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 = someFn(p);
    if (q != someFn(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

Reordering 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

<table>
<thead>
<tr>
<th>Time</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:35</td>
<td>Leave for store.</td>
<td></td>
</tr>
<tr>
<td>12:45</td>
<td>Buy milk.</td>
<td>Leave for store.</td>
</tr>
<tr>
<td>12:50</td>
<td>Arrive home, put milk away.</td>
<td>Arrive at store.</td>
</tr>
<tr>
<td>12:55</td>
<td></td>
<td>Buy milk.</td>
</tr>
<tr>
<td>1:00</td>
<td></td>
<td>Arrive home, put milk away. Oh no!</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 synch 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

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

Thread B

leave note B
if (!noteA){ // Y
   if (!milk)
      buy milk
}
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
  – Peterson’s algorithm: even more complex
# Roadmap

**Concurrent Applications**

**Shared Objects**

<table>
<thead>
<tr>
<th>Bounded Buffer</th>
<th>Barber Chair</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Synchronization Objects**

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

**Atomic Instructions**

<table>
<thead>
<tr>
<th>Interrupt Disable</th>
<th>Test-and-Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Hardware Reality**

<table>
<thead>
<tr>
<th>Multiple Processors</th>
<th>Hardware Interrupts</th>
</tr>
</thead>
<tbody>
<tr>
<td></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)
Too Much Milk, #4

Locks allow concurrent code to be much simpler:

```python
lock.acquire()
if (!milk) buy milk
lock.release()
```

- How do we implement locks? (Later)
  - Hardware support for read/modify/write instructions
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!
  – DO NOT throw lock for someone else to release
• Never access shared data without lock
  – Danger!
Will this code work?

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

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

tryget() {
    item = NULL;
    lock.acquire();
    if (front < tail) {
        item = buf[front % MAX]
        front++;
    }
    lock.release();
    return item;
}
Initially: front = tail = 0; lock = FREE; MAX is buffer capacity

tryput(item) {
    lock.acquire();
    if ((tail - front) < size) {
        buf[tail % MAX] = item;
        tail++;
    }
    lock.release();
}
Question

• If `tryget` returns `NULL`, do we know the buffer is empty?
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 change shared state so that testSharedState is 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 % size] = 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
  – front + MAX >= tail

• These are also true on return from wait

• And at lock release

• Allows for proof of correctness
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?
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 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
Mesa vs. Hoare semantics

• Mesa (Hansen = 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);
    lock.release();
    return item = buf[front % MAX];
}

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

Concurrent Applications

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


Implementing Synchronization

Take 1: using memory load/store
  – See too much milk solution/Peterson’s algorithm

Take 2:
  Lock::acquire() { disable interrupts }
  Lock::release() { enable interrupts }
Lock Implementation, Uniprocessor

Lock::acquire()
{
    disableInterrupts();
    if(value == BUSY){
        waiting.add(current TCB);
        suspend();
    } else {
        value = BUSY;
    }
    enableInterrupts();
}

Lock::release()
{
    disableInterrupts();
    if (!waiting.Empty()){  
        thread = waiting.remove();
        readyList.append(thread);
    } else {
        value = FREE;
    }
    enableInterrupts();
}
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

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 ready list to implement locks

Spinlock::acquire() {
   while (testAndSet(&lockValue) == BUSY) {
      ;
   }
}
Spinlock::release() {
   lockValue = FREE;
   memorybarrier();
}
Lock Implementation, Multiprocessor

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

Lock::release() {
   disableInterrupts();
   spinLock.acquire();
   if (!waiting.Empty()) {
      thread = waiting.remove();
      readyList.append(thread);
   } else {
      value = FREE;
   }
   spinLock.release();
   enableInterrupts();
}
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 the lock
• Slow path
  – If lock is BUSY or someone is waiting, see previous slide

• User-level locks
  – Fast path: acquire lock using test&set
  – Slow path: system call to kernel, use kernel lock
Lock Implementation, Linux

```c
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:
```
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: interrupt handler, fork/join
Semaphore Bounded Buffer

get() {
    empty.P();
    mutex.P();
    item = buf[front % size]
    front++;
    mutex.V();
    full.V();
    return item;
}

put(item) {
    full.P();
    mutex.P();
    buf[last % size] = item;
    last++;
    mutex.V();
    empty.V();
}

Initially: front = last = 0; size is buffer capacity
empty/full are semaphores
Implementing Condition Variables using Semaphores (Take 1)

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

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

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

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

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

signal() {
  if !queue.Empty()
    sem = queue.Remove();
    sem.V();  // wake up waiter
}

Synchronization Summary

• 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()