CSCI-GA.3033-017

Special Topics:

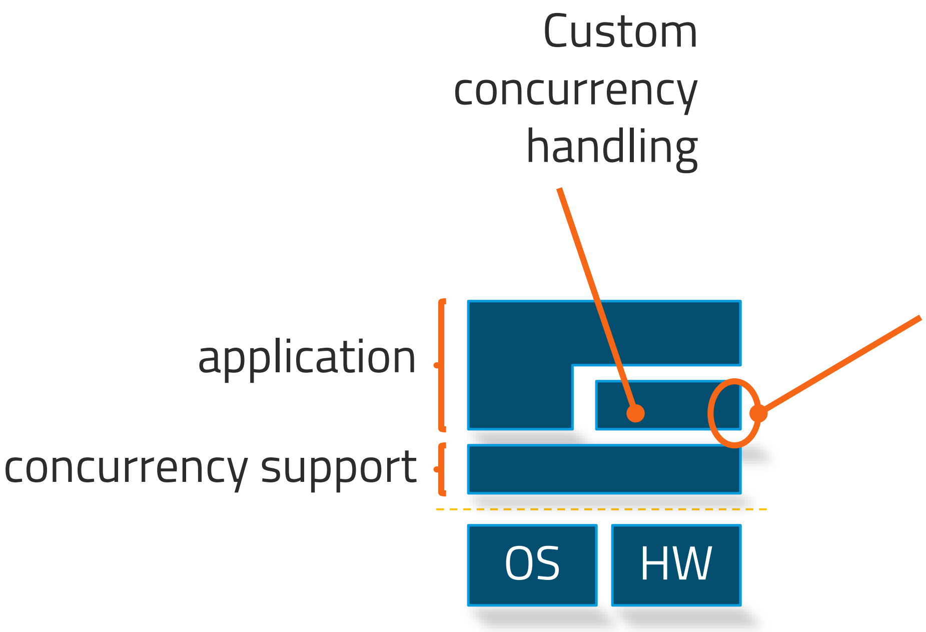
Multicore Programming

Lecture 7
Synchronized Structures Part 1

Christopher Mitchell, Ph.D.

cmitchell@cs.nyu.edu || http://z80.me

Context



We're going to look at how to create more higher level synchronized data structures

- This week: Concurrent queues
- Next week: Building up to a lock-free hash table

Outline

- Lock-Based Concurrent Queue
- Unlocking: A Lock-Free Concurrent Queue
- Understanding The ABA Problem
- Solving ABA: Memory Management and Reuse

Lock-Based Concurrent Queue

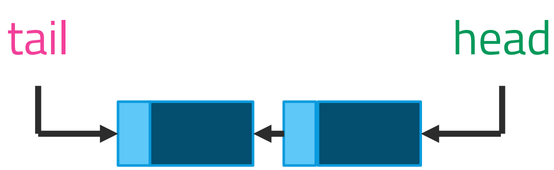
- Consider a naïve concurrent queue
  - How many locks?
  - What problems with 1? (Correctness, performance, …?)
  - What problems with 2? (Correctness, performance, …?)
- Concept Reminder: Scoped locks
  - Automatically unlocked when destroyed
  - C++ scoped lock:

```
std::lock_guard<…> scoped_lock(my_mutex);
```

Concurrent Queue:
Protecting Shared State

```
elem* dequeue() {
    lock_guard(mutex);
    elem* node = nullptr;
    if (head != nullptr) {
        node = head;
        head = head->next;
    }
    if (head == nullptr) {
        tail = nullptr;
    }
    return node;
    // mutex unlocked
}
```

```
enqueue(const& cnode) {
    lock_guard(mutex);
    elem* node =
        new elem(cnode)
    if (tail != nullptr) {
        tail->next = node;
    } else {
        head = node;
    }
    tail = node;
    // mutex unlocked
}
```



Both methods touch both head and tail.

