Synchronization

Today: Implementation issues

Readers/Writers Lock

- A common variant for mutual exclusion
  - One writer at a time, if no readers
  - Many readers, if no writer
- How might we implement this?
  - ReaderAcquire(), ReaderRelease()
  - WriterAcquire(), WriterRelease()
  - Need a lock to keep track of shared state
  - Need condition variables for waiting if readers/writers are in progress
  - Some state variables

Readers/Writers Lock

Lock lock = FREE

CV okToRead = nil

CV okToWrite = nil

AW = 0 //active writers

AR = 0   // active readers

WW = 0 // waiting writers

WR = 0 // waiting readers

```
Lock lock = FREE

CV okToRead = nil
CV okToWrite = nil

AW = 0
AR =  0
WW = 0
WR = 0
```

```
lock.Acquire();
while (AW > 0 || WW > 0) {
    WR++;
    okToRead.wait(&lock);
    WR--;
}
AR++;
lock.Release();

Read data

lock.Acquire();
AR--;
if (AR == 0 && WW > 0)
  okToWrite.Signal();
lock.Release();
```

```
lock.Acquire();
while (AW > 0 || AR > 0) {
    WW++;
    okToRead.wait(&lock);
    WW--;
}
AW++;
lock.Release();

Write data

lock.Acquire();
AW--;
if (WW > 0)
  okToWrite.Signal();
else if (WR > 0)
 okToRead.Broadcast();
lock.Release();
```

Readers/Writers Lock

- Can readers starve?
  - Yes: writers take priority
- Can writers starve?
  - Yes: a waiting writer may not be able to proceed, if another writer slips in between signal and wakeup

Readers/Writers Lock, w/o Writer Starvation Take 1

```
Writer() {
  lock.Acquire();
  // check if another thread is already waiting
  while ((AW + AR + WW) > 0) {
      WW++;
      okToWrite.Wait(&lock);
      WW--;
   }
  AW++;
  lock.Release();
```

Readers/Writers Lock w/o Writer Starvation Take 2

```
// check in
lock.Acquire();
myPos = numWriters++;
while ((AW + AR > 0 ||
        myPos > nextToGo) {
    WW++;
    okToWrite.Wait(&lock);
    WW--;
}
AW++;
lock.Release();
```

```
// check out
lock.Acquire();
AW--;
nextToGo++;
if (WW > 0) {
  okToWrite.Signal(&lock);
} else if (WR > 0)
   okToRead.Bcast(&lock);
lock.Release();
```

Readers/Writers Lock w/o Writer Starvation Take 3

```
// check in
lock.Acquire();
myPos = numWriters++;
myCV = new CV;
writers.Append(myCV);
while ((AW + AR > 0 ||
            myPos > nextToGo) {
    WW++;
    myCV.Wait(&lock);
    WW--;
}
AW++;
delete myCV;
lock.Release();
```

```
// check out
lock.Acquire();
AW--;
nextToGo++;
  if (WW > 0) {
        cv = writers.RemoveFront();
        cv.Signal(&lock);
  } else if (WR > 0)
        okToRead.Broadcast(&lock);
lock.Release();
```

Mesa vs. Hoare semantics

- Mesa
  - Signal puts waiter on ready list
  - Signaller keeps lock and processor
- Hoare
  - Signal gives processor and lock to waiter
  - When waiter finishes, processor/lock goes back to signaller
- All systems you will use are Mesa

FIFO Bounded Buffer (Hoare semantics)

```
get() {
    lock.acquire();
    if (front == tail) {
        empty.wait(&lock);
    }
    item = buf[front % MAX];
    front++;
    full.signal(&lock);
    lock.release();
    return item;
}
```

```
put(item) {
    lock.acquire();
    if ((tail – front) == MAX) {
        full.wait(&lock);
    }
    buf[last % MAX] = item;
    last++;
    empty.signal(&lock);
  // CAREFUL: someone else ran
    lock.release();
}
```

Initially: front = tail = 0; MAX is buffer capacity

empty/full are condition variables

FIFO Bounded Buffer (Mesa semantics)

- Create a condition variable for every waiter
- Queue condition variables (in FIFO order)
- Signal picks the front of the queue to wake up
- CAREFUL if spurious wakeups!

