Peterson's Algorithm: Flags and Turns!

```
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 countLabel(cs) == 1

        # Leave critical section
        flags[self] = False

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

Prevents out-of-order execution

I'd like to enter...

...but you go first!

Wait until alone or it's my turn

Leave

#states = 104 diameter = 5

#components: 37

no issues found

What about a proof?

- To understand why it works...
- We need to show that, for any execution, all states reached satisfy mutual exclusion
  - i.e., that mutual exclusion is an invariant
- See the Harmony book for a proof!
  - or come talk to me!

Peterson's Reconsidered

- Mutual Exclusion can be implemented with atomic LOAD and STORE instructions
  - multiple STOREs and LOADs
- Peterson's can be generalized to more than 2 processes (as long as the number of processes is known) but it is a mess...
  - ...and even more STOREs and LOADs

Too inefficient in practice!

Peterson's even more Reconsidered!

- It assumes LOAD and STORE instructions are atomic, but that is not guaranteed on a real processor
  - Suppose x is a 64-bit integer, and you have a 32-bit CPU
  - Then $x = 0$ requires 2 STORES (and reading x two LOADs
    - because it occupies 2 words!
  - Same holds if x is a 32-bit integer, but it is not aligned on a word boundary

Concurrent Writing

- Say x is a 32 bit word @ 0x12340002
- Consider two threads, T1 and T2
  - T1: $x =$ 0xFFFFFFFF (i.e., $x = -1$)
  - T2: $x = 0$
- After T1 and T2 are done, x may be any of
  - 0, 0xFFFFFFFF, 0xFFFF0000, or 0X0000FFFF
- The outcome of concurrent write operations to a variable is undefined

Concurrent Reading

- Say x is a 32 bit word @ 0x12340002, initially 0
- Consider two 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 be any of
  - 0, 0xFFFFFFFF, 0xFFFF0000, or 0X0000FFFF
- The outcome of concurrent read and write operations to a variable is 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's "sequential" statement

- sequential $turn, flags$
- Ensures that LOADs and STOREs are atomic
  - concurrent operations appear to be executed sequentially
  - this is called sequential consistency
- Say x's current value is 3; T1 STOREs 4 into x; T2 LOADs x
  - with atomic LOAD/STORE, T2 reads 3 or 4
  - with modern CPUs/compilers, what T2 reads is undefined

Sequential Consistency

- Java has a similar notion
  - volatile int x (not the same as in C/C++)
- Loading/Storing sequentially consistent variables is more expensive than loading/storing ordinary variables
  - it restricts CPU or compiler optimizations

So, what do we do?

Interlock Instructions

- Machine instructions that do multiple shared memory accesses atomically
- TestAndSet s
  - returns the old value of s (LOAD r0,s)
  - sets s to True (STORE s, 1)
- 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

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)
```

Test..

..and set

Test and set not atomic!!

A 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)
```

Same idea as before, but now with an atomic test&set!

Lock is repeatedly "tried", checking on a condition in a tight loop ("spinning")

Locks

- Think of locks as "baton passing"
  - at most one thread can "hold" False

Specifying a Lock

```
def Lock():
    result = False

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

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

An object, and the behavior of the methods that are invoked on it

- uses atomically to specify the behavior of these methods when executed in isolation

Locks and Critical Sections

Two important invariants

- $T@\text{cs} \Rightarrow T$ holds the lock
- At most one thread can hold the lock

Implementing* a lock

*Just one way of doing so

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

def Lock():
    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 use an atomic STORE instruction

Specification

```
def Lock():
    result = False

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

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

What an abstraction does

Implementation

```
Def Lock()
    result = False

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

def atomic_store(var, val):
    atomically !var = val

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

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

How the abstraction does it

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 what if two threads are on the same core?
  - when there is no preemption?
    - all threads may get stuck while one is trying to obtain the spinlock
  - when there is preemption?
    - still delays and a waste of CPU cycles while a thread is trying to obtain a spinlock

Beyond Spinlocks

- We would like to be able to suspend a thread that is trying to acquire a lock that is being held
  - until the lock is ready
- A context switch!

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 (p must be a pointer).
  - go (!p) r
    - adds a thread with the given context (i.e., the one pointed by p) to the bag of threads. Threads resumes from stop expression, returning r

Lock specification using stop and go

```
import list

def Lock():
    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

add stopped context at the end of queue associated with lock

restart thread at head of queue and remove it from queue

Lock specification using stop and go

```
import list

def Lock():
    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)
```

Similar to Linux "futex": with no contention (hopefully the common case) acquire() and release() are cheap. With contention, a context switch is required

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 the 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 instruction	Good for implementing concurrent data structures

Using Locks

- Data structures maintain some invariant
  - Consider a linked list
    - There is a head, a tail, and a list of nodes such as the head points to the first node, tail points to the last one, and each node points to the next one, except for the tail, which points to None. However, if the list is empty, head and tail are both None
- You can assume the invariant holds right after acquiring the lock
- You must make sure invariant holds again right before releasing the lock

Building a Concurrent Queue

- q = queue.new(): allocates a new queue
- queue.put(q, v): adds 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():
    result = []

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

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

Sequential

```
import list

def Queue():
    result = []

