Atomic Section ≠ Critical Section

Atomic Section	Critical Section
Only one thread can execute	Multiple threads can execute concurrently, just not within a critical section
Rare programming language paradigm	Ubiquitous: locks available in many mainstream programming languages
Good for specifying interlock instruction	Good for implementing concurrent data structures

Using Locks

- Data structures maintain some invariant
    - Consider a linked list
        - There is a head, a tail, and a list of nodes such as the head points to the first node, tail points to the last one, and each node points to the next one, except for the tail, which points to None. However, if the list is empty, head and tail are both None
- You can assume the invariant holds right after acquiring the lock
- You must make sure invariant holds again right before releasing the lock

Building a Concurrent Queue

- $q = \text{queue.new}()$: allocates a new queue
- $\text{queue.put}(q, v)$: adds v to the tail of queue q
- $v = \text{queue.get}(q)$: returns
  - None if q is empty, or
  - v if v was at the head of the queue

Specifying a Concurrent Queue

```
def Queue() returns empty:
    empty = []

def put(q, v):
    !q += [v,]


def get(q) returns next:
    if !q == []:
        next = None
    else:
        next = (!q)[0]
        del (!q)[0]
```

Sequential

```
def Queue() returns empty:
    empty = []

def put(q, v):
    atomically !q += [v,]

def get(q) returns next:
    atomically:
        if !q == []:
            next = None
        else:
            next = (!q)[0]
            del (!q)[0]
```

Concurrent

Example of using a Queue

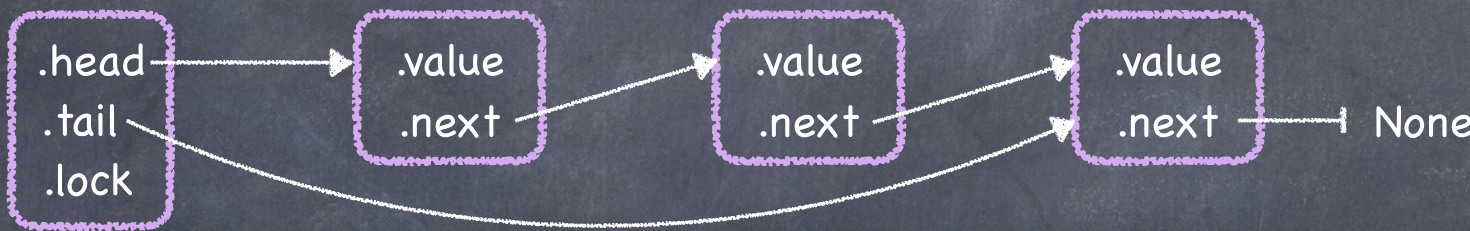
```
import queue

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)
```

enqueue v onto q

dequeue and check

create a queue

Queue implementation, v1




```
from synch import Lock, acquire, release
from alloc import malloc, free

def Queue() returns empty:
    empty = { .head: None, .tail: None, .lock: Lock() }

def put(q, v):
    let node = malloc({ .value: v, .next: None }):
        acquire(?q→lock)
        if q→tail == None:
            q→tail = q→head = node
        else:
            q→tail→next = node
            q→tail = node
        release(?q→lock)
```

