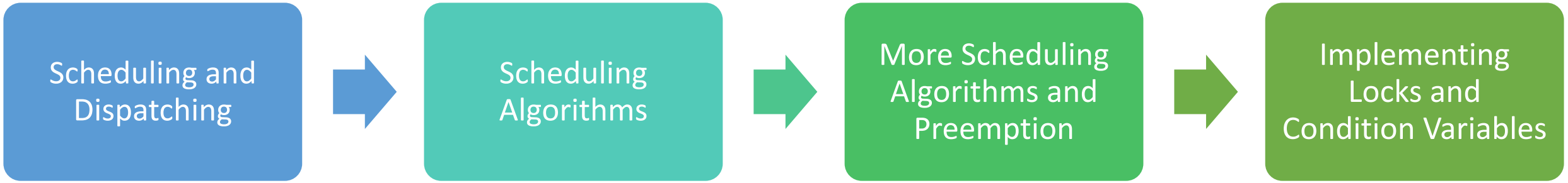
CS111, Lecture 20
Implementing Locks and Condition Variables

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



Lecture 17 **Lecture 18** **Lecture 19** **This Lecture**

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

Learning Goals

- Understand how interrupts are enabled and disabled when switching between threads
- See how our understanding of thread dispatching/scheduling allows us to implement locks

Plan For Today

- **Recap:** Preemption and Interrupts
- Implementing Locks
- Implementing Condition Variables

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

Plan For Today

- **Recap: Preemption and Interrupts**
- Implementing Locks
- Implementing Condition Variables

```
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

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)

Interrupts

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

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

```
// within timer code

// (omitted) timer disables interrupts here
your_timer_handler();
// (omitted) timer re-enables interrupts here
```

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!

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;
    }
}
```

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.

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 *restores them back to the previous state when it is destroyed*. This is the method we want to use where possible.

Disabling/Enabling Interrupts

```
void importantFunc() {
    IntrGuard guard;
    ...
}
```

IntrGuard is like unique_lock but for interrupts. It saves the current interrupt state (enabled/disabled) when it's created and turns interrupts off. When it is deleted, it restores interrupts to the saved state.

Key idea: if interrupts are already disabled when an IntrGuard is created, it keeps them disabled.

Disabling/Enabling Interrupts

```
void importantFunc() {
    intr_enable(false);
    ...
    otherFunc();
    ...
    intr_enable(true);
}

void otherFunc() {
    intr_enable(false);
    ...
    intr_enable(true);
}
```

Oops - interrupts are re-enabled here, since **otherFunc** re-enabled them!

Interrupts

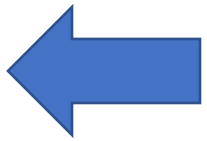
What about when we switch to a thread that we've already run before? Do we need to enable interrupts there too?

No – if a thread is paused that means when it was 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.

Enabling/Disabling Interrupts

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

void timer_interrupt_handler() {
    ...
    context_switch(...);
}
```

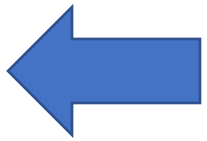


When the timer fires, we context switch to another thread

Enabling/Disabling Interrupts

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

void timer_interrupt_handler() {
    ...
    context_switch(...);
}
```



When we are switched back to, we resume right here! Then we exit the timer handler and resume the thread.

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).
- Interrupts are a global state – not per-thread.
- We're assuming a single-core machine, where disabling interrupts is sufficient to guarantee no other thread will run.

Plan For Today

- **Recap:** Preemption and Interrupts
- **Implementing Locks**
- Implementing Condition Variables

```
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
    }
}
```

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

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

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);

        // block/switch to next
        // ready thread
        blockThread();
    }
}
```

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

Can you think of an example race condition that could occur if we do not disable interrupts here and two threads lock a single mutex at the same time?

Mutex

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

void Lock::lock() {
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);

        // block/switch to next
        // ready thread
        blockThread();
    }
}
```

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

Can you think of an example race condition that could occur if we do not disable interrupts here and two threads lock a single mutex at the same time?

Example: thread 1 is in the middle of getting ownership, but then the timer fires, we switch to thread 2, and it locks the mutex. Then thread 1 resumes and *also* gets the mutex.

