

SE 350
Operating
Systems



Lecture 4: Multithreaded Kernels

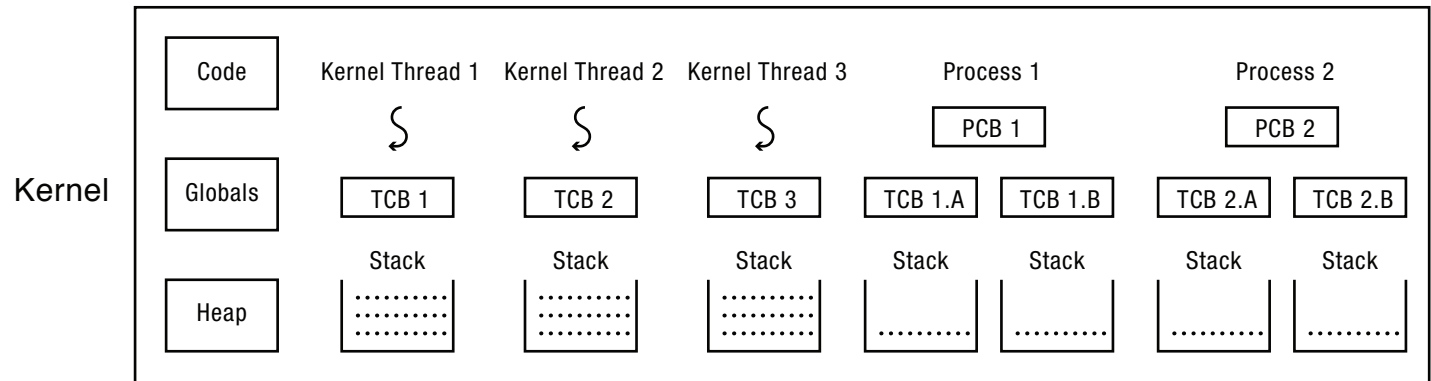
Prof. Seyed Majid Zahedi

<https://ece.uwaterloo.ca/~smzahedi>

Outline

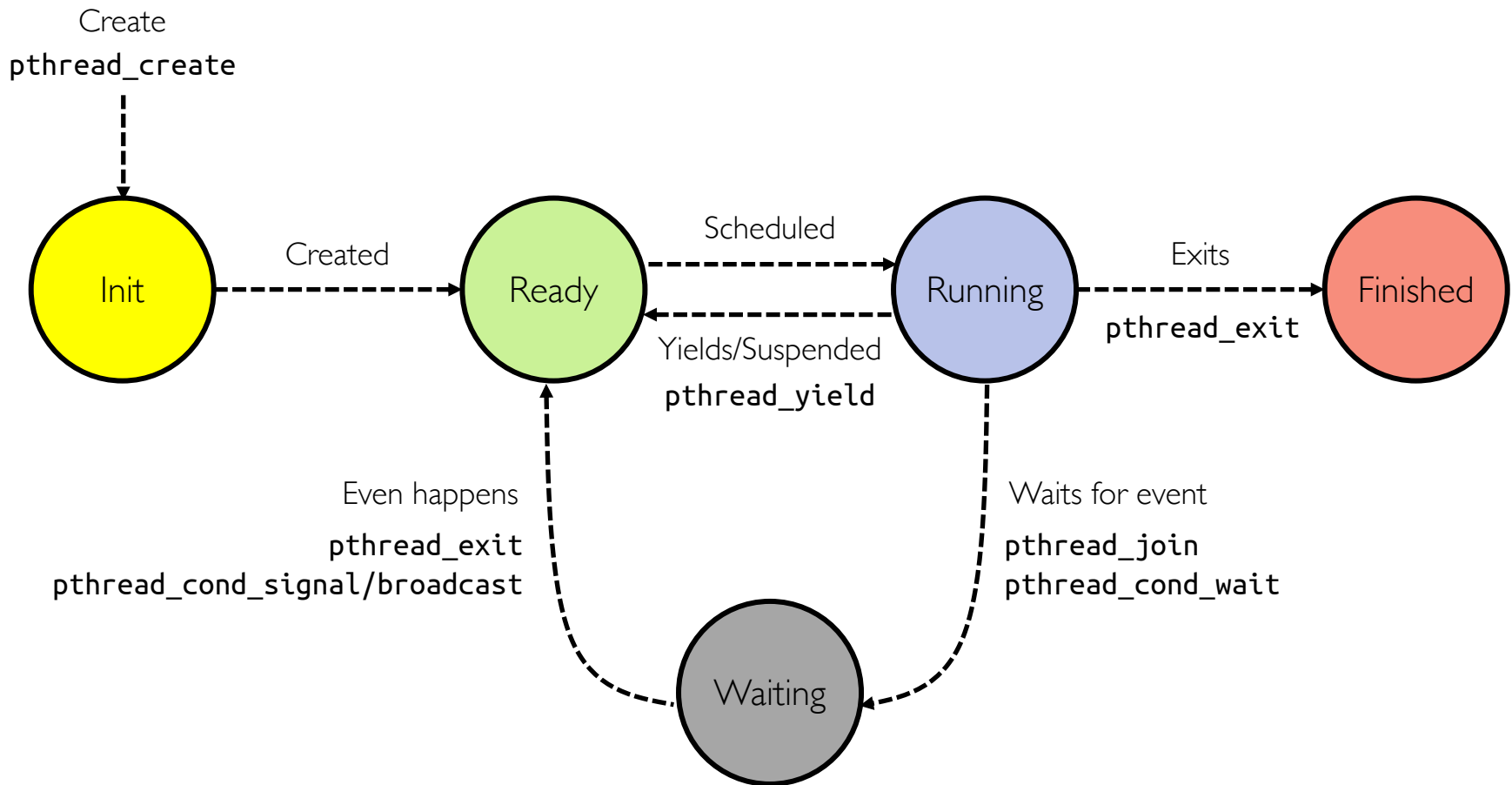
- Thread implementation
 - Create, yield, switch, etc.
- Kernel- vs. user-managed threads
- Implementation of synchronization objects
 - Mutex, semaphore, condition variable

Kernel-managed Multithreading



- System library allocates **user-space stack** for each user-level thread
- System library then uses system calls to create, join, yield, exit threads
- Kernel handles scheduling and context switching using **kernel-space stacks**

Recall: Thread Lifecycle



A process can go directly from ready or waiting to finished (example: main thread calls exit)

What Triggers a Context Switch?

