Concurrent Programming: Critical Sections and Locks

CS 4410
Operating Systems

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

[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

```
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.

#states 2
2 components, 0 bad states
No issues

#states 11
Safety Violation
T0: __init__() [0-3,17-25] { shared: True }
T2: g() [13-16] { shared: False }
T1: f() [4-8] { shared: False }
Harmony assertion failed

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.

#states 2
2 components, 0 bad states
No issues

#states 11
Safety Violation
T0: __init__() [0-3,17-25] { shared: True }
T2: g() [13-16] { shared: False }
T1: f() [4-8] { shared: False }
Harmony assertion failed

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.

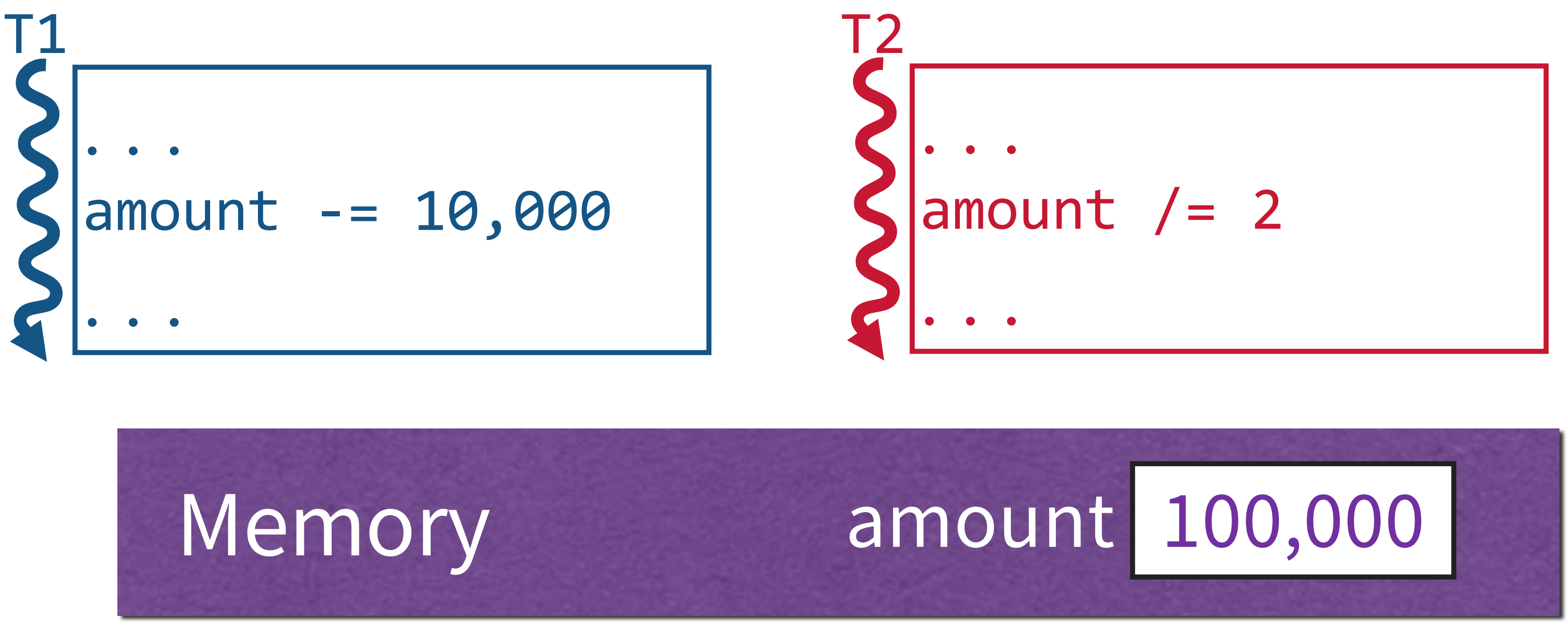
#states 2
2 components, 0 bad states
No issues

#states 11
Safety Violation
T0: __init__() [0-3,17-25] { shared: True }
T2: g() [13-16] { shared: False }
T1: f() [4-8] { shared: False }
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

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
. . .
```

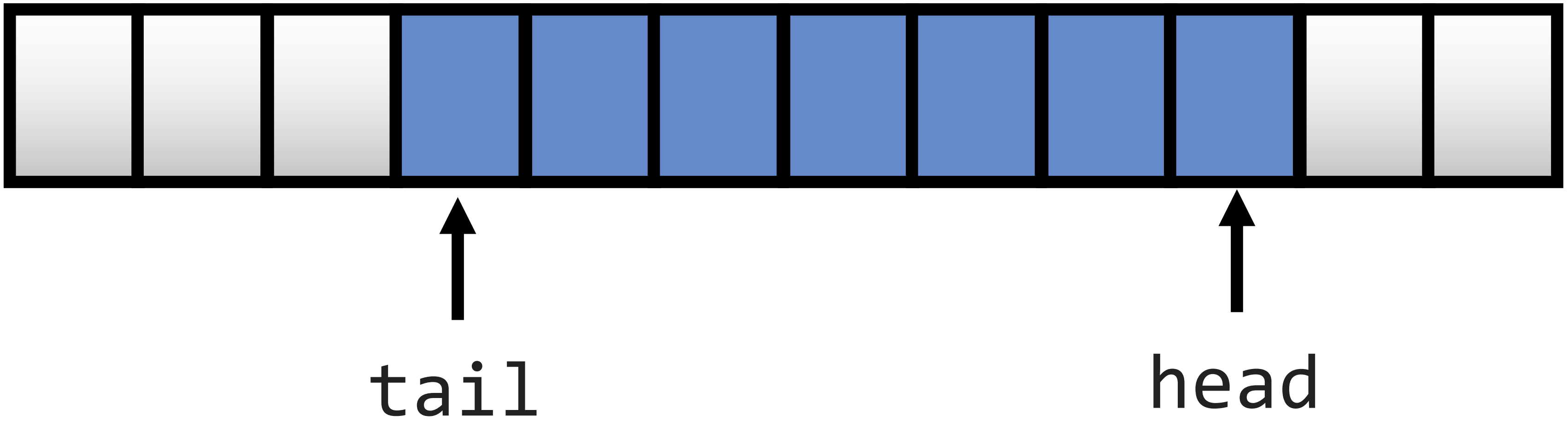
Memory amount 50,000

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 last spring

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

My take on the problem

- Handwritten proofs just as likely to have bugs as programs
  - or even more likely as you can’t test handwritten proofs
- Lack of mainstream tools to check concurrent algorithms
- Tools that do exist are great but have a steep learning curve

Examples of existing tools

```
bool turn, flag[2];          // the shared variables, booleans
byte ncrit;                  // nr of procs in critical section

active [2] proctype user()   // two processes
{
    assert(_pid == 0 || _pid == 1);
again:
    flag[_pid] = 1;
    turn = _pid;
    (flag[1 - _pid] == 0 || turn == 1 - _pid);

    ncrit++;
    assert(ncrit == 1);
    ncrit--;

    flag[_pid] = 0;
    goto again
}
```

Spin

```
--algorithm Peterson {
  variables flag = [i \in {0, 1} |-> FALSE
     \* Declares the global variables flag
     \* flag is a 2-element array with init
  fair process (proc \in {0,1}) {
    \* Declares two processes with identifie
    \* The keyword fair means that no proces
    \* always take a step.
   a1: while (TRUE) {
          skip ;  \* the noncritical section
   a2:    flag[self] := TRUE ;
   a3:    turn := 1 - self ;
   a4:    await (flag[1-self] = FALSE) \/ (turn
   cs:    skip ;  \* the critical section
   a5:    flag[self] := FALSE
                                          }
```

PlusCal

TLA+

VARIABLES *flag*, *turn*, *pc*

$vars \triangleq \langle flag, turn, pc \rangle$

$Init \triangleq \land\ flag = [i \in \{0,1\} \mapsto \text{FALSE}]$
$\qquad\quad \land\ turn = 0$
$\qquad\quad \land\ pc = [self \in \{0,1\} \mapsto \text{"a0"}]$

$a3a(self) \triangleq$
$\quad \land\ pc[self] = \text{"a3a"}$
$\quad \land\ \text{IF } flag[Not(self)]$
$\qquad \text{THEN } pc' = [pc \text{ EXCEPT } ![self] = \text{"a3b"}]$
$\qquad \text{ELSE } pc' = [pc \text{ EXCEPT } ![self] = \text{"cs"}]$
$\quad \land\ \text{UNCHANGED } \langle flag, turn \rangle$

* remaining actions omitted

$proc(self) \triangleq a0(self) \lor \ldots \lor a4(self)$

$Next \triangleq \exists\, self \in \{0,1\} : proc(self)$

$Spec \triangleq Init \land \Box[Next]_{vars}$

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
    done1 = True

def T2():
    amount /= 2
    done2 = True
```

Example (same as before)

```
def T1():
    amount −= 10000
    done1 = True

def T2():
    amount /= 2
    done2 = True

def main():
    await done1 and done2
    assert (amount == 40000) or (amount == 45000), amount

done1 = done2 = False
amount = 100000
spawn T1()
spawn T2()
spawn main()
```

Example (same as before)

```
def T1():
    amount −= 10000
    done1 = True

def T2():
    amount /= 2
    done2 = True

def main():
    await done1 and done2
    assert (amount == 40000) or (amount == 45000), amount

done1 = done2 = False
amount = 100000
spawn T1()
spawn T2()
spawn main()
```

Equivalent to:

```
while not (done1 and done2):
    pass
```

Example (same as before)

```
def T1():
    amount −= 10000
    done1 = True

def T2():
    amount /= 2
    done2 = True

def main():
    await done1 and done2
    assert (amount == 40000) or (amount == 45000), amount

done1 = done2 = False
amount = 100000
spawn T1()
spawn T2()
spawn main()
```

Assertion: useful to check properties

Example (same as before)

```
def T1():
    amount -= 10000
    done1 = True

def T2():
    amount /= 2
    done2 = True

def main():
    await done1 and done2
    assert (amount == 40000) or (amount == 45000), amount

done1 = done2 = False
amount = 100000
spawn T1()
spawn T2()
spawn main()
```

Output amount if assertion fails

An important note on assertions

- An assertion is not part of your algorithm
- Semantically, an assertion is a no-op
  - it's expected never to fail because it is supposed to state a fact

That said…

- Assertions are super-useful
  - @*label*: **assert** P is a type of *invariant*:

$$(pc = label) \Rightarrow P$$

- Use them liberally
  - In C, Java, …, they're automatically removed in production code
  - Or automatically optimized out if you have a really good compiler
- They are great for testing
- They are *executable documentation*
  - comments tend to get outdated over time

That said…

That said…

- Comment them out before you submit a programming assignment
  - you don't want your assertions to fail while we are testing your code ☺

Back to example

```
def T1():
    amount −= 10000
    done1 = True

def T2():
    amount /= 2
    done2 = True

def main():
    await done1 and done2
    assert (amount == 40000) or (amount == 45000), amount

done1 = done2 = False
amount = 100000
spawn T1()
spawn T2()
spawn main()
```

Initialize shared variables

Example (same as before)

```
def T1():
    amount −= 10000
    done1 = True

def T2():
    amount /= 2
    done2 = True

def main():
    await done1 and done2
    assert (amount == 40000) or (amount == 45000), amount

done1 = done2 = False
amount = 100000
spawn T1()
spawn T2()
spawn main()
```

Spawn three processes (threads)

Example (same as before)

```
def T1():
    amount −= 10000
    done1 = True

def T2():
    amount /= 2
    done2 = True

def main():
    await done1 and done2
    assert (amount == 40000) or (amount == 45000), amount

done1 = done2 = False
amount = 100000
spawn T1()
spawn T2()
spawn main()
```

```
#states = 100 diameter = 5
==== Safety violation ====
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]            5 { amount: 100000, done1: False, done2: False }
T2/() [10-17].        17 { amount: 50000,  done1: False, done2: True  }
T1/() [5-8]            8 { amount: 90000,  done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
>>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

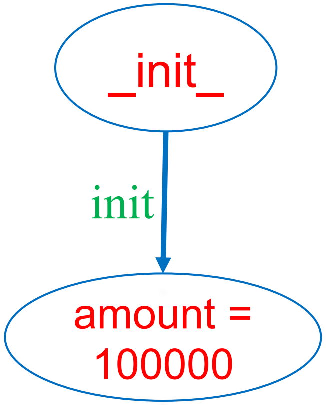
Simplified model (ignoring main)

T1a: LOAD amount	T2a: LOAD amount
T1b: SUB 10000	T2b: DIV 2
T1c: STORE amount	T2c: STORE amount

Simplified model (ignoring main)

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

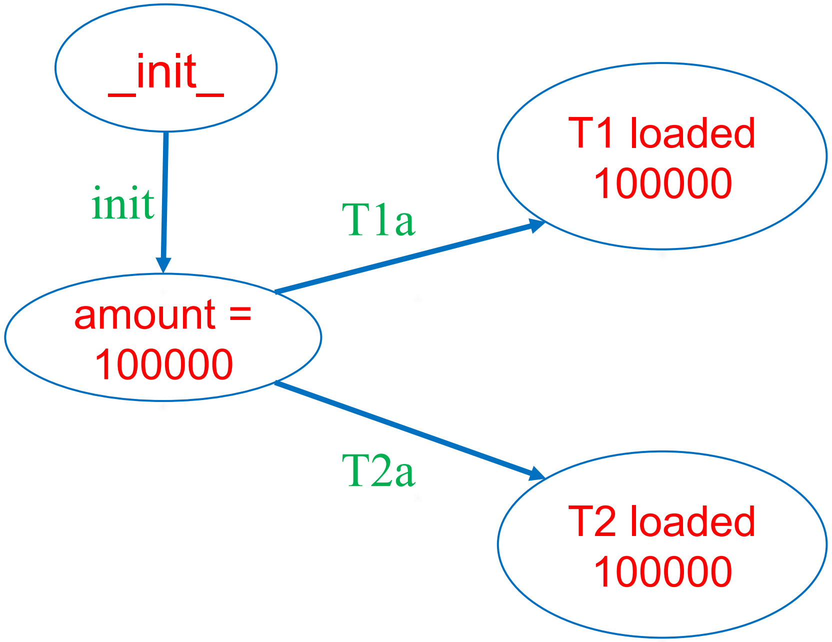
T2a: LOAD amount
T2b: DIV 2
T2c: STORE amount



Simplified model (ignoring main)

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

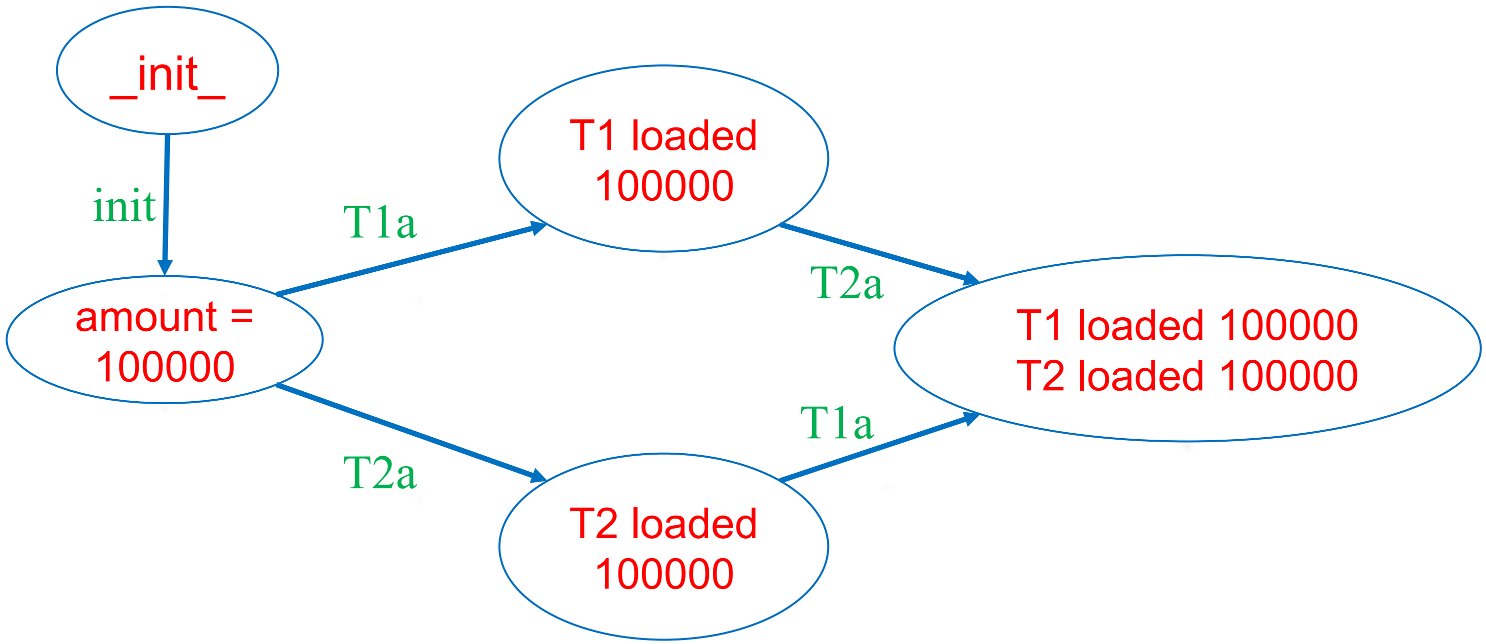
T2a: LOAD amount
T2b: DIV 2
T2c: STORE amount



Simplified model (ignoring main)

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

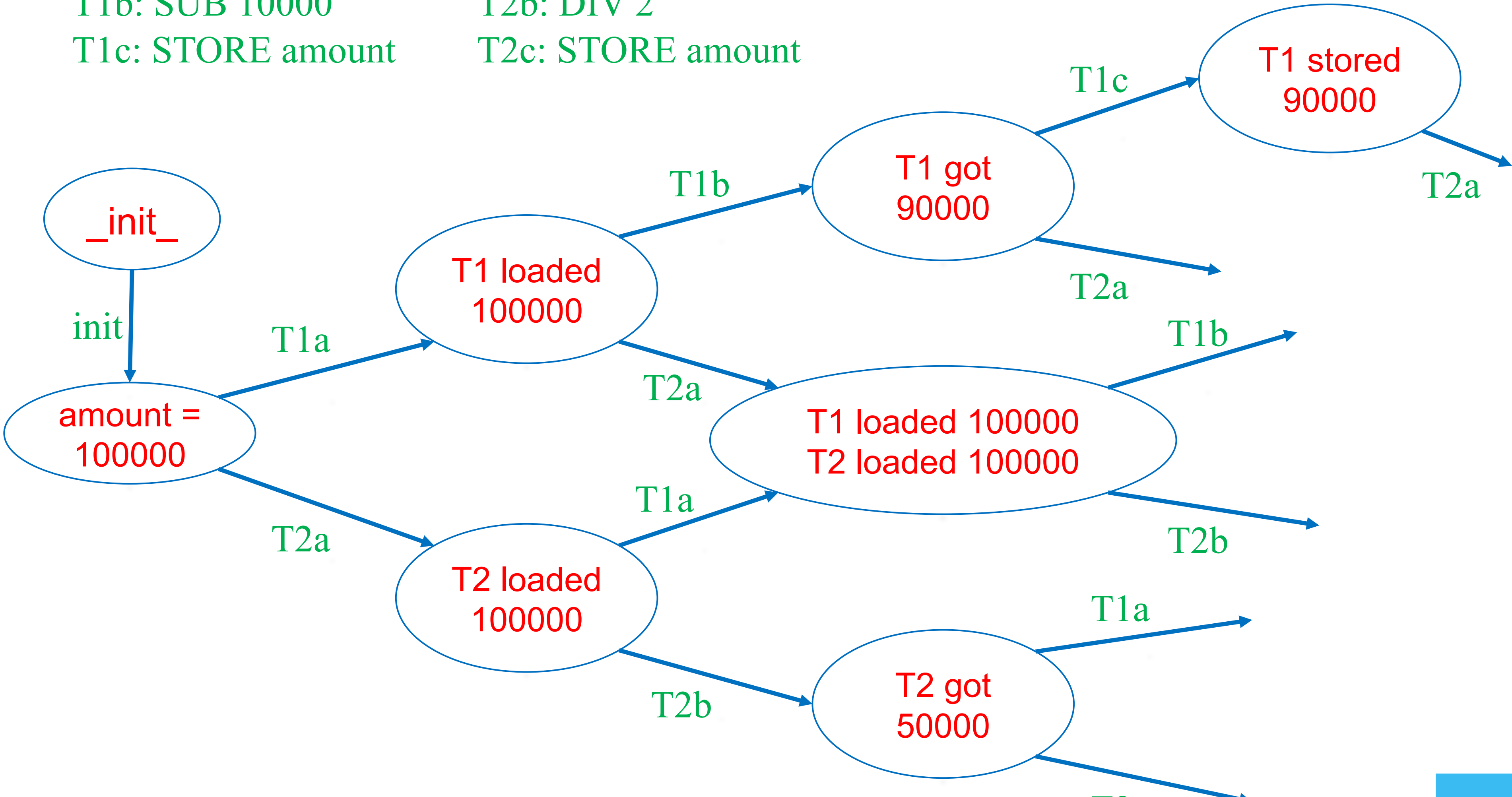
T2a: LOAD amount
T2b: DIV 2
T2c: STORE amount



Simplified model (ignoring main)

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

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



Harmony Output

```
#states = 100 diameter = 5
==== Safety violation ====
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]             5  { amount: 100000, done1: False, done2: False }
T2/() [10-17].          17 { amount: 50000,   done1: False, done2: True  }
T1/() [5-8]              8 { amount: 90000,   done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
>>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

Output

#states in the state graph

```
#states = 100 diameter = 5
==== Safety violation ====
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]             5  { amount: 100000, done1: False, done2: False }
T2/() [10-17].         17 { amount: 50000,   done1: False, done2: True  }
T1/() [5-8]             8 { amount: 90000,   done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
>>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

Output

something went wrong in (at least) one path in the graph (*assertion failure*)

```
#states = 100 diameter = 5
==== Safety violation ====
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]              5 { amount: 100000, done1: False, done2: False }
T2/() [10-17]           17 { amount: 50000,  done1: False, done2: True  }
T1/() [5-8]              8 { amount: 90000,  done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
>>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

Output

shortest path to assertion failure

```
#states = 100 diameter = 5
==== Safety violation ====
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]            5 { amount: 100000, done1: False, done2: False }
T2/() [10-17]         17 { amount: 50000,  done1: False, done2: True  }
T1/() [5-8]            8 { amount: 90000,  done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
>>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

turns (the five lines from `__init__/()` to `main/()`)

Output

init

```
#states = 100 diameter = 5
==== Safety violation ====
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]            5 { amount: 100000, done1: False, done2: False }
T2/() [10-17]         17 { amount: 50000,  done1: False, done2: True  }
T1/() [5-8]            8 { amount: 90000,  done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
>>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

Output

```
        #states = 100 diameter = 5
        ==== Safety violation ====
_init_  __init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1ab    T1/() [1-4]               5  { amount: 100000, done1: False, done2: False }
        T2/() [10-17]            17 { amount: 50000,   done1: False, done2: True  }
        T1/() [5-8]               8 { amount: 90000,   done1: True,  done2: True  }
        main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
        >>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

Output

```
        #states = 100 diameter = 5
        ==== Safety violation ====
_init_  __init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1ab    T1/() [1-4]                5 { amount: 100000, done1: False, done2: False }
T2abc   T2/() [10-17]             17 { amount: 50000,  done1: False, done2: True  }
        T1/() [5-8]                8 { amount: 90000,  done1: True,  done2: True  }
        main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
        >>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

Output

```
        #states = 100 diameter = 5
        ==== Safety violation ====
_init_  __init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1ab    T1/() [1-4]               5  { amount: 100000, done1: False, done2: False }
T2abc   T2/() [10-17]            17 { amount: 50000,   done1: False, done2: True  }
T1c     T1/() [5-8]               8 { amount: 90000,   done1: True,  done2: True  }
        main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
        >>> Harmony Assertion (file=test.hny, line=11) failed: 90000
```

Output

#states = 100 diameter = 5

==== Safety violation ====

init	__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1ab	T1/() [1-4] 5 { amount: 100000, done1: False, done2: False }
T2abc	T2/() [10-17] 17 { amount: 50000, done1: False, done2: True }
T1c	T1/() [5-8] 8 { amount: 90000, done1: True, done2: True }
main	main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }

>>> Harmony Assertion (file=test.hny, line=11) failed: 90000

Output

#states = 100 diameter = 5

==== Safety violation ====

init	__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1ab	T1/() [1-4] 5 { amount: 100000, done1: False, done2: False }
T2abc	T2/() [10-17] 17 { amount: 50000, done1: False, done2: True }
T1c	T1/() [5-8] 8 { amount: 90000, done1: True, done2: True }
main	main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }

>>> Harmony Assertion (file=test.hny, line=11) failed: 90000

Output

name of a thread

```
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]                5  { amount: 100000, done1: False, done2: False }
T2/() [10-17].            17 { amount: 50000,   done1: False, done2: True  }
T1/() [5-8]                8 { amount: 90000,   done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
```

Output

"steps" =
list of program counters
of machine instructions
executed

```
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]             5  { amount: 100000, done1: False, done2: False }
T2/() [10-17]           17 { amount: 50000,   done1: False, done2: True  }
T1/() [5-8]              8 { amount: 90000,   done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
```

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 Push True	
7 Store done1	T1d: done1 = True
8 Return	
9 Jump 40	
10 Frame T2 ()	
11 Load amount	T2a: LOAD amount
12 Push 2	T2b: DIV 2
13 2-ary /	
14 Store amount	T2c: STORE amount
15 Push True	
16 Store done2	T2d: done2 = True
17 Return	
18 …	

```
def T1():
    amount −= 10000
    done1 = True
```

```
def T2():
    amount /= 2
    done2 = True
```

Harmony Machine Code

```
0  Jump 40               PC := 40
1  Frame T1 ()
2  Load amount
3  Push 10000
4  2-ary –
5  Store amount
6  Push True
7  Store done1
8  Return
9  Jump 40
10 Frame T2 ()
11 Load amount
12 Push 2
13 2-ary /
14 Store amount
15 Push True
16 Store done2
17 Return
18 …
```

Harmony Machine Code

```
0  Jump 40              PC := 40
1  Frame T1 ()
2  Load amount          push amount onto the stack of thread T1
3  Push 10000
4  2-ary –
5  Store amount
6  Push True
7  Store done1
8  Return
9  Jump 40
10 Frame T2 ()
11 Load amount
12 Push 2
13 2-ary /
14 Store amount
15 Push True
16 Store done2
17 Return
18 …
```

Harmony Machine Code

```
0  Jump 40            PC := 40
1  Frame T1 ()
2  Load amount        push amount onto the stack of thread T1
3  Push 10000         push 10000 onto the stack of thread T1
4  2-ary –            replace top two elements of stack with difference
5  Store amount
6  Push True
7  Store done1
8  Return
9  Jump 40
10 Frame T2 ()
11 Load amount
12 Push 2
13 2-ary /
14 Store amount
15 Push True
16 Store done2
17 Return
18 …
```

Harmony Machine Code

```
0 Jump 40                PC := 40
1 Frame T1 ()
2 Load amount            push amount onto the stack of thread T1
3 Push 10000             push 10000 onto the stack of thread T1
4 2-ary –                replace top two elements of stack with difference
5 Store amount           store top of the stack of T1 into amount
6 Push True
7 Store done1
8 Return
9 Jump 40
10 Frame T2 ()
11 Load amount
12 Push 2
13 2-ary /
14 Store amount
15 Push True
16 Store done2
17 Return
18 …
```

Harmony Machine Code

0 Jump 40	PC := 40
1 Frame T1 ()	
2 Load amount	push amount onto the stack of process T1
3 Push 10000	push 10000 onto the stack of process T1
4 2-ary –	replace top two elements of stack with difference
5 Store amount	store top of the stack of T1 into amount
6 Push True	push True onto the stack of thread T1
7 Store done1	store top of the stack of T1 into done1
8 Return	
9 Jump 40	
10 Frame T2 ()	
11 Load amount	
12 Push 2	
13 2-ary /	
14 Store amount	
15 Push True	
16 Store done2	
17 Return	
18 …	

Harmony Machine Code

0 Jump 40	PC := 40
1 Frame T1 ()	
2 Load amount	push amount onto the stack of process T1
3 Push 10000	push 10000 onto the stack of process T1
4 2-ary –	replace top two elements of stack with difference
5 Store amount	store top of the stack of T1 into amount
6 Push True	push True onto the stack of thread T1
7 Store done1	store top of the stack of T1 into done1
8 Return	
9 Jump 40	
10 Frame T2 ()	
11 Load amount	push amount onto the stack of thread T2
12 Push 2	push 2 onto the stack of thread T2
13 2-ary /	replace top two elements of stack with division
14 Store amount	store top of the stack of T2 into amount
15 Push True	push True onto the stack of thread T2
16 Store done2	store top of the stack of T2 into done2
17 Return	
18 …	

Output

current program counter
(after turn)

```
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]           5  { amount: 100000, done1: False, done2: False }
T2/() [10-17]         17 { amount: 50000,  done1: False, done2: True  }
T1/() [5-8]           8  { amount: 90000,  done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
```

Output

current state
(after turn)

```
__init__/() [0,40-58] 58 { amount: 100000, done1: False, done2: False }
T1/() [1-4]            5 { amount: 100000, done1: False, done2: False }
T2/() [10-17]         17 { amount: 50000,  done1: False, done2: True  }
T1/() [5-8]            8 { amount: 90000,  done1: True,  done2: True  }
main/() [19-23,25-34,36-37] 37 { amount: 90000, done1: True, done2: True }
```

Harmony Virtual Machine *State*

Three parts:

1. code (which never changes)
2. values of the shared variables
3. states of each of the running processes
   – "contexts"

State represents one vertex in the graph model

Context (state of a process)

- Method name and parameters
- PC (program counter)
- stack (+ implicit stack pointer)
- local variables
  - parameters (aka arguments)
  - “result”
    - there is no **return** statement
  - local variables
    - declared in **var**, **let**, and **for** statements

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… …puts
  - **cons**…
    - **c**…
    - ca… …en with "-c x=4" flag to harmony
  - Outp…:
    - **print** x + y
    - **assert** x + y < 10, (x, y)

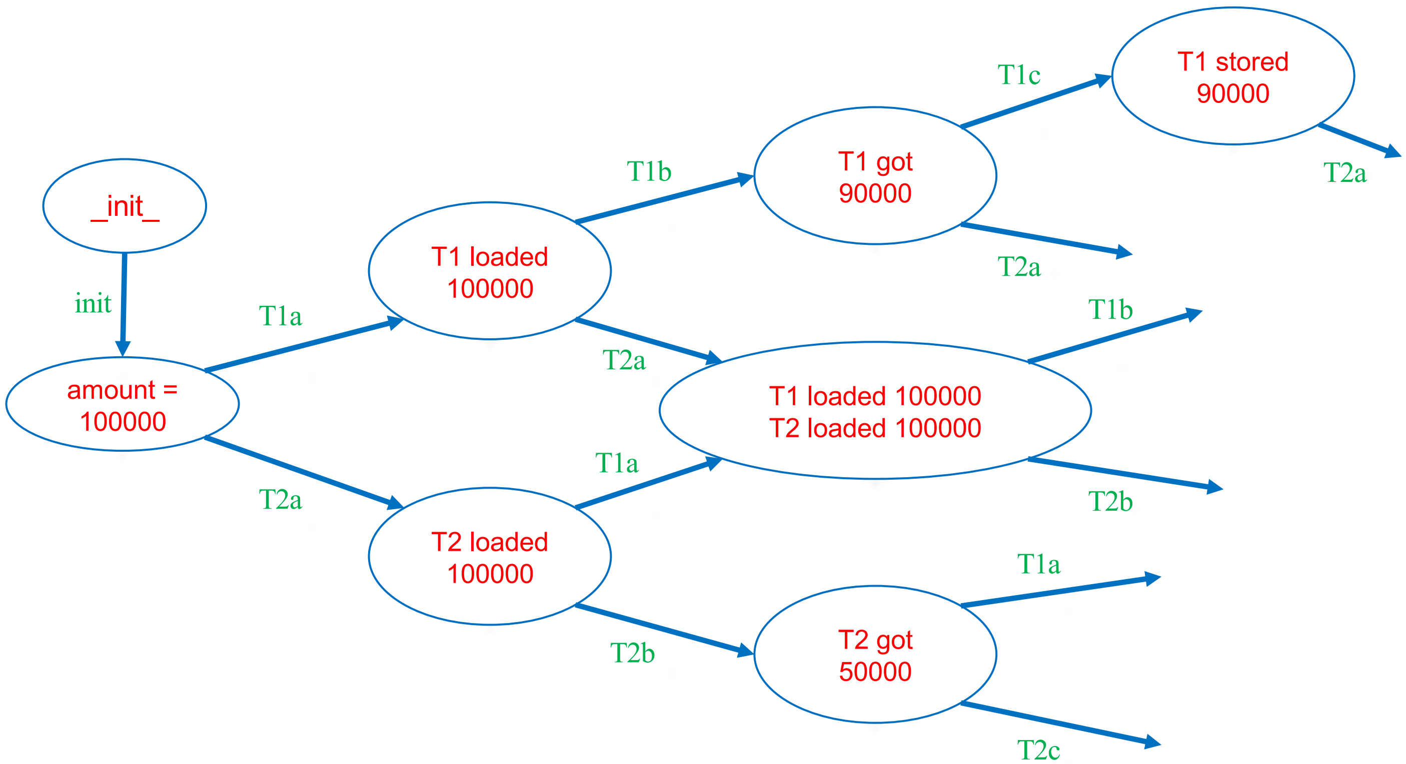
No open(), read(), or
or input() statements

Non-determinism in Harmony

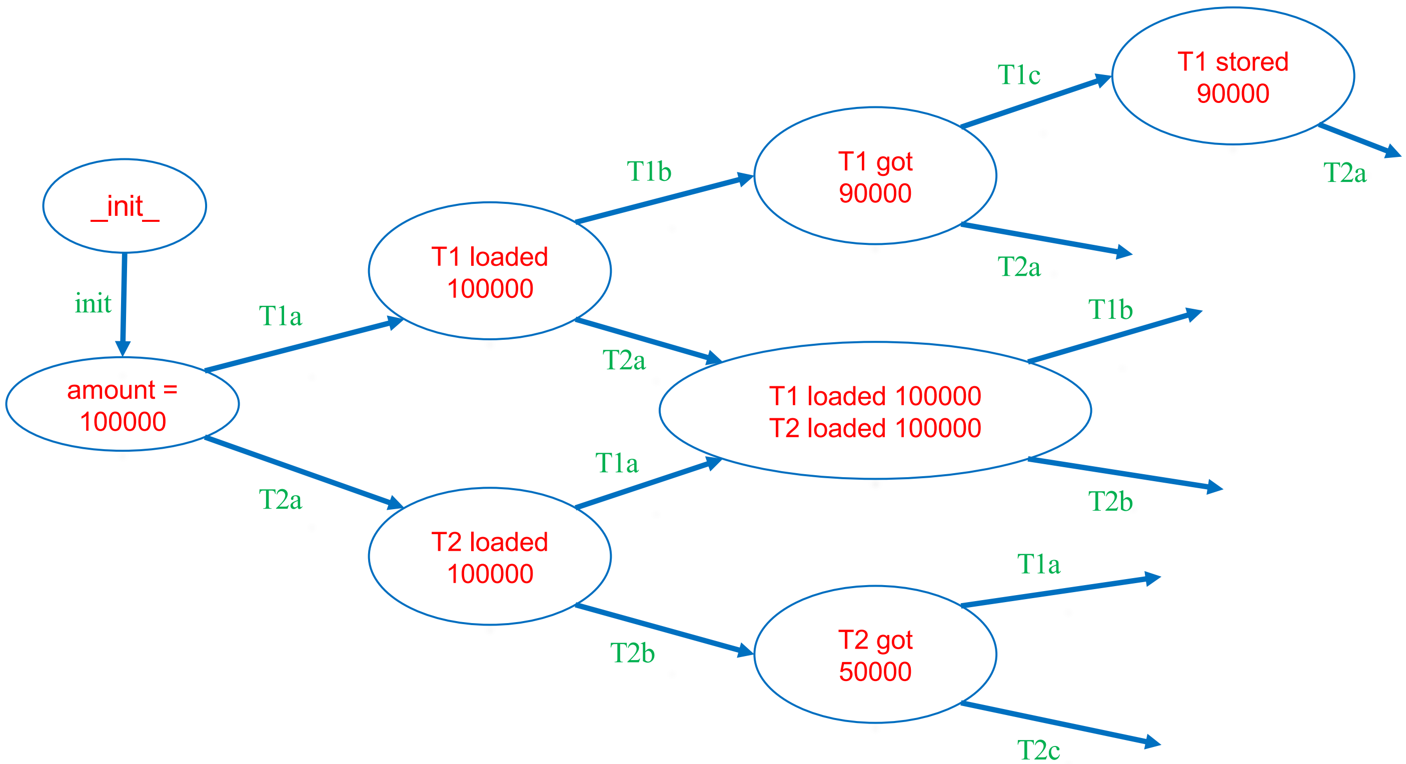
Three sources:

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

Limitation: models must be finite!



Limitation: models must be finite!

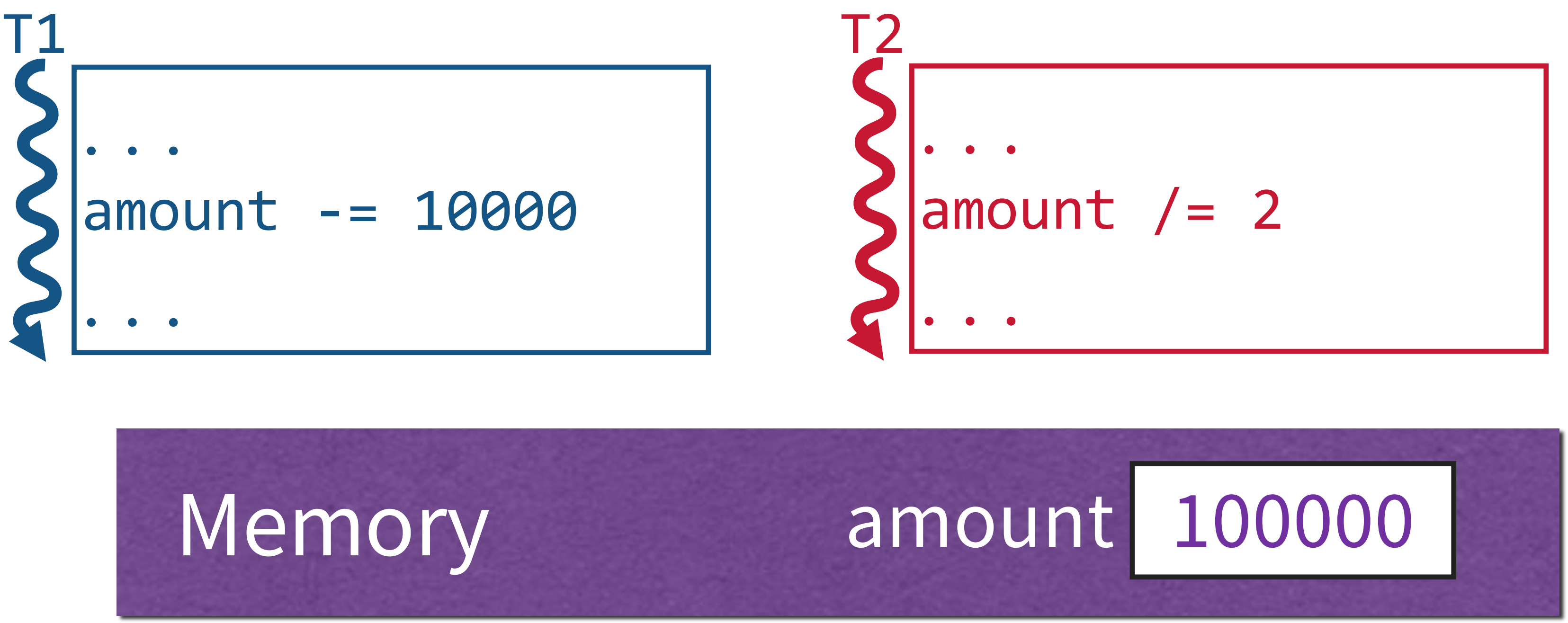


- But models are allowed to have cycles.
- Executions are allowed to be unbounded!
- Harmony checks for *possibility* of termination

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: all threads make it into the CS if desired

Fairness: equal chances of getting into CS

… in practice, fairness rarely guaranteed

Critical Section

Must be serialized due to shared memory access

T1

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

T2

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

<u>Goals</u>

Mutual Exclusion: 1 thread in a critical section at time

Progress: at least one thread makes it into the CS if desired and no other thread is there

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(self):
    while True:
        …    # code outside critical section
        …    # code to enter the critical section
        …    # critical section itself
        …    # code to exit the critical section

spawn thread(1)
spawn thread(2)
…
```

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

Critical Sections in Harmony

```
def thread(self):
    while True:
        …    # code outside critical section
        …    # code to enter the critical section
        cs: assert countLabel(cs) == 1
        …    # code to exit the critical section

spawn thread(1)
spawn thread(2)
…
```

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

Critical Sections in Harmony

```
def thread(self):
    while choose( { False, True } ):
        …    # code outside critical section
        …    # code to enter the critical section
        cs: assert countLabel(cs) == 1
        …    # code to exit the critical section

spawn thread(1)
spawn thread(2)
…
```

- How do we check mutual exclusion?
- How do we check progress?
  - *if code to enter/exit the critical section cannot terminate, Harmony with balk*

Sounds like you need a lock…

- True, but this is an O.S. class!
- The question is:

How does one build a lock?

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

You have to program them!

First attempt: a naïve lock

```
lockTaken = False

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

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        lockTaken = False

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

Figure 5.3: [code/naiveLock.hny] Naïve implementation of a shared lock.

First attempt: a naïve lock

```
lockTaken = False

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

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        lockTaken = False

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

wait till lock is free, then take it

Figure 5.3: [code/naiveLock.hny] Naïve implementation of a shared lock.

First attempt: a naïve lock

```
lockTaken = False

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

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        lockTaken = False

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

wait till lock is free, then take it

release the lock

Figure 5.3: [code/naiveLock.hny] Naïve implementation of a shared lock.

First attempt: a naïve lock

```
1   lockTaken = False
2
3   def thread(self):
4       while choose({ False, True }):
5           # Enter critical section
6           await not lockTaken
7           lockTaken = True
8
9           # Critical section
10          @cs: assert atLabel(cs) == { (thread, self): 1 }
11
12          # Leave
13          lockTake
14
15  spawn thread(
16  spawn thread(
```

```
==== Safety violation ====
__init__/() [0,26-36]                    36 { lockTaken: False }
thread/0 [1-2,3(choose True),4-7]         8 { lockTaken: False }
thread/1 [1-2,3(choose True),4-8]         9 { lockTaken: True }
thread/0 [8-19]                          19 { lockTaken: True }
>>> Harmony Assertion (file=code/naiveLock.hny, line=10) failed
```

Figure 5.3: [code/naiveLock.hny] Naive implementation of a shared lock.

Second attempt: *flags*

```
flags = [ False, False ]

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

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

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

Figure 5.5: [code/naiveFlags.hny] Naïve use of flags to solve mutual exclusion.

Second attempt: *flags*

```
flags = [ False, False ]

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

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

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

enter, then wait for other

Figure 5.5: [code/naiveFlags.hny] Naïve use of flags to solve mutual exclusion.

Second attempt: *flags*

```
flags = [ False, False ]

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

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

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

enter, then wait for other

leave

Figure 5.5: [code/naiveFlags.hny] Naïve use of flags to solve mutual exclusion.

Second attempt: *flags*

```
flags = [ False, False ]

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

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

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

Figure 5.5: [code/naiveFlags.hny] Naïve use of flags to solve mutual exclusion.

Second attempt: *flags*

```
flags = [ False, False ]

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

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

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

```
==== Non-terminating State ===
__init__/() [0,36-46]                    46 { flags: [False, False] }
thread/0 [1-2,3(choose True),4-12] 13 { flags: [True,  False] }
thread/1 [1-2,3(choose True),4-12] 13 { flags: [True,  True] }
blocked thread: thread/1 pc = 13
blocked thread: thread/0 pc = 13
```

Figure 5.5: [code/n

Third attempt: *turn* variable

```
turn = 0

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

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section

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

Figure 5.7: [code/naiveTurn.hny] Naïve use of turn variable to solve mutual exclusion.

Third attempt: *turn* variable

```
turn = 0

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

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section

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

after you...

Figure 5.7: [code/naiveTurn.hny] Naïve use of turn variable to solve mutual exclusion.

Third attempt: *turn* variable

```
turn = 0

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

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section

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

after you...

wait for your turn

Figure 5.7: [code/naiveTurn.hny] Naïve use of turn variable to solve mutual exclusion.

Third attempt: *turn* variable

```
turn = 0

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

        # Critical section
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section

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

```
==== Non-terminating State ===
__init__/() [0,28-38]                                   38 { turn: 0 }
thread/0 [1-2,3(choose True),4-26,2,3(choose True),4]    5 { turn: 1 }
thread/1 [1-2,3(choose False),4,27]                     27 { turn: 1 }
blocked thread: thread/0 pc = 5
```

Figure 5.7: [co

Peterson's Algorithm: *flags & turn*

```
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)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

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

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

Peterson's Algorithm: *flags & turn*

```
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)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

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


Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

Peterson's Algorithm: *flags & turn*

```
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)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

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

"you go first"

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

Peterson's Algorithm: *flags & turn*

```
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)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

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

"you go first"

wait until alone or it's my turn

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

Peterson's Algorithm: *flags & turn*

```
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)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

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

"you go first"

wait until alone or it's my turn

leave

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

Peterson's Algorithm: *flags & turn*

```
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)

        # critical section is here
        @cs: assert atLabel(cs) == { (thread, self): 1 }

        # Leave critical section
        flags[self] = False

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

#states = 104 diameter = 5
#components: 37
no issues found

Figure 6.1: [code/Peterson.hny] Peterson's Algorithm

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)

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]

Peterson’s Reconsidered

- Mutual Exclusion can be implemented with atomic LOAD and STORE instructions to access shared memory
  - multiple STOREs and LOADs
- Peterson’s can be generalized to >2 processes
  - even more STOREs and LOADs

Too inefficient in practice

Peterson's Reconsidered *More*

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

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
  - For example, address of x is 0x12340002

Concurrent writing

- Suppose x is a 32 bit word @ 0x12340002
- Suppose you have 2 threads, T1 and T2
  - T1: $x = 0xFFFFFFFF$ (i.e., $x = -1$)
  - T2: $x = 0$
- After T1 and T2 are done, x may be
  - 0, 0xFFFFFFFF, 0xFFFF0000, or 0x0000FFFF
- Because of this, programming languages will typically leave the outcome of concurrent write operations to a variable *undefined*.

Concurrent reading

- Suppose x is a 32 bit word @ 0x12340002
- Suppose x is initially 0
- Suppose you have 2 threads, T1 and T2
  - T1: x = 0xFFFFFFFF (i.e., $x = -1$)
  - T2: $y = x$ (i.e., T2 reads x)
- After T1 and T2 are done, y may contain
  - 0, 0xFFFFFFFF, 0xFFFF0000, or 0x0000FFFF
- Because of this, programming languages will typically leave the outcome of concurrent read and write operations to a variable *undefined*.

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 modern CPUs/compilers, 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

So, what do we do?

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 shared variable and p == ?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 shared variable and p == ?x, then we say that p is a *pointer* to x
- Finally, !p refers to the value of x

Where?

There!

Test-and-Set in Harmony

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

- For example:

```
lock1 = False
lock2 = True
r1 = test_and_set(?lock1)
r2 = test_and_set(?lock2)
assert lock1 and lock2
assert (not r1) and r2
```

Recall: bad lock implementation

```
lockTaken = False

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

        # Critical section
        cs: assert countLabel(cs) == 1

        # Leave critical section
        lockTaken = False

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

Good implementation (“spinlock”)

```
lockTaken = False

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

def thread(self):
    while choose( { False, True } ):
        # enter critical section
        while test_and_set(?lockTaken):
            pass

        cs: countLabel(cs) == 1

        # exit critical section
        atomically lockTaken = False

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

“Locks”

Best understood as “baton passing”

- At most one thread, or *shared*, can “hold” 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
```

"Ghost" state

- We say that a lock is *held* or *owned* by a thread
  - implicit "ghost" state
  - nonetheless can be used for reasoning

- Two important invariants:
  1. $T@cs \Rightarrow T$ holds the lock
  2. at most one thread can hold the lock

Most systems (incl. "standard" Harmony modules) do not keep track of who holds a particular lock, if anybody

Implementing a lock

(just one way of doing so)

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

def Lock() returns result:
    result = False

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

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

specification of the CPU's test_and_set functionality

must also use an atomic STORE instruction

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 Lock() returns result:
    result = False

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

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

Specification: describes *what an abstraction does*

Implementation: describes *how*

Using a lock for a critical section

```
import synch

const NTHREADS = 2

lock = synch.Lock()

def thread():
    while choose({ False, True }):
        synch.acquire(?lock)
        cs: assert countLabel(cs) == 1
        synch.release(?lock)

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

Spinlocks and Time Sharing

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

```
import list

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 (list.head(lk→suspended)) ()
            lk→suspended = list.tail(lk→suspended)
```

.acquired: boolean

.suspended: queue of contexts

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

```
import list

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

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.

```
def release(lk):
    atomically:
        assert lk→acquired
        if lk→suspended == []:
            lk→acquired = False
        else:
            go (list.head(lk→suspended)) ()
            lk→suspended = list.tail(lk→suspended)
```

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

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**.

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*

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

```
import list

def Queue() returns empty:
    empty = []

def put(q, v):
    !q = list.append(!q, v)

def get(q) returns next:
    if !q == []:
        next = None
    else:
        next = list.head(!q)
        !q = list.tail(!q)
```

(a) [code/queuespec.hny] Sequential

```
import list

def Queue() returns empty:
    empty = []

def put(q, v):
    atomically !q = list.append(!q, v)

def get(q) returns next:
    atomically:
        if !q == []:
            next = None
        else:
            next = list.head(!q)
            !q = list.tail(!q)
```

(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

Figure 11.2: [code/queuedemo.hny] Using a concurrent queue

Specifying a concurrent queue

```
import list

def Queue() returns empty:
    empty = []

def put(q, v):
    !q = list.append(!q, v)

def get(q) returns next:
    if !q == []:
        next = None
    else:
        next = list.head(!q)
        !q = list.tail(!q)
```

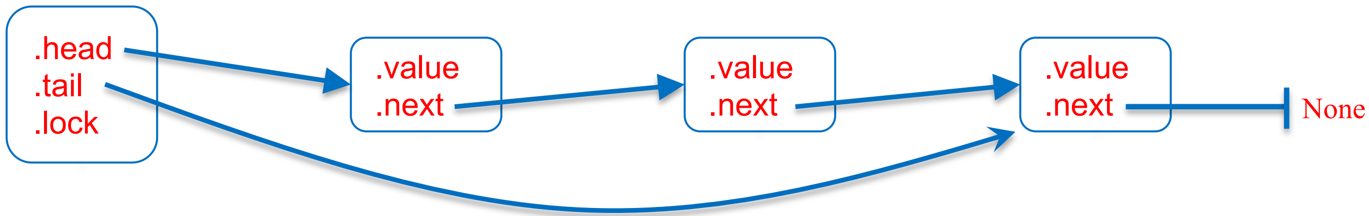
(a) [code/queuespec.hny] Sequential

```
import list

def Queue() returns empty:
    empty = []

def put(q, v):
    atomically !q = list.append(!q, v)

def get(q) returns next:
    atomically:
        if !q == []:
            next = None
        else:
            next = list.head(!q)
            !q = list.tail(!q)
```

(b) [code/queue.hny] Concurrent

not a good implementation because operations are O(n)

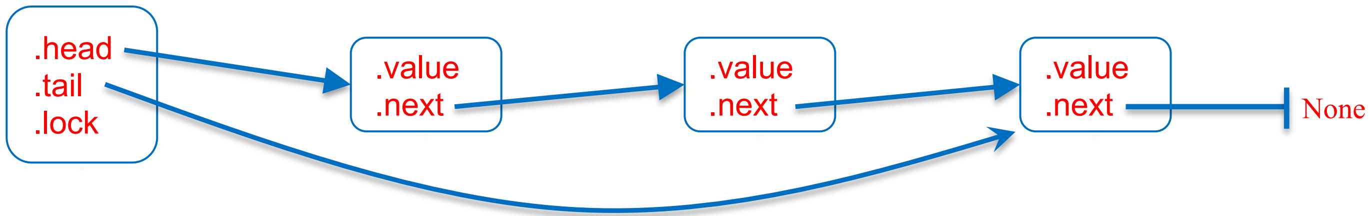
Queue implementation, v1




```
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)
```

Figure 11.3 in Harmony book

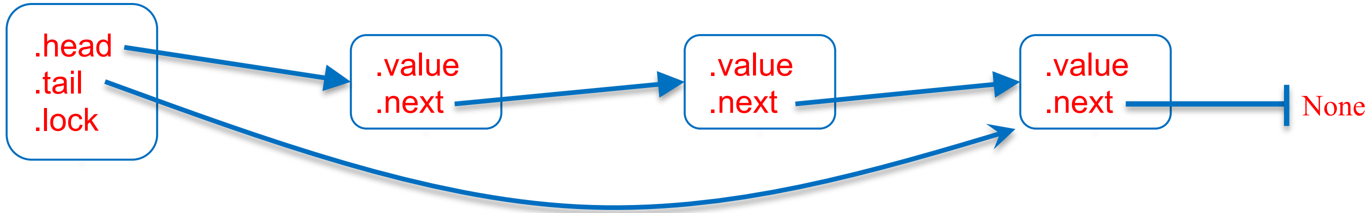
Queue implementation, v1



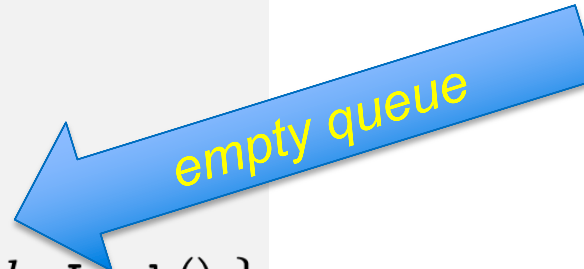

```
from synch import Lock, acquire, release
from alloc import malloc, free

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

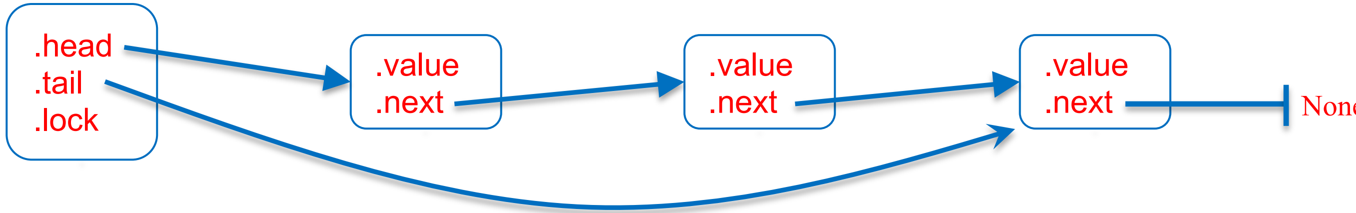
Queue implementation, v1




```
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)
```



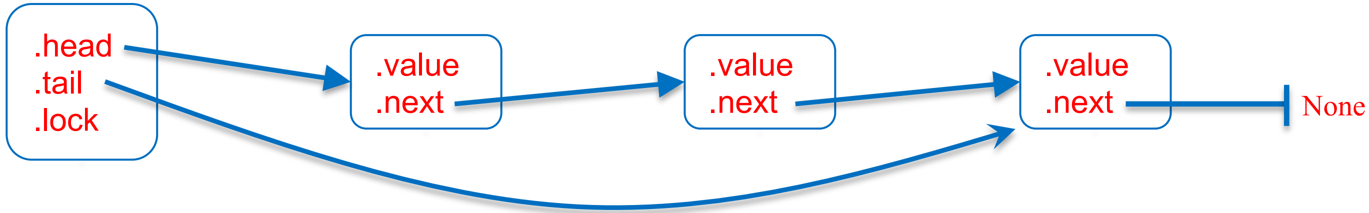
Queue implementation, v1




```
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

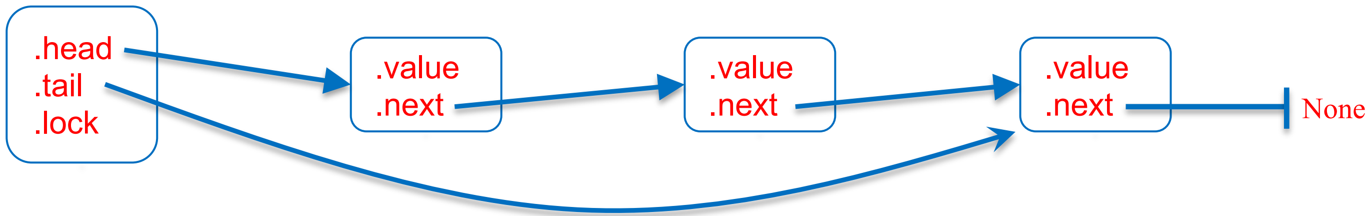
Queue implementation, v1




```
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

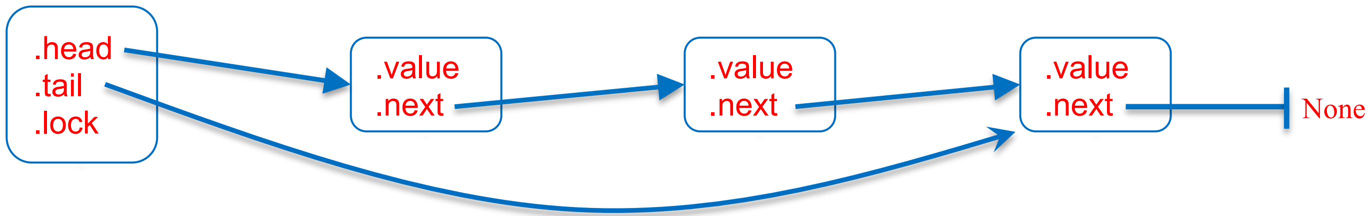



```
from synch import Lock, acquire, release
from alloc import malloc, free

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)
```

grab lock

the hard stuff

Queue implementation, v1



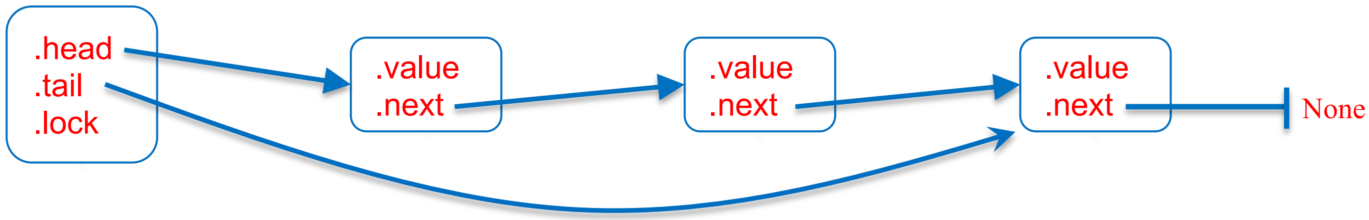

```
from synch import Lock, acquire, release
from alloc import malloc, free

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)
```

grab lock

the hard stuff

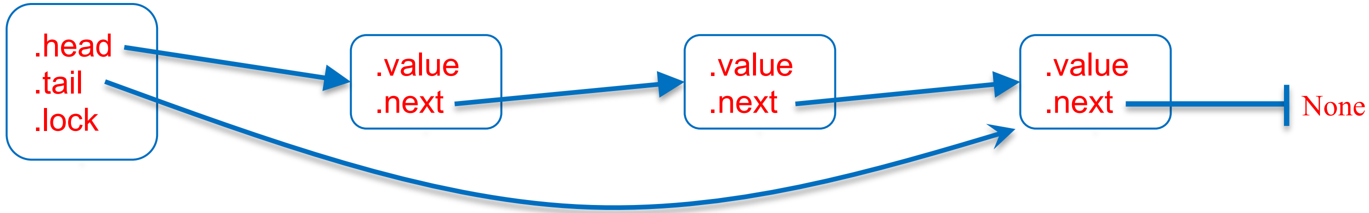
release lock

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)
```

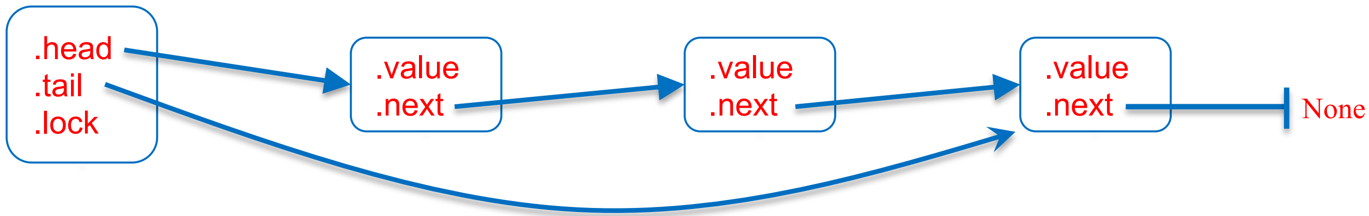
Queue implementation, v1




```
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?

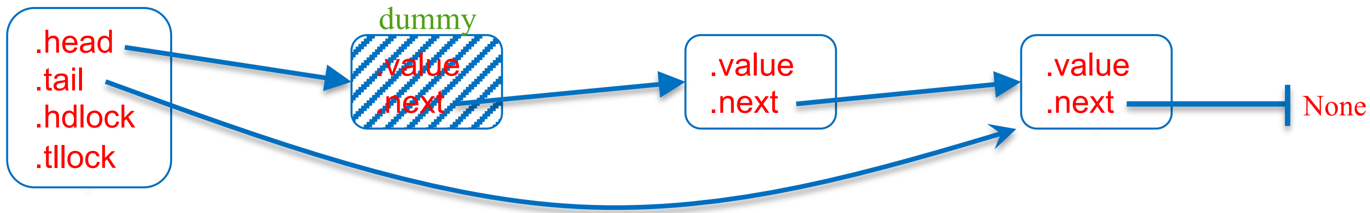
- Answer: all important
  - any resource that needs scheduling
    - CPU run queue
    - disk, network, printer waiting queue
    - lock waiting queue
  - inter-process communication
    - Posix pipes:
      - cat file | tr a-z A-Z | grep RVR
  - actor-based concurrency
  - …

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
  - …

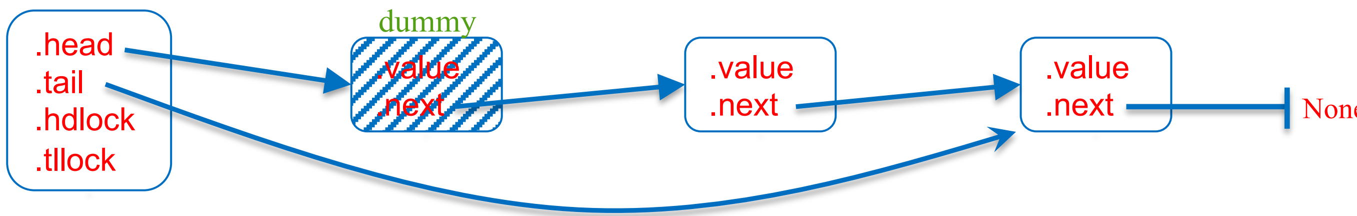
Good performance is critical!

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)
```

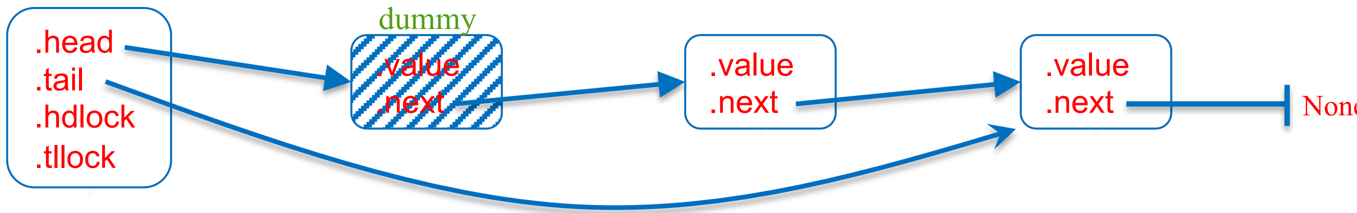
Concurrent queue v2: 2 locks




```
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)
```

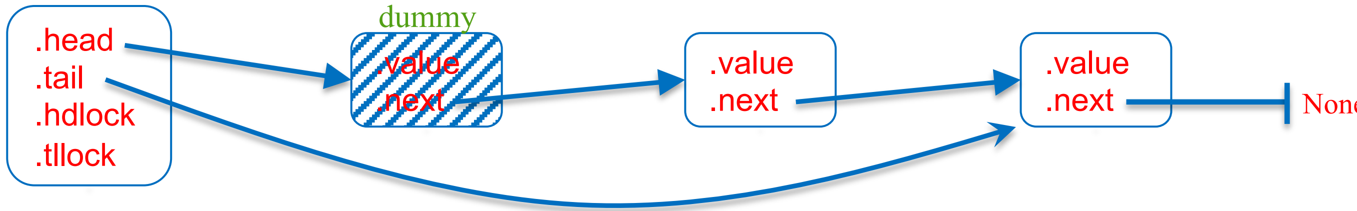
atomically q->tail->next = node

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)
```

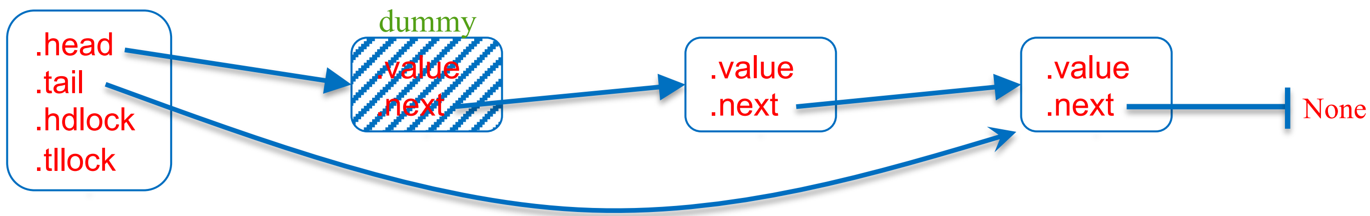
Concurrent queue v2: 2 locks




```
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)
```

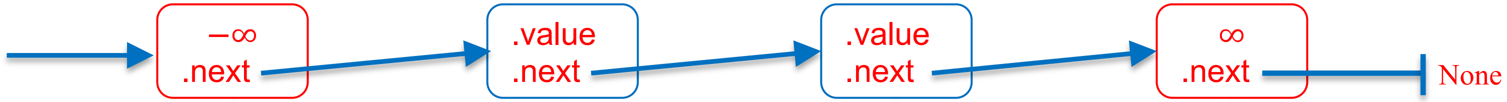
No contention for concurrent enqueue and dequeue operations!
➔ more concurrency ➔ faster

BUT: 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

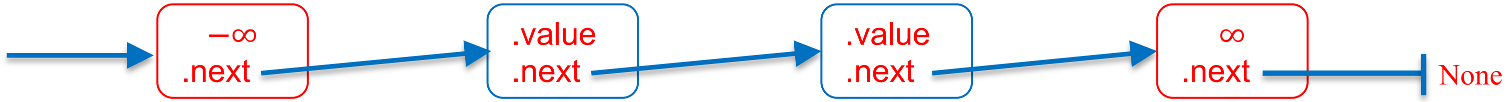
Sorted Linked List with Lock per Node




```
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))
```

empty list

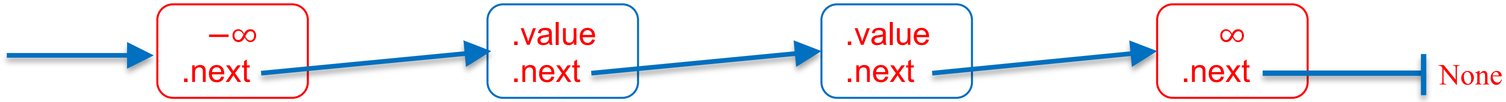
Sorted Linked List with Lock per Node




```
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

Testing a Concurrent 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)
```

Figure 11.2: [code/queuedemo.hny] Using a concurrent queue

Testing a Concurrent 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)
```

- ad hoc
- unsystematic

Figure 11.2: [code/queuedemo.hny] Using a concurrent queue

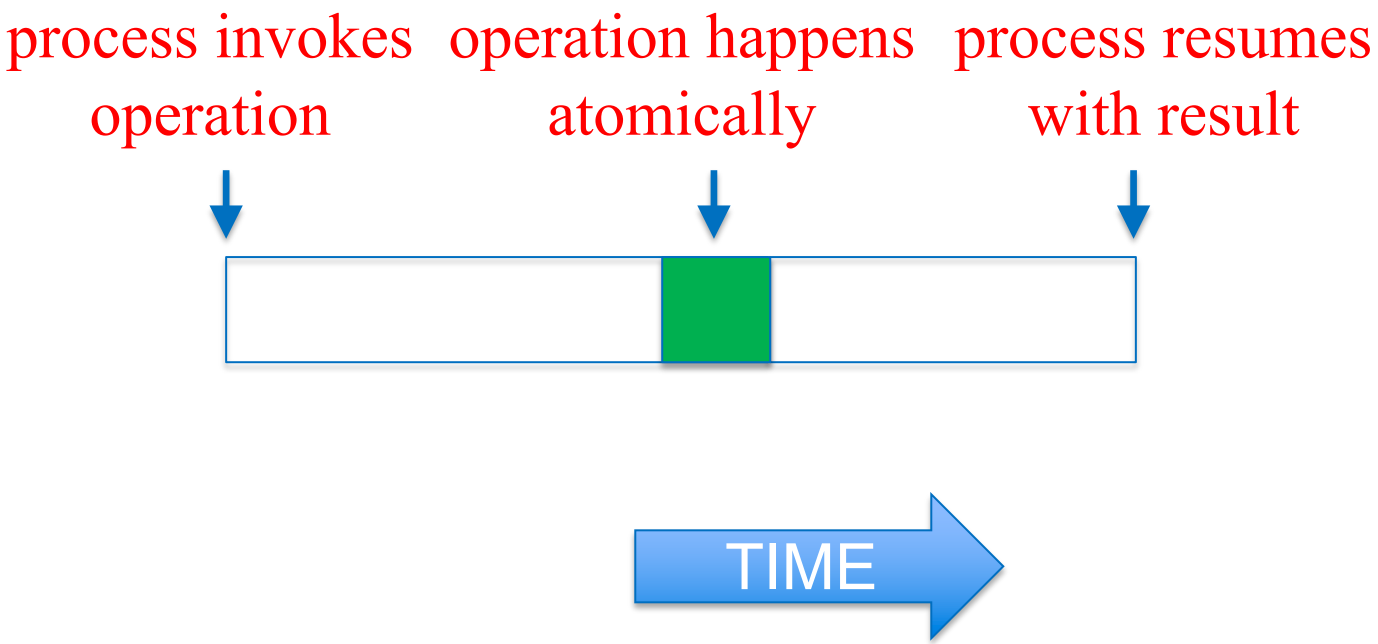
Systematic Testing

- Sequential case
  - try all "sequences" of 1 operation
    - put or get
  - 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 a sequence 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!

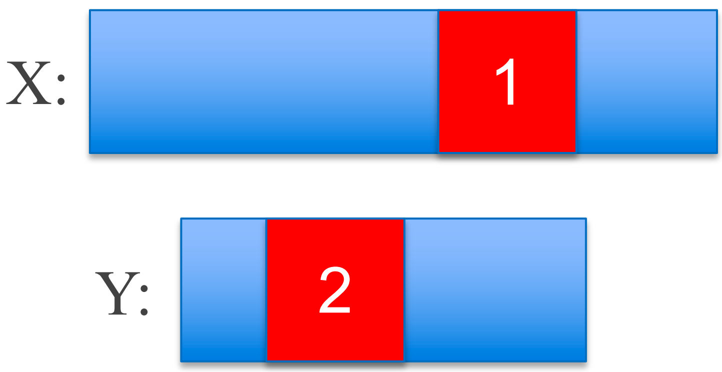
Specification

- 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:

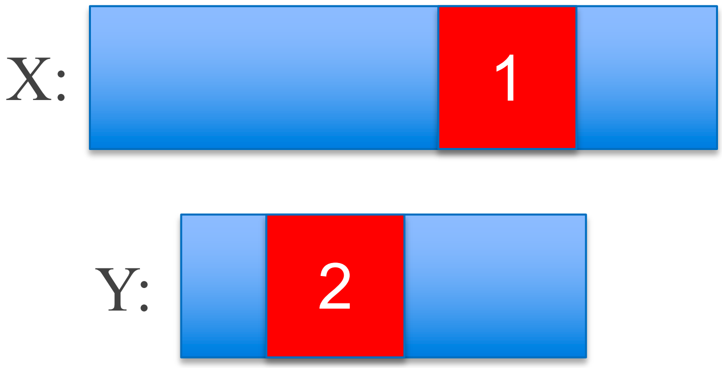
1. customer X orders burger, order ends up with cook 1
2. customer Y orders burger, order ends up with cook 2
3. cook 1 was busy with something else, so cook 2 grabs the lock first
4. cook 2 cooks burger for Y
5. cook 2 releases lock
6. cook 1 grabs lock
7. cook 1 cooks burger for X
8. cook 1 releases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

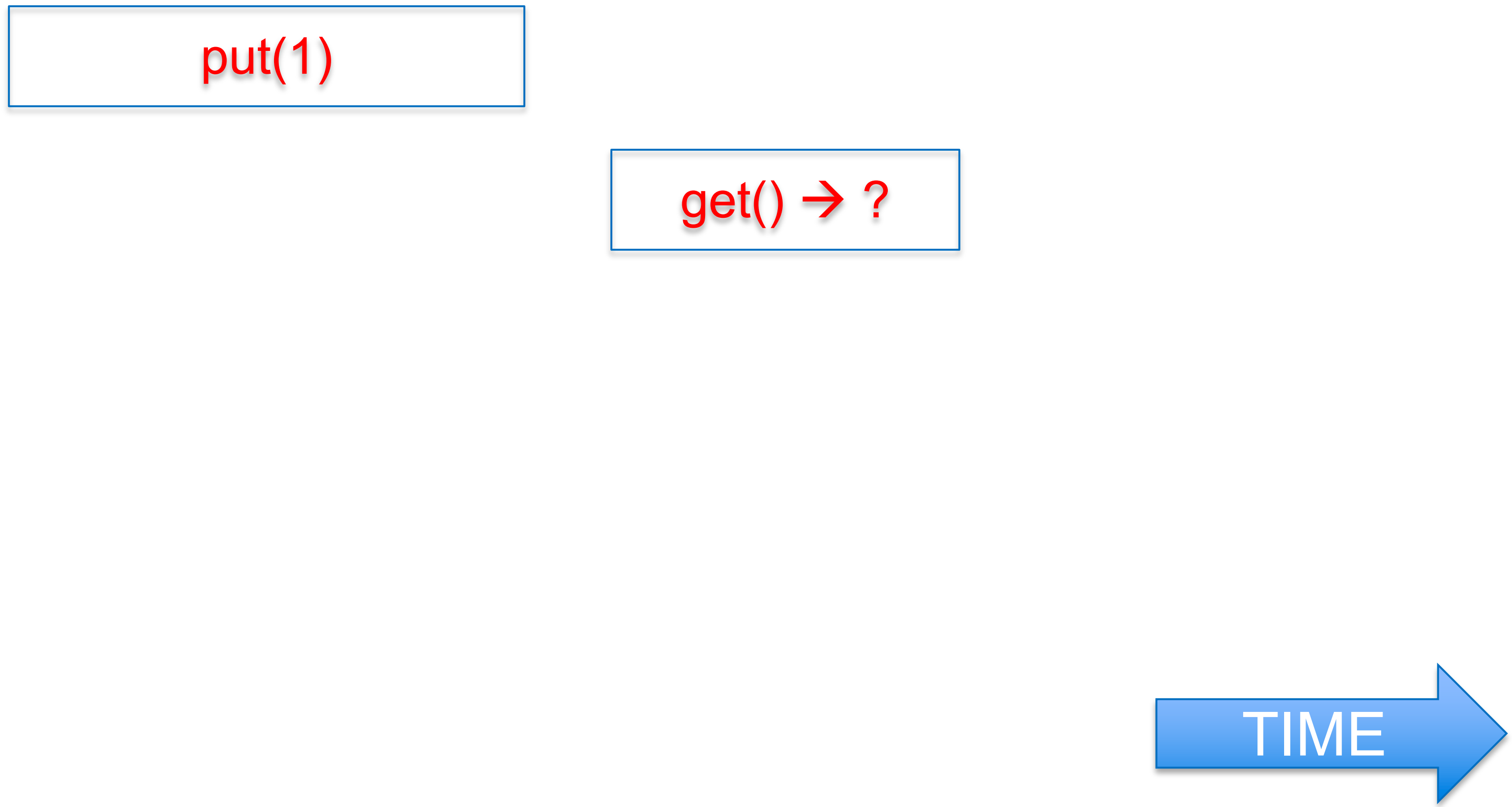
1. customer X orders burger, order ends up with cook 1
2. customer Y orders burger, order ends up with cook 2
3. cook 1 was busy with something else, so cook 2 grabs the lock first
4. cook 2 cooks burger for Y
5. cook 2 releases lock
6. cook 1 grabs lock
7. cook 1 cooks burger for X
8. cook 1 releases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



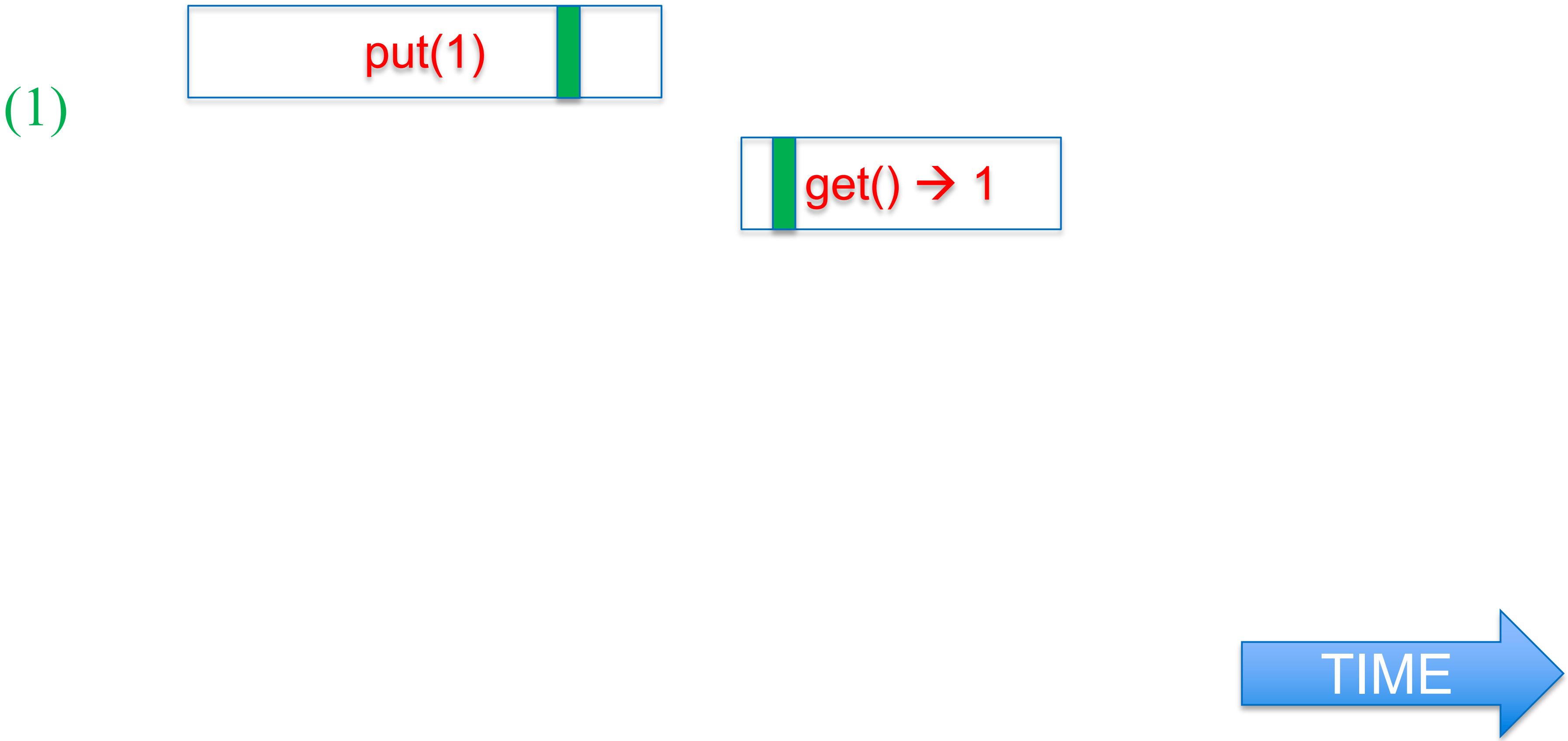
- *can't happen if Y orders burger after X receives burger*
- *but if operations overlap, any ordering can happen…*

Correct Behaviors

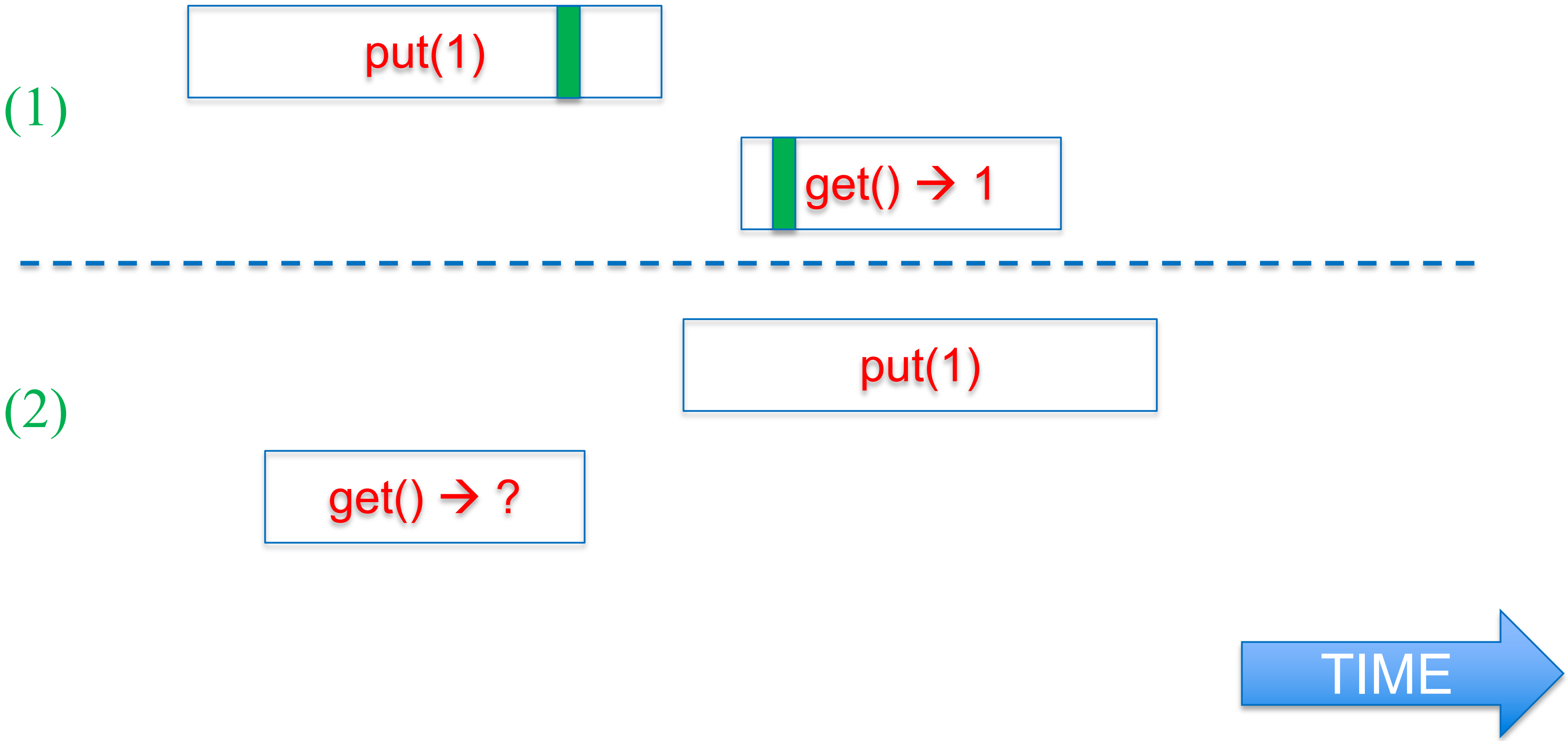
(1)



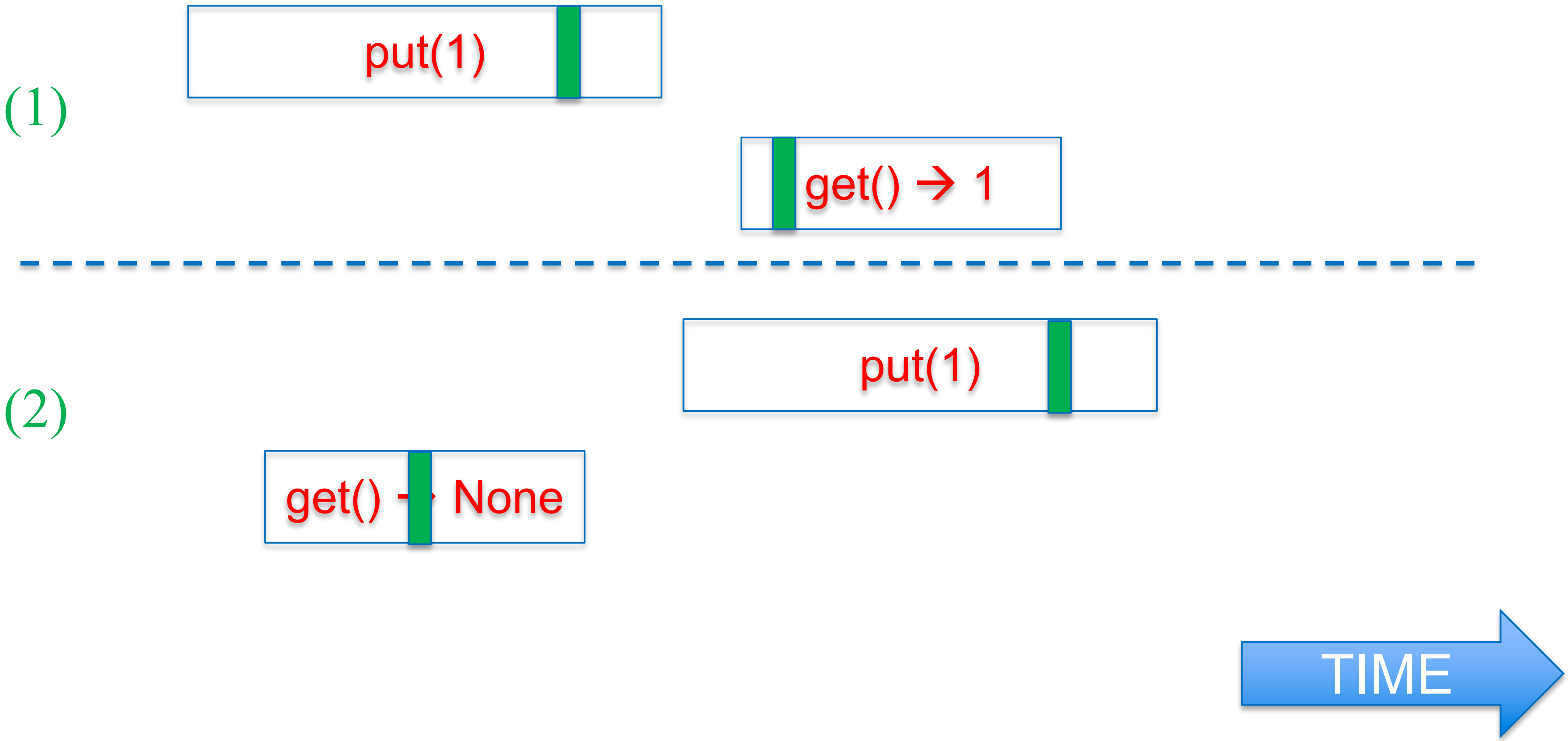
Correct Behaviors



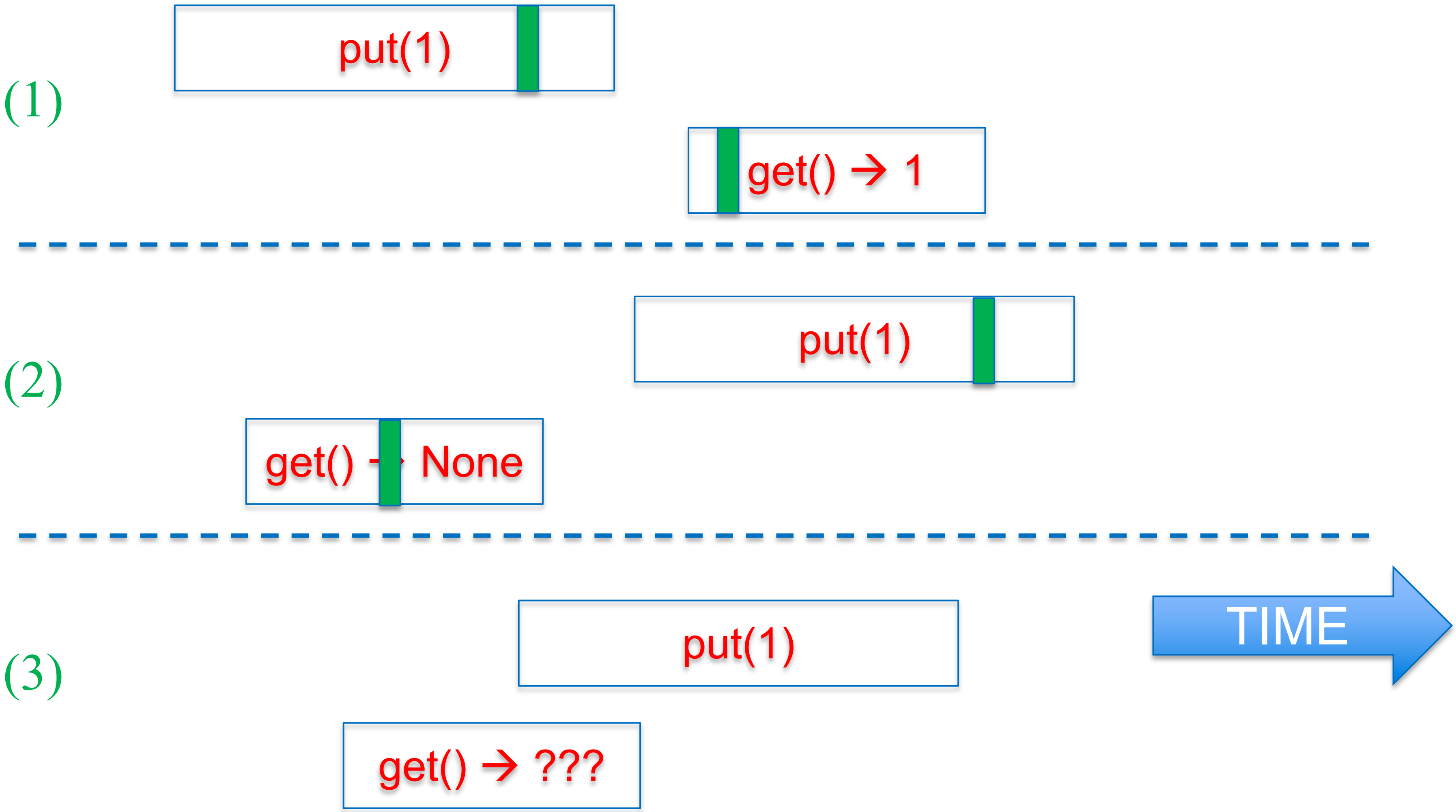
Correct Behaviors



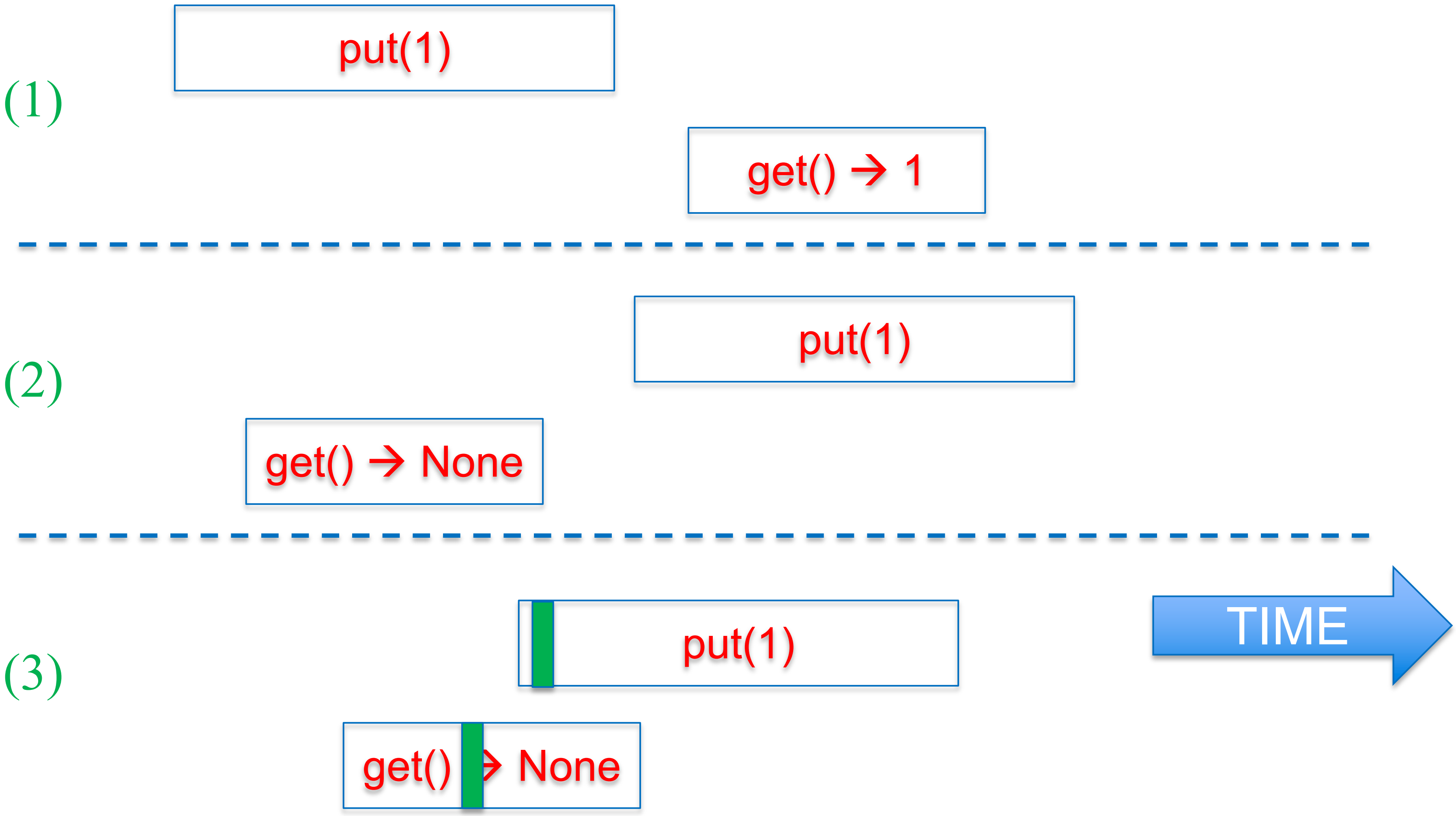
Correct Behaviors



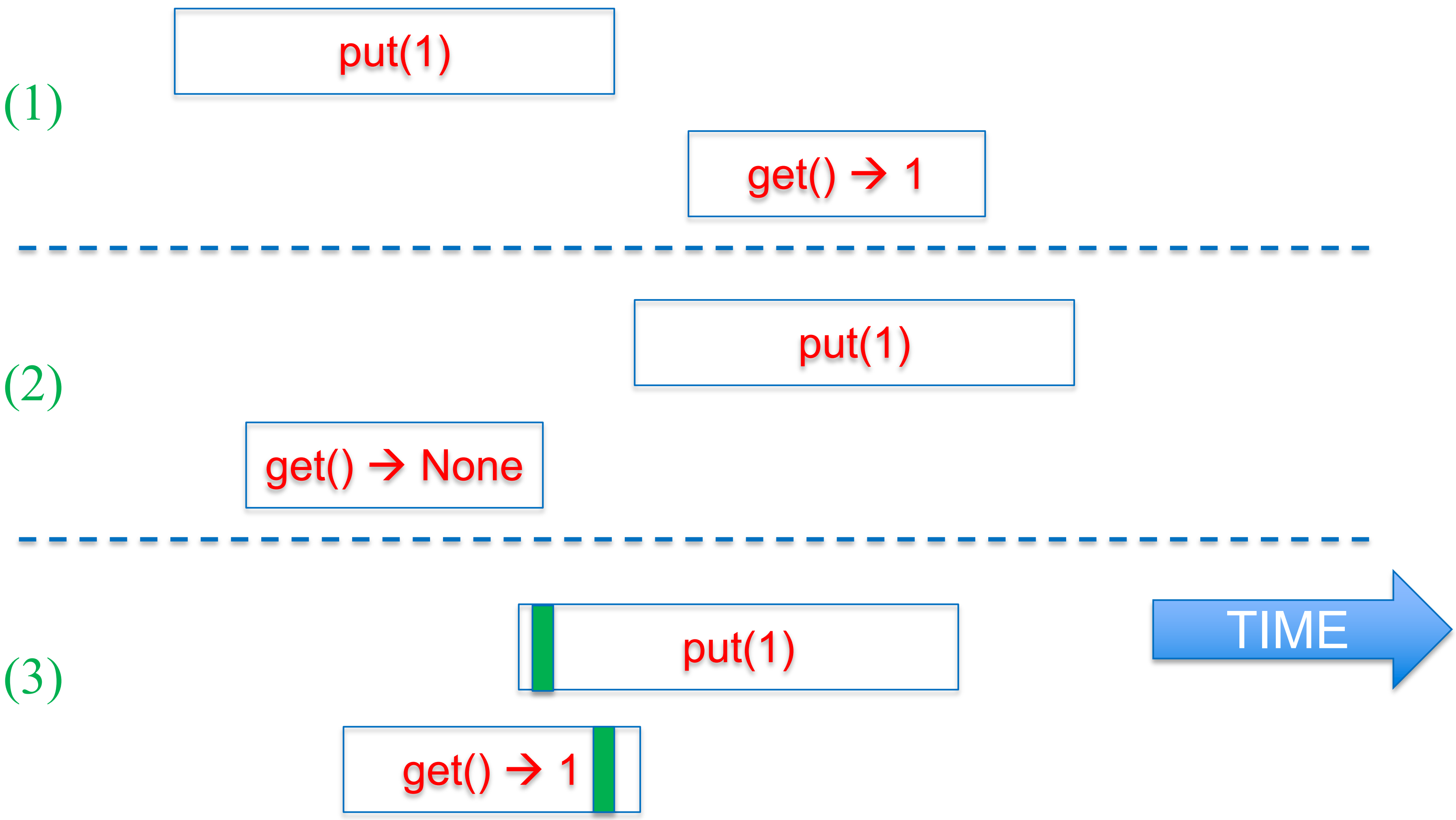
Correct Behaviors



Correct Behaviors



Correct Behaviors



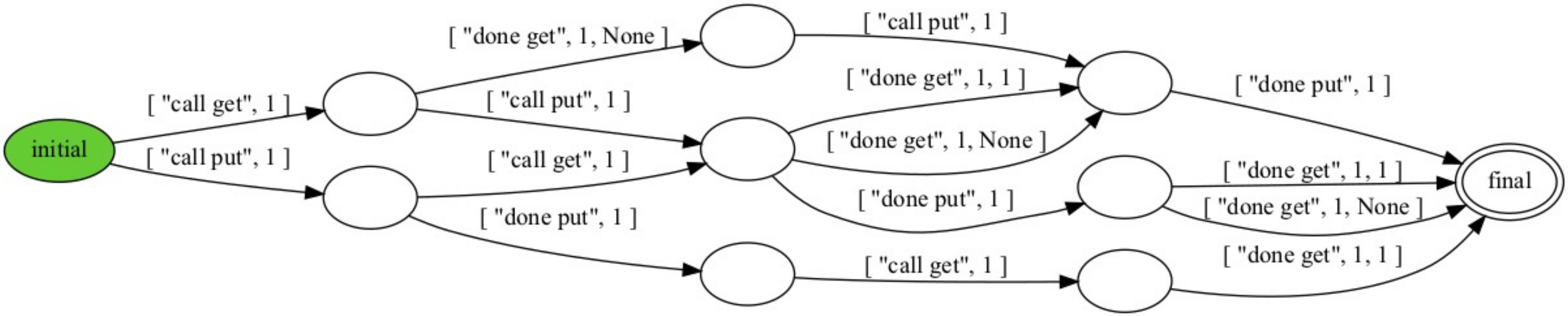
Testing Concurrent Objects

- 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 a sequence 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 -cNOPS=2 -o q.png qtestpar.hny
```

Testing: comparing behaviors

```
$ harmony -o queue4.hfa code/qtestpar.hny
$ harmony -B queue4.hfa -m queue=queueconc code/qtestpar.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