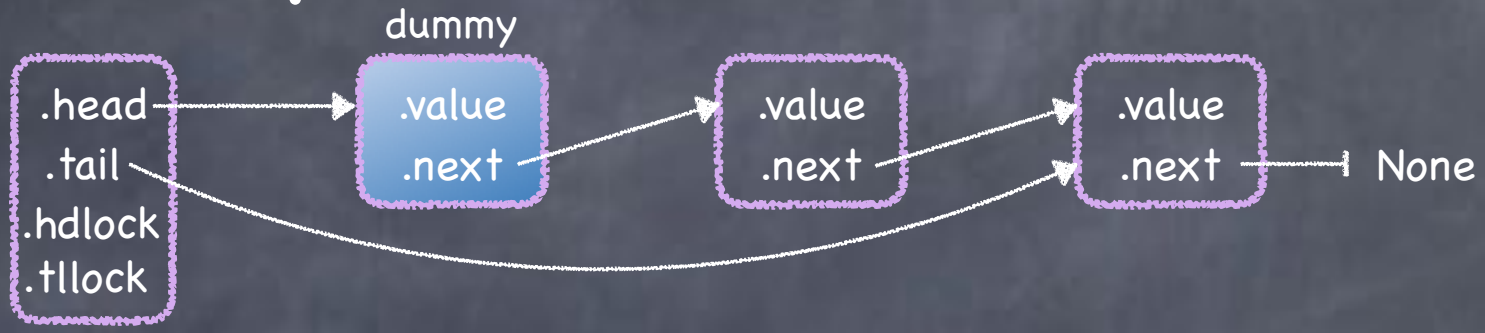
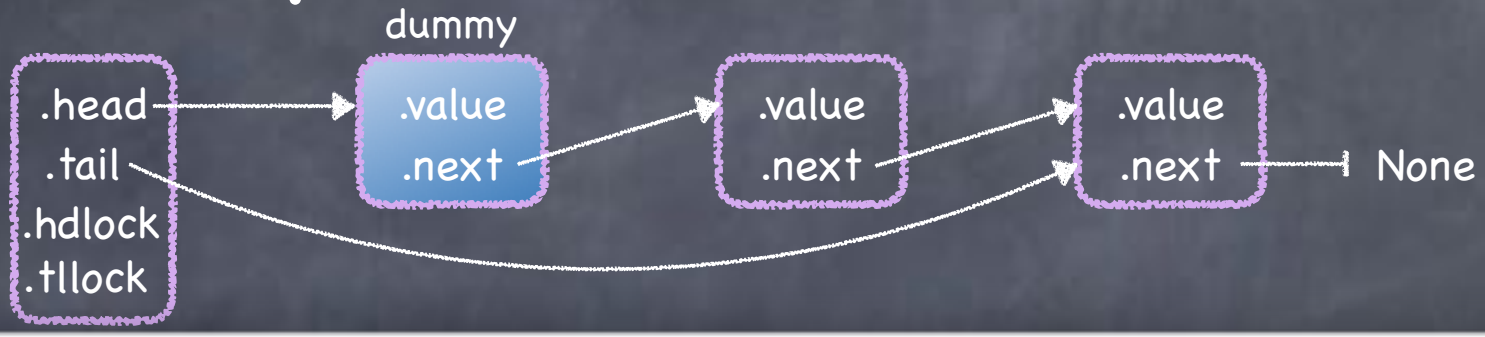
Queue implementation, v2:2 locks



- 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

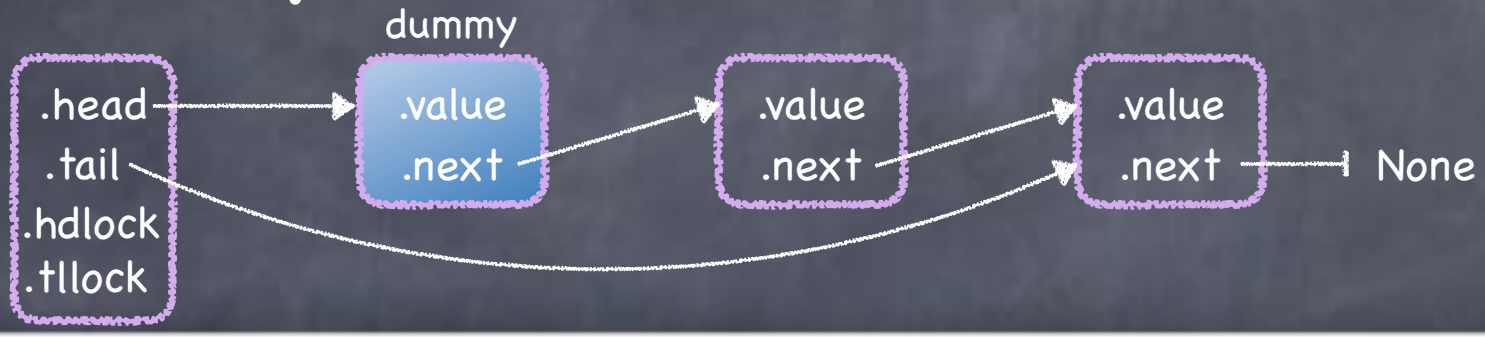
Queue implementation, v2:2 locks




```
1   from synch import Lock, acquire, release, atomic_load, atomic_store
2   from alloc import malloc, free
3
4   def Queue():
5       let dummy = malloc({ .value: (), .next: None }):
6           result = { .head: dummy, .tail: dummy, .hdlock: Lock(), .tllock: Lock() }
7
8   def put(q, v):
9       let node = malloc({ .value: v, .next: None }):
10          acquire(?q→tllock)
11          atomic_store(?q→tail→next, node)
12          q→tail = node
13          release(?q→tllock)
```

Why an atomic_store here?

Queue implementation, v2:2 locks




```
15  def get(q):
16      acquire(?q→hdlock)
17      let dummy = q→head
18      let node = atomic_load(?dummy→next):
19          if node == None:
20              result = None
21              release(?q→hdlock)
22          else:
23              result = node→value
24              q→head = node
25              release(?q→hdlock)
26              free(dummy)
```

...and here?

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


```
1   from synch import Lock, acquire, release
2   from alloc import malloc, free
3
4   def _node(v, n): # allocate and initialize a new list node
5       result = malloc({ .lock: Lock(), .value: v, .next: n })
6
7   def _find(lst, v):
8       var before = lst
9       acquire(?before→lock)
10      var after = before→next
11      acquire(?after→lock)
12      while after→value < (0, v):
13          release(?before→lock)
14          before = after
15          after = before→next
16          acquire(?after→lock)
17      result = (before, after)
18
19  def SetObject():
20      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


```
1   from synch import Lock, acquire, release
2   from alloc import malloc, free
3
4   def _node(v, n): # allocate and initialize a new list node
5       result = malloc({ .lock: Lock(), .value: v, .next: n })
6
7   def _find(lst, v):
8       var before = lst
9       acquire(?before→lock)
10      var after = before→next
11      acquire(?after→lock)
12      while after→value < (0, v):
13          release(?before→lock)
14          before = after
15          after = before→next
16          acquire(?after→lock)
17      result = (before, after)
18
19  def SetObject():
20      result = _node((-1, None), _node((1, None), None))
```

Hand-over-hand locking

empty list:

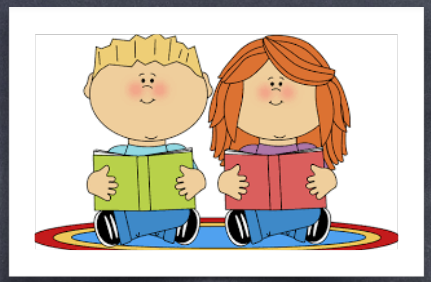
Sorted lists with lock per node


```
22  def insert(lst, v):
23      let before, after = _find(lst, v):
24          if after→value != (0, v):
25              before→next = _node((0, v), after)
26          release(?after→lock)
27          release(?before→lock)
28
29  def remove(lst, v):
30      let before, after = _find(lst, v):
31          if after→value == (0, v):
32              before→next = after→next
33              release(?after→lock)
34              free(after)
35          else:
36              release(?after→lock)
37          release(?before→lock)
38
39  def contains(lst, v):
40      let before, after = _find(lst, v):
41          result = after→value == (0, v)
42          release(?after→lock)
43          release(?before→lock)
```

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

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

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

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

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

Reader/Writer Lock Specification

```
1   def RWlock():
2       result = { .nreaders: 0, .nwriters: 0 }
3
4   def read_acquire(rw):
5       atomically when rw→nwriters == 0:
6           rw→nreaders += 1
7
8   def read_release(rw):
9       atomically rw→nreaders -= 1
10
11  def write_acquire(rw):
12      atomically when (rw→nreaders + rw→nwriters) == 0:
13          rw→nwriters = 1
14
15  def write_release(rw):
16      atomically rw→nwriters = 0
```

R/W Locks: Test for Mutual Exclusion

