Lecture 23: A Queue Without Locks & Progress

COSC 273: Parallel and Distributed Computing

Spring 2023

Announcements

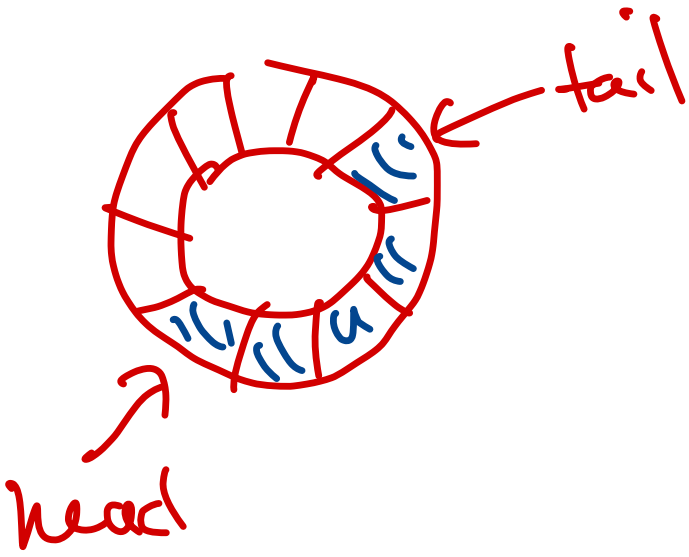
Prelim version by Weds.

1. Homework 03 Posted Soon
   - due Friday, April 14th
2. Final Projects Announced Soon
   - small groups
3. Short quiz on Friday, April 7th
   - Given two implementations, which is faster?
   - Reason about parallelism/locality of reference

Question From Last Time

Is it possible to implement a (sequentially consistent? linearizable?) queue without locks?

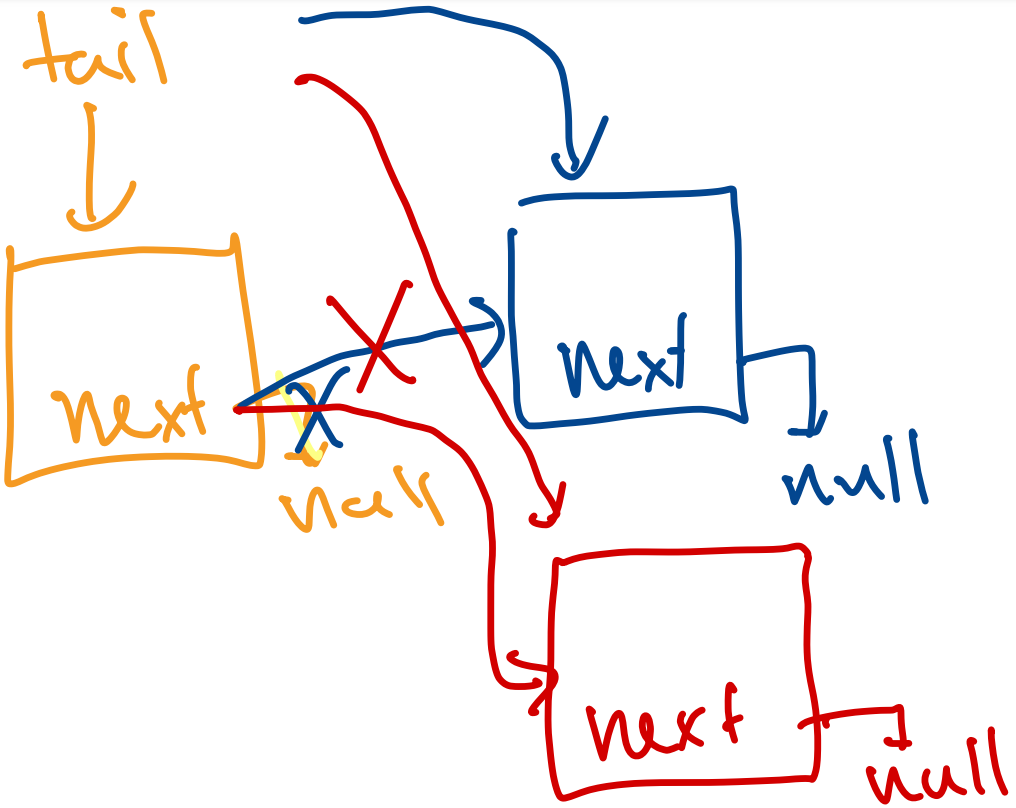
Wrap-around queue



Enqueue Without Locks

What could go wrong with concurrent enq?

```
    public void enq (T value) {
(1) → Node nd = new Node(value);
(2) → tail.next = nd;
(3) → tail = nd;
    }
```