- **Synchronous event:** thread invokes a system call or an exceptions happens
 - E.g., **yield**, **join**, **write**, **read**, or segmentation fault, divide by zero, etc.
 - This is called a voluntary context switch

```
void compute_PI() {  
    while(TRUE) {  
        compute_next_digit();  
        thread_yield();  
    }  
}
```

- **Asynchronous event:** interrupts happens
 - E.g., timer interrupt, new packet arrives, a DMA request finishes
 - This is called an involuntary context switch

System Call, Interrupt, and Exception Handlers

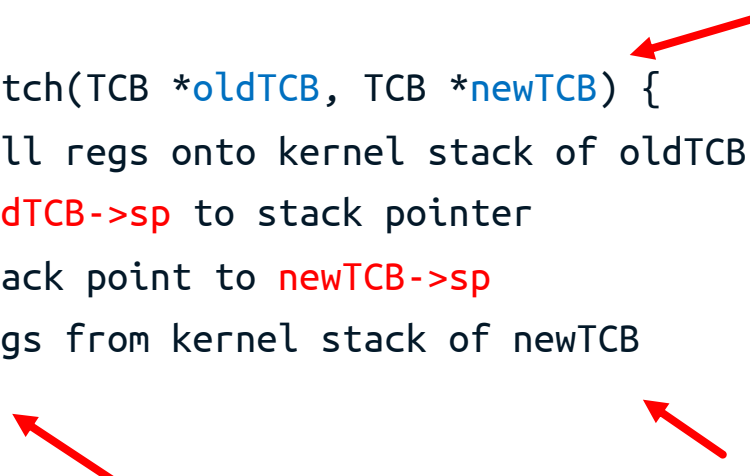
```
handler() {  
    // this runs in kernel mode  
    // SP points to a kernel stack  
    Push regs that might be used by handler on kernel stack  
  
    // (handle the event)  
  
    Handler_Exit  
        Pop regs that were pushed  
        Return  
}
```



Switch Between Threads

```
// We enter as oldTCB, but we return as newTCB
// Returns with newTCB's registers and stack
```

```
thread_switch(TCB *oldTCB, TCB *newTCB) {
    Push all regs onto kernel stack of oldTCB
    Set oldTCB->sp to stack pointer
    Set stack point to newTCB->sp
    Pop regs from kernel stack of newTCB
    Return
}
```



newTCB could be a thread that was context switched before and we are context switching back to it, or it could be a **newly created** thread

Where does this return to?

If **newTCB** is not newly created, then we return to **kernel code** that called **thread_switch**
(return address is stored on **newTCB**'s stack)

If **newTCB** is newly created, then it should have an **entry point** address on its stack

What is popped here?

If **newTCB** is not newly created, then we pop what we pushed **last time** we context switched it

If **newTCB** is newly created, then it should have **dummy data frame** on top of its stack

Threads Entry Point

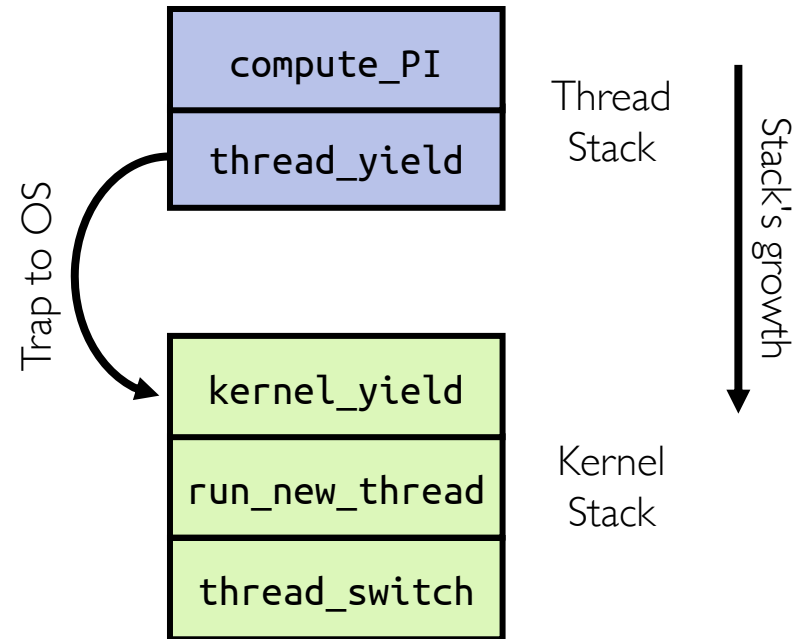
- For kernel threads, no mode switch is required
 - Could directly jump to function that thread will run
- For user threads, switch from kernel to user mode is required
 - Need one level of indirection
 - Could jump to a kernel code that then jumps to user code and changes mode atomically
 - E.g., could jump to **Handler_Exit**

Creating New User Threads

```
thread_create(void *(*func)(void*), void *args) {  
    // Allocate TCB  
    TCB *tcb = new TCB()  
    // Allocate kernel stack (note that stack grows downwards)  
    tcb->sp = new Stack(stack_size) + stack_size;  
    // Set up kernel stack  
    // (1) Push func and args  
    * (--tcb->sp) = args;  
    * (--tcb->sp) = func;  
    // (2) push data for Handle_Exit  
    push_dummy_handler_frame(&tcb->sp);  
    * (--tcb->sp) = Handler_Exit  
    // (3) Push dummy data for thread_switch  
    push_dummy_switch_frame(&tcb->sp);  
    // Set state of thread to read  
    tcb->state = READY;  
    // Put tcb on ready list  
    readyList.add(tcb);  
}
```

Stack for Yielding Thread

```
void run_new_thread() {  
    // Prevent interrupt from stopping us  
    // in the middle of switch  
    disable_interrupts();  
    // Choose another TCB from ready list  
    chosenTCB = scheduler.getNextTCB();  
    if (chosenTCB != runningTCB) {  
        // Move running thread onto ready list  
        runningTCB->state = READY;  
        ready_list.add(runningTCB);  
        // Switch to the new thread  
        thread_switch(runningTCB, chosenTCB);  
  
        // We're running again!  
        runningTCB->state = RUNNING;  
        // Do any cleanup  
        do_cleanup_housekeeping();  
    }  
    // Enable interrupts again  
    enable_interrupts();  
}
```