dynamic memory allocation

create empty queue

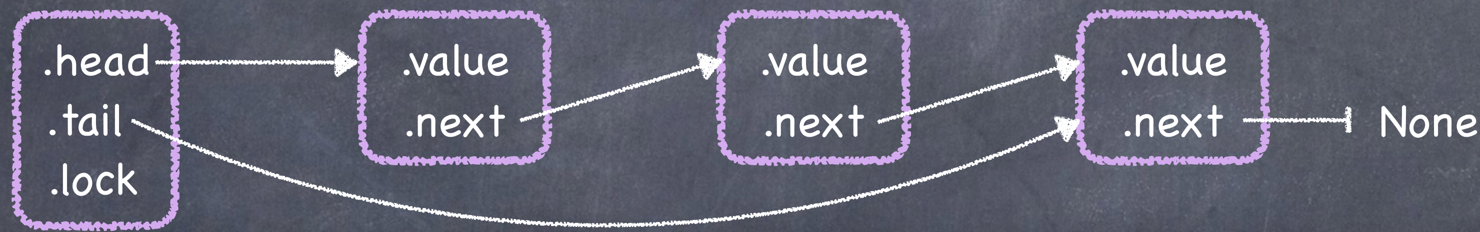
allocate node

grab lock

The Hard Stuff

release lock

Queue implementation, v1




```
def get(q) returns next:
    acquire(?q→lock)
    let node = q→head:
        if node == None:
            next = None
        else:
            next = node→value
            q→head = node→next
            if q→head == None:
                q→tail = None
            free(node)
    release(?q→lock)
```

grab lock

empty queue

The Hard Stuff

free dynamically allocated memory

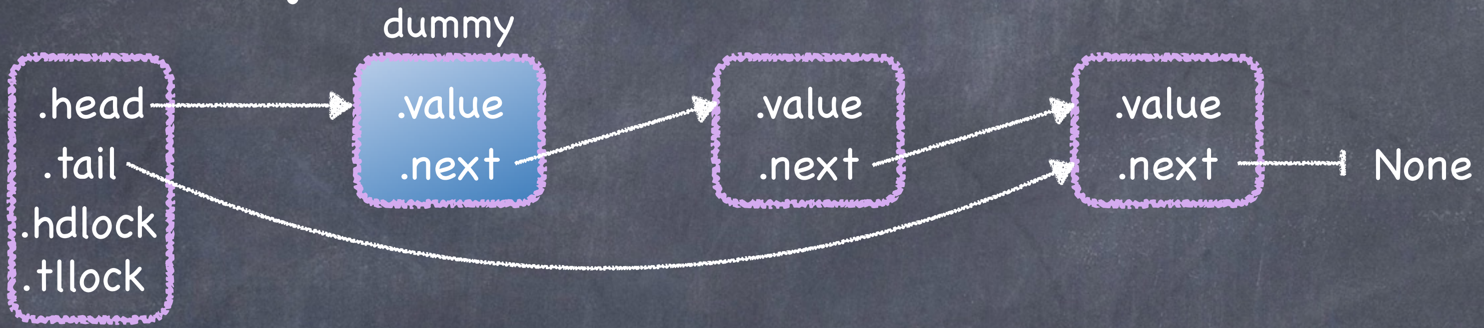
release lock

How important are concurrent queues?

- All important!
  - any resource that needs scheduling
    - CPU ready queue
    - disk, network, printer waiting queue
    - lock waiting queue
  - inter-process communication
    - Posix pipes: cat file | sort
  - actor-based concurrency
  - ...

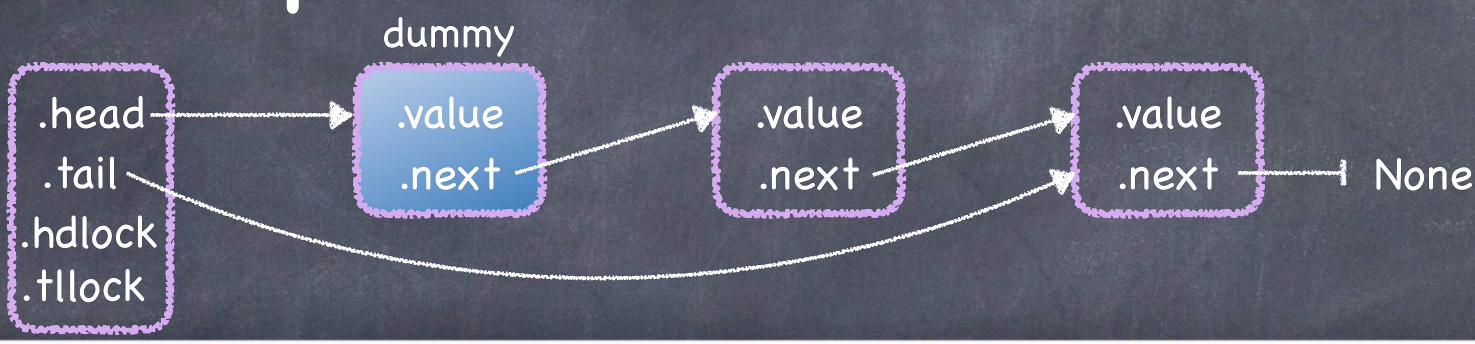
Performance
is
critical!

