Concurrent Programming: Critical Sections and Locks

CS 4410
Operating Systems

[Robbert van Renesse]

An Operating System is a Concurrent Program

- The ”kernel contexts” of each of the processes share many data structures
  - ready queue, wait queues, file system cache, and much more
- Sharing is further complicated by interrupt handlers that also access those data structures

Synchronization Lectures Outline

- What is the problem?
  - no determinism, no atomicity
- What is the solution?
  - some form of locks
- How to implement locks?
  - there are multiple ways
- How to specify concurrent problems?
  - atomic operations
- How to construct correct concurrent code?
  - invariants
- How to test concurrent programs
  - comparing behaviors

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**

Non-Determinism

```
shared = True

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

f()
g()
```

(a) [code/prog1.hny] Sequential

```
shared = True

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

spawn f()
spawn g()
```

(b) [code/prog2.hny] Concurrent

Figure 3.1: A sequential and a concurrent program.

Non-Determinism

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

(a) [code/prog1.hny] Sequential

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

(b) [code/prog2.hny] Concurrent

Figure 3.1: A sequential and a concurrent program.

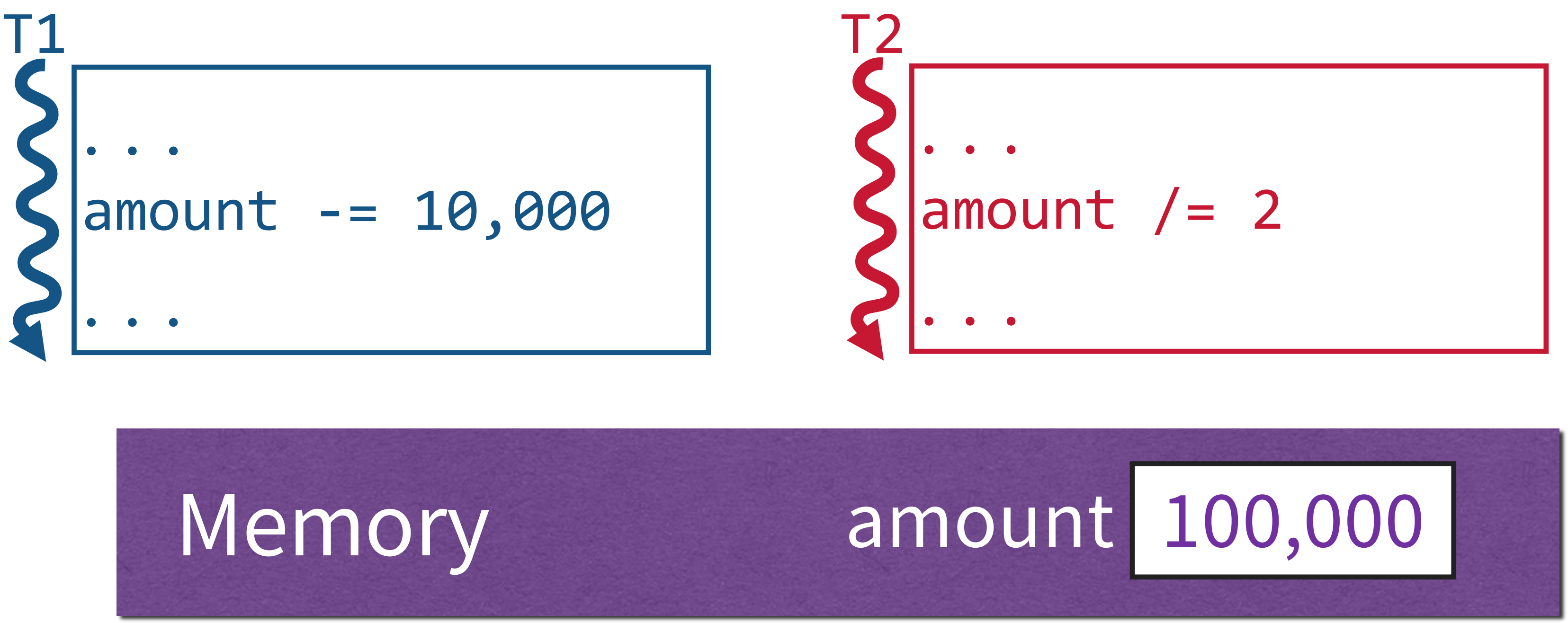
#states 2
2 components, 0 bad states
No issues

- 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

Non-Atomicity

2 threads updating a shared variable **amount**

- One thread (you) wants to decrement amount by $10K
- Other thread (IRS) wants to decrement amount by 50%

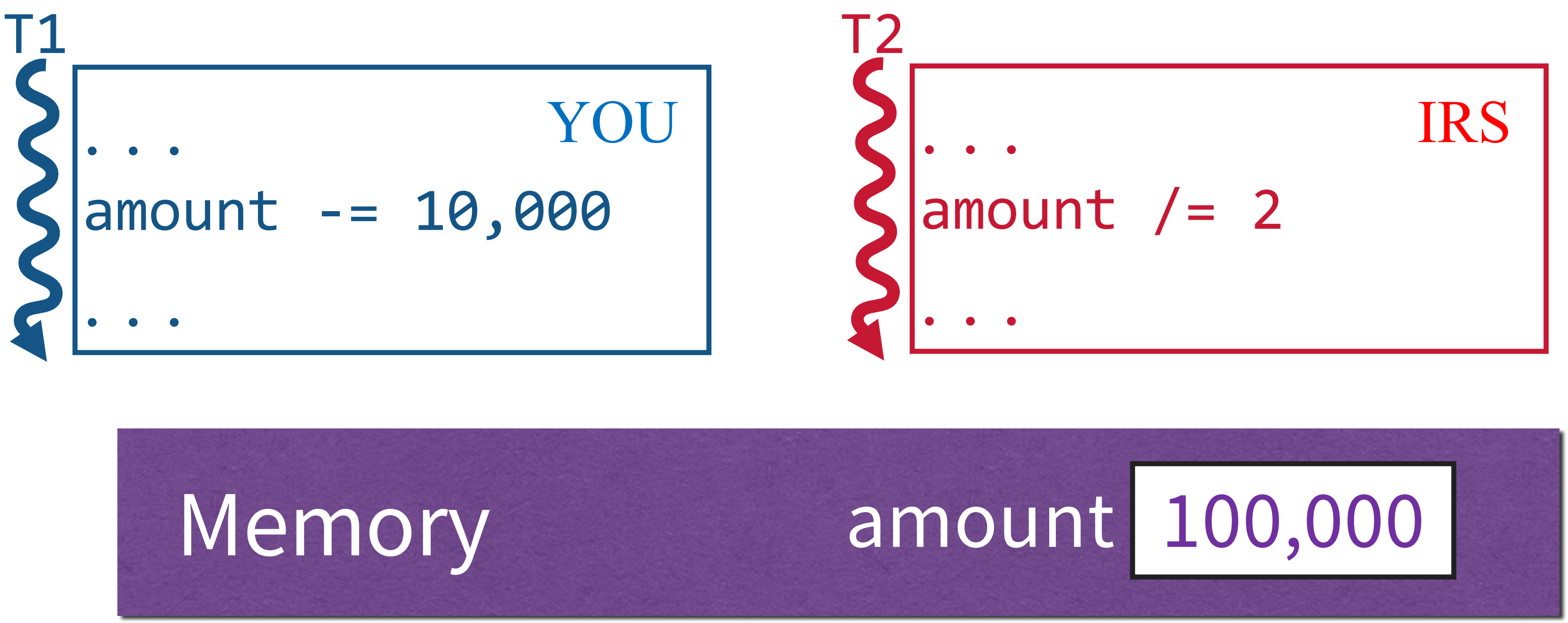


What happens when both threads are running?

Non-Atomicity

2 threads updating a shared variable **amount**

- One thread (you) wants to decrement amount by $10K
- Other thread (IRS) wants to decrement amount by 50%



What happens when both threads are running?

Non-Atomicity

Might execute like this:

T2
```
. . .
r2 = load from amount
r2 = r2 / 2
store r2 to amount
. . .
```

T1
```
. . .
r1 = load from amount
r1 = r1 - 10,000
store r1 to amount
. . .
```

Memory amount 40,000

Or vice versa (T1 then T2 → 45,000)…

either way is fine…

Non-Atomicity

Or it might execute like this:

T1
```
. . .
r1 = load from amount
r1 = r1 – 10,000
store r1 to amount
. . .
```

T2
```
. . .
r2 = load from amount

. . .

r2 = r2 / 2
store r2 to amount
. . .
```

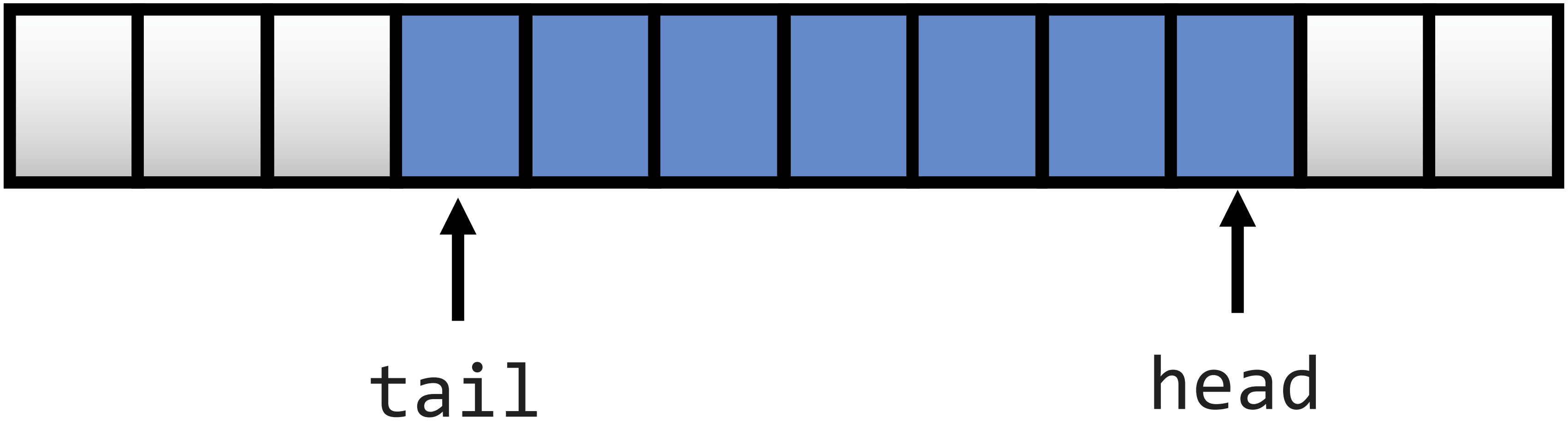
Memory amount 50,000

Lost Update!

Wrong ..and very difficult to debug

Example: Races with Shared Queue

- 2 concurrent enqueue() operations?
- 2 concurrent dequeue() operations?



What could possibly go wrong?

Race Conditions

= *timing dependent error involving shared state*

- Once thread A starts, it needs to “race” to finish
- Whether race condition happens depends on thread schedule
  - Different “schedules” or “interleavings” exist

    (a schedule is a total order on machine instructions)

All possible interleavings should be safe!

Race Conditions are Hard to Debug

- Number of possible interleavings is huge
- Some interleavings are good
- Some interleavings are bad
  - But bad interleavings may rarely happen!
  - Works 100x ≠ no race condition
- Timing dependent: small changes hide bugs
  - add print statement → bug no longer seems to happen

My experience until spring 2020

1. Students develop their code in Python or C
2. They test by running code many times
3. They submit their code, confident that it is correct
4. RVR tests the code with his secret and evil methods
   - uses homebrew library that randomly samples from possible interleavings (“fuzzing”)
5. Finds most submissions are broken
6. RVR unhappy, students unhappy

Why is that?

- Several studies show that heavily used code implemented, reviewed, and tested by expert programmers have lots of concurrency bugs
- Even professors who teach concurrency or write books and papers about concurrency get it wrong sometimes

Enter *Harmony*

- A new concurrent programming language
  - heavily based on Python syntax to reduce learning curve for many
- A new underlying virtual machine
  - quite different from any other:

it tries *all* possible executions of a program until it finds a problem, if any

(this is called "model checking")

Example (same as before)

```
def T1():
    amount −= 10000

def T2():
    amount /= 2

spawn T1()
spawn T2()
```

Harmony Machine Code

0 Jump 40

1 Frame T1 ()
2 Load amount — T1a: LOAD amount
3 Push 10000 — T1b: SUB 10000
4 2-ary –
5 Store amount — T1c: STORE amount
6 Return

7 Frame T2 () — T2a: LOAD amount
8 Load amount
9 Push 2 — T2b: DIV 2
10 2-ary /
11 Store amount — T2c: STORE amount
12 Return

```
def T1():
    amount −= 10000
```

```
def T2():
    amount /= 2
```

Harmony Virtual Machine *State*

Three parts:

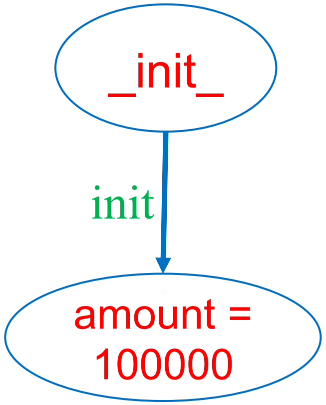
1. code (which never changes)
2. values of the shared variables
3. states of each of the running threads
   - "contexts"
     - PC, stack

State represents one vertex in the graph model

Simplified model

T1a: LOAD amount
T1b: SUB 10000
T1c: STORE amount

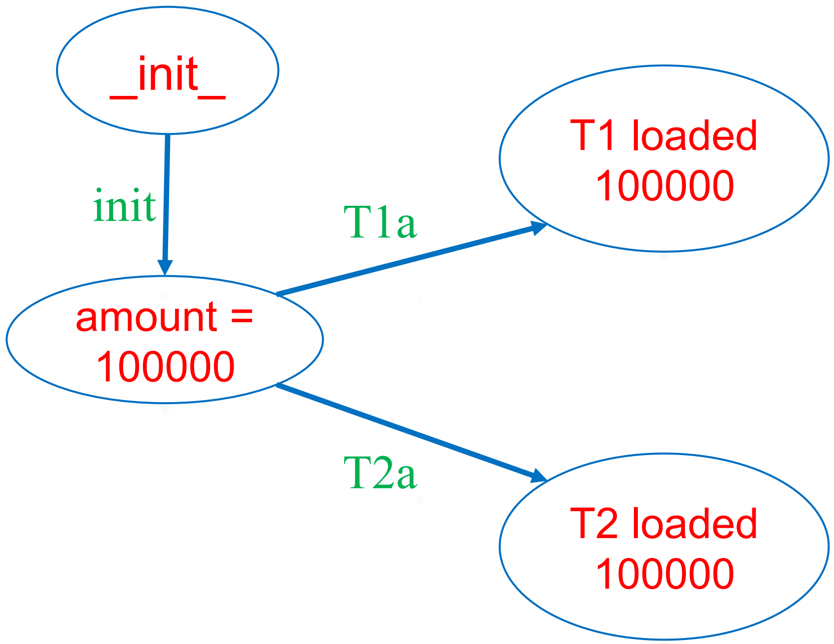
T2a: LOAD amount
T2b: DIV 2
T2c: STORE amount



Simplified model (ignoring main)

T1a: LOAD amount
T1b: SUB 10000
T1c: STORE amount

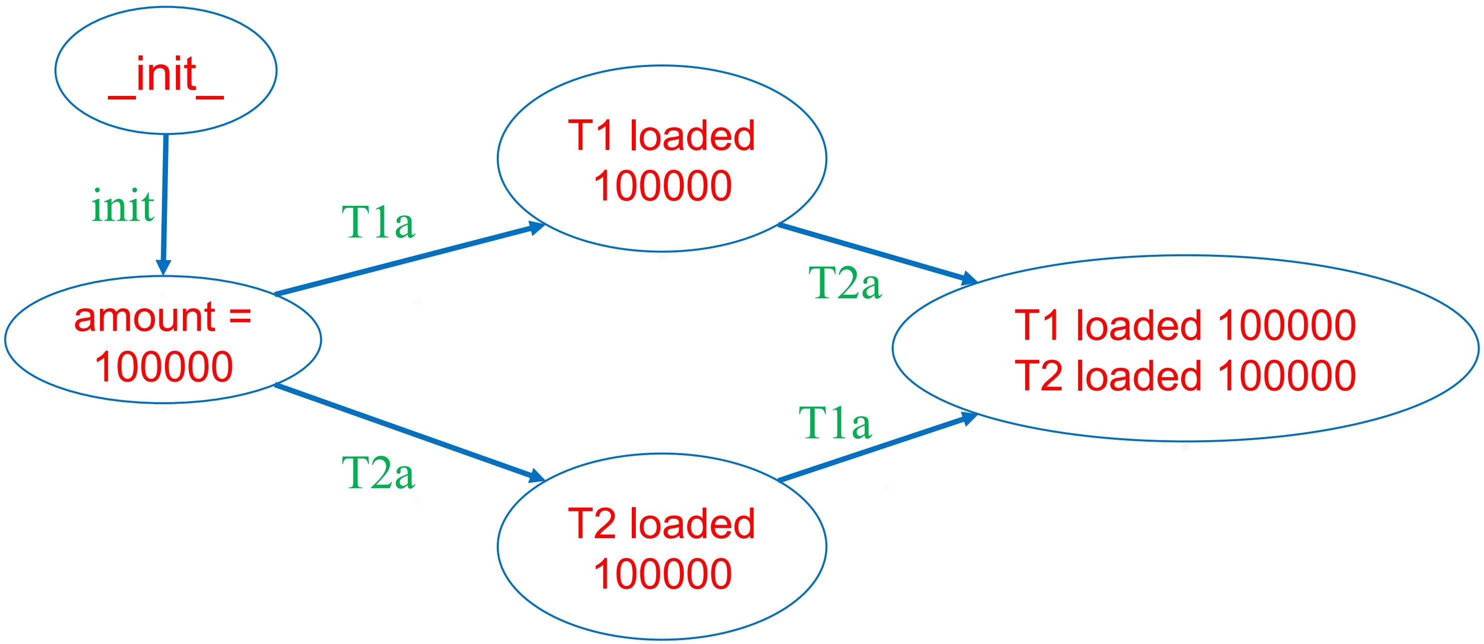
T2a: LOAD amount
T2b: DIV 2
T2c: STORE amount



Simplified model (ignoring main)

T1a: LOAD amount
T1b: SUB 10000
T1c: STORE amount

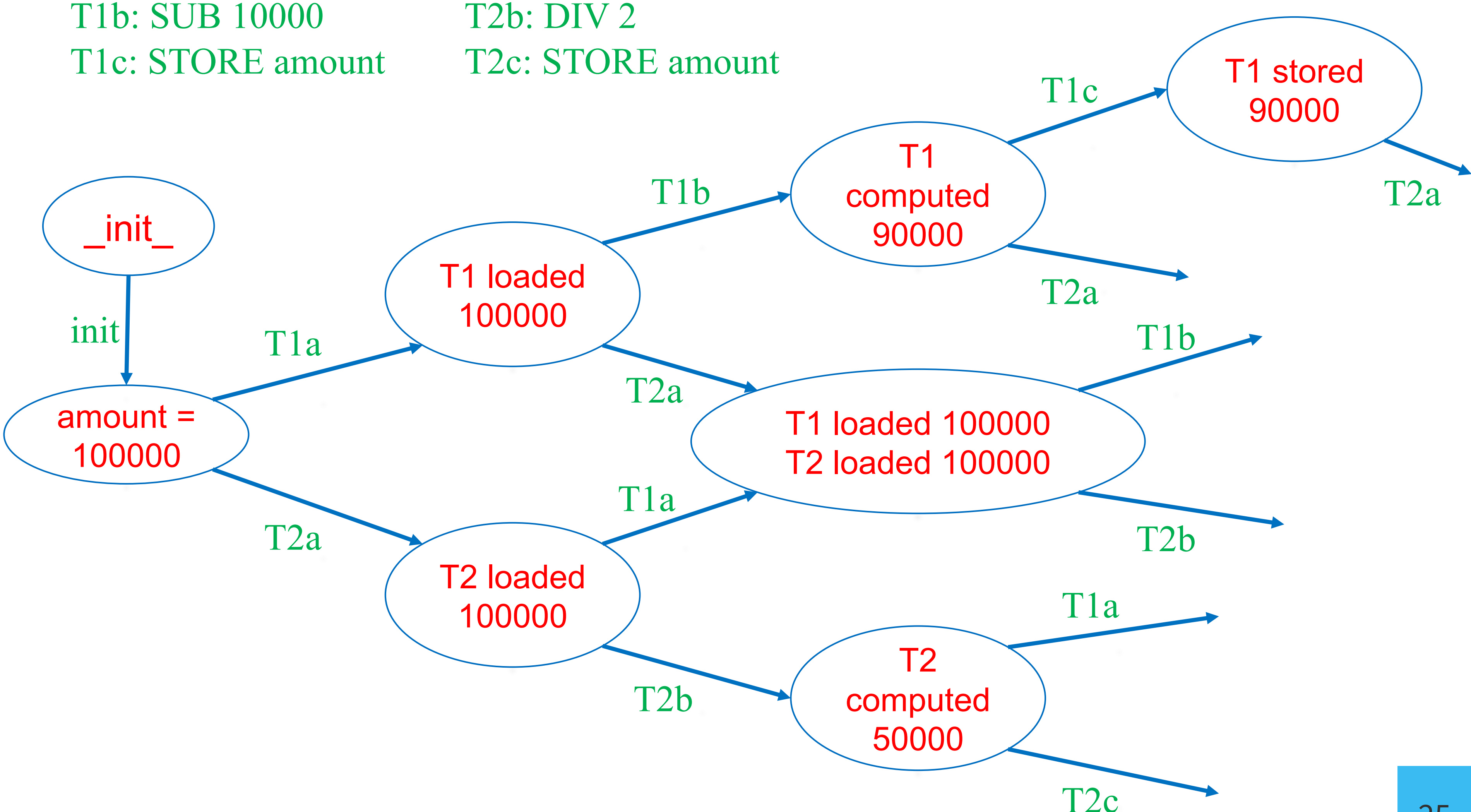
T2a: LOAD amount
T2b: DIV 2
T2c: STORE amount



Simplified model (ignoring main)

T1a: LOAD amount
T1b: SUB 10000
T1c: STORE amount

T2a: LOAD amount
T2b: DIV 2
T2c: STORE amount



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)
{ } is empty set	{ } is empty dictionary
few operator precedence rules --- use parentheses often	many operator precedence rules
variables global unless declared otherwise	depends... Sometimes must be explicitly declared global
no **return**, **break**, **continue**	various flow control escapes
no classes	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)

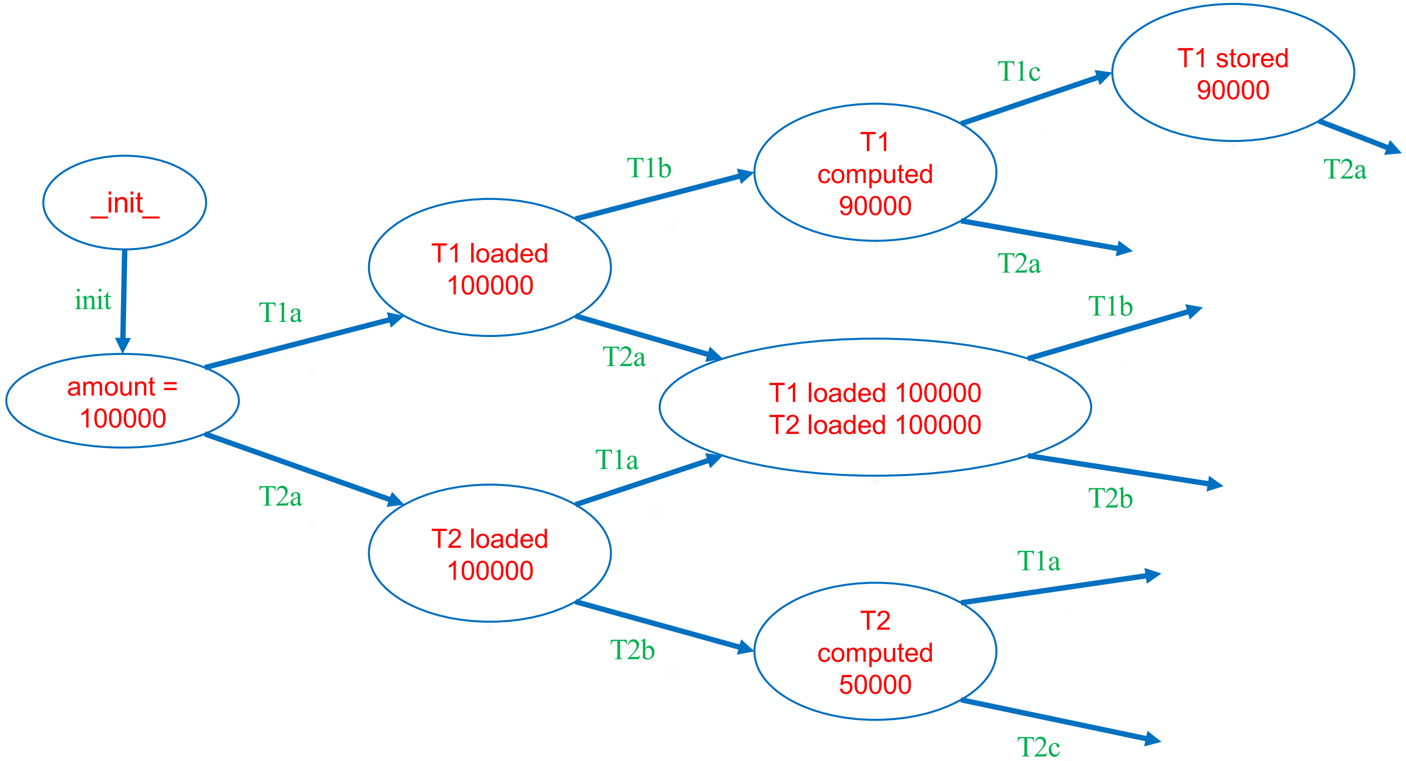
No open(), read(), or
or input() statements

Non-determinism in Harmony

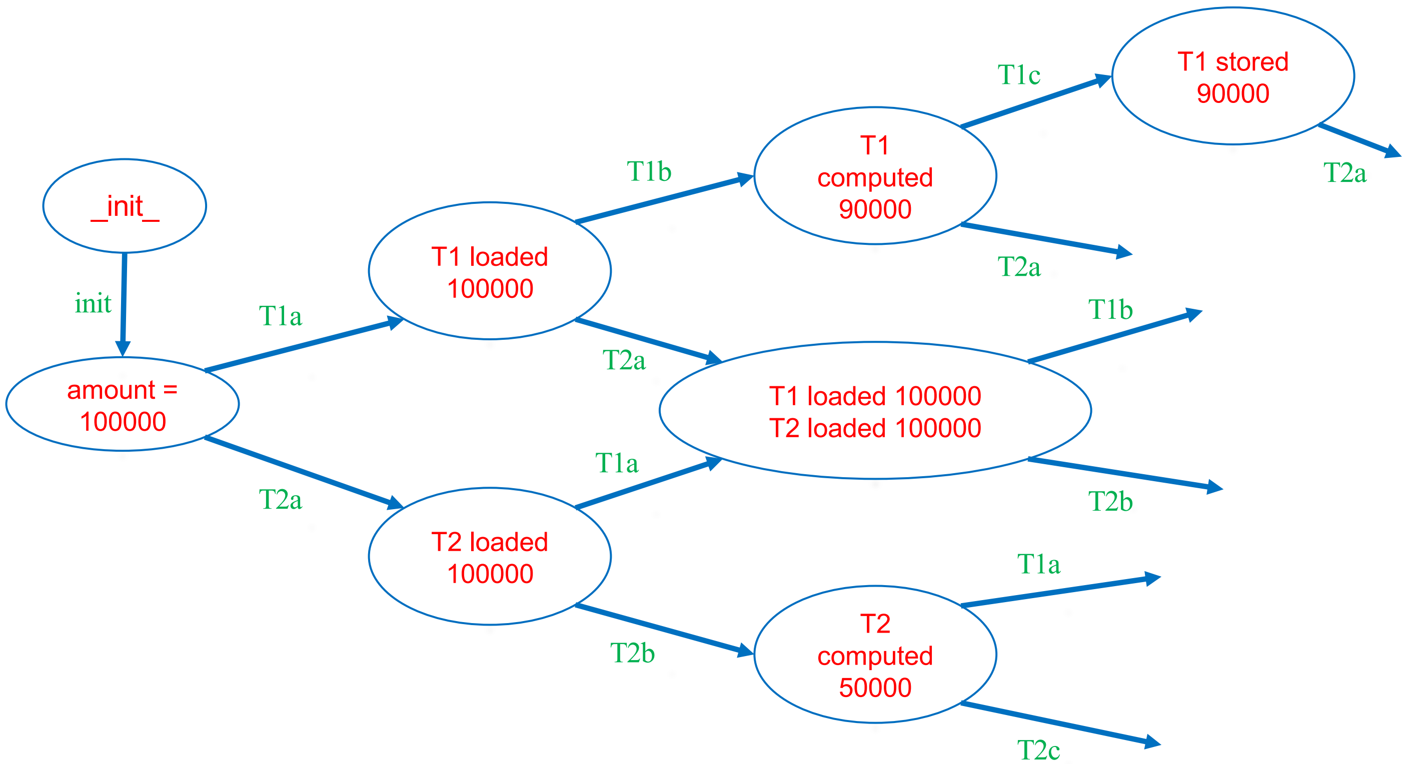
Three sources:

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

Limitation: models must be finite!



Limitation: models must be finite!

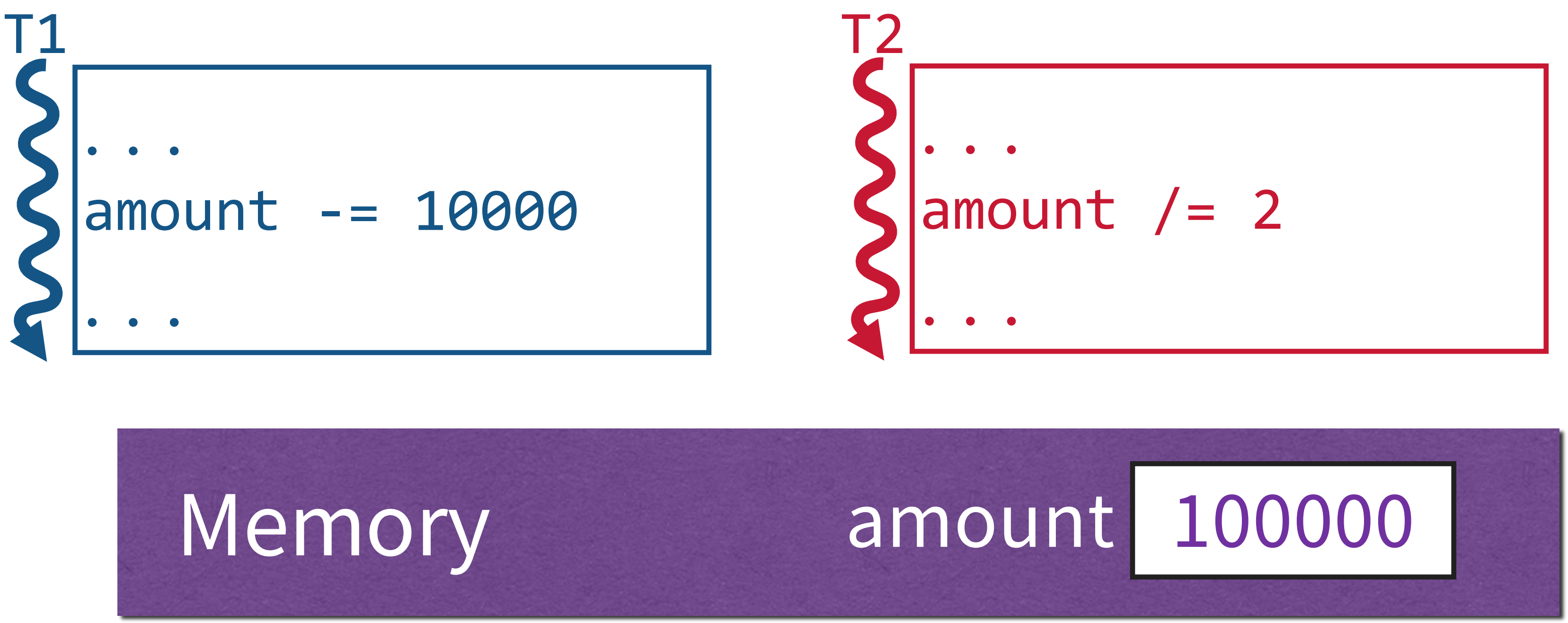


- But models are allowed to have cycles.
- Executions are allowed to be unbounded!
- Harmony checks for *possibility* of termination

Back to our problem…

2 threads updating a shared variable **amount**

- One thread wants to decrement amount by $10K
- Other thread wants to decrement amount by 50%



How to “serialize” these executions?

Critical Section

Must be serialized due to shared memory access

T1
```
. . .
CSEnter()
  amount -= 10000
CSExit()
. . .
```

T2
```
. . .
CSEnter()
  amount /= 2
CSExit()
. . .
```

<u>Goals</u>

Mutual Exclusion: 1 thread in a critical section at time

Progress: a thread can get in when there’s no other thread

Fairness: equal chances of getting into CS

… in practice, fairness rarely guaranteed or needed

Critical Section

Must be serialized due to shared memory access

T1

```
. . .
CSEnter()
  Critical section
CSExit()
. . .
```

T2

```
. . .
CSEnter()
  Critical section
CSExit()
. . .
```

Goals

Mutual Exclusion: 1 thread in a critical section at time

Progress: a thread can get in when there's 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

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

Progress: Harmony checks that all thread *can* terminate

Specification vs implementation

- Spec is fine, but this is an O.S. class!
- Sounds like we need a lock
- The question is:

How does one build a lock?

- Harmony is a concurrent programming language. *Really, doesn't Harmony have locks?*

You have to build them too!

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)
```

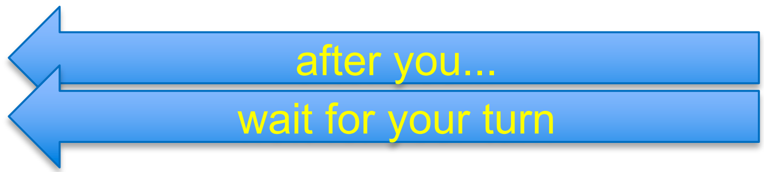
Third attempt: *turn* variable

```
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)
```



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

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)

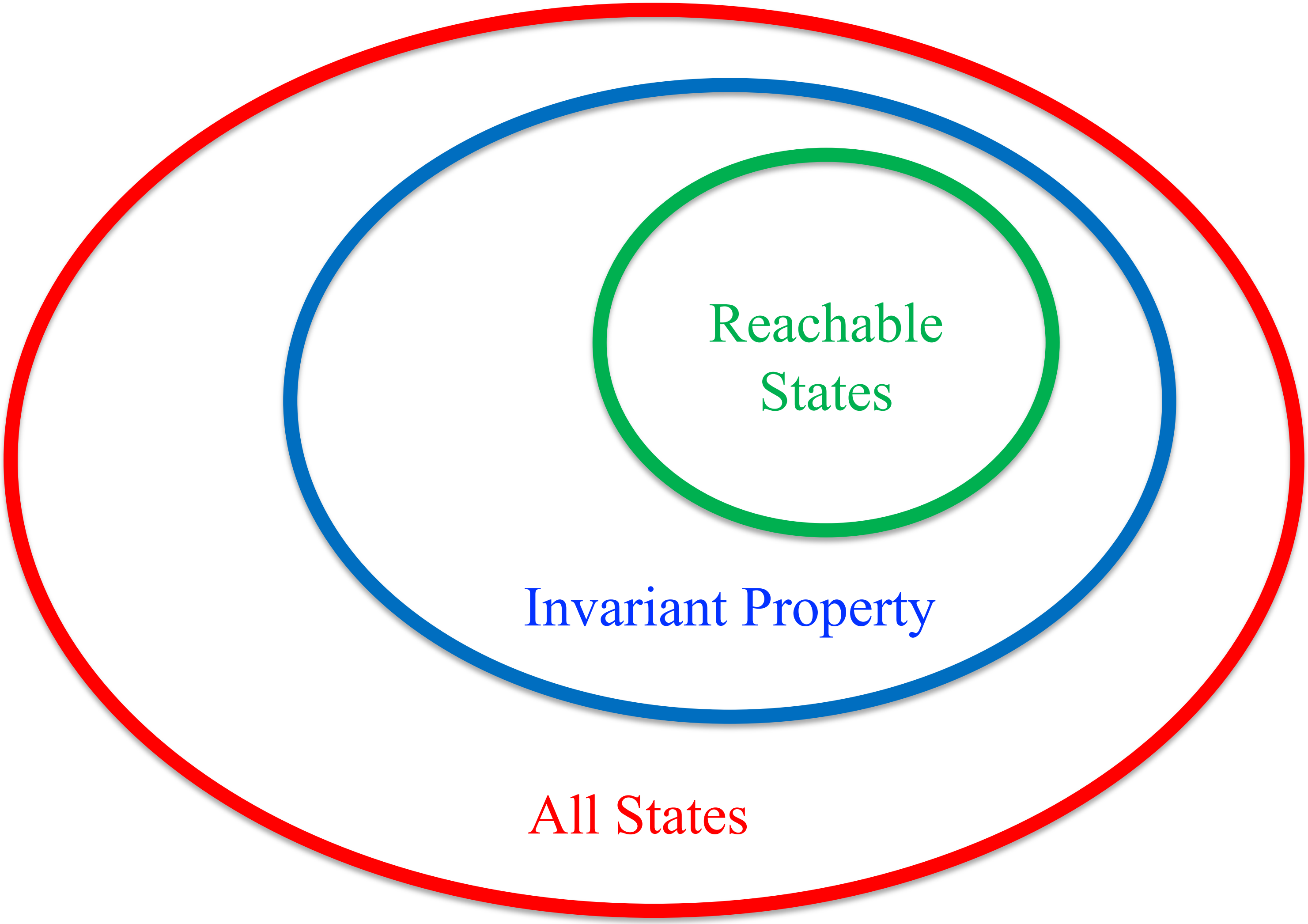
What is a property?

A property is a set of states

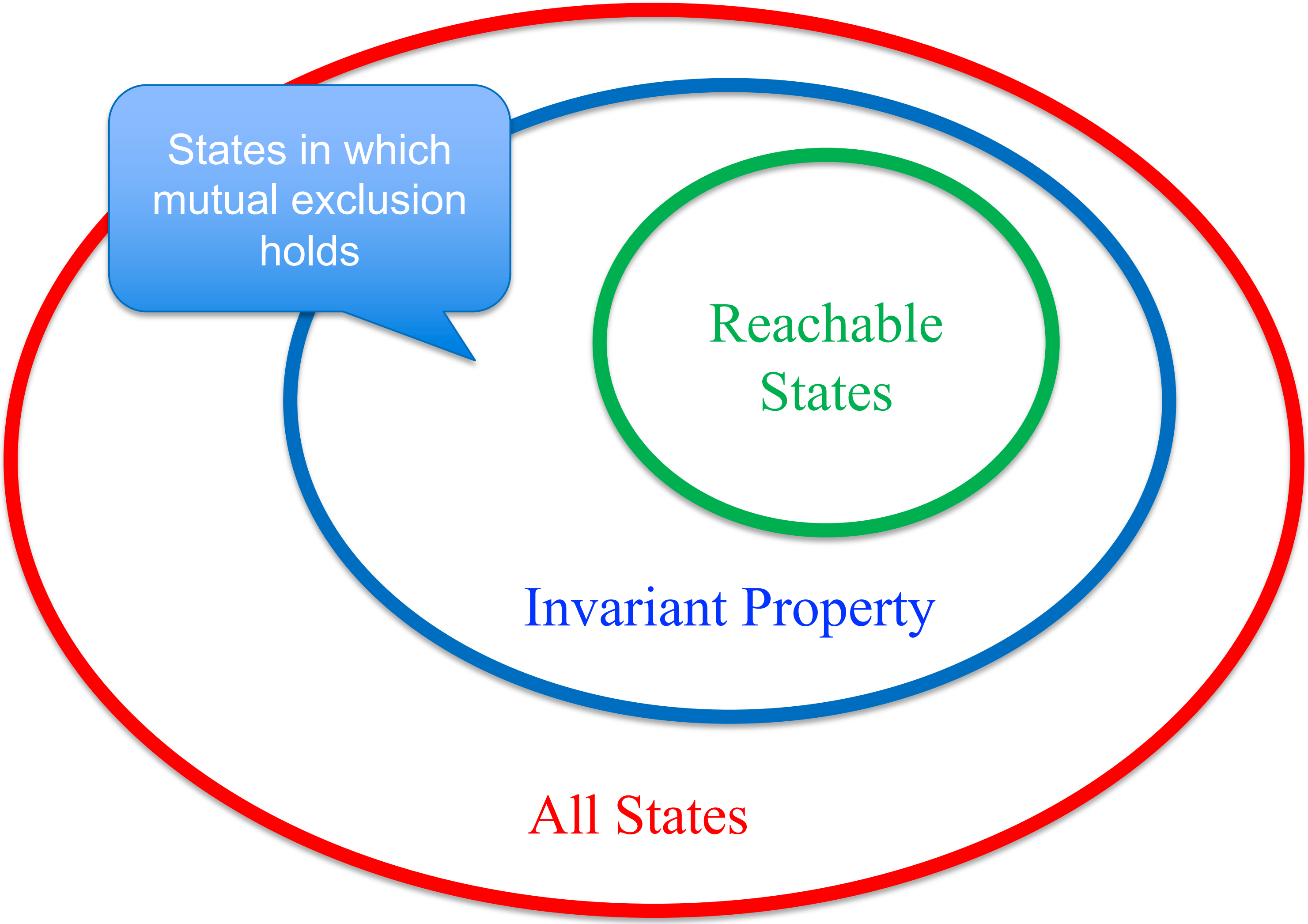
often succinctly described using a predicate

(all states that satisfy the predicate and no others)

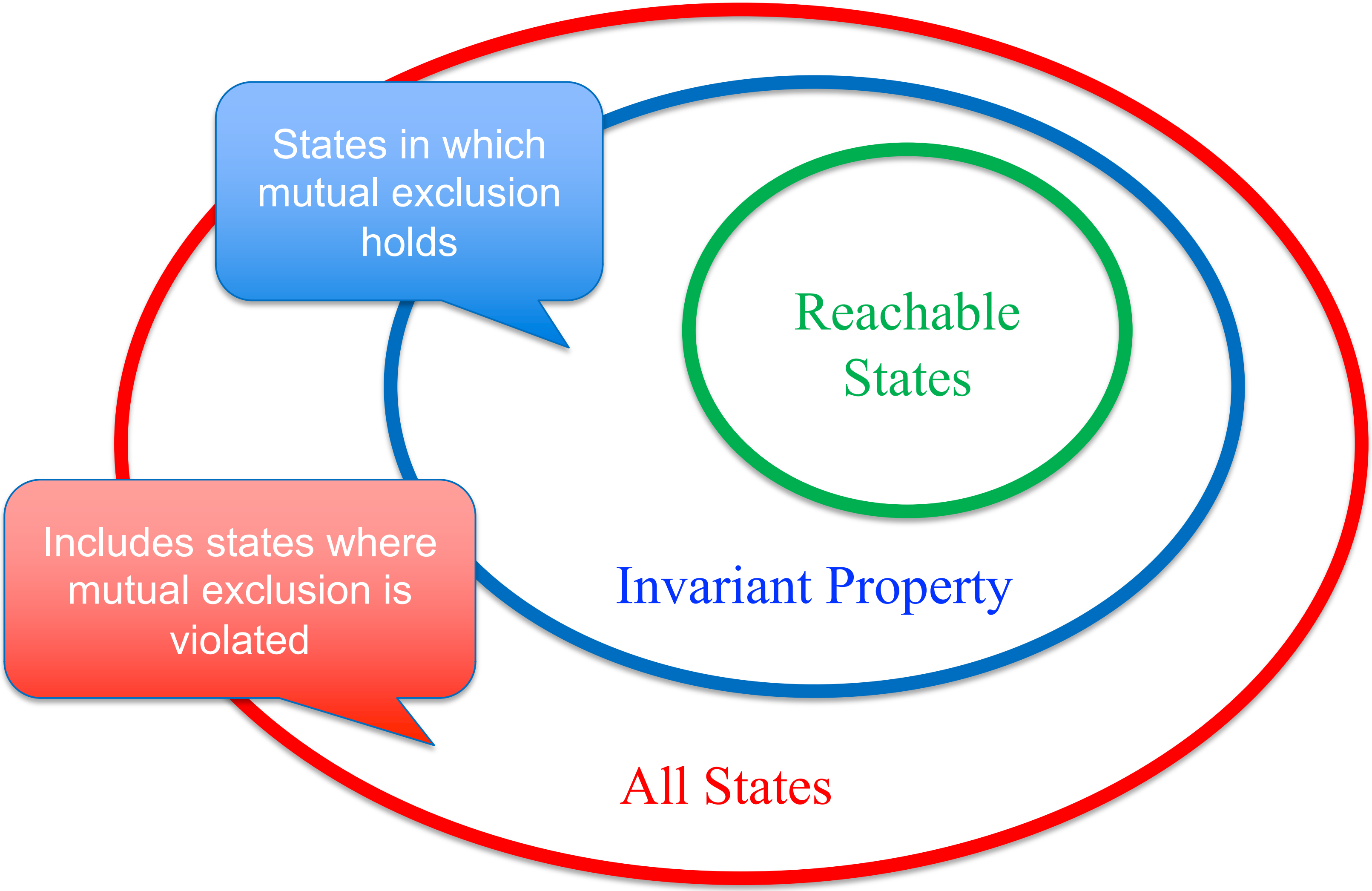
Invariant Property



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]

But there's a problem

- It turns out that mutual exclusion is hard to prove directly
  - it's hard to show that, if mutual exclusion holds in a state, it will also hold in the next state after making one execution step
  - not a good basis for induction
- Need a stronger invariant that implies mutual exclusion

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)
```

Candidate invariant to prove

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

Non-atomic load/store example

- Suppose *x* is a 64-bit integer
- Suppose you have a 32-bit CPU
- Then ”*x* = 0” requires 2 stores
  - because x occupies 2 words
- Similarly, reading *x* requires 2 loads
- Same is true is *x* is a 32-bit integer but *x* is not aligned on a word boundary
  - Writing to *x* would require two LOAD and two STORE operations on memory!

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 threads access the same variable
- And at least one is a STORE
- Then the semantics of the outcome is *undefined*

Harmony "sequential" statement

- **sequential** *turn*, *flags*
- 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* ;
- 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

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

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

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

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*

```
import lock

const NTHREADS = 5

in_cs = 0
invariant in_cs in { 0, 1 }

thelock = lock.Lock()

def thread():
    while choose({ False, True }):
        lock.acquire(?thelock)

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

        lock.release(?thelock)

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

“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

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

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

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()
```

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 regions of the code
- Beware of code with multiple critical sections
  - e.g., code that uses multiple locks

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 ⟹ data structure consistent
- *Or equivalently*:
  - data structure inconsistent ⟹ 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]
```

(a) [code/queue_nonatom.hny] 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]
```

(b) [code/queue.hny] 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

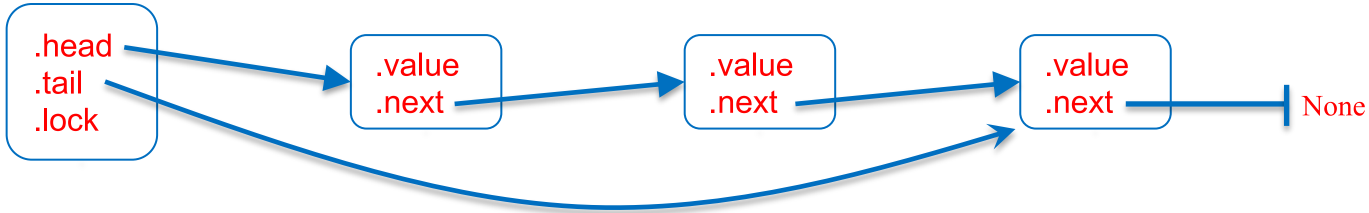
Specifying a concurrent queue

```
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

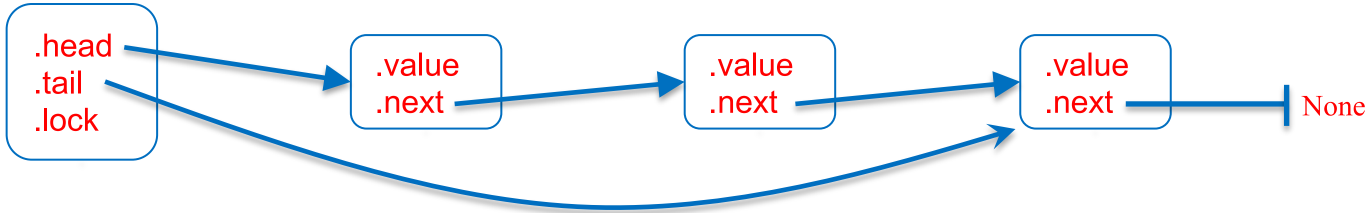
- *operations are O(n)*
- *code uses* ***atomically***

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)
```

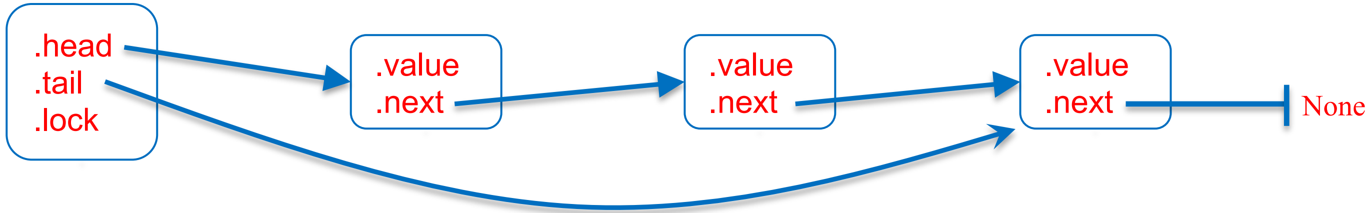
Queue implementation, v1




```
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)
```

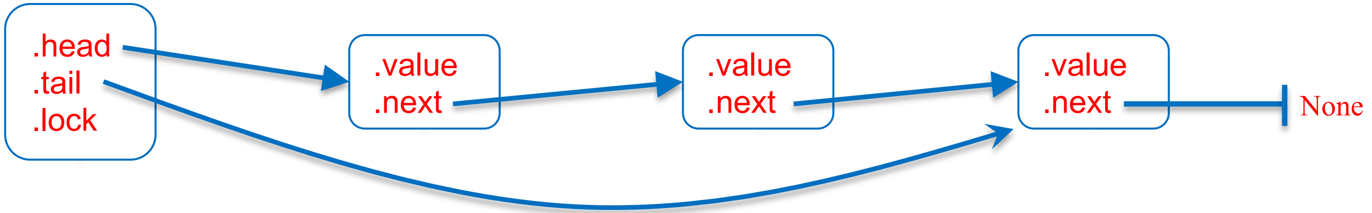
dynamic memory allocation

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)
```

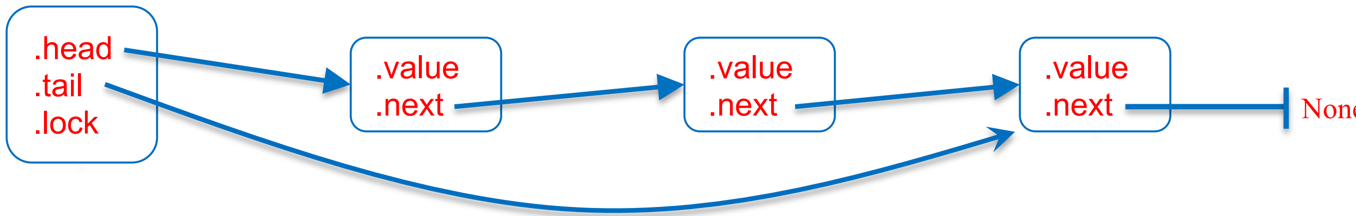
Queue implementation, v1




```
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

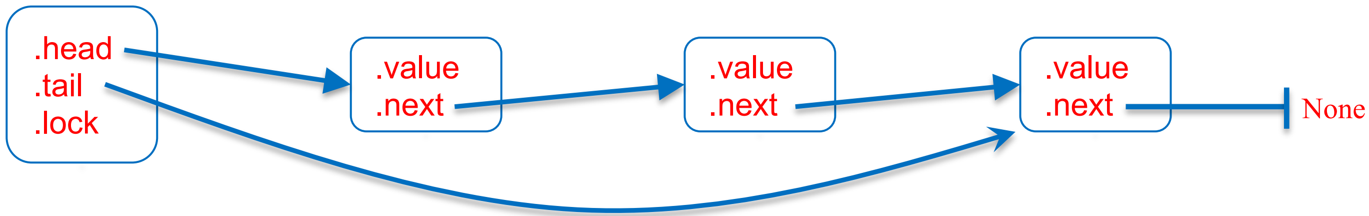
Queue implementation, v1




```
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

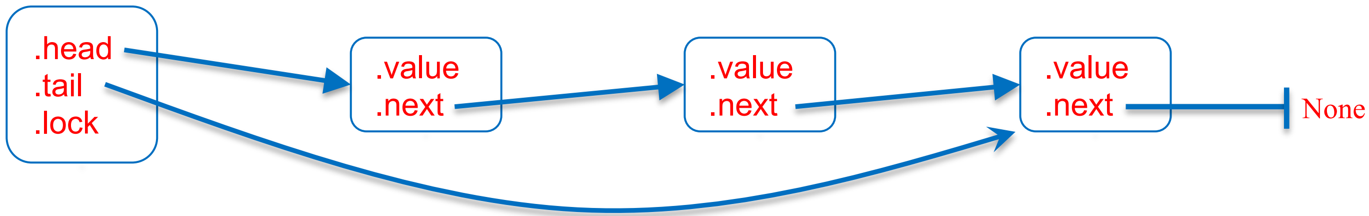
Queue implementation, v1




```
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

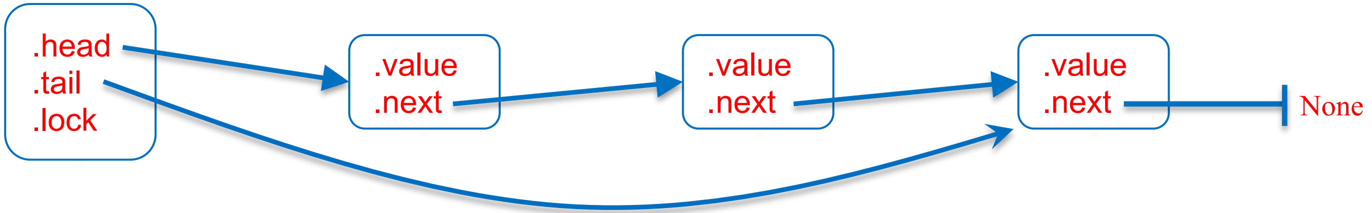
Queue implementation, v1




```
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)
```

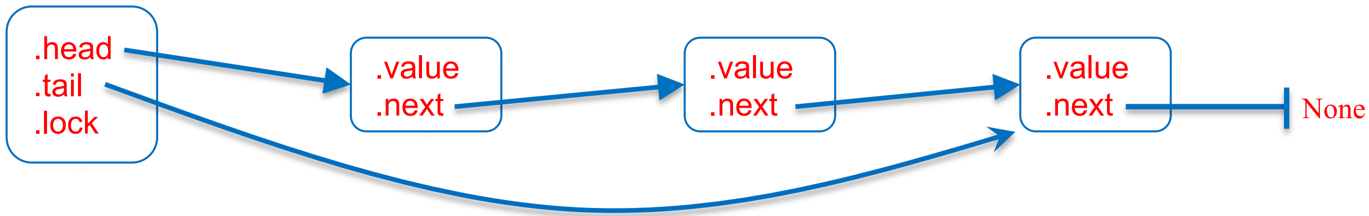
release lock

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)
```

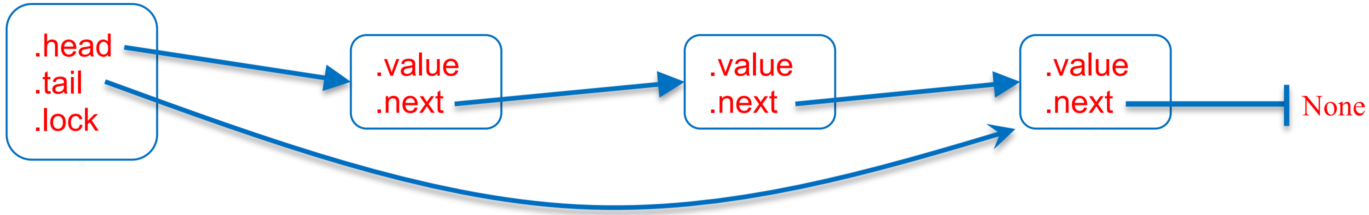
Queue implementation, v1




```
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)

How important are concurrent queues?

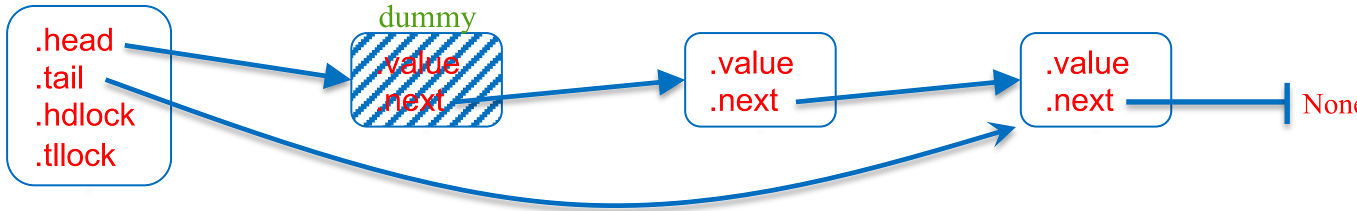
- Answer: all important
  - any resource that needs scheduling
    - CPU run queue
    - disk, network, printer waiting queue
    - lock waiting queue
  - inter-process communication
    - Posix pipes:
      - cat file | tr a-z A-Z | grep RVR
  - actor-based concurrency
  - …

How important are concurrent queues?

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

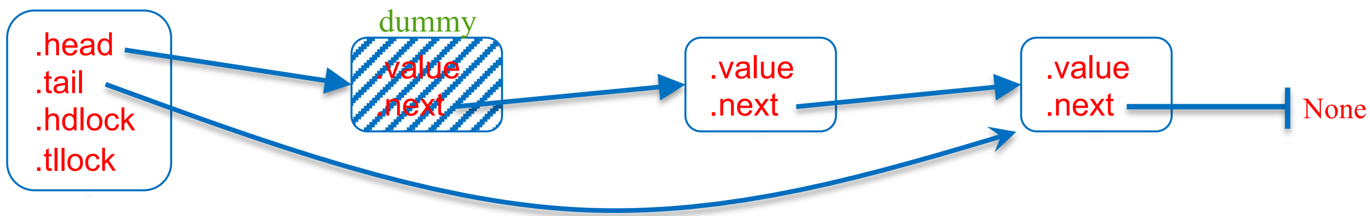
Good performance is critical!

Concurrent queue v2: 2 locks




```
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)
```

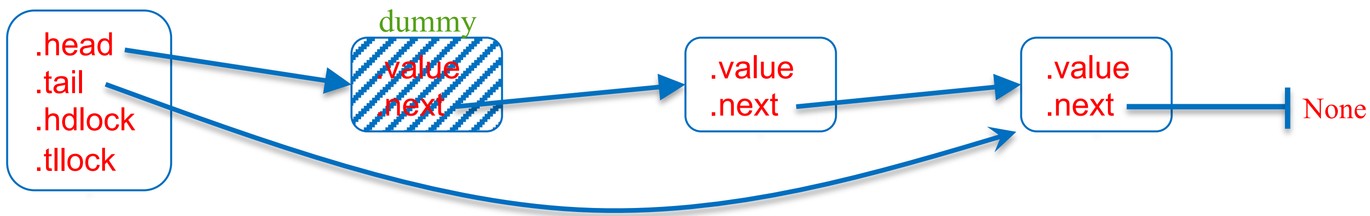
Concurrent queue v2: 2 locks




```
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)
```

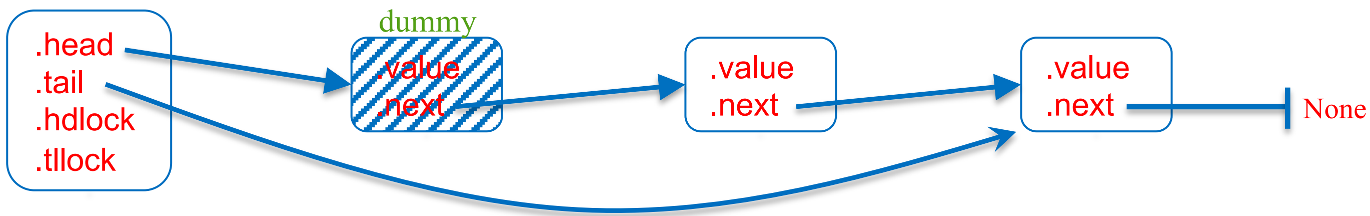
atomically q->tail->next = node

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)
```

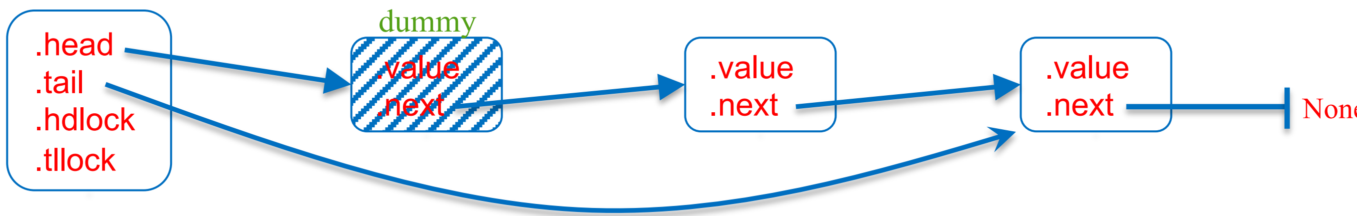
Concurrent queue v2: 2 locks




```
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)
```

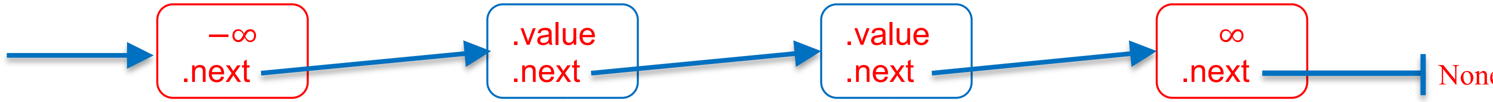
No contention for concurrent enqueue and dequeue operations!
➔ more concurrency ➔ faster

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

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



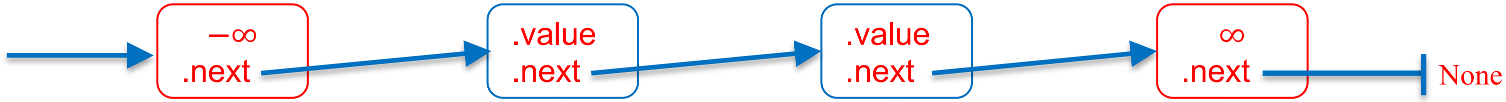

```
from synch import Lock, acquire, release
from alloc import malloc, free

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

def _find(lst, v):
    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)
    result = (before, after)

def SetObject():
    result = _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

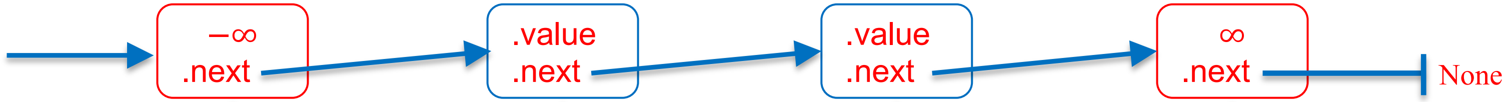
Sorted Linked List with Lock per Node




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

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

def _find(lst, v):
    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)
    result = (before, after)

def SetObject():
    result = _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): # allocate and initialize a new list node
    result = malloc({ .lock: Lock(), .value: v, .next: n })

def _find(lst, v):
    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)
    result = (before, after)

def SetObject():
    result = _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
            release(?after→lock)
            free(after)
        else:
            release(?after→lock)
        release(?before→lock)

def contains(lst, v):
    let before, after = _find(lst, v):
        result = 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
            release(?after→lock)
            free(after)
        else:
            release(?after→lock)
        release(?before→lock)

def contains(lst, v):
    let before, after = _find(lst, v):
        result = 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

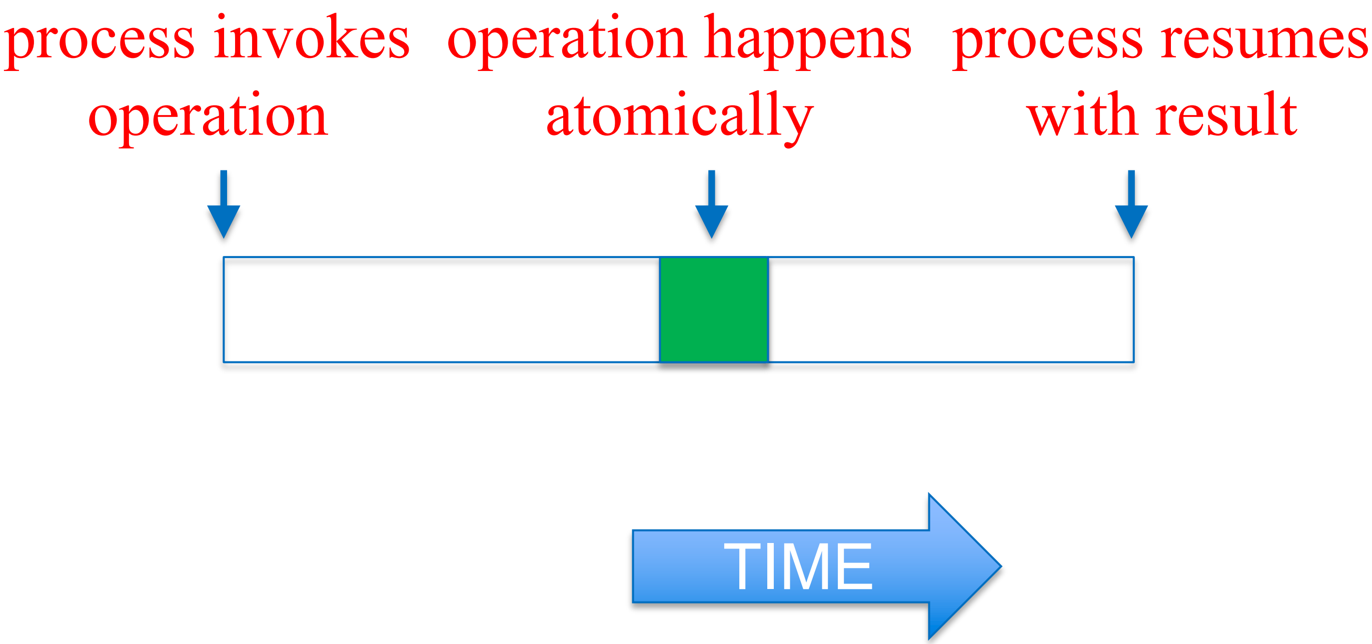
Systematic Testing

- 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!

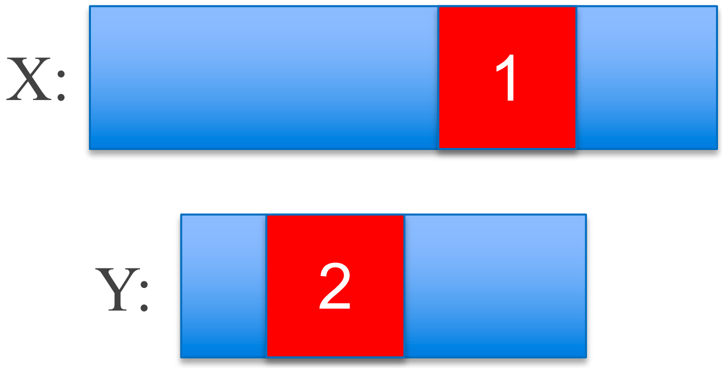
Specification

- 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:

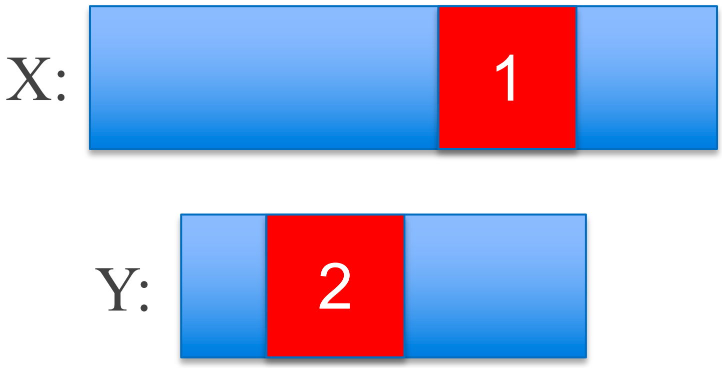
1. customer X orders burger, order ends up with cook 1
2. customer Y orders burger, order ends up with cook 2
3. cook 1 was busy with something else, so cook 2 grabs the lock first
4. cook 2 cooks burger for Y
5. cook 2 releases lock
6. cook 1 grabs lock
7. cook 1 cooks burger for X
8. cook 1 releases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



Implementation?

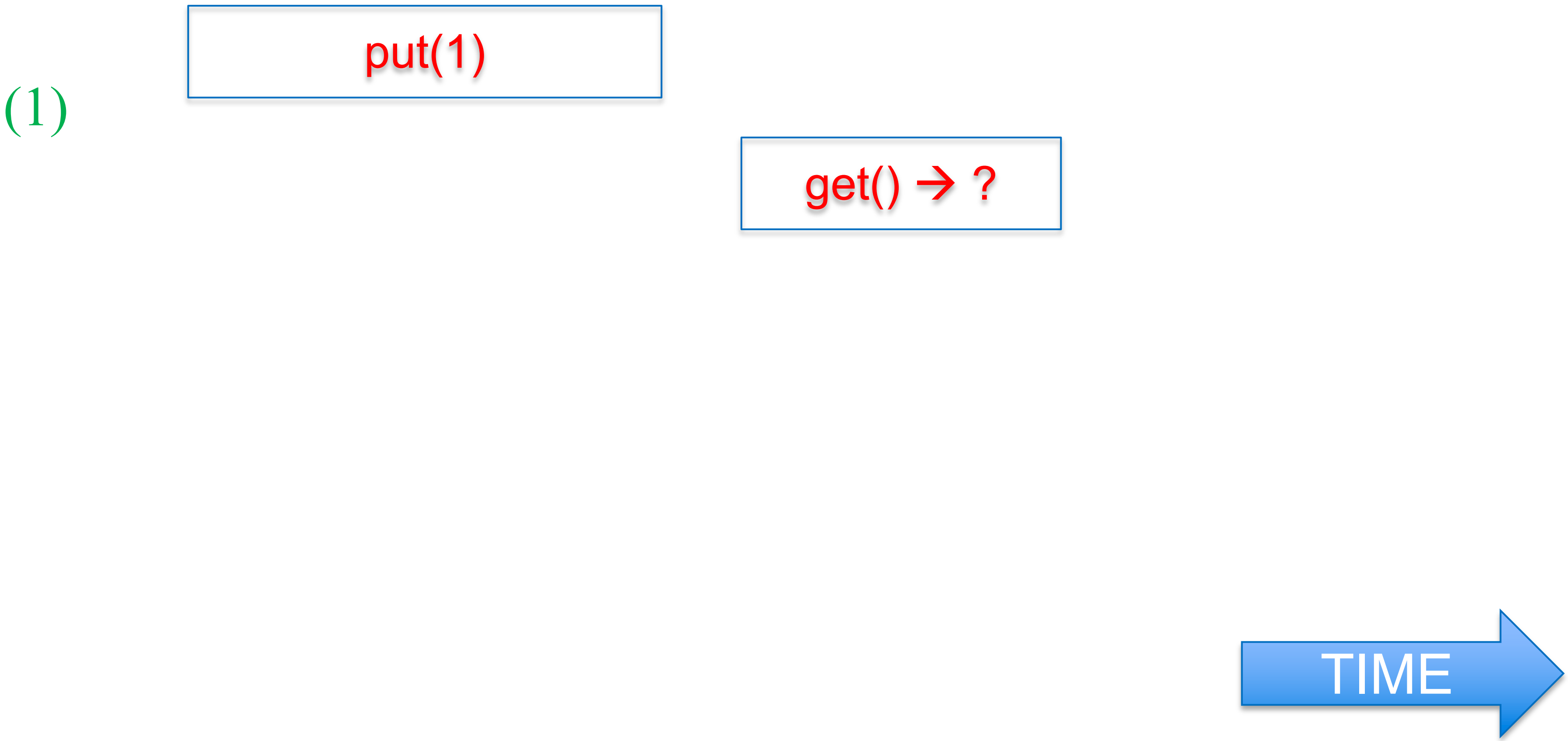
- Suppose the diner has one small hot plate and two cooks
- Cooks use a lock for access to the hot plate
- Possible scenario:

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

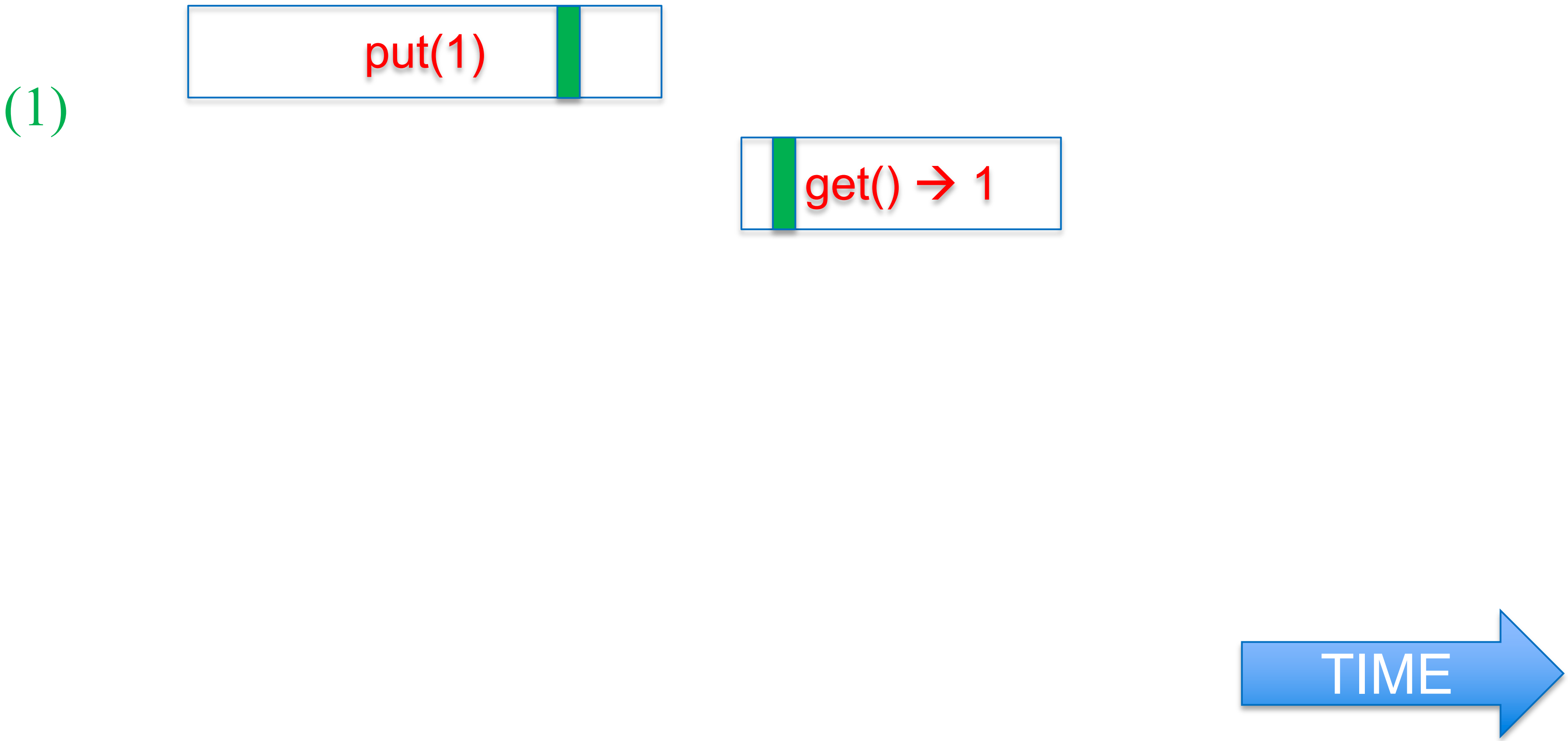


- *can't happen if Y orders burger after X receives burger*
- *but if operations overlap, any ordering can happen…*

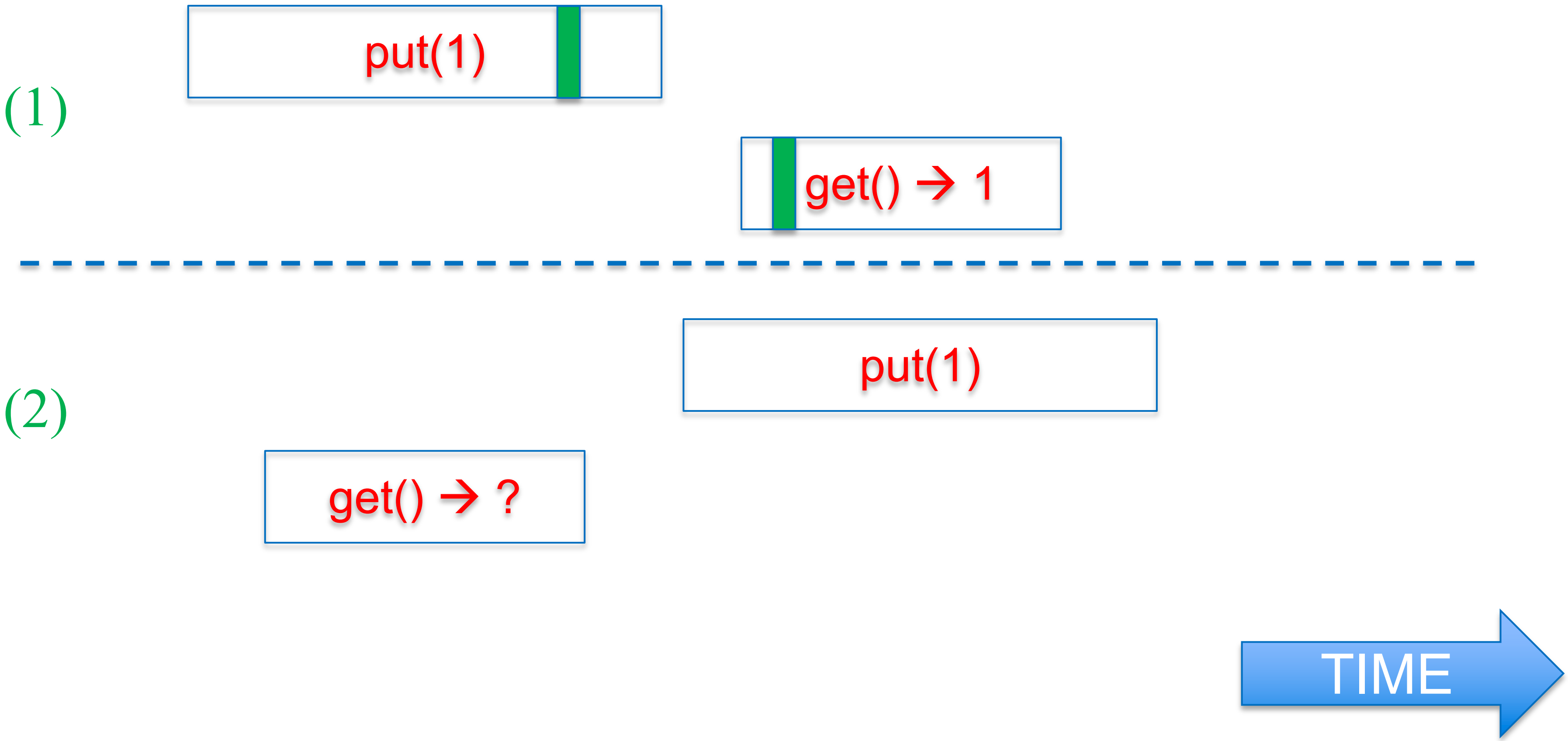
Correct Behaviors



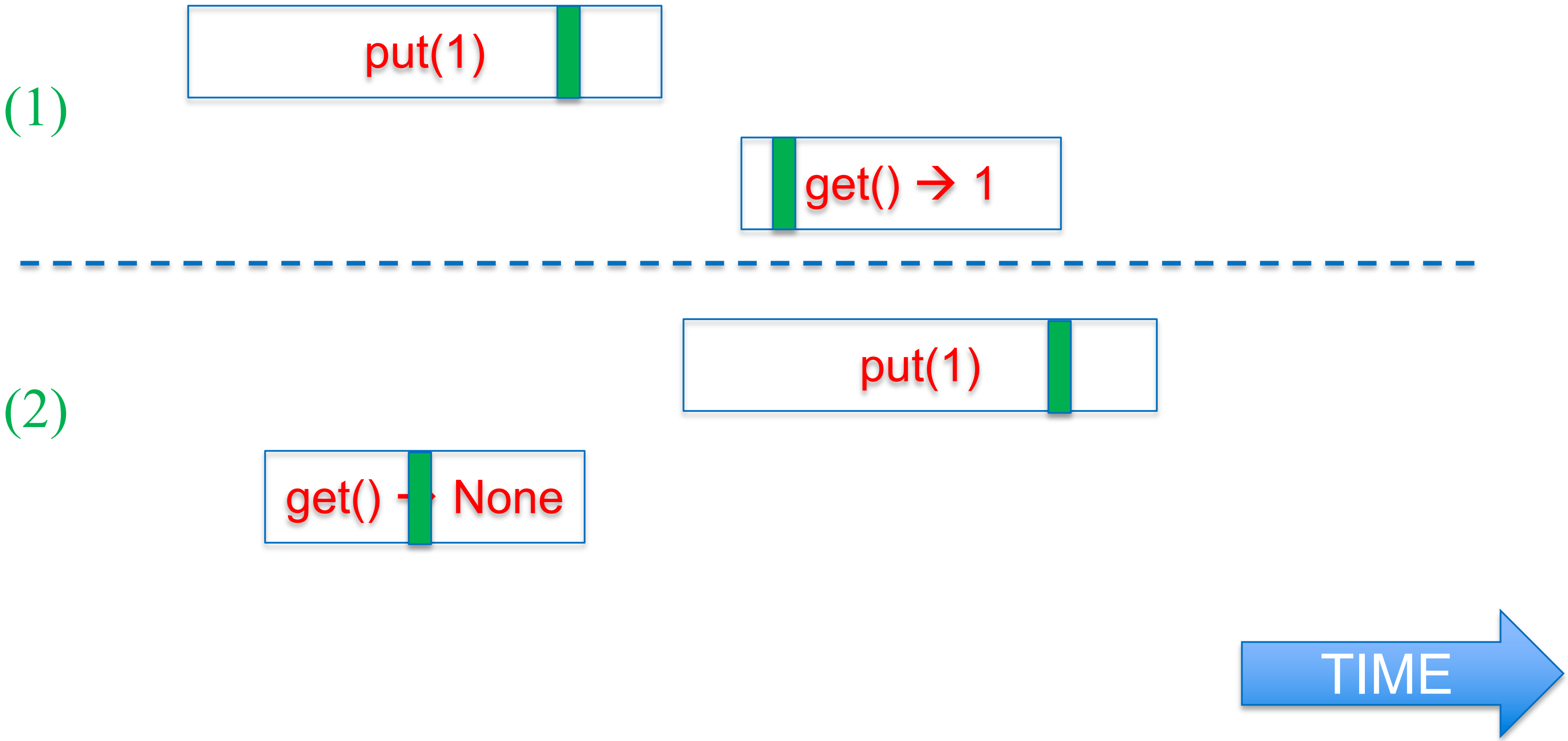
Correct Behaviors



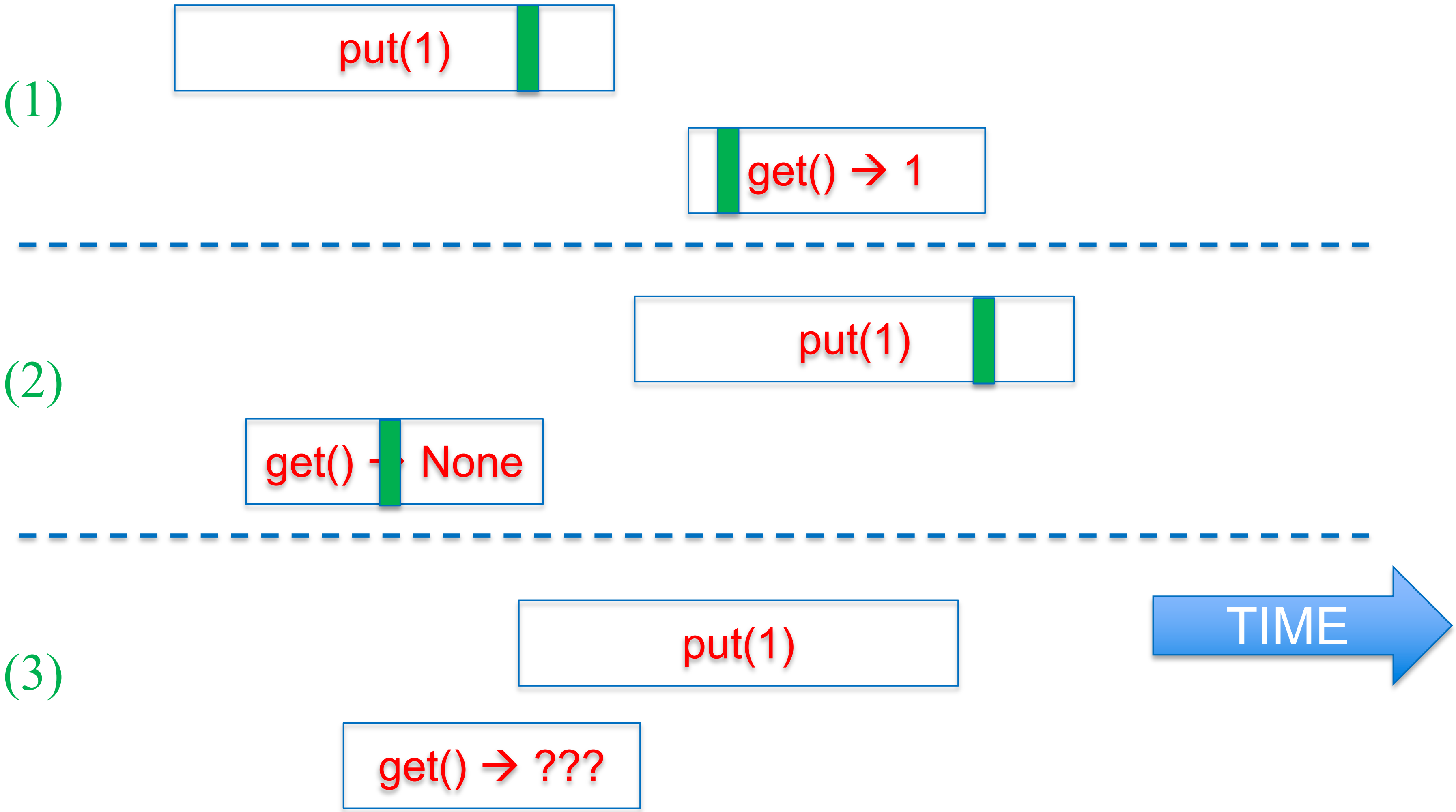
Correct Behaviors



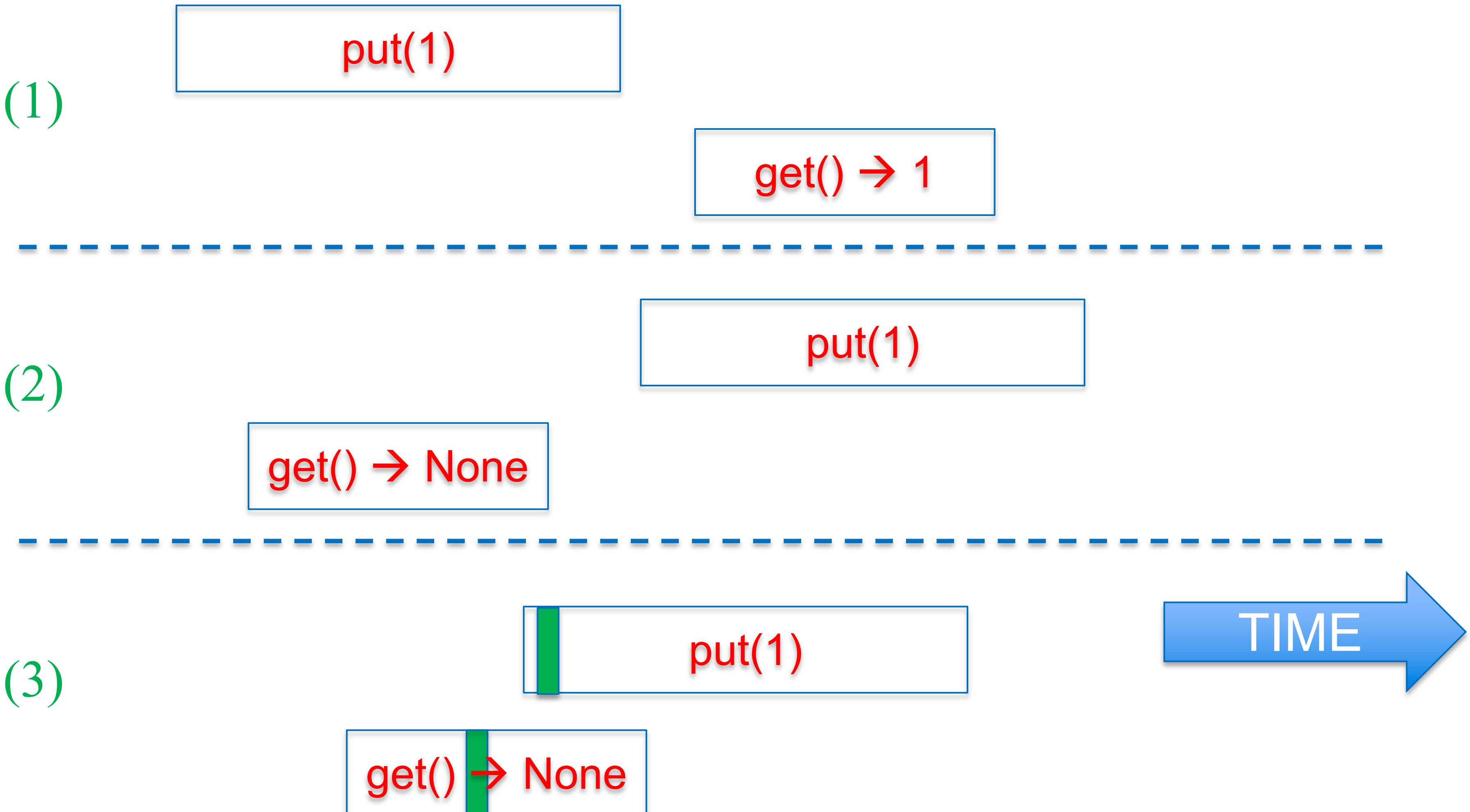
Correct Behaviors



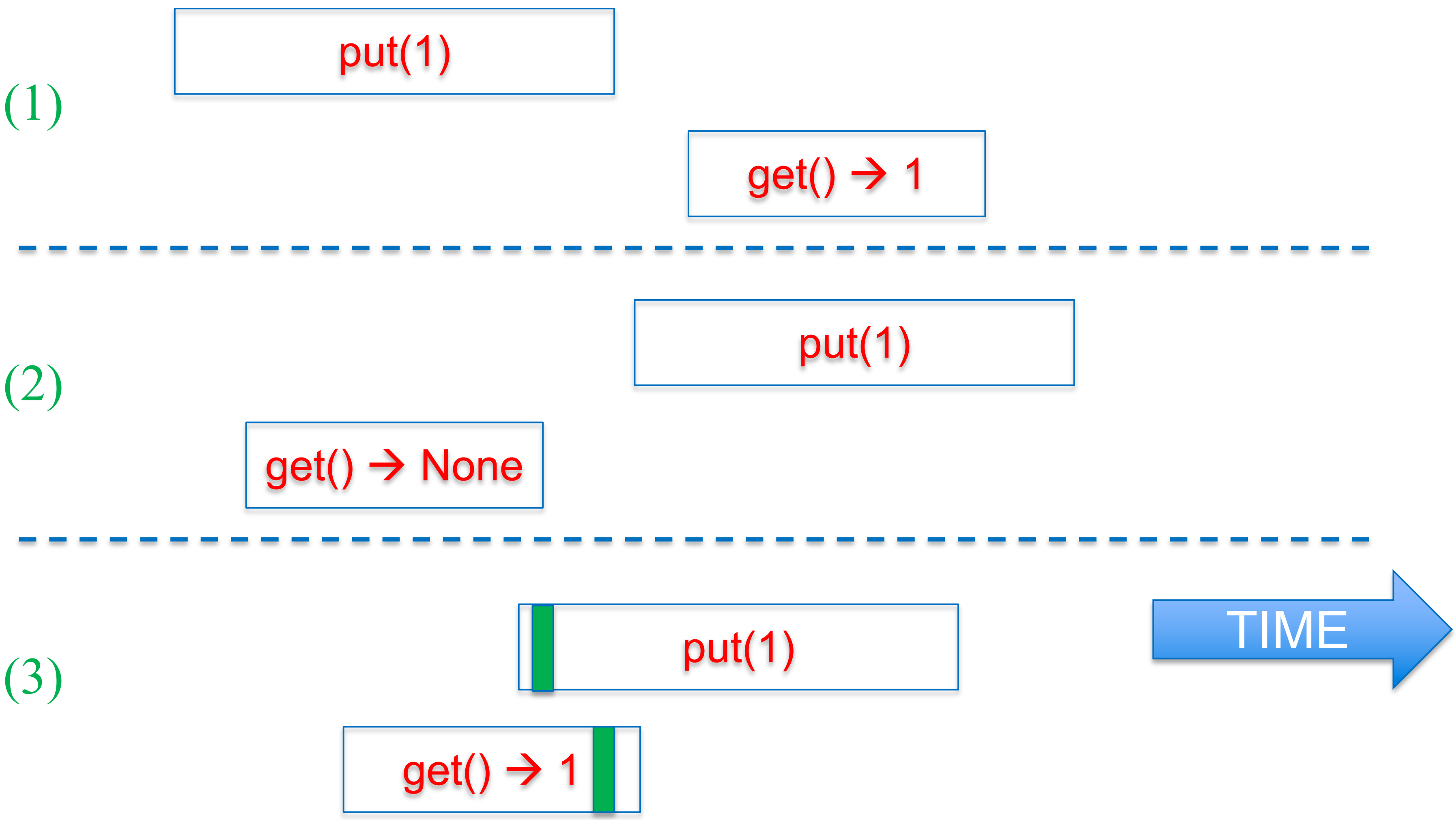
Correct Behaviors



Correct Behaviors



Correct Behaviors



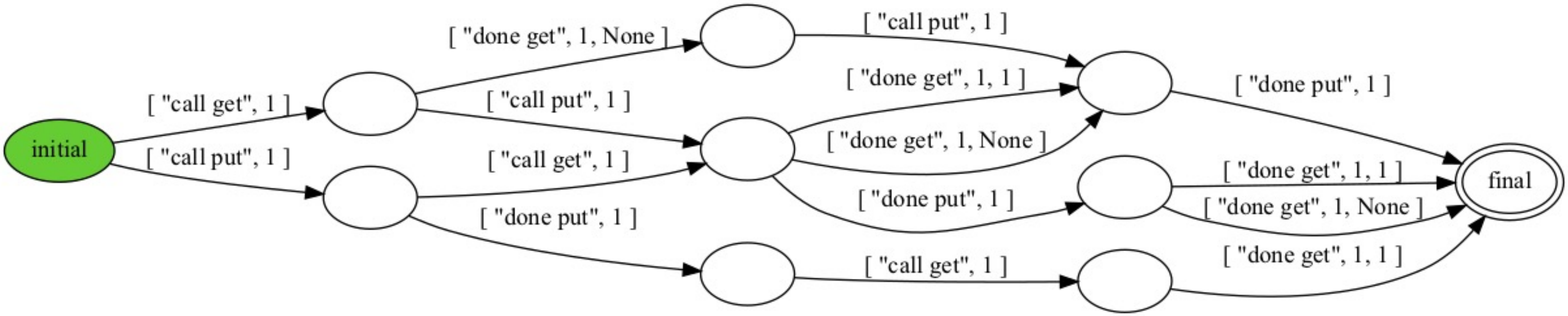
Testing Concurrent Objects

- 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 NOPS = 4
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)

nputs = choose {1..NOPS-1}
for i in {1..nputs}:
    spawn put_test(i)
for i in {1..NOPS-nputs}:
    spawn get_test(i)
```

Behavior (NOPS=2: 1 get, 1 put)




```
$ harmony -c NOPS=2 -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