Concurrent Queue: Considering Correctness

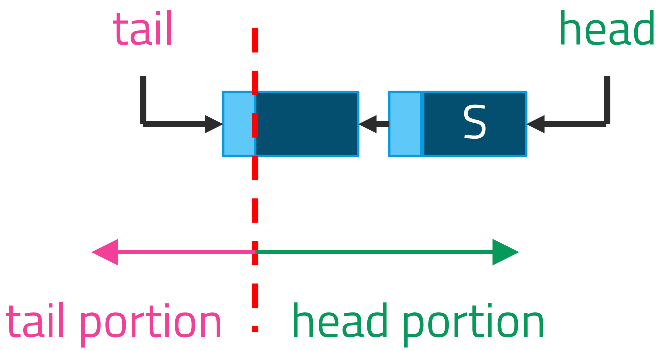
- enqueue() and deque() must be protected under the same single lock
  - Why?
- Correctness: Linearizability
  - We dealt with this in Lab 1
  - Definition: There is *some* total serial order among all operations
    - Corollary: Each operation appears instantaneous
    - Corollary: No operation can see the intermediate state of another

Concurrent Queue: Considering Correctness

- Can't split the lock as-is: no more linearizability
  - Another operation could see tail update before head update, or vice versa
- Linearization point: right after lock released
  - If there's no obvious linearization point, algorithm may be wrong, hard to reason about, or both.
- We can still split the lock and be linearizable!
  - How?

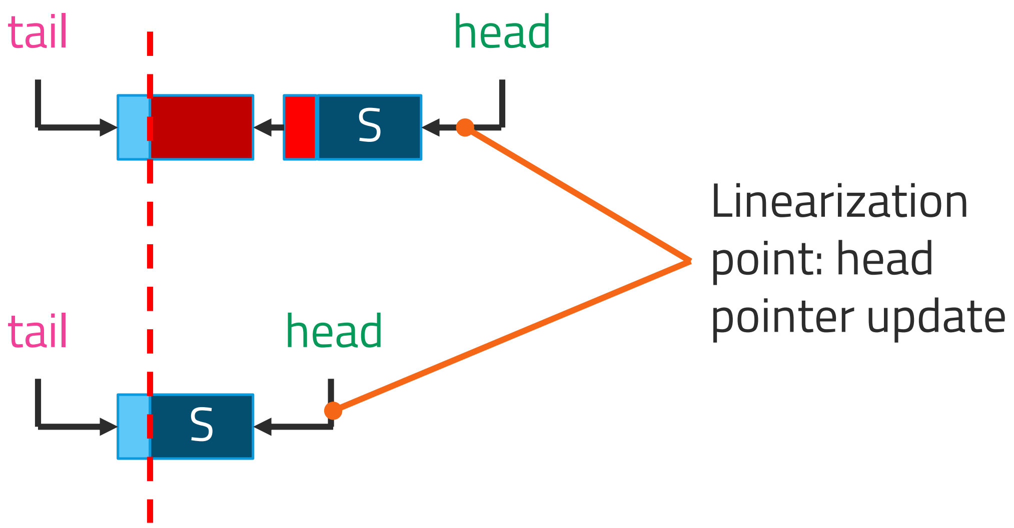
Towards a Lock-Free Concurrent Queue

- Key concept: Sentinel
  - Dummy node, always first in the list (ie, what head points to)
  - Make enqueue() and dequeue() touch distinct portions of the state
  - dequeue() checks if tail and head point to same element
  - Assume pointer reads and writes are atomic operations
- Still a blocking, locked structure



