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 sequential specification and the implementation
- We check whether running the test program against the implementation produces the behaviors (e.g., returns the same values) as running it against the sequential specification

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 runs 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 an interleaving is correct?

- We run the test program against both the concurrent 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

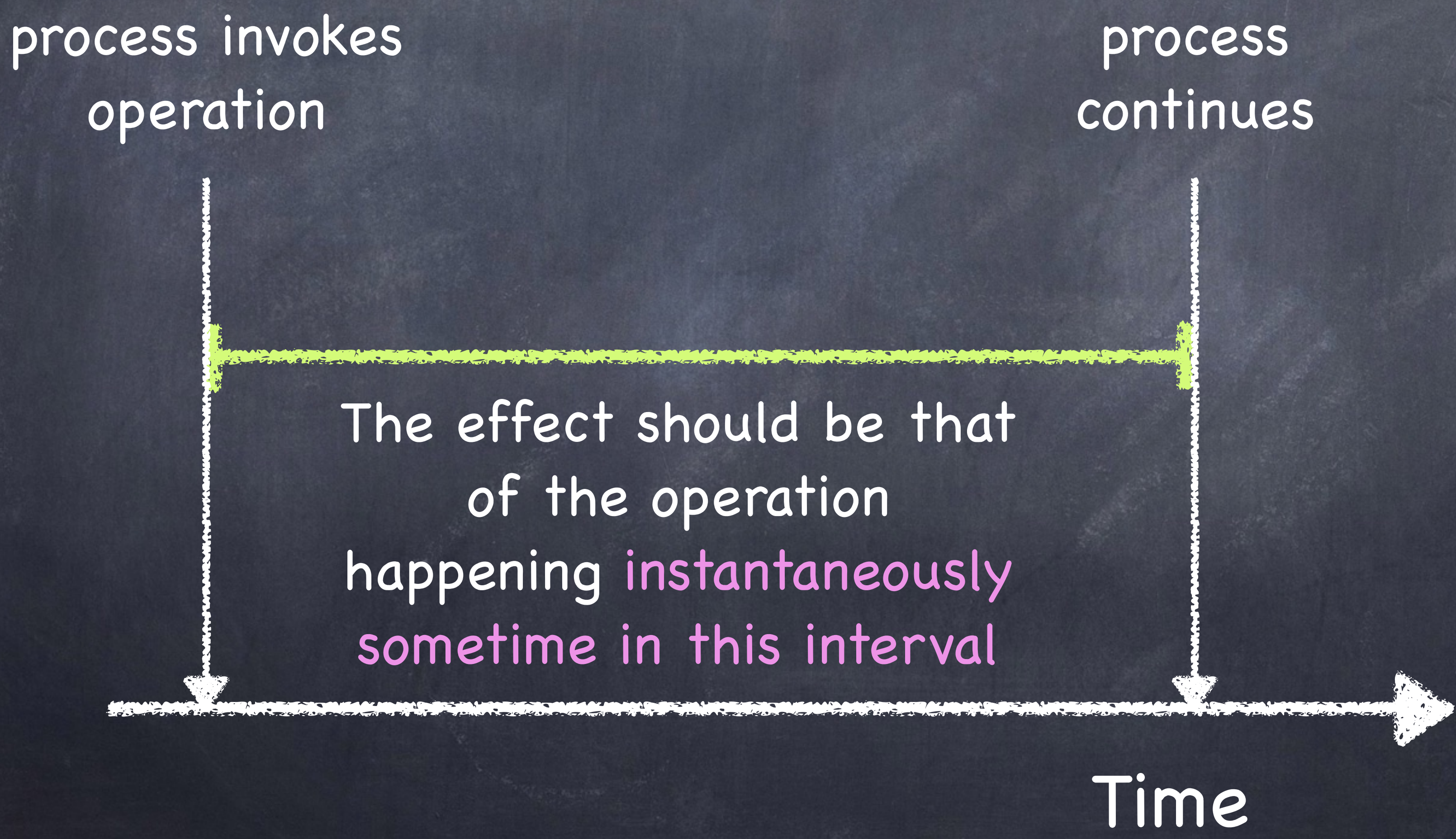
```
import queue