- Easily extends to case where queue is LIFO, priority, priority donation, …
  - With Hoare semantics, not as easy

FIFO Bounded Buffer (Mesa semantics, put() is similar)

```
get() {
    lock.acquire();
    myPosition = numGets++;
    cv = new CV;
    nextGet.append(cv);
    while (front < myPosition
            || front == tail) {
      cv.wait(&lock);
    }
    delete cv;
    item = buf[front % MAX];
    front++;
    if (next = nextPut.remove()) {
        next->signal(&lock);
    }
    lock.release();
    return item;
}
```

Initially: front = tail = numGets = 0; MAX is buffer capacity
nextGet, nextPut are queues of Condition Variables

Implementing Synchronization

Concurrent Applications

Semaphores | Locks | Condition Variables

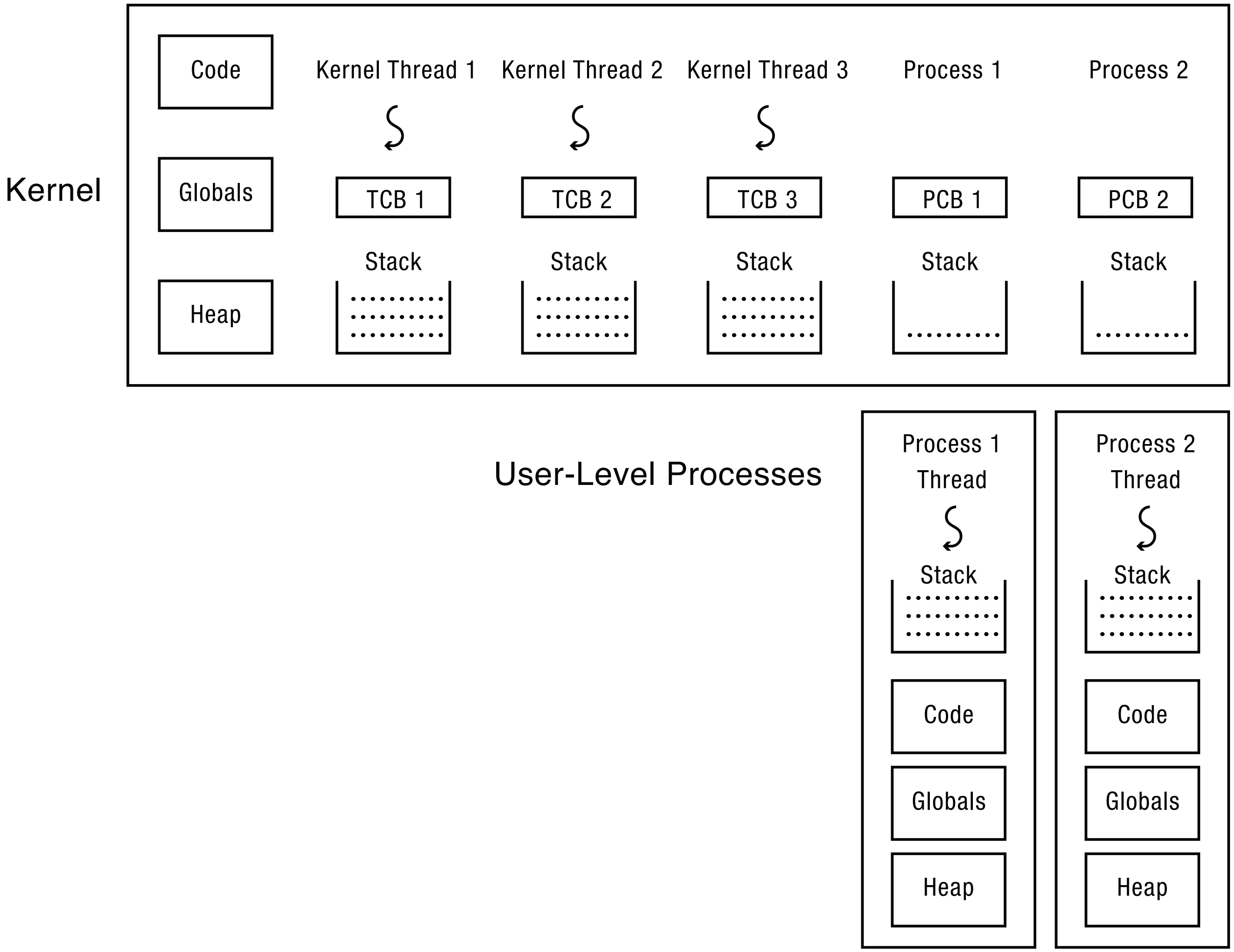
Interrupt Disable | Atomic Read/Modify/Write Instructions