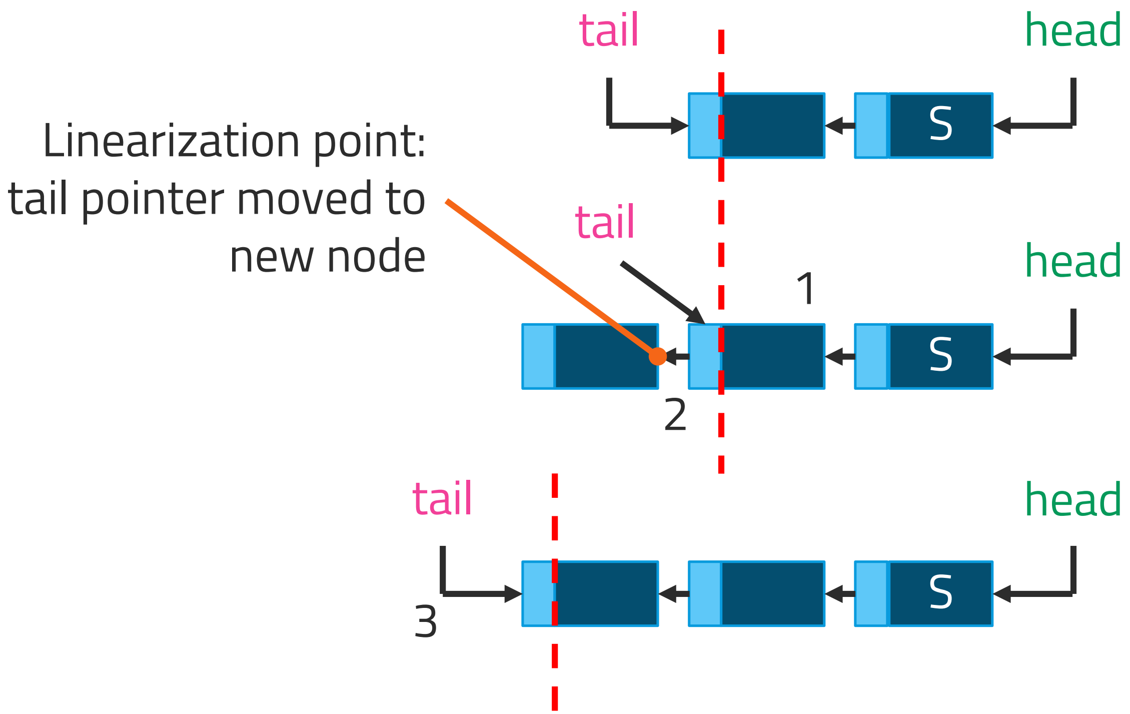
Dequeuing with Sentinel

- dequeue(): only touches head portion



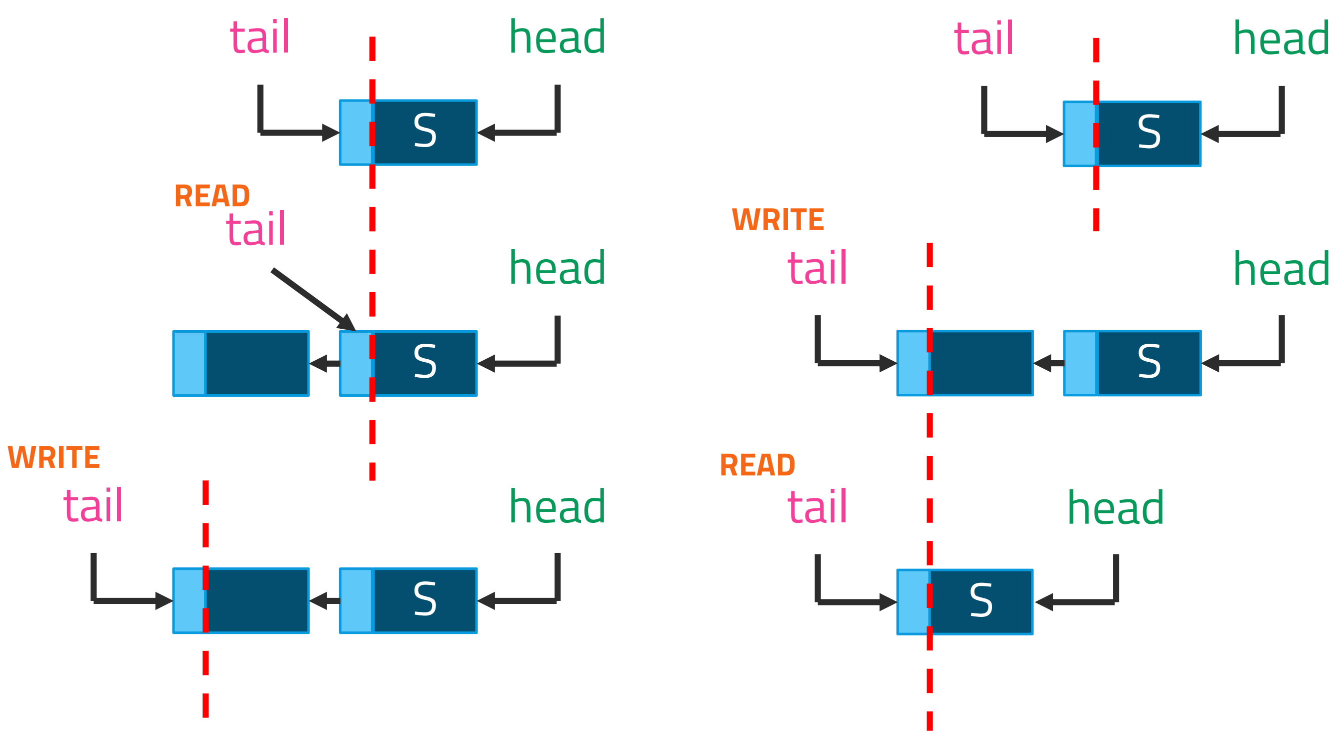
Enqueuing with Sentinel

- enqueue(): only touches tail portion



Simultaneous Enqueue and Dequeue on Empty Queue

- Depends on ordering of tail pointer write in dequeue() and read in enqueue()
- Write->write reordering would break this! (Why?)



