EECS 482
Introduction to Operating Systems

Winter 2018

Harsha V. Madhyastha

Recap

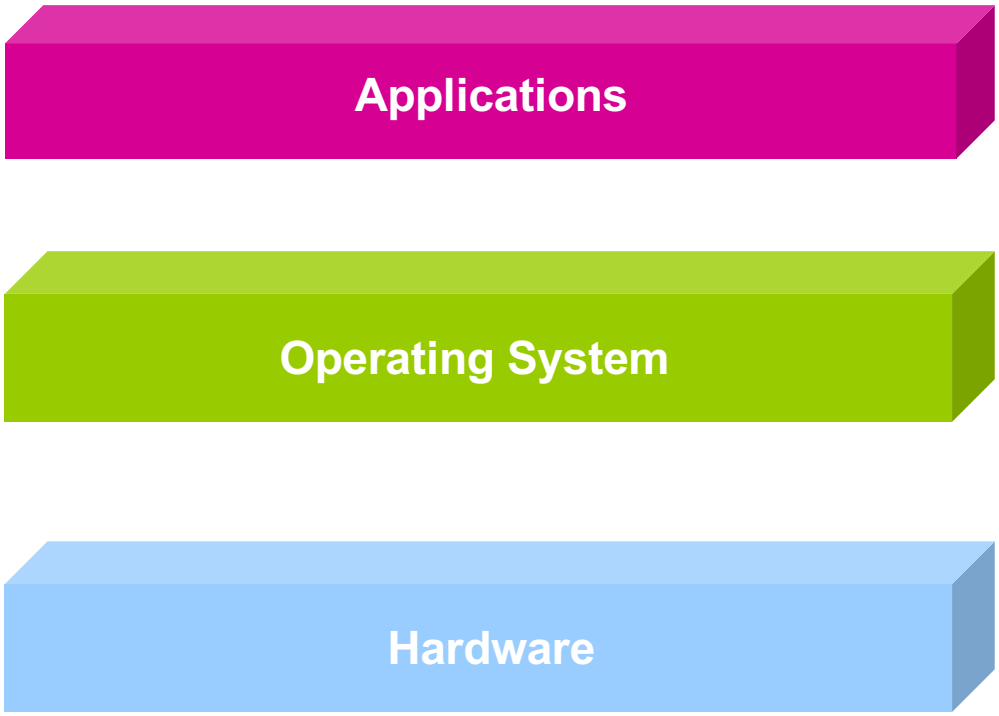
- How to handle non-running threads?
  - Save private state in TCB to resume execution later
  - TCB is on ready queue or waiting queue for lock, cv, …
- How to switch between threads?
  - Transfer control from current thread to OS
  - Copy state of current thread from CPU to its TCB
  - Load state of next thread from its TCB to CPU

Example Timeline

- T1 running on CPU1
- T1 calls wait()
  - Switch to OS code
  - Copy CPU to T1's TCB, add TCB to waitq
  - Copy T2's TCB to CPU, remove from readyq
- T2 creates thread T3
  - Switch to OS code
  - Init new TCB for T3 and add it to readyq
- T2 resumes on CPU1
- T3 starts on CPU2
- T2 calls join()
  - Switch to OS code
  - Suspend CPU1
- T3 completes
  - Switch to OS code
  - Add T2's TCB to readyq

High-level synchronization

- Raise the level of abstraction to make life easier for programmers



Concurrent programs

High-level synchronization primitives
(lock, monitor, semaphore)

Atomic operations
(load/store, interrupt enable/ disable, test&set)

Implementing high-level synchronization primitives

- Data structures used must be thread-safe
- Cannot use high-level synchronization primitives
  - Need to use atomic operations provided by hardware

Atomicity on uniprocessor

- Prevent context switches in critical section
  - No internal events (e.g., don't call yield(), lock(), ...)
  - No external events? Disable interrupts

```
disable interrupts
if (no milk) {
   buy milk
}
enable interrupts
```

```
disable interrupts
if (water leak) {
    call plumber
}
enable interrupts
```

- Problems?

```
disable interrupts
while (1);
```

Busy waiting implementation

```
lock() {

      while (status != FREE);

      status = BUSY

}
```

```
unlock() {

      status = FREE

}
```

Add atomicity

```
lock() {
        disable interrupts
        while (status != FREE);


        status = BUSY
        enable interrupts
}
```

```
unlock() {
        disable interrupts
        status = FREE
        enable interrupts
}
```

Lock implementation #1

```
lock() {
        disable interrupts
        while (status != FREE){
                enable interrupts
                disable interrupts
        }
        status = BUSY
        enable interrupts
}
```

```
unlock() {
        disable interrupts
        status = FREE
        enable interrupts
}
```

Need for other atomic primitives

- On uniprocessor, disabling interrupts prevents current thread from being switched out
- But this doesn't work on a multiprocessor
  - Other processors are still running threads
  - Not acceptable to stop all other CPUs from executing
- Could use atomic load/store
  - Example: "Too much milk" solution #3
  - But, already know this is a bad solution
  - Hard because **if (noNote) leave note** not atomic

Atomic Test-And-Set

- Atomically:
  - Set memory location to 1
  - Return previous value of location

```
test_and_set (X) {
    old = X;
    X = 1;
    return old;
}
```

- In Project 2, use *exchange* in std::atomic

Lock implementation #2

```
// status = 0 means lock is free
lock() {
      while (test_and_set(status) == 1) {
      }
}

unlock() {
      status = 0
}
```

- Only one thread sees transition from 0 to 1
- Problems?

