CS111ACE Section 3

Synchronization!

Attendance Form -> tinyurl.com/CS111A-allergy
PollEV (For anonymous Q+A) -> pollev.com/tripmaster419
Game Plan

- Favorite Spot on Campus?
- Case Study on Threads
- [Review] The Dispatcher
- Synchronization Formalism + Implementing Locks
- (If we have time) Condition Variables

We’ll talk about deadlock when we discuss scheduling next week!
Game Plan

- What is your favorite spot on campus?
[Review] Race Conditions

A Race Condition is a **bug** that occurs when 2 or more threads interact with **shared** memory.
A Race Condition is a bug that occurs when 2 or more threads interact with shared memory.

In this example, Thread 1 and Thread 2 share a variable called counter, whose value starts at 1.
A Race Condition is a **bug** that occurs when 2 or more threads interact with **shared** memory.

In this example, Thread 1 and Thread 2 share a variable called `counter`, whose value starts at 1.

In code, this just looks like:

```c
counter++;  
```

in both threads!
Multithreading Example

- Take a look at the following function. You may assume two threads call this function concurrently. Is there a race condition?

You may assume K_NUM_FULL is some large constant.

```c
struct Data {
    int num1 = 0;
    int num2 = 0;
};

void fillStruct (struct Data &d) {
    while (true) {
        if (d.num1 < K_NUM_FULL) {
            d.num1++;
        } else if (d.num2 < K_NUM_FULL) {
            d.num2++;
        } else {
            break;
        }
    }
}
```
Scheduling

- As you can see, the OS is responsible for moving threads *on and off* the CPU very quickly!
- Under what circumstances does the **dispatcher** get invoked (which switches a thread off of the CPU?)

![Diagram showing threads and cores](image.png)
Dispatching

The dispatcher may get invoked for the following reasons:
Dispatching

The dispatcher may get invoked for the following reasons:

1. The thread has finished its execution (func from thread(func) has completed)
Dispatching

The dispatcher may get invoked for the following reasons:

1. The thread has finished its execution (**func** from `thread(func)` has completed)
2. The thread can “**Trap**” into the OS
   a. Any time a thread does something that requires OS help (unknowingly), the OS will take over to help out.
Dispatching

The dispatcher may get invoked for the following reasons:

1. The thread has finished its execution (\texttt{func} from \texttt{thread(func)} has completed)
2. The thread can “Trap” into the OS
   a. Any time a thread does something that requires OS help (unknowingly), the OS will take over to help out.
      i. Errors (seg fault)
      ii. System calls (read, write, fork, etc.)
      iii. Page Faults (we’ll learn about these later)
Dispatching

The dispatcher may get invoked for the following reasons:

1. The thread has finished its execution (*func* from `thread(func)` has completed)
2. The thread can “Trap” into the OS
3. The thread can *give up* the core willingly (when??)
Dispatching

The dispatcher may get invoked for the following reasons:

1. The thread has finished its execution (func from thread(func) has completed)
2. The thread can “Trap” into the OS
3. The thread can give up the core willingly (waiting)
4. The thread can be interrupted by an outside event.
   a. Some important I/O arrives (like mouse click) / OS replaces thread
Dispatching

The dispatcher may get invoked for the following reasons:

1. The thread has finished its execution (func from thread(func) has completed)
2. The thread can “Trap” into the OS
3. The thread can give up the core willingly (waiting)
4. The thread can be interrupted by an outside event.
   a. Some important I/O arrives (like mouse click) / OS replaces thread
   b. Some scheduling algorithms use a timer to preempt + switch threads every time the timer fires.

In order to ensure *fairness* and prevent *starvation*, the OS tries to schedule threads as evenly as possible. This usually involves the OS needing to remove running threads from the CPU to give other threads CPU time.
Dispatching

The dispatcher may get invoked for the following reasons:

1. The thread has finished its execution (\texttt{func} from \texttt{thread(func)} has completed)
2. The thread can “Trap” into the OS
3. The thread can \textit{give up} the core willingly (\texttt{waiting})
4. The thread can be \textit{interrupted} by an \texttt{outside event}.
   a. Some important I/O arrives (like mouse click) / \texttt{OS replaces thread}
   b. Some scheduling algorithms use a timer to preempt + switch threads every time the timer fires.

