ECE 350 Real-time Operating Systems

Lecture 3: Multithreaded Kernels

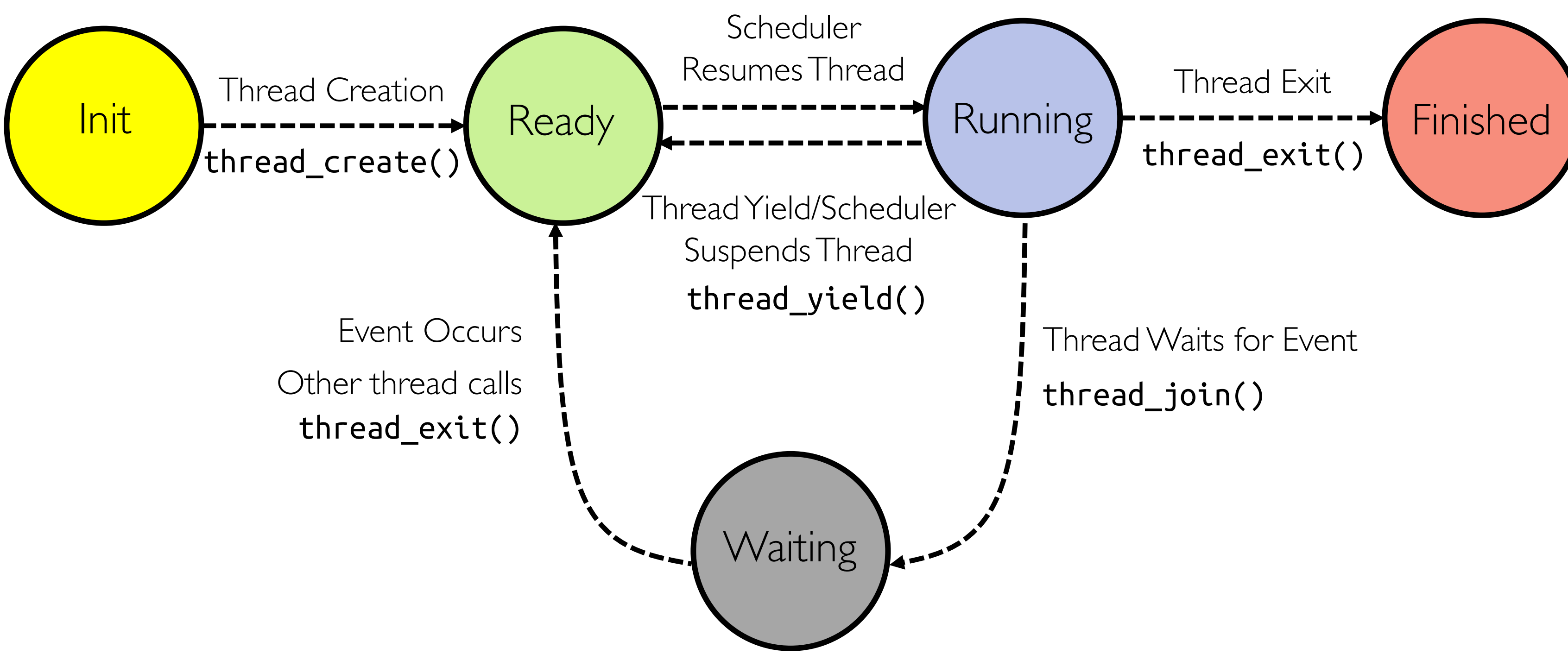
Prof. Seyed Majid Zahedi

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

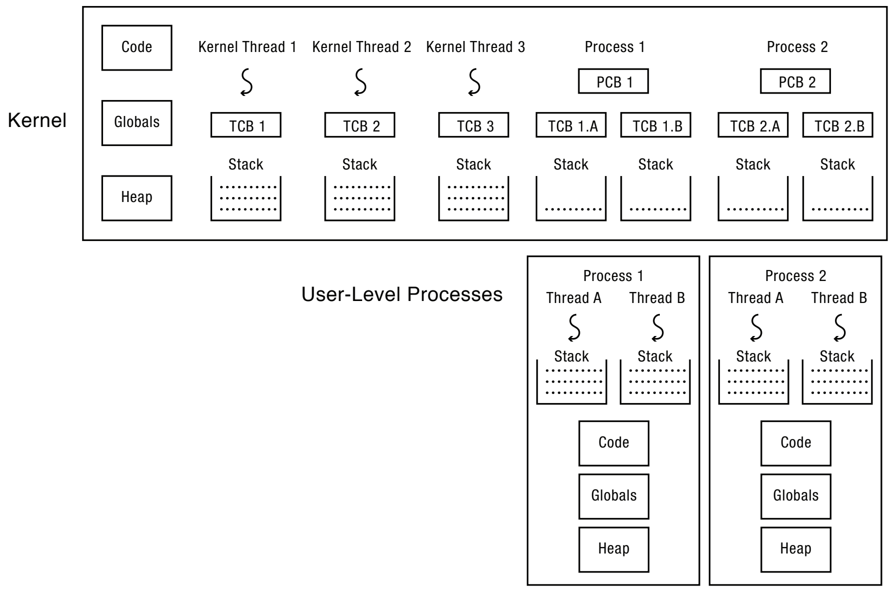
Outline

- Implementation of kernel threads
  - Create, yield, switch, etc.
- User-level threads with and without kernel support
- Implementation of synchronization objects
  - Mutex, semaphore, condition variable