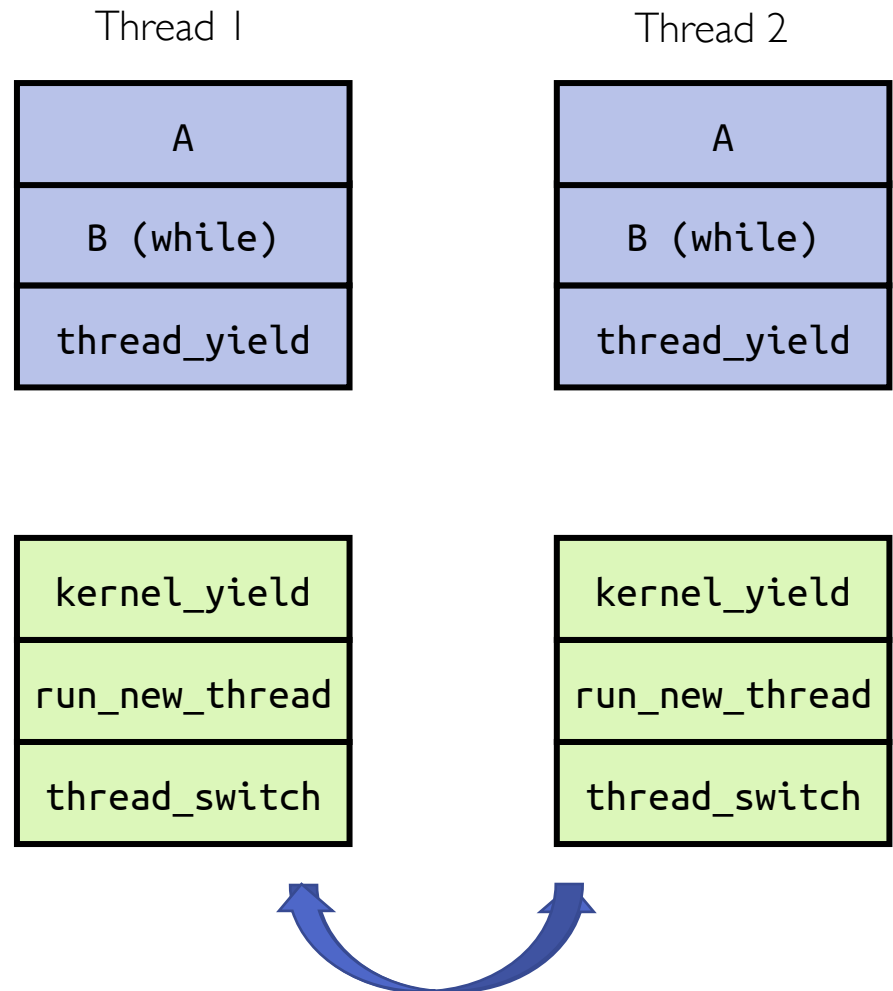


Start from here whenever another thread switches back to this thread

How Do Stacks Look Like?

- Two threads run following code

```
A() {  
    B();  
}  
  
B() {  
    while(TRUE) {  
        thread_yield();  
    }  
}
```

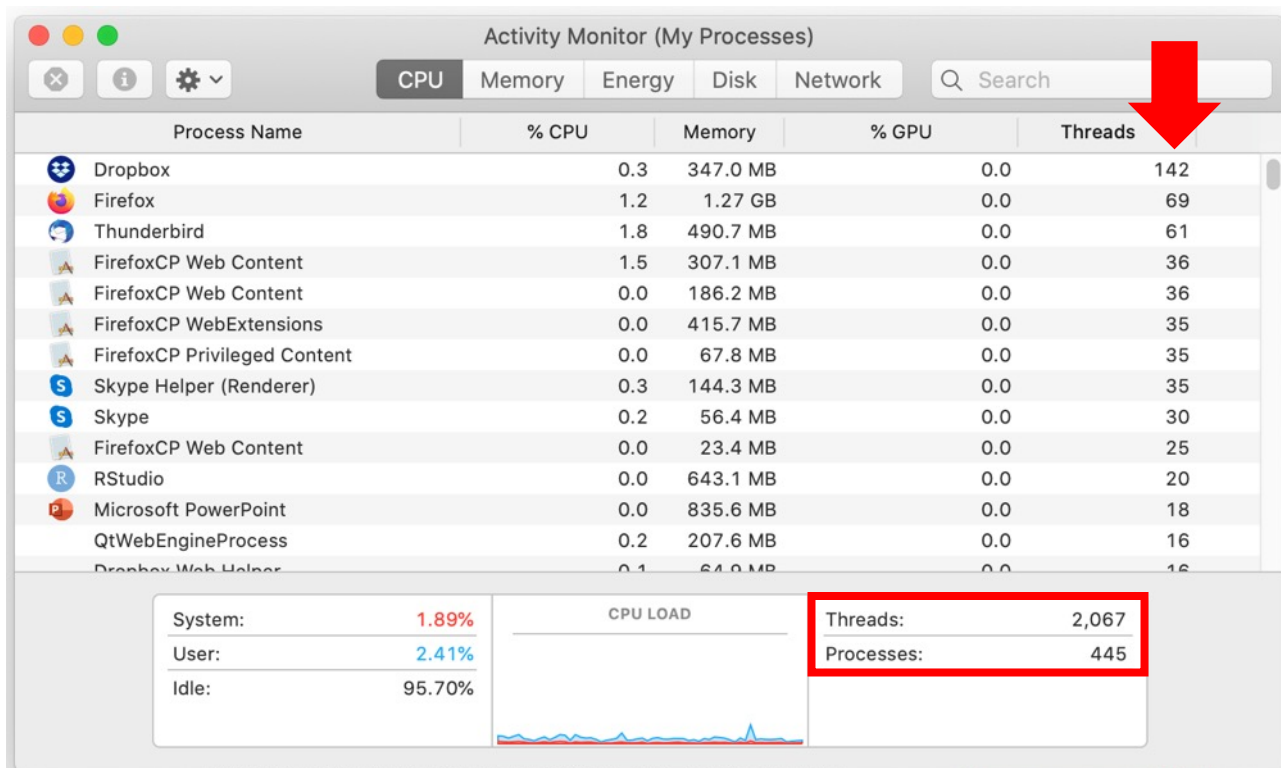


Outline

- Thread implementation
 - Create, yield, switch, etc.
- Kernel- vs. user-managed threads
- Implementation of synchronization objects
 - Mutex, semaphore, condition variable

Recall: Some Numbers

- Many process are **multi-threaded**, so thread context switches may be either **within-process** or **across-processes**

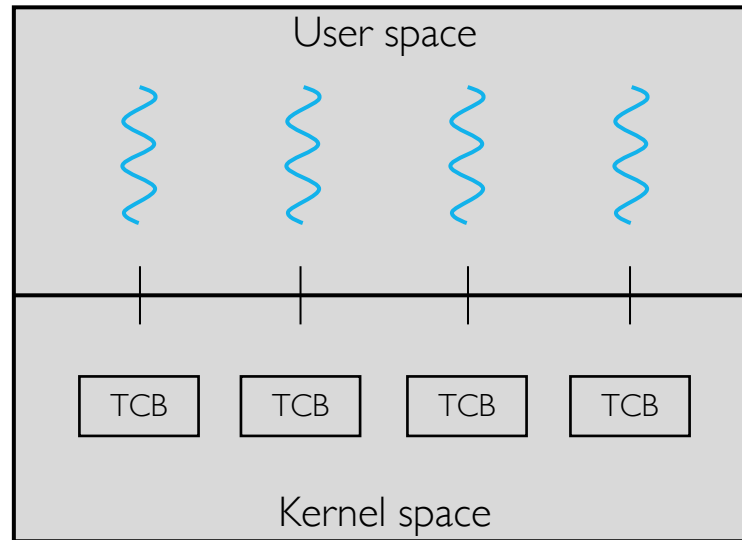


Some Numbers (cont.)

