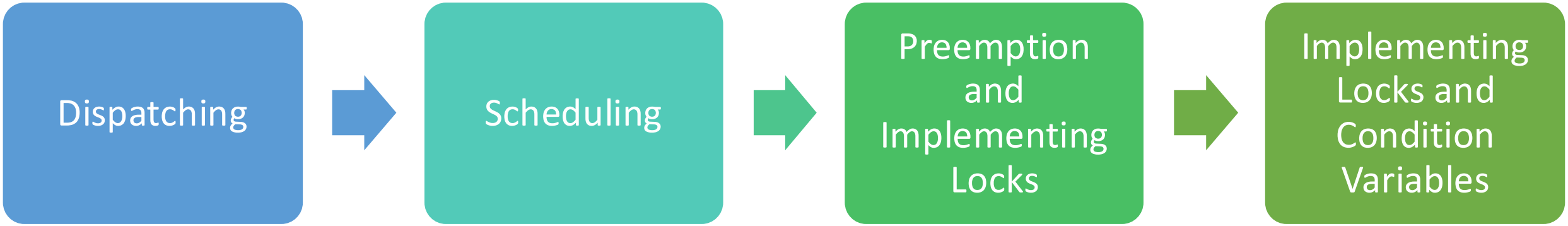
CS111, Lecture 19
Preemption and Implementing Locks

This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides and notes created by John Ousterhout, Jerry Cain, Chris Gregg, and others.
NOTICE RE UPLOADING TO WEBSITES: This content is protected and may not be shared, uploaded, or distributed. (without expressed written permission)

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.

Also improves overall resource utilization – I/O bound threads can be prioritized over CPU-bound threads.

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

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 do this, we need to dig further into **interrupts**.

Regaining Control

There are several ways control can switch back to the OS:

1. "Traps" (events that require OS attention):
    1. System calls (like **read** or **waitpid**)
    2. Errors (illegal instruction, address violation, etc.)
    3. Page fault (accessing memory that must be loaded in) – more later…
2. **"Interrupts" (events occurring outside current thread):**
    1. Character typed at keyboard
    2. Completion of disk operation
    3. Timer – to make sure OS eventually regains control

At this point, OS could then decide to run a different thread.

Interrupts

An interrupt is a ”system notification” from hardware or software that something has happened - *interrupts* the CPU and what it was currently doing.

- Examples: character typed at a keyboard, disk op completed, hardware timer fires
- OS sets up code to run when different interrupts happen (“interrupt vector”)

Interrupts

An interrupt is a ”system notification” from hardware or software that something has happened - *interrupts* the CPU and what it was currently doing.

- Examples: character typed at a keyboard, disk op completed, **hardware timer fires**
- OS sets up code to run when different interrupts happen (“interrupt vector”)

We’ve provided a **timer** implementation that lets you run code every X microseconds.

- Fires a “timer interrupt” at specified interval

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

Interrupts

An interrupt is a ”system notification” from hardware or software that something has happened - *interrupts* the CPU and what it was currently doing.

- Examples: character typed at a keyboard, disk op completed, **hardware timer fires**
- OS sets up code to run when different interrupts happen (“interrupt vector”)

Key Idea: we can use this timer to implement round robin. Context switch when the timer fires! (For simplicity, on assign5 we’ll always do a context switch when the timer fires - even if a thread finished early, and another started early, we still switch every X ms)

Disabling Interrupts

Key idea: sometimes, we won't want interrupts to come in.

Example: when we're in the middle of handling an interrupt! Risk of infinite interrupt chain.

The OS (and only the OS) can **temporarily *disable* interrupts.** (like "do not disturb" for interrupts).