Recall: Thread Lifecycle



Kernel-managed Multithreading



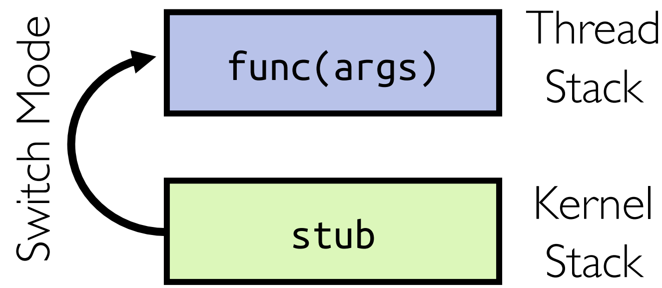
- User-level library allocates user-level stack for each user-level thread
- User-level library then uses syscalls to create, join, yield, exit threads
- Kernel handles scheduling and context switching
- Simple, but a lot of transitions between user and kernel mode

Creating New Threads

```
void thread_create(thread_t *thread, void *(*func)(void*), void *args) {
     // Allocate TCB and stack (starts at top of allocated region and grows down)
     TCB *tcb = new TCB();
     thread->tcb = tcb;
     tcb->stack_size = INITIAL_STACK_SIZE;
     tcb->stack = new Stack(INITIAL_STACK_SIZE);
     tcb->sp = tcb->stack + INITIAL_STACK_SIZE;
     // Set pc so that thread starts running at kernel routine stub(func, arg)
     tcb->pc = stub;
     // When called, stub expects to find its arguments (i.e., func, arg) on the stack
     *(tcb->sp) = args;
     tcb->sp--;
     *(tcb->sp) = func;
     tcb->sp--;
     // Push dummy frame onto stack so that thread_switch works correctly (more on this later)
     push_dummy_switch_frame(tcb);
     tcb->state = READY;
     // Put tcb on ready list
     readyList.add(tcb);
 }
```

How Does stub Look Like?

```
void stub(void *(*func)(void*), void *args) {
    do_startup_housekeeping();
    // run function
    (*func)(args);
    // If func doesn't call exit, call it here
    thread_exit();
}
```



- Startup housekeeping includes
  - Things like recording start time of thread and other statistics
  - Switching to user mode, enabling interrupts, changing status to **`RUNNING`** etc.
- Stack will grow and shrink with execution of thread
- Final return from thread returns into **`stub()`** which calls **`thread_exit()`** which wake up another sleeping thread if there are any

Context Switching Between Threads

- What triggers context switch?
  - Voluntary: thread returns control voluntarily
    - E.g., executing `thread_yield()`, `thread_join()`, `thread_exit()`
  - Involuntary: thread gets preempted
    - E.g., interrupts, exceptions
- How does voluntary context switch differ from involuntary one?
  - Voluntary switches usually involve thread library function
  - Involuntary switches usually involve interrupt handler which decides what to do next

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

Switching Threads

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

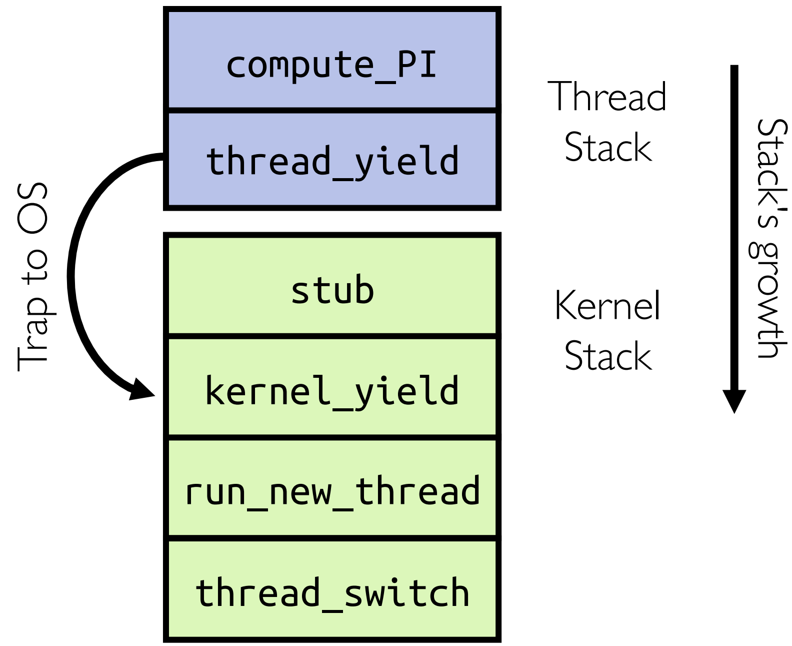
void thread_switch(TCB *oldTCB, TCB *newTCB) {
    // Push regs onto kernel stack for oldTCB
    pushad;
    // Save oldTCB's stack pointer
    oldTCB->sp = sp;
    // Switch to newTCB's stack
    sp = newTCB->sp;
    // Pop regs from kernel stack for newTCB
    popad;
    return();
}
```

Where does this return to?

- Returns to return address stored on `newTCB`'s stack

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();
}
```



What return address is pushed onto stack?

- Address of next line in context of runningTCB

When is this line executed?

