Synchronization
Synchronization Motivation

• When threads concurrently read/write shared memory, program behavior is undefined
  – Two threads write to same variable; which one wins?
• Thread schedule is non-deterministic
  – Behavior changes from run to run
• Compiler/hardware instruction reordering
• Multi-word operations are not atomic
Synchronization Motivation

Thread 1

\[ p = \text{someFn}(); \]
\[ \text{isInitialized} = \text{true}; \]

Thread 2

\[ \text{while (! isInitialized )} \]
\[ ; \]
\[ q = \text{aFn}(p); \]

\[ \text{if } q \neq \text{aFn(someFn())} \]
\[ \text{panic} \]
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 ops after begin
## 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**

```java
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
  – Even more complex: see Peterson’s algorithm
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 ask if the lock is free. Suppose it returns true. Is the lock:
  - free?
  - busy?
  - don’t know?
Locks allow concurrent code to be much simpler:

```
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!
  – Only the lock holder can release
  – 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(lock);
    if (p == NULL) {
        p = newP();
    }
    release_lock(lock);
}
use p->field1
```

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

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

Initially: front = last = 0; lock = FREE; size is buffer capacity

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

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

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

• For waiting inside a critical section
  – Called only when holding a lock

• Wait: atomically release lock and relinquish processor
  – Reacquire lock and continue executing when signalled

• 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);

    lock.release();
}
Example: Bounded Buffer

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

put(item) {
    lock.acquire();
    while ((last - front) == size)
        full.wait(lock);
    buf[last % size] = item;
    last++;
    empty.signal(lock);
    lock.release();
}

Initially: front = last = 0; size is buffer capacity
empty/full are condition variables
Pre/Post Conditions

• What is state of the bounded buffer at lock acquire?
  – front <= last
  – front + buffer size >= last

• These are also true on return from wait

• Also true 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
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()
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 == last)  
        empty.wait(lock);  
    item = buf[front % size];  
    front++;  
    full.signal(lock);  
    lock.release();  
    return item;  
}

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

Initially: front = last = 0; size 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();
    if (front == last) {
        self = new Condition;
        nextGet.Append(self);
        while (front == last)
            self.wait(lock);
        nextGet.Remove(self);
        delete self;
    }
    item = buf[front % size]
    front++;
    if (!nextPut.empty())
        nextPut.first()->signal(lock);
    lock.release();
    return item;
}

Initially: front = last = 0; size is buffer capacity
nextGet, nextPut are queues of Condition Variables
Implementing Synchronization

<table>
<thead>
<tr>
<th>Concurrent Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Semaphores</strong></td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td><strong>Interrupt Disable</strong></td>
</tr>
<tr>
<td><strong>Multiple Processors</strong></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

LockAcquire()
{
    disableInterrupts();
    if(value == BUSY){
        waiting.add(myTCB);
        myTCB->state = WAITING;
        next = readyList.remove();
        switch(myTCB,next);
        myTCB->state = RUNNING;
    } else {
        value = BUSY;
    }
    enableInterrupts();
}

LockRelease()
{
    disableInterrupts();
    if (!waiting.Empty()){  
        next = waiting.Remove();
        next->state = READY;
        readyList.add(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

• Does it matter which type of RMW instruction we use?
  – Not for implementing locks and condition variables!
Spinlocks

Locks 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 and to implement locks

SpinlockAcquire() {
    while (testAndSet(&lockValue) == BUSY)
    ;
}

SpinlockRelease() {
    lockValue = FREE;
    memorybarrier();
}
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...

• Instead:
  – One spinlock per lock
  – One spinlock for the scheduler ready list
    • Per-core ready list: one spinlock per core
Lock Implementation, Multiprocessor

```
LockAcquire()
  spinLock.Acquire();
  disableInterrupts();
  if(value == BUSY){
    waiting.add(current TCB);
    suspend();
  } else {
    value = BUSY;
  }
  enableInterrupts();
  spinLock.Release();
}

LockRelease() {
  spinLock.Acquire();
  disableInterrupts();
  if (!waiting.Empty()){
    thread = waiting.Remove();
    readyList.Append(thread);
  } else {
    value = FREE;
  }
  enableInterrupts();
  spinLock.Release();
}
```
What thread is currently running

• Thread scheduler needs to find TCB of the currently running thread
  – To suspend and switch to new thread
  – To check if the current thread holds a lock before acquiring or releasing it

• On a uniprocessor: just use a global

• On a multiprocessor: various methods:
  – Compiler dedicates a register
  – Hardware may have special per-processor register
  – Fixed size stacks: put a pointer to the TCB at the bottom of the stack
    • Find by masking current stack pointer
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() ) {
        next = waiting.remove();
        scheduler->makeReady(next);
    } else {
        value = FREE;
    }
    spinLock.release();
    enableInterrupts();
}
Lock Implementation, Linux

• Most locks are free most of the time
  – Why?
    – Linux implementation takes advantage of this property

• Fast path
  – If lock is FREE, and no one is waiting, test&set

• Slow path
  – If lock is BUSY or someone is waiting, see multiproc implementation

• User-level locks
  – Fast path: acquire lock using test&set
  – Slow path: system call to kernel to use kernel lock
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()
{
    fullSlots.P();
    mutex.P();
    item = buf[front % size]
    front++;
    mutex.V();
    emptySlots.V();
    return item;
}

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

Initially: front = last = 0; size is buffer capacity
mutex = 1; emptySlots = size; fullSlots = 0
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)

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

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

```java
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
        sem.V();  // wake up waiter
    }
```
Communicating Sequential Processes (CSP/Google Go)

• A thread per shared object
  – Only that thread is allowed to touch object’s data
  – To call a method on the object, send thread a message (with method name and args)
  – Thread waits in a loop: get msg; do operation

• No user-code memory races!
Lock example: Bounded Buffer

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

tryput(item) {
    lock.acquire();
    if ((last – front) < size) {
        buf[last % size] = item;
        last++;
    }
    lock.release();
}

Initially: front = last = 0; lock = FREE; size is buffer capacity
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/Condition Vars 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 and 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
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()