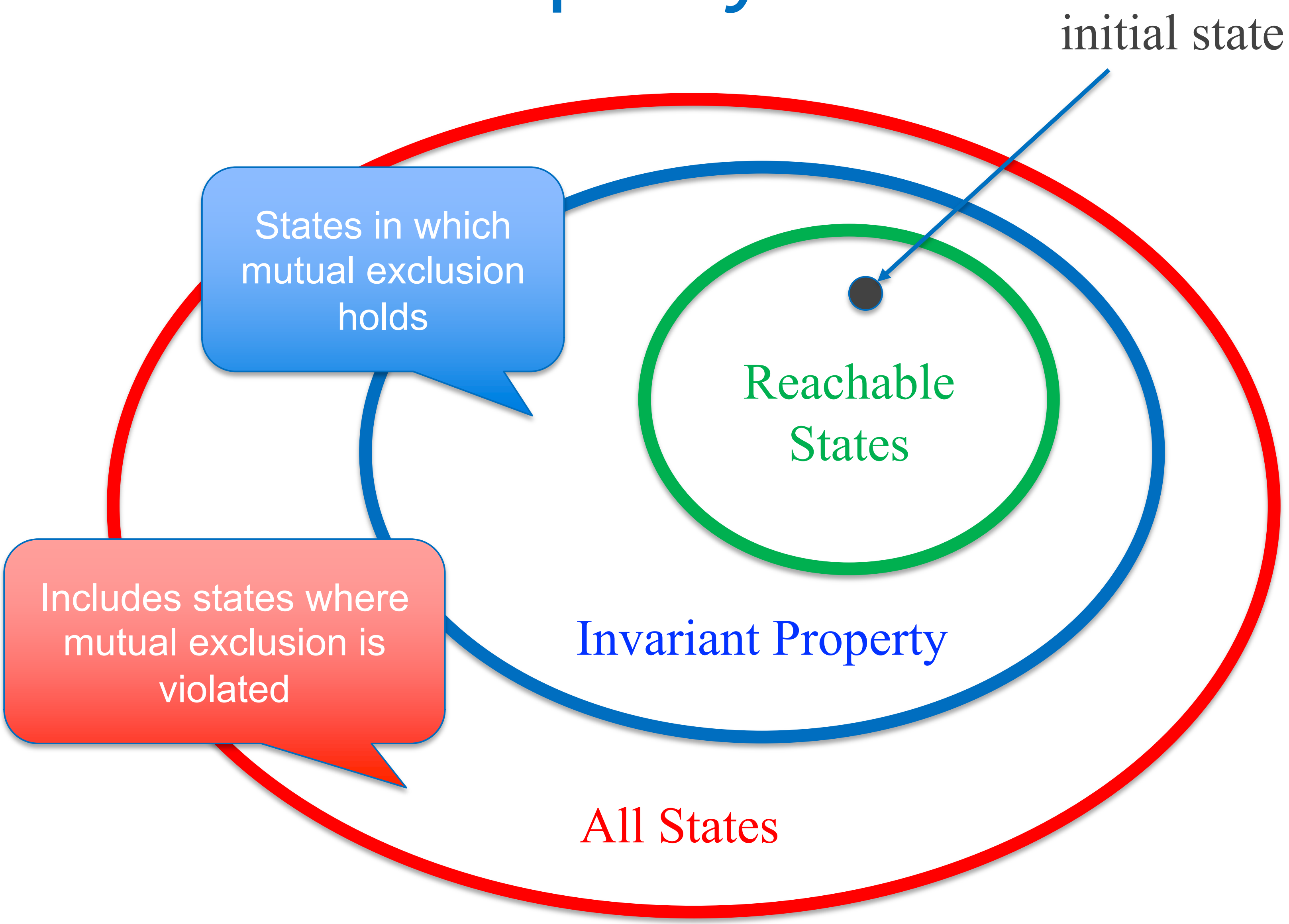
Invariant Property



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

Peterson's Reconsidered

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

```
in_cs = 0
invariant in_cs in { 0, 1 }

sequential flags, turn
flags = [ False, False ]
turn = choose({0, 1})

def thread(self):
    while choose({ False, True }):
        # Enter critical section
        flags[self] = True
        turn = 1 - self
        await (not flags[1 - self]) or (turn == self)

        atomically in_cs += 1
        # Critical section
        atomically in_cs -= 1

        # Leave critical section
        flags[self] = False

spawn thread(0)
spawn thread(1)
```

Loads and Stores are atomic

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

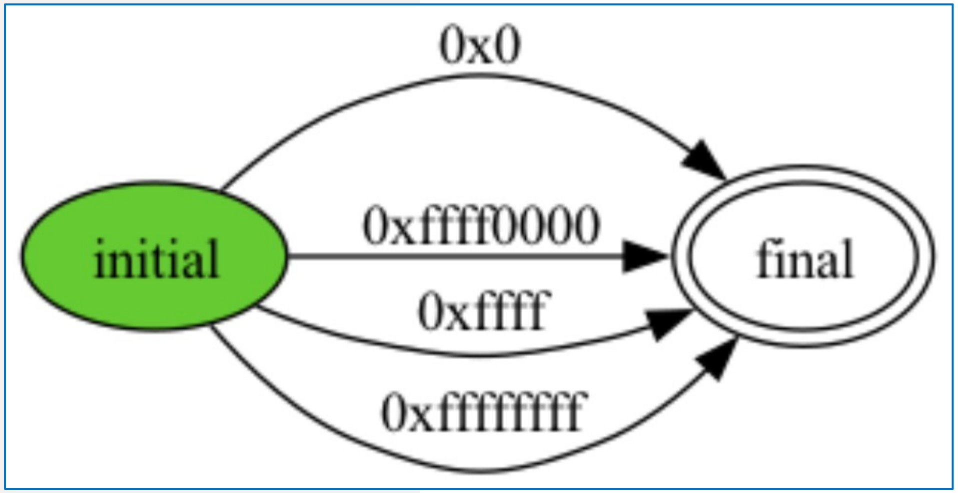
- CPU with 16-bit architecture
- 32-bit integer variable *x* stored in memory in two adjacent locations (aligned on word boundary)
- Initial value is 0
- Thread 1 writes FFFFFFFF to *x* (requires 2 STOREs)
- Thread 2 reads *x* (requires 2 LOADs)
- What are the possible values that thread 2 will read?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

Example modeled in Harmony

```
const MEM_SIZE  = 0x10
const WORD_SIZE = 16

const WORD_MASK = (1 << WORD_SIZE) - 1
memory = [0,] * MEM_SIZE

def hw_load(address) returns value:
    atomically value = memory[address]

def hw_store(address, value):
    atomically memory[address] = value

def load_double(address) returns value:
    value = hw_load(address) | (hw_load(address + 1) << WORD_SIZE)

def store_double(address, value):
    hw_store(address, value & WORD_MASK)
    hw_store(address + 1, (value >> WORD_SIZE) & WORD_MASK)

def f():
    store_double(0x6, 0xffffffff)

def g():
    print hex(load_double(0x6))

spawn f()
spawn g()
```

Example modeled in Harmony

```
const MEM_SIZE  = 0x10
const WORD_SIZE = 16

const WORD_MASK = (1 << WORD_SIZE) - 1
memory = [0,] * MEM_SIZE

def hw_load(address) returns value:
    atomically value = memory[address]

def hw_store(address, value):
    atomically memory[address] = value

def load_double(address) returns value:
    value = hw_load(address) | (hw_load(address + 1) << WORD_SIZE)

def store_double(address, value):
    hw_store(address, value & WORD_MASK)
    hw_store(address + 1, (value >> WORD_SIZE) & WORD_MASK)

def f():
    store_double(0x6, 0xffffffff)

def g():
    print hex(load_double(0x6))

spawn f()
spawn g()
```



Concurrent writing

- Hardware may also cause problems
  - e.g., buffering of writes to memory for improved performance
- 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 access the same variable
- And at least one access is a STORE
- Then the semantics of the outcome is *undefined*

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

- Mutual Exclusion can be implemented with atomic LOAD and STORE instructions to access shared memory
  - 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

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* to x
- Finally, !p refers to the value of x

Where?

There!

Specifying a lock

```
def Lock() returns result:
    result = False

def acquire(lk):
    atomically when not !lk:
        !lk = True

def release(lk):
    atomically:
        assert !lk
        !lk = False
```

Specifying a lock

```
def Lock() returns result:
    result = False

def acquire(lk):
    atomically when not !lk:
        !lk = True

def release(lk):
    atomically:
        assert !lk
        !lk = False
```

returns initial value

acquires lock atomically once available

releases lock atomically

Critical Section using a *lock*

```
from synch import Lock, acquire, release

shared = 0
thelock = Lock()

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

spawn f()
spawn f()

finally shared == 2
```

Critical Section using a *lock*

```
from synch import Lock, acquire, release

shared = 0
thelock = Lock()

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

spawn f()
spawn f()

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 (most?) systems do not keep track of who holds a particular lock, if anybody

Implementing a lock

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., TestAndSet *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”)

```
def test_and_set(s) returns result:
    atomically:
        result = !s
        !s = True

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

def Lock() returns result:
    result = False

def acquire(lk):
    while test_and_set(lk):
        pass

def release(lk):
    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

```
def Lock() returns result:
    result = False

def acquire(lk):
    atomically when not !lk:
        !lk = True

def release(lk):
    atomically:
        assert !lk
        !lk = False
```

```
def test_and_set(s) returns result:
    atomically:
        result = !s
        !s = True

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

def Lock() returns result:
    result = False

def acquire(lk):
    while test_and_set(lk):
        pass

def release(lk):
    atomic_store(lk, False)
```

Specification: describes *what an abstraction does*

Implementation: describes *how*

fine line

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**

```
def Lock() returns result:
    result = { .acquired: False, .suspended: [] }

def acquire(lk):
    atomically:
        if lk->acquired:
            stop ?lk->suspended[len lk->suspended]
            assert lk->acquired
        else:
            lk->acquired = True

def release(lk):
    atomically:
        assert lk->acquired
        if lk->suspended == []:
            lk->acquired = False
        else:
            go (lk->suspended[0]) ()
            del lk->suspended[0]
```

.acquired: boolean
.suspended: queue of contexts

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

```
def Lock() returns result:
    result = { .acquired: False, .suspended: [] }

def acquire(lk):
    atomically:
        if lk->acquired:
            stop ?lk->suspended[len lk->suspended]
            assert lk->acquired
        else:
            lk->acquired = True

def release(lk):
    atomically:
        assert lk->acquired
        if lk->suspended == []:
            lk->acquired = False
        else:
            go (lk->suspended[0]) ()
            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**

```
def Lock() returns result:
    result = { .acquired: False, .suspended: [] }

def acquire(lk):
    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.

```
    atomically:
        assert lk->acquired
        if lk->suspended == []:
            lk->acquired = False
        else:
            go (lk->suspended[0]) ()
            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 ≠ 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: locks available in many mainstream programming languages
good for specifying interlock instructions	good for implementing concurrent data structures

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()
```

Review

Summary

- 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 may be undefined and almost always is a bug
- A *Race Condition* occurs when the correctness of the program depends on ordering of variable access
  - Race Condition does not imply Data Race

Summary, cont'd

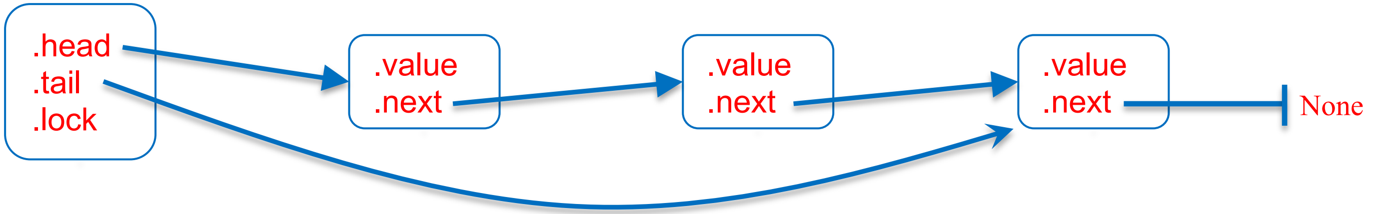
- A *Critical Section* consists of one or more regions of code in which at most thread can execute at a time
  - usually protected by a *lock*
  - not the same as atomic because threads can continue to execute other code
- Beware of code with multiple critical sections
  - e.g., code that uses multiple locks

Concurrent Data Structure Consistency

CORNELL UNIVERSITY FOUNDED A.D. 1865
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 $\Longrightarrow$ data structure consistent
- *Or equivalently*:
  - data structure inconsistent $\Longrightarrow$ 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

```
def Queue() returns empty:
    empty = []

def put(q, v):
    !q += [v,]

def get(q) returns next:
    if !q == []:
        next = None
    else:
        next = (!q)[0]
        del (!q)[0]
```

Specifying a concurrent queue

```
def Queue() returns empty:
    empty = []

def put(q, v):
    !q += [v,]

def get(q) returns next:
    if !q == []:
        next = None
    else:
        next = (!q)[0]
        del (!q)[0]
```

Sequential

```
def Queue() returns empty:
    empty = []

def put(q, v):
    atomically !q += [v,]

def get(q) returns next:
    atomically:
        if !q == []:
            next = None
        else:
            next = (!q)[0]
            del (!q)[0]
```

Concurrent

Example of using a queue

```
import queue

def sender(q, v):
    queue.put(q, v)

def receiver(q):
    let v = queue.get(q):
        assert v in { None, 1, 2 }

demoq = queue.Queue()
spawn sender(?demoq, 1)
spawn sender(?demoq, 2)
spawn receiver(?demoq)
spawn receiver(?demoq)
```

enqueue *v* onto !*q*

dequeue and check

create queue

Specifying a concurrent queue

```
def Queue() returns empty:
    empty = []

def put(q, v):
    atomically !q += [v,]

def get(q) returns next:
    atomically:
        if !q == []:
            next = None
        else:
            next = (!q)[0]
            del (!q)[0]
```

Not a good implementation because

- operations are O(n)
- code uses **atomically**

  compiler cannot generate code

Implementing a concurrent queue

How important are concurrent queues?