```
1   import RW
2
3   const NOPS = 3
4
5   rw = RW.RWlock()
6
7   def thread():
8       while choose({ False, True }):
9           if choose({ "read", "write" }) == "read":
10              RW.read_acquire(?rw)
11  In CS       rcs: assert (countLabel(rcs) >= 1) and (countLabel(wcs) == 0)
12              RW.read_release(?rw)
13          else: # write
14              RW.write_acquire(?rw)
15  In CS       wcs: assert (countLabel(rcs) == 0) and (countLabel(wcs) == 1)
16              RW.write_release(?rw)
17
18  for i in {1..NOPS}:
19      spawn thread()
```

Multiple Readers

No Writer

1 Writer and No Readers

Cheating R/W Lock Implementation

```
1   import synch
2
3   def RWlock():
4       result = synch.Lock()
5
6   def read_acquire(rw):
7       synch.acquire(rw);
8
9   def read_release(rw):
10      synch.release(rw);
11
12  def write_acquire(rw):
13      synch.acquire(rw);
14
15  def write_release(rw):
16      synch.release(rw);
```

Cheating R/W Lock Implementation

```
1   import synch
2
3   def RWlock():
4       result = synch.Lock()
5
6   def read_acquire(rw):
7       synch.acquire(rw);
8
9   def read_release(rw):
10      synch.release(rw);
11
12  def write_acquire(rw):
13      synch.acquire(rw);
14
15  def write_release(rw):
16      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

```
1   import synch
2
3   def RWlock():
4       result = synch.Lock()
5
6   def read_acquire(rw):
7       synch.acquire(rw);
8
9   def read_release(rw):
10      synch.release(rw);
11
12  def write_acquire(rw):
13      synch.acquire(rw);
14
15  def write_release(rw):
16      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

```
1   from synch import Lock, acquire, release
2
3   def RWlock():
4       result = { .lock: Lock(), .nreaders: 0, .nwriters: 0 }
5
6   def read_acquire(rw):
7       acquire(?rw→lock)
8       while rw→nwriters > 0:
9           release(?rw→lock)
10          acquire(?rw→lock)
11      rw→nreaders += 1
12      release(?rw→lock)
13
14  def read_release(rw):
15      acquire(?rw→lock)
16      rw→nreaders -= 1
17      release(?rw→lock)
18
19  def write_acquire(rw):
20      acquire(?rw→lock)
21      while (rw→nreaders + rw→nwriters) > 0:
22          release(?rw→lock)
23          acquire(?rw→lock)
24      rw→nwriters = 1
25      release(?rw→lock)
26
27  def write_release(rw):
28      acquire(?rw→lock)
29      rw→nwriters = 0
30      release(?rw→lock)
```

Busy waiting

Acquire the lock
Test the condition
Release the lock
Repeat

It has the same behaviors as the implementation!

Busy-Waiting Implementation

```
1   from synch import Lock, acquire, release
2
3   def RWlock():
4       result = { .lock: Lock(), .nreaders: 0, .nwriters: 0 }
5
6   def read_acquire(rw):
7       acquire(?rw→lock)
8       while rw→nwriters > 0:
9           release(?rw→lock)
10          acquire(?rw→lock)
11      rw→nreaders += 1
12      release(?rw→lock)
13
14  def read_release(rw):
15      acquire(?rw→lock)
16      rw→nreaders -= 1
17      release(?rw→lock)
18
19  def write_acquire(rw):
20      acquire(?rw→lock)
21      while (rw→nreaders + rw→nwriters) > 0:
22          release(?rw→lock)
23          acquire(?rw→lock)
24      rw→nwriters = 1
25      release(?rw→lock)
26
27  def write_release(rw):
28      acquire(?rw→lock)
29      rw→nwriters = 0
30      release(?rw→lock)
```

Busy waiting

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 nefore 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 - 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

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

Binary Semaphore Specification

```
1   def BinSema(acquired):
2       result = acquired
3
4   def Lock():
5       result = BinSema(False)
6
7   def acquire(binsema):
8       atomically when not !binsema:
9           !binsema =  True
10
11  def release(binsema):
12          assert !binsema
13          atomically !binsema = False
```

Waiting with Semaphores

```
1   import synch
2
3   condition = BinSema(True)
4
5   def T0():
6       acquire(?condition)
7
8   def T1()
9       release(?condition)
10
11
12  spawn(T0)
13  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?