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

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

Concurrent

Example of using a Queue

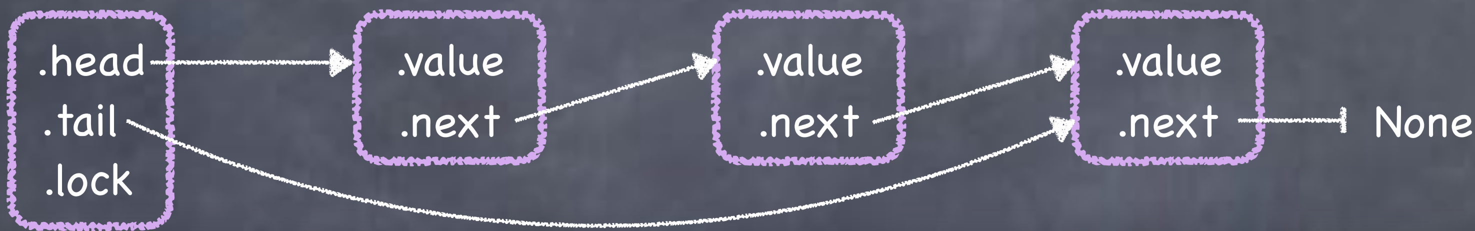
```
import queue

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 a queue

Queue implementation, v1




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

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

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

dynamic memory allocation

create empty queue

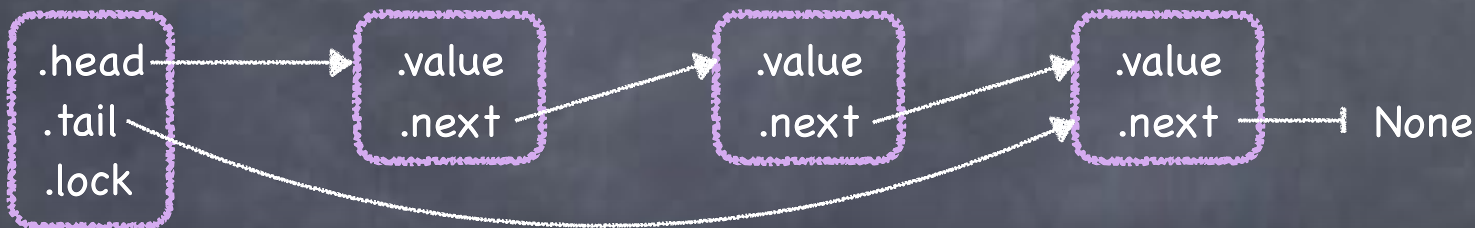
allocate node

grab lock

The Hard Stuff

release lock

Queue implementation, v1




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

grab lock

empty queue

The Hard Stuff

free dynamically allocated memory

release lock

How important are concurrent queues?

- All important!
  - any resource that needs scheduling
    - CPU ready queue
    - disk, network, printer waiting queue
    - lock waiting queue
  - inter-process communication
    - Posix pipes: cat file | sort
  - actor-based concurrency
  - ...

Performance is critical!

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

Systematic Testing

- Sequential case:
  - Try all sequences consisting of 1 operation
    - put or get
  - Try all sequences consisting of 2 operations
    - put+put, put+get, get+put, get+get
  - Try all sequences consisting of 3 operations
  - ...

How do we know if a sequence is correct?

- We run the test program against both the specification and the implementation
- We then perform the same sequence of operations using the code in both sequential specification and the implementation and check if these sequences produce the same behaviors (e.g., they return the same values)

Systematic Testing

- Concurrent case:
  - Can't run same sequence of operations on both
    - even if both are correct, nondeterminism of concurrency may have the two run produce different results
  - Instead:
    - Try all interleavings of 1 operation
    - Try all interleavings in a sequence of 2 ops
    - Try all interleavings in a sequence of 3 ops
    - ...

How do we know if a sequence is correct?

- We run the test program against both the specification and the implementation
  - this produces two DFAs, which capture all possible behaviors of the program
- We then verify whether the DFA produced running against the specification is the same as the one produced running against the implementation

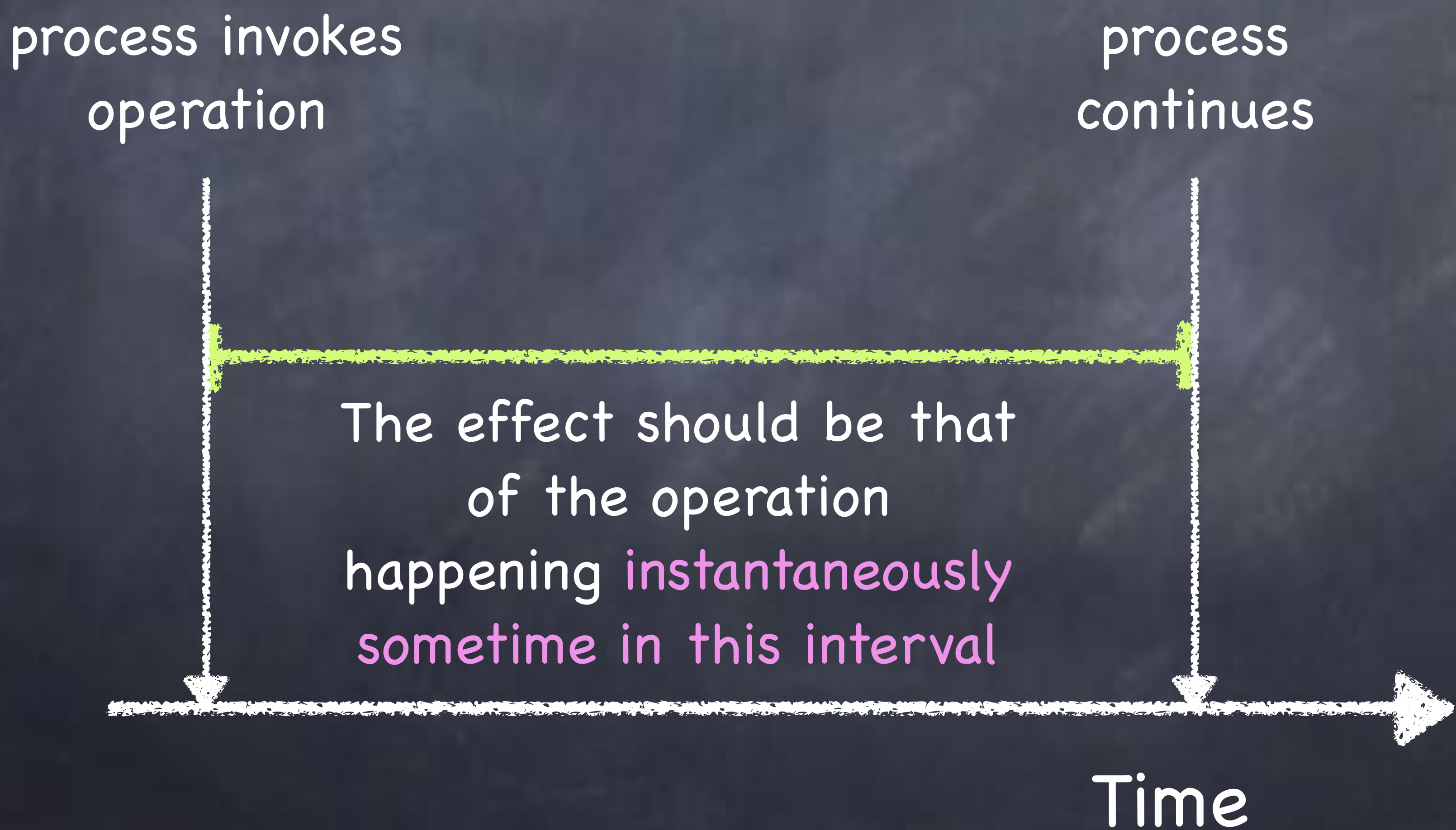
Queue test program

