Concurrency and Synchronization
Motivation

• Operating systems (and application programs) often need to be able to handle multiple things happening at the same time
  – Process execution, interrupts, background tasks, system maintenance

• Humans are not very good at keeping track of multiple things happening simultaneously

• Threads and synchronization are an abstraction to help bridge this gap
Why Concurrency?

• Servers
  – Multiple connections handled simultaneously

• Parallel programs
  – To achieve better performance

• Programs with user interfaces
  – To achieve user responsiveness while doing computation

• Network and disk bound programs
  – To hide network/disk latency
Definitions

• A thread is a single execution sequence that represents a separately schedulable task
  – Single execution sequence: familiar programming model
  – Separately schedulable: OS can run or suspend a thread at any time

• Protection is an orthogonal concept
  – Can have one or many threads per protection domain
Threads in the Kernel and at User-Level

• Multi-process kernel
  – Multiple single-threaded processes
  – System calls access shared kernel data structures

• Multi-threaded kernel
  – multiple threads, sharing kernel data structures, capable of using privileged instructions
  – UNIX daemon processes -> multi-threaded kernel

• Multiple multi-threaded user processes
  – Each with multiple threads, sharing same data structures, isolated from other user processes
  – Plus a multi-threaded kernel
Thread Abstraction

- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule
Question

Why do threads execute at variable speed?
# Programmer vs. Processor View

<table>
<thead>
<tr>
<th>Programmer’s View</th>
<th>Possible Execution #1</th>
<th>Possible Execution #2</th>
<th>Possible Execution #3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x = x + 1;</td>
<td>x = x + 1;</td>
<td>x = x + 1;</td>
</tr>
<tr>
<td></td>
<td>y = y + x;</td>
<td>y = y + x;</td>
<td>y = y + x;</td>
</tr>
<tr>
<td></td>
<td>z = x + 5y;</td>
<td>z = x + 5y;</td>
<td>z = x + 5y;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Thread is suspended.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Other thread(s) run.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Thread is resumed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>y = y + x;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>z = x + 5y;</td>
</tr>
</tbody>
</table>
Possible Executions

One Execution

Thread 1
Thread 2
Thread 3

Another Execution

Thread 1
Thread 2
Thread 3

Another Execution

Thread 1
Thread 2
Thread 3
Thread Operations

- **thread_create(thread, func, args)**
  - Create a new thread to run func(args)

- **thread_yield()**
  - Relinquish processor voluntarily

- **thread_join(thread)**
  - In parent, wait for forked thread to exit, then return

- **thread_exit**
  - Quit thread and clean up, wake up joiner if any
Thread Data Structures

<table>
<thead>
<tr>
<th>Shared State</th>
<th>Thread 1’s Per–Thread State</th>
<th>Thread 2’s Per–Thread State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>Thread Control Block (TCB)</td>
<td>Thread Control Block (TCB)</td>
</tr>
<tr>
<td></td>
<td>Stack Information</td>
<td>Stack Information</td>
</tr>
<tr>
<td></td>
<td>Saved Registers</td>
<td>Saved Registers</td>
</tr>
<tr>
<td></td>
<td>Thread Metadata</td>
<td>Thread Metadata</td>
</tr>
<tr>
<td>Global Variables</td>
<td>Stack</td>
<td>Stack</td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thread Lifecycle

- **Thread Creation**: `sthread_create()`
- **Ready**
  - Scheduler Resumes Thread
  - Thread Yield/Scheduler Suspends Thread: `sthread_yield()`
  - Event Occurs, Other Thread Calls: `sthread_join()`
- **Waiting**
  - Thread Waits for Event: `sthread_join()`
- **Running**
  - Scheduler Resumes Thread
  - Thread Exit: `sthread_exit()`
- **Finished**
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

Kernel

- Code
- Globals
- Heap

- TCB 1
- Stack

- TCB 2
- Stack

- TCB 3
- Stack

- PCB 1
- Stack

- PCB 2
- Stack

User-Level Processes

- Process 1
  - Thread
  - Stack
  - Code
  - Globals
  - Heap

- Process 2
  - Thread
  - Stack
  - Code
  - Globals
  - Heap
Implementing threads

• Thread_fork(func, args)
  – Allocate thread control block
  – Allocate stack
  – Build stack frame for base of stack (stub)
  – Put func, args on stack
  – Put thread on ready list
  – Will run sometime later (maybe right away!)