Busy waiting

- Problem with lock implementations #1 and #2
  - Waiting thread uses lots of CPU time just checking for lock to become free
  - Better to sleep and let other threads run
- Solution: Combine lock with thread scheduling
  - lock()/unlock() manipulate thread queues
  - Waiting thread gives up CPU, so other threads can run
  - Someone wakes up thread when lock is free

Lock implementation #3

```
lock() {
        disable interrupts
        if (status == FREE) {
                status = BUSY
        } else {
                add thread to queue of threads waiting for lock
                switch to next ready thread
        }
        enable interrupts
}

unlock() {
        disable interrupts
        status = FREE
        if (any thread is waiting for this lock) {
                move waiting thread to ready queue
                status = BUSY
        }
        enable interrupts
}
```

Handoff Lock

- unlock() gives lock directly to locking thread
  - Lock status is never free during handoff

- Alternatives:
  - Set status to free, wake up all
    - » Inefficient: Only 1 thread can make progress
  - Set status to free, wake up head of waiting queue
    - » Doesn't allow specific scheduling (e.g., FIFO)

Lock implementation #3

```
lock() {
        disable interrupts
        if (status == FREE) {
                status = BUSY
        } else {
                add thread to queue of threads waiting for lock
                switch to next ready thread
        }
        enable interrupts
}
unlock() {
        disable interrupts
        status = FREE
        if (any thread is waiting for this lock) {
                move waiting thread to ready queue
                status = BUSY
        }
        enable interrupts
}
```

Lock implementation #3

```
lock() {
        disable interrupts
        if (status == FREE) {
                status = BUSY
        } else {
                add thread to ready queue
                switch to next ready thread
        }
        enable interrupts
}
unlock() {
        disable interrupts
        status = FREE
        if (any thread is waiting for this lock) {
                move waiting thread to ready queue
                status = BUSY
        }
        enable interrupts
}
```

Lock implementation #3

```
lock() {
        disable interrupts
        if (status == FREE) {
                status = BUSY
        } else {
                add thread to queue of threads waiting for lock
                switch to next ready thread
        }
        enable interrupts
}
unlock() {
        disable interrupts
        status = FREE
        if (any thread is waiting for this lock) {
                move waiting thread to ready queue
                status = BUSY
        }
        enable interrupts
}
```

Problem Scenario

- Thread L is attempting to acquire mutex
- Thread U holds mutex; is about to release it