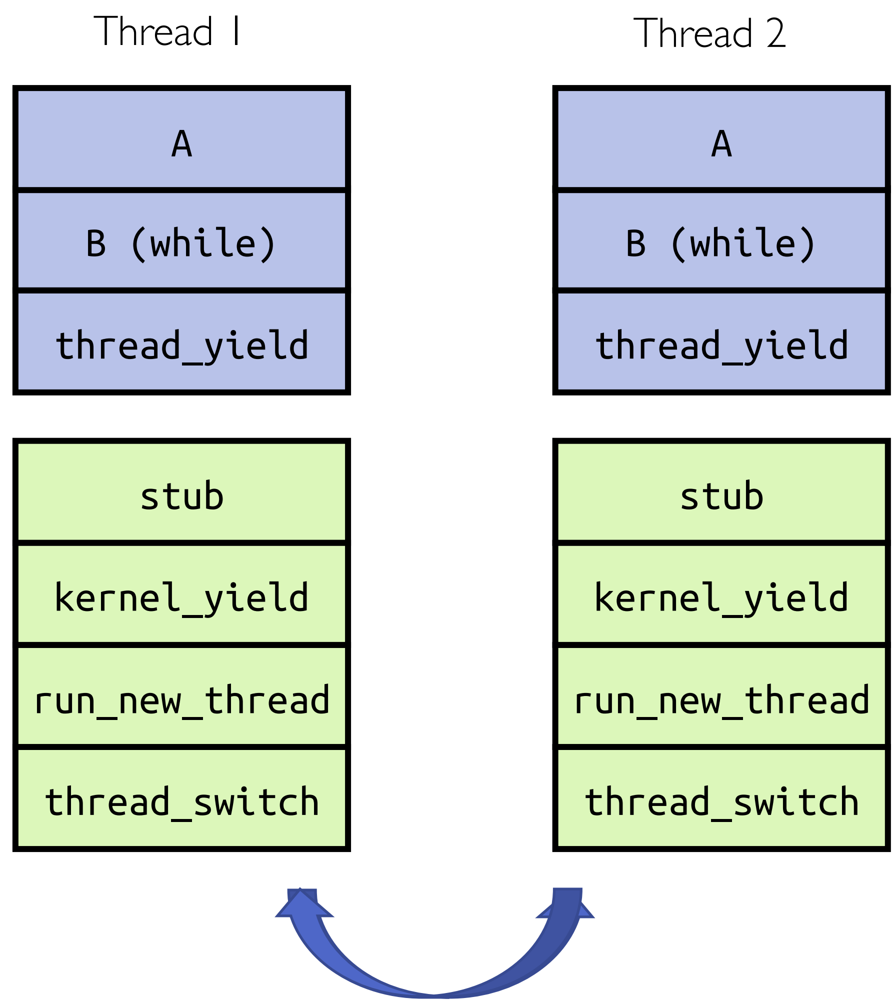
- Whenever another thread switches back to this thread

How Do Stacks Look Like?

- Two threads run following code

```
A() {
    B();
}

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



Switch Details

- What if you make mistakes in implementing switch?
  - Suppose you forget to save/restore register 32
  - Get intermittent failures depending on when context switch occurred and whether new thread uses register 32
  - System will give wrong result without warning

- Can you devise exhaustive test to test switch code?
  - No! Too many combinations and inter-leavings

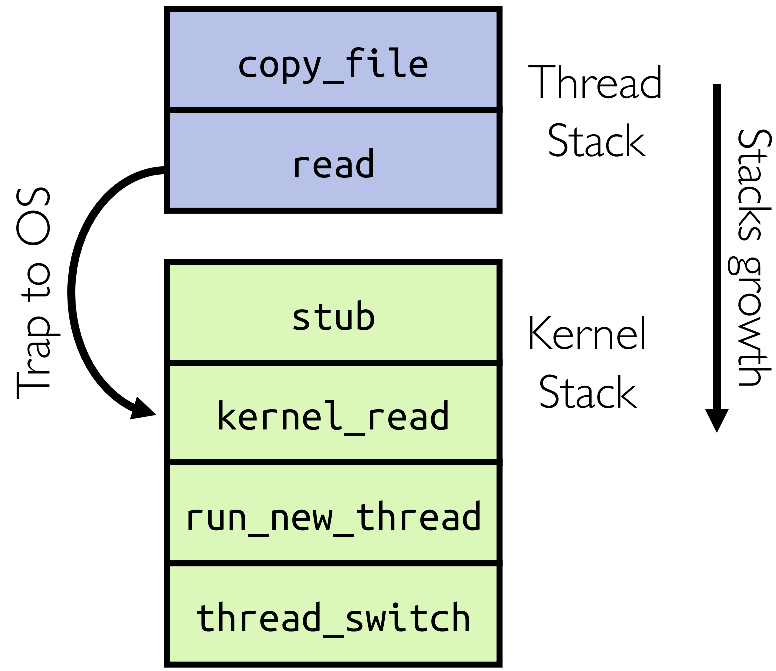
A Subtlety: `dummy_switch_frame(newTCB)`

- Newly-created thread will run after OS runs **`switch`**
- Kernel stack of new thread should be the same as others
- Recall:

```
thread_switch(oldTCB, newTCB) {
    pushad;
    oldTCB->sp = sp;
    sp = newTCB->sp;
    popad;
    return();
}