```
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)
```

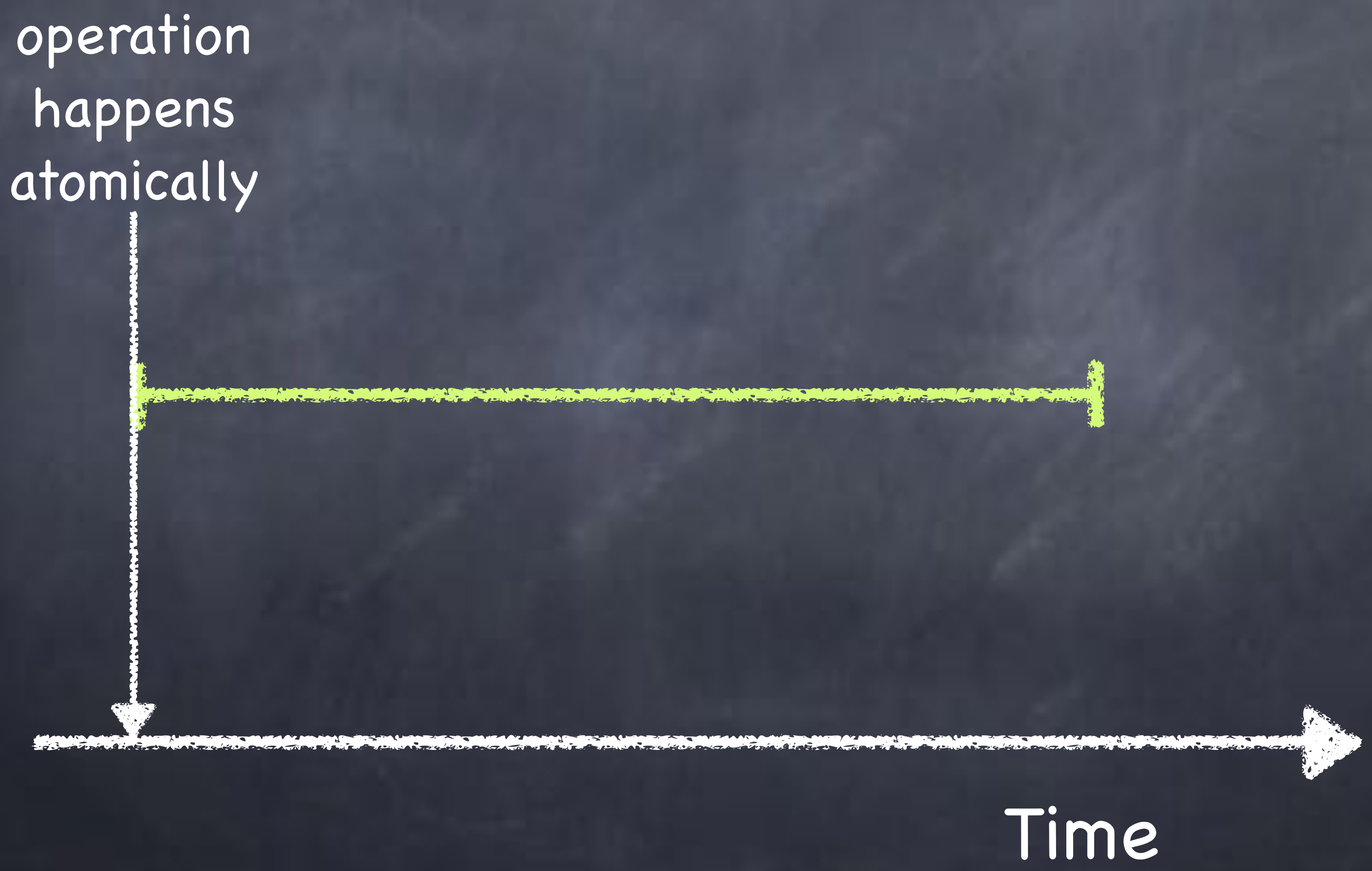
* always at least one put and one get

NOPS threads, nondeterministically choosing* to execute put or get