- Frequency of performing context switches is $\sim 10\text{-}100\text{ms}$
- Context switch time in Linux is $\sim 3\text{-}4\text{ us}$ (Intel i7 & Xeon E5)
 - Thread switching faster than process switching ($\sim 100\text{ ns}$)
- Switching across cores is $\sim 2\times$ more expensive than within-core
- Context switch time increases sharply with size of working set*
 - Can increase $\sim 100\times$ or more
- Moral: overhead of context switching depends mostly on cache limits and process or thread's hunger for memory

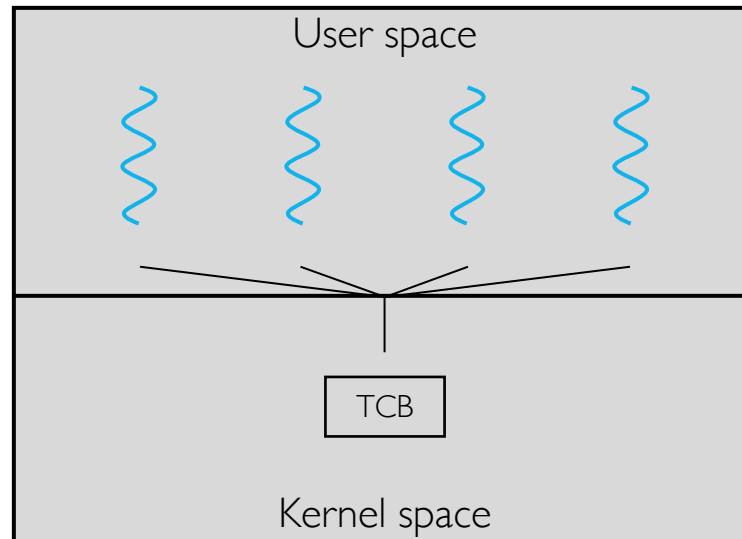
* Working set is subset of memory used by process in time window

Kernel- vs. User-managed Threads



- We have been talking about kernel-managed threads
- Each user thread maps to one TCB (1:1 mapping)
- Every thread can run or block independently
- This approach is relatively expensive
 - Need to make crossing into kernel mode to schedule

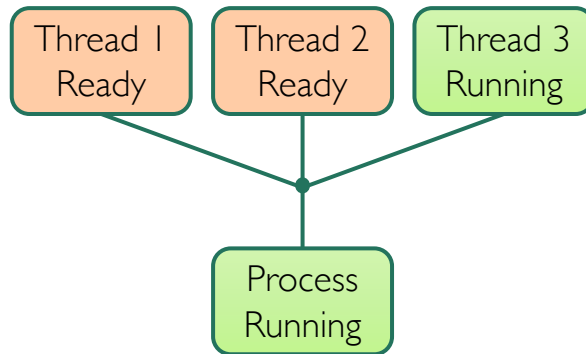
User-managed Threads



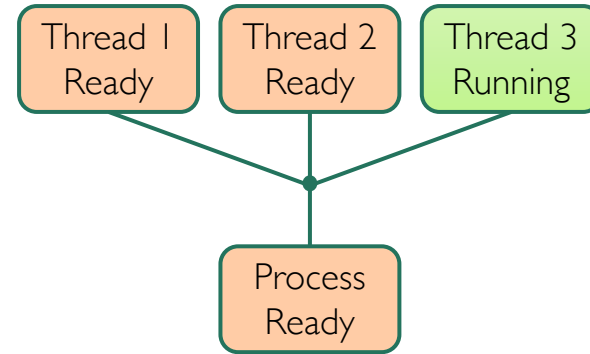
- Alternative is for user-level library to do all thread management tasks
- User process creates threads, maintains their state, and schedules them
- Kernel is not aware of existence of multiple threads
- Kernel only allocates single TCB to user process (N:1 mapping)
- Examples: [Solaris Green Threads](#), [GNU Portable Threads](#)

User-managed Threads: Thread vs. Process State

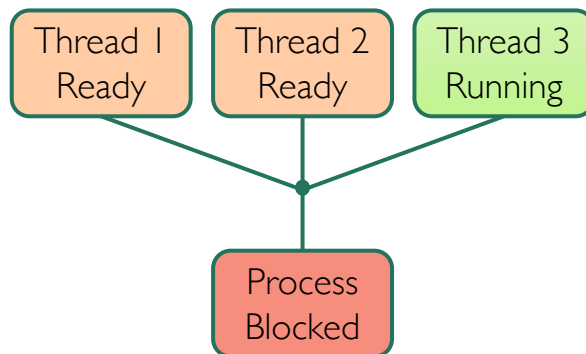
Thread 3 is running on CPU



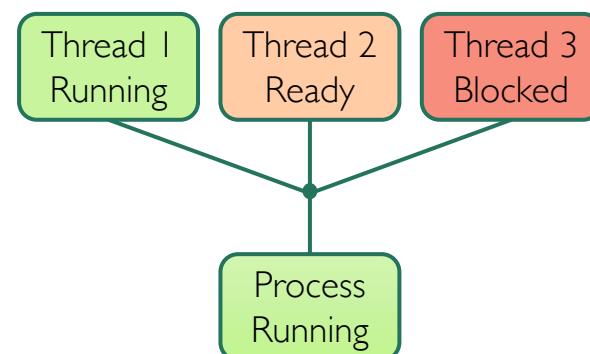
Kernel has suspended user process



Thread 3 requests I/O



Thread 3 is blocked on user-level mutex



Downside of User-managed Threads

- Multiple threads may not run in parallel on multicore
- When one thread blocks on I/O, all threads block
- Alternative: *scheduler activations*
 - Notify user-level scheduler of relevant kernel events

Classification of OSes

- Most operating systems have either
 - One or many address spaces
 - One or many threads per address space

# threads Per AS: # of addr spaces:	One	Many
One	MS/DOS, early Macintosh	Traditional UNIX
Many	Embedded systems (Geoworks, VxWorks, JavaOS, Pilot(PC), etc.)	Mach, OS/2, Linux, Windows 10, Win NT to XP, Solaris, HP-UX, OS X

Outline

- Thread implementation
 - Create, yield, switch, etc.
- Kernel- vs. user-managed threads
- Implementation of synchronization objects
 - Mutex, semaphore, condition variable

Implementing Synchronization Objects

Programs	Bounded Buffers
Synch Objects	Mutex Semaphore Monitor
Atomic Inst	Load/Store Disable Interrupts Test&Set