push_dummy_switch_frame(newTCB) {
    *(newTCB->sp) = stub; // return to beginning of stub
    newTCB->sp--;
    newTCB->sp -= SizeOfPopad;
}
```

What Happens When Threads Block on I/O?



- User code invokes system call
- Read operation is initiated
- OS runs new thread or switches to ready thread

Involuntary Context Switch

- What happens if thread never does any I/O, never waits, and never yields?
- Could `compute_PI` grab all resources and never release processor?
  - Must find way that dispatcher can regain control!
- OS utilizes external events
- Interrupts are signals from hardware or software that stop running code and transfer control to kernel
  - E.g., timer is like alarm clock that goes off some milliseconds
- Interrupts are hardware-invoked context switch
- Interrupt handler is not a thread
  - No separate step to choose what to run next
  - Always run the interrupt handler immediately

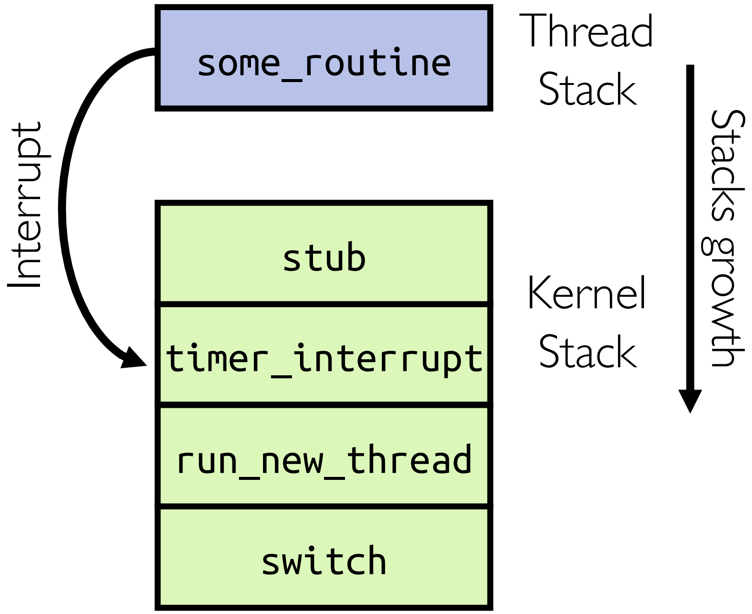
Aside: How to Track Running TCB?

- Problem: scheduler needs to know TCB of running thread
  - E.g., to suspend and switch to another thread
- On uniprocessor, just use single global variable
  - This doesn't work in multiprocessor, all kernel threads share code
- On multiprocessor, there are various methods
  - Compiler solution: dedicated register
    - E.g., r31 points to TCB running on each CPU; each CPU has its own r31
  - Hardware solution: special per-processor register
  - Software solution: fixed-size stacks
    - Put pointer to running TCB at the bottom of its stack
    - Find it by masking the current stack pointer

Timer Interrupt to Return Control

- Solution to our dispatcher problem
  - Use the timer interrupt to force scheduling decisions

```
void timer_interrupt() {
    do_periodic_houseKeeping();
    run_new_thread();
}
```

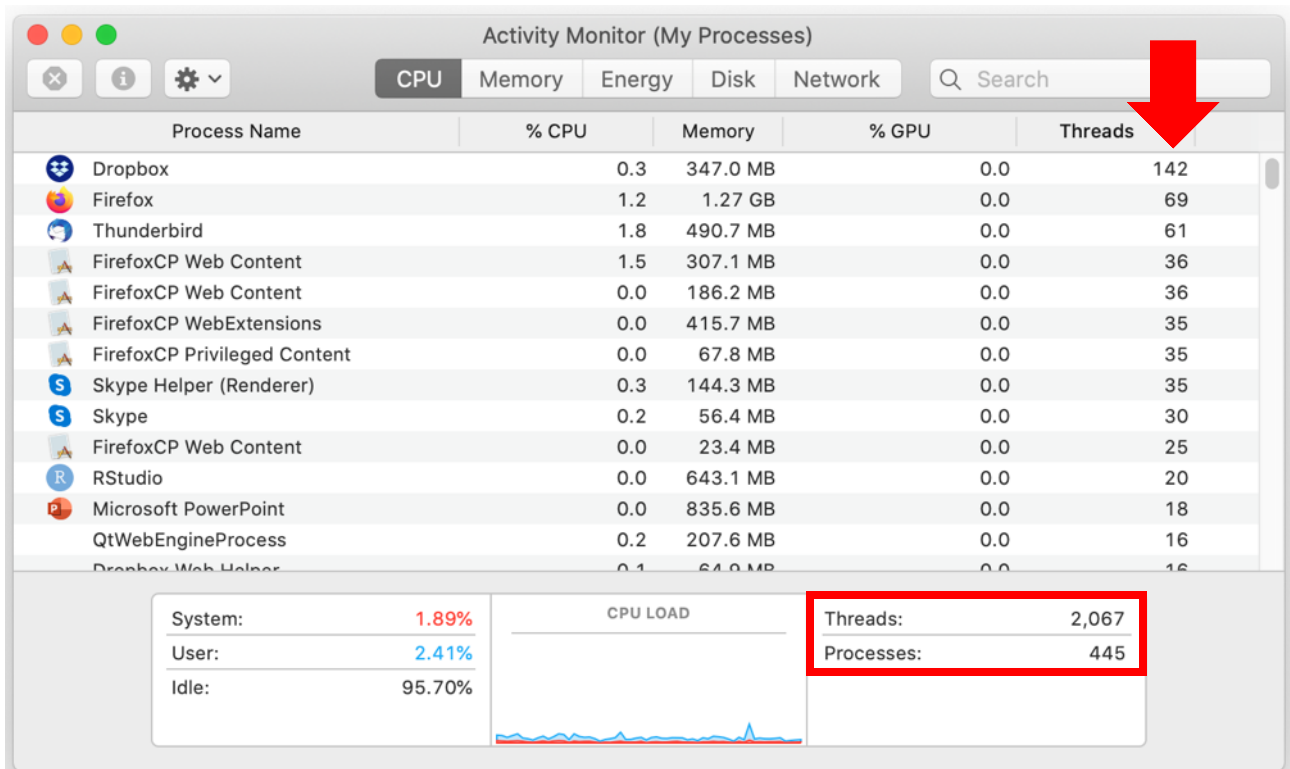


Outline

- Implementation of kernel threads
  - Create, yield, switch, etc.
- User-level threads with and without kernel support
- Implementation of synchronization objects
  - Mutex, semaphore, condition variable

Some Numbers

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



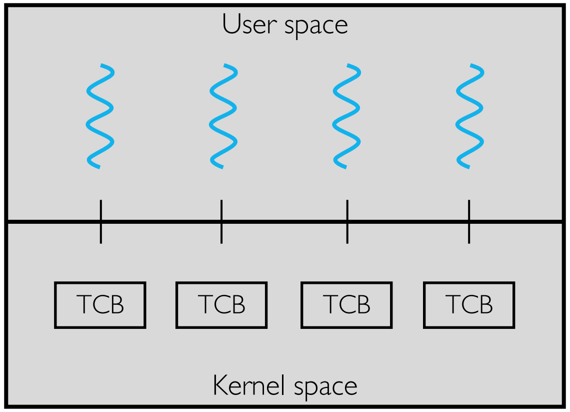
Some Numbers (cont.)

- Frequency of performing context switches is ~10-100ms
- Context switch time in Linux is ~3-4 us (Intel i7 & Xeon E5)
  - Thread switching faster than process switching (~100 ns)
- Switching across cores is ~2x more expensive than within-core
- Context switch time increases sharply with size of working set*
  - Can increase ~100x or more

- Moral: overhead of context switching depends mostly on cache limits and process or thread's hunger for memory

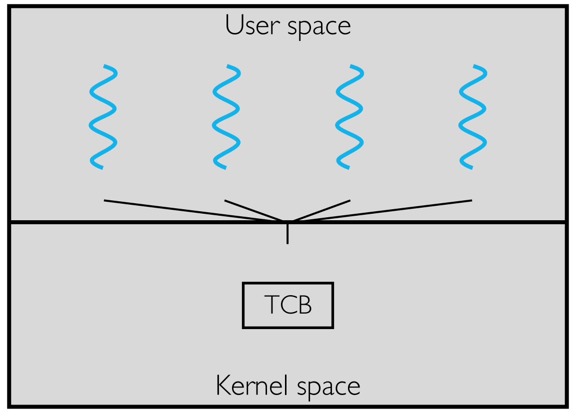
* Working set is subset of memory used by process in time window

Kernel- vs. User-managed Threads



- We have been talking about kernel-managed threads
  - Each user-mode thread maps to one TCB
  - Every thread can run or block independently
  - One process may have several threads waiting on different events
  - Examples: Windows, Linux
- Downside of kernel-managed threads: a bit expensive
  - Need to make crossing into kernel mode to schedule
  - Solution: user-managed threads

User-managed Threads



- Lighter weight option
  - Many user-mode threads are mapped to single TCB
  - User program provides scheduler and thread package
  - Examples: Solaris Green Threads, GNU Portable Threads
- 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*
    - Have kernel inform user level when thread blocks …

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

- Implementation of kernel threads
  - Create, yield, switch, etc.
- User-level threads with and without kernel support
- Implementation of synchronization objects
  - Mutex, semaphore, condition variable

Implementing Synchronization Objects

Programs	Bounded Buffers
Synch Objects	Mutex Semaphores Condition Variables
Atomic Inst	Load/Store Disable Interrupts Test&Set

Mutex Implementation - Take 1: Using only Load and Store