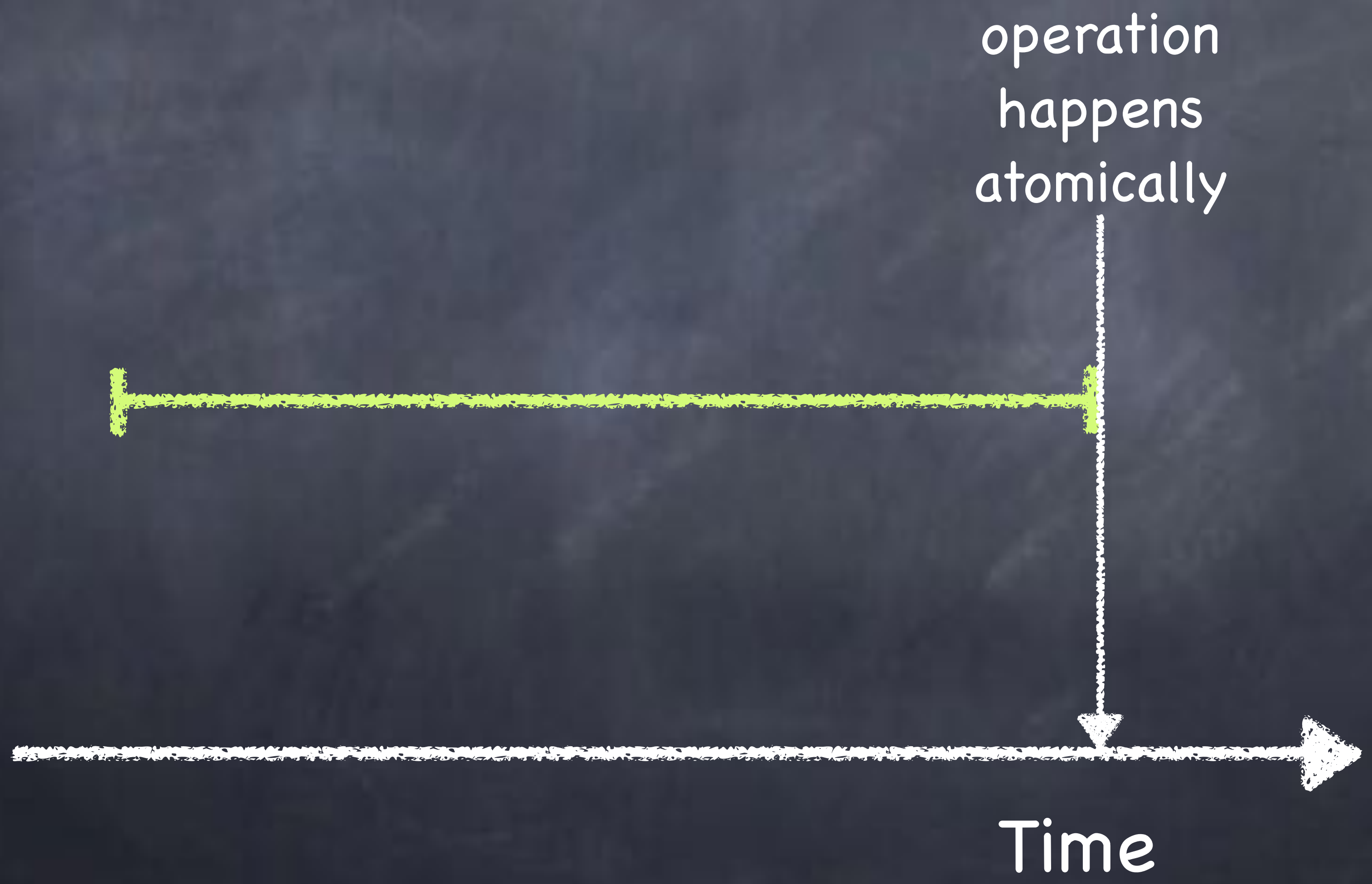
Life of an Atomic Operation



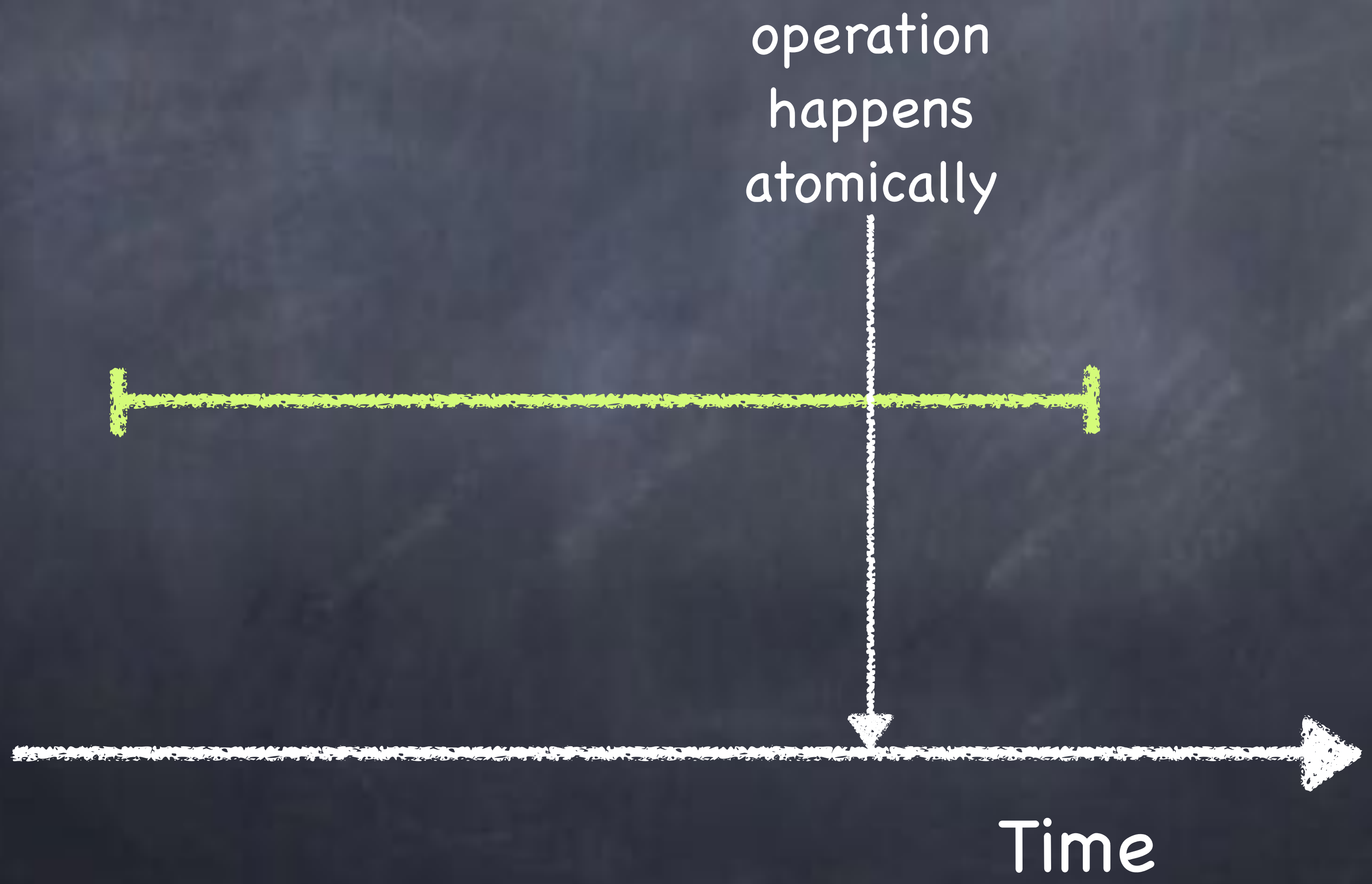
Life of an Atomic Operation



Life of an Atomic Operation

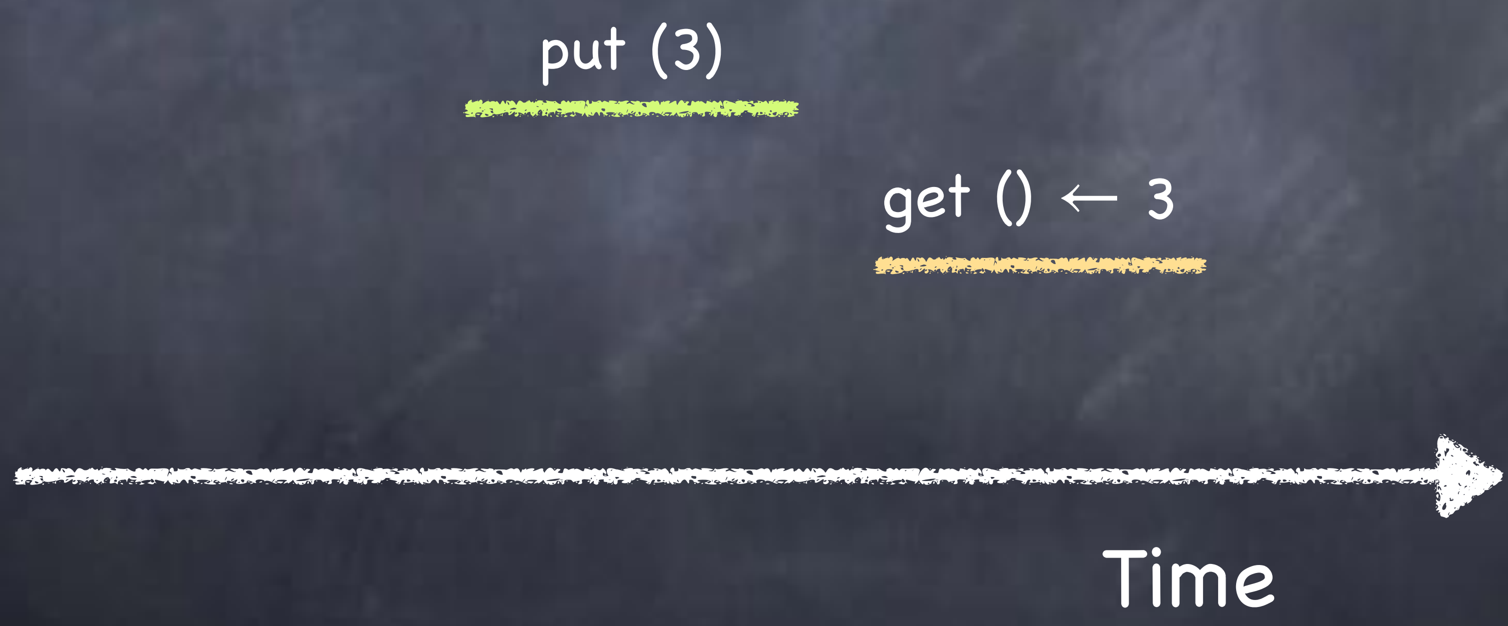


Life of an Atomic Operation



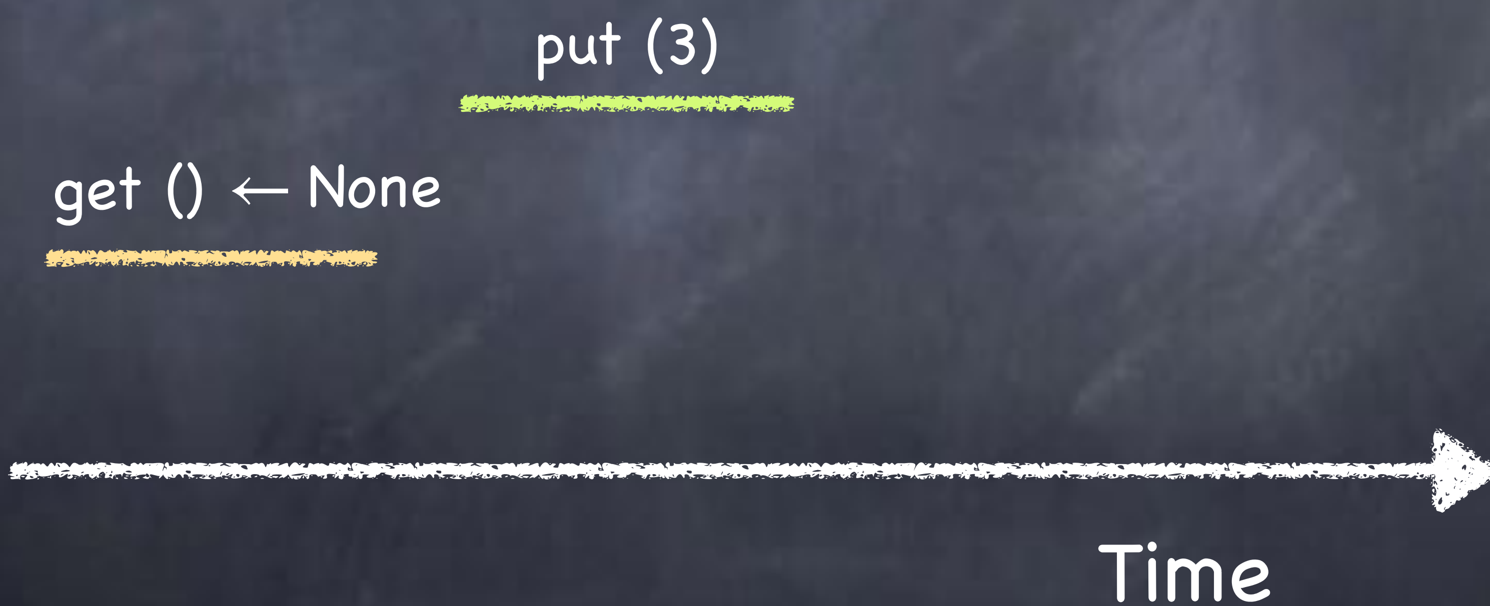
Correct Behaviors

Suppose the queue is initially empty



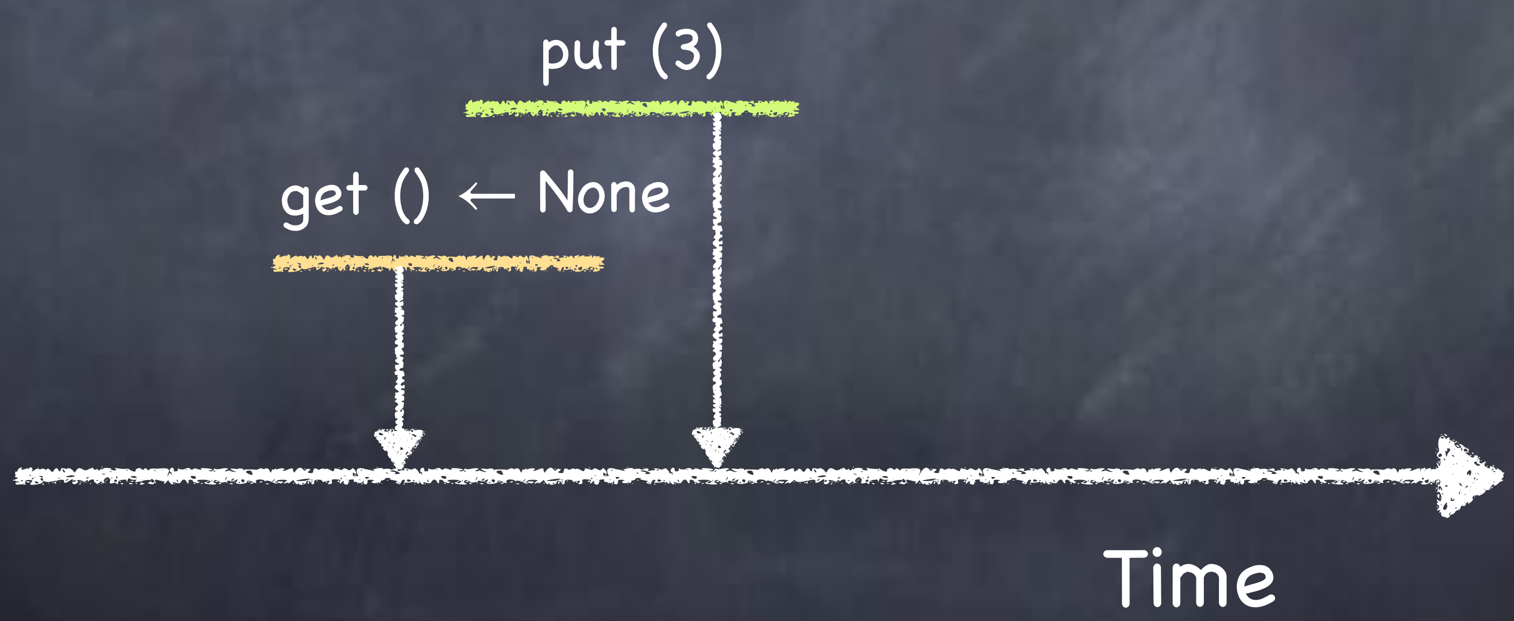
Correct Behaviors

Suppose the queue is initially empty



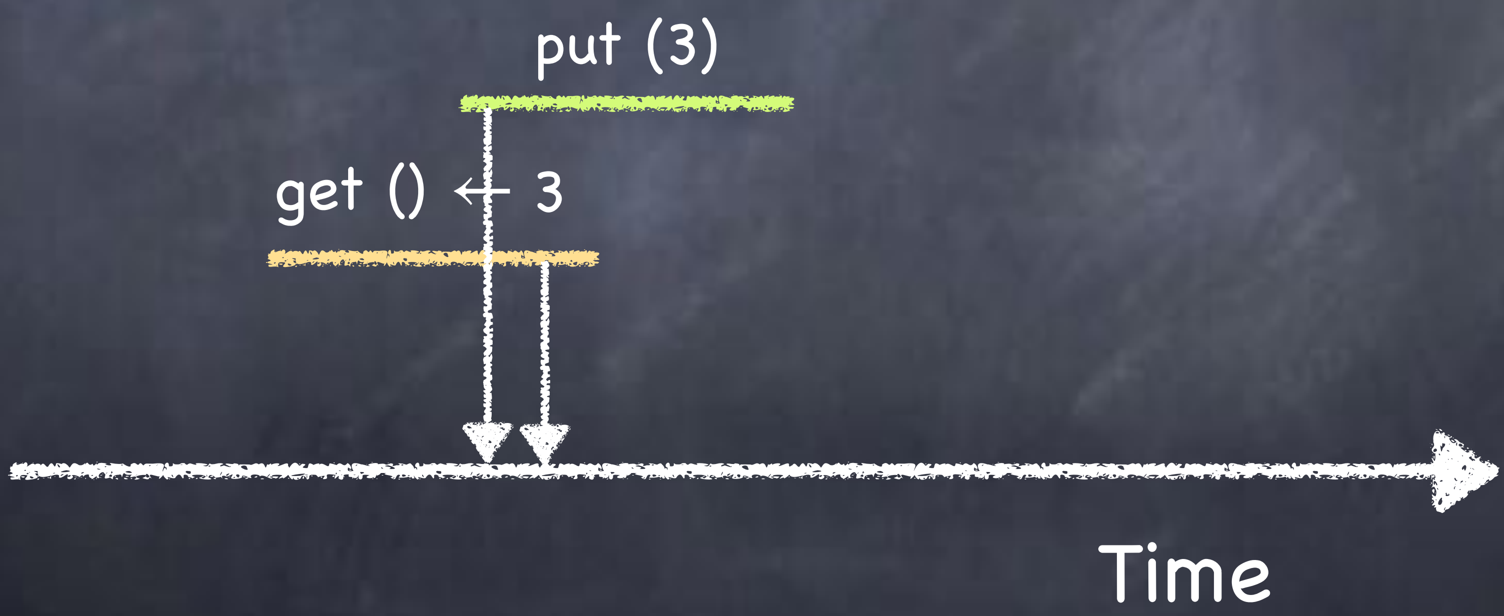
Correct Behaviors

Suppose the queue is initially empty

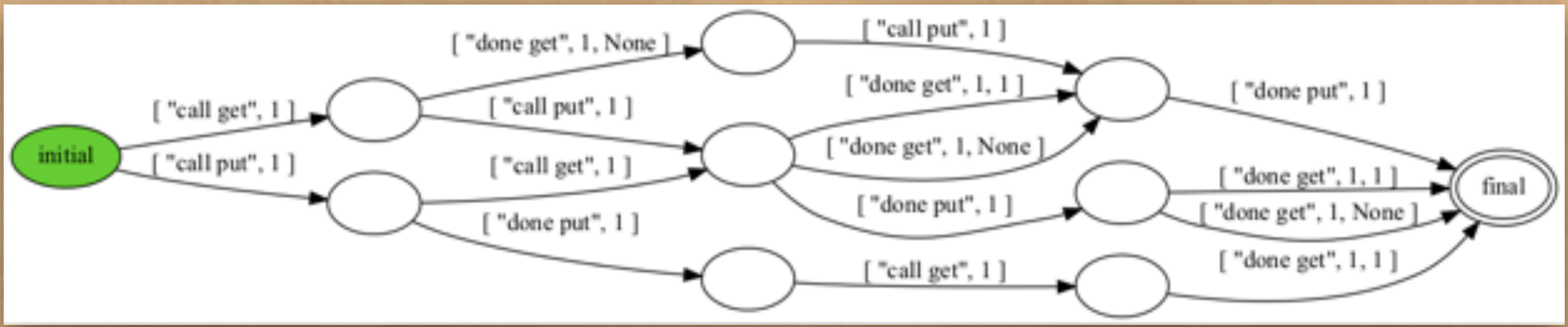


Correct Behaviors

Suppose the queue is initially empty



Queue test program



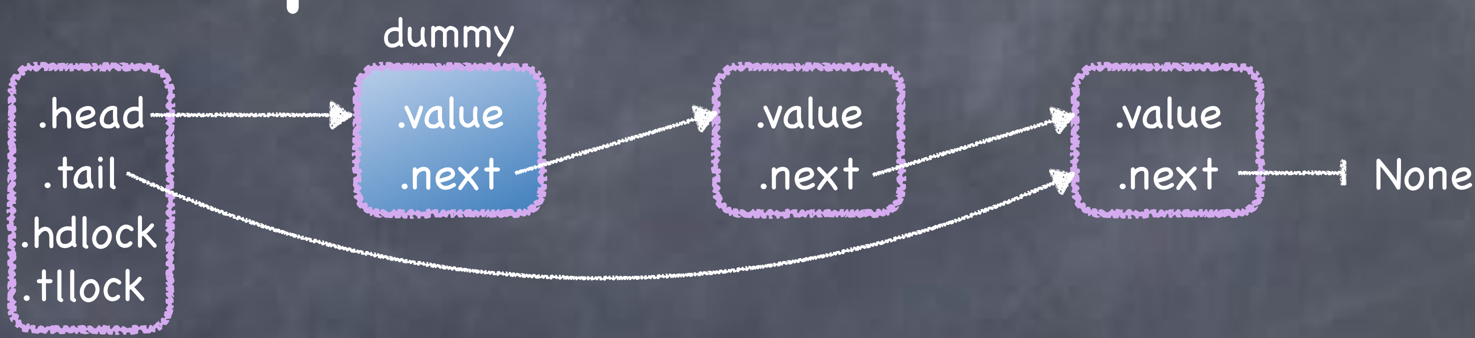