- User programs cannot disable interrupts (what would happen if they could?)
- Whether interrupts are on or off is global state (not per thread)
- Similar to a critical section – want to have interrupts off as little as possible (why? to avoid delaying notifications, and maintaining responsiveness)

Timer Demo

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


void timer_interrupt_handler() {
    intr_enable(false);
    cout << "Timer interrupt occurred!" << end
    intr_enable(true);
}

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

We always want this behavior; additionally, so that the *entire* interrupt handler is run with interrupts disabled, the **timer** code will be the one that disables and re-enables interrupts for us.



interrupt.cc

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) {}
}
```

Timer disables interrupts for us before running our function.

Timer re-enables interrupts when our function finishes.



interrupt.cc

Demo: Round Robin

To implement round robin, 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)`

Interrupt Enabling and Disabling

The OS (and only the OS) can **temporarily *disable* interrupts.** (like ”do not disturb” for interrupts).

Challenge: when we switch to run a user thread, interrupts need to be enabled (why?)

It turns out that this happens already in our current implementation for already-running threads! Let’s see how. (Note: we assume 1 core).

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;
    }
}
```

The Interrupt Handshake

Key idea: when going from Thread A -> Thread B, A disables and B re-enables. "Interrupt handshake".

- **A switches away via timer handler:** interrupts disabled
- **B resumes in the timer handler:** interrupts re-enabled

What about new threads? Will interrupts be enabled when they run for the first time? 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;
    }
}
```

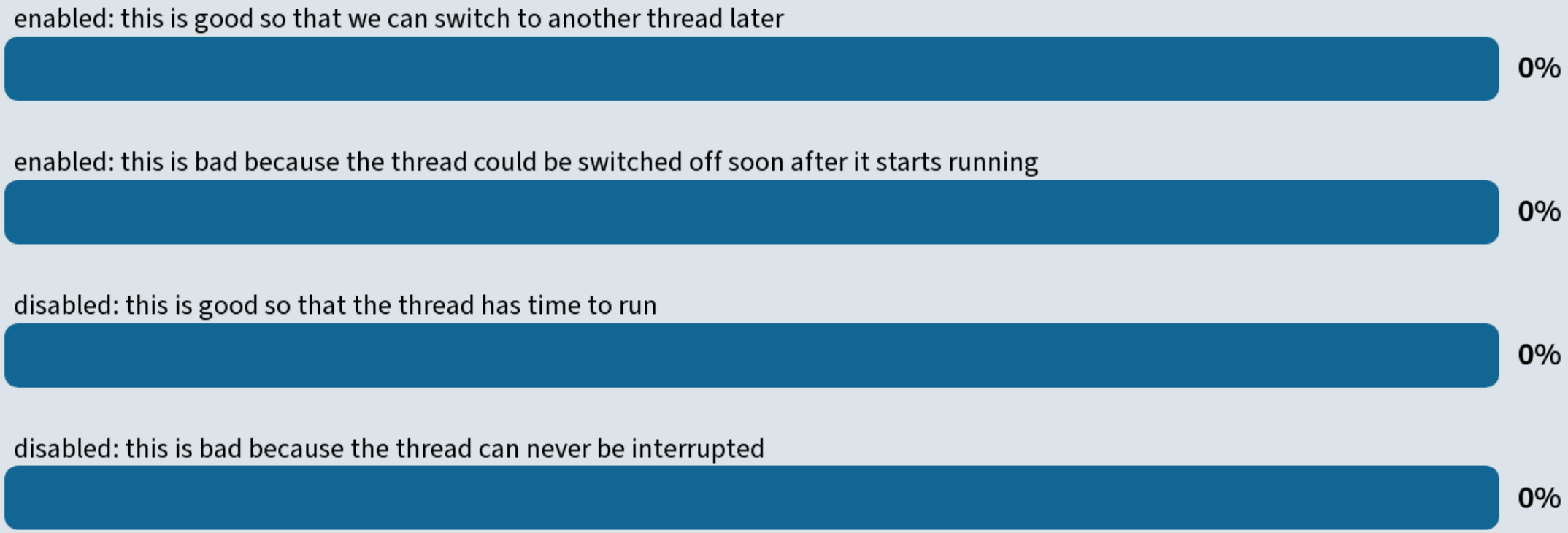
Thread #2

```
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

L19. What will the interrupt state be when thread #2 first runs?



Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

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) {
        cou

    }
}
```

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!

Enabling Interrupts

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

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

You'll need to do this on assign5. *Only necessary for new threads;* existing threads previously switched off via the timer handler, so they will resume in the timer handler and have interrupts re-enabled after returning.

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 a *new* thread always enables interrupts when it is first run
- “Interrupt Handshake” - when going from Thread A -> Thread B, A disables and B re-enables.

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