A Lock-Free Queue

Lock-Free Queue?

- It's possible!
- Current locks: 2
  - Single concurrent enqueue()
  - Single concurrent dequeue()
  - No protection between enqueue() and dequeue()
- Necessary atomic primitive: CAS
  - Compare and Swap

Compare-And-Swap

`CAS(address, expected, desired)`

Atomically:

1. Check if `(*address == expected)`
2. If so, set `*address = desired`, return true
3. Otherwise, return false

- C++11:

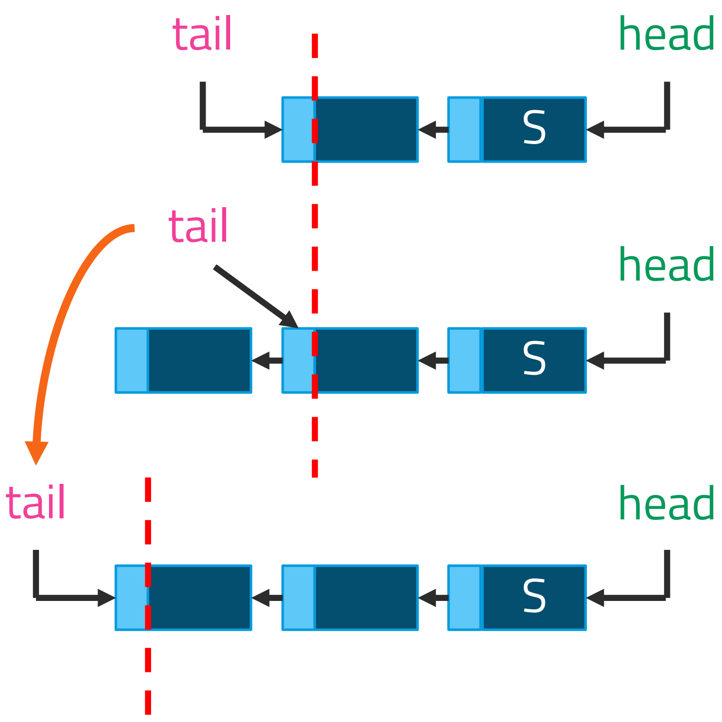
  `std::atomic<T> type` -> variable + locking!

```
template<class T>
bool atomic_compare_exchange_strong(volatile std::atomic<T>* obj,
                                    T* expected, T desired);
```

“One”-Step Lock-Free Enqueue

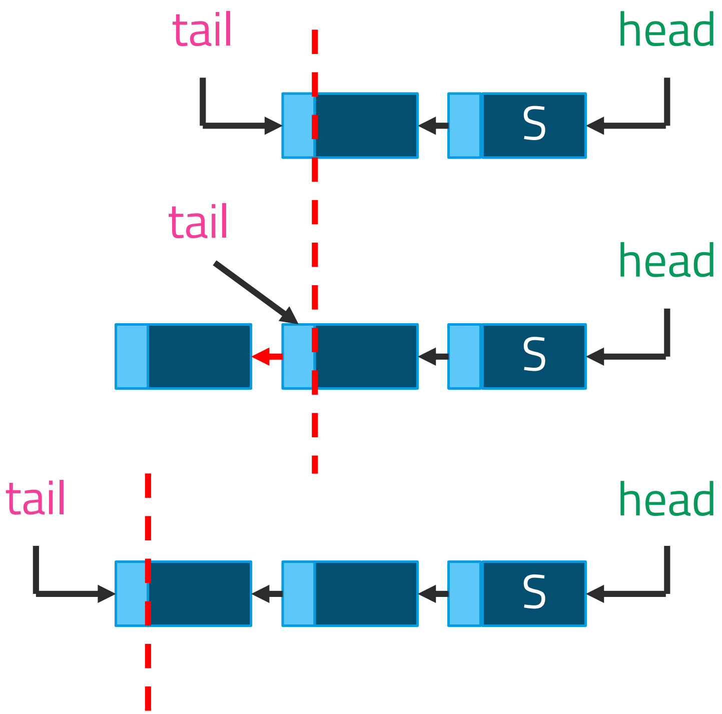
```
enqueue(elem* node) {
   …
   tail->next = node;
   CAS1{
     if (tail->next == node)
       tail = tail->next;
   }
}
```