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
    }
}
```

Possible scenario (2 threads):

1. Thread #1 locks mutex
2. Thread #2 locks mutex, adds itself to the queue, enables interrupts
3. *Right before thread #2 blocks, thread #1 unlocks the mutex and unblocks thread #2*
4. Thread #2 then proceeds to block.
5. Nobody unblocks thread #2 ☹

Lock

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

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

Instead, we must re-enable interrupts at the end of **lock()**. This means that once a thread *unblocks* to acquire the lock, it wakes up after **blockThread()** and re-enables interrupts.

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
    }
}
```

Lock

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

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

Problem: what happens when we switch to the next ready thread? Interrupts will be disabled!

Lock

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

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

Problem: what happens when we switch to the next ready thread? Interrupts will be disabled!

Key Idea: we know that every possible way a thread resumes (e.g. timer), it will re-enable interrupts. Therefore, this isn't a problem.

Enabling/Disabling Interrupts

Interrupts **ON**

Thread #1

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

Thread #2

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

Enabling/Disabling Interrupts

Interrupts **OFF**

Thread #1

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

Thread #2

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

Enabling/Disabling Interrupts

Interrupts **OFF**

Thread #1

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

Thread #2

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

Enabling/Disabling Interrupts

Interrupts **OFF**

Thread #1

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

Thread #2

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

Enabling/Disabling Interrupts

Interrupts **ON**

Thread #1

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

Thread #2

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

Enabling/Disabling Interrupts

Interrupts **ON**

Thread #1

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

Thread #2

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

Enabling/Disabling Interrupts

Interrupts **ON**

Thread #1

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

Thread #2

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

Enabling/Disabling Interrupts

Interrupts **OFF**

Thread #1

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

Thread #2

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

Enabling/Disabling Interrupts

Interrupts **OFF**

Thread #1

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

Thread #2

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

Enabling/Disabling Interrupts

Interrupts **OFF**

Thread #1

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

Thread #2

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

Enabling/Disabling Interrupts

Interrupts **OFF**

Thread #1

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

Thread #2

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

Enabling/Disabling Interrupts

Interrupts **ON**

Thread #1

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

(assume thread 1 reenables interrupts when resumed and disables them when paused)

Thread #2 (blocked)

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

Enabling/Disabling Interrupts

Interrupts **ON**

Thread #1

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

(assume thread 1 reenables interrupts when resumed and disables them when paused)

Thread #2 (blocked)

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

Enabling/Disabling Interrupts

Interrupts **OFF**

Thread #1

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

(assume thread 1 reenables interrupts when resumed and disables them when paused)

Thread #2 (blocked)

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

Enabling/Disabling Interrupts

Interrupts **OFF**

Thread #1

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

(assume thread 1 reenables interrupts when resumed and disables them when paused)

Thread #2

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

Enabling/Disabling Interrupts

Interrupts **ON**

Thread #1

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

(assume thread 1 reenables interrupts when resumed and disables them when paused)

Thread #2

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

Enabling/Disabling Interrupts

Interrupts **OFF**

Thread #1

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

(assume thread 1 reenables interrupts when resumed and disables them when paused)

Thread #2

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

Enabling/Disabling Interrupts

Interrupts **OFF**

Thread #1

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

(assume thread 1 reenables interrupts when resumed and disables them when paused)

Thread #2

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

Enabling/Disabling Interrupts

Interrupts **ON**

Thread #1

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

(assume thread 1 reenables interrupts when resumed and disables them when paused)

Thread #2

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

Plan For Today

- **Recap:** Preemption and Interrupts
- Implementing Locks
- **Implementing Condition Variables**

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

Implementing Condition Variables

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

wait(mutex& m)

notify_one()

notify_all()

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

wait

1. Should atomically put the thread to sleep and unlock the specified lock
2. When that thread wakes up, it should reacquire the specified lock before returning

notify_one and notify_all

notify_one

- Should wake up/unblock the first waiting thread (we are guaranteeing FIFO in our implementation)

notify_all

- Should wake up/unblock **all** waiting threads

For both: if no-one waiting, does nothing.

Plan For Today

- Recap: Preemption and Interrupts
- Implementing Locks
- Implementing Condition Variables

Next time: Virtual Memory

Lecture 20 takeaway: Locks consist of a waiting queue and redispatching to make threads sleep. Condition variables also need to make threads sleep until they are notified.