Mutex Implementation - Take I: Disabling Interrupts

- Recall: context switching is triggered in two ways
 - Voluntary: thread does something to relinquish CPU
 - Involuntary: interrupts cause dispatcher to take CPU
- On uniprocessors, we can avoid context switching by
 - Avoiding voluntary context switches
 - Preventing involuntary context switches by disabling interrupts
- Naïve implementation of mutex in uniprocessors

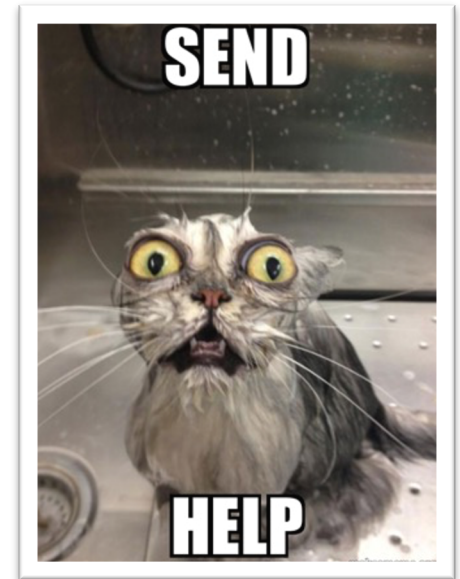
```
class Mutex {  
    public:  
        void lock() { disable_interrupts(); };  
        void unlock() { enable_interrupts(); };  
}
```

Problems with Naïve Implementation of Mutex

- OS cannot let users use this!

```
Mutex::lock();  
while(TRUE);
```

- It does not work well in multiprocessors
 - Other CPUs could be interrupted



- Real-time OSes should provide guarantees on timing!
 - Critical sections might be arbitrarily long
 - What happens with I/O or other important events?
 - “Reactor about to meltdown. Help?”

Implementation of Mutex - Take 2: Disabling Interrupts + Lock Variable

Key idea: maintain lock variable and impose mutual exclusion only during operations on that variable

```
class Mutex {  
    private:  
        int value = FREE;  
        Queue waiting;  
    public:  
        void lock();  
        void unlock();  
}
```

Take 2 (cont.)

```
Mutex::lock() {  
    disable_interrupts();  
    if (value == BUSY) {  
        // Add TCB to waiting queue  
        waiting.add(runningTCB);  
        runningTCB->state = WAITING;  
        // Pick new thread to run  
        chosenTCB = ready_list.get_nextTCB();  
        // Switch to new thread  
        thread_switch(runningTCB, chosenTCB);  
        // We're back! We have locked mutex!  
        runningTCB->state = RUNNING;  
    } else {  
        value = BUSY;  
    }  
    enable_interrupts();  
}
```

```
Mutex::unlock() {  
    disable_interrupts();  
    if (!waiting.empty()) {  
        // Make another TCB eady  
        next = waiting.remove();  
        next->state = READY;  
        ready_list.add(next);  
    } else {  
        value = FREE;  
    }  
    enable_interrupts();  
}
```

- Enable/disable interrupts also act as a memory barrier operation forcing all memory writes to complete first

Take 2: Discussion

- Why do we need to disable interrupts at all?
 - Avoid interruption between checking and setting lock value
 - Otherwise, two threads could think that they both have locked the mutex

```
Mutex::lock() {  
    disable_interrupts();  
    if (value == BUSY) {  
        ...  
    } else {  
        value = BUSY;  
    }  
    enable_interrupts();  
}
```

} Critical section of mutex
(different from critical section of program)

- Unlike previous solution, critical section (inside **lock()**) is very short
 - User of mutex can take as long as they like in **their own critical section** (doesn't impact global machine behavior)
 - Critical interrupts taken in time!

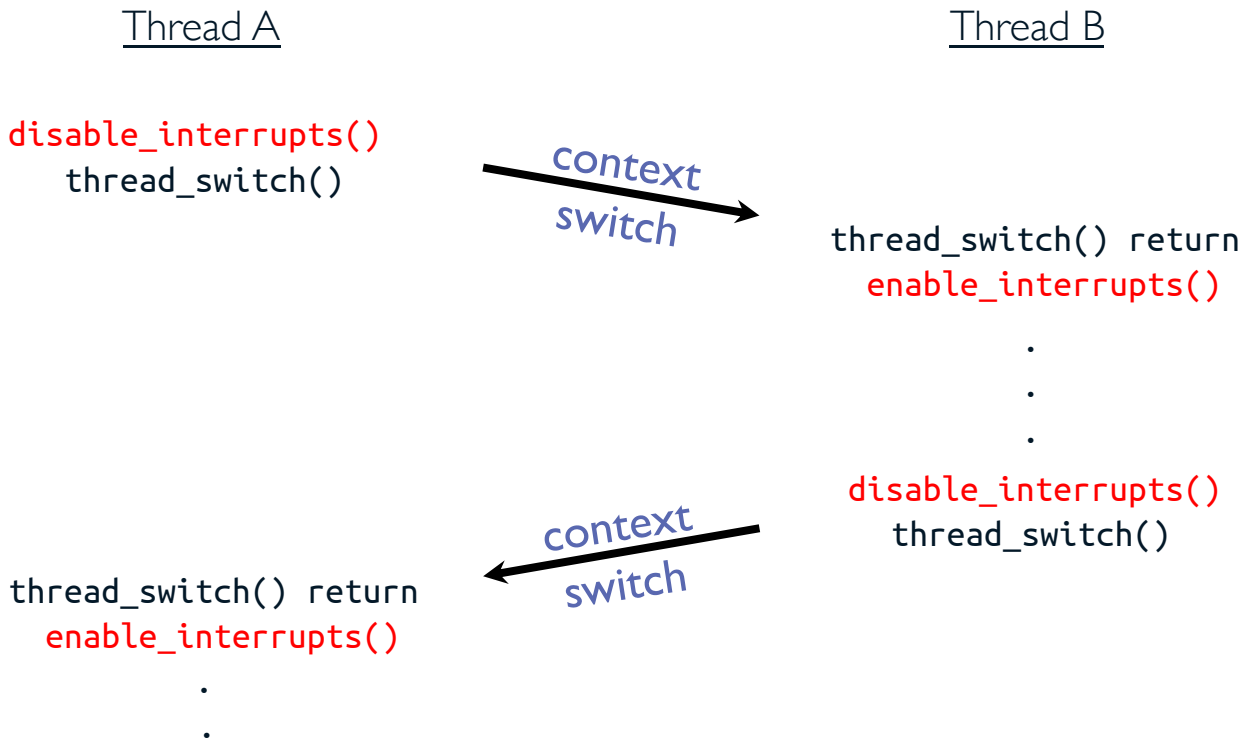
Re-enabling Interrupts

```
enable interrupts here? → Mutex::lock() {  
    disable_interrupts();  
    if (value == BUSY) {  
        waiting.add(runningTCB);  
        runningTCB->state = WAITING;  
        chosenTCB = ready_list.get_nextTCB();  
        thread_switch(runningTCB, chosenTCB);  
        runningTCB->state = RUNNING;  
    } else {  
        value = BUSY;  
    }  
    enable_interrupts();  
}
```

- Before putting thread on wait queue?
 - `unlock()` can check waiting queue and not wake up thread
- After putting thread on wait queue?
 - `unlock()` puts thread on ready queue, but thread still thinks it needs to go to sleep!
 - Thread goes to sleep while keeping mutex locked (deadlock!)
- After `thread_switch()`? But ... how?