```
$ harmony -c NOPS=2 -o spec.png code/qtestpar.hny
```

Testing: comparing behaviors

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

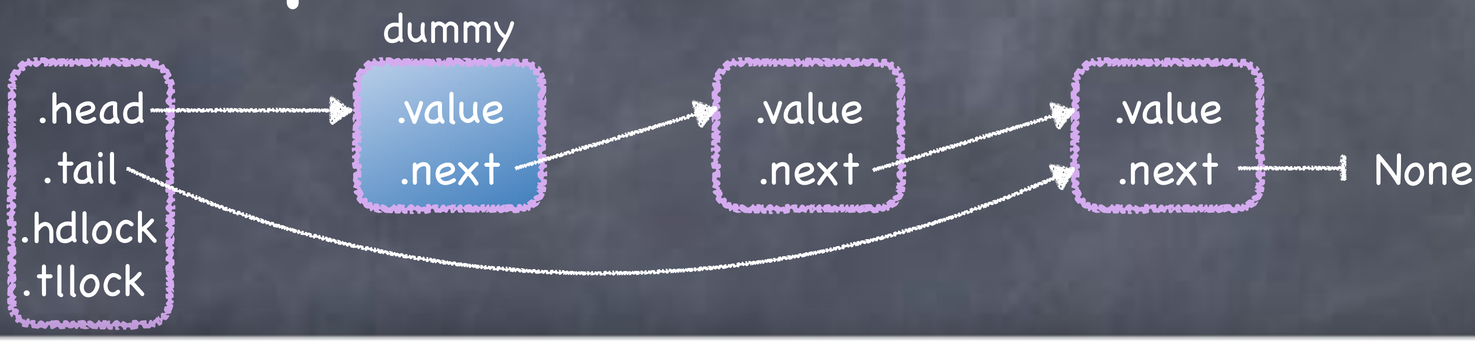
- The first command outputs the behavior of the running test program against the specification in file queue4.hfa
- The second command runs the test program against the implementation and checks if its behavior matches that stored in queue4.hfa

Queue implementation, v2:2 locks



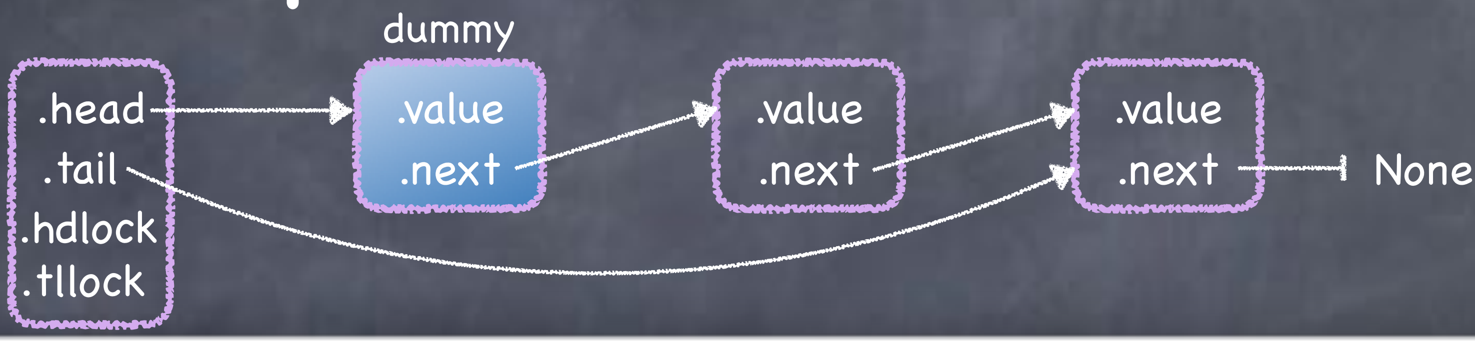
- Separate locks for head and tail
  - put and get can proceed concurrently
- Trick: a dummy node at the head of the queue
  - last node to be dequeued (except at the beginning)
  - head and tail never None

Queue implementation, v2:2 locks




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

def Queue():
    let dummy = malloc({ .value: (), .next: None }):
        result = { .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)
```