• stub(func, args):
  – Call (*func)(args)
  – If return, call thread_exit()
Thread Stack

• What if a thread puts too many procedures on its stack?
  – What happens in Java?
  – What happens in the Linux kernel?
  – What happens in OS/161?
  – What should happen?
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

• Save registers on old stack
• Switch to new stack, new thread
• Restore registers from new stack
• Return

• Exactly the same with kernel threads or user threads
  – xv6 hint: thread switch between kernel threads, not between user process and kernel thread
/* a0: pointer to old thread control block */
/* a1: pointer to new thread control block */
/* Allocate stack space for 10 registers. */
addi sp, sp, -40
/* Save the registers */
sw ra, 36(sp)
sw gp, 32(sp)
sw s8, 28(sp)
sw s6, 24(sp)
sw s5, 20(sp)
sw s4, 16(sp)
sw s3, 12(sp)
sw s2, 8(sp)
sw s1, 4(sp)
sw s0, 0(sp)
/* Store old stack pointer in old thread */
sw sp, 0(a0)

/* Get new stack pointer from new thread */
lw sp, 0(a1)
nop       /* delay slot for load */
/* Now, restore the registers */
lw s0, 0(sp)
lw s1, 4(sp)
lw s2, 8(sp)
lw s3, 12(sp)
lw s4, 16(sp)
lw s5, 20(sp)
lw s6, 24(sp)
lw s8, 28(sp)
lw gp, 32(sp)
lw ra, 36(sp)
nop       /* delay slot for load */
j ra       /* and return. */
addi sp, sp, 40   /* in delay slot */
x86 switch_threads

# Save caller's register state
pushl %ebx
pushl %ebp
pushl %esi
pushl %edi

# NOTE: %eax, etc. are ephemeral
pushl %esi
pushl %edi

# Get offset of struct thread.stack
mov thread_stack_stack_ofs, %edx

# Save current stack pointer
movl SWITCH_CUR(%esp), %eax

# Change stack pointer;
movl SWITCH_NEXT(%esp), %ecx
movl (%ecx,%edx,1), %esp

# stack points to new TCB
ret

# Restore caller's register state.
popl %edi
popl %esi
popl %ebp
popl %ebx
movl %esp, (%eax,%edx,1)
A Subtlety

• Thread_create puts new thread on ready list
• When it first runs, some thread calls switchframe
  – Saves old thread state to stack
  – Restores new thread state from stack
• Set up new thread’s stack as if it had saved its state in switchframe
  – “returns” to stub at base of stack to run func
Two Threads Call Yield

Thread 1’s instructions
“return” from thread_switch into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 1 state to TCB
load thread 2 state

Thread 2’s instructions
“return” from thread_switch into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 2 state to TCB
load thread 1 state

Processor’s instructions
“return” from thread_switch into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 1 state to TCB
load thread 2 state
“return” from thread_switch into stub
call go
call thread_yield
choose another thread
call thread_switch
save thread 2 state to TCB
load thread 1 state

return from thread_switch
return from thread_yield
call thread_yield
choose another thread
call thread_switch

return from thread_switch
return from thread_yield
call thread_yield
choose another thread
call thread_switch

...
Involuntary Thread/Process Switch

• Timer or I/O interrupt
  – Tells OS some other thread should run

• Simple version
  – End of interrupt handler calls switch()
  – When resumed, return from handler resumes kernel thread or user process
  – Thus, processor context is saved/restored twice (once by interrupt handler, once by thread switch)
Faster Thread/Process Switch