How to Re-enable After `thread_switch()`?

- It is responsibility of next thread to re-enable interrupts
 - This invariant should be carefully maintained
- When sleeping thread wakes up, returns to `lock()` and re-enables interrupts



Problems with Take 2

- User libraries cannot use this implementation (why?)
- Doesn't work well on multiprocessor
 - Disabling interrupts on all processors requires messages and would be very time consuming
- Alternative solution: **atomic read-modify-write instructions**
 - Read value from an address and then write new value to it *atomically*
 - Make HW responsible for implementing this correctly
 - Uniprocessors (not too hard)
 - Multiprocessors (requires help from cache coherence protocol)
 - Unlike disabling interrupts, this can be used in both uniprocessors and multiprocessors

Examples of Read-Modify-Write Instructions

- ```
test&set (&address) {
 result = M[address];
 M[address] = 1;
 return result;
}
```

```
/* most architectures */
/* return result from
 "address" and set value at
 "address" to 1 */
```
- ```
swap (&address, register) {  
    temp = M[address];  
    M[address] = register;  
    register = temp;  
}
```

```
/* x86 */  
/* swap register's value to  
   value at "address" */
```
- ```
compare&swap (&address, reg1, reg2) {
 if (reg1 == M[address]) {
 M[address] = reg2;
 return success;
 } else {
 return failure;
 }
}
```

```
/* 68000 */
```

# Spinlock with test&set()

---

- Simple implementation

```
class Spinlock {
 private:
 int value = 0
 public:
 void lock() { while(test&set(value)); };
 void unlock() { value = 0; };
}
```

- Unlocked mutex: **test&set** reads **0** and sets **value = 1**
- Locked mutex: **test&set** reads **1** and sets **value = 1** (no change)
- What is wrong with this implementation?
  - Waiting threads consume cycles while **busy-waiting**



# Spinlock with `test&set()`: Discussion

---

- Upside?
  - Machine can receive interrupts
  - User code can use this mutex
  - Works on multiprocessors
- Downside?
  - This is very wasteful as threads consume CPU cycles (busy-waiting)
  - Waiting threads may delay the thread that has locked mutex (no one wins!)
  - **Priority inversion**: if busy-waiting thread has higher priority than the thread that has locked mutex then there will be no progress! (more on this later)
- In semaphores and monitors, threads may wait for arbitrary long time!
  - Even if busy-waiting was OK for mutexes, it's not OK for other primitives
  - Exam/quiz solutions should avoid busy-waiting!



# Implementation of Mutex - Take 3: Using Spinlock

---

- Can we implement mutex with **test&set** without busy-waiting?
  - We cannot eliminate busy-waiting, but we can minimize it!
  - **Idea:** only busy-wait to atomically check mutex value

```
class Mutex {
 private:
 int value = FREE;
 Spinlock mutex_spinlock;
 Queue waiting;
 public:
 void lock();
 void unlock();
}
```

```
class Scheduler {
 private:
 Queue readyList;
 Spinlock scheduler_spinlock;
 public:
 void suspend(Spinlock *spinlock);
 void make_ready(TCB *tcb);
}
```

# Implementation of Mutex - Take 3 (cont.)

---

```
Mutex::lock() {
 mutex_spinlock.lock();
 if (value == BUSY) {
 // Add TCB to waiting queue
 waiting.add(runningTCB);
 scheduler->suspend(&mutex_spinlock)
 // Scheduler unlocks mutex_spinlock
 } else {
 value = BUSY;
 mutex_spinlock.unlock();
 }
}
```

```
Mutex::unlock() {
 mutex_spinlock.lock();
 if (!waiting.empty()) {
 // Make another TCB ready
 next = waiting.remove();
 scheduler->make_ready(next);
 } else {
 value = FREE;
 }
 mutex_spinlock.unlock();
}
```

Can interrupt handler use this lock?

- No! Interrupt handler is not a thread, it cannot be suspended

How should we protect data shared by interrupt handler and kernel thread?

- Use spinlocks!
- To avoid deadlock, kernel thread should disable interrupts before locking the spinlock.
- Otherwise, interrupt handler could spin forever if spinlock is locked by a kernel thread!

# Implementation of Mutex - Take 3 (cont.)

---

```
Scheduler::suspend(Spinlock *spinlock) {
 disable_interrupts();
 scheduler_spinlock.lock();
 spinlock->unlock();
 runningTCB->state = WAITING;
 chosenTCB = ready_list.get_nextTCB();
 thread_switch(runningTCB, chosenTCB);
 runningTCB->state = RUNNING;
 scheduler_spinlock.unlock();
 enable_interrupts();
}
```

```
Scheduler::make_ready(TCB *tcb) {
 disable_interrupts();
 scheduler_spinlock.lock();
 ready_list.add(tcb);
 thread->state = READY;
 scheduler_spinlock.unlock();
 enable_interrupts();
}
```

Why disable interrupts?

- To avoid **deadlock**!
- Interrupt handler could spin forever if it needs scheduler's spinlock!

What might happen if we unlock **mutex\_spinlock** before **suspend()**?

- Then **make\_ready()** could run before **suspend()**, which is very bad!



