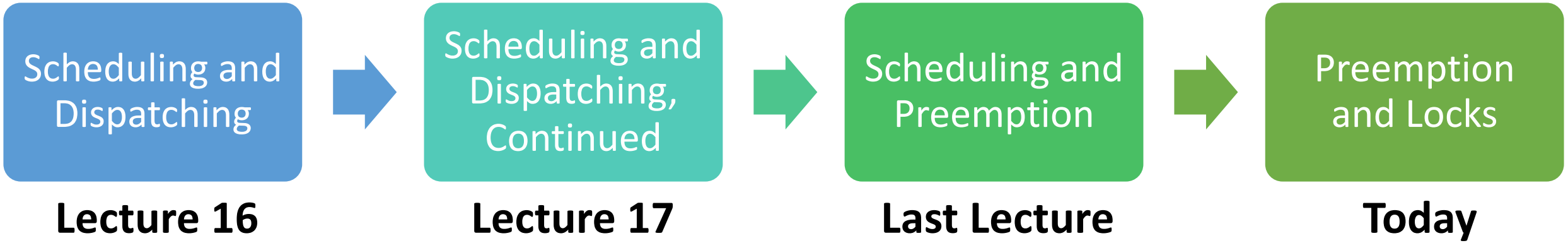
# CS111, Lecture 19
## Preemption and Implementing Locks

**Topic 3: Multithreading** - How can we have concurrency within a single process? How does the operating system support this?

# CS111 Topic 3: Multithreading, Part 2




**assign5:** implement your own version of **thread**, **mutex** and **condition_variable**!

# Learning Goals

- Learn about the assign5 infrastructure and how to implement a dispatcher with *preemption*
- See how our understanding of thread dispatching/scheduling allows us to implement locks

# Plan For Today

- **Recap:** Scheduling
- Preemption and Interrupts
- Implementing Locks

```
cp -r /afs/ir/class/cs111/lecture-code/lect19 .
```

# Plan For Today

- **Recap: Scheduling**
- Preemption and Interrupts
- Implementing Locks

```
cp -r /afs/ir/class/cs111/lecture-code/lect19 .
```

# Scheduling Algorithms

How do we decide whether a scheduling algorithm is good?

- Minimize response time (time to useful result)
  - e.g. keystroke -> key appearing, or “make” -> program compiled
  - Assume useful result is when the thread blocks or completes
- Use resources efficiently
  - keep cores + disks busy
  - low overhead (minimize context switches)
- Fairness (e.g. with many users, or even many jobs for one user)

# Scheduling

**Key Question:** How does the operating system decide which thread to run next? (e.g. many **ready** threads). Assume just 1 core.

We discussed 4 main designs:

1. **First-come-first-serve (FIFO / FCFS):** keep threads in ready queue, add threads to the back, run thread from front until completion or blocking.
2. **Round Robin:** run thread for one time slice, then add to back of queue if wants more time
3. **Shortest Remaining Processing Time (SRPT):** pick the thread that will complete or block the soonest and run it to completion.
4. **Priority-Based Scheduling:** threads have priorities, and we have one ready queue per priority. Threads adjust priorities based on time slice usage, or based on recent CPU usage (4.4 BSD Unix)

# Plan For Today

- Recap: Scheduling
- **Preemption and Interrupts**
- Implementing Locks

```
cp -r /afs/ir/class/cs111/lecture-code/lect19 .
```

# Preemption and Interrupts

On assign5, you'll implement a **dispatcher with scheduling** using the Round Robin approach.

- *Preemptive*: threads can be kicked off in favor of others (after time slice)

To implement this, we've provided a **timer** implementation that lets you run code every X microseconds.

- Fires a timer interrupt at specified interval

# Timer Demo

```
atomic<size_t> counter(0);

void timer_interrupt_handler() {
    cout << "Timer interrupt occurred with counter " << counter
         << endl;
}

int main(int argc, char *argv[]) {
    // specify microsecond interval and function to call
    timer_init(500000, timer_interrupt_handler);
    while (true) {
        counter++;
    }
}
```

**interrupt.cc**

# Timers and Preemption

**Idea:** we can use the timer handler to trigger a context switch!

(For simplicity, on assign5 we'll always do a context switch when the timer fires)

# Demo: context-switch-preemption-buggy.cc

# Interrupts

When the timer handler is called, it's called with (all) interrupts **disabled**. Why? To avoid a timer handler interrupting a timer handler.

When the timer handler finishes, interrupts are **re-enabled**.

**Problem:** because we context switch in the middle of the timer handler, when we start executing another thread **for the first time**, we will have interrupts **disabled** and the timer won't be heard anymore!

**Solution:** manually enable interrupts when a thread is first run.

# Disabling/Enabling Interrupts

The assignment starter code provides the following:

```
void intr_enable(bool on);
```

There is also a provided variable type **IntrGuard** that is like a **unique_lock** but for interrupts; it disables interrupts when created, and enables them when it is destroyed.

Interrupts are a global state – not per-thread.

# Enabling Interrupts

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!  Hello." << endl;
    }
}
```

On assign5: when a program creates a thread and gives you the function that thread should run, you will run that thread initially by enabling interrupts first and *then* running their specified function.

# Interrupts

What about when we switch to a thread that we've already run before? Do we need to enable interrupts there too?

# Enabling/Disabling Interrupts

Interrupts **ON**

## Thread #1

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

## Thread #2

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
Hello." << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **ON**

## Thread #1

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

## Thread #2

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
Hello." << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **ON**

## Thread #1

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

## Thread #2

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
Hello." << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **ON**

**Thread #1**

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

TIMER! ⏰

**Thread #2**

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
" << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **OFF**

## Thread #1

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

## Thread #2

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
Hello." << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **OFF**

## Thread #1

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

## Thread #2

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
Hello." << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **ON**

## Thread #1

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

## Thread #2

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
Hello." << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **ON**

## Thread #1

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

## Thread #2

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
Hello." << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **ON**

**Thread #1**

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

TIMER! ⏰

**Thread #2**

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
" << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **OFF**

## Thread #1

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

## Thread #2

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
Hello." << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **OFF**

## Thread #1

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

## Thread #2

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
Hello." << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **OFF**

## Thread #1

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

## Thread #2

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
Hello." << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **ON**

## Thread #1

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

## Thread #2

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
Hello." << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **ON**

## Thread #1

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

## Thread #2

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
Hello." << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **ON**

**Thread #1**

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

TIMER! ⏰

**Thread #2**

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
" << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **OFF**

## Thread #1

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

## Thread #2

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
Hello." << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **OFF**

## Thread #1

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

## Thread #2

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
Hello." << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **OFF**

## Thread #1

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

## Thread #2

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
Hello." << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **ON**

**Thread #1**

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

**Thread #2**

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
Hello." << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Enabling/Disabling Interrupts

Interrupts **ON**

## Thread #1

```
int main(...) {
    ...
    while (true) {
        cout << "I am the main thread"
             << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

## Thread #2

```
void other_func() {
    intr_enable(true);
    while (true) {
        cout << "Other thread here!
Hello." << endl;
    }
}

void timer_interrupt_handler() {
    Thread *temp = current_thread;
    current_thread =
nonrunning_thread;
    nonrunning_thread = temp;

    context_switch(*nonrunning_thread,
*current_thread);
}
```

# Interrupts

What about when we switch to a thread that we've already run before? Do we need to enable interrupts there too?

**No** – if a thread is paused, that means when it was running the timer handler was called and it context switched to another thread. Therefore, when that thread resumes, **it will resume at the end of the timer handler**, where interrupts are re-enabled.

# Yield

Another trigger that may switch threads is a function you will implement called **yield**.

- Yield is an assign5 function that can be called by a thread to give up the CPU voluntarily even though it can still do work (how considerate!)
- When you implement yield, the same idea applies for interrupt re-enabling as for the timer handler.

# Interrupts

On assign5, there are other places where interrupts can cause complications.

- E.g. we could be in the middle of adding to the ready queue, but then the timer fires and we go to remove something from the ready queue!
- This sounds like a race condition problem we can solve with **mutexes**!....right?
- **Not in this case** – because we are the OS, and *we* implement mutexes! And they rely on the thread dispatching code in this assignment.
- Therefore, the mechanism for avoiding race conditions is to enable/disable interrupts when we don't want to be interrupted (e.g. by timer).

# Plan For Today

- **Recap:** Scheduling
- Preemption and Interrupts
- **Implementing Locks**

```
cp -r /afs/ir/class/cs111/lecture-code/lect19 .
```

# Implementing Locks

Now that we understand how thread dispatching/scheduling works, we can write our own **mutex** implementation! Mutexes need to block threads (functionality the dispatcher / scheduler provides).

What does the design of a lock look like? What state does it need?

- Track whether it is locked / unlocked
- The lock "owner" (if any) – perhaps combine with first bullet
- A list of threads waiting to get this lock

# Implementing Locks

Now that we understand how thread dispatching/scheduling works, we can write our own **mutex** implementation! Mutexes need to block threads (functionality the dispatcher / scheduler provides).

What does the design of a lock look like? What state does it need?

- Track whether it is locked / unlocked
- The lock "owner" (if any) – perhaps combine with first bullet
- A list of threads waiting to get this lock

We can keep a queue of threads (for fairness). (Hint: C++ has a built-in **queue** data structure)

# Lock

1. If this lock is unlocked, mark it as locked by the current thread
2. Otherwise, add the current thread to the back of the waiting queue

```
// Instance variables
int locked = 0;
ThreadQueue q;

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();  // block/switch to next ready thread
    }
}
```

# Lock

1. If this lock is unlocked, mark it as locked by the current thread
2. Otherwise, add the current thread to the back of the waiting queue

```
// Instance variables
int locked = 0;
ThreadQueue q;

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();  // block/switch to next ready thread
    }
}
```

Wait – we could be interrupted by interrupts! (E.g. timer). We need to prevent that.

# Lock

```
// Instance variables
int locked = 0;
ThreadQueue q;

void Lock::lock() {
    intr_enable(false);
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();  // block/switch to next ready thread
    }
}
```

Where should we re-enable interrupts?

# Lock

```
// Instance variables
int locked = 0;
ThreadQueue q;

void Lock::lock() {
    intr_enable(false);
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        intr_enable(true); // ??
        blockThread();  // block/switch to next ready thread
    }
}
```

Where should we re-enable interrupts?

# Lock

```
// Instance variables
int locked = 0;
ThreadQueue q;

void Lock::lock() {
    intr_enable(false);
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        intr_enable(true); // ??
        blockThread();  // block/swit
    }
}
```

Where should we re-enable interrupts?

What possible problem would arise if we re-enabled interrupts before blocking?

**Respond with your thoughts on PollEv:** pollev.com/cs111 or text CS111 to 22333 once to join.

# What possible problem would arise if we re-enabled interrupts before blocking?

# Lock

```
// Instance variables
int locked = 0;
ThreadQueue q;

void Lock::lock() {
    intr_enable(false);
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        intr_enable(true); // ??
        blockThread();  // block/switch to next ready thread
    }
}
```

Where should we re-enable interrupts?

If we re-enable before blocking, it's possible that another thread swoops in and unlocks the lock and then we block, possibly forever.

# Lock

```
// Instance variables
int locked = 0;
ThreadQueue q;

void Lock::lock() {
    intr_enable(false);
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();  // block/swit
    }
    intr_enable(true);
}
```

We must re-enable interrupts when we get the lock. This means that once a thread *unblocks* to acquire the lock, it wakes up after **blockThread()** and re-enables interrupts. It also assumes that the thread we switch to once we block will also re-enable interrupts (e.g. maybe it was paused by a timer).

# Unlock

1. If no-one is waiting for this lock, mark it as unlocked
2. Otherwise, keep it locked, but unblock the next waiting thread

```
// Instance variables
int locked = 0;
ThreadQueue q;

void Lock::unlock() {
    if (q.empty() {
        locked = 0;
    } else {
        unblockThread(q.remove()); // add to ready queue
    }
}
```

# Unlock

1. If no-one is waiting for this lock, mark it as unlocked
2. Otherwise, keep it locked, but unblock the next waiting thread

```
// Instance variables
int locked = 0;
ThreadQueue q;

void Lock::unlock() {
    IntrGuard guard;
    if (q.empty() {
        locked = 0;
    } else {
        unblockThread(q.remove()); // add to ready queue
    }
}
```

# Plan For Today

- Recap: Scheduling
- Preemption and Interrupts
- Implementing Locks

**Next time:** more about locks and condition variables

**Lecture 19 takeaway:** To implement preemption and locks, we must make sure to correctly enable and disable interrupts. Locks consist of a waiting queue and redispatching to make threads sleep.