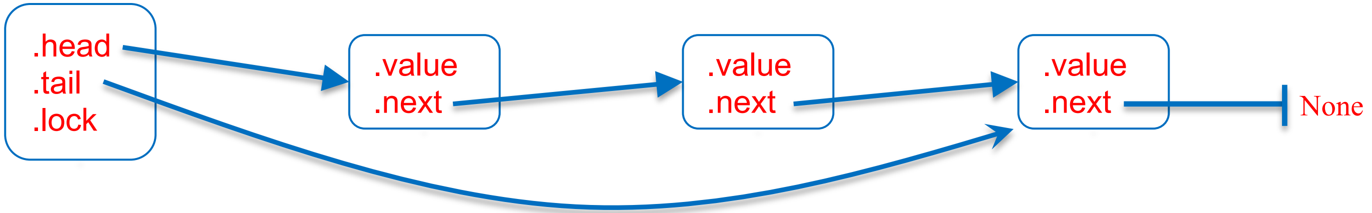
- Answer: all important
  - any resource that needs scheduling
    - CPU run queue
    - disk, network, printer waiting queue
    - lock waiting queue
  - inter-process communication
    - Posix pipes:
      - cat file | tr a-z A-Z | grep RVR
  - actor-based concurrency
  - …

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
  - …

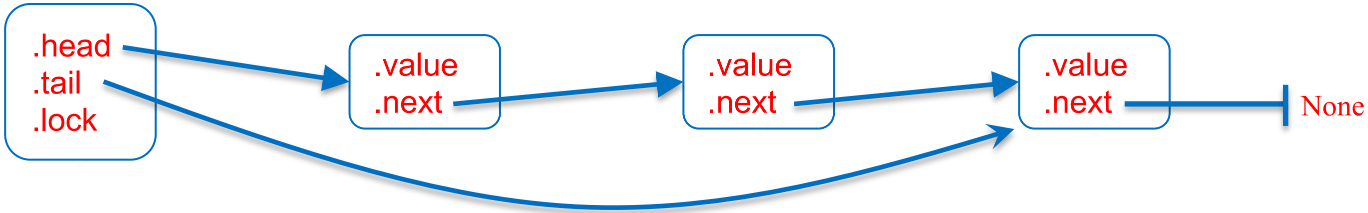
Good performance is critical!

Queue implementation, v1




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue() returns empty:
    empty = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q->lock)
        if q->tail == None:
            q->tail = q->head = node
        else:
            q->tail->next = node
            q->tail = node
        release(?q->lock)
```

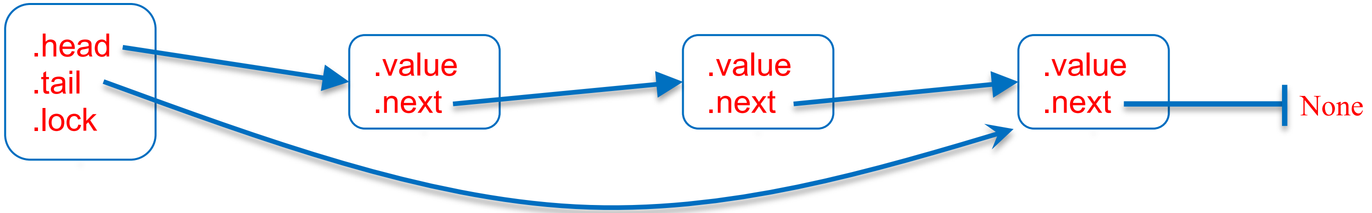
Queue implementation, v1



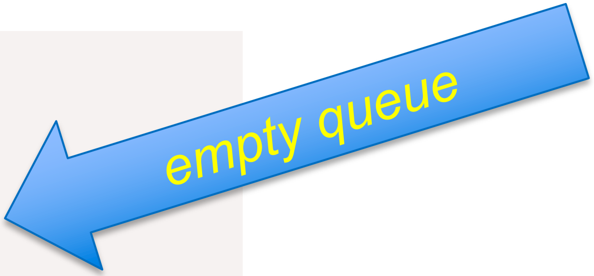

```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue() returns empty:
    empty = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q->lock)
        if q->tail == None:
            q->tail = q->head = node
        else:
            q->tail->next = node
            q->tail = node
        release(?q->lock)
```

dynamic memory allocation

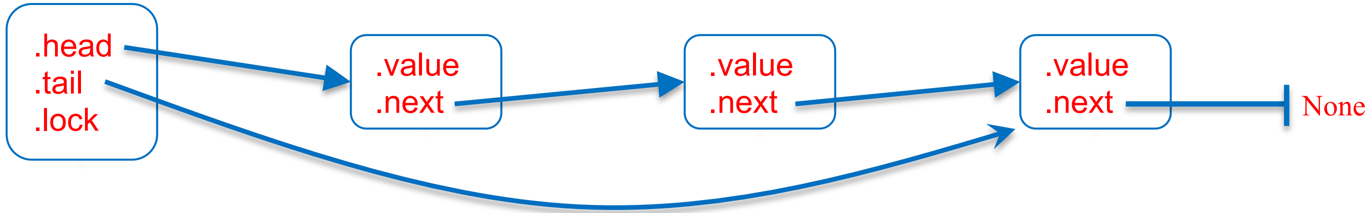
Queue implementation, v1




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue() returns empty:
    empty = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q->lock)
        if q->tail == None:
            q->tail = q->head = node
        else:
            q->tail->next = node
            q->tail = node
        release(?q->lock)
```



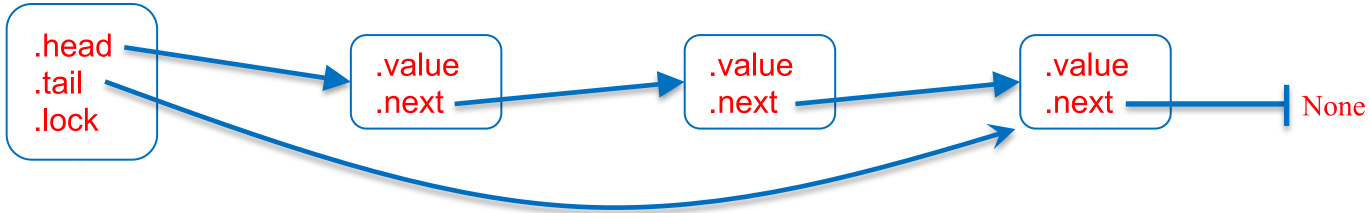
Queue implementation, v1




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue() returns empty:
    empty = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q->lock)
        if q->tail == None:
            q->tail = q->head = node
        else:
            q->tail->next = node
            q->tail = node
        release(?q->lock)
```

allocate node

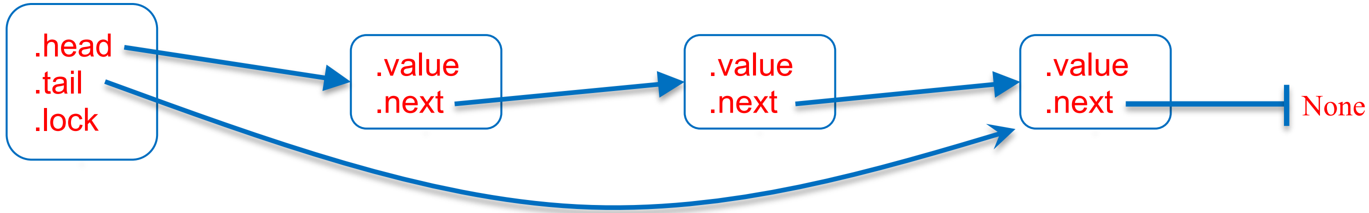
Queue implementation, v1




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue() returns empty:
    empty = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q->lock)
        if q->tail == None:
            q->tail = q->head = node
        else:
            q->tail->next = node
            q->tail = node
        release(?q->lock)
```

grab lock

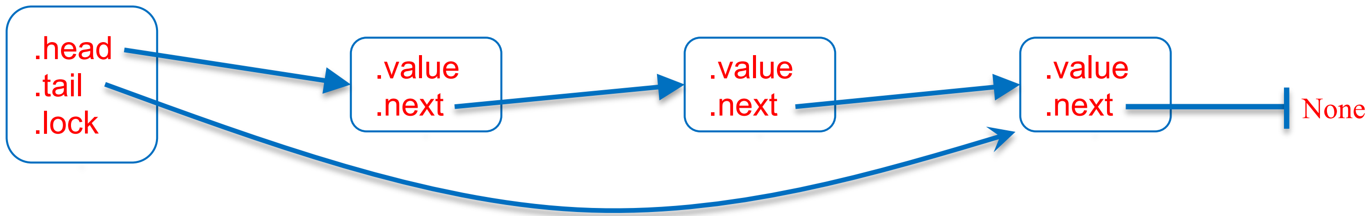
Queue implementation, v1




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue() returns empty:
    empty = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q->lock)
        if q->tail == None:
            q->tail = q->head = node
        else:
            q->tail->next = node
            q->tail = node
        release(?q->lock)
```

the hard stuff

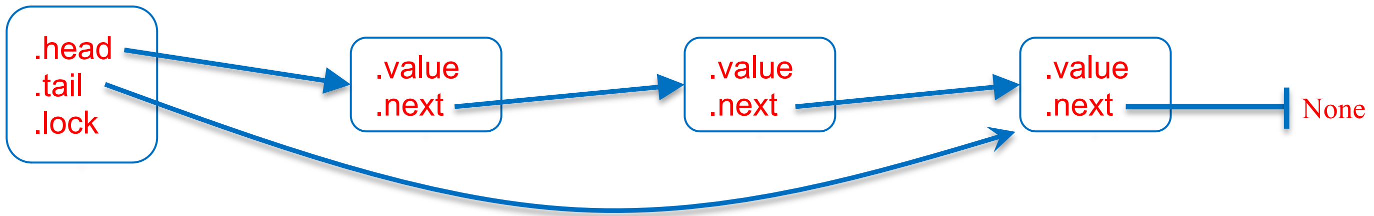
Queue implementation, v1




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue() returns empty:
    empty = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q->lock)
        if q->tail == None:
            q->tail = q->head = node
        else:
            q->tail->next = node
            q->tail = node
        release(?q->lock)
```

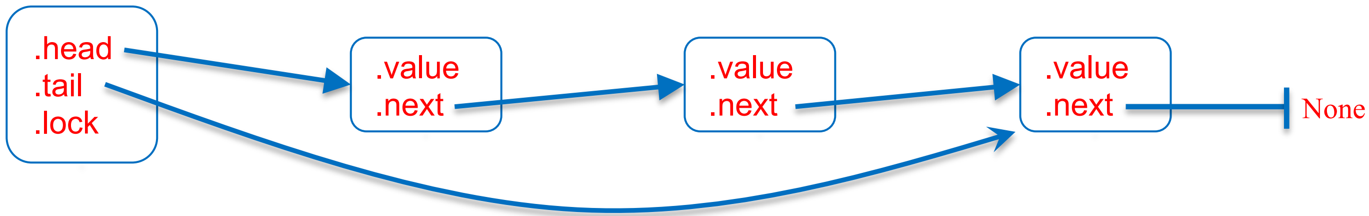
release lock

Queue implementation, v1




```
def get(q) returns next:
    acquire(?q->lock)
    let node = q->head:
        if node == None:
            next = None
        else:
            next = node->value
            q->head = node->next
            if q->head == None:
                q->tail = None
            free(node)
    release(?q->lock)
```

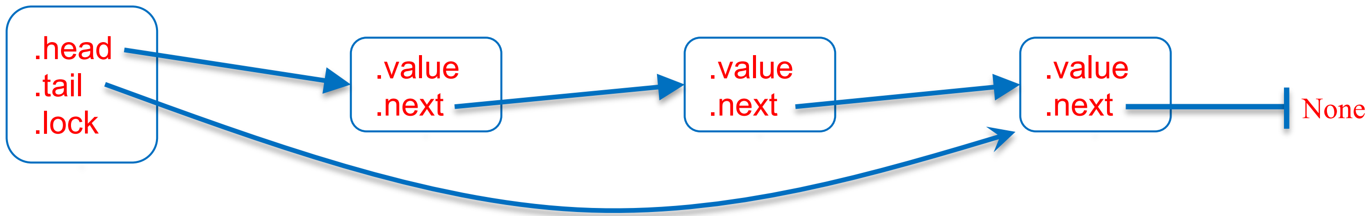
Queue implementation, v1




```
def get(q) returns next:
    acquire(?q->lock)
    let node = q->head:
        if node == None:
            next = None
        else:
            next = node->value
            q->head = node->next
            if q->head == None:
                q->tail = None
            free(node)
    release(?q->lock)
```

the hard stuff

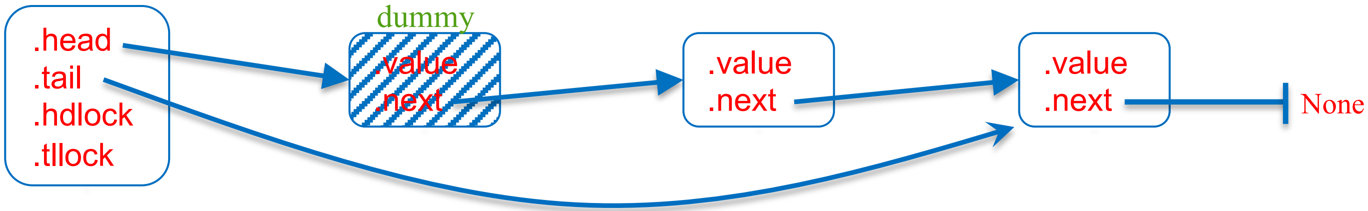
Queue implementation, v1




```
def get(q) returns next:
    acquire(?q->lock)
    let node = q->head:
        if node == None:
            next = None
        else:
            next = node->value
            q->head = node->next
            if q->head == None:
                q->tail = None
            free(node)
    release(?q->lock)
```

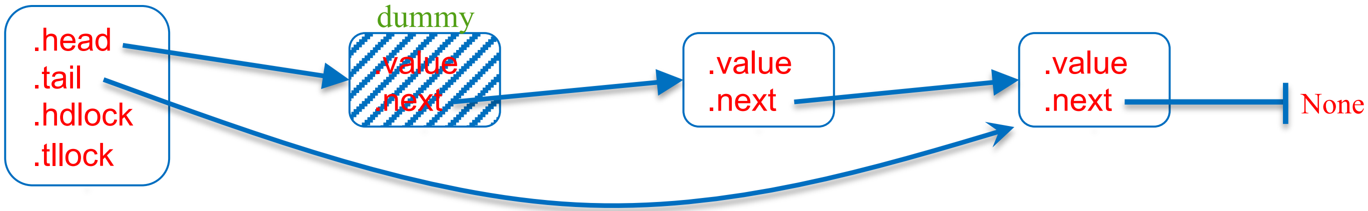
malloc'd memory must be explicitly released (cf. C)

Concurrent queue v2: 2 locks

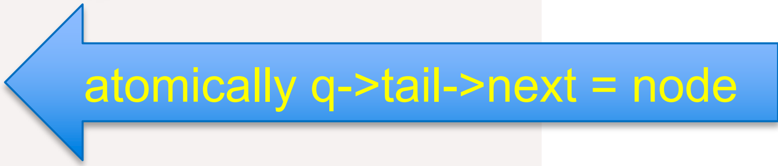



```
from synch import Lock, acquire, release, atomic_load, atomic_store
from alloc import malloc, free

def Queue() returns empty:
    let dummy = malloc({ .value: (), .next: None }):
        empty = { .head: dummy, .tail: dummy,
                  .hdlock: Lock(), .tllock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q->tllock)
        atomic_store(?q->tail->next, node)
        q->tail = node
        release(?q->tllock)
```