# Mutex Using Interrupts vs. Spinlock

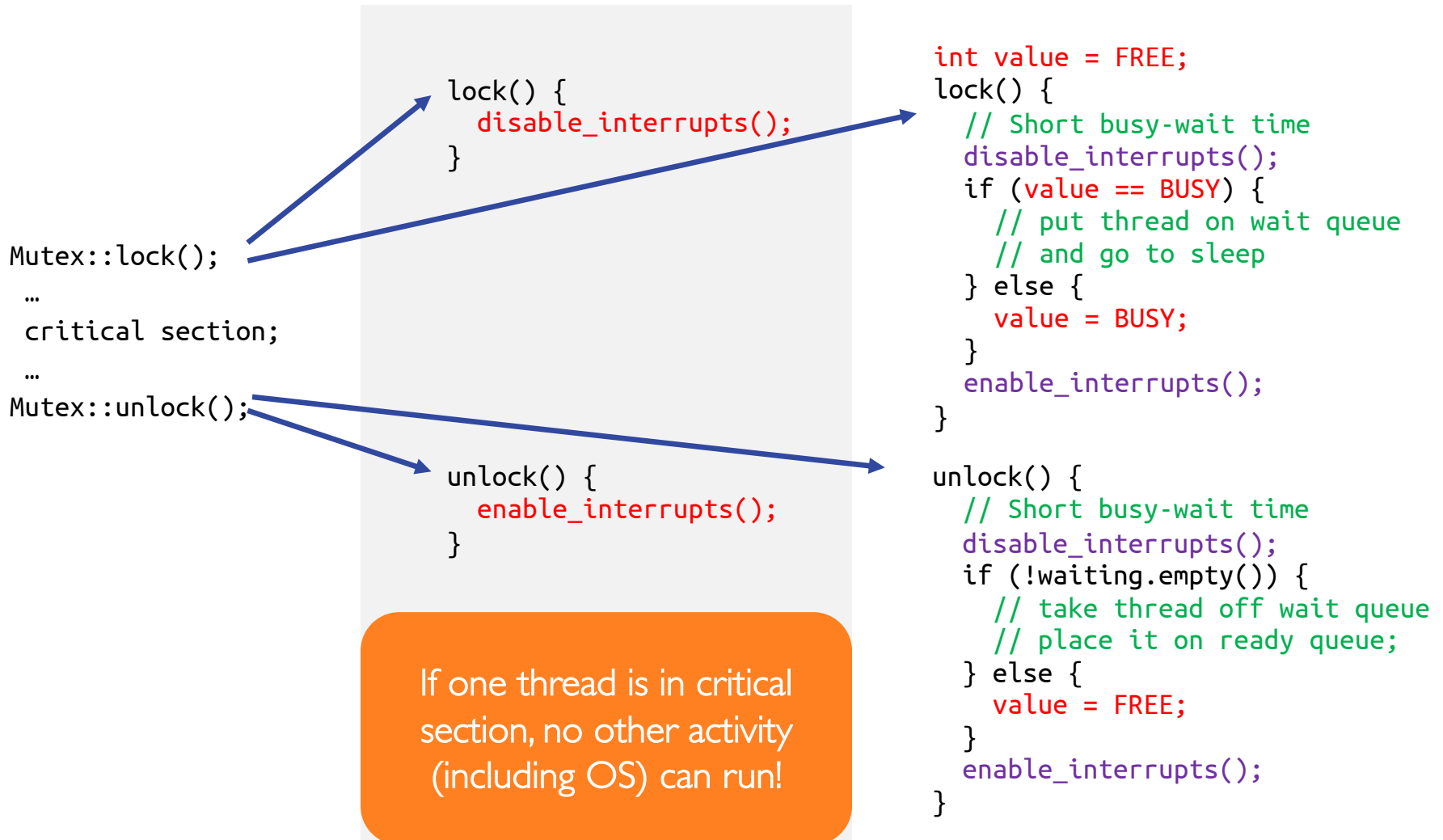
---

```
lock() {
 disable_interrupts();
 if (value == BUSY) {
 // put thread on wait queue and
 // go to sleep
 } else {
 value = BUSY;
 }
 enable_interrupts();
}
```

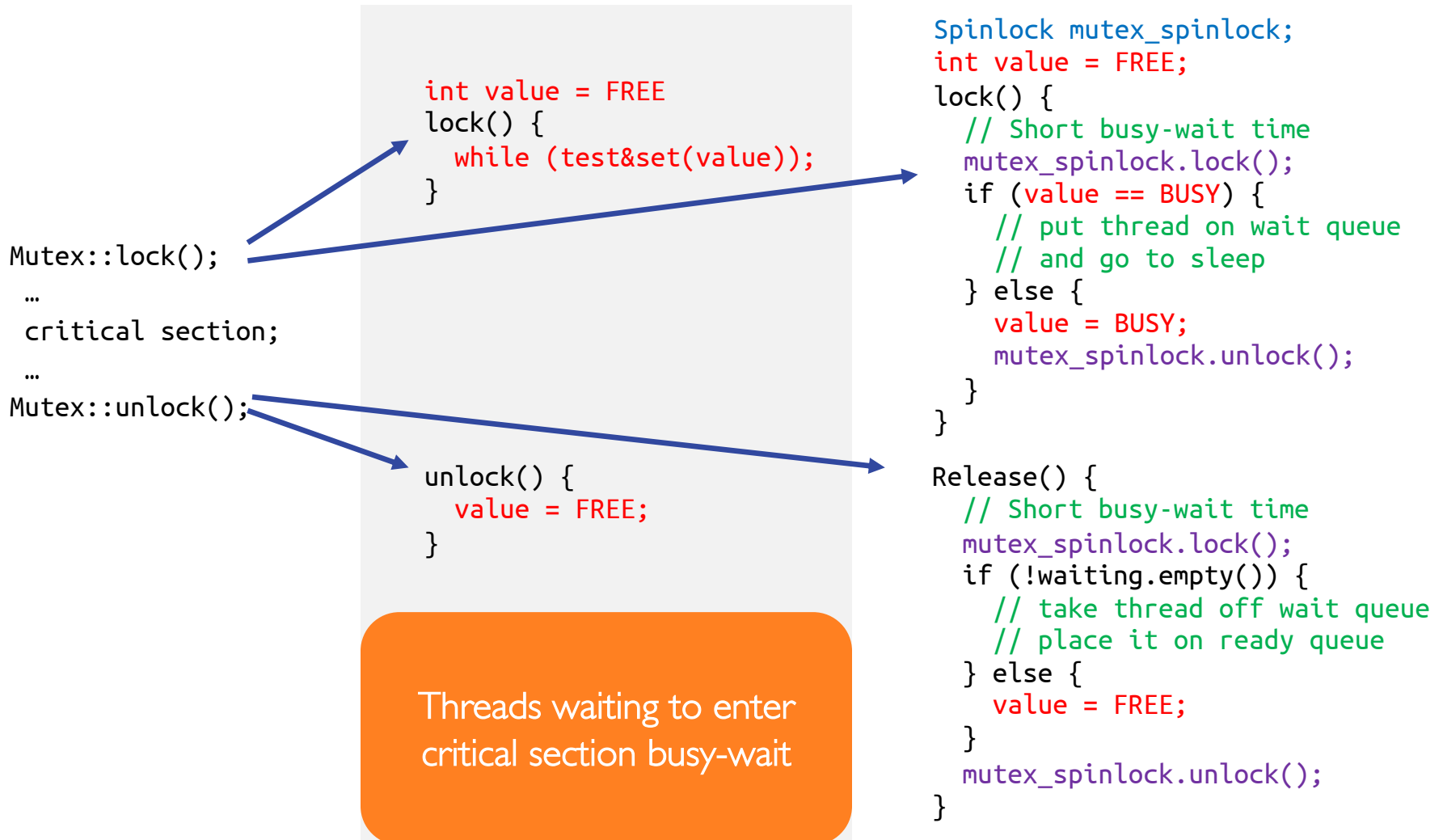
```
lock() {
 mutex_spinlock.lock();
 if (value == BUSY) {
 // put thread on wait queue and
 // go to sleep
 } else {
 value = BUSY;
 mutex_spinlock.unlock();
 }
}
```