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

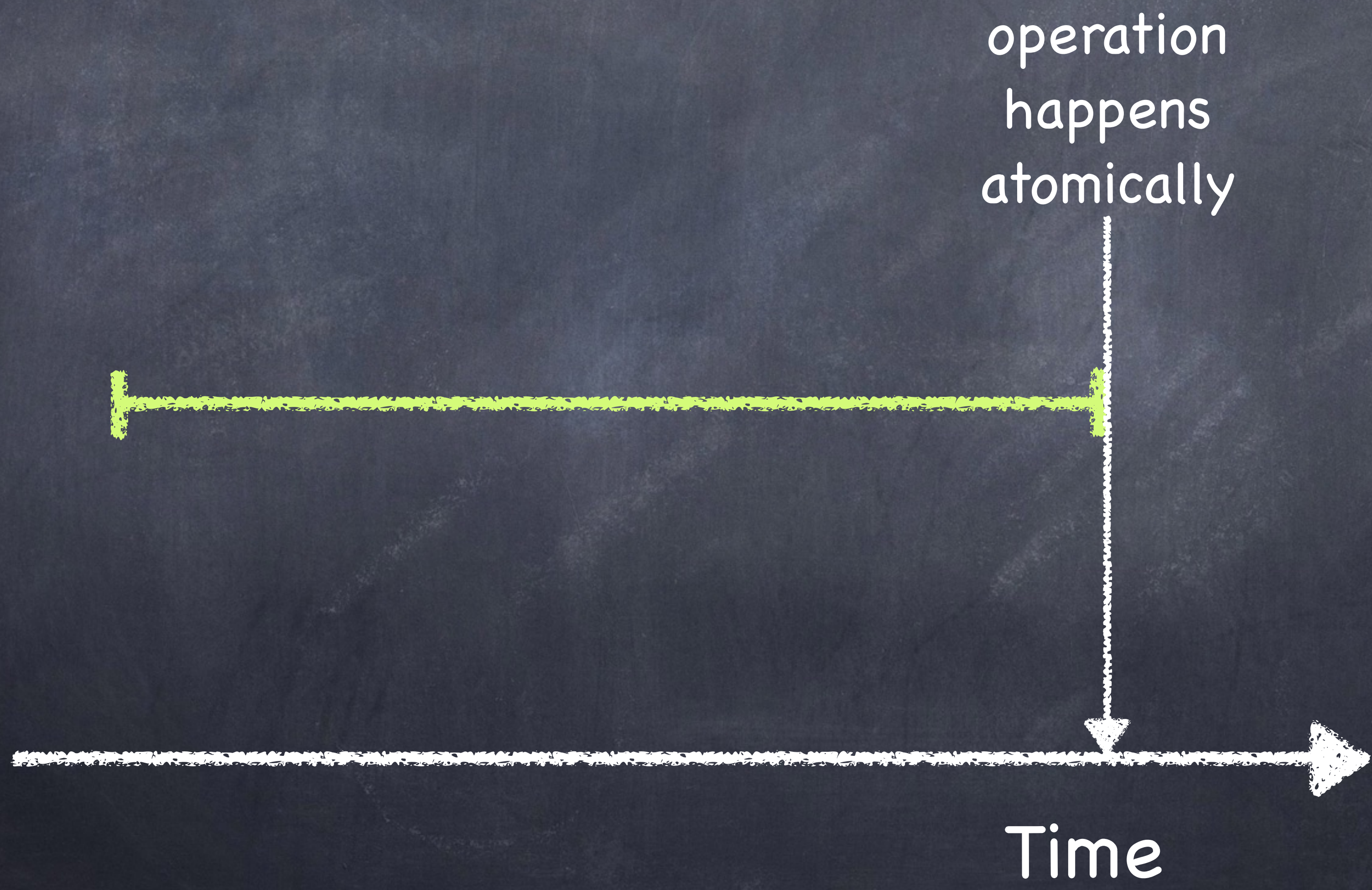
But which behaviors of the implementation are correct?

Life of an Atomic Operation

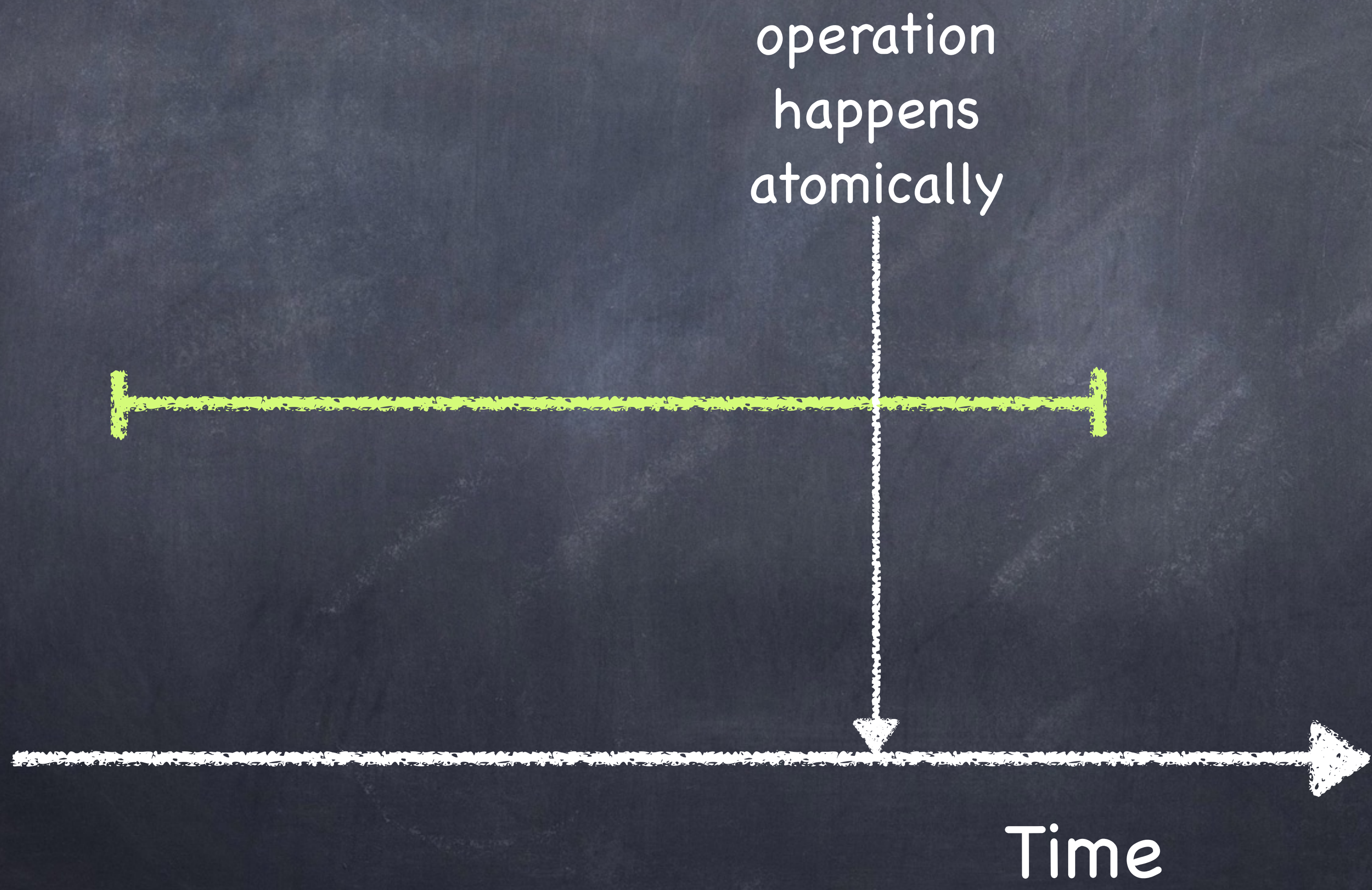


Life of an Atomic Operation

Life of an Atomic Operation

