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

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():
    return _node(-1, None), _node((1, None), None)

empty list: [None]
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):
    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

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

```python
def insert(lst, v):
    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):
    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):
    before, after = find(lst, v):
    result = after.value == (0, v)
release(after.lock)
release(before.lock)
```
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

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

```python
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!
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():
    result = {'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

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

It has the same behaviors as the implementation!

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

```python
from synch import Lock, acquire, release

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

def read_acquire(rw):
    acquire(rw->lock)
    while rw->nwriters > 0:
        release(rw->lock)
        acquire(rw->lock)
    rw->nreades += 1
    release(rw->lock)

def read_release(rw):
    acquire(rw->lock)
    rw->nreades -= 1
    release(rw->lock)

def write_acquire(rw):
    acquire(rw->lock)
    while (rw->nreades + 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)
```
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:

```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></th>
<th>Locks</th>
<th>Binary Semaphores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially “unlocked”</td>
<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>
<td>Can be acquired and released by different threads</td>
</tr>
<tr>
<td>Mostly used to implement critical sections</td>
<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

```
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!

```python
lk = BinSema(False)
acquire(lk)
release(lk)
```

- Initialized to False
- grab lock
- release lock
What else can we do with binary semaphores?
Conditional Critical Sections

A critical section with an associated condition

- queue.get(), but wait until queue is not empty
  - don’t want two threads to run code at the same time
  - don’t want any thread to run queue.get() when the queue is empty
- print(), but wait until printer is idle
- RW.read_acquire(), but only when there are no writers in the critical section
One Critical Section, multiple conditions

Some conditional critical sections can have multiple conditions:

- R/W lock
  - readers are waiting for writers to leave
  - writers are waiting for readers and writers to leave
- bounded queue
  - dequeuers waiting for queue to be not empty
  - enqueueuers waiting for queue to be not full
- ...
...
High level idea: selective baton passing

- To execute inside the CS, thread needs the baton
- Threads can be waiting for various conditions
  - while they do, they don’t hold the baton
- When a thread with the baton leaves the CS, it checks whether there are threads waiting for a condition that now holds
  - If so, it passes the baton to one such thread
  - If not, the CS is vacated, and the baton can be picked up by another thread when it comes along
Split Binary Semaphores

Implement baton passing with multiple binary semaphores

\( N \) conditions require \( N + 1 \) binary semaphores

- one of each condition
- one to enter the CS in the first place
**Split Binary Semaphores**

**Invariant:** At most one of these semaphores is released (i.e., its value is False)

- If all are acquired (True), baton held by some thread (some thread in CS)
- If one is released (False), no thread holds baton (CS is empty)
  - if it is the “entry” semaphore, no thread is waiting on a condition that holds—any thread can enter CS
  - if it is one of the condition semaphores, some thread waiting on that condition can enter CS

Hoare 1973
Jabs...

Nurse administers C and F vaccines, one patient at a time

Threads

Nurse’s office: critical section
Rooms: waiting conditions

Nurse’s office

Covid room

Semaphores

Flu waiting room

At any time, exactly one semaphore or thread is green

{ if thread, in CS
  if sema, released (False) }

{ if thread, outside CS
  if sema, acquired (True) }
Jabs...

Nurse administers C and F vaccines, one patient at a time

- Nurse’s office: critical section
- Rooms: waiting conditions

At any time, exactly one semaphore or thread is green (and thus, at most one semaphore is green (Invariant))
What this models

- Reader/writer lock
  - Nurse’s office: critical section
  - Waiting Room 1: readers waiting for writer to leave
  - Waiting Room 2: writers waiting for readers and writer to leave

- Bounded queue
  - Nurse’s office: critical section
  - Waiting Room 1: dequeuers waiting for non-empty queue
  - Waiting Room 2: enqueuers waiting for not-full queue

...
Jabs...

Nurse administers C and F vaccines, one patient at a time

- if thread, in CS
- if sema, released (False)

- if thread, outside CS
- if sema, acquired (True)

At any time, exactly one semaphore or thread is green
Nurse administers C and F vaccines, one patient at a time

At any time, exactly one semaphore or thread is green

Nurse’s office: critical section
Rooms: waiting conditions
Jabs...

Nurse administers C and F vaccines, one patient at a time

Nurse’s office: critical section
Rooms: waiting conditions

At any time, exactly one semaphore or thread is green
Jabs...

Nurse administers C and F vaccines, one patient at a time

Nurse’s office: critical section
Rooms: waiting conditions

Thread 1 needs to wait for Condition 1

At any time, exactly one semaphore or thread is green
Jabs...

Nurse administers C and F vaccines, one patient at a time

- if thread, in CS
  - if sema, released (False)
- if thread, outside CS
  - if sema, acquired (True)

No thread waiting for a condition that holds

At any time, exactly one semaphore or thread is green

Nurse's office: critical section
Rooms: waiting conditions
Nurse administers C and F vaccines, one patient at a time

No thread waiting for a condition that holds

At any time, exactly one semaphore or thread is green

Nurse’s office: critical section
Rooms: waiting conditions
Jabs...

Nurse administers C and F vaccines, one patient at a time

\[
\begin{align*}
\text{if thread, in CS} & \quad \text{if thread, outside CS} \\
\text{if sema, released (False)} & \quad \text{if sema, acquired (True)}
\end{align*}
\]

Thread 2 can enter the CS

At any time, exactly one semaphore or thread is green

Nurse’s office: critical section

Rooms: waiting conditions
Jabs...

Nurse administers C and F vaccines, one patient at a time

- If thread, in CS
- If sema, released (False)

- If thread, outside CS
- If sema, acquired (True)

At any time, exactly one semaphore or thread is green

Thread 2 entered the critical section

Nurse’s office: critical section

Rooms: waiting conditions
Jabs...

Nurse administers C and F vaccines, one patient at a time

Thread 2 enables Condition 1 and wants to leave

Nurse’s office: critical section
Rooms: waiting conditions

At any time, exactly one semaphore or thread is green
Nurse administers C and F vaccines, one patient at a time

- If thread, in CS
- If sema, released (False)

- If thread, outside CS
- If sema, acquired (True)

Thread 2 left, Condition 1 holds

At any time, exactly one semaphore or thread is green

Nurse’s office: critical section
Rooms: waiting conditions
Jabs...

Nurse administers C and F vaccines, one patient at a time

Nurse’s office: critical section
Rooms: waiting conditions

Thread 1 (and only Thread 1) can enter CS

At any time, exactly one semaphore or thread is green
Jabs...

Nurse administers C and F vaccines, one patient at a time

- If thread, in CS
- If sema, released (False)

- If thread, outside CS
- If sema, acquired (True)

At any time, exactly one semaphore or thread is green

Nurse’s office: critical section
Rooms: waiting conditions
Jabs...

Nurse administers C and F vaccines, one patient at a time

- Nurse’s office: critical section
- Rooms: waiting conditions

At any time, exactly one semaphore or thread is green

If thread, in CS
  - if sema, released (False)

If thread, outside CS
  - if sema, acquired (True)
Nurse administers C and F vaccines, one patient at a time

At any time, **exactly one** semaphore or thread is green

Nurse’s office: critical section

Rooms: waiting conditions
Nurse administers C and F vaccines, one patient at a time

Nurse’s office: critical section

At any time, exactly one semaphore or thread is green
Nurse administers C and F vaccines, one patient at a time

Nurse’s office: critical section
Rooms: waiting conditions

At any time, exactly one semaphore or thread is green
Nurse administers C and F vaccines, one patient at a time

At any time, exactly one semaphore or thread is green

Nurse’s office: critical section
Rooms: waiting conditions

Thread 3 needs to wait for Condition 2
Nurse administers C and F vaccines, one patient at a time

At any time, exactly one semaphore or thread is green
Let’s build a R/W lock this way

Many roads lead to Rome...
Reader/Writer Lock
Specification (again)

```python
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
```
Reader/Writer Lock: Implementation

```
from synch import BinSema, acquire, release

def RWlock():
    result = {
        .nreaders: 0, .nwriters: 0, .mutex: BinSema(False),
        .r_gate: { .sema: BinSema(True), .count: 0 },
        .w_gate: { .sema: BinSema(True), .count: 0 }
    }
```

**Accounting**
- $nreaders$: #readers in the CS
- $r\_gate\.count$: #readers waiting to enter CS
- $nwriters$: #writers in the CS
- $w\_gate\.count$: #writers waiting to enter CS

**Invariants**
- If $n$ readers in the critical section, then $nreaders \geq n$
- If $n$ writers in the critical section, then $nwriters \geq n$
- $\forall (nreaders \geq 0 \land nwriters = 0)$
  $\lor (nreaders = 0 \land nwriters \leq 1)$
Reader/Writer Lock: Implementation

Waiting condition

Enter reader gate

Going through

Leave: let others try too

Note: acquire and release operations alternate
Reader/Writer Lock: Implementation