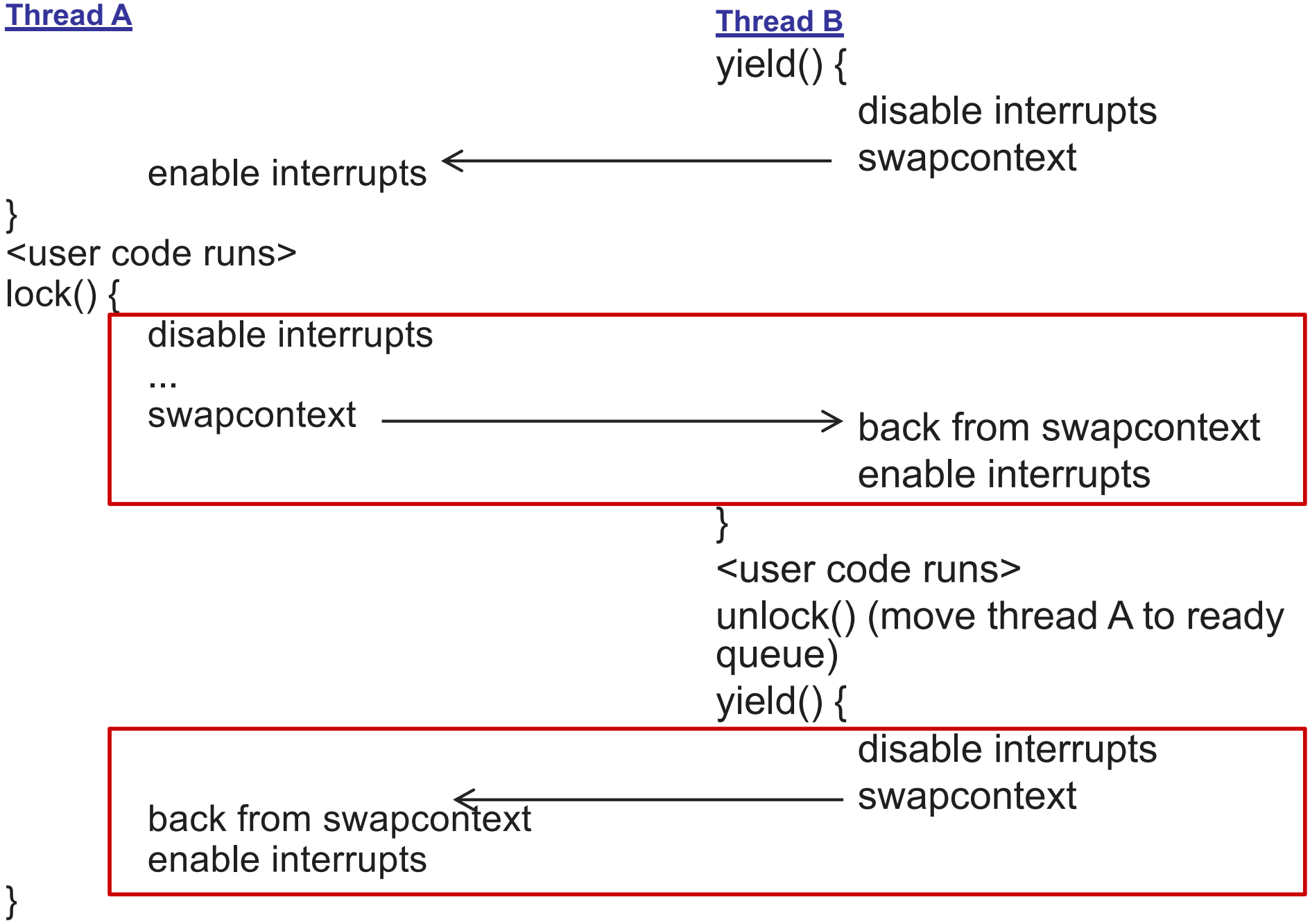
- lock() puts L's TCB on mutex's wait queue
- lock() enables interrupts
- Interrupt causes context switch to U, putting L's TCB on ready queue
- unlock() moves L's TCB to ready queue

Enabling/disabling interrupts

- Adding thread to lock wait queue + switching must be **atomic**
- Thread must leave interrupts disabled when calling switch

- Inside lock(), what causes thread to return from switch?
- What can lock() assume about the state of interrupts when switch returns?

- Switch invariant
  - All threads promise to disable interrupts before switching context
  - All threads assume interrupts are disabled when returning from switch
  - Re-enable interrupts before returning to user-level code


Thread A
Thread B
yield() {
disable interrupts
swapcontext
enable interrupts
}
<user code runs>
lock() {
disable interrupts
...
swapcontext
back from swapcontext
enable interrupts
}
<user code runs>
unlock() (move thread A to ready queue)
yield() {
disable interrupts
swapcontext
back from swapcontext
enable interrupts
}

Project 2

- Use assertions liberally
  - Assert any property that you expect to be true
  - Enables catching errors closer to where they occur

- Example:
  - At any time, any thread is either
    - » Executing on at most one CPU, or
    - » In at most one queue

Reminders

- Do homework questions on semaphores before discussion section on Friday

- Honor code:
  - Can discuss/lookup questions related to problems, project spec, or C++ syntax
  - Not okay to discuss solutions!

Locks on multiprocessors

- Disabling interrupts insufficient to ensure atomicity if there are multiple CPUs
- Need to use atomic ***test_and_set***

Lock implementation #4

```
//guard is initialized to 0
lock() {
        disable interrupts
        while (test_and_set(guard)) {}

        if (status == FREE) {
                status = BUSY
        } else {
                add thread to queue of threads waiting for lock
                switch to next ready thread
        }
        guard = 0
        enable interrupts
}
```

Lock implementation #4

```
unlock() {
        disable interrupts
        while (test_and_set(guard)) {}

        status = FREE
        if (any thread is waiting for this lock) {
                move waiting thread to ready queue
                status = BUSY
        }

        guard = 0
        enable interrupts
}
```

What's the switch invariant for multiprocessors?

Is this correct?

```
//guard is initialized to 0
lock() {
        while (test_and_set(guard)) {}
        disable interrupts

        if (status == FREE) {
                status = BUSY
        } else {
                add thread to queue of threads waiting for lock
                switch to next ready thread
        }
        guard = 0
        enable interrupts
}
```

Summary of lock solution

- High-level idea:
  - Atomically add thread to a waiting list and sleep
- How did we achieve this?
  - Disable interrupts and test_and_set(guard) to protect critical section
  - Switch to another thread and hand off task of enabling interrupts and resetting guard
- What if no other thread to run?
  - Atomically suspend CPU with interrupts enabled

Project 2

- Covered everything you need to know to implement all of the project

- Work on the project incrementally
  - CPU and thread basics (cpu::init, thread constructor, and yield)
  - Enable/disable interrupts for atomicity
  - Implement mutex and cv
  - Add support for multi-processors