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 that was dequeued (except at the beginning)
  - head and tail never None

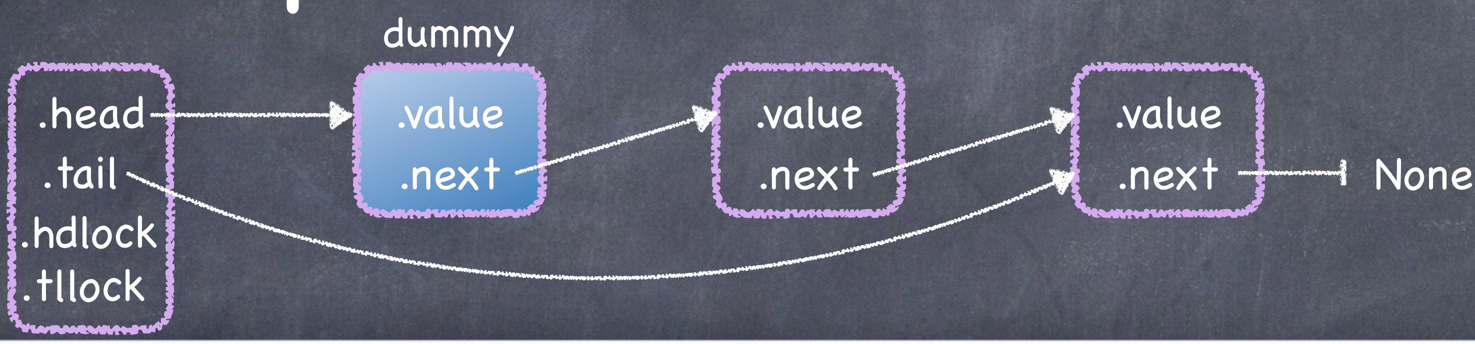
Queue implementation, v2:2 locks




```
from synch import Lock, acquire, release, atomic_load, atomic_store
from alloc import malloc, free

def Queue() returns empty:
    let dummy = malloc({ .value: (), .next: None }):
        empty = { .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




```
def get(q) returns next:
    acquire(?q→hdlock)
    let dummy = q→head
    let node = atomic_load(?dummy→next):
        if node == None:
            next = None
            release(?q→hdlock)
        else:
            next = node→value
            q→head = node
            release(?q→hdlock)
            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
    - think of having to walk a queue...
- 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) returns node: # allocate and initialize a new list node
    node = malloc({ .lock: Lock(), .value: v, .next: n })

def _find(lst, v) returns pair:
    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)
    pair = (before, after)

def SetObject() returns object:
    object = _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


```
from synch import Lock, acquire, release
from alloc import malloc, free

def _node(v, n) returns node: # allocate and initialize a new list node
    node = malloc({ .lock: Lock(), .value: v, .next: n })

def _find(lst, v) returns pair:
    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)
    pair = (before, after)

def SetObject() returns object:
    object = _node((−1, None), _node((1, None), None))
```

Hand-over-hand locking

empty list:

Sorted lists with lock per node


```
def insert(lst, v):
    let 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):
    let before, after = _find(lst, v):
        if after→value == (0, v):
            before→next = after→next
            free(after)
        release(?after→lock)
        release(?before→lock)

def contains(lst, v) returns present:
    let before, after = _find(lst, v):
        present = after→value == (0, v)
        release(?after→lock)
        release(?before→lock)
```

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