```python
def write_acquire(rw):
    acquire(?rw→mutex)
    if (rw→nreaders + rw→nwriters) > 0:
        rw→w_gate.count += 1; release_one(rw)
        acquire(?rw→w_gate.sema); rw→w_gate.count -= 1
        rw→nwriters += 1
        release_one(rw)

def write_release(rw):
    acquire(?rw→mutex); rw→nwriters -= 1; release_one(rw)
```

- **Waiting condition**
- **Enter writer gate**
- **Enter main gate**
- **Similar structure to read_acquire()**
Reader/Writer Lock: Implementation

When leaving the critical section:

```python
def release_one(rw):
    if (rw.nwriters == 0) and (rw.r_gate.count > 0):
        release(rw.r_gate.sema)
    elif ((rw.nreaders + rw.nwriters) == 0) and (rw.w_gate.count > 0):
        release(rw.w_gate.sema)
    else:
        release(rw.mutex)
```

If no writers in the Critical Section and there are readers waiting, then let a reader in!
Reader/Writer Lock: Implementation

```python
def release_one(rw):
    if (rw→nwriters == 0) and (rw→r_gate.count > 0):
        release(?rw→r_gate.sema)
    elif ((rw→nreaders + rw→nwriters) == 0) and (rw→w_gate.count > 0):
        release(?rw→w_gate.sema)
    else:
        release(?rw→mutex)
```
Reader/Writer Lock: Implementation

```
def release_one(rw):
    if (rw->nwriters == 0) and (rw->r_gate.count > 0):
        release(?rw->r_gate.sema)
    elif ((rw->nreaders + rw->nwriters) == 0) and (rw->w_gate.count > 0):
        release(?rw->w_gate.sema)
    else:
        release(?rw->mutex)

when leaving the critical section:

otherwise...

let anyone in!
```
Reader/Writer Lock: Implementation

when leaving the critical section:

```python
def release_one(rw):
    if (rw->nwriters == 0) and (rw->r_gate.count > 0):
        release(?rw->r_gate.sema)
    elif ((rw->nreaders + rw->nwriters) == 0) and (rw->w_gate.count > 0):
        release(?rw->w_gate.sema)
    else:
        release(?rw->mutex)
```

Can these two conditions be reversed?

What is the effect of that?
Reader/Writer Lock: Implementation

```python
def release_one(rw):
    if (rw->nwriters == 0) and (rw->r_gate.count > 0):
        release(?rw->r_gate.sema)
    elif ((rw->nreaders + rw->nwriters) == 0) and (rw->w_gate.count > 0):
        release(?rw->w_gate.sema)
    else:
        release(?rw->mutex)
```

What happens if multiple readers are waiting and a writer leaves?

Does it let all the readers in or just one?
A Hierarchy of Critical Sections

- We have two different critical sections...
- ...that occur at different levels of abstraction
  - the first relies a R/W lock
    - protects access to some database (say)
    - allows multiple readers in the CS
  - the second relies on split binary semaphores
    - protects the shared variables (nreaders, r_gate.count, etc) and implements the conditions we use to implement R/W locks
    - allows only on thread at a time in CS
Starvation

- Our R/W implementation can starve writers

- Change the waiting and release conditions:
  - when a reader tries to enter the CS, wait if there is a writer in the CS or there are writers waiting to enter CS
  - exiting reader prioritizes releasing a waiting writer
  - exiting writer prioritizes releasing a waiting reader

See Chapter 17 in the Harmony book
## Conditional Critical Sections

We know of two ways to implement them:

<table>
<thead>
<tr>
<th>Busy Waiting</th>
<th>Split Binary Semaphores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait for condition in loop, acquiring lock before testing for condition, and releasing it if condition does not hold</td>
<td>Use a collection of binary semaphores and keep track of state, including information about waiting threads</td>
</tr>
<tr>
<td>Easy to understand the code</td>
<td>State tracking is complicated</td>
</tr>
<tr>
<td>OK-ish for true multi-core, but bad for virtual threads</td>
<td>Good for both multicore and virtual threading</td>
</tr>
</tbody>
</table>