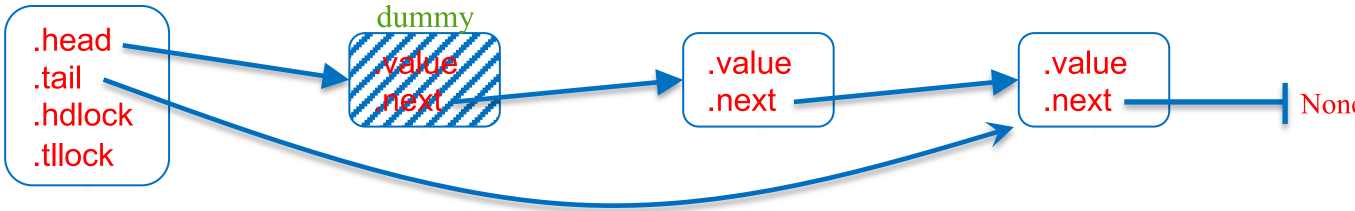
Concurrent queue v2: 2 locks




```
from synch import Lock, acquire, release, atomic_load, atomic_store
from alloc import malloc, free

def Queue() returns empty:
    let dummy = malloc({ .value: (), .next: None }):
        empty = { .head: dummy, .tail: dummy,
                        .hdlock: Lock(), .tllock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q->tllock)
        atomic_store(?q->tail->next, node)
        q->tail = node
        release(?q->tllock)
```

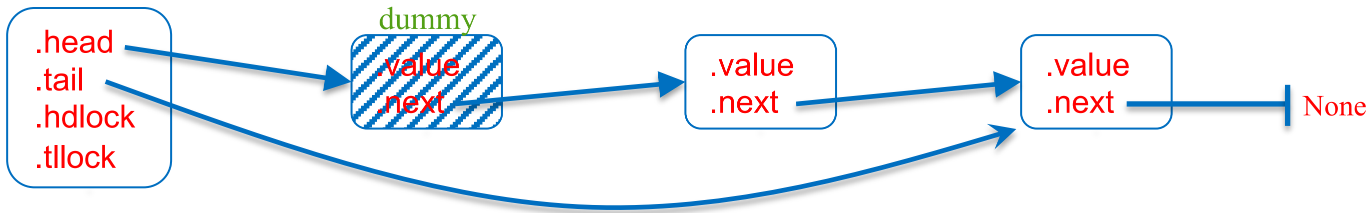


Concurrent queue v2: 2 locks




```
def get(q) returns next:
    acquire(?q->hdlock)
    let dummy = q->head
    let node = atomic_load(?dummy->next):
        if node == None:
            next = None
            release(?q->hdlock)
        else:
            next = node->value
            q->head = node
            release(?q->hdlock)
            free(dummy)
```

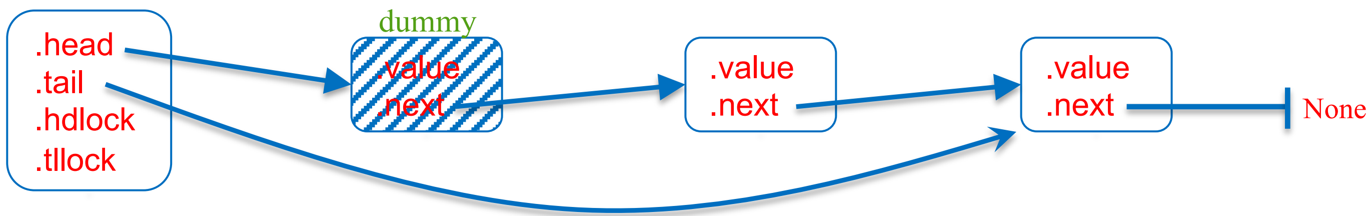
Concurrent queue v2: 2 locks




```
def get(q) returns next:
    acquire(?q->hdlock)
    let dummy = q->head
    let node = atomic_load(?dummy->next):
        if node == None:
            next = None
            release(?q->hdlock)
        else:
            next = node->value
            q->head = node
            release(?q->hdlock)
            free(dummy)
```

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

Concurrent queue v2: 2 locks




```
def get(q) returns next:
    acquire(?q->hdlock)
    let dummy = q->head
    let node = atomic_load(?dummy->next):
        if node == None:
            next = None
            release(?q->hdlock)
        else:
            next = node->value
            q->head = node
            release(?q->hdlock)
            free(dummy)
```

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

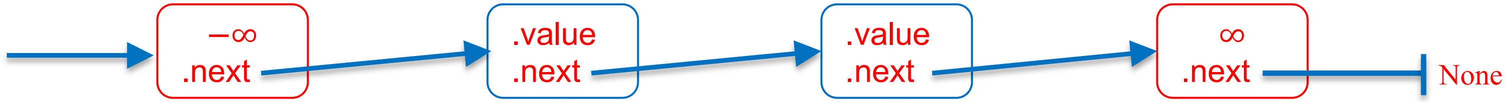
Needs to avoid data race on *dummy*→*next* when queue is empty

Fine-Grained Locking

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



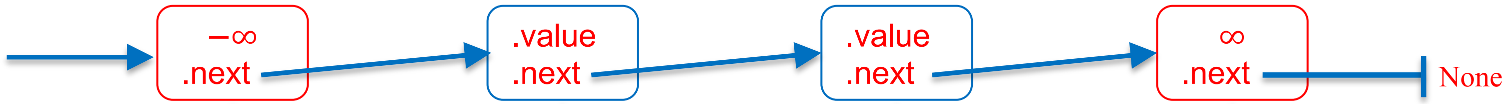

```
from synch import Lock, acquire, release
from alloc import malloc, free

def _node(v, n) returns node:   # allocate and initialize
    node = malloc({ .lock: Lock(), .value: v, .next: n })

def _find(lst, v) returns pair:
    var before = lst
    acquire(?before->lock)
    var after = before->next
    acquire(?after->lock)
    while after->value < (0, v):
        release(?before->lock)
        before = after
        after = before->next
        acquire(?after->lock)
    pair = (before, after)

def SetObject() returns object:
    object = _node((-1, None), _node((1, None), None))
```

- $-\infty$ represented by (-1, None)
- v represented by (0, v)
- ∞ represented by (1, None)

Note that ∀v: (-1, None) < (0, v) < (1, None)
(lexicographical ordering)

empty list

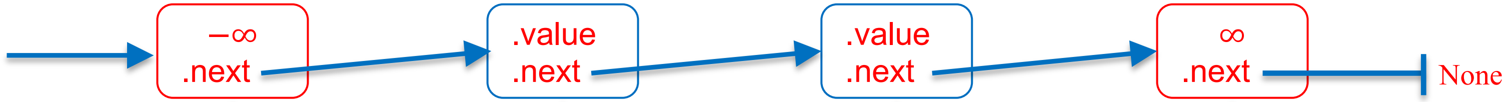
Sorted Linked List with Lock per Node




```
from synch import Lock, acquire, release
from alloc import malloc, free

def _node(v, n) returns node:    # allocate and initialize
    node = malloc({ .lock: Lock(), .value: v, .next: n })

def _find(lst, v) returns pair:
    var before = lst
    acquire(?before->lock)
    var after = before->next
    acquire(?after->lock)
    while after->value < (0, v):
        release(?before->lock)
        before = after
        after = before->next
        acquire(?after->lock)
    pair = (before, after)

def SetObject() returns object:
    object = _node((-1, None), _node((1, None), None))
```