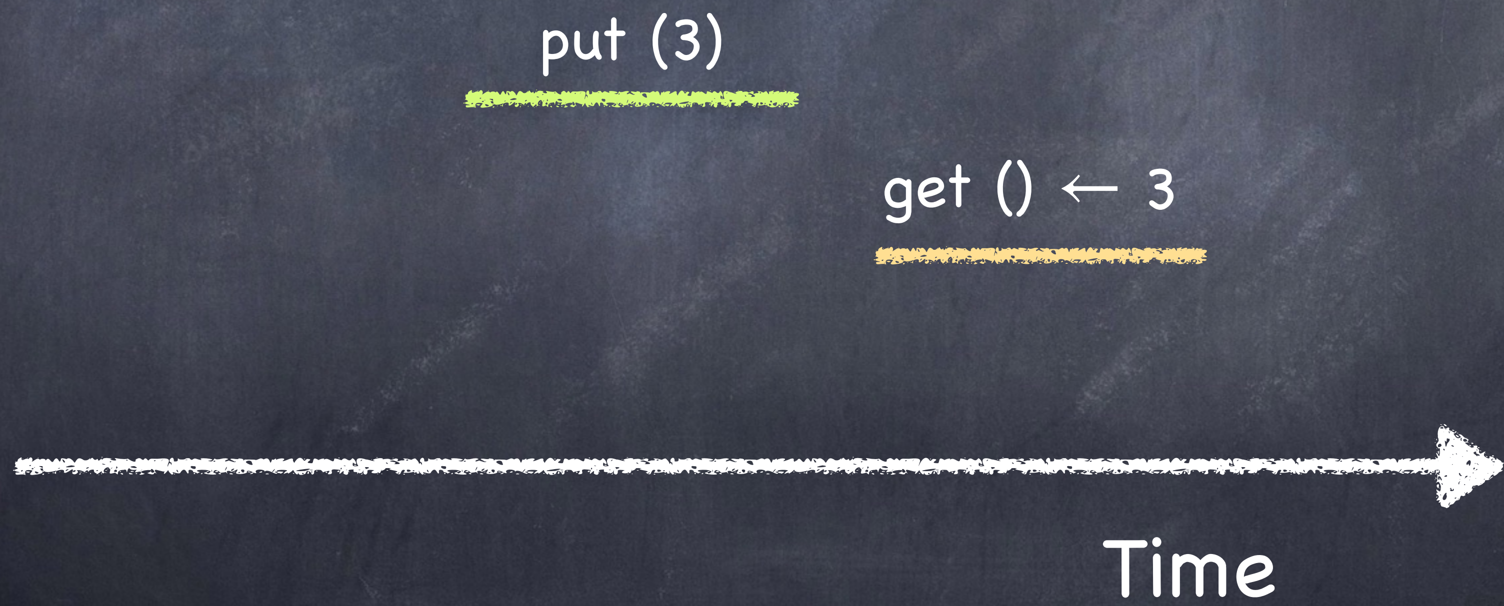


Life of an Atomic Operation



Correct Behaviors

Suppose the queue is initially empty



Correct Behaviors

Suppose the queue is initially empty

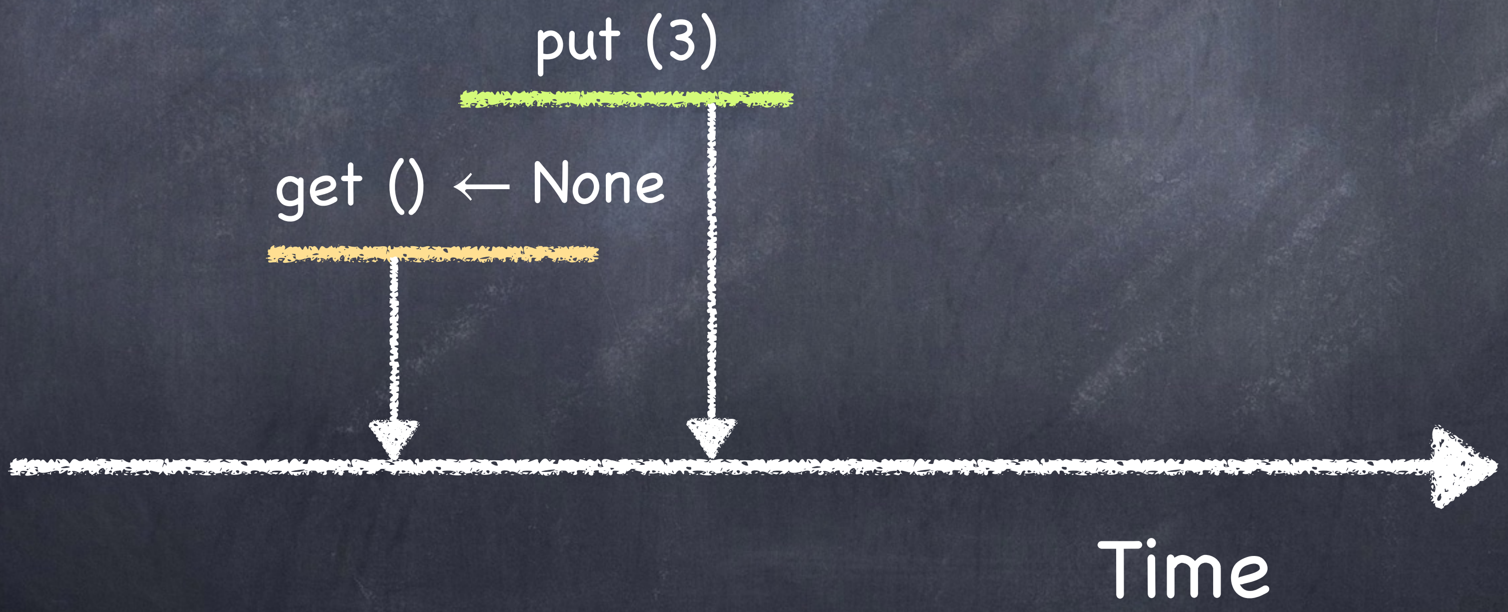
put (3)

get () ← None

Time

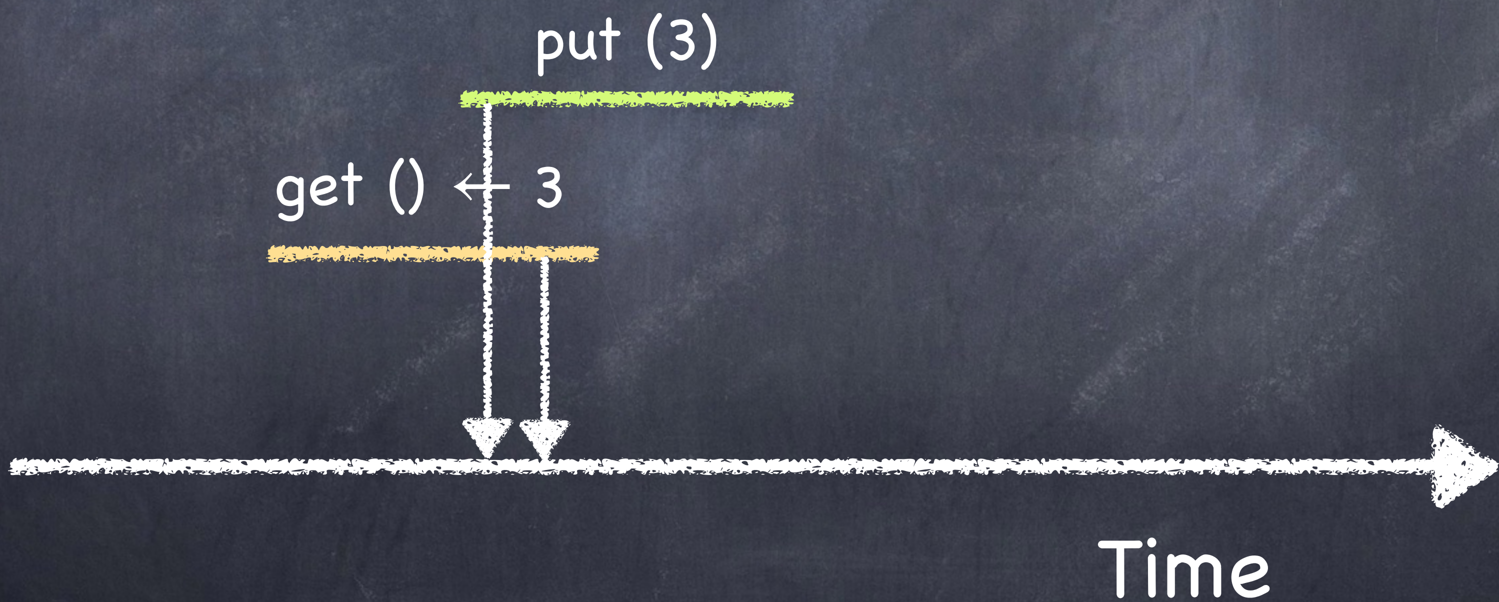
Correct Behaviors