```
// Thread A
valueA = BUSY;
turn = 1;
while (valueB == BUSY && turn == 1);
// critical section
valueA = FREE;
```

```
// Thread B
valueB = BUSY;
turn = 0;
while (valueA == BUSY && turn == 0);
// critical section
valueB = FREE;
```

- This works, but it's very unsatisfactory
  - Way too complex – even for two threads!
    - It's hard to convince yourself that this really works
    - Reasoning is even harder when modern compilers/hardware reorder instructions
  - Thread A's code is different from thread B's – what if there are lots of threads?
    - Code would have to be slightly different for each thread (see *Peterson's algorithm*)
  - Thread A is busy-waiting while waiting for thread B (consuming CPU cycles)

Mutex Implementation - Take 2: Disabling Interrupts

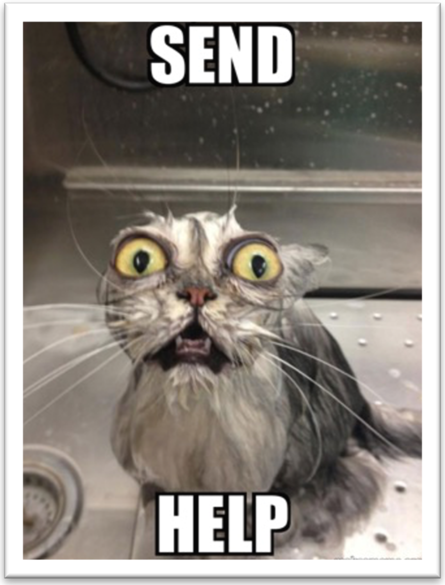
- Recall: context switching is triggered in two ways
  - Voluntary: thread does something to relinquish CPU
  - Involuntary: interrupts cause dispatcher to take CPU
- On uniprocessors, we can avoid context switching by
  - Avoiding voluntary context switches
  - Preventing involuntary context switches 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 2.5: 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();
}
```

Implementation of Mutex - Take 2.5 (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, chosedTCB);
        // 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