Why can’t we use this? ☹



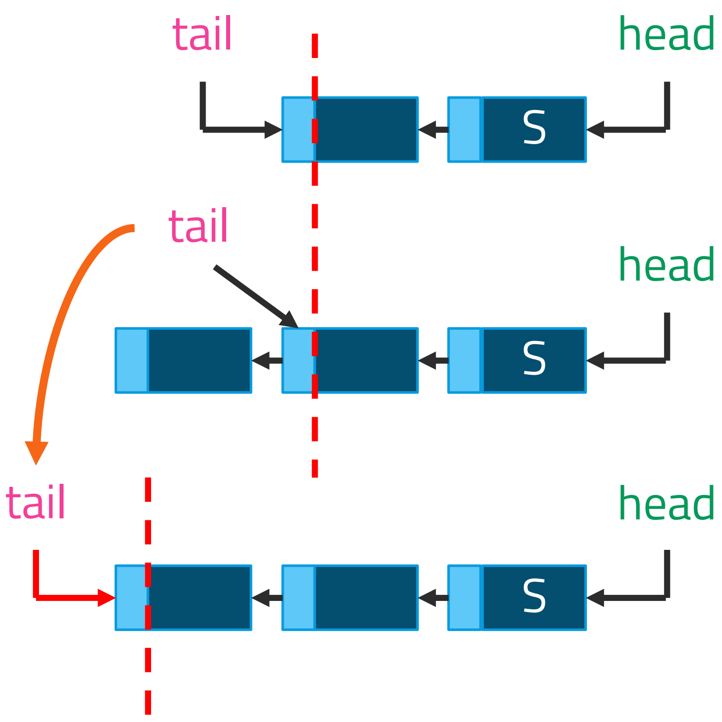
Two-Step Lock-Free Enqueue

```
enqueue(elem* node) {
  …
  elem* cur_tail = tail;
  CAS1{
    if (tail->next == cur_tail->next)
      tail->next = node;
  }
}
```



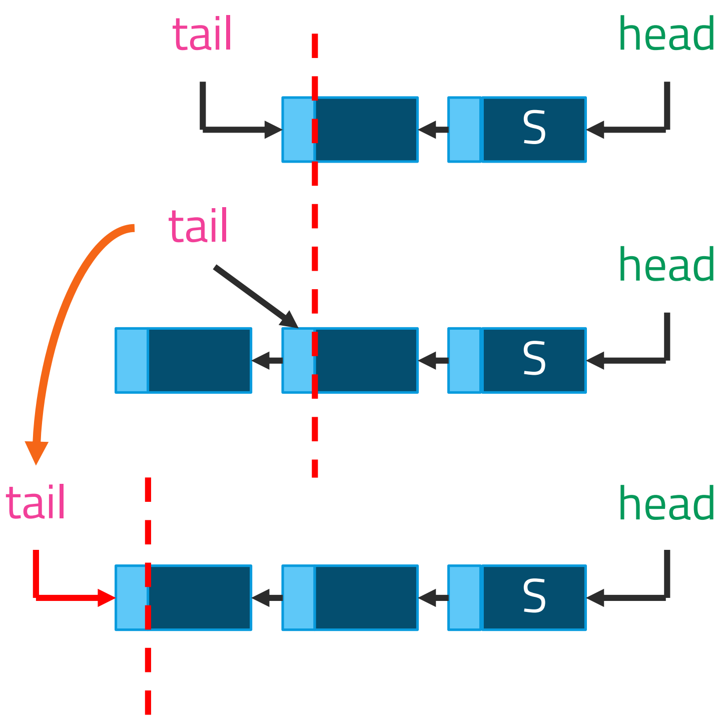
Two-Step Lock-Free Enqueue

```
enqueue(elem* node) {
  …
  elem* cur_tail = tail;
  CAS1{
    if (tail->next == cur_tail->next)
      tail->next = node;
  }
  CAS2{
    if (tail == cur_tail)
      tail = node;
  }
}
```