In order to ensure \textit{fairness} and prevent \textit{starvation}, the OS tries to schedule threads as evenly as possible. This usually involves the OS needing to remove running threads from the CPU to give other threads CPU time.
Game Plan

● Favorite Spot on Campus?
● Case Study on Threads
● [Review] The Dispatcher
● Synchronization Formalism + Implementing Locks
● (If we have time) Condition Variables
Announcements

● As a reminder, my Office Hours are Thursday mornings from 9-11AM at the 2nd floor of Old Union

● The Canvas for CS111A is now up! Videos from our sections will be posted there
  ○ I forgot to get audio for our first section :sad, but I promise I will do better this week!

● This Thursday, 4/20, the CS198 (Section Leading Program) will be hosting an informational session to give prospective applicants a look into what the job is like! There will be Boba available for the first few people
  ○ The event is at 8PM and it’s in 200-030! I’ll send a flyer out in our recap email
Game Plan

- Favorite Spot on Campus?
- Case Study on Threads
- [Review] The Dispatcher
- Synchronization Formalism + Implementing Locks
- (If we have time) Condition Variables
Synchronization Formalism

- To make a critical section atomic, we use something called a mutex
Synchronization Formalism

- To make a **critical section atomic**, we use something called a **mutex**

  **Critical Section**: A region of code susceptible to race conditions

  **Atomic**: Safe from race conditions
Synchronization Formalism

- To make a critical section atomic, we use something called a mutex.

Critical Section: A region of code susceptible to race conditions

Specifically, it’s defined by a region in which data is read/written that can be shared between multiple threads.

Atomic: Safe from race conditions

Specifically, it’s a property such that the state of a region of code (i.e. the variable values) can only be observed before or after the region, but never during.

- It’s kind of like a cocoon (you can’t see what happens internally between caterpillar and butterfly stages)
Synchronization Formalism

- To make a **critical section atomic**, we use something called a **mutex**
- A **mutex** is a variable that threads share. It has two methods, `lock()` and `unlock()`
Synchronization Formalism

- To make a **critical section atomic**, we use something called a **mutex**
- A **mutex** is a variable that threads share. It has two methods, **lock()** and **unlock()**
- When one thread **locks** the shared **mutex**, any other thread that tries to **lock()** will **block** until the first thread **unlocks()** the **mutex**.
Synchronization Formalism

- What actually happens when you lock a mutex?
Synchronization Formalism

- What actually happens when you `lock` a mutex?

If you only have 1 core, all you need to do is **disable interrupts**.

If you **disable interrupts**, the dispatcher cannot preempt your execution (think of it as the ultimate lock).

```c
void Lock::lock() {
    intrDisable();
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
    intrEnable();
}

void Lock::unlock() {
    intrDisable();
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
    intrEnable();
}
```
Synchronization Formalism

- What actually happens when you lock a mutex?

If you only have 1 core, all you need to do is disable interrupts.

If you disable interrupts, the dispatcher cannot preempt your execution (think of it as the ultimate lock).

Think Pair Share Question: Why do we only set the locked flag in one case in both lock() and unlock? Could we set it in the else statements too? Why or why not?
Synchronization Formalism

- What actually happens when you lock a mutex?

On multicore machines, disabling interrupts only applies to a single core, but other cores can still mess with your data!

To get around it, we use the exchange method provided on atomic variables.
Synchronization Formalism

- What actually happens when you lock a mutex?

On multicore machines, disabling interrupts only applies to a single core, but other cores can still mess with your data!

To get around it, we use the exchange method provided on atomic variables

ex.
std::atomic<int> spinlock;
int return_val = spinlock.exchange(1);

Exchange does 2 things:
1. Write the argument value to the variable (in this case to spinlock)
2. Return the value that was in spinlock before the overwrite.
Synchronization Formalism

- What actually happens when you lock a mutex?

On multicore machines, disabling interrupts only applies to a single core, but other cores can still mess with your data!

To get around it, we use the exchange method provided on atomic variables

```cpp
std::atomic<int> spinlock;
int return_val = spinlock.exchange(1);
```

exchange() is an atomic method. What does that mean, and why is it important?

Exchange does 2 things:

1. Write the argument value to the variable (in this case to spinlock)
2. Return the value that was in spinlock before the overwrite.
Synchronization Formalism

- What actually happens when you lock a mutex?

On multicore machines, disabling interrupts only applies to a single core, but other cores can still mess with your data!

To get around it, we use the \texttt{exchange method} provided on \texttt{atomic variables}

\begin{verbatim}
std::atomic<int> spinlock;
int return_val = spinlock.exchange(1);
\end{verbatim}

\texttt{exchange()} is an \texttt{atomic} method. What does that mean, and why is it important?

We can use \texttt{exchange()} to \texttt{emulate} the effects of disabling interrupts on multicore machines!

Exchange does 2 things:

1. Write the argument value to the variable (in this case to \texttt{spinlock})
2. Return the value that was in \texttt{spinlock} before the overwrite.
void Lock::lock() {
    intrDisable();
    while (spinlock.exchange(1)) {
        /* Do nothing */
    }
    if (!locked) {
        locked = 1;
        spinlock = 0;
    } else {
        q.add(currentThread);
        currentThread->state = BLOCKED;
        spinlock = 0;
        reschedule();
    }
    intrEnable();
}

void Lock::unlock() {
    intrDisable();
    while (spinlock.exchange(1)) {
        /* Do nothing */
    }
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
    spinlock = 0;
    intrEnable();
}
Synchronization Formalism

Questions for you to think about (think + pair + share)
1. What does it mean to set spinlock to 0 in either function?
2. Why do I set spinlock twice in lock()? More specifically, why can’t I just do what I do in unlock (and set it to 0 once right before I reenable interrupts?)
3. Do we need the while(spinlock) structure in unlock too?
4. (Optional Challenge Problem) What, if anything is this code missing?

```c
void Lock::lock() {
    intrDisable();
    while (spinlock.exchange(1)) {
        /* Do nothing */
    }
    if (!locked) {
        locked = 1;
        spinlock = 0;
    } else {
        q.add(currentThread);
        currentThread->state = BLOCKED;
        spinlock = 0;
        reschedule();
    }
    intrEnable();
}
```

```c
void Lock::unlock() {
    intrDisable();
    while (spinlock.exchange(1)) {
        /* Do nothing */
    }
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
    spinlock = 0;
    intrEnable();
}
```
Condition Variables

Condition Variables let us wait for certain conditions to be met

They have 2 methods: wait() and notify()

Conceptually, wait() does 3 things (it does these things atomically):
1. Release the lock held by the current thread
2. Sleep the current thread until it is explicitly woken up by another thread
3. Reclaim the lock before returning

notify() wakes up threads that previously called wait()

- It is important to note that notify calls are not "sticky," meaning that if a wait() occurs after a notify(), the thread will not be woken up – it needs to wait for a notify() that happens after it sleeps.
When Would I use a Condition Variable?

Check out this code for the Pipe class from lecture:

```cpp
void Pipe::put(char c) {
    mutex.lock();
    while (count == SIZE) {
        mutex.unlock();
        mutex.lock();
    }
    count++;
    buffer[nextPut] = c;
    nextPut++;
    if (nextPut == SIZE) {
        nextPut = 0;
    }
    mutex.unlock();
}
```
When Would I use a Condition Variable?

Check out this code for the Pipe class from lecture:

As you can see, we need to wait for the count to get smaller than SIZE before we can put a char in the pipe!

Let’s talk about the region in the red bar here. What does it do? Feel free to discuss with a partner

- Why do we need to unlock and then lock? Isn’t that completely redundant?
- Why a while loop? Why not an if() statement?

```cpp
void Pipe::put(char c) {
    mutex.lock();
    while (count == SIZE) {
        mutex.unlock();
        mutex.lock();
    }
    count++;
    buffer[nextPut] = c;
    nextPut++;
    if (nextPut == SIZE) {
        nextPut = 0;
    }
    mutex.unlock();
}
```
When Would I use a Condition Variable?

Check out this code for the Pipe class from lecture:

As you can see, we need to **wait** for the count to get smaller than SIZE before we can put a char in the pipe!

While this approach **works**, it’s not **ideal**. Why is that? (if you have the slides, don’t look ahead :)

```cpp
void Pipe::put(char c) {
    mutex.lock();
    while (count == SIZE) {
        mutex.unlock();
        mutex.lock();
    }
    count++;
    buffer[nextPut] = c;
    nextPut++;
    if (nextPut == SIZE) {
        nextPut = 0;
    }
    mutex.unlock();
}
```
When Would I use a Condition Variable?

Check out this code for the Pipe class from lecture:

```cpp
void Pipe::put(char c) {
    mutex.lock();
    while (count == SIZE) {
        mutex.unlock();
        mutex.lock();
    }
    count++;
    buffer[nextPut] = c;
    nextPut++;
    if (nextPut == SIZE) {
        nextPut = 0;
    }
    mutex.unlock();
}
```

As you can see, we need to **wait** for the count to get smaller than SIZE before we can put a char in the pipe!

This approach uses **busy waiting**. This means that we’re in a while loop where we’re constantly doing some work (in this case locking and unlocking in the hope that someone will preempt us and decrease count). This is inefficient.
When Would I use a Condition Variable?

Check out this code for the Pipe class from lecture:

As you can see, we need to wait for the count to get smaller than SIZE before we can put a char in the pipe!

This approach uses busy waiting. This means that we’re in a while loop where we’re constantly doing some work (in this case locking and unlocking in the hope that someone will preempt us and decrease count). This is inefficient.

In a perfect world, what if we could just sleep (off the core) until someone else woke us up and told us that our condition was met?
When Would I use a Condition Variable?

Condition variables do this for us!

The way we use them is:

```cpp
void Pipe::put(char c) {
    mutex.lock();
    while (count == SIZE) {
        mutex.unlock();
        mutex.lock();
    }
    count++;
    buffer[nextPut] = c;
    nextPut++;
    if (nextPut == SIZE) {
        nextPut = 0;
    }
    mutex.unlock();
}
```
When Would I use a Condition Variable?

Condition variables do this for us!

The way we use them is:

```cpp
void Pipe::put(char c) {
    mutex.lock();
    while (count == SIZE) {
        mutex.unlock();
        mutex.lock();
    }
    count++;
    buffer[nextPut] = c;
    nextPut++;
    if (nextPut == SIZE) {
        nextPut = 0;
    }
    mutex.unlock();
}
```

This will sleep the current thread (i.e. not busy wait) until another thread notifies us!
Pipe get() and put() Side-by-Side

Discuss these two questions with a partner!
1. Why is it always correct to call notify_all(), but not notify_one()?
2. Now that we get notified when our condition is met, can we use if instead of while in get()?

There are 2 kinds of notify() calls:

**notify_one()** wakes up a single thread who called wait()

**notify_all()** wakes up ALL threads that called wait.

If you’ve written correct code, it is always correct to call notify_all() instead of notify_one(), but it’s not true the other way around.
Pipe get() and put() Side-by-Side

Let’s say I replaced the .wait() method with its actual functionality:

```cpp
char Pipe::get() {
    char c;
    mutex.lock();
    while (count == 0) {
        charAdded.wait(mutex);
    }
    count--;
    c = buffer[nextGet];
    nextGet++;
    if (nextGet == SIZE) {
        nextGet = 0;
    }
    charRemoved.notify_one();
    mutex.unlock();
    return c;
}
```

```cpp
void Pipe::put(char c) {
    mutex.lock();
    while (count == SIZE) {
        charRemoved.wait(mutex);
    }
    count++;
    buffer[nextPut] = c;
    nextPut++;
    if (nextPut == SIZE) {
        nextPut = 0;
    }
    charAdded.notify_one();
    mutex.unlock();
}
```
Pipe get() and put() Side-by-Side

Let’s say I replaced the .wait() method with its actual functionality:

```cpp
char Pipe::get() {
    char c;
    mutex.lock();
    while (count == 0) {
        charAdded.wait(mutex);
    }
    count--;
    c = buffer[nextGet];
    nextGet++;
    if (nextGet == SIZE) {
        nextGet = 0;
    }
    charRemoved.notify_one();
    mutex.unlock();
    return c;
}
```

```cpp
void Pipe::put(char c) {
    mutex.lock();
    while (count == SIZE) {
    }
    count++;
    buffer[nextPut] = c;
    nextPut++;
    if (nextPut == SIZE) {
        nextPut = 0;
    }
    charAdded.notify_one();
    mutex.unlock();
}
```

mutex.unlock()  
sleep_until_notify()  
mutex.lock()
Let’s say I replaced the .wait() method with its actual functionality: **Is this code safe? Why or why not?**

```cpp
char Pipe::get() {
    char c;
    mutex.lock();
    while (count == 0) {
        charAdded.wait(mutex);
    }
    count--;
    c = buffer[nextGet];
    nextGet++;
    if (nextGet == SIZE) {
        nextGet = 0;
    }
    charRemoved.notify_one();
    mutex.unlock();
    return c;
}

void Pipe::put(char c) {
    mutex.lock();
    while (count == SIZE) {
        count++;
        buffer[nextPut] = c;
        nextPut++;
        if (nextPut == SIZE) {
            nextPut = 0;
        }
        charAdded.notify_one();
        mutex.unlock();
    }
```
Learning Goals

After another action-packed day of CS111A, we’ve all gotten better at:

- Examining and being able to identify critical sections in code.
- Using mutexes to synchronize threads
- (Hopefully) reasoning about why we use condition variables, how they work internally, and how wait() - notify() semantics work