Mutex Implementation: 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 form 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?

```
Mutext::lock() {
    disable_interrupts();
    if (value == BUSY) {
        waiting.add(runningTCB);
        runningTCB->state = WAITING;
        chosenTCB = ready_list.get_nextTCB();
        thread_switch(runningTCB, chosedTCB);
        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

Thread A		Thread B
`disable_interrupts()` `thread_switch()`	context switch →	

		`thread_switch() return`
		`enable_interrupts()`
		.
		.
		.
		`disable_interrupts()`
		`thread_switch()`

`thread_switch() return`
`enable_interrupts()`
.
.
	← context switch	

Problems with Take 2.5

- 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

Recall: Examples of Read-Modify-Write Instructions

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

- ```
  swap (&address, register) {          /* x86 */
        temp = M[address];             /* swap register's value to
        M[address] = register;            value at "address" */
        register = temp;
  }
  ```

- ```
  compare&swap (&address, reg1, reg2) {   /* 68000 */
        if (reg1 == M[address]) {
            M[address] = reg2;
            return success;
        } else {
            return failure;
        }
  }
  ```
  ```

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 **`text&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;
    }
    mutext_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_spilock` 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;` ⇒ `spinlock.lock;`
    - `enable interrupts` ⇒ `spinlock.unlock;`

Recap: Mutexes Using Interrupts

```
Mutex::lock();
 …
 critical section;
 …
Mutex::unlock();
```

```
lock() {
  disable_interrupts();
}
```

```
unlock() {
  enable_interrupts();
}
```

If one thread is in critical section, no other activity (including OS) can run!

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

```
unlock() {
  // Short busy-wait time
  disable_interrupts();
  if (!waiting.empty()) {
    // take thread off wait queue
    // place it on ready queue;
  } else {
    value = FREE;
  }
  enable_interrupts();
}
```

Recap: Mutexes Using Spinlock (test&set)

```
Mutex::lock();
 …
 critical section;
 …
Mutex::unlock();
```

```
int value = FREE
lock() {
  while (test&set(value));
}
```

```
unlock() {
  value = FREE;
}
```

Threads waiting to enter critical section busy-wait

```
Spinlock mutex_spinlock;
int value = FREE;
lock() {
  // Short busy-wait time
  mutex_spinlock.lock();
  if (value == BUSY) {
    // put thread on wait queue
    // and go to sleep
  } else {
    value = BUSY;
    mutex_spinlock.unlock();
  }
}
```

```
Release() {
  // Short busy-wait time
  mutex_spinlock.lock();
  if (!waiting.empty()) {
    // take thread off wait queue
    // place it on ready queue
  } else {
    value = FREE;
  }
  mutex_spinlock.unlock();
}
```

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-level code using kernel-level threads
- Does not work properly for user-level code using kernel-level 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 inv ariant on interrupts - disable before calling thread_switch() and enable when thread_switch() returns

Recall: Rules for Using Mutex

- Mutex should be initially free
- Never access shared data without locking mutex
  - Danger! Don't do it even if it's tempting!
- Always lock mutex before accessing shared data
  - Best place for locking: beginning of procedure!
- Always unlock mutex after finishing with shared data
  - Best place for unlocking: end of procedure!
  - Only the one who has locked mutex can unlock it
  - DO NOT throw mutex for someone else to unlock

Lock Before Accessing Shared Data, ALWAYS!

```
getP() {
   if (p == NULL) {
      mutex.lock();
      if (p == NULL) {
         temp = malloc(sizeof(…));
         temp->field1 = …;
         temp->field2 = …;
         p = temp;
      }
      mutex.unlock();
   }
  return p;
}
```

- Safe but expensive solution is

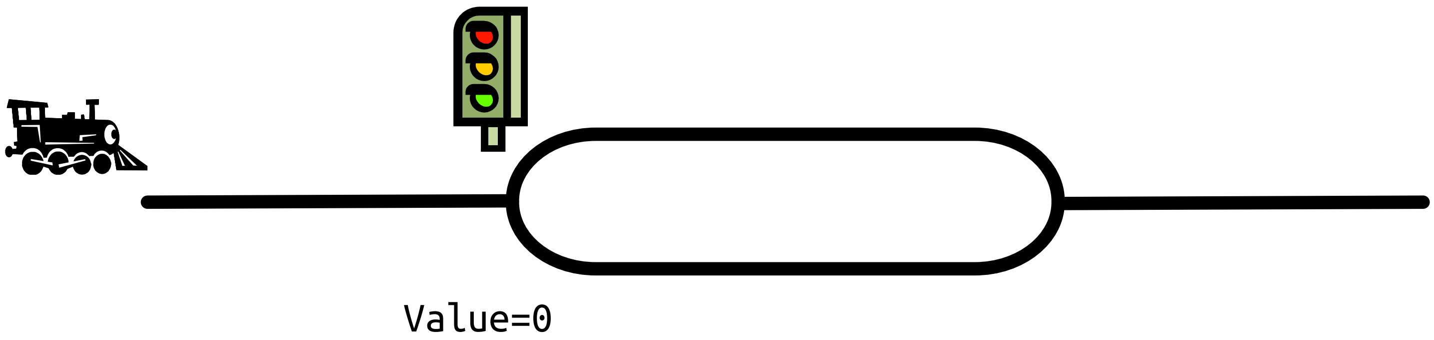
```
getP() {
   mutex.lock();
   if (p == NULL) {
      temp = malloc(sizeof(…));
      temp->field1 = …;
      temp->field2 = …;
      p = temp;
   }
   mutex.unlock();
   return p;
}
```

Does this work?

- No! Compiler/HW could make `p` point to `temp` before its fields are set
- This is called double-checked locking

Recall: Semaphores

- First defined by Dijkstra in late 60s
- Main synchronization primitive used in original UNIX
- Semaphore has non-negative integer value and 2 operations
  - `P()`: atomic operation that waits for semaphore to become positive, then decrements it by one
  - `V()`: atomic operation that increments semaphore by one, waking up a waiting `P()`, if any



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

Semaphores are Harmful!