• What happens on a timer (or other) interrupt?
  – Interrupt handler saves state of interrupted thread
  – Decides to run a new thread
  – Throw away current state of interrupt handler!
  – Instead, set saved stack pointer to trapframe
  – Restore state of new thread
  – On resume, pops trapframe to restore 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
Synchronization
Synchronization Motivation

- When threads concurrently read/write shared memory, program behavior is undefined
  - Two threads write to the same variable; which one should win?
- Thread schedule is non-deterministic
  - Behavior changes when re-run program
- Compiler/hardware instruction reordering
- Multi-word operations are not atomic
Question: Can this panic?

Thread 1

p = someComputation();
pInitialized = true;

Thread 2

while (!pInitialized)

    q = someFunction(p);

if (q != someFunction(p))
    panic
Why Reordering?

• Why do compilers reorder instructions?
  – Efficient code generation requires analyzing control/data dependency
  – If variables can spontaneously change, most compiler optimizations become impossible

• Why do CPUs reorder instructions?
  – Write buffering: allow next instruction to execute while write is being completed

Fix: memory barrier
  – Instruction to compiler/CPU
  – All ops before barrier complete before barrier returns
  – No op after barrier starts until barrier returns
## Too Much Beer Example

<table>
<thead>
<tr>
<th>Time</th>
<th>Activity</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:35</td>
<td>Leave for store.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:45</td>
<td>Buy beer.</td>
<td></td>
<td>Leave for store.</td>
</tr>
<tr>
<td>9:50</td>
<td>Arrive home, put beer away.</td>
<td></td>
<td>Arrive at store.</td>
</tr>
<tr>
<td>9:55</td>
<td></td>
<td></td>
<td>Buy beer.</td>
</tr>
<tr>
<td>10:00</td>
<td></td>
<td></td>
<td>Arrive home, put beer away. No room!</td>
</tr>
</tbody>
</table>
Definitions

Race condition: output of a concurrent program depends on the order of operations between threads

Mutual exclusion: only one thread does a particular thing at a time
  – Critical section: piece of code that only one thread can execute at once

Lock: prevent someone from doing something
  – Lock before entering critical section, before accessing shared data
  – Unlock when leaving, after done accessing shared data
  – Wait if locked (all synchronization involves waiting!)
Too Much Beer, Try #1

• Correctness property
  – Someone buys if needed (liveness)
  – At most one person buys (safety)

• Try #1: leave a note
  
  if (!note)
  
    if (!beer) {
      leave note
      buy beer
      remove note
    }
Too Much Beer, Try #2

Thread A
leave note A
if (!note B) {
  if (!beer)
    buy beer
}
remove note A

Thread B
leave note B
if (!noteA) {
  if (!beer)
    buy beer
}
remove note B
Too Much Beer, Try #3

Thread A

leave note A
while (note B) // X
do nothing;
if (!beer)
  buy beer;
remove note A

Thread B

leave note B
if (!noteA) {  // Y
  if (!beer)
    buy beer;
}  
remove note B

Can guarantee at X and Y that either:
(i) Safe for me to buy
(ii) Other will buy, ok to quit
Lessons

• Solution is complicated
  – “obvious” code often has bugs

• Modern compilers/architectures reorder instructions
  – Making reasoning even more difficult

• Generalizing to many threads/processors
  – Even more complex: see Peterson’s algorithm
# Roadmap

**Concurrent Applications**

<table>
<thead>
<tr>
<th>Semaphores</th>
<th>Locks</th>
<th>Condition Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interrupt Disable</strong></td>
<td><strong>Atomic Read/Modify/Write Instructions</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multiple Processors</strong></td>
<td><strong>Hardware Interrupts</strong></td>
<td></td>
</tr>
</tbody>
</table>
Locks

• Lock::acquire
  – wait until lock is free, then take it
• Lock::release
  – release lock, waking up anyone waiting for it

1. At most one lock holder at a time (safety)
2. If no one holding, acquire gets lock (progress)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)
Question: Why only Acquire/Release?

• Suppose we add a method to a lock, to ask if the lock is free. Suppose it returns true. Is the lock:
  – Free?
  – Busy?
  – Don’t know?
Too Much Beer, #4

Locks allow concurrent code to be much simpler:

```java
lock.acquire();
if (!beer)
    buy beer
lock.release();
```
Lock Example: Malloc/Free

char *malloc (n) {
    heaplock.acquire();
    p = allocate memory
    heaplock.release();
    return p;
}

void free(char *p) {
    heaplock.acquire();
    put p back on free list
    heaplock.release();
    }

Rules for Using Locks

• Lock is initially free
• Always acquire before accessing shared data structure
  – Beginning of procedure!
• Always release after finishing with shared data
  – End of procedure!
  – Only the lock holder can release
  – DO NOT throw lock for someone else to release
• Never access shared data without lock
  – Danger!
Double Checked Locking

if (p == NULL) {
    lock.acquire();
    if (p == NULL) {
        p = newP();
    }
    lock.release();
} 

use p->field1

newP() {
    tmp = malloc(sizeof(p));
    tmp->field1 = ... 
    tmp->field2 = ... 
    return tmp;
}
Single Checked Locking

lock.acquire();
    if (p == NULL) {
        p = newP();
    }
lock.release();
use p->field1

newP() {
    tmp = malloc(sizeof(p));
    tmp->field1 = ...
    tmp->field2 = ...
    return tmp;
}
Example: Bounded Buffer

tryget() {
    lock.acquire();
    item = NULL;
    if (front < tail) {
        item = buf[front % MAX];
        front++;
    }
    lock.release();
    return item;
}

tryput(item) {
    lock.acquire();
    success = FALSE;
    if ((tail – front) < MAX) {
        buf[tail % MAX] = item;
        tail++;
        success = TRUE;
    }
    lock.release();
    return success;
}

Initially: front = tail = 0; lock = FREE; MAX is buffer capacity
Question

• If tryget returns NULL, do we know the buffer is empty?

• If we poll tryget in a loop, what happens to a thread calling tryput?
Condition Variables

• Waiting inside a critical section
  – Called only when holding a lock

• Wait: atomically release lock and relinquish processor
  – Reacquire the lock when wakened

• Signal: wake up a waiter, if any

• Broadcast: wake up all waiters, if any
methodThatWaits() {
  lock.acquire();
  // Read/write shared state
  while (!testSharedState()) {
    cv.wait(&lock);
  }
  // Read/write shared state
  lock.release();
}

methodThatSignals() {
  lock.acquire();
  // Read/write shared state
  // If testSharedState is now true
  cv.signal(&lock);
  // Read/write shared state
  lock.release();
}
Example: Bounded Buffer

```plaintext
class BoundedBuffer {
    private:
        int front, tail;
        std::mutex lock;
        std::condition_variable empty, full;
        std::vector<int> buf;
        const int MAX;

    public:
        BoundedBuffer(int capacity) : MAX(capacity) { front = tail = 0; }

        int get() {
            lock.lock();
            while (front == tail) {
                empty.wait(&lock);
            }
            int item = buf[front % MAX];
            front = (front + 1) % MAX;
            full.signal(&lock);
            lock.unlock();
            return item;
        }

        void put(int item) {
            lock.lock();
            while ((tail - front) == MAX) {
                full.wait(&lock);
            }
            buf[tail % MAX] = item;
            tail = (tail + 1) % MAX;
            empty.signal(&lock);
            lock.unlock();
        }
};
```
Pre/Post Conditions

• What is state of the bounded buffer at lock acquire?
  – front <= tail
  – tail – front <= MAX

• These are also true on return from wait

• And at lock release

• Allows for proof of correctness
Question

Does the kth call to get return the kth item put?

Hint: wait must re-acquire the lock after the signaller releases it.
methodThatWaits() {
    lock.acquire();
    // Pre-condition: State is consistent
    // Read/write shared state
    while (!testSharedState()) {
        cv.wait(&lock);
    }
    // WARNING: shared state may have changed! But
    // testSharedState is TRUE
    // and pre-condition is true
    // Read/write shared state
    lock.release();
}

methodThatSignals() {
    lock.acquire();
    // Pre-condition: State is consistent
    // Read/write shared state
    if (testSharedState() is now true) cv.signal(&lock);
    // NO WARNING: signal keeps lock
    // Read/write shared state
    lock.release();
}
Rules for Condition Variables

• ALWAYS hold lock when calling wait, signal, broadcast
  – Condition variable is sync FOR shared state
  – ALWAYS hold lock when accessing shared state

• Condition variable is memoryless
  – If signal when no one is waiting, no op
  – If wait before signal, waiter wakes up

• Wait atomically releases lock
  – What if wait, then release?
  – What if release, then wait?
Rules for Condition Variables, cont’d

• When a thread is woken up from wait, it may not run immediately
  – Signal/broadcast put thread on ready list
  – When lock is released, anyone might acquire it

• Wait MUST be in a loop
  while (needToWait()) {
    condition.Wait(&lock);
  }

• Simplifies implementation
  – Of condition variables and locks
  – Of code that uses condition variables and locks
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.
Structured Synchronization

• Identify objects or data structures that can be accessed by multiple threads concurrently
  – In kernel, everything!
• Add locks to object/module
  – Grab lock on start to every method/procedure
  – Release lock on finish
• If need to wait
  – while(needToWait()) { condition.Wait(lock); }
  – Do not assume when you wake up, signaller just ran
• If do something that might wake someone up
  – Signal or Broadcast
• Always leave shared state variables in a consistent state
  – When lock is released, or when waiting
Remember the rules

- Use consistent structure
- Always use locks and condition variables
- Always acquire lock at beginning of procedure, release at end
- Always hold lock when using a condition variable
- Always wait in while loop
- Never spin in sleep()
## Implementing Synchronization

Concurrent Applications

<table>
<thead>
<tr>
<th></th>
<th>Semaphores</th>
<th>Locks</th>
<th>Condition Variables</th>
</tr>
</thead>
</table>

| Interrupt Disable  | Atomic Read/Modify/Write Instructions |

|                    | Multiple Processors | Hardware Interrupts |

- Implementing Synchronization
- Concurrent Applications
- Semaphores
- Locks
- Condition Variables
- Interrupt Disable
- Atomic Read/Modify/Write Instructions
- Multiple Processors
- Hardware Interrupts
Implementing Synchronization (Take 1)

Use memory load/store instructions
  – See too much beer solution/Peterson’s algorithm
  – Complex
  – Need memory barriers
  – Hard to test/verify correctness
Implementing Synchronization (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
  – OS/161 on 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;
    (void)testAndClear(&lockValue); // membarrier
}
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
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
Readers/Writers Lock

ReaderAcquire()
  lock.Acquire();
  while (AW > 0) {
    WR++;
    okToRead.wait(&lock);
    WR--;
  }
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();

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

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

lock.Acquire();
while (AW > 0 || AR > 0) {
  WW++;
  okToRead.wait(&lock);
  WW--;
}
AW++;
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 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();
}
// check in
lock.Acquire();
myPos = numWriters++;
while ((AW + AR > 0 || myPos > nextToGo) {
    WW++;     // check out
    lock.Acquire();
    AW--;     // check out
    nextToGo++;
    if (WW > 0) {
        okToWrite.Signal(&lock);
    } else if (WR > 0)
        okToRead.Bcast(&lock);
    lock.Release();
}
WW--;
lock.Release();
Readers/Writers Lock w/o 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.Front();
    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 given back to signaller
  – Nested signals possible!
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
eempty/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++;
    self = new Condition;
    nextGet.append(self);
    while (front < myPosition || front == tail) {
        self.wait(&lock);
    }
    lock.release();
    return item;
}

delete self;
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
Semaphores

• Semaphore has a non-negative integer value
  – P() atomically waits for value to become > 0, then decrements
  – V() atomically increments value (waking up waiter if needed)

