Main Points

- Challenges
  - Why does execution depend on interleaving of threads?

- Locks and condition variables
  - How can we enforce a logical sequence of operations?

- Designing shared objects
  - What is a good way of writing multithreaded programs?
Synchronization Motivation

- When threads concurrently read/write shared memory, program behavior is undefined
  - Two threads write to a variable; which one should win?
- Thread schedule is non-deterministic
  - Behavior changes over different runs of the same program
- Compiler and hardware reorder instructions
**Question: Can This Panic?**

// Thread 1

```c
p = someComputation();
pInitialized = true;
```

// Thread 2

```c
While (!pInitialized);
    q = someFunc(p);
    If (q != someFunc(p))
        panic();
```
Why Reordering?

• Why do compilers reorder instructions?
  • Generating efficient code needs ctrl/data dependency analysis
  • 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

<table>
<thead>
<tr>
<th>Time</th>
<th>Roommate A</th>
<th>Roommate B</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:35</td>
<td>Leave for store.</td>
<td>Leave for store.</td>
</tr>
<tr>
<td>12:40</td>
<td>Arrive at store.</td>
<td>Arrive at store.</td>
</tr>
<tr>
<td>12:45</td>
<td>Buy milk.</td>
<td>Buy milk.</td>
</tr>
<tr>
<td>12:50</td>
<td>Arrive home, put milk away.</td>
<td>Arrive home, put milk away. Oh no!</td>
</tr>
<tr>
<td>12:55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01:00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Definitions

• **Race condition**: output of a concurrent program depends on 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
Definitions (cont.)

- **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!)

- **Safety**: program never enters a bad state
  - Never more than one person buys milk

- **Liveness**: program eventually enters a good state
  - Someone eventually buys milk if needed
if (!milk) {
    if (!note) {
        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 (!note A) {
    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 (!note A) {
    // Y
    if (!milk)
        buy milk;
}
remove note B;
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

### Additional Concepts

- Interrupt Disable
- Atomic Read/Modify/Write Instructