Winter 2025

EEСS 482: Introduction to Operating Systems

Lecture 4: Thread-safe queue

Prof. Ryan Huang

Administration

Change of midterm exam time

- February 26th, 5 pm to 7 pm

- If you have a conflict with another exam, contact the staff via Piazza (if you haven't done so)

Recap: mutual exclusion

Ensure only one thread is doing something at a time

- E.g., only 1 person goes shopping at a time

Constrain interleavings of threads

- No two threads can do the certain thing *at the same time*

Allow us to have larger atomic blocks

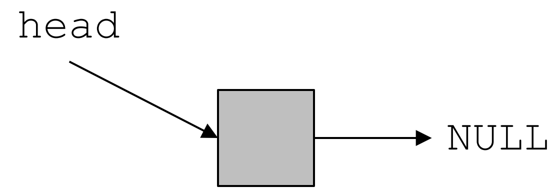
Possible (but difficult) to provide with atomic load, atomic store

High-level sync primitives like locks make life easier!

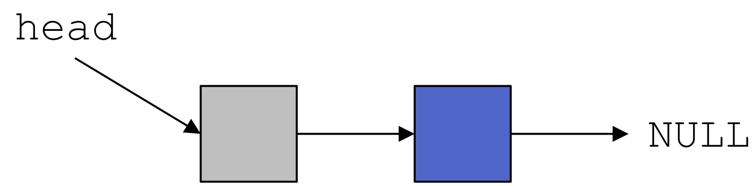
Thread-safe queue

```c
struct node {
    int data;
    struct node *next;
}
```

Empty list

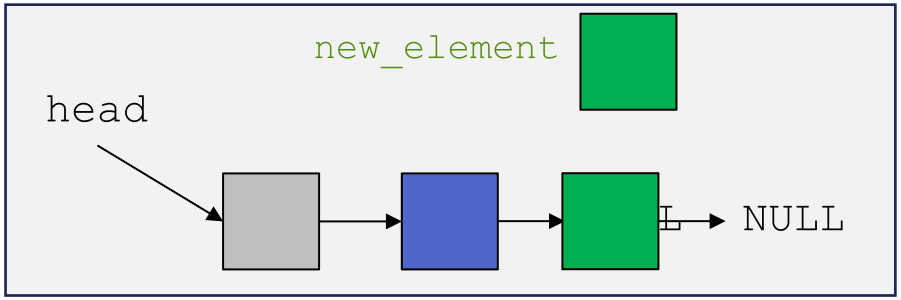


List with one node

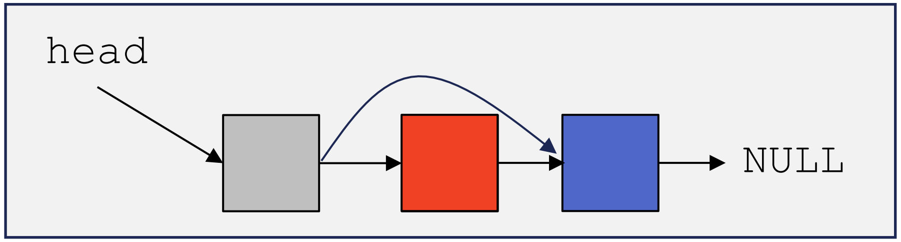


Thread-safe queue

```
enqueue(node *new_element) {
    node *ptr;
    // find tail of queue
    for (ptr = head; ptr->next != NULL; ptr = ptr->next) {}


    // add new element to tail of queue
    ptr->next = new_element;
    new_element->next = NULL;
}
```




```
node *dequeue() {
    node *ptr = NULL;
    // if something on queue, then remove it
    if (head->next != NULL) {
        ptr = head->next;
        head->next = head->next->next;
    }
    return(ptr);
}
```



Problems if two threads manipulate queue at same time?