"During system conception it transpired that we used the semaphores in two completely different ways. The difference is so marked that, looking back, one wonders whether it was really fair to present the two ways as uses of the very same primitives. On the one hand, we have the semaphores used for <u>mutual exclusion</u>, on the other hand, the <u>private semaphores</u>."

Dijkstra "The structure of the 'THE'-Multiprogramming System" *Communications of the ACM* v. 11 n. 5 May 1968.

Recall: Monitors and Condition Variables

- Problem: semaphores are dual purpose:
  - They are used for both mutex and scheduling constraints
  - Example: the fact that flipping of **P**'s in bounded buffer gives deadlock is not immediately obvious

- Solution: use mutexes for mutual exclusion and condition variables (CV) for scheduling constraints

- Definition: monitor is one mutex with zero or more condition variables for managing concurrent access to shared data
  - Some languages like Java provide this natively
  - Most others use actual mutex and condition variables

Recall: Condition Variables Operations

- `wait(Mutex *mutex)`
  - Atomically unlock mutex and relinquish processor
  - Relock the mutex when wakened
- `signal()`
  - Wake up a waiter, if any
- `broadcast()`
  - Wake up all waiters, if any

Recall: Properties of Condition Variables

- Condition variables are memoryless
  - No internal memory except a queue of waiting threads
  - No effect in calling `signal/broadcast` on empty queue

- ALWAYS lock mutex before calling `wait(), signal(), broadcast()`
  - In Birrell paper, he says you can call `signal()` without locking – IGNORE HIM (this is only an optimization)

- Calling `wait()` atomically adds thread to wait queue and unlocks mutex

- Re-enabled waiting threads may not run immediately
  - No atomicity between `signal/broadcast` and the return from `wait`

Example: Bounded Buffer Implementation with Monitors

```
Mutex mutex;
CV emptyCV, fullCV;

produce(item) {
   mutex.lock();                          // lock mutex
   while (queue.size() == MAX)            // wait until there is
      fullCV.wait(&mutex);                // space
   queue.add(item);
   emptyCV.signal();                      // signal waiting costumer
   mutex.unlock();                        // unlock mutex
}
consume() {
   mutex.lock();                          // get lock
   while (queue.empty())                  // wait until there is item
      emptyCV.wait(&mutex);
   item = queue.remove();
   fullCV.signal();                       // signal waiting producer
   mutex.unlock();                        // unlock mutex
   return item;
}
```

Mesa vs. Hoare Monitors

- Consider piece of `consume()` code

```
while (queue.empty())
    emptyCV.wait(&mutex);
```

- Why didn't we do this?

```
if (queue.empty())
    emptyCV.wait(&mutex);
```

- Answer: it depends on the type of scheduling
  - Hoare style
  - Mesa style

Hoare Monitors

- Signaler gives up mutex and processor to waiter – waiter runs immediately
- Waiter gives up mutex and processor back to signaler when it exits critical section or if it waits again

```
…
mutex.lock()
…
emptyCV.signal();
…
mutex.unlock();
```

mutex & CPU

mutex & CPU

```
mutex.lock()
…
if (queue.empty())
  emptyCV.wait(&mutex);
…
mutex.unlock();
```

Mesa Monitors

- Signaler keeps mutex and processor
- Waiter placed on ready queue with no special priority
- Practically, need to check condition again after wait
- Most real operating systems

```
…
mutex.lock()
…
emptyCV.signal();
…
mutex.unlock();
```

Put waiting thread on ready queue

schedule waiting thread

```
mutex.lock()
…
while (queue.empty())
  emptyCV.wait(&mutex);
…
mutex.unlock();
```

Mesa Monitor: Why "`while()`"?

- What if we use "**`if`**" instead of "**`while`**" in bounded buffer example?

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

```
produce(item) {
  mutex.lock();
  if (queue.size() == MAX)
    fullCV.wait(&mutex);
  queue.add(item);
  emptyCV.signal();
  mutex.unlock();
}
```

Use "**`if`**" instead of "**`while`**"

Mesa Monitor: Why "`while()`"? (cont.)

App. Shared State

queue

Monitor

mutex: unlocked

emptyCV queue → NULL

CPU State

Running: T1

ready queue → NULL

…

T1 (Running)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "while()"? (cont.)

App. Shared State

queue

Monitor

mutex: locked (T1)
emptyCV queue → NULL

CPU State

Running: T1
ready queue → NULL
...

T1 (Running)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "`while()`"? (cont.)

App. Shared State

queue

Monitor

mutex: unlocked

emptyCV queue → T1

CPU State

Running:

ready queue → NULL

…

`wait(&lock)` puts thread on emptyCV queue and releases lock

T1 (Waiting)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "while()"? (cont.)

App. Shared State

queue

Monitor

mutex: unlocked
emptyCV queue → T1

CPU State

Running: T2
ready queue → NULL
...

T1 (Waiting)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T2 (Running)

```
produce(item) {
  mutex.lock();
  if (queue.size()==MAX)
    fullCV.wait(&mutex);
  queue.add(item);
  emptyCV.signal();
  mutex.unlock();
}
```

Mesa Monitor: Why “while()”? (cont.)

App. Shared State

queue

Monitor

mutex: locked (T2)

emptyCV queue → T1

CPU State

Running: T2

ready queue → NULL

…

T1 (Waiting)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T2 (Running)

```
produce(item) {
  mutex.lock();
  if (queue.size()==MAX)
    fullCV.wait(&mutex);
  queue.add(item);
  emptyCV.signal();
  mutex.unlock();
}
```

Mesa Monitor: Why "while()"? (cont.)

