

Specifying a Lock

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

Implementing* a lock

*Just one way of doing so

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

*Specification of the
CPU's test-and-set
functionality*

*Must use an atomic
STORE instruction*

Specification

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

What an abstraction
does

Implementation

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

How the abstraction
does it

Using a lock for a critical section

```
1  import synch
2
3  const NTHREADS = 2
4
5  lock = synch.Lock()
6
7  def thread():
8      while choose({ False, True }):
9          synch.acquire(?lock)
10         cs: assert countLabel(cs) == 1
11         synch.release(?lock)
12
13  for i in {1..NTHREADS}:
14      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!

Support for context switching in Harmony

- Harmony allows contexts to be saved and restored (i.e., enables a 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

```
1  import list
2
3  def Lock():
4      result = { .acquired: False, .suspended: [] }
5
6  def acquire(lk):
7      atomically:
8          if lk→acquired:
9              stop ?lk→suspended[len lk→suspended]
10             assert lk→acquired
11         else:
12             lk→acquired = True
13
14  def release(lk):
15      atomically:
16          assert lk→acquired
17          if lk→suspended == []:
18              lk→acquired = False
19          else:
20              go (list.head(lk→suspended)) ()
21              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

```
1  import list
2
3  def Lock():
4      result = { .acquired: False, .suspended: [] }
5
6  def acquire(lk):
7      atomically:
8          if lk→acquired:
9              stop ?lk→suspended[len lk→suspended]
10             assert lk→acquired
11         else:
12             lk→acquired = True
13
14  def release(lk):
15      atomically:
16          assert lk→acquired
17          if lk→suspended == []:
18              lk→acquired = False
19          else:
20              go (list.head(lk→suspended)) ()
21              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 \neq Critical Section

Atomic Section	Critical Section
Only one thread can execute	Multiple threads can execute concurrently, just not within a critical section
Rare programming language paradigm	Ubiquitous: 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

```
1  import list
2
3  def Queue():
4      result = []
5
6  def put(q, v):
7      !q = list.append(!q, v)
8
9  def get(q):
10     if !q == []:
11         result = None
12     else:
13         result = list.head(!q)
14         !q = list.tail(!q)
15
```

Sequential

```
1  import list
2
3  def Queue():
4      result = []
5
6  def put(q, v):
7      atomically !q = list.append(!q, v)
8
9  def get(q):
10     atomically:
11         if !q == []:
12             result = None
13         else:
14             result = list.head(!q)
15             !q = list.tail(!q)

```

Concurrent

Example of using a Queue

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

enqueue v onto q

dequeue and check

create a queue

Queue implementation, v1



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

dynamic memory allocation

```
3
4 def Queue():
```

```
5     result = { .head: None, .tail: None, .lock: Lock() }
```

create empty queue

```
6
7 def put(q, v):
```

```
8     let node = malloc({ .value: v, .next: None }):
```

allocate node

```
9         acquire(?q->lock)
```

grab lock

```
10        if q->head == None:
```

```
11            q->head = q->tail = node
```

```
12        else:
```

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

```
14            q->tail = node
```

```
15        release(?q->lock)
```

The Hard Stuff

release lock

Queue implementation, v1



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

grab lock

empty queue

*The Hard
Stuff*

free dynamically allocated memory

release lock

How important are concurrent queues?

👁 All important!

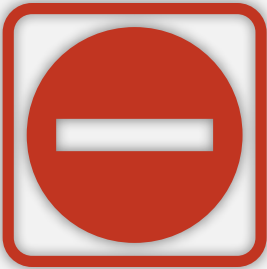
- ❑ any resource that needs scheduling
 - ▶ CPU ready queue
 - ▶ disk, network, printer waiting queue
 - ▶ lock waiting queue
- ❑ inter-process communication
 - ▶ Posix pipes: `cat file | sort`
- ❑ actor-based concurrency
- ❑ ...



Performance
is
critical!

Testing a Concurrent Queue?

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



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

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

** always at least one
put and one get*

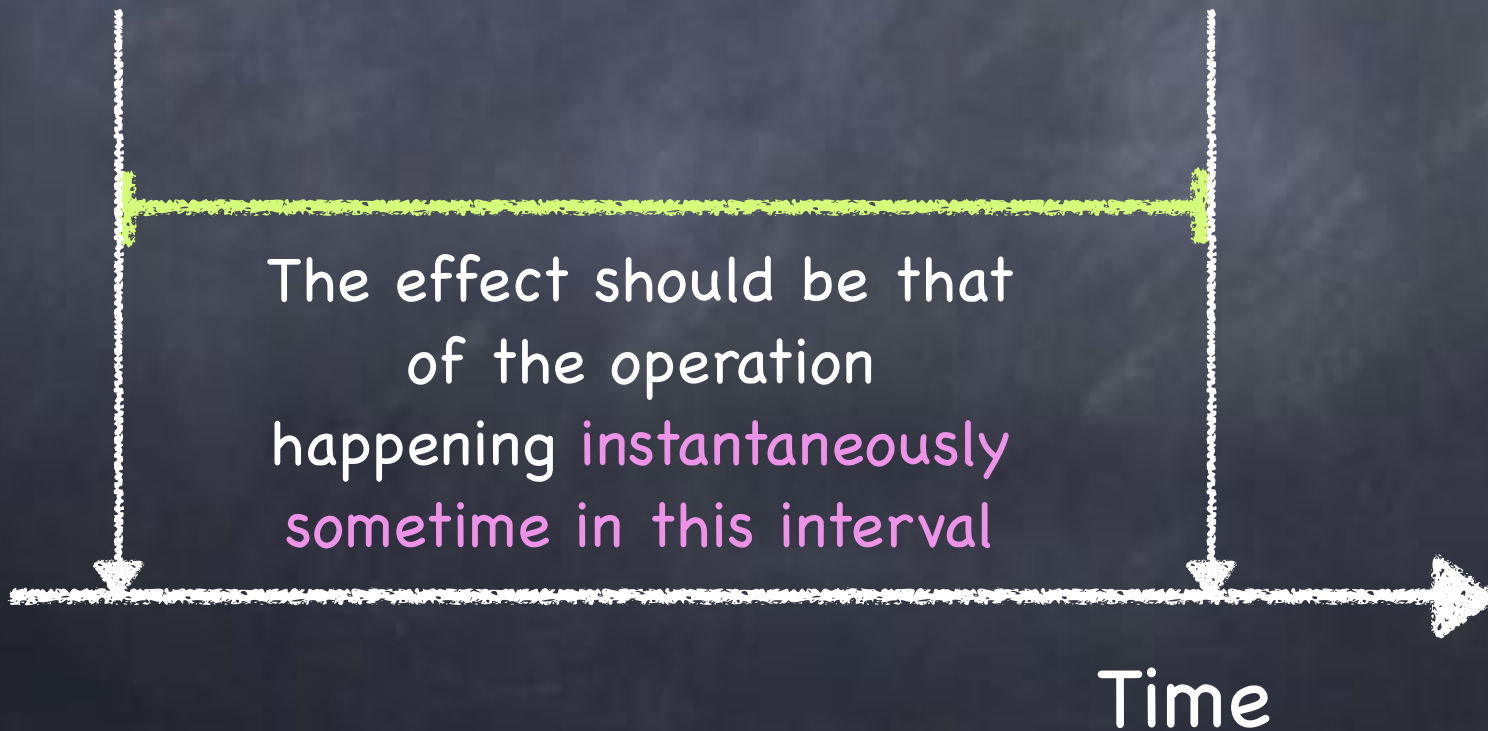
*NOPS threads,
nondeterministically
choosing* to execute
put or get*

But which behaviors
of the implementation
are **correct**?

Life of an Atomic Operation

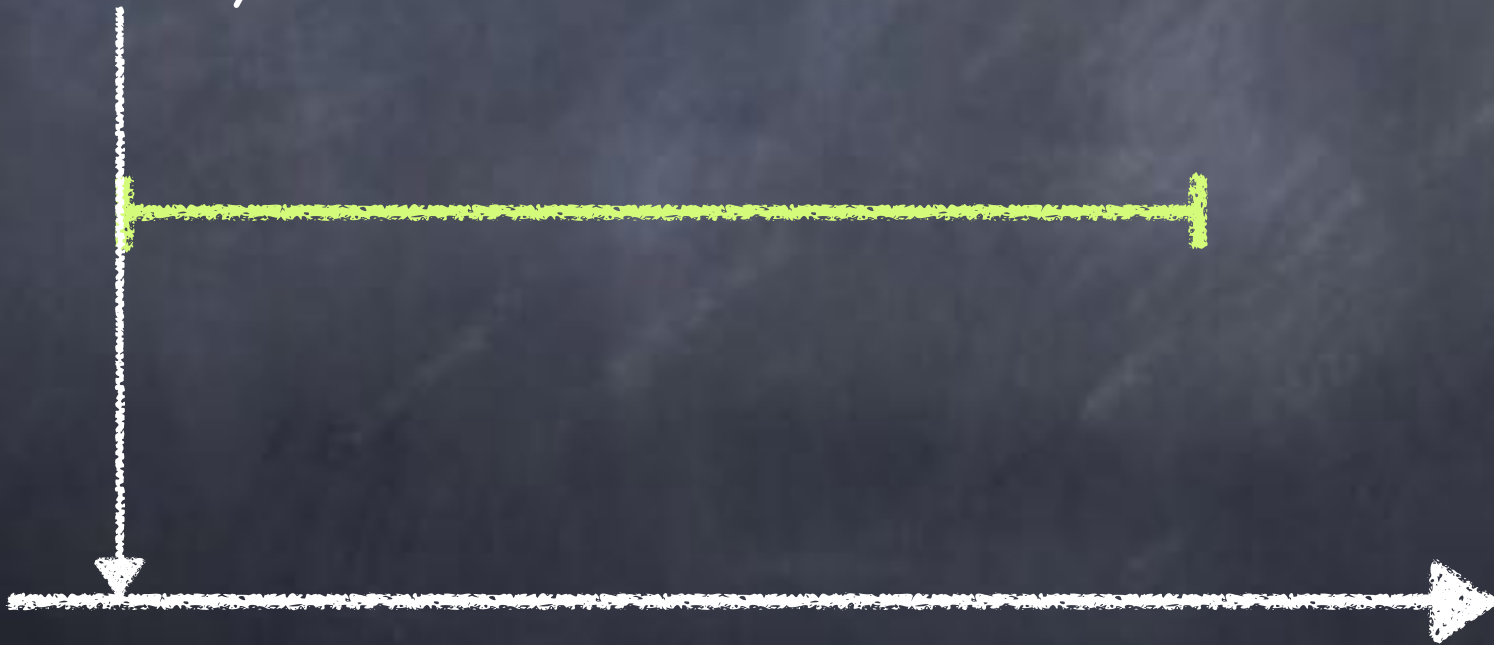
process invokes
operation

process
continues



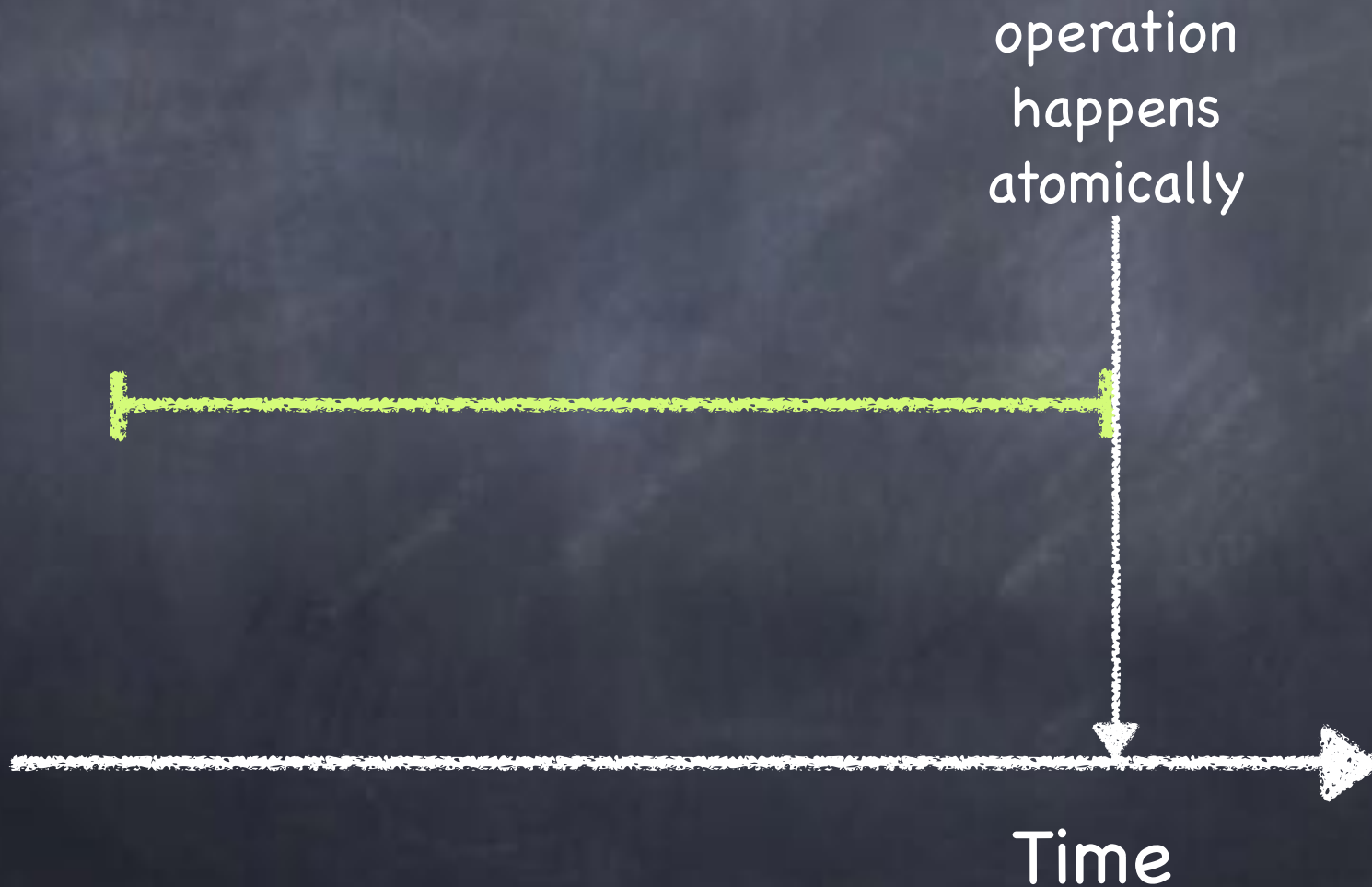
Life of an Atomic Operation

operation
happens
atomically

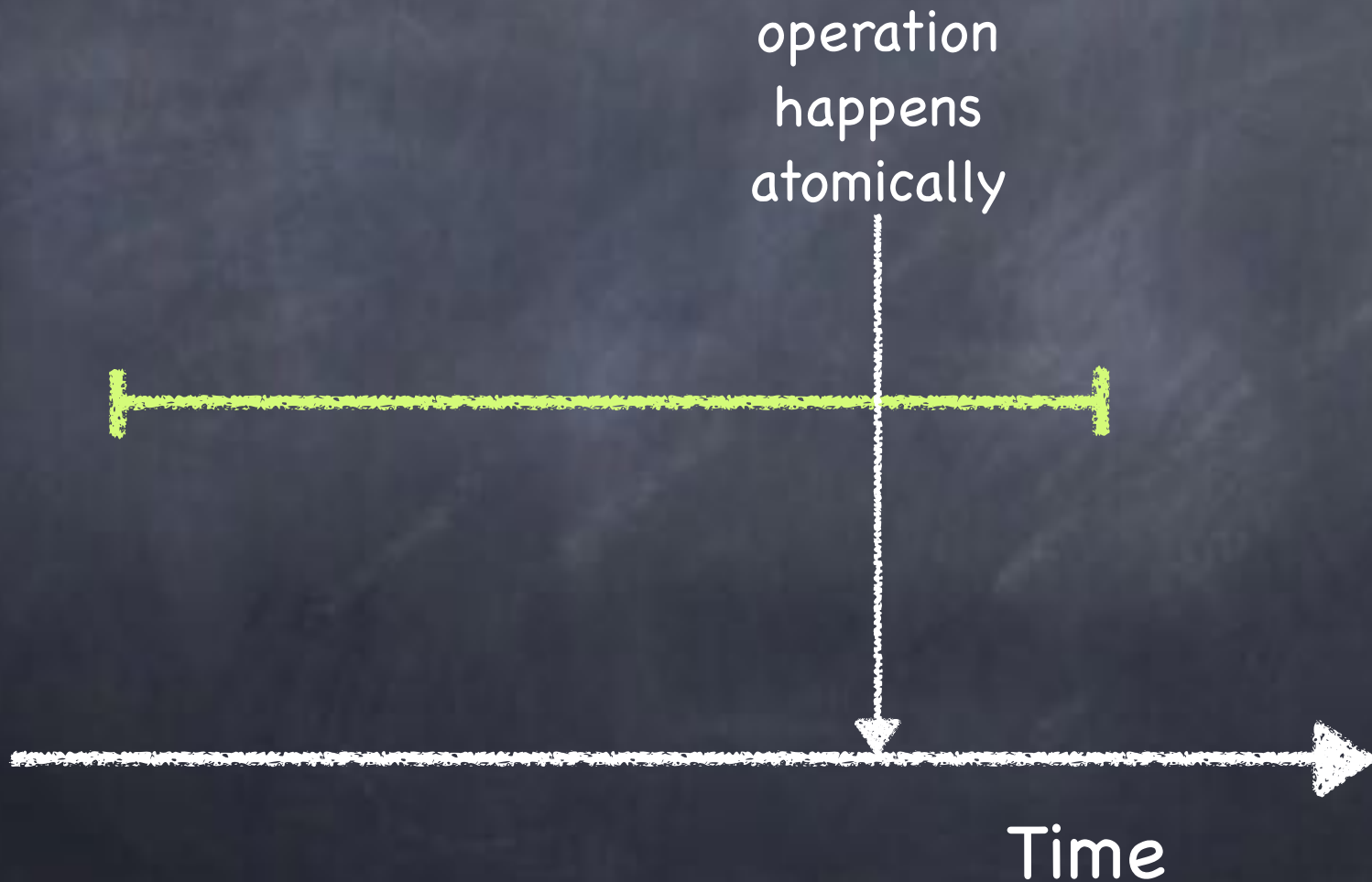


Time

Life of an Atomic Operation



Life of an Atomic Operation



Correct Behaviors

Suppose the queue is initially empty

put (3)

get () \leftarrow 3



Time

Correct Behaviors

Suppose the queue is initially empty

put (3)

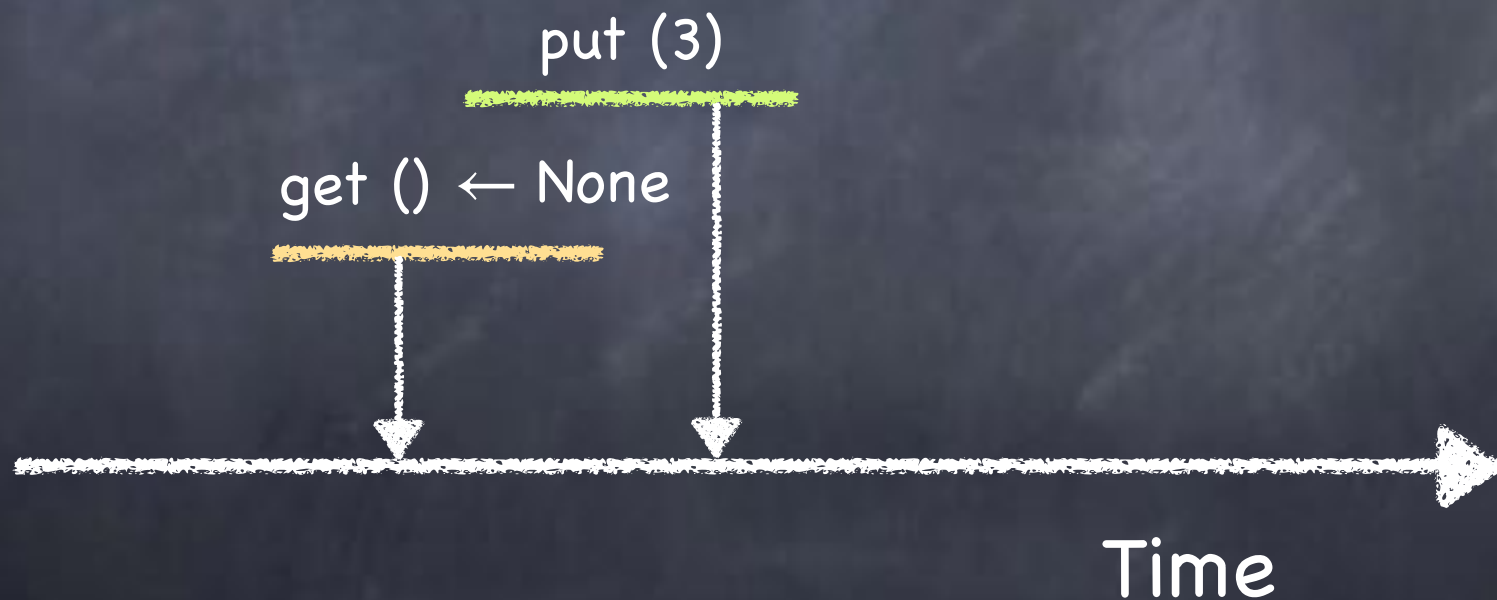
get () ← None



Time

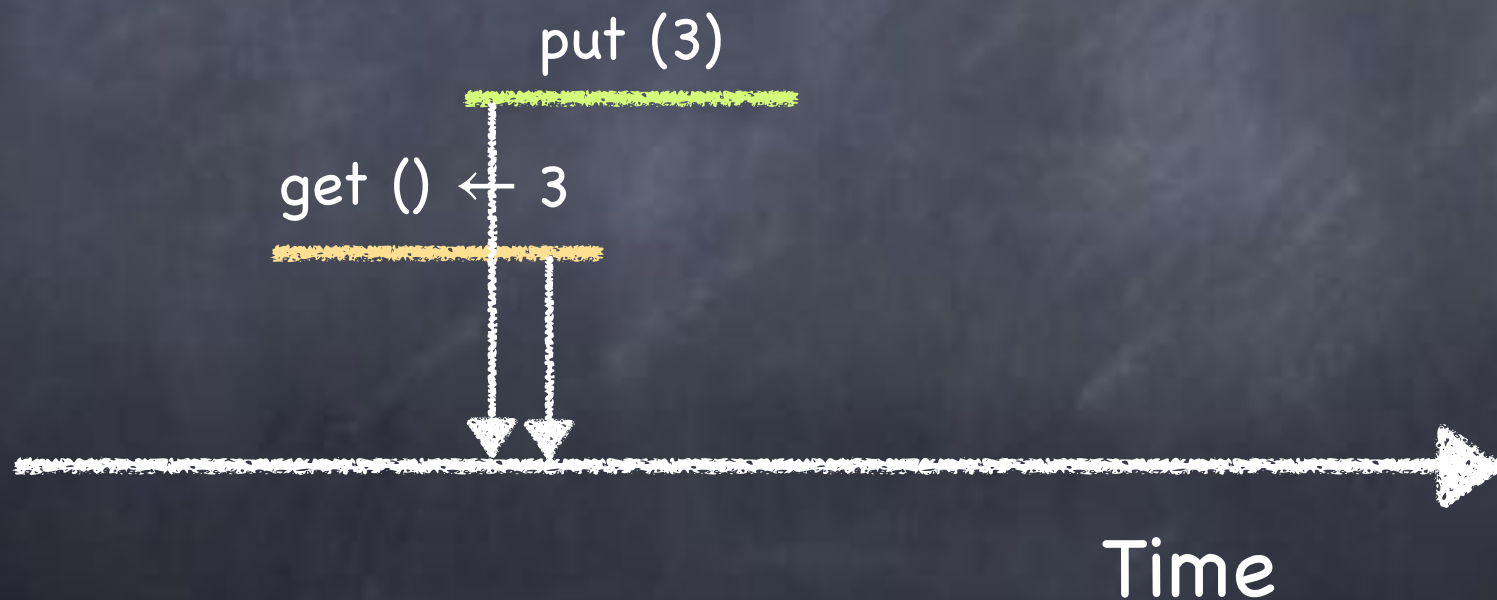
Correct Behaviors

Suppose the queue is initially empty

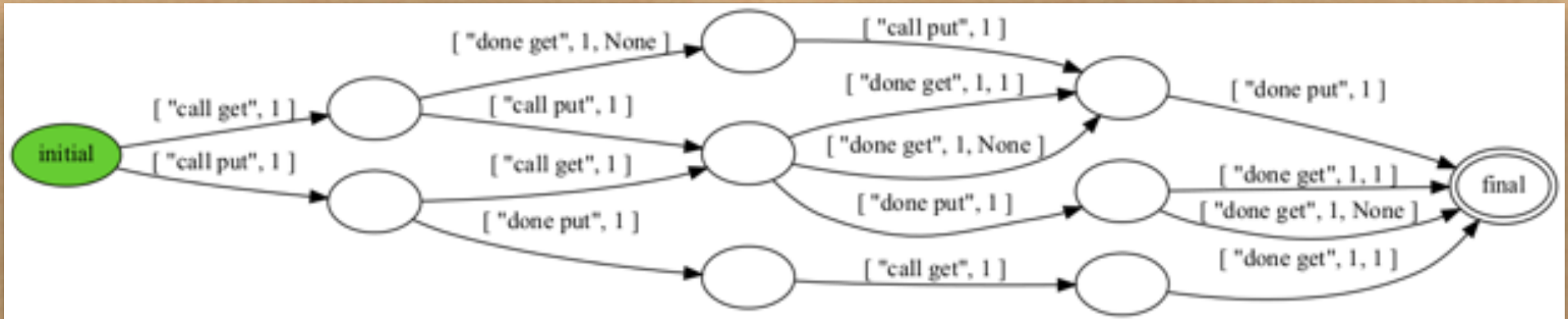


Correct Behaviors

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