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

Sorted Linked List with Lock per Node




```
from synch import Lock, acquire, release
from alloc import malloc, free

def _node(v, n) returns node:    # allocate and initialize
    node = malloc({ .lock: Lock(), .value: v, .next: n })

def _find(lst, v) returns pair:
    var before = lst
    acquire(?before->lock)
    var after = before->next
    acquire(?after->lock)
    while after->value < (0, v):
        release(?before->lock)
        before = after
        after = before->next
        acquire(?after->lock)
    pair = (before, after)

def SetObject() returns object:
    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)
        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)
        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

Systematic Testing

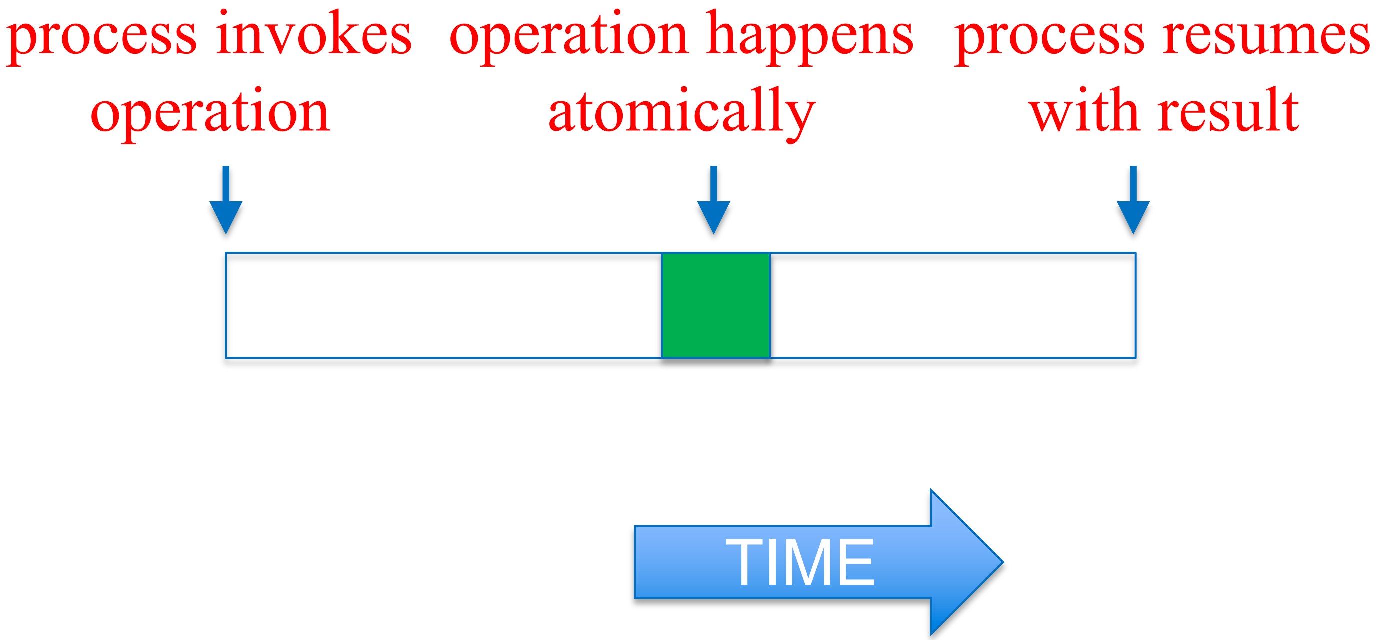
Systematic Testing

- 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

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 (afterwards)
3. customer Y is served a burger
4. customer X is served a burger (afterwards)

Concurrency and Overlap

Is the following a possible scenario?

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

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

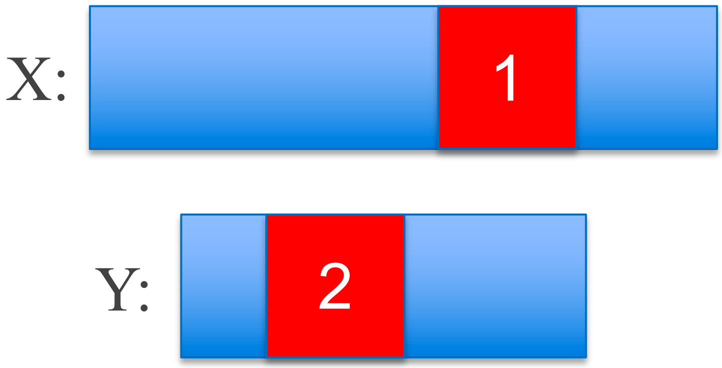
Specification

- 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
- Allows many implementations

Implementation?

- Suppose the diner has one small hot plate and two cooks
- Cooks use a lock for access to the hot plate
- Possible scenario:

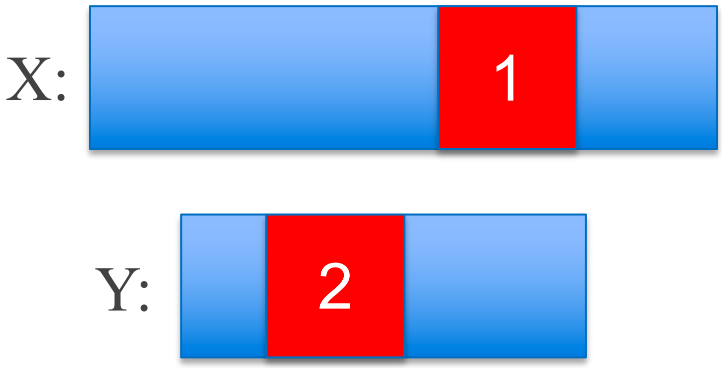
1. customer X orders burger, order ends up with cook 1
2. customer Y orders burger, order ends up with cook 2
3. cook 1 was busy with something else, so cook 2 grabs the lock first
4. cook 2 cooks burger for Y
5. cook 2 releases lock
6. cook 1 grabs lock
7. cook 1 cooks burger for X
8. cook 1 releases 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:

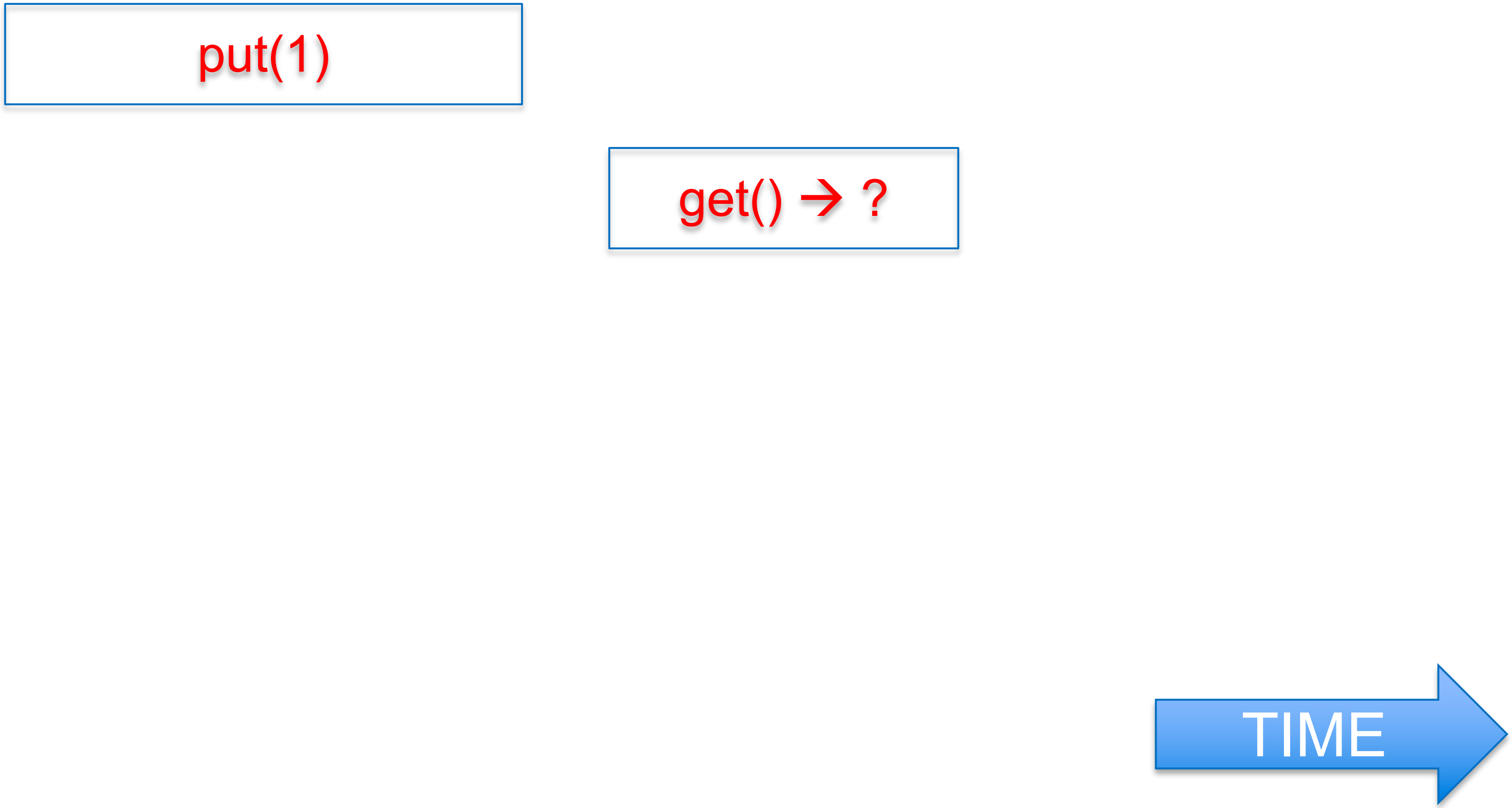
1. customer X orders burger, order ends up with cook 1
2. customer Y orders burger, order ends up with cook 2
3. cook 1 was busy with something else, so cook 2 grabs the lock first
4. cook 2 cooks burger for Y
5. cook 2 releases lock
6. cook 1 grabs lock
7. cook 1 cooks burger for X
8. cook 1 releases lock
9. customer Y receives burger
10. customer X receives burger



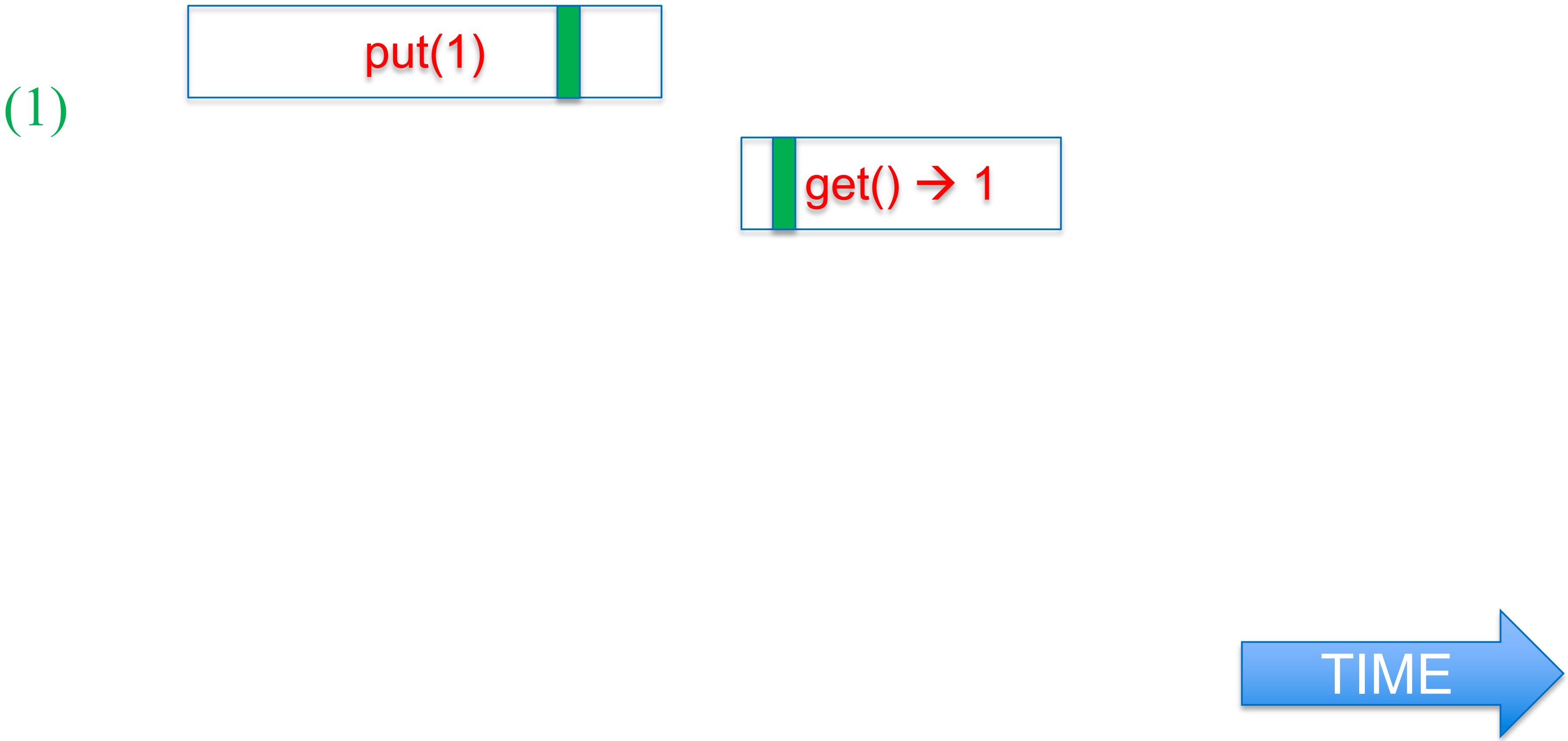
- *can't happen if Y orders burger after X receives burger*
- *but if operations overlap, any ordering can happen…*

Correct Behaviors

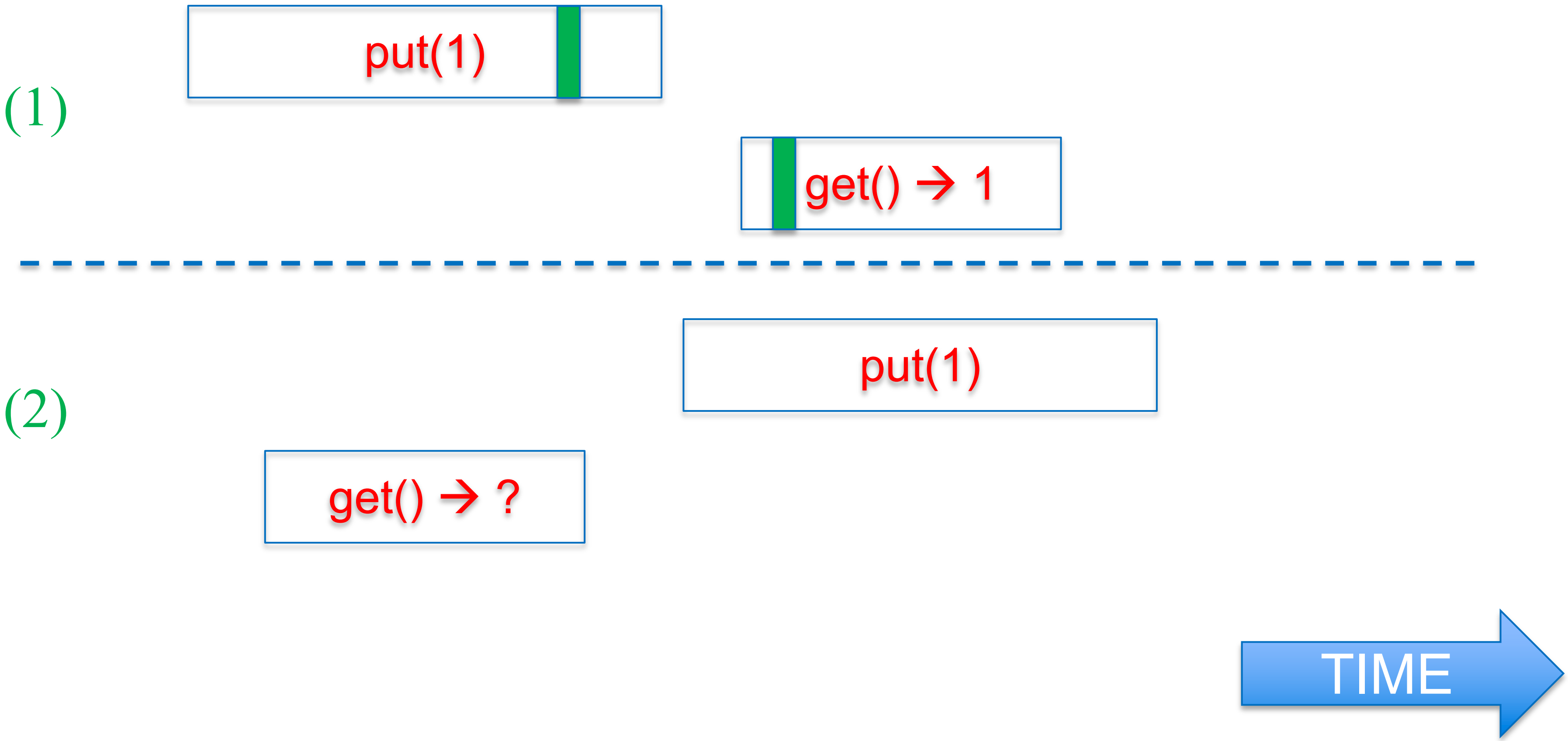
(1)



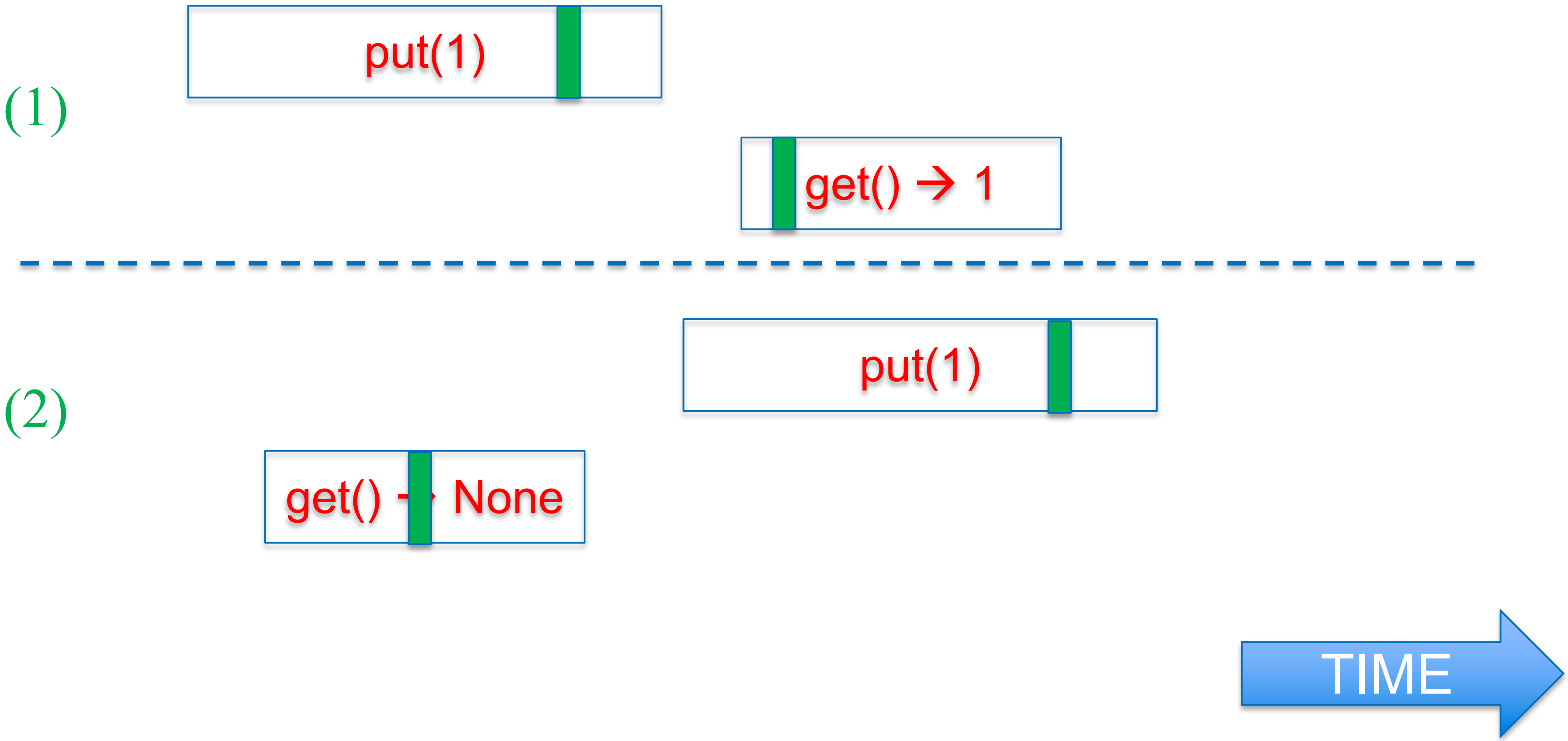
Correct Behaviors



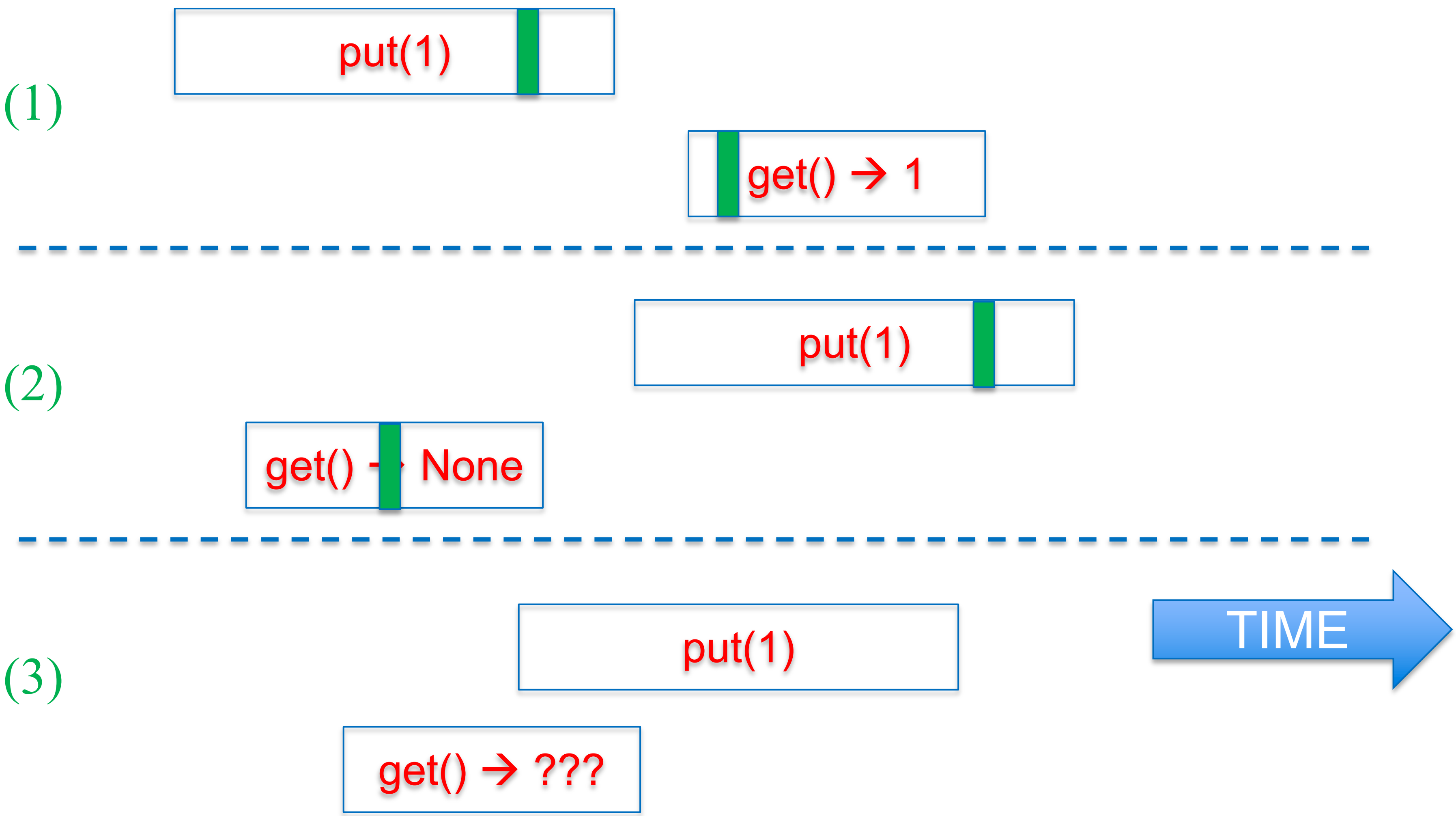
Correct Behaviors



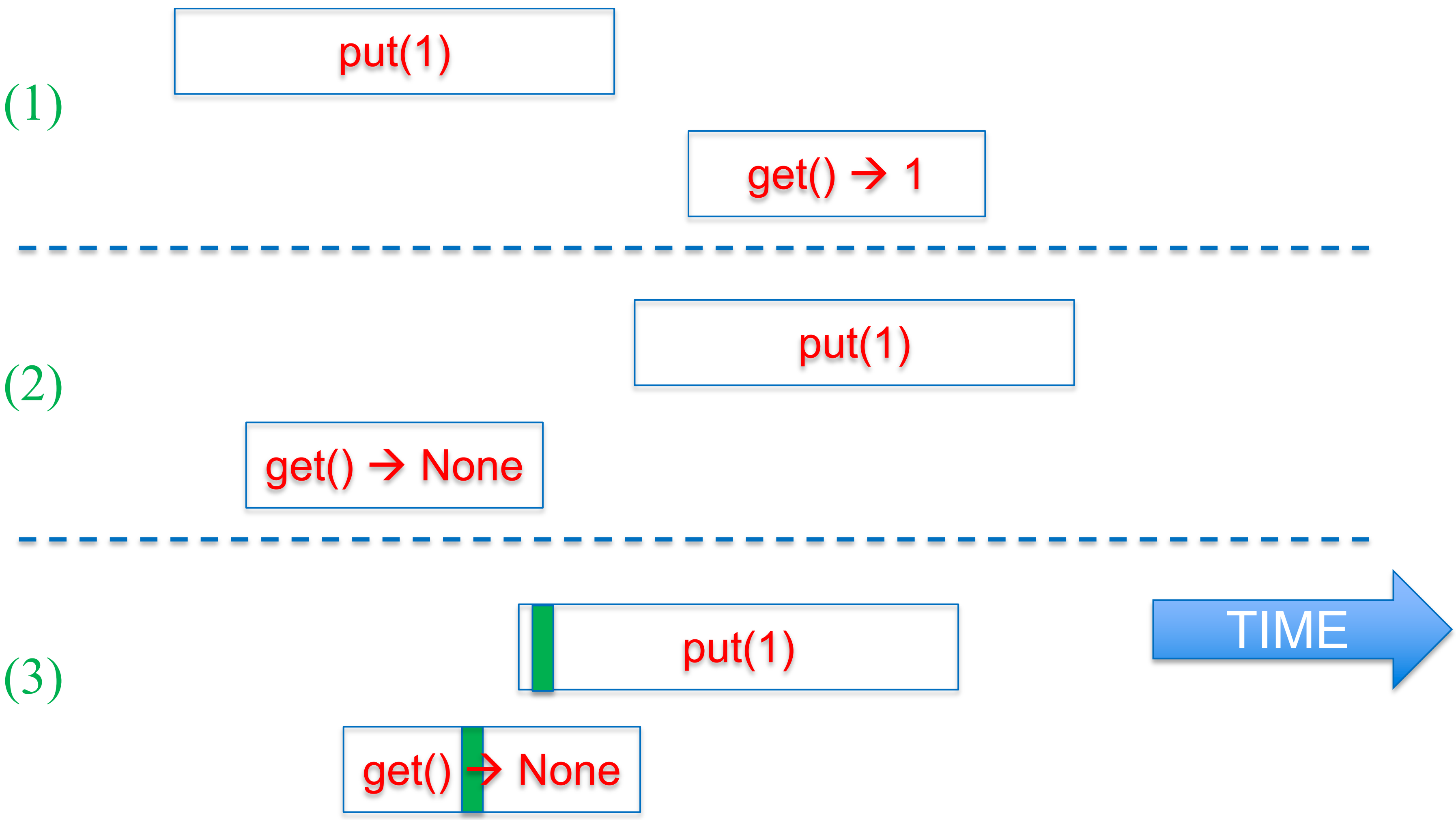
Correct Behaviors



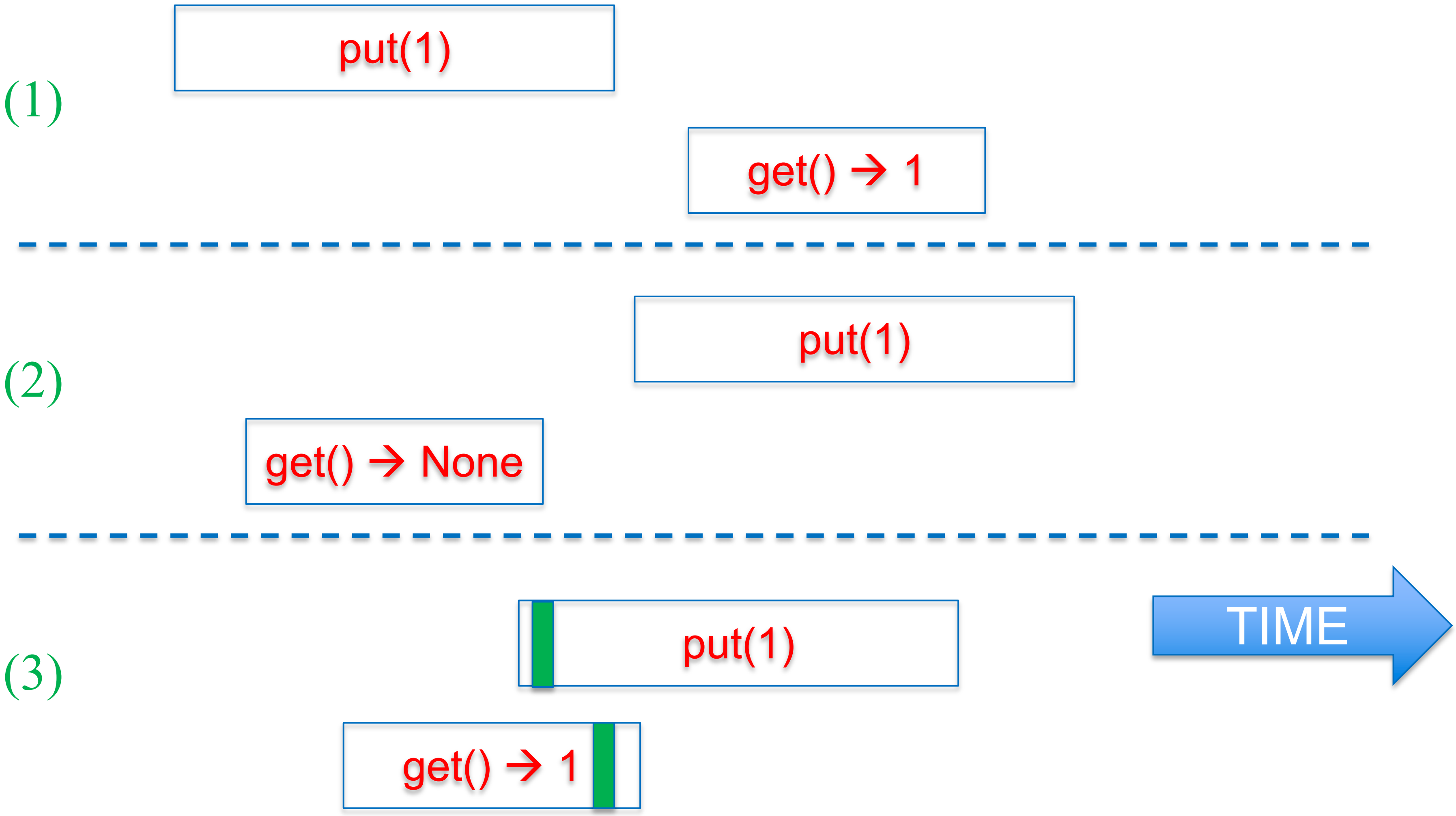
Correct Behaviors



Correct Behaviors



Correct Behaviors



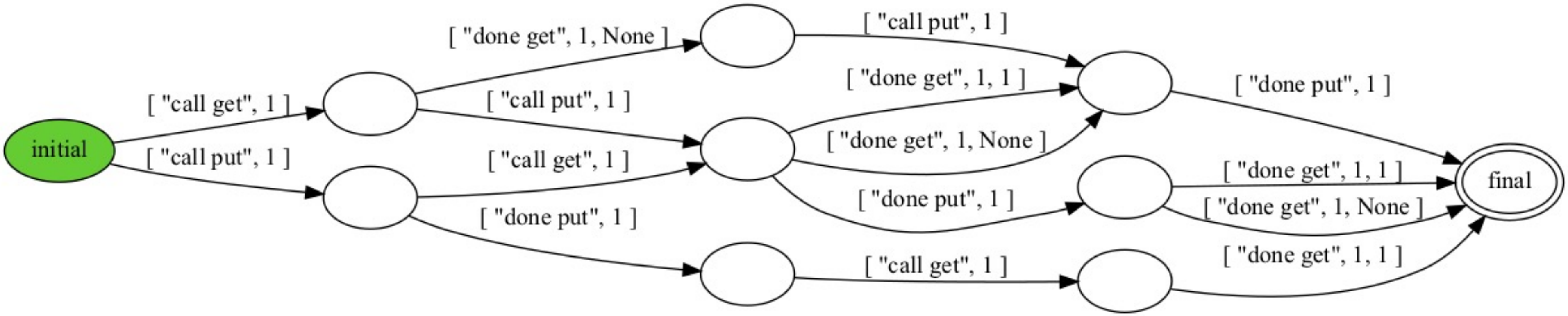
Testing Concurrent Objects

- 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

Concurrent queue test program

