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 Milk Example

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

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 Milk, Try #1

- Correctness property
  - Someone buys if needed (liveness)
  - At most one person buys (safety)
- Try #1: leave a note

```
if (!note)
    if (!milk) {
        leave note
        buy milk
        remove note
    }
```

Too Much Milk, Try #2

Thread A

```
leave note A
if (!note B) {
    if (!milk)
        buy milk
}
remove note A
```

Thread B

```
leave note B
if (!noteA) {
    if (!milk)
        buy milk
}
remove note B
```

Too Much Milk, Try #3

Thread A	Thread B
leave note A	leave note B
while (note B) // X	if (!noteA) { // Y
do nothing;	if (!milk)
if (!milk)	buy milk
buy milk;	}
remove note A	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

Semaphores | Locks | Condition Variables

Interrupt Disable | Atomic Read/Modify/Write Instructions

Multiple Processors | Hardware Interrupts

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 Milk, #4

Locks allow concurrent code to be much simpler:

```
lock.acquire();
if (!milk)
    buy milk
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

Condition Variable Design Pattern

```
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

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

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

Initially: front = tail = 0; MAX is buffer capacity
empty/full are condition variables

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.

Pre/Post Conditions

```
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

Java Manual

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 OS/161 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

Semaphores　　Locks　　Condition Variables

Interrupt Disable　　Atomic Read/Modify/Write Instructions

Multiple Processors　　Hardware Interrupts

Implementing Synchronization

Take 1: using memory load/store

– See too much milk solution/Peterson's algorithm

Take 2:

```
Lock::acquire()
  { oldIPL = setInterrupts(OFF); }
Lock::release()
  { setInterrupts(oldIPL); }
```

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;
   }
   setInterrupts(oldIPL);
}
```

```
Lock::release() {
   oldIPL = setInterrupts(OFF);
   if (!waiting.Empty()) {
      next = waiting.remove();
      next->state = READY;
      readyList.add(next);
   } else {
      value = FREE;
   }
   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
- Examples
  - Test and set
  - Intel: xchgb, lock prefix
  - Compare and swap
- Any of these can be used for implementing locks and condition variables!

Spinlocks

A spinlock is a lock where the processor 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

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

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);
    } else {
        value = BUSY;
    }
    spinLock.release();
}
```

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

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);
}
```

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

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

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

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

nextGet, nextPut are queues of Condition Variables

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)

```
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)

```
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)

- 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
- No memory races!

Example: Bounded Buffer

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

```
put(item) {
    lock.acquire();
    while ((tail – front) == MAX) {
        full.wait(lock);
    }
    buf[tail % MAX] = item;
    tail++;
    empty.signal(lock);
    lock.release();
}
```

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

empty/full are condition variables

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()