Multiple Processors | Hardware Interrupts

Implementing Threads: Roadmap

- Kernel threads
  - Thread abstraction only available to kernel
  - To the kernel, a kernel thread and a single threaded user process look quite similar
- Multithreaded processes using kernel threads (Linux, MacOS, Windows)
  - Kernel thread operations available via syscall
- User-level threads (Windows)
  - Thread operations without system calls

Multithreaded OS Kernel



Thread Context Switch

- Voluntary
  - Thread_yield
  - Thread_join (if child is not done yet)
- Involuntary
  - Interrupt or exception
  - Some other thread is higher priority

Voluntary thread context switch

- Called by old thread
- Save registers on old stack
- Switch to new stack, new thread
- Restore registers from new stack
- Return to new thread

- Exactly the same with kernel threads or user threads

x86 swtch

```
push %rbp
push %rbx
push %r11
push %r12
push %r13
push %r14
push %r15

mov %rsp, (%rdi)
mov %rsi, %rsp
```

```
pop %r15
pop %r14
pop %r13
pop %r12
pop %r11
pop %rbx
pop %rbp

ret
```

// save/restore callee save registers, not caller save

A Subtlety

- Thread_create puts new thread on ready list
- Some thread calls switch, picks that thread to run next
  - Saves old thread state to stack
  - Restores new thread state from stack
- What does the new thread stack contain so this will work?
  - Set up thread's stack as if it had saved its state in switch
  - "returns" to PC saved at base of stack to run thread

Two Threads Call Yield

Thread 1's instructions	**Thread 2's instructions**	**Processor's instructions**
"return" from thread_switch into stub		"return" from thread_switch into stub
call go		call go
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch
save thread 1 state to TCB		save thread 1 state to TCB
load thread 2 state		load thread 2 state
	"return" from thread_switch into stub	"return" from thread_switch into stub
	call go	call go
	call thread_yield	call thread_yield
	choose another thread	choose another thread
	call thread_switch	call thread_switch
	save thread 2 state to TCB	save thread 2 state to TCB
	load thread 1 state	load thread 1 state
return from thread_switch		return from thread_switch
return from thread_yield		return from thread_yield
call thread_yield		call thread_yield
choose another thread		choose another thread
call thread_switch		call thread_switch

Involuntary Thread/Process Switch (Simple, Slow Version)

- Timer or I/O interrupt
  - Tells OS some other thread/process should run
- End of interrupt handler calls switch, before resuming the trapframe
- When thread is switched back in, resumes the handler
- Handler restores the trapframe to resume the user process

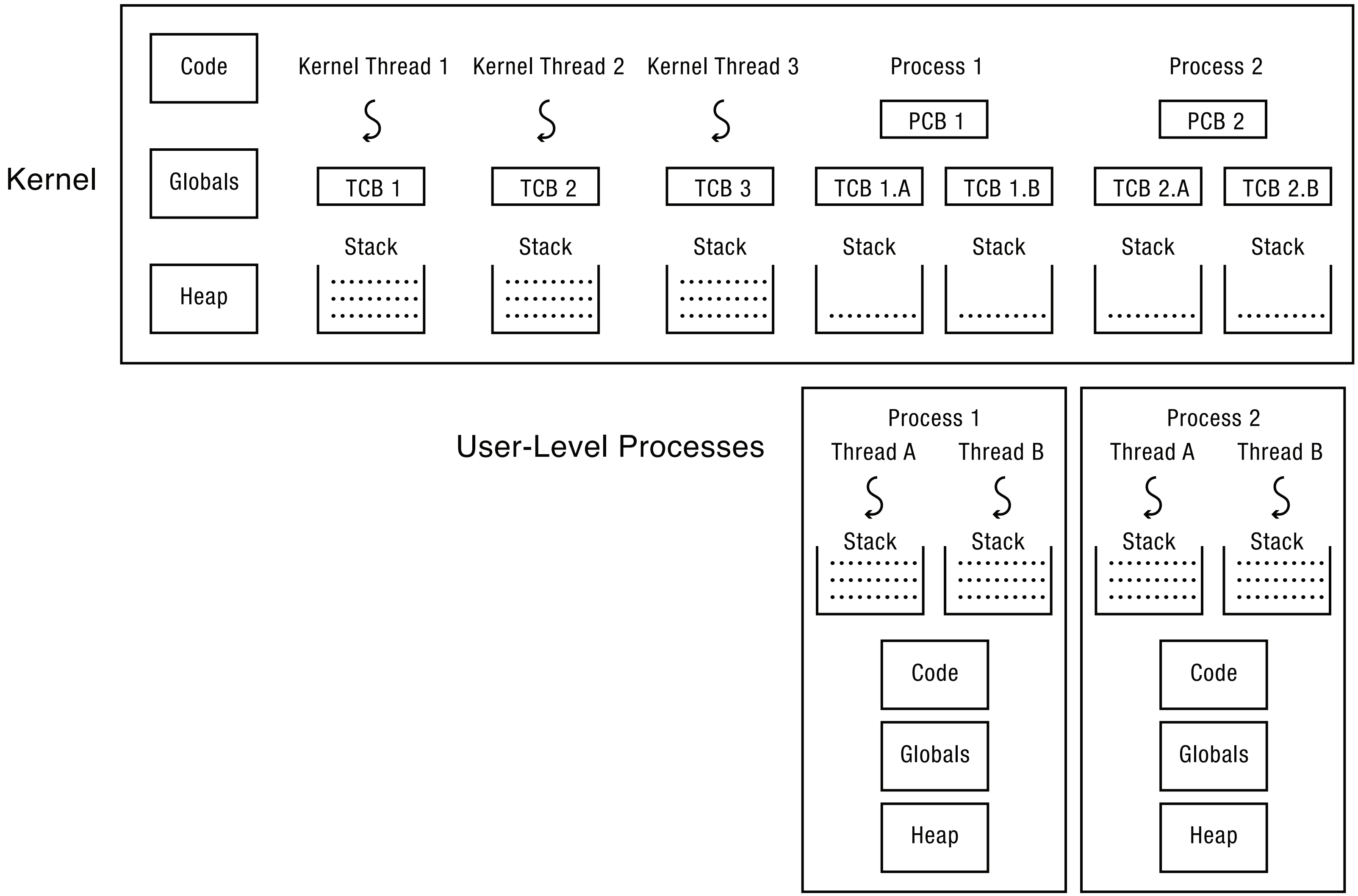
Involuntary Thread/Process Switch (Fast Version)