```
import queue

const N_PUT = 2
const N_GET = 2
q = queue.Queue()

def put_test(self):
    print("call put", self)
    queue.put(?q, self)
    print("done put", self)

def get_test(self):
    print("call get", self)
    let v = queue.get(?q):
        print("done get", self, v)

for i in {1..N_PUT}:
    spawn put_test(i)
for i in {1..N_GET}:
    spawn get_test(i)
```

Behavior (1 get, 1 put)




```
$ harmony -c N_GET=1 –c N_PUT=1 -o spec.png code/queue_btest1.hny
```

Testing: comparing behaviors

```
$ harmony -o queue4.hfa code/queue_btest1.hny
$ harmony -B queue4.hfa -m queue=queue_lock code/queue_btest1.hny
```

- The first command outputs the behavior of running the test program against the specification in file queue4.hfa
- The second command runs the test program against the implementation and checks if its behavior matches that stored in queue4.hfa

Black Box Testing

- Not allowed to look under the covers
  - can't use *rw*->nreaders, 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?
- In general testing cannot ensure correctness
  - only a correctness proof can
  - testing may or may not expose a bug
  - model checking helps expose bugs

Conditional Waiting

Review

- Concurrent Programming is Hard!
  - Non-Determinism
  - Non-Atomicity
- *Critical Sections* simplify things by avoiding data races
  - mutual exclusion
  - progress
  - *Need both mutual exclusion and progress!*
- Critical Sections use a *lock*
  - Thread needs lock to enter the critical section
  - Only one thread can get the section's lock

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 writers) in a reader/writer lock
  - …

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

```
def RWlock() returns lock:
    lock = { .nreaders: 0, .nwriters: 0 }

def read_acquire(rw):
    atomically when rw->nwriters == 0:
        rw->nreaders += 1

def read_release(rw):
    atomically rw->nreaders -= 1

def write_acquire(rw):
    atomically when (rw->nreaders + rw->nwriters) == 0:
        rw->nwriters = 1

def write_release(rw):
    atomically rw->nwriters = 0
```

Reader/Writer Lock Specification

```
def RWlock() returns lock:
    lock = { .nreaders: 0, .nwriters: 0 }

def read_acquire(rw):
    atomically when rw->nwriters == 0:
        rw->nreaders += 1

def read_release(rw):
    atomically rw->nreaders -= 1

def write_acquire(rw):
    atomically when (rw->nreaders + rw->nwriters) == 0:
        rw->nwriters = 1

def write_release(rw):
    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 $nwriters \geq n$
- $(nreaders \geq 0 \land nwriters = 0) \lor (nreaders = 0 \land 0 \leq nwriters \leq 1)$

