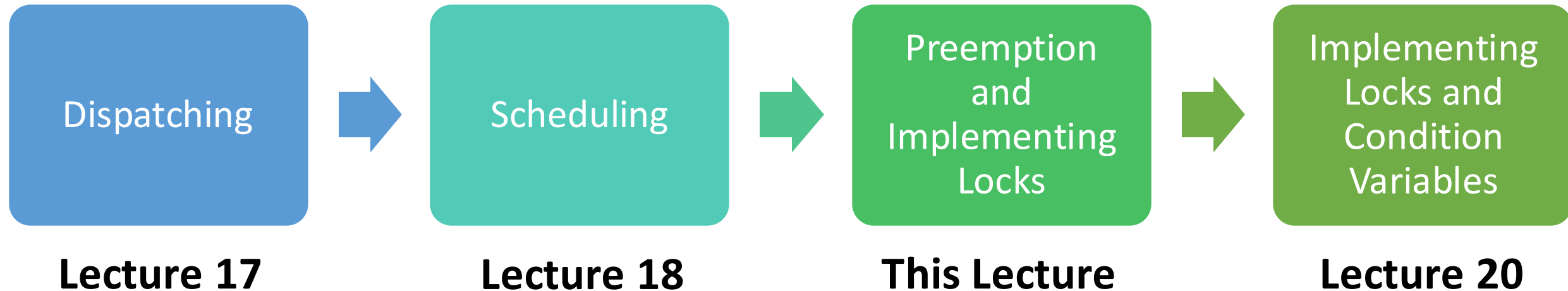


# CS111, Lecture 19

## Preemption and Implementing Locks

# CS111 Topic 3: Multithreading, Part 2

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



**assign5:** implement your own version of **thread**, **mutex** and **condition\_variable**!

# Learning Goals

- Learn about the assign5 infrastructure and how to implement a dispatcher/scheduler with *preemption*
- Understand more about how interrupts work and how they can cause race conditions
- Use our understanding of threads and interrupts 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

**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)

# Shortest Remaining Processing Time

**SRPT:** pick the thread that will finish the most quickly and run it to completion. This is the optimal solution for minimizing average response time.

What are some problems/challenges with the SRPT approach?

**Problem #1:** how do we know which one will finish most quickly? (we must be able to predict the future...)

**Problem #2:** if we have many short-running threads and one long-running one, the long one will not get to run

# SRPT

Another advantage of SRPT: improves overall resource utilization

- If a thread is **I/O-Bound** – e.g. constantly reading from disk (frequently waits for disk), it will get priority vs. thread that needs lots of CPU time – **CPU Bound**.
  - “I/O-Bound” - the time to complete them is dictated by how long it takes for some external mechanism to complete its work (disk, network)
  - “CPU-Bound” - the time to complete them is dictated by how long it takes us to do the CPU computation
  - E.g. with I/O-bound thread wanting 1ms before reading from disk vs. CPU-bound thread wanting 100ms, we will run I/O bound thread first.

Gives preference to those who need the least.



# 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 **combined scheduler+dispatcher** 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

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

# Timer Demo

```
// this program runs timer_interrupt_handler every 0.5 seconds
```

```
void timer_interrupt_handler() {  
    cout << "Timer interrupt occurred!" << endl;  
}
```

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



**interrupt.cc**

# Timer and Interrupts

We can use the timer to trigger a context switch!

- For simplicity, on assign5 we'll always do a context switch when the timer fires (e.g. even if a thread finished early, and another started early, we still switch every X ms)
- **Want to avoid:** what if the timer goes off while we are handling the timer going off?
- **Key detail:** the timer disables interrupts when running your timer handler, to avoid the timer interrupting itself. Interrupts are re-enabled once the handler finishes. ("do not disturb")
- Interrupt disabling is **global** state (not per thread), cannot be done by user programs.

# Timer Demo

// this program runs timer\_interrupt\_handler every 0.5 seconds

**Interrupts globally**

**disabled at start**

```
void timer_interrupt_handler() {  
    cout << "Timer interrupt occurred!" << endl;  
}
```

**Interrupts globally re-enabled after**

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



**interrupt.cc**

# Approximate timer.cc Code

```
void timer_interrupt() {  
    if (!enabled_flag) {  
        // defers timer alarm until later  
        interrupted = 1;  
        return;  
    }  
  
    intr_enable(false);  
    timer_handler(); // calls our timer handler  
    intr_enable(true);  
}
```

# Timer and Interrupts

We can use the timer to trigger a context switch! Let's see what this looks like.



[`context-switch-preemption-buggy.cc`](#) (and non-buggy version)

# Enabling/Disabling Interrupts

If we are switching between two already-running threads, interrupts will always be properly enabled and disabled. “Interrupt handshake”. Let’s see how!

(Note: assumption we are running on a single-core system, and we’ll focus on switching between already-running threads for now).



# Enabling/Disabling Interrupts

Interrupts  
ON

## Thread #1 (Running)

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

```
int main(int argc, char *argv[]) {  
    ...  
    while (true) {  
        cout << "I am the main thread"  
              << endl;  
    }  
}
```

## Thread #2

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

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

# Enabling/Disabling Interrupts

Interrupts  
OFF

## Thread #1 (Running)

```
→ void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}  
  
int main(int argc, char *argv[]) {  
    ...  
    while (true) {  
        cout << "I am the main thread"  
              << endl;  
    }  
}
```

## Thread #2

```
→ void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}  
  
void other_func() {  
    while (true) {  
        cout << "Other thread here!  
Hello." << endl;  
    }  
}
```

Timer! 

# Enabling/Disabling Interrupts

Interrupts  
OFF

## Thread #1 (Running)

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

```
int main(int argc, char *argv[]) {  
    ...  
    while (true) {  
        cout << "I am the main thread"  
              << endl;  
    }  
}
```

## Thread #2

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

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

# Enabling/Disabling Interrupts

Interrupts  
OFF

## Thread #1

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}  
  
int main(int argc, char *argv[]) {  
    ...  
    while (true) {  
        cout << "I am the main thread"  
              << endl;  
    }  
}
```

## Thread #2 (Running)

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}  
  
void other_func() {  
    while (true) {  
        cout << "Other thread here!  
Hello." << endl;  
    }  
}
```

# Enabling/Disabling Interrupts

Interrupts  
ON

## Thread #1

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

```
int main(int argc, char *argv[]) {  
    ...  
    while (true) {  
        cout << "I am the main thread"  
              << endl;  
    }  
}
```

## Thread #2 (Running)

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

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

# Enabling/Disabling Interrupts

Interrupts  
ON

## Thread #1

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}  
  
int main(int argc, char *argv[]) {  
    ...  
    while (true) {  
        cout << "I am the main thread"  
              << endl;  
    }  
}
```

## Thread #2 (Running)

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}  
  
void other_func() {  
    while (true) {  
        cout << "Other thread here!  
Hello." << endl;  
    }  
}
```

# Enabling/Disabling Interrupts

Interrupts  
OFF

## Thread #1

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}  
  
int main(int argc, char *argv[]) {  
    ...  
    while (true) {  
        cout << "I am the main thread"  
              << endl;  
    }  
}
```

## Thread #2 (Running)

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}  
  
void other_func() {  
    while (true) {  
        cout << "Other thread here!  
Hello." << endl;  
    }  
}
```

Timer! 

# Enabling/Disabling Interrupts

Interrupts  
OFF

## Thread #1

```
void timer_interrupt_handler() {  
    ...  
→ context_switch(*nonrunning_thread,  
                *current_thread);  
}  
  
int main(int argc, char *argv[]) {  
    ...  
    while (true) {  
        cout << "I am the main thread"  
              << endl;  
    }  
}
```

## Thread #2 (Running)

```
void timer_interrupt_handler() {  
    ...  
→ context_switch(*nonrunning_thread,  
                *current_thread);  
}  
  
void other_func() {  
    while (true) {  
        cout << "Other thread here!  
Hello." << endl;  
    }  
}
```



# Enabling/Disabling Interrupts

Interrupts  
OFF

## Thread #1 (Running)

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}  
  
int main(int argc, char *argv[]) {  
    ...  
    while (true) {  
        cout << "I am the main thread"  
              << endl;  
    }  
}
```


## Thread #2

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}  
  
void other_func() {  
    while (true) {  
        cout << "Other thread here!  
Hello." << endl;  
    }  
}
```

# Enabling/Disabling Interrupts

Interrupts  
ON


## Thread #1 (Running)



```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

```
int main(int argc, char *argv[]) {  
    ...  
    while (true) {  
        cout << "I am the main thread"  
              << endl;  
    }  
}
```

## Thread #2



```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

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

# Enabling/Disabling Interrupts

Interrupts  
ON

## Thread #1 (Running)

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

```
int main(int argc, char *argv[]) {  
    ...  
    while (true) {  
        cout << "I am the main thread"  
              << endl;  
    }  
}
```

## Thread #2

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

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

# Enabling/Disabling Interrupts

What about when a thread runs for the first time? Will interrupts be enabled?

**Key note:** new threads start running at the start of their function, not the timer handler.

# Will interrupts be enabled when Thread #2 first runs?

Interrupts  
ON

## Thread #1 (Running)

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

```
int main(int argc, char *argv[]) {  
    ...  
    while (true) {  
        cout << "I am the main thread"  
              << endl;  
    }  
}
```

## Thread #2

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

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

**Respond on PollEv:**  
[pollev.com/cs111](https://pollev.com/cs111)



## What will the interrupt state be when thread #2 first runs?

enabled: this is good so that we can switch to another thread later

0%

enabled: this is bad because the thread could be switched off soon after it starts running

0%

disabled: this is good so that the thread has time to run

0%

disabled: this is bad because the thread can never be interrupted

0%

# Enabling/Disabling Interrupts

Interrupts  
ON

## Thread #1 (Running)

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

```
int main(int argc, char *argv[]) {  
    ...  
    while (true) {  
        cout << "I am the main thread"  
              << endl;  
    }  
}
```

## Thread #2

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

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

# Enabling/Disabling Interrupts

Interrupts  
OFF

## Thread #1 (Running)

```
→ void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}  
  
int main(int argc, char *argv[]) {  
    ...  
    while (true) {  
        cout << "I am the main thread"  
              << endl;  
    }  
}
```

Timer! 

## Thread #2

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}  
  
→ void other_func() {  
    while (true) {  
        cout << "Other thread here!  
Hello." << endl;  
    }  
}
```



# Enabling/Disabling Interrupts

Interrupts  
OFF

## Thread #1 (Running)

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

```
int main(int argc, char *argv[]) {  
    ...  
    while (true) {  
        cout << "I am the main thread"  
              << endl;  
    }  
}
```

## Thread #2

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

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

# Enabling/Disabling Interrupts

Interrupts  
OFF

## Thread #1 (Running)

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

```
int main(int argc, char *argv[]) {  
    ...  
    while (true) {  
        cout << "I am the main thread"  
              << endl;  
    }  
}
```

## Thread #2 (Running)

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

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

# Enabling/Disabling Interrupts

Interrupts  
OFF

## Thread #1 (Running)

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

```
int main(int argc, char *argv[]) {  
    ...  
    while (true) {  
        cout << "I am the main thread"  
              << endl;  
    }  
}
```

## Thread #2 (Running)

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

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

**Problem:** when we start executing another thread for the first time, it won't re-enable interrupts, so the timer won't be heard anymore!

# Enabling/Disabling Interrupts

Interrupts  
OFF

## Thread #1 (Running)

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

```
int main(int argc, char *argv[]) {  
    ...  
    while (true) {
```

## Thread #2 (Running)

```
void timer_interrupt_handler() {  
    ...  
    context_switch(*nonrunning_thread,  
                  *current_thread);  
}
```

```
void other_func() {  
    while (true) {  
        cout << "Other thread here!"
```

**Existing thread:** freeze frame is in timer handler, interrupts automatically re-enabled on exit.

**New thread:** freeze frame is at start of function, no automatic re-enabling of interrupts!

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

# Enabling Interrupts

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

```
void other_func() {  
    intr_enable(true); // provided func to enable/disable  
    while (true) {  
        cout << "Other thread here! Hello." << endl;  
    }  
}
```

You'll need to do this on assign5 when a thread is first run. *Only necessary for new threads*; if a thread is paused that means when it was previously 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.

# Interrupts So Far

- Interrupts can be turned on and off globally
- When the timer fires, it disables interrupts while the timer handler is running, and re-enables them after
- We must make sure that the *new* thread always enables interrupts when it is switched to

# 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).

# 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; // blocked threads, not ready threads
```

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

# 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() {
```

```
    if (q.empty()) {
```

```
        locked = 0;
```

```
    } else {
```

```
        unblockThread(q.remove()); // add to ready queue
```

```
    }
```

```
}
```

Mutex remains locked. And new owner may not run immediately!

# Implementing Locks

With our understanding of threads and how they are run and switched between, we can understand how a mutex works – cool!

**Question:** could race conditions occur in our mutex implementation?

**Yes.** We can be interrupted at any time to switch to another thread.

We can have race conditions *within the thing that helps us prevent race conditions?* How are we supposed to fix *that*?

- We can't use a mutex, because we're writing the code to implement it!

More next time...

# Recap

- Recap: Scheduling
- Preemption and Interrupts
- Implementing Locks

**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 context switching to make threads sleep.

**Next time:** More about mutex and condition variable implementations