Queue implementation, v2:2 locks

- Separate locks for head and tail
  - put and get can proceed concurrently
- Trick: put 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

```python
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)
```

Why an `atomic_store` here?
Queue implementation, v2:2 locks

```python
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)

one lock per node

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

Hand-over-hand locking

```python
def _node(v, n):
    # allocate and initialize a new list node
    result = alloc({ .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))
```
Sorted lists with lock per node

Multiple threads can access the list simultaneously, but they can't overtake one another!
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

Model access to an object (e.g., a database), shared among several threads:

- some threads only read the object
- others only write it

Safety

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

\[ (#r \geq 0) \land (0 \leq #w \leq 1) \land ((#r \geq 0) \Rightarrow (#w = 0)) \]
Reader/Writer Lock Specification

def RWlock():
    result = { .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

```python
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()
```
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!
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);

Cheating R/W
Lock Implementation

But, at least,
no bad behavior!

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():
    result = { .lock: Lock(), .nread: 0, .nwrite: 0 }

def read_acquire(rw):
    acquire(rw→lock)
    while rw→nwrite > 0:
        release(rw→lock)
        acquire(rw→lock)
    rw→nread += 1
    release(rw→lock)

def read_release(rw):
    acquire(rw→lock)
    rw→nread -= 1
    release(rw→lock)

def write_acquire(rw):
    acquire(rw→lock)
    while (rw→nread + rw→nwrite) > 0:
        release(rw→lock)
        acquire(rw→lock)
    rw→nwrite = 1
    release(rw→lock)

def write_release(rw):
    acquire(rw→lock)
    rw→nwrite = 0
    release(rw→lock)

Acquire the lock
Test the condition
Release the lock
Repeat

It has the same behaviors as the implementation!
Busy-Waiting Implementation

from synch import Lock, acquire, release

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

def read.acquire(rw):
    acquire(rw→lock)
    while rw→nreaders > 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.
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
  - ...

Waiting
Busy Waiting: not a good way

Wait until queue is not empty:

```python
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 – it shortened it to **P**
- He said *Verhogen* instead of release – it shortened it to **V**

Still very popular nomenclature

To remember it:
- **Procure** (acquire)
- **Vacate** (release)
# Semaphores v. Locks

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>Usually acquired and released by the same thread</td>
<td>Can be acquired and released by different threads</td>
</tr>
<tr>
<td>Mostly used to implement critical sections</td>
<td>Can be used to implement critical sections as well as waiting for special conditions</td>
</tr>
</tbody>
</table>
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
Waiting with Semaphores

```python
import synch  

condition = BinSema(True)  

def T0():  
    acquire(?condition)  

def T1():  
    release(?condition)  

t1 = spawn(T0)  
t2 = 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?