Testing a Concurrent Queue?

```python
import queue

def sender(q, v):
    queue.put(q, v)

def receiver(q):
    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)
```
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

```python
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)

nput = choose {1..NOPS−1}
for i in {1..nput}:
    spawn put_test(i)
for i in {1..NOPS−nput}:
    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

The effect should be that of the operation happening instantaneously sometime in this interval.
Life of an Atomic Operation

operation happens atomically

Time
Life of an Atomic Operation

operation happens atomically
Life of an Atomic Operation

Operation happens atomically
Correct Behaviors

Suppose the queue is initially empty

put (3)

get () ← 3

Time
Correct Behaviors

Suppose the queue is initially empty

- put (3)
- get () ← None

Time
Correct Behaviors

Suppose the queue is initially empty

put (3)

get () ← None
Correct Behaviors

Suppose the queue is initially empty

Time

put (3)

get () ← 3
Queue test program

$ harmony -c NOPS=2 -o spec.png 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

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

\[(\#r \geq 0) \land (0 \leq \#w \leq 1) \land ((\#r > 0) \implies (\#w = 0))\]
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
import RW

const NOPS = 3

rw = RW.RWLock()

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

        for i in {1..NOPS}:
            spawn thread()
Cheating R/W
Lock Implementation

```python
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

```python
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

```python
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)

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

```python
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)
```

It has the same behaviors as the implementation!

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

Wasteful!
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`
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

```python
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

<table>
<thead>
<tr>
<th></th>
<th>Locks</th>
<th>Binary Semaphores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially “unlocked” (False)</td>
<td>Can be initialized to False or True</td>
<td></td>
</tr>
<tr>
<td>Usually acquired and released by the same thread</td>
<td>Can be acquired and released by different threads</td>
<td>Can be used to implement critical sections as well as waiting for special conditions</td>
</tr>
<tr>
<td>Mostly used to implement critical sections</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Waiting with Semaphores

```python
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!

\[
lk = \text{BinSema}(\text{False})
\]

- **acquire(?lk)**: grab lock
- **release(?lk)**: release lock

*Initialized to False*
What else can we do with binary semaphores?