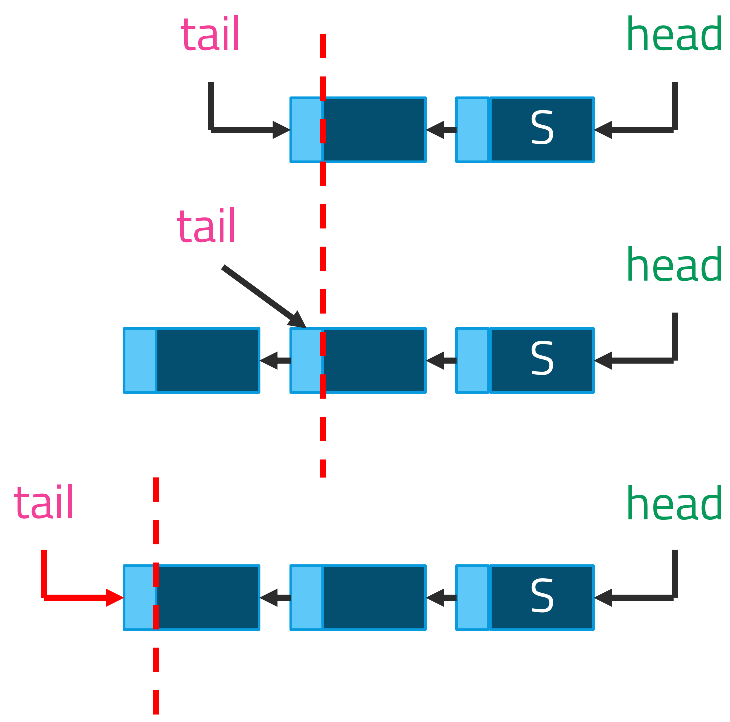
Two-Step Lock-Free Enqueue: Complete Code

```
enqueue(const value& val) {
  elem* node = new elem();
  node->val = val;
  node->next = nullptr;
  while(true) {
    elem* cur_tail = tail;
    if (CAS(tail->next, cur_tail->next,
            node))
    {
      if (CAS(tail, cur_tail, node)) {
        break;
      }
    }
  }
}
```



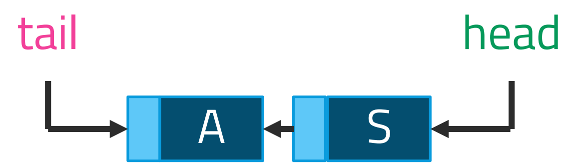
Two-Step Lock-Free Enqueue Gotcha

- What happens during the two CAS operations?
- Tail might not point to last node
  - Why? On CAS2, node may not be real tail; another thread may have set tail->next to own node in CAS1.
  - New invariant: tail may only point to last node or second-to-last node

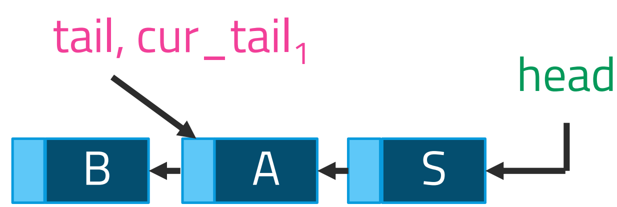


Two-Step Lock-Free Enqueue Gotcha

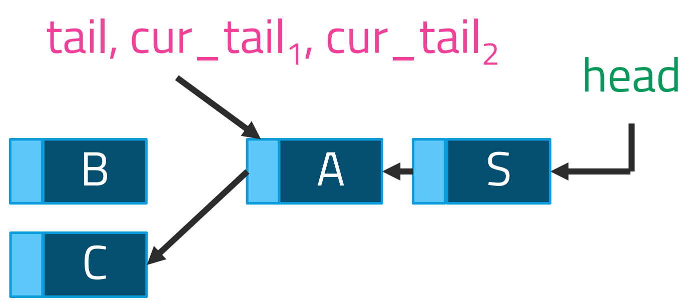
Thread 1 wants to enqueue node B
Thread 2 wants to enqueue node C