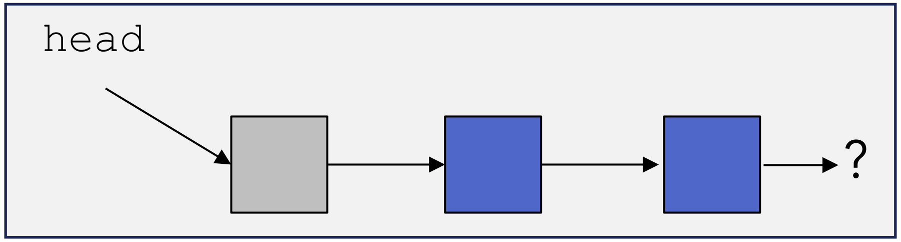
Thread-safe queue with locks

```
enqueue(node *new_element) {
    qmutex.lock();
    node *ptr;
    // find tail of queue
    for (ptr = head; ptr->next != NULL; ptr = ptr->next) {}


    // add new element to tail of queue
    ptr->next = new_element;
    new_element->next = NULL;
    qmutex.unlock();
}


node *dequeue() {
    qmutex.lock();
    node *ptr = NULL;
    // if something on queue, then remove it
    if (head->next != NULL) {
        ptr = head->next;
        head->next = head->next->next;
    }
    qmutex.unlock();
    return(ptr);
}
```

Can enqueuer() unlock anywhere?

```
enqueue(node *new_element) {
    qmutex.lock();
    node *ptr;
    // find tail of queue
    for (ptr = head; ptr->next != NULL; ptr = ptr->next) {}


    // add new element to tail of queue
    ptr->next = new_element;
    new_element->next = NULL;
    qmutex.unlock();
}
```

?
```
qmutex.unlock();
new_element->next = NULL;
```




```
node *dequeue() {
    qmutex.lock();
    node *ptr = NULL;
    // if something on queue, then remove it
    if (head->next != NULL) {
        ptr = head->next;
        head->next = head->next->next;
    }
    qmutex.unlock();
    return(ptr);
}
```

Thread-safety invariants

When can enqueue() unlock?

- Only when queue is in a safe state

"Safe state" = invariant

- Condition that is "always" true for the linked list
- Example: each node appears exactly once; last node points to NULL

Is invariant ever allowed to be false?

- Invariant may be broken only when the lock is held
- Only the thread holding the lock may break the invariant

  ➔ Hold lock whenever you're manipulating shared data

What if you are only reading the data?

What does acquiring a lock accomplish?

Before acquisition:

- acquiring a lock stops this thread from entering a critical section if another thread currently holds the lock

After acquisition:

- acquiring a lock stops other threads from entering their critical section

Does this work?

```
enqueue(node *new_element) {
    lock
    node *ptr;
    // find tail of queue
    for (ptr=head; ptr->next != NULL; ptr = ptr->next) {}
    unlock

    lock
    // add new element to tail of queue
    ptr->next = new_element;
    new_element->next = NULL;
    unlock
}
```

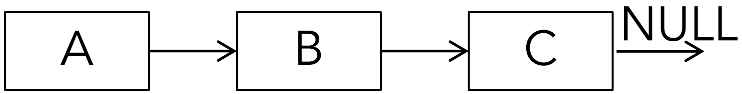
Beware of assumptions that were verified in a prior lock region.

Fine-grained locking

Instead of one lock for entire queue, use one lock per node

- Why would you want to do this?

Lock each node as the queue is traversed, then release as soon as it's safe, so other threads can also access the queue



1. lock A
2. read A
3. unlock A

4. lock B
5. read B
6. unlock B

What problems could occur?

How to fix?

1. lock A
2. read A
3. **unlock A**
4. **lock B**
5. read B
6. unlock B

Hand-over-hand locking

- Lock next node before releasing last node
- Use in Project 4

Administration

Project 1

- Most material covered (condition variables covered next)
- Set up your project environment
  - C++17 or C++20
- Avoid needing to type your password each time you compile (use password-less ssh key or ssh-agent)

What if you wanted dequeue() to wait if queue is empty?

```
node *dequeue() {

    // wait for queue to be non-empty
    while (head->next == NULL) {}

    qmutex.lock();

    // remove element
    ptr = head->next;
    head->next = head->next->next;


    qmutex.unlock();
    return(ptr);
}
```

Problems?

What if you wanted dequeue() to wait if queue is empty?

```
dequeue() {
    qmutex.lock();
    // wait for queue to be non-empty
    while(head->next == NULL) {
        qmutex.unlock();
        qmutex.lock();
    }
    // remove element
    ptr = head->next;
    head->next = head->next->next;
    qmutex.unlock();
    return(ptr);
}
```

Correct

- Lock is held continuously between seeing node on queue and dequeueing it
- Allows another thread to enqueue while dequeue() is waiting

But uses busy waiting

Waiting without busy waiting

Waiting dequeuer adds itself to set of waiting dequeuers, then "goes to sleep"

```
if (queue is empty) {
    add myself to waiting set
    go to sleep and wait for wakeup
}
```

Enqueuer wakes up sleeping dequeuer

Waiting without busy waiting

```
enqueue()
    lock
    add new item to tail of queue
    if (dequeuer is waiting) {
        remove waiting dequeuer from waiting set
        wake up dequeuer
    }
    unlock
```

```
dequeue()
    lock
    if (queue is empty) {
        unlock?
        add myself to waiting set
        unlock?
        go to sleep and wait for wakeup
        lock?
    }
    remove item from queue
    unlock
```

What's wrong here?

Two types of synchronization

Mutual exclusion ("not at the same time")

- Ensures that only one thread is in critical section
- Provided by locks

Ordering constraints ("before/after")

- One thread waits for another to do something
  - E.g., dequeuer must wait for enqueuer to add something to queue
- Provided by condition variables

Condition variables

Enable thread to sleep inside a critical section by calling cv::wait()

- Release lock
- Put thread onto waiting set
- Go to sleep

atomic

- After being woken, acquire lock when it's free

Each condition variable has a set of waiting threads

- These threads are "waiting on that condition"

Each condition variable is associated with a lock

Wake waiting threads with signal (wakes one waiting thread) or broadcast (wakes all waiting threads)

Condition variables

Condition variable = a place for threads to wait

- Threads in cv::wait are waiting on that condition

`wait(mutex)` atomic

- Release **`mutex`**, add thread to waiting set, go to sleep. After waking, re-acquire **`mutex`** (blocking if needed).
- Each condition variable is associated with a lock
- Invariant must be true when releasing lock

`signal()` and **`broadcast()`**

- Wake up one thread (signal) or all threads (broadcast) that are waiting on this condition variable
- If no thread is waiting, signal/broadcast does nothing

Thread-safe queue with condition variables

```
mutex queueMutex;
cv queueCV;
enqueue()
    queueMutex.lock()
    add new element to tail of queue

    queueCV.signal()


    queueMutex.unlock()
dequeue()
    queueMutex.lock()
    if (queue is empty) {
  while
     queueCV.wait(queueMutex)


    }
    remove item from queue
    queueMutex.unlock()
    return removed item
```

atomic

```
queueMutex.unlock()
add myself to waiting set
go to sleep
queueMutex.lock()
```

But this is still broken!
Always use while around wait!

Who is responsible for ensuring condition is met?

When waiter is woken, it doesn't hold the lock

- So, it must re-check the condition it was waiting for
- When could it avoid re-checking?

Most (all?) languages and operating systems use such condition variables

- Waiter is responsible for ensuring condition is met
- Signaller is responsible for telling waiter to re-check condition

Thread-safe queue with condition variables

```
mutex queueMutex;
cv queueCV;
enqueue()
    queueMutex.lock()
    add new element to tail of queue
    queueCV.signal()
    queueMutex.unlock()


dequeue()
    queueMutex.lock()
    while (queue is empty) {
      queueCV.wait(queueMutex)
    }
    remove item from queue
    queueMutex.unlock()
    return removed item
```

Compare busy waiting and cv.wait

```
lock

. . .

while (queue is empty) {

    unlock

    lock

}

. . .

unlock
```

```
lock

. . .

while (queue is empty) {
                    unlock
    cv.wait         add thread to wait queue
                    go to sleep
                    lock
}

. . .

unlock
```