- Interrupt handler saves state of interrupted thread on trapframe
- At end of handler, switch to a new thread
- We don't need to come back to the interrupt handler!
- Instead: change switch so that it can restore directly from the trapframe
- On resume, pop trapframe to restore directly to the interrupted thread

Multithreaded User Processes (Take 1)

- User thread = kernel thread (Linux, MacOS)
  - System calls for thread fork, join, exit (and lock, unlock,…)
  - Kernel does context switch
  - Simple, but a lot of transitions between user and kernel mode

Multithreaded User Processes (Take 1)



Multithreaded User Processes (Take 2)

- Green threads (early Java)
  - User-level library, within a single-threaded process
  - Library does thread context switch
  - Preemption via upcall/UNIX signal on timer interrupt
  - Use multiple processes for parallelism
    - Shared memory region mapped into each process

Multithreaded User Processes (Take 3)

- Scheduler activations (Windows 8)
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - Thread library decides what thread to run next
- Upcall whenever kernel needs a user-level scheduling decision
  - Process assigned a new processor
  - Processor removed from process
  - System call blocks in kernel

Implementing Locks (Take 1)

Use memory load/store instructions

- See too much milk solution/Peterson's algorithm
- Complex
- Need memory barriers
- Hard to test/verify correctness

Implementing Locks (Take 2)

```
Lock::acquire() {
    oldIPL = setInterrupts(OFF);
    lockHolder = myTCB;
}
Lock::release() {
    ASSERT(lockholder == myTCB);
    lockHolder = NULL;
    setInterrupts(oldIPL); // implies memory barrier
}
```

Lock Implementation, Uniprocessor

```
Lock::acquire() {
    oldIPL = setInterrupts(OFF);
    if (value == BUSY) {
        waiting.add(myTCB);
        myTCB->state = WAITING;
        next = readyList.remove();
        switch(myTCB, next);
        myTCB->state = RUNNING;
    } else {
        value = BUSY;
        lockHolder = myTCB;
    }
    setInterrupts(oldIPL);
}
```

```
Lock::release() {
    ASSERT(lockHolder == myTCB);
    oldIPL = setInterrupts(OFF);
    if (!waiting.Empty()) {
        next = waiting.remove();
        next->state = READY;
        readyList.add(next);
        lockHolder = next;
    } else {
        value = FREE;
        lockHolder = NULL;
    }
    setInterrupts(oldIPL);
}
```

What thread is currently running?

- Thread scheduler needs to know the TCB of the currently running thread
  - To suspend and switch to a new thread
  - To check if the current thread holds a lock before acquiring or releasing it
- On a uniprocessor, easy: just use a global variable
  - Change the value in switch
- On a multiprocessor?