1. Thread 1 caches tail in cur_tail, performs CAS 1.

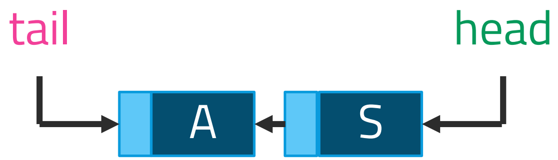


2. Thread 2 caches tail in cur_tail, performs CAS 1

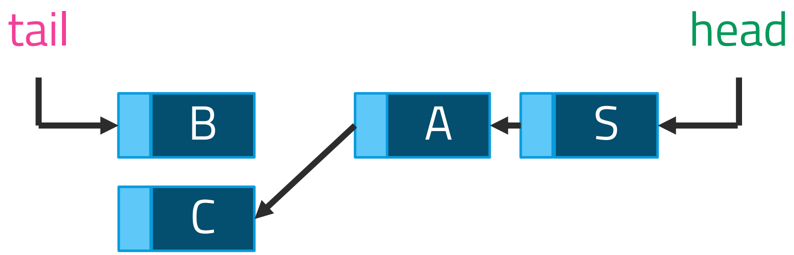


Two-Step Lock-Free Enqueue Gotcha

Thread 1 wants to enqueue node B
Thread 2 wants to enqueue node C

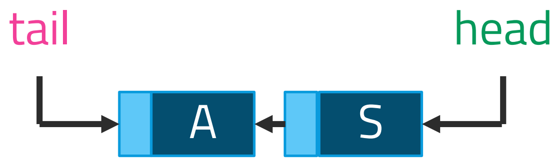


3. Thread 1 finishes enqueue with CAS 2. Tail hasn't moved, so its CAS 2 succeeds

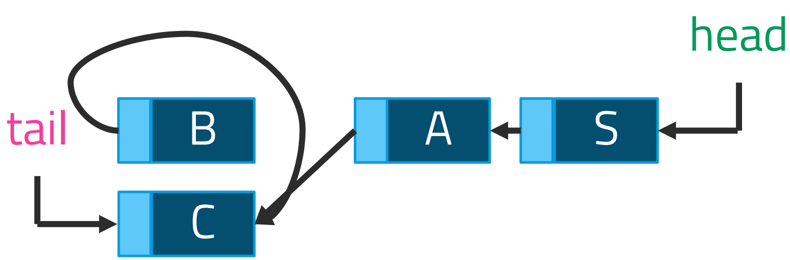


Two-Step Lock-Free Enqueue Gotcha

Thread 1 wants to enqueue node B
Thread 2 wants to enqueue node C



4. Thread 2 tries to finish enqueue with CAS 2. It fails, because tail has changed. It starts over and performs CAS 1 and CAS 2.



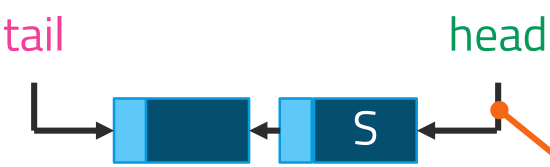
5. If enqueue checked that tail->next was nullptr before updating tail->next, this problem would be avoided, but the tail pointer could then become stale.

Two-Step Lock-Free Enqueue Fix

```
enqueue(const value& val) {
  elem* node = new elem();
  node->val = val;
  node->next = nullptr;
  while(true) {
    elem* cur_tail = tail;
    elem* next = cur_tail->next;
    if (cur_tail == tail) {
      if (next == nullptr) {
        if (CAS(tail->next, next, node)) {
          break;
        }
      } else {
        CAS(tail, cur_tail, next);  // Correct stale tail
      }
    }
  }
  // If the following fails, someone else will handle it.
  CAS(tail, cur_tail, node);
}
```

Two-Step Lock-Free Dequeue

- What if dequeue() misses a node because of invariant (ie, tail points to second-to-last node)?
  - We’ll get this on the next slide.
- General lock-free approach:




```
value dequeue() {
  while(true) {
    elem* cur_head = head;
    elem* cur_next = head->next;
    if (head == tail) {
      return null_val;
    }
    value val = head->next;
    if (CAS(head, cur_head, cur_next)) {
      free(cur_head);  // Old sentinel
      return val;
    }
  }
}
```

Two-Step Lock-Free Dequeue

```
val dequeue() {
  value val = null_val;
  elem* free_node = nullptr;
  while(true) {
    elem* cur_head = head;
    elem* cur_tail = tail;
    elem* next = cur_head->next;  // Catch half-done enqueue()!
    if (cur_head == cur_tail) {
      if (next == nullptr) {
        return nullptr;     // Empty queue
      }
      CAS(tail, cur_tail, next);   // Fix: tail->next != nullptr
    } else {
      free_node = head;
      val = next->val;
      if (CAS(head, cur_head, next) {
        break;
      }
    }
  }
  free(free_node);
  return val;
}
```

