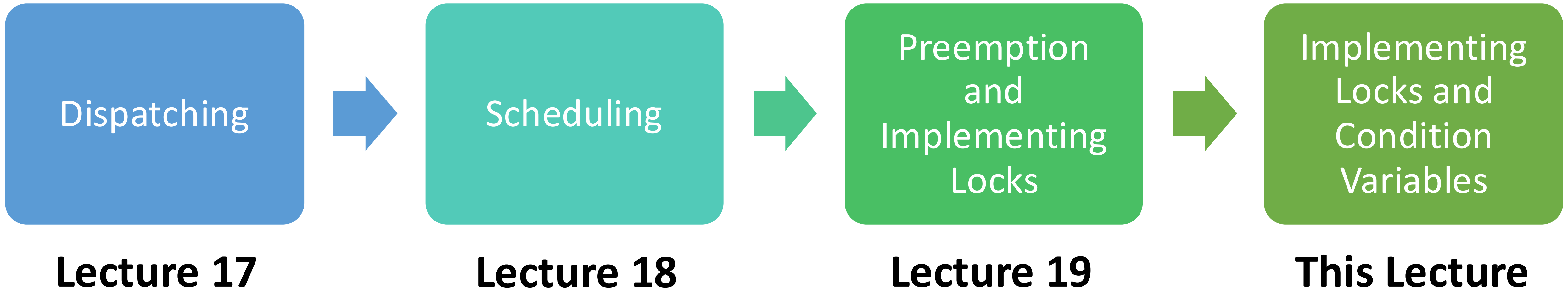
CS111, Lecture 20
Implementing Locks and Condition Variables

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
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 **<u>timer</u>** 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

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”)

Interrupts

We can register code to run whenever a timer interrupts occurs.

```
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) {
        sleep(1);
    }
}
```

Interrupts

We can use this to implement round robin - the code will trigger a context switch.

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

Problem: we don’t want a timer interrupt (or any other interrupt, for that matter) coming in right in the middle of this.
Solution: timer will disable interrupts at the start of our handler and re-enable them at the end, automatically.

Interrupts

Can imagine timer does something like this for us:

```
void timer_interrupt_handler() {
    intr_enable(false);
    ...
    context_switch(current_thread, new_thread);
    intr_enable(true);
}
```

(note: really done right before and right after the function is called)

Enabling/Disabling Interrupts

Interrupts
ON

Thread #1 (Running)

```
// timer disables interrupts, then
runs our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after


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

Thread #2

```
// timer disables interrupts, then runs
our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after


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

Enabling/Disabling Interrupts

Interrupts
OFF

Thread #1 (Running)

```
// timer disables interrupts, then
runs our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after


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

Thread #2

```
// timer disables interrupts, then runs
our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after



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

Timer! ⏰

Enabling/Disabling Interrupts

Interrupts
OFF

Thread #1 (Running)

```
// timer disables interrupts, then
runs our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after


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

Thread #2

```
// timer disables interrupts, then runs
our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after



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

Timer! ⏰

Enabling/Disabling Interrupts

Interrupts
OFF

Thread #1

```
// timer disables interrupts, then
runs our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after


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

Thread #2 (Running)

```
// timer disables interrupts, then runs
our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after


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

Enabling/Disabling Interrupts

Interrupts
ON

Thread #1

```
// timer disables interrupts, then
runs our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after


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

Thread #2 (Running)

```
// timer disables interrupts, then runs
our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after


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

Enabling/Disabling Interrupts

Interrupts
ON

Thread #1

```
// timer disables interrupts, then
runs our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after


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

Thread #2 (Running)

```
// timer disables interrupts, then runs
our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after


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

Interrupts

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!

Existing Thread

Interrupts
OFF

Thread #1

```
// timer disables interrupts, then
runs our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after


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

Thread #2 (Running)

```
// timer disables interrupts, then runs
our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after


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

Existing Thread

Interrupts
ON

Thread #1

```
// timer disables interrupts, then
runs our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after


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

Thread #2 (Running)

```
// timer disables interrupts, then runs
our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after



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

New Thread

Interrupts
OFF

Thread #1

```
// timer disables interrupts, then
runs our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after


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

Thread #2 (Running)

```
// timer disables interrupts, then runs
our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after


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

New Thread

Interrupts
OFF

Thread #1

```
// timer disables interrupts, then
runs our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after


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

Thread #2 (Running)

```
// timer disables interrupts, then runs
our function
void timer_interrupt_handler() {
    ...
context_switch(*nonrunning_thread,
               *current_thread);
}
// timer re-enables interrupts after



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

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.

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
    }
}
```

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
    }
}
```

Plan For Today

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

Lock

We can be interrupted while executing this code – for instance, say two threads try to lock at the same time. How could two threads both get ownership of the lock? (Hint: similar to ticket-selling with a conditional followed by an update)

```
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/cs111

Say two threads try to lock at the same time. How could two threads both get ownership of the lock?

Nobody has responded yet.

Hang tight! Responses are coming in.

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.

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.

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
    }
}
```

Lock

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

void Lock::lock() {
    intr_enable(false);
    if (!locked) {
        locked = 1;
        intr_enable(true);
    } 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. What problems could we potentially run into? (Hint: like condition variable problem of unlocking before waiting)

Lock

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

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

We could be interrupted here by the current owner – it could unlock the mutex and mark us as ready, but then we block!

Lock

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

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

Possible scenario (2 threads):

1. Thread #1 locks mutex
2. Thread #2 attempts to lock 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);
}
```

Two new questions:

1. What happens if someone calls lock and they already chose to disable interrupts?
2. This means when we switch to another thread, interrupts are disabled. Is that a problem?

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);
}
```

Two new questions:

1. **What happens if someone calls lock and they already chose to disable interrupts?**
2. This means when we switch to another thread, interrupts are disabled. Is that a problem?

Disabling/Enabling Interrupts

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

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

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

Lock

```
int locked = 0;
ThreadQueue q;
void Lock::lock() {
    bool interruptsEnabled = intr_enabled();
    if (interruptsEnabled) {
        intr_enable(false);
    }
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();  // block/switch to next ready thread
    }
    if (interruptsEnabled) {
        intr_enable(true);
    }
}
```

Remember whether interrupts were on before, and disable them if they are on

Lock

```
int locked = 0;
ThreadQueue q;
void Lock::lock() {
    bool interruptsEnabled = intr_enabled();
    if (interruptsEnabled) {
        intr_enable(false);
    }
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();  // block/switch to next ready thread
    }
    if (interruptsEnabled) {
        intr_enable(true);
    }
}
```

Re-enable them only if they were on before

Disabling/Enabling Interrupts

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

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
    }
}
```

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?

The Interrupt Handshake

Key idea: it's not a problem if interrupts are disabled when we switch threads, because this fits the same pattern we have already seen where, when going from Thread A -> Thread B, A disables and B re-enables.

- **Examples:**
  - **A switches away via timer handler:** interrupts disabled
  - **A switches away here via blockThread():** interrupts disabled
  - **B resumes in the timer handler:** interrupts re-enabled
  - **B is a new thread:** interrupts re-enabled
  - **B resumes and gets ownership of lock:** interrupts re-enabled

Lock

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

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

Later wake up here, exit **lock()**, re-enable interrupts because of IntrGuard

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

<u>Thread #1 (running)</u>

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

```
Enter timer handler, where
interrupts are disabled at start.
```

<u>Thread #2</u>

```
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

<u>Thread #1 (running)</u>

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

<u>Thread #2 (blocked)</u>

```
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++ feature: **static**

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