What thread is currently running? (Multiprocessor Version)

- Compiler dedicates a register
  - MIPS: s7 points to TCB running on this CPU
- Hardware register holds processor number
  - x86 RDTSCP: read timestamp counter and processor ID
  - OS keeps an array, indexed by processor ID, listing current thread on each CPU
- Fixed-size thread stacks: put a pointer to the TCB at the bottom of its stack
  - Find it by masking the current stack pointer

Mutual Exclusion Support on a Multiprocessor

- Read-modify-write instructions
  - Atomically read a value from memory, operate on it, and then write it back to memory
  - Intervening instructions prevented in hardware
  - Implies a memory barrier
- Examples
  - Test and set   // read old value, set value to 1
  - Intel: xchgb   // read old value, set new value
  - Compare and swap // test if old value has changed

    // if not change it

Spinlocks

A spinlock waits in a loop for the lock to become free

- Assumes lock will be held for a short time
- Used to protect the CPU scheduler and to implement locks, CVs

```
loop: // pointer to lock value in (%eax)
lock xchgb (%eax), 1
jnz loop
```

Spinlocks

```
Spinlock::acquire() {
   while (testAndSet(&lockValue) == BUSY)
      ;
   lockHolder = myTCB;
}
Spinlock::release() {
   ASSERT(lockHolder == myTCB);
   lockHolder = NULL;
   memorybarrier();
   lockValue = FREE;
}
```

Spinlocks and Interrupt Handlers

- Suppose an interrupt handler needs to access some shared data => acquires spinlock
  - To put a thread on the ready list (I/O completion)
  - To switch between threads (time slice)
- What happens if a thread holds that spinlock with interrupts enabled?
  - Deadlock is possible unless ALL uses of that spinlock are with interrupts disabled

How Many Spinlocks?

- Various data structures
  - Queue of waiting threads on lock X
  - Queue of waiting threads on lock Y
  - List of threads ready to run
- One spinlock per kernel? Bottleneck!
- One spinlock per lock
- One spinlock for the scheduler ready list
  - Per-core ready list: one spinlock per core
  - Scheduler lock requires interrupts off!

Lock Implementation, Multiprocessor

```
Lock::acquire() {
  spinLock.acquire();
  if (value == BUSY) {
    waiting.add(myTCB);
    suspend(&spinlock);
    ASSERT(lockHolder ==
              myTCB);
  } else {
    value = BUSY;
    lockHolder = myTCB;
  }
  spinLock.release();
}
```

```
Lock::release() {
  ASSERT(lockHolder = myTCB);
  spinLock.acquire();
  if (!waiting.Empty()) {
    next = waiting.remove();
    lockHolder = next;
    sched.makeReady(next);
  } else {
    value = FREE;
    lockHolder = NULL;
  }
  spinLock.release();
}
```

Lock Implementation, Multiprocessor

```
Sched::suspend(SpinLock *sl) {
   TCB *next;
   oldIPL = setInterrupts(OFF);
   schedSL.acquire();
   sl->release();
   myTCB->state = WAITING;
   next = readyList.remove();
   switch(myTCB, next);
   myTCB->state = RUNNING;
   schedSL.release();
   setInterrupts(oldIPL);
}
```

```
Sched::makeReady(TCB
    *thread) {
   oldIPL =setInterrupts(OFF);
   schedSL.acquire();
   readyList.add(thread);
   thread->state = READY;
   schedSL.release();
   setInterrupts(oldIPL);
}
```

Lock Implementation, Linux

- Most locks are free most of the time. Why?
  - Linux implementation takes advantage of this fact
- Fast path
  - If lock is FREE and no one is waiting, two instructions to acquire the lock
  - If no one is waiting, two instructions to release
- Slow path
  - If lock is BUSY or someone is waiting (see multiproc)
- Two versions: one with interrupts off, one w/o

Lock Implementation, Linux

```
struct mutex {
 /* 1: unlocked ; 0: locked;
    negative : locked,
    possible waiters */
 atomic_t count;
 spinlock_t wait_lock;
 struct list_head wait_list;
};
```

```
// atomic decrement
// %eax is pointer to count
lock decl (%eax)
jns 1f // jump if not signed
        // (if value is now 0)
call slowpath_acquire
1:
```

Application Locks

- A system call for every lock acquire/release?
  - Context switch in the kernel!
- Instead:
  - Spinlock at user level
  - "Lazy" switch into kernel if spin for period of time
- Or scheduler activations:
  - Thread context switch at user level