Approach: fix the tail, then dequeue

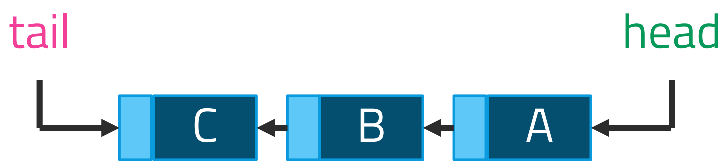
Lock-Free Queue Linearizability

- Still linearizable
- dequeue() “happens” with atomic head = head->next
- enqueue() “happens” with atomic tail->next = node
- No transient state is misleading, even the misplaced tail!
  - dequeue() and enqueue() both handle it

The ABA Problem

The ABA Problem

1. dequeue() wants to dequeue a node



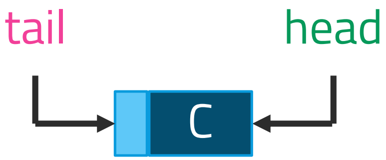

```
elem* node = next;
if (CAS(head, cur_head, next) {
  return node;
}
```

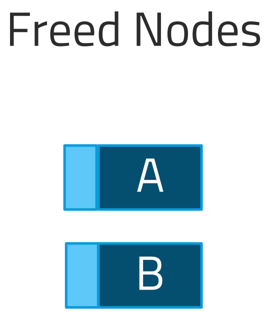
The ABA Problem

2. Another dequeue() interrupts and dequeues A

The ABA Problem

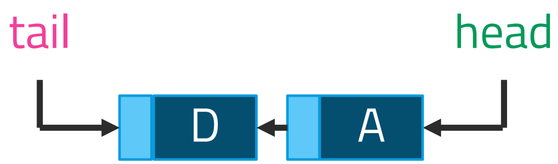
3. Another dequeue() interrupts and dequeues B

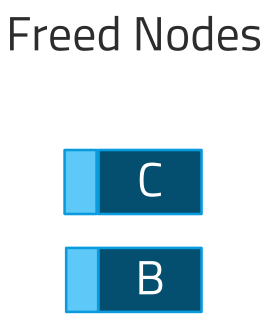




The ABA Problem

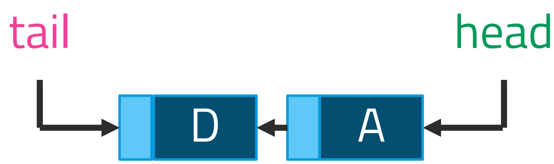
4. An enqueue() re-uses A's memory for a new node
5. [Optional: other enqueue()s/dequeue()s occur]

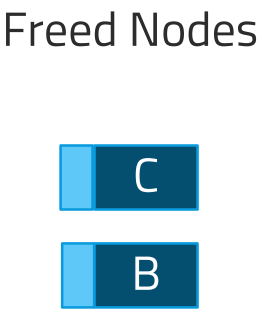




The ABA Problem

6. Finally, the original dequeue() completes






```
elem* node = next;
if (CAS(head, cur_head, next) {
  return node;
}
```

Still A, but next was B, and B was freed (and could contain anything!)

The ABA Problem

- Pointers are spatially unique, **not** also temporally unique
  - Re-using memory is necessary, but can cause unexpected problems
- If we re-use memory, we want the CAS to fail, even if the pointer points to the same memory
  1. Additional pointer information
  2. Free-list tracking

The ABA Solution

- Version our pointers
  - 128-bit pointer
    - 64 bits of target address
    - 64-bit counter
  - Requires 128-bit CAS (64-bit x86 supports this)
- Track “freed” pointers in free list with associated counter
  - Re-use freed pointers but increment counter

Lab 2 Check-In

- Lab 2: Performance testing
  - httperf: http://www.labs.hpe.com/research/linux/httperf/httperf-man-0.9.pdf
  - Must run at most one httperf client per machine
  - Try running on a different machine than your server
  - Experiment with parameters
    - We will provide a script to generate workloads

Conclusions

- Turning a single lock into multiple (smaller-scope) locks can be done
  - Careful invariant consideration
  - Linearizability checking
- Removing locks entirely is possible
  - Atomicity still needed: CAS, TAS
  - Substantial engineering effort
- Next time:
  - Lock-free ordered lists
  - Lock-free hash tables