Suppose the queue is initially empty

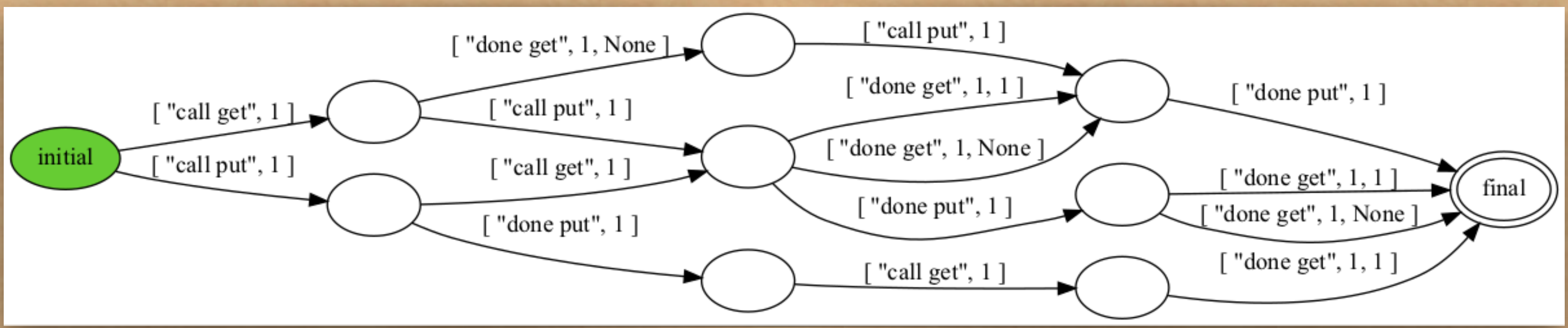


Correct Behaviors

Suppose the queue is initially empty



Queue test program




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

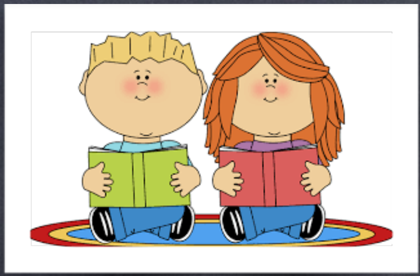
Testing: comparing behaviors