R/W Locks: test for mutual exclusion

```
import rwlock

nreaders = nwriters = 0
invariant ((nreaders >= 0) and (nwriters == 0)) or
            ((nreaders == 0) and (nwriters == 1))

const NOPS = 4

rw = rwlock.RWlock()

def thread():
    while choose({ False, True }):
        if choose({ "read", "write" }) == "read":
            rwlock.read_acquire(?rw)
            atomically nreaders += 1
            atomically nreaders -= 1
            rwlock.read_release(?rw)
        else:                          # write
            rwlock.write_acquire(?rw)
            atomically nwriters += 1
            atomically nwriters -= 1
            rwlock.write_release(?rw)

for i in {1..NOPS}:
    spawn thread()
```

no writer, one or more readers

one writer, no readers

Cheating R/W lock implementation

```
import synch

def RWlock() returns lock:
    lock = synch.Lock()

def read_acquire(rw):
    synch.acquire(rw)

def read_release(rw):
    synch.release(rw)

def write_acquire(rw):
    synch.acquire(rw)

def write_release(rw):
    synch.release(rw)
```

The *lock* protects the application's critical section

Cheating R/W lock implementation

```
import synch

def RWlock() returns lock:
    lock = synch.Lock()

def read_acquire(rw):
    synch.acquire(rw)

def read_release(rw):
    synch.release(rw)

def write_acquire(rw):
    synch.acquire(rw)

def write_release(rw):
    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

```
from synch import Lock, acquire, release

def RWlock() returns lock:
    lock = { .lock: Lock(), .nreaders: 0, .nwriters: 0 }

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 write_acquire(rw):
    acquire(?rw->lock)
    while rw->nreaders > 0 or rw->nwriters > 0:
        release(?rw->lock)
        acquire(?rw->lock)
    rw->nwriters = 1
    release(?rw->lock)

def write_release(rw):
    acquire(?rw->lock)
    rw->nwriters = 0
    release(?rw->lock)
```

Busy Waiting Implementation

```
from synch import Lock, acquire, release

def RWlock() returns lock:
    lock = { .lock: Lock(), .nreaders: 0, .nwriters: 0 }

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 write_acquire(rw):
    acquire(?rw->lock)
    while rw->nreaders > 0 or rw->nwriters > 0:
        release(?rw->lock)
        acquire(?rw->lock)
    rw->nwriters = 1
    release(?rw->lock)

def write_release(rw):
    acquire(?rw->lock)
    rw->nwriters = 0
    release(?rw->lock)
```

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

Busy Waiting Implementation

```
from synch import Lock, acquire, release

def RWlock() returns lock:
    lock = { .lock: Lock(), .nreaders: 0, .nwriters: 0 }

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 write_acquire(rw):
    acquire(?rw->lock)
    while rw->nreaders > 0 or rw->nwriters > 0:
        release(?rw->lock)
        acquire(?rw->lock)
    rw->nwriters = 1
    release(?rw->lock)

def write_release(rw):
    acquire(?rw->lock)
    rw->nwriters = 0
    release(?rw->lock)
```

waiting conditions

Busy Waiting Implementation

```
from synch import Lock, acquire, release

def RWlock() returns lock:
    lock = { .lock: Lock(), .nreaders: 0, .nwriters: 0 }

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 write_acquire(rw):
    acquire(?rw->lock)
    while rw->nreaders > 0 or rw->nwriters > 0:
        release(?rw->lock)
        acquire(?rw->lock)
    rw->nwriters = 1
    release(?rw->lock)

def write_release(rw):
    acquire(?rw->lock)
    rw->nwriters = 0
    release(?rw->lock)
```

Good: has the same behaviors as the implemention

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*

```
from synch import *

def RWlock() returns lock:
    lock = {
            .nreaders: 0, .nwriters: 0, .mutex: Lock(),
            .r_cond: Condition(), .w_cond: Condition()
        }
```

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

```
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

```
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)
```

similar to busy waiting

R/W Lock, reader part

```
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)
```

similar to busy waiting

but need this

R/W Lock, reader part

```
def read_acquire(rw):
    acquire(?rw->mutex)
    while rw->nwriters > 0:
        wait(?rw->r_cond, ?rw->mutex)
    rw->nreade          1
    releas

def read_r
    acquir
    rw->nr
    if rw->nreaders == 0:
        notify(?rw->w_cond)
    release(?rw->mutex)
```

similar to busy waiting

but need this

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

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

```
def write_acquire(rw):
    acquire(?rw->mutex)
    while rw->nreaders > 0 or rw->nwriters > 0:
        wait(?rw->w_cond, ?rw->mutex)
    rw->nwriters = 1
    release(?rw->mutex)

def write_release(rw):
    acquire(?rw->mutex)
    rw->nwriters = 0
    notify_all(?rw->r_cond)
    notify(?rw->w_cond)
    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 the lock and is running
- Now suppose a clock interrupt occurs, T2 starts running and tries to acquire the lock
- Non-busy-waiting acquisition:
    - T2 is put on a waiting queue and T1 resumes and runs until T1 releases the lock (which puts T2 back on the run queue)
- Busy-waiting acquisition:
    - T2 keeps running (wasting CPU) until the lock is available until the next clock interrupt
    - T1 and T2 switch back and forth until T1 releases the lock

Busy Waiting vs Condition Variables

Busy Waiting	Condition Variables
Use a lock and a loop	Use a lock *and a collection of condition variables* 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

Busy Waiting: just don’t do it

Why no naked waits? (reason 1)

- By the time waiter gets the lock back, condition may no longer hold
  - 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()
- But this too 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

Naked waits: just don't do it

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
  - sometimes you can even 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

```
from synch import Lock, acquire, release

def Account(balance) returns account:
    account = { .lock: Lock(), .balance: balance }

accounts = [ Account(3), Account(7), Account(0) ]

def transfer(a1, a2, amount):
    acquire(?accounts[a1].lock)
    if amount <= accounts[a1].balance:
        accounts[a1].balance -= amount
        acquire(?accounts[a2].lock)
        accounts[a2].balance += amount
        release(?accounts[a2].lock)
    release(?accounts[a1].lock)

spawn transfer(0, 1, 1)
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

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

synch:34 **def acquire(binsema):**

934	LoadVar binsema
935	DelVar binsema
936	Load
937	StoreVar result
938	ReturnOp(result)
939	Jump 1214
940	Frame held(binsema)

Threads

ID	Status	Stack Trace		Stack Top
T0	terminated	__init__()		
T1	blocked	transfer(0, 1, 1)	a1: 0, a2: 1, amount: 1	
		acquire(?accounts[1]["lock"])	binsema: ?accounts[1]["lock"]	
T2	blocked	transfer(1, 0, 2)	a1: 1, a2: 0, amount: 2	
		acquire(?accounts[0]["lock"])	binsema: ?accounts[0]["lock"]	

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 $\exists T_1, \ldots T_n : T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$

Example: Mutual Exclusion

```
from synch import Lock, acquire, release

def Account(balance) returns account:
    account = { .lock: Lock(), .balance: balance }

accounts = [ Account(3), Account(7), Account(0) ]

def transfer(a1, a2, amount):
    acquire(?accounts[a1].lock)
    if amount <= accounts[a1].balance:
        accounts[a1].balance -= amount
        acquire(?accounts[a2].lock)
        accounts[a2].balance += amount
        release(?accounts[a2].lock)
    release(?accounts[a1].lock)

spawn transfer(0, 1, 1)
spawn transfer(1, 0, 2)
```

Mutual exclusion

Mutual exclusion

Example: Hold & Wait

```
from synch import Lock, acquire, release

def Account(balance) returns account:
    account = { .lock: Lock(), .balance: balance }

accounts = [ Account(3), Account(7), Account(0) ]

def transfer(a1, a2, amount):
    acquire(?accounts[a1].lock)
    if amount <= accounts[a1].balance:
        accounts[a1].balance -= amount
        acquire(?accounts[a2].lock)
        accounts[a2].balance += amount
        release(?accounts[a2].lock)
    release(?accounts[a1].lock)

spawn transfer(0, 1, 1)
spawn transfer(1, 0, 2)
```

Thread holds a1.lock

Threads wants a2.lock

Example: No Preemption

```
from synch import Lock, acquire, release

def Account(balance) returns account:
    account = { .lock: Lock(), .balance: balance }

accounts = [ Account(3), Account(7), Account(0) ]

def transfer(a1, a2, amount):
    acquire(?accounts[a1].lock)
    if amount <= accounts[a1].balance:
        accounts[a1].balance -= amount
        acquire(?accounts[a2].lock)
        accounts[a2].balance += amount
        release(?accounts[a2].lock)
    release(?accounts[a1].lock)

spawn transfer(0, 1, 1)
spawn transfer(1, 0, 2)
```

Only holder can release lock

Example: Circular Wait

```
from synch import Lock, acquire, release

def Account(balance) returns account:
    account = { .lock: Lock(), .balance: balance }

accounts = [ Account(3), Account(7), Account(0) ]

def transfer(a1, a2, amount):
    acquire(?accounts[a1].lock)
    if amount <= accounts[a1].balance:
        accounts[a1].balance -= amount
        acquire(?accounts[a2].lock)
        accounts[a2].balance += amount
        release(?accounts[a2].lock)
    release(?accounts[a1].lock)

spawn transfer(0, 1, 1)
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 (for example, by executing one thread at a time)

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

Deadlock Prevention

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
  - make sure bounded buffer is large enough

2. Negate Hold & Wait

```
from synch import Lock, acquire, release

def Account(balance) returns account:
    account = { .lock: Lock(), .balance: balance }

accounts = [ Account(3), Account(7) ]

def transfer(a1, a2, amount):
    acquire(?accounts[a1].lock)
    if amount <= accounts[a1].balance:
        accounts[a1].balance -= amount
        release(?accounts[a1].lock)
        acquire(?accounts[a2].lock)
        accounts[a2].balance += amount
        release(?accounts[a2].lock)
    else:
        release(?accounts[a1].lock)

spawn transfer(0, 1, 1)
spawn transfer(1, 0, 2)
```

Release resource before acquiring another

2: Negate Hold & Wait, badly

```
from synch import Lock, acquire, release

def Account(balance) returns account:
    account = { .lock: Lock(), .balance: balance }

accounts = [ Account(3), Account(7) ]

invariant all(a.balance >= 0 for a in accounts)

def transfer(a1, a2, amount):
    acquire(?accounts[a1].lock)
    var funds_available = amount <= accounts[a1].balance
    release(?accounts[a1].lock)
    if funds_available:
        acquire(?accounts[a1].lock)
        accounts[a1].balance -= amount
        release(?accounts[a1].lock)
        acquire(?accounts[a2].lock)
        accounts[a2].balance += amount
        release(?accounts[a2].lock)

spawn transfer(0, 1, 2)
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

```
def Lock() returns lock:
    lock = False

def acquire2(lk1, lk2):
    atomically when not (!lk1 or !lk2):
        !lk1 = !lk2 = True

def release(lk):
    atomically !lk = False

def Account(balance) returns account:
    account = { .lock: Lock(), .balance: balance }

accounts = [ Account(3), Account(7) ]

def transfer(a1, a2, amount):
    acquire2(?accounts[a1].lock, ?accounts[a2].lock)
    if amount <= accounts[a1].balance:
        accounts[a1].balance -= amount
        accounts[a2].balance += amount
    release(?accounts[a1].lock)
    release(?accounts[a2].lock)

spawn transfer(0, 1, 1)
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

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

4: Negate circular wait

```
from synch import Lock, acquire, release

def Account(balance) returns account:
    account = { .lock: Lock(), .balance: balance }

accounts = [ Account(3), Account(7) ]

def transfer(a1, a2, amount):
    acquire(?accounts[min(a1, a2)].lock)
    acquire(?accounts[max(a1, a2)].lock)
    if amount <= accounts[a1].balance:
        accounts[a1].balance -= amount
        accounts[a2].balance += amount
    release(?accounts[a1].lock)
    release(?accounts[a2].lock)

spawn transfer(0, 1, 1)
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
- 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
  - an unsafe state is not necessarily deadlocked, but deadlock is unavoidable
  - 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

```
from synch import Lock, acquire, release

def Account(balance) returns account:
    account = { .lock: Lock(), .balance: balance }

accounts = [ Account(3), Account(7), Account(0) ]

def transfer(a1, a2, amount):
    acquire(?accounts[a1].lock)
    if amount <= accounts[a1].balance:
        accounts[a1].balance -= amount
        acquire(?accounts[a2].lock)
        accounts[a2].balance += amount
        release(?accounts[a2].lock)
    release(?accounts[a1].lock)

spawn transfer(0, 1, 1)
spawn transfer(1, 0, 2)
```

How?