- Replace
  - disable interrupts;  $\Rightarrow$  spinlock.lock;
  - enable interrupts  $\Rightarrow$  spinlock.unlock;

# Recap: Mutexes Using Interrupts



# Recap: Mutexes Using Spinlock (test&set)



# Mutex Implementation in Linux

---

- Most mutexes are free most of the time
  - Linux implementation takes advantage of this fact
- Hardware supports powerful atomic operations
  - E.g., atomic increment, decrement, exchange, etc.
  - Linux implementation takes advantage of these too
- Fast path
  - If mutex is unlocked, and no one is waiting, two instructions to lock
  - If no one is waiting, two instructions to unlock
- Slow path
  - If mutex is locked or someone is waiting, use take 3 implementation



# Mutex Implementation in Linux (cont.)

---

```
struct Mutex {
 // 1: unlocked; < 1: locked
 atomic_t count;
 Spinlock mutex_spinlock;
 Queue waiting;
}

// code for lock()
lock decl (%eax)
// jump if not signed
// i.e., if value is now 0
jns 1f
call slow_path_lock
1:
//critical section
```

- For `Mutex::lock()`, Linux uses *macro*
  - To void making procedure call on fast path
- x86 *lock* prefix before *decl* instruction signifies to processor that instruction should be executed atomically

# Mutex Implementations: Discussion

---

- Our lock implementations are procedure calls
- Work well for kernel threads
- Does not work properly for user threads
  - Because system call may often disable interrupts/save state to TCB
  - But same basic idea works – e.g., in Linux, user-level mutex has two paths - Fast path: lock using test&set and slow path: system call to kernel, use kernel mutex
- How do lock-initiated and timer-interrupt-initiated switches interleave?
  - Turns out, they just work as long as we maintain the invariant on interrupts -disable before calling thread\_switch() and enable when thread\_switch() returns



# Implementing Synchronization Objects

---

|               |                                        |
|---------------|----------------------------------------|
| Programs      | Bounded Buffers                        |
| Synch Objects | Mutex Semaphore Monitor                |
| Atomic Inst   | Load/Store Disable Interrupts Test&Set |

# Implementation of Semaphore

---

```
Semaphore::P() {
 semaphore_spinlock.lock();
 if (value == 0) {
 waiting.add(myTCB);
 scheduler->suspend(&semaphore_spinlock);
 } else {
 value--;
 }
 semaphore_spinlock.unlock();
}
```

```
Semaphore::V() {
 semaphore_spinlock.lock();
 if (!waiting.empty()) {
 next = waiting.remove();
 scheduler->make_ready(next);
 } else {
 value++;
 }
 semaphore_spinlock.unlock();
}
```

Can interrupt handler use this semaphore?

- It cannot use **P** (why?), but it might want to use **V** (more on this later)
- In that case, interrupts should be disabled at the beginning of **P** and **V** and enabled at the end

# Implementing Synchronization Objects

---

|               |                                        |
|---------------|----------------------------------------|
| Programs      | Bounded Buffers                        |
| Synch Objects | Mutex Semaphore Monitor                |
| Atomic Inst   | Load/Store Disable Interrupts Test&Set |

# Implementation of Condition Variables

---

```
class CV {
 private:
 Queue waiting;
 public:
 void wait(Mutex *mutex);
 void signal();
 void broadcast();
}

CV::wait(Mutex *mutex) {
 waiting.add(myTCB);
 scheduler.suspend(&mutex);
 mutex->lock();
}

CV::signal() {
 if (!waiting.empty()) {
 thread = waiting.remove();
 scheduler.make_ready(thread);
 }
}

void CV::broadcast() {
 while (!waiting.empty()) {
 thread = waiting.remove();
 scheduler.make_ready(thread);
 }
}
```

Why doesn't `class CV` need `cv_spinlock`?

- Since `mutex` is locked whenever `wait`, `signal`, or `broadcast` is called, we already have mutually exclusive access to condition wait queue

# Implementation of Condition Variable using Semaphores (Take I)

---

```
wait(*mutex) {
 mutex->unlock();
 semaphore.P();
 mutex->lock();
}

signal() {
 semaphore.V();
}
```

- Does this work?
  - No! `signal()` should not have memory!

# Implementation of Condition Variable using Semaphores (Take 2)

---

```
wait(*mutex) {
 mutex->unlock();
 semaphore.P();
 mutex->lock();
}

signal() {
 if (semaphore's queue is not empty)
 semaphore.V();
}
```

- Does this work?
  - No! For one, not legal to look at contents of semaphore's queue.
  - But also, unlocking mutex and going to sleep should happen atomically – signaler can slip in after mutex is unlocked, and before waiter is put on wait queue, which means waiter never wakes up!



# Implementation Condition Variable using Semaphores (Take 3)

---

Key idea: have separate semaphore for each waiting thread  
and put semaphores in ordered queue

```
wait(*mutex) {
 semaphore = new Semaphore; // a semaphore per waiting thread
 queue.add(semaphore); // queue for waiting threads
 mutex->unlock();
 semaphore.P();
 mutex->lock();
}

signal() {
 if (!queue.empty()) {
 semaphore = queue.remove()
 semaphore.V();
 }
}
```

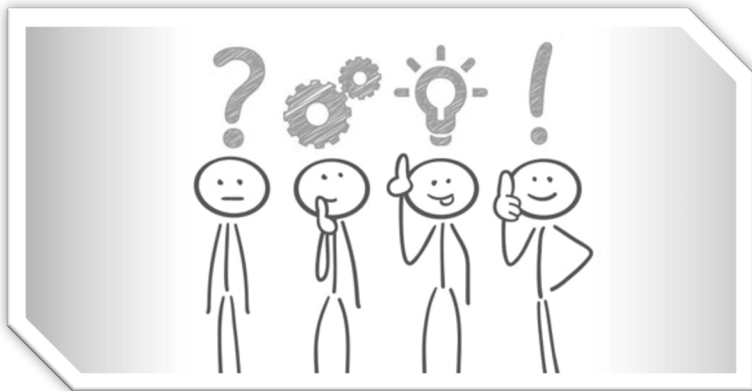
# Summary

---

- Use HW atomic primitives as needed to implement synchronization
  - Disabling of Interrupts, test&set, swap, compare&swap
- Define lock variable to implement mutex,
  - Use HW atomic primitives to protect modifications of that variable
- Maintain the invariant on interrupts
  - Disable interrupts before calling `thread_switch()` and enable them when `thread_switch()` returns
- Be very careful not to waste machine resources
  - Shouldn't disable interrupts for long
  - Shouldn't busy-wait for long

# Questions?

---



# Acknowledgment

---

- Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny