CS111, Lecture 20

Implementing Locks and Condition Variables

😷 masks recommended

CS111 Topic 3: Multithreading, Part 2

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

Dispatching → Scheduling → Preemption and Implementing Locks → Implementing Locks and Condition Variables

Dispatching	Scheduling	Preemption and Implementing Locks	Implementing Locks and Condition Variables
Lecture 17	**Lecture 18**	**Lecture 19**	**This Lecture**

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

Learning Goals

- See how our understanding of thread dispatching/scheduling allows us to fully implement locks
- Understand the general design for how to implement condition variables

Plan For Today

- **Recap:** Preemption and Locks so far
- Implementing Locks
- Implementing Condition Variables
- assign5

Plan For Today

- **Recap: Preemption and Locks so far**
- Implementing Locks
- Implementing Condition Variables
- assign5

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);  // 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.

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.

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

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
    }
}
```

Race Conditions

```
// 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());
    }
}
```

One possible problem: 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.

Plan For Today

- **Recap:** Preemption and Locks so far
- **Implementing Locks**
- Implementing Condition Variables
- assign5

Locks and Race Conditions

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!
- We need to *disable interrupts* – for a single-core system, this is sufficient to guarantee that no other thread will run.

Where should we enable/disable interrupts?

```
// 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
    }
}
```

Respond on PollEv: pollev.com/cs111fall23
or text CS111FALL23 to 22333 once to join.

Where should we enable and disable interrupts?

Nobody has responded yet.

Hang tight! Responses are coming in.

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
    }
}
```

There's an air gap where we could switch to another thread after re-enabling interrupts but before we block. That other thread could be the current owner – it could unlock the mutex and mark us as ready, but then we resume and block forever!

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

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() {
    intr_enable(false);
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove()); // add to ready queue
    }
    intr_enable(true);
}
```

Problem: what if someone calls these when interrupts are already disabled? This will re-enable them, which the caller won't want!

Disabling/Enabling Interrupts

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

void Lock::unlock() {
    intr_enable(false);
    ...
    intr_enable(true);
}
```

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

Disabling/Enabling Interrupts

```
void Lock::unlock() {
    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.

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
    }
}
```

What happens when we switch to the next ready thread? Interrupts will be disabled! Is that a problem?

Lock

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

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

What happens when we switch to the next ready thread? Interrupts will be disabled! Is that a problem?

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

The Interrupt Handshake

So far, we have seen how when switching to another thread, the current thread disables interrupts before context switching, and the new thread re-enables them.

- Scenario 1: current thread enters timer handler, disables interrupts, context switches. New thread **resumes in timer handler, exits, re-enables interrupts**.
- Scenario 2: current thread enters timer handler, disables interrupts, context switches. New thread **runs for first time, but manually enables interrupts** at start.

Now, new possible way to context switch – from within **lock**. But this fits the same pattern – when switching away, interrupts are disabled, and when we switch back, we re-enable interrupts.

Enabling/Disabling Interrupts

Interrupts
ON

Thread #1 (running)

```
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 (running)

```
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 (running)

```
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 (running)

```
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 (running)

```
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 (running)

```
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();
    }
}
```

Timer!

Enabling/Disabling Interrupts

Interrupts
OFF

Thread #1 (running)

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

Enter timer handler, where interrupts are disabled at start.

Thread #2

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

Enabling/Disabling Interrupts

Interrupts **OFF**

Thread #1 (running)

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

Enter timer handler, where interrupts are disabled at start.

Thread #2

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

Resume in timer handler, where interrupts are re-enabled at end.

Enabling/Disabling Interrupts

Interrupts
ON

Thread #1

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

Thread #2 (running)

```
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 (running)

```
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 (running)

```
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 (running)

```
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 (running)

```
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();
    }
}
```

Resume in timer handler, where interrupts are re-enabled at end.

Thread #2 (blocked)

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

Enabling/Disabling Interrupts

Interrupts
ON

Thread #1 (running)

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

Thread #2 (blocked)

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

Enabling/Disabling Interrupts

Interrupts **OFF**

Thread #1 (running)

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

Thread #2 (blocked)

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

Enabling/Disabling Interrupts

Interrupts **OFF**

Thread #1 (running)

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

Thread #2

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

Enabling/Disabling Interrupts

Interrupts
ON

Thread #1 (running)

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

Thread #2

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

Enabling/Disabling Interrupts

Interrupts
ON

Thread #1 (running)

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

Thread #2

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

Timer! ⏰

Enabling/Disabling Interrupts

Interrupts **OFF**

Thread #1 (running)

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

Enter timer handler, where interrupts are disabled at start.

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());
    }
}
```

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());
    }
}
```

Thread #2

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

Interrupts

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

- 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 in our Thread and Mutex/Condition Variable implementations is to enable/disable interrupts when we don't want to be interrupted (e.g. by timer).
- 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!
- 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.

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.

Plan For Today

- **Recap:** Preemption and Locks so far
- Implementing Locks
- **Implementing Condition Variables**
- assign5

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 Locks so far
- Implementing Locks
- Implementing Condition Variables
- **assign5**

assign5

- Implement **Thread**, **Mutex** and **Condition**
- **Mutex** and **Condition** will use public methods from your **Thread** class
- Use new C++ features: **static** and **initialization lists**

Plan For Today

- Recap: Preemption and Locks so far
- Implementing Locks
- Implementing Condition Variables
- assign5

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.

Next time: Virtual Memory