Deadlock Avoidance

```
from synch import Lock, acquire, release

def Account(balance) returns account:
    account = { .lock: Lock(), .balance: balance }

accounts = [ Account(3), Account(7), Account(0) ]

def transfer(a1, a2, amount):
    acquire(?accounts[a1].lock)
    if amount <= accounts[a1].balance:
        accounts[a1].balance -= amount
        acquire(?accounts[a2].lock)
        accounts[a2].balance += amount
        release(?accounts[a2].lock)
    release(?accounts[a1].lock)

spawn transfer(0, 1, 1)
spawn transfer(1, 0, 2)
```

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

Avoidance specified in Harmony

```
from synch import Lock, acquire, release

def Account(balance) returns account:
    account = { .lock: Lock(), .balance: balance }

active = {}
accounts = [ Account(3), Account(7) ]

def transfer(a1, a2, amount):
    atomically when ({ a1, a2 } & active) == {}:
        active |= { a1, a2 }

    acquire(?accounts[a1].lock)
    if amount <= accounts[a1].balance:
        accounts[a1].balance -= amount
        acquire(?accounts[a2].lock)
        accounts[a2].balance += amount
        release(?accounts[a2].lock)
    release(?accounts[a1].lock)

    atomically:
        active -= { a1, a2 }
```

keep track of which accounts are active

enforce no intersection with active transfers

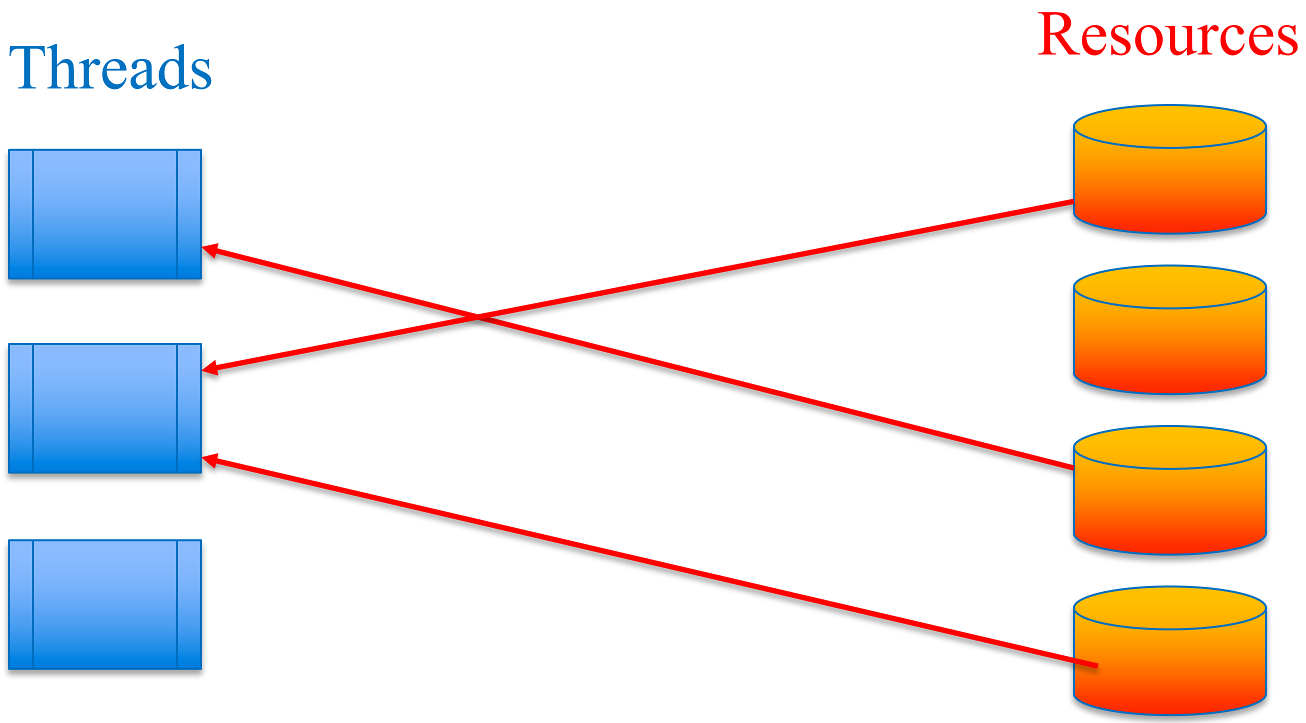
update scheduler state

Deadlock Detection and Recovery

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

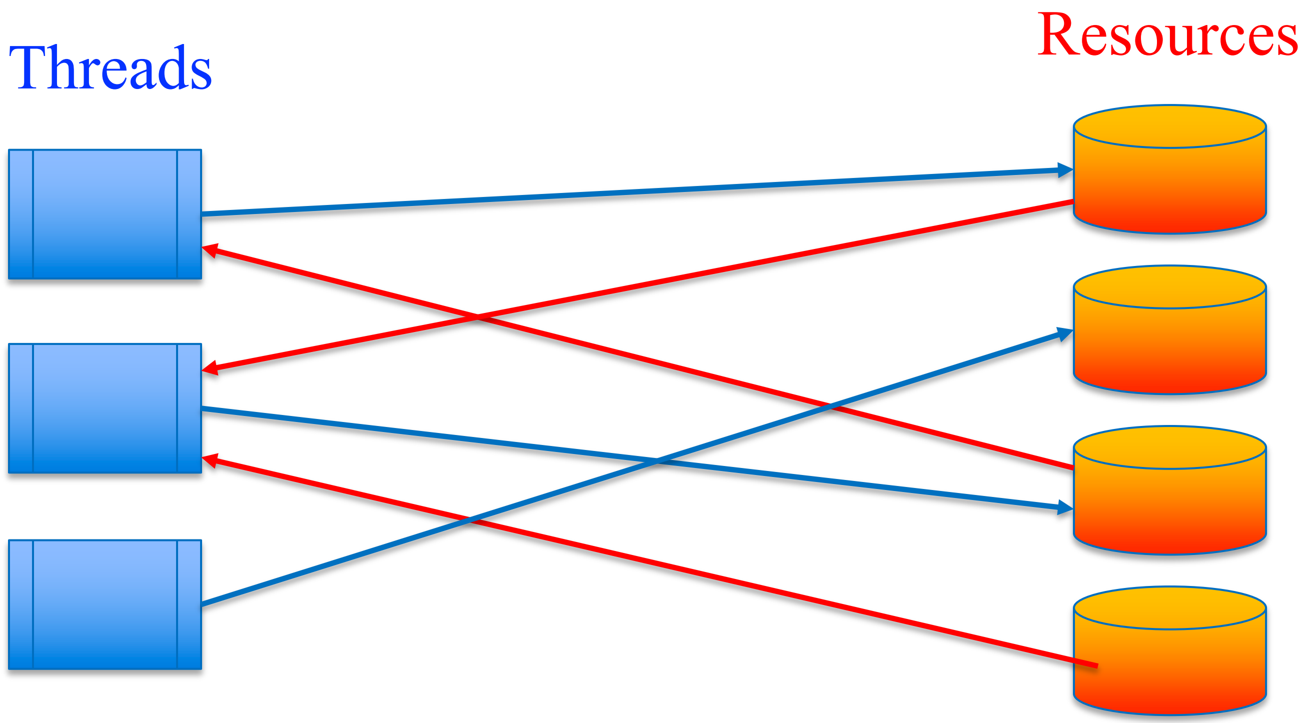
Deadlock Detection

- Keep track of allocation of resources to threads



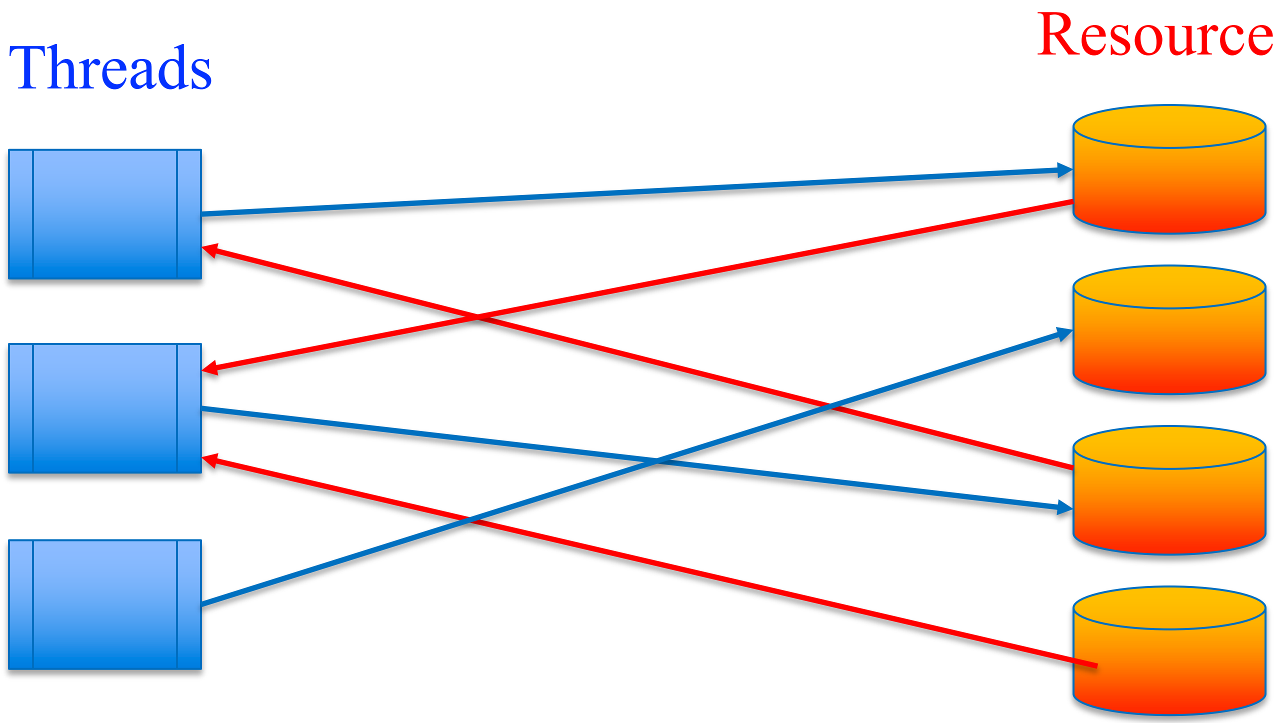
Deadlock Detection

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



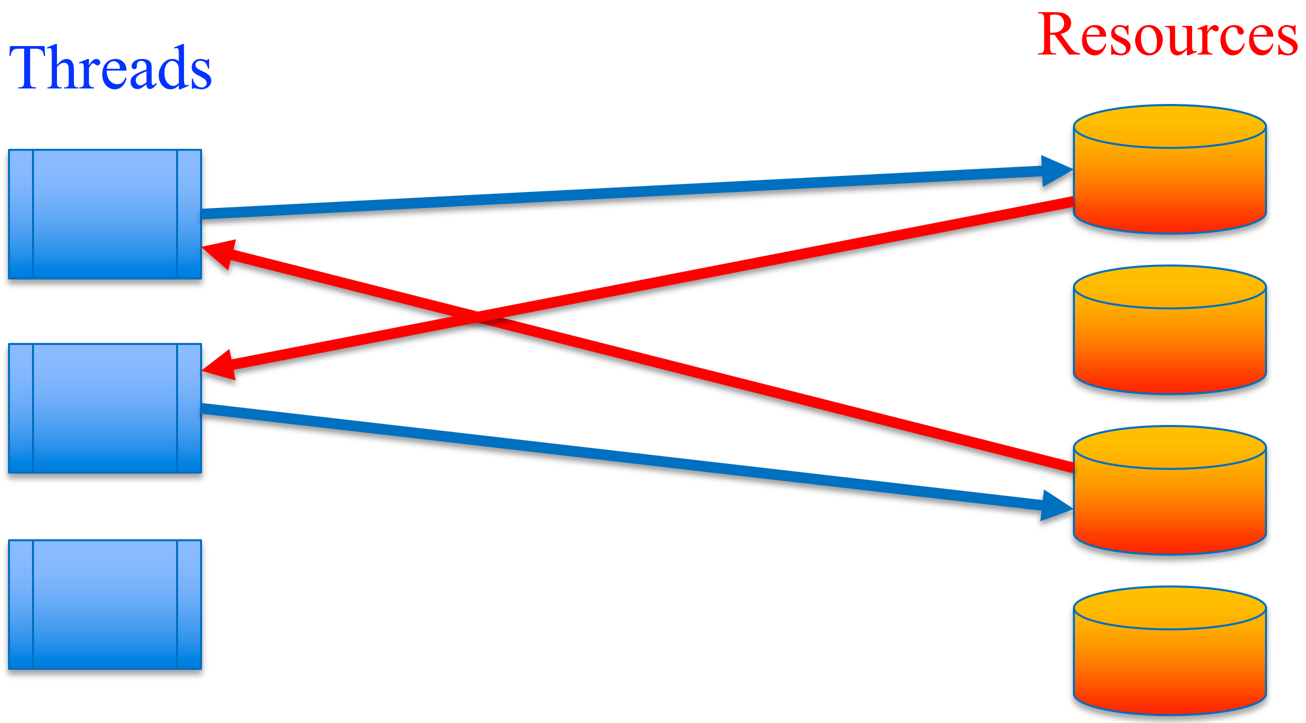
Deadlock Detection

- Known as the Resource Allocation Graph
- Deadlock ≡ cycle in the graph



Deadlock Detection

- Known as the Resource Allocation Graph
- Deadlock ≡ 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 graph empty (no nodes), then no cycles
  - No cycles $\Longrightarrow$ No deadlock

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 cyle
  - 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 University
Cornell CIS
COMPUTING AND INFORMATION SCIENCE

[Robbert van Renesse]

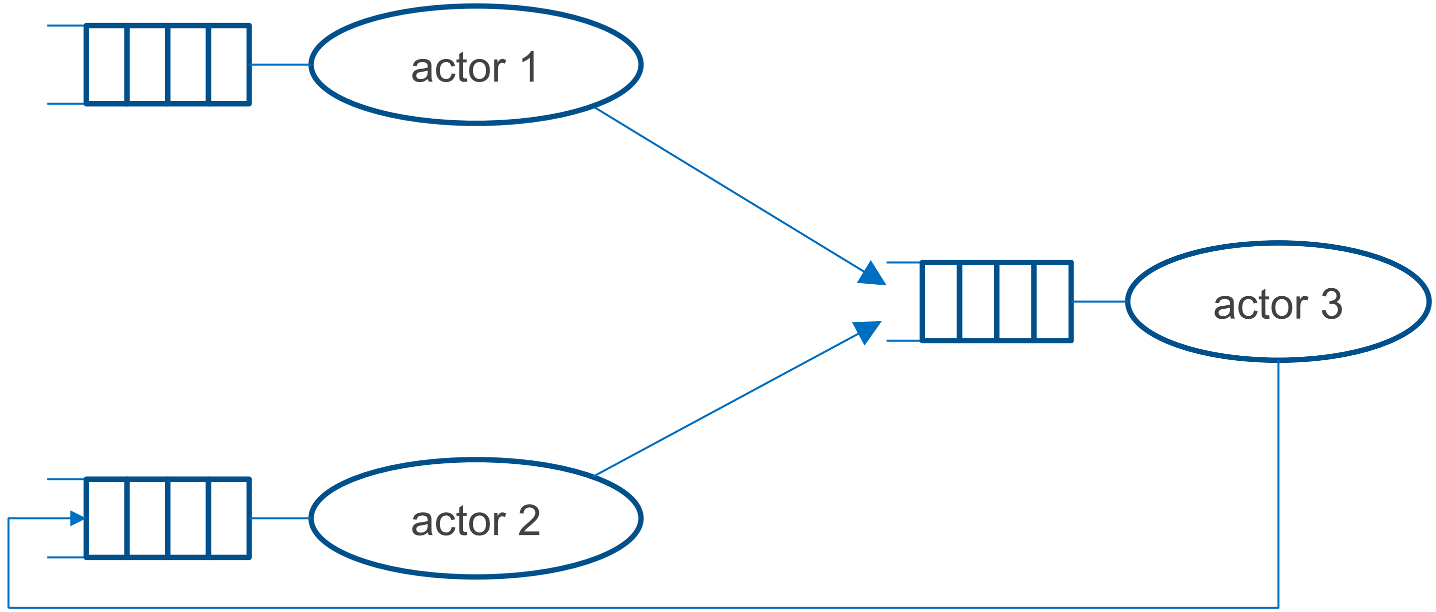
Actor Model

- 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)

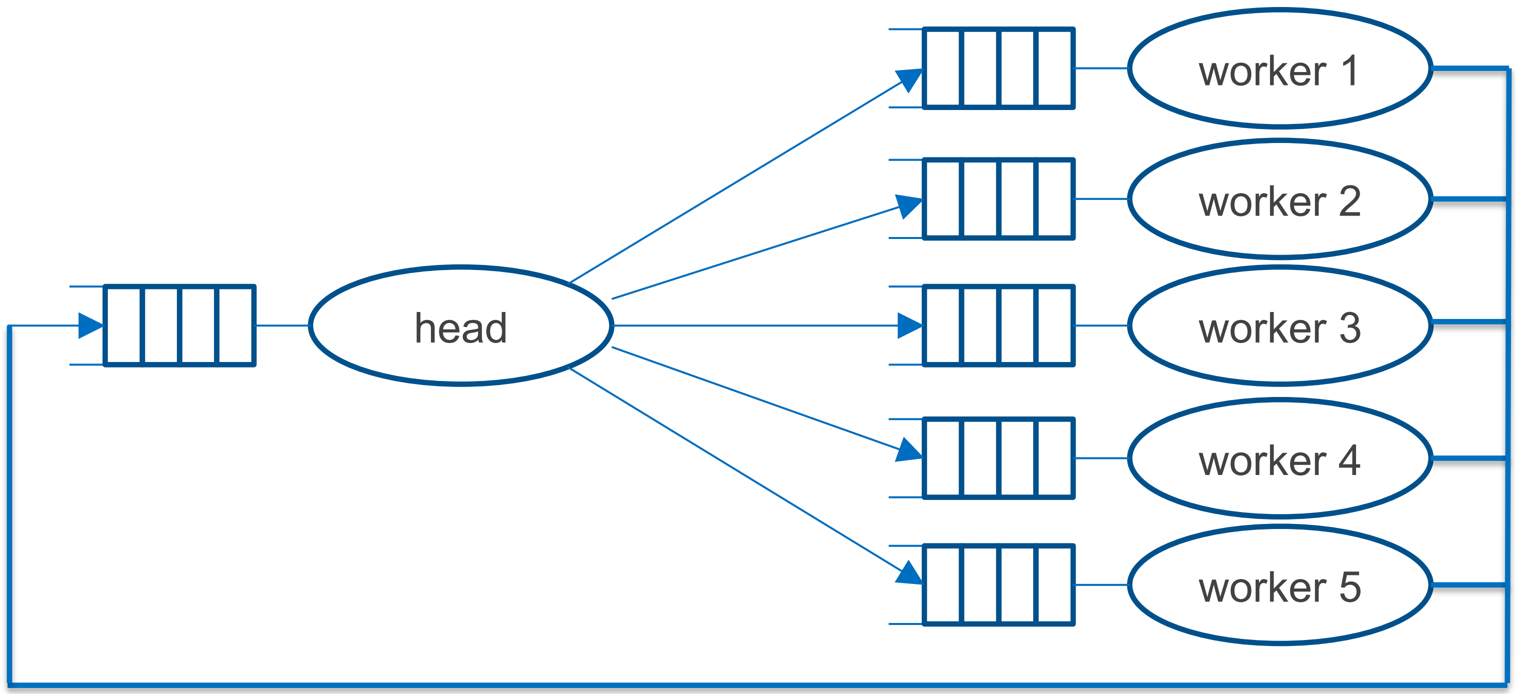


Conditional Critical Sections with Actors

- 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

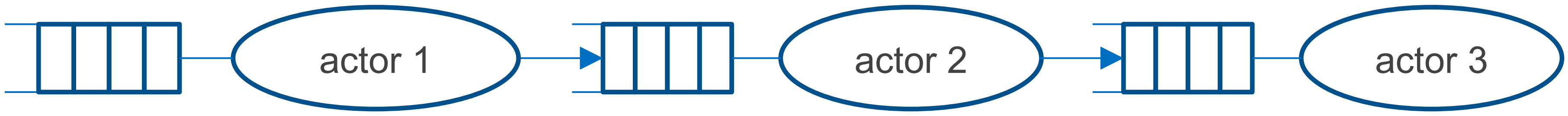


Parallel processing example

```
from synch import *

ranges = { (2,10), (11,20), (21,30) }
queues = { r:Queue() for r in ranges }
mainq = Queue()

def isPrime(v) returns prime:
    prime = True
    var d = 2
    while prime and (d < v):
        if (v % d) == 0:
            prime = False
        d += 1

def worker(q):
    while True:
        let rq, (start, finish) = get(q):
            for p in { start .. finish }:
                if isPrime(p):
                    put(rq, p)

def main(rq, workers):
    for r:q in workers:
        put(q, (rq, r))
    while True:
        print get(rq)

for r in ranges:
    spawn eternal worker(?queues[r])
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

```
from synch import *

const MAX = 10

def isPrime(v) returns prime:
    prime = True
    var d = 2
    while prime and (d < v):
        if (v % d) == 0:
            prime = False
        d += 1

q1 = q2 = q3 = Queue()

def actor0():
    for v in {2..MAX}:
        put(?q1, v)
```

```
def actor1():
    while True:
        let v = get(?q1):
            put(?q2, (2 ** v) - 1)

def actor2():
    while True:
        let v = get(?q2):
            if isPrime(v):
                put(?q3, v)

def actor3():
    while True:
        let v = get(?q3):
            print(v)

spawn actor0()
spawn eternal actor1()
spawn eternal actor2()
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

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

```
import barrier

const NWORKERS = 2

vec1 = [ 1, 2, 3, 4 ]
vec2 = [ 5, 6, 7, 8 ]
barr = barrier.Barrier(NWORKERS)
output = [ 0, ] * NWORKERS

def split(self, v) returns x:
    x = (self * len(v)) / NWORKERS

def dotproduct(self, v1, v2):
    assert len(v1) == len(v2)
    var total = 0
    for i in { split(self, v1) .. split(self + 1, v1)  - 1}:
        total += v1[i] * v2[i]
    output[self] = total
    barrier.bwait(?barr)
    print sum(output)

for i in { 0 .. NWORKERS - 1 }:
    spawn dotproduct(i, vec1, vec2)
```

Test program for barriers

```
import barrier

const NTHREADS = 3
const NROUNDS = 4

barr = barrier.Barrier(NTHREADS)
before = after = [0,] * NTHREADS

invariant min(before) >= max(after)

def thread(self):
    for _ in { 1 .. NROUNDS }:
        before[self] += 1
        barrier.bwait(?barr)
        after[self] += 1

for i in { 0 .. NTHREADS - 1 }:
    spawn thread(i)
```

work done before barrier

work done after barrier

Test program for barriers

```
import barrier

const NTHREADS = 3
const NROUNDS = 4

barr = barrier.Barrier(NTHREADS)
before = after = [0,] * NTHREADS

invariant min(before) >= max(after)

def thread(self):
    for _ in { 1 .. NROUNDS }:
        before[self] += 1
        barrier.bwait(?barr)
        after[self] += 1

for i in { 0 .. NTHREADS - 1 }:
    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

```
def Barrier(required) returns barrier:
    barrier = { .required: required, .n: 0 }

def bwait(b):
    atomically b->n += 1
    atomically await b->n == b->required
```

State:

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

Barrier Specification, Attempt 1

```
def Barrier(required) returns barrier:
    barrier = { .required: required, .n: 0 }

def bwait(b):
    atomically b->n += 1
    atomically await b->n == b->required
```

turnstile

State:

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

Barrier Specification, Attempt 1

```
def Barrier(required) returns barrier:
    barrier = { .required: required, .n: 0 }

def bwait(b):
    atomically b->n += 1
    atomically await b->n == b->required
```

turnstile

waiting area

State:

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

Barrier Specification, Attempt 1

```
def Barrier(required) returns barrier:
    barrier = { .required: required, .n: 0 }

def bwait(b):
    atomically b->n += 1
    atomically await b->n == b->required
```

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

Only works one round

Barrier Specification, Attempt 2

```
def Barrier(required) returns barrier:
    barrier = { .required: required, .n: 0 }

def bwait(b):
    atomically:
        b->n += 1
        if b->n == b->required:
            b->n = 0
    atomically await b->n == 0
```

Barrier Specification, Attempt 2

```
def Barrier(required) returns barrier:
    barrier = { .required: required, .n: 0 }

def bwait(b):
    atomically:
        b->n += 1
        if b->n == b->required:
            b->n = 0
    atomically await b->n == 0
```

Still only works once

Barrier Specification, Attempt 3

```
def Barrier(required) returns barrier:
    barrier = { .required: required, .n: [0, 0] }

def turnstile(b, i):
    atomically:
        b->n[i] += 1
        if b->n[i] == b->required:
            b->n[1 - i] = 0
    atomically await b->n[i] == b->required

def bwait(b):
    turnstile(b, 0)
    turnstile(b, 1)
```

Barrier Specification, Attempt 3

```
def Barrier(required) returns barrier:
    barrier = { .required: required, .n: [0, 0] }

def turnstile(b, i):
    atomically:
        b->n[i] += 1
        if b->n[i] == b->required:
            b->n[1 - i] = 0
    atomically await b->n[i] == b->required

def bwait(b):
    turnstile(b, 0)
    turnstile(b, 1)
```

Works, but double waiting is inefficient

Barrier Specification, final version

```
def Barrier(required) returns barrier:
    barrier = { .required: required, .n: 0, .color: 0 }

def bwait(b):
    var color = None
    atomically:
        color = b->color
        b->n += 1
        if b->n == b->required:
            b->color ^= 1
            b->n = 0
    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

```
from synch import *

def Barrier(required) returns barrier:
    barrier = {
        .mutex: Lock(), .cond: Condition(),
        .required: required, .n: 0, .color: 0
    }

def bwait(b):
    acquire(?b->mutex)
    b->n += 1
    if b->n == b->required:
        b->color ^= 1
        b->n = 0
        notify_all(?b->cond)
    else:
        let color = b->color:
            while b->color == color:
                wait(?b->cond, ?b->mutex)
    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 completed 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 fu[illegible]ty
  - Bus won't go until it is f[illegible]
- Resource must [illegible]eted emptied before it [illegible] used
  - [illegible] must get off at destination b[illegible]re 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

```
from synch import *

def RollerCoaster(nseats): result = {
    .mutex: Lock(), .nseats: nseats, .entered: 0, .left: nseats,
    .empty: Condition(), .full: Condition()
}

def enter(b):
    acquire(?b->mutex)
    while b->entered == b->nseats:  # wait for car to empty out
        wait(?b->empty, ?b->mutex)
    b->entered += 1
    if b->entered != b->nseats:     # wait for car to fill up
        while b->entered < b->nseats:
            wait(?b->full, ?b->mutex)
    else:                           # car is ready to go
        b->left = 0
        notify_all(?b->full)         # wake up others waiting in car
    release(?b->mutex)

def exit(b):
    acquire(?b->mutex)
    b->left += 1
    if b->left == b->nseats:    # car is empty
        b->entered = 0
        notify_all(?b->empty)    # wake up riders wanting to go
    release(?b->mutex)
```

JOE MCBRIDE / GETTY IMAGES

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

```
count = 0
done = False

finally count == 1

def handler():
    count += 1
    done = True

def main():
    trap handler()
    await done

spawn main()
```

check count == 1 in the final state

invoke handler() at some future time

Within the same thread!
(trap ≠ spawn)

But what now?

```
count = 0
done = False

finally count == 2

def handler():
    count += 1
    done = True

def main():
    trap handler()
    count += 1
    await done

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 T0: **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?

```
from synch import Lock, acquire, release

countlock = Lock()
count = 0
done = False

finally count == 2

def handler():
    acquire(?countlock)
    count += 1
    release(?countlock)
    done = True

def main():
    trap handler()
    acquire(?countlock)
    count += 1
    release(?countlock)
    await done

spawn main()
```

Locks to the rescue?

```
1  from synch import Lock, acq
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

```
count = 0
done = False

finally count == 2

def handler():
    count += 1
    done = True

def main():
    trap handler()
    setintlevel(True)
    count += 1
    setintlevel(False)
    await done

spawn main()
```

disable interrupts (→ `setintlevel(True)`)

enable interrupts (→ `setintlevel(False)`)

Interrupt-Safe Methods

```
count = 0
done = False

finally count == 2

def increment():
    let prior = setintlevel(True):
        count += 1
        setintlevel(prior)

def handler():
    increment()
    done = True

def main():
    trap handler()
    increment()
    await done

spawn main()
```

disable interrupts

restore old interrupt level

Interrupt-safe *AND* Thread-safe?

```
from synch import Lock, acquire, release

count = 0
countlock = Lock()
done = [ False, False ]

finally count == 4

def increment():
    let prior = setintlevel(True):
        acquire(?countlock)
        count += 1
        release(?countlock)
        setintlevel(prior)

def handler(self):
    increment()
    done[self] = True

def thread(self):
    trap handler(self)
    increment()
    await done[self]

spawn thread(0)
spawn thread(1)
```

Interrupt-safe *AND* Thread-safe?

```
from synch import Lock, acquire, release

count = 0
countlock = Lock()
done = [ False, False ]

finally count == 4

def increment():
    let prior = setintlevel(True):
        acquire(?countlock)
        count += 1
        release(?countlock)
        setintlevel(prior)

def handler(self):
    increment()
    done[self] = True

def thread(self):
    trap handler(self)
    increment()
    await done[self]

spawn thread(0)
spawn thread(1)
```

wait for own interrupt

Interrupt-safe *AND* Thread-safe?

```
from synch import Lock, acquire, release

count = 0
countlock = Lock()
done = [ False, False ]

finally count == 4

def increment():
    let prior = setintlevel(True):
        acquire(?countlock)
        count += 1
        release(?countlock)
        setintlevel(prior)

def handler(self):
    increment()
    done[self] = True

def thread(self):
    trap handler(self)
    increment()
    await done[self]

spawn thread(0)
spawn thread(1)
```

first disable interrupts

wait for own interrupt

Interrupt-safe *AND* Thread-safe?

```
from synch import Lock, acquire, release

count = 0
countlock = Lock()
done = [ False, False ]

finally count == 4

def increment():
    let prior = setintlevel(True):
        acquire(?countlock)
        count += 1
        release(?countlock)
        setintlevel(prior)

def handler(self):
    increment()
    done[self] = True

def thread(self):
    trap handler(self)
    increment()
    await done[self]

spawn thread(0)
spawn thread(1)
```

first disable interrupts

then acquire a lock

wait for own interrupt

Interrupt-safe *AND* Thread-safe?

```
from synch import Lock, acquire, release

count = 0
countlock = Lock()
done = [ False, False ]

finally count == 4

def increment():
    let prior = setintlevel(True):
        acquire(?countlock)
        count += 1
        release(?countlock)
        setintlevel(prior)

def handler(self):
    increment()
    done[self] = True

def thread(self):
    trap handler(self)
    increment()
    await done[self]

spawn thread(0)
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*