Queue implementation, v2:2 locks




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

Faster!

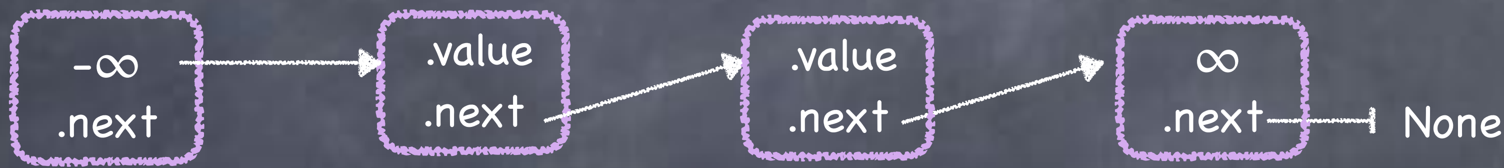
No contention for concurrent enqueue and dequeue ops ⇒ more concurrency

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 fine-grain locking
- A global lock is easy, but limits concurrency
- Fine-grain (local) locks can improve concurrency, but tend to be tricky to get right

Sorted lists with lock per node



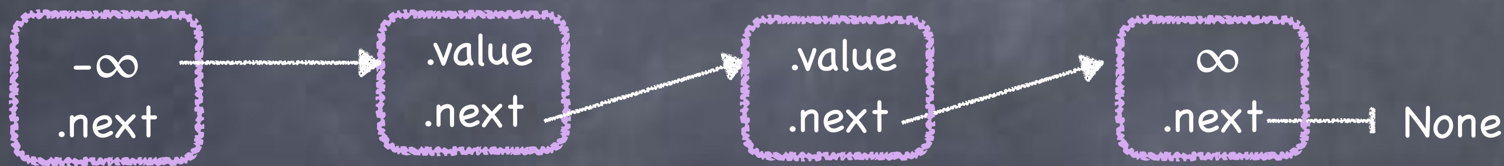

```
from synch import Lock, acquire, release
from alloc import malloc, free

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→value < v ≤ after→value

empty list:

Sorted lists with lock per node

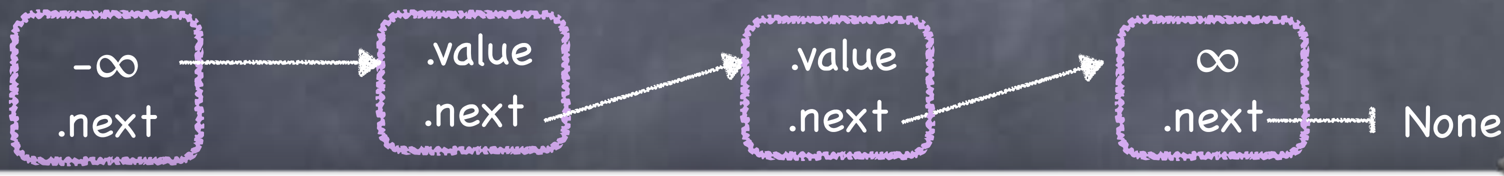



```
from synch import Lock, acquire, release
from alloc import malloc, free

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))
```

Hand-over-hand locking

empty list:

Sorted lists 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!