• Semaphores are like integers except:
  – Only operations are P and V
  – Operations are atomic
    • If value is 1, two P’s will result in value 0 and one waiter

• Semaphores are useful for
  – Unlocked wait/wakeup: interrupt handler, fork/join
Semaphore Implementation

Semaphore::P() {
  oldIPL = setInterrupts(OFF);
  spinLock.acquire();
  if (value == 0) {
    waiting.add(myTCB);
    suspend(&spinlock);
  } else {
    value--;
  }
  spinLock.release();
  setInterrupts(oldIPL);
}

Semaphore::V() {
  oldIPL = setInterrupts(OFF);
  spinLock.acquire();
  if (!waiting.Empty()) {
    next = waiting.remove();
    sched.makeReady(next);
  } else {
    value++;
  }
  spinLock.release();
  setInterrupts(oldIPL);
}
Semaphore Bounded Buffer

get() {
    fullSlots.P();
    mutex.P();
    item = buf[front % MAX];
    front++;
    mutex.V();
    emptySlots.V();
    return item;
}

put(item) {
    emptySlots.P();
    mutex.P();
    buf[last % MAX] = item;
    last++;
    mutex.V();
    fullSlots.V();
}

Initially: front = last = 0; MAX is buffer capacity
mutex = 1; emptySlots = MAX; fullSlots = 0;
Implementing Condition Variables using Semaphores (Take 1)

```c
wait(lock) {
    lock.release();
    semaphore.P();
    lock.acquire();
}

signal() {
    semaphore.V();
}
```
Implementing Condition Variables using Semaphores (Take 2)

wait(lock) {
    lock.release();
    semaphore.P();
    lock.acquire();
}

signal() {
    if (semaphore is not empty)
        semaphore.V();
}
Implementing Condition Variables using Semaphores (Take 3)

```java
wait(lock) {
    semaphore = new Semaphore;
    queue.Append(semaphore);  // queue of waiting threads
    lock.release();
    semaphore.P();
    lock.acquire();
}

signal() {
    if (!queue.Empty()) {
        semaphore = queue.Remove();
        semaphore.V();  // wake up waiter
    }
}
```
Communicating Sequential Processes (CSP/Google Go)

• Threads communicate through channels
  – Bounded buffer: put/get

• Good match for data flow processing
  – Producer/consumer

• No memory races!
CSP/Google Go

• What about general computation?
  – Is CSP as powerful as locks/condition variables?

• A thread per shared object
  – Only thread allowed to touch object’s data
  – To call a method on the object, send thread a message with method name, arguments
  – Thread waits in a loop, get msg, do operation
Bounded Buffer (CSP)

while (cmd = getNext()) {
  if (cmd == GET) {
    if (front < tail) {
      // do get
      // send reply
      // if pending put, do it
      // and send reply
    } else
      // queue get operation
  } else { // cmd == PUT
    if ((tail - front) < MAX) {
      // do put
      // send reply
      // if pending get, do it
      // and send reply
    } else
      // queue put operation
  }
}
Locks/CVs vs. CSP

• Create a lock on shared data
  = create a single thread to operate on data

• Call a method on a shared object
  = send a message/wait for reply

• Wait for a condition
  = queue an operation that can’t be completed just yet

• Signal a condition
  = perform a queued operation, now enabled
Remember the rules

• Use consistent structure
• Always use locks and condition variables
• Always acquire lock at beginning of procedure, release at end
• Always hold lock when using a condition variable
• Always wait in while loop
• Never spin in sleep()