App. Shared State

queue

Monitor

mutex: locked (T2)

emptyCV queue → NULL

CPU State

Running: T2

ready queue → T1

...

`signal()` wakes up and moves it to ready queue

T1 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T2 (Running)

```
produce(item) {
  mutex.lock();
  if (queue.size()==MAX)
    fullCV.wait(&mutex);
  queue.add(item);
  emptyCV.signal();
  mutex.unlock();
}
```

Mesa Monitor: Why "while()"? (cont.)

App. Shared State

queue

Monitor

mutex: locked (T2)
emptyCV queue → NULL

CPU State

Running: T2
ready queue → T1, T3
...

T1 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T2 (Running)

```
produce(item) {
  mutex.lock();
  if (queue.size()==MAX)
    fullCV.wait(&mutex);
  queue.add(item);
  emptyCV.signal();
  mutex.unlock();
}
```

T3 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "`while()`"? (cont.)

App. Shared State

queue

Monitor

mutex: unlocked
emptyCV queue → NULL

CPU State

Running:
ready queue → T1,T3
…

T1 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T2 (Terminated)

```
produce(item) {
  mutex.lock();
  if (queue.size()==MAX)
    fullCV.wait(&mutex);
  queue.add(item);
  emptyCV.signal();
  mutex.unlock();
}
```

T3 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "`while()`"? (cont.)

App. Shared State

queue

Monitor

mutex: unlocked
emptyCV queue → NULL

CPU State

Running: T3
ready queue → T1

T3 is scheduled first

T1 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T3 (Running)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "`while()`"? (cont.)

App. Shared State

queue

Monitor

mutex: locked (T3)
emptyCV queue → NULL

CPU State

Running: T3
ready queue → T1
...

T1 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T3 (Running)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "`while()`"? (cont.)

App. Shared State

queue

Monitor

mutex: locked (T3)
emptyCV queue → NULL

CPU State

Running: T3
ready queue → T1
…

T1 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T3 (Running)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "`while()`"? (cont.)

App. Shared State

queue

Monitor

mutex: unlocked

emptyCV queue → NULL

CPU State

Running:

ready queue → T1

…

T1 (Ready)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

T3 (Terminated)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "`while()`"? (cont.)

App. Shared State

queue

Monitor

mutex: locked (T1)
emptyCV queue → NULL

CPU State

Running: T1
ready queue → NULL
...

T1 (Running)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "`while()`"? (cont.)

App. Shared State

queue

Monitor

mutex: locked (T1)
emptyCV queue → NULL

CPU State

Running: T1
ready queue → NULL
…

T1 (Running)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Error!

Mesa Monitor: Why "`while()`"? (cont.)

App. Shared State

queue

Monitor

mutex: locked (T1)
emptyCV queue → NULL

CPU State

Running: T1
ready queue → NULL
…

T1 (Running)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Check again if empty!

Mesa Monitor: Why "`while()`"? (cont.)

App. Shared State

queue

Monitor

mutex: unlocked

emptyCV queue → T1

CPU State

Running:

ready queue → NULL

…

T1 (Waiting)

```
consume() {
  mutex.lock();
  if (queue.empty())
    emptyCV.wait(&mutex);
  item = queue.remove();
  fullCV.signal();
  mutex.unlock();
  return item;
}
```

Mesa Monitor: Why "`while()`"? (cont.)

When waiting upon a *Condition*, a spurious wakeup is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a *Condition* should always be waited upon in a loop, testing the state predicate that is being waited for

From Java User Manual

Condition Variable vs. Semaphore

- CV's `signal()` has no memory
  - If `signal()` is called before `wait()`, then signal is waisted
- Semaphore's `V()` has memory
  - If `V()` is called before `P()`, `P()` will not wait
- Generally, it's better to use monitors but not always
- Example: interrupt handlers
  - Shared memory is used concurrently by interrupt handler and kernel thread
  - Interrupt handler cannot use mutexes
  - Kernel thread checks for data and calls `wait()` if there is no data
  - Interrupt handler write to shared memory and calls `signal()`
    - This is called *naked notify* because interrupt handler hasn't locked mutex (why?)
  - This may not work if signal comes before kernel thread calls wait
  - Common solution is to use semaphores instead

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 `class CV` does not 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 1)

```
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

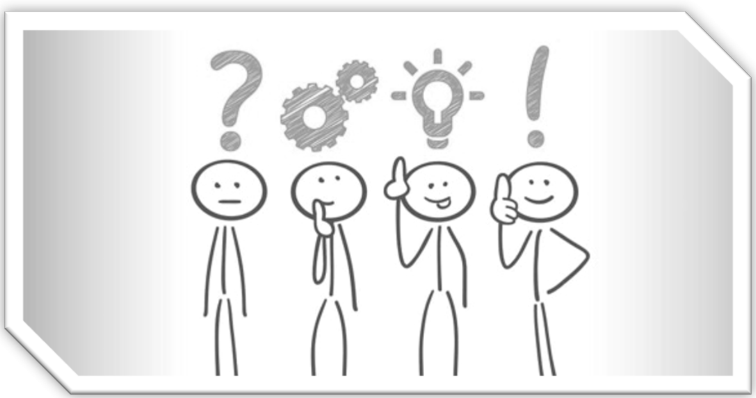
```
wait(*mutex) {
    semaphore = new Semaphore; // a semaphore per waiting thread
    queue.add(semaphore);  // queue for waiting threads
    mutex->unlock();
    semaphore.P();
    mutex->lock();
}

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