```
$ 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

Review

- Concurrent programming is hard!
  - Non-Determinism
  - Non-Atomicity
- Critical Sections simplify things
  - mutual exclusion
  - progress
- Critical Sections use a lock
  - Threads need lock to enter the CS
  - Only one thread can get the section's lock



Readers-Writers

- Models access to an object (e.g., a database), shared among several threads
  - some threads only read the object
  - others only write it
- Safety

$$(\#r \geq 0) \wedge (0 \leq \#w \leq 1) \wedge ((\#r > 0) \Rightarrow (\#w = 0))$$

How to get more concurrency?

- Idea: allow multiple read-only operations to execute concurrently
  - In many cases, reads are much more frequent than writes
- Reader/Writer lock
  - at most one writer, and, if no writer, any number of readers

$$(\#r \geq 0) \wedge (0 \leq \#w \leq 1) \wedge ((\#r > 0) \Rightarrow (\#w = 0))$$

Reader/Writer Lock Specification

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

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

def read_release(rw):
    atomically rw→nreaders −= 1

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

def write_release(rw):
    atomically rw→nwriters = 0
```

R/W Locks: Test for Mutual Exclusion

```
import RW

const NOPS = 3

rw = RW.RWlock()

def thread():
    while choose({ False, True }):
        if choose({ "read", "write" }) == "read":
            RW.read_acquire(?rw)
            rcs: assert (countLabel(rcs) >= 1) and (countLabel(wcs) == 0)
            RW.read_release(?rw)
        else: # write
            RW.write_acquire(?rw)
            wcs: assert (countLabel(rcs) == 0) and (countLabel(wcs) == 1)
            RW.write_release(?rw)

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

In CS

In CS

Multiple Readers

No Writer

1 Writer and No Readers

Cheating R/W Lock Implementation

```
import synch

def RWlock():
    result = synch.Lock()

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

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

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

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

Only 1 Reader gets a lock at a time!

Cheating R/W Lock Implementation

```
import synch

def RWlock():
    result = synch.Lock()

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

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

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

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

Only 1 Reader gets a lock at a time!

It is missing behaviors allowed by the specification

Cheating R/W Lock Implementation

```
import synch

def RWlock():
    result = synch.Lock()

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

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

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

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

Only 1 Reader gets a lock at a time!

It is missing behaviors allowed by the specification

But, at least, no bad behavior!

Busy-Waiting Implementation

```
from synch import Lock, acquire, release

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

def read_acquire(rw):
    acquire(?rw→lock)
    while rw→nwriters > 0:
        release(?rw→lock)
        acquire(?rw→lock)
    rw→nreaders += 1
    release(?rw→lock)

def read_release(rw):
    acquire(?rw→lock)
    rw→nreaders −= 1
    release(?rw→lock)

def write_acquire(rw):
    acquire(?rw→lock)
    while (rw→nreaders + rw→nwriters) > 0:
        release(?rw→lock)
        acquire(?rw→lock)
    rw→nwriters = 1
    release(?rw→lock)

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

Busy waiting

Acquire the lock
Test the condition
Release the lock
Repeat

The lock protects nreaders and nwriters, not the RW critical section!

It has the same behaviors as the implementation!

Busy-Waiting Implementation

```
from synch import Lock, acquire, release

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

def read_acquire(rw):
    acquire(?rw→lock)
    while rw→nwriters > 0:
        release(?rw→lock)
        acquire(?rw→lock)
    rw→nreaders += 1
    release(?rw→lock)

def read_release(rw):
    acquire(?rw→lock)
    rw→nreaders −= 1
    release(?rw→lock)

def write_acquire(rw):
    acquire(?rw→lock)
    while (rw→nreaders + rw→nwriters) > 0:
        release(?rw→lock)
        acquire(?rw→lock)
    rw→nwriters = 1
    release(?rw→lock)

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

Busy waiting

It has the same behaviors as the implementation!

Wasteful!

Process continuously scheduled to try to get the lock even if it is not available

Conditional Waiting

Conditional Waiting

- Threads wait for each other to prevent multiple threads in the CS
- But there may be other reasons:
  - Wait until queue is not empty before executing get()
  - Wait until there are no readers (or writers) in a reader/writer block
  - ...

Busy Waiting: not a good way

- Wait until queue is not empty:

```
done = False
while not done:
   next = get(q)
   done = next != None
```

- Wastes CPU cycles
- Creates unnecessary contention

Binary Semaphores

Dijkstra 1962

Binary Semaphore

- Boolean variable (much like a lock)
- Three operations
  - binsema = BinSema(False or True)
    - initializes binsema
  - acquire (?binsema)
    - waits until !binsema is False, then sets !binsema to True
  - release(?binsema)
    - sets !binsema to False
    - can only be called if !binsema = True

P & V

- Dijkstra was Dutch
  - He said Probeer-te-verlagen instead of acquire - and shortened it to P
  - He said Verhogen instead of release - and shortened it to V
  - Still very popular nomenclature
  - To remember it:
    - Procure (acquire)
    - Vacate (release)

Binary Semaphore Specification

```
def BinSema(acquired):
    result = acquired

def Lock():
    result = BinSema(False)

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

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

Semaphores v. Locks

Locks	Binary Semaphores
Initially "unlocked" (False)	Can be initialized to False or True
Usually acquired and released by the same thread	Can be acquired and released by different threads
Mostly used to implement critical sections	Can be used to implement critical sections as well as waiting for special conditions

Waiting with Semaphores

```
import synch

condition = BinSema(True)

def T0():
    acquire(?condition)

def T1()
    release(?condition)


spawn(T0)
spawn(T1)
```

Encode condition as a binary semaphore

Wait for condition to come true

Signal condition has become true

What happens if T0 runs first?

What happens if T1 runs first?

Semaphores can be locks too!

```
lk = BinSema(False)

acquire(?lk)

release(?lk)
```

Initialized to False

grab lock

release lock

What else can we do with binary semaphores?