Possible Linearization Point?

```
public void enq (T value) {
    Node nd = new Node(value);
    tail.next = nd;
    tail = nd;
}
```

elt is "logically" added to linked list & accessible to other threads

Maybe? Can see there's a potential problem if we try to set tail.next and tail.next ≠ null

New Tech: `AtomicReferences`

```
// an AtomicReference pointing to someNode
var nd = new AtomicReference<Node>(someNode);

// try to update nd to refer to updated
nd.compareAndSet(expected, update);
```

Effect of `compareAndSet(expected, update)`:

- if `nd`'s current value is `expected`, then update value to `update`
  - return `true`
- if `nd`'s current value is not `expected`, do not update its value
  - return `false`

How Could Atomic References Help?

When can('t) we update `tail.next` and `tail`?

```
public void enq (T value) {
  Node nd = new Node(value);
  tail.next = nd;
  tail = nd;
}
```

(1)
(2)
(3)

tail: atomic ref

tail.next. compareAndSet(null, nd)

↳ keep trying until success

If tail not updated, update it!

Makes it to (2) succeeds, doesn't exec (3)

Enqueue Idea

To do:

1. update `tail.next` to `nd`
2. update `tail` to `nd`

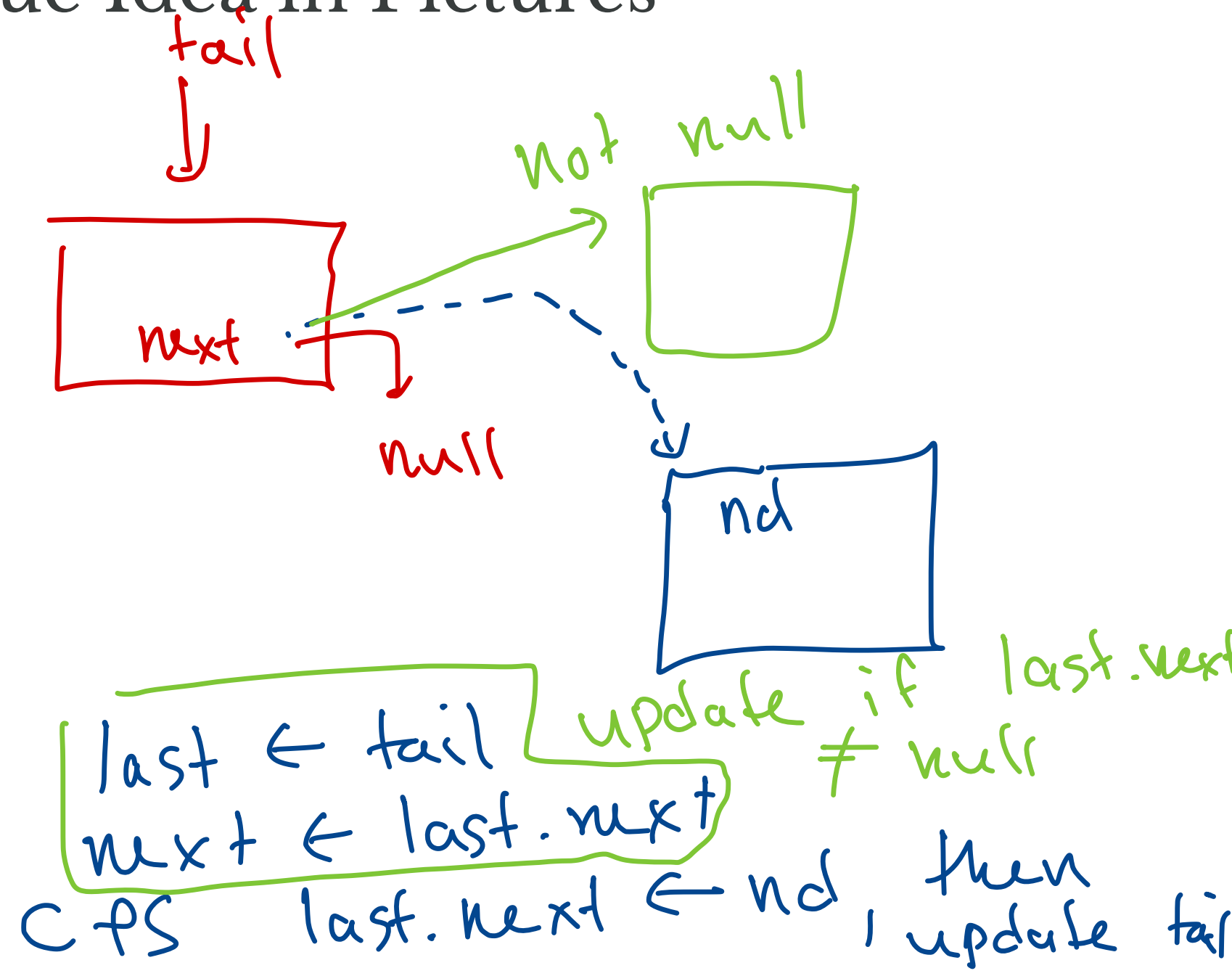
Under what conditions can we apply these?

- Can update `tail.next` *only* if `tail.next == null`
- Try to update `tail.next` to `nd`:
  1. set `last` to `tail`, `next` to `tail.next`
  2. check if `last` is still `null`
  3. update `last.next` to `nd` *only if* `last.next` is still `null`
  4. if 3 fails, try to update `tail` to `next`

↪ repeat from 1.

↓ Compare & set

Enqueue Idea in Pictures



LockFreeQueue

```
public class LockFreeQueue<T> implements SimpleQueue<T> {
    private AtomicReference<Node> head;
    private AtomicReference<Node> tail;
    ...
    public void enq(T item) {...}
    public T deq() throws EmptyException {...}
    class Node {
        public T value;
        public AtomicReference<Node> next;
        ...
    }
}
```

Lock Free enq


```
public void enq(T item) {
    if (item == null) throw new NullPointerException();
    Node node = new Node(item);
    while (true) {
        Node last = tail.get();
        Node next = last.next.get();
        if (last == tail.get()) {
            if (next == null) {
                if (last.next.compareAndSet(next, node))
                    tail.compareAndSet(last, node); return;
            } else {
                tail.compareAndSet(last, next);}}}}
```

last is still tail

last is still @ end

new elt added to queue

What happens if every other thread stops?

→ then my thread succeeds in next iteration of loop.

Linearization Point (if any)?

```
public void enq(T item) {
    if (item == null) throw new NullPointerException();
    Node node = new Node(item);
    while (true) {
        Node last = tail.get();
        Node next = last.next.get();
        if (last == tail.get()) {
            if (next == null) {
                if (last.next.compareAndSet(next, node))
                    tail.compareAndSet(last, node); return;
            } else {
                tail.compareAndSet(last, next);}}}}
```

critical

Not linearization pt because op can fail, but determines when new item is in queue

Exercise

How could we redesign deq?

```
public T deq() throws EmptyException {
        if (head.next == null){throw new EmptyException();}
        value = head.next.value;
        head = head.next;
        return value;
}
```

Comparing Progress

Which lock is “better?”

- `UnboundedQueue`?
- `LockFreeQueue`?

more "obviously" correct

Which is faster?

Deadlock?

everyone's progress depends on no one crashing

UnboundedQueue Enqueue

```
public void enq (T value) {
    enqLock.lock();
    try {
        Node nd = new Node(value);
        tail.next = nd;
        tail = nd;
    } finally {
        enqLock.unlock();
    }
}
```

LockFreeQueue Enqueue

```
public void enq(T item) {
    if (item == null) throw new NullPointerException();
    Node node = new Node(item);
    while (true) {
        Node last = tail.get();
        Node next = last.next.get();
        if (last == tail.get()) {
            if (next == null) {
                if (last.next.compareAndSet(next, node))
                    tail.compareAndSet(last, node); return;
            } else {
                tail.compareAndSet(last, next);}}}}
```

UnboundedQueue Progress

Guarantee: **Starvation Freedom** (assuming lock is starvation-free)

- if *all* pending method calls continue to take steps, then *every* pending method call completes in a finite number of steps
- this is **blocking progress:** if even one thread stops taking steps, then all other threads can be impeded

Question. When is this "good?"

LockFreeQueue Progress

Guarantee: **Lock Freedom**

- if *some* pending method call makes progress, then *some* pending method call completes in a finite number of steps
- this is **nonblocking progress:** if some threads stall, others are still guaranteed to make progress

Question. When is this "good?"

Which Guarantee Is Better

1. Starvation Freedom?
2. Lock Freedom?

It depends!

Progress, 4 Ways

Blocking Progress:

- **deadlock freedom** if *all* threads take steps, *some* completes in finite time
- **starvation freedom** if *all* threads take steps, *all* complete in finite time

Nonblocking Progress:

- **lock freedom** if *some* threads take steps, *some* completes in finite time
- **wait freedom** *all* threads taking steps complete in finite time

What About Performance?

Demo: `concurrent-queues.zip`

Coming Up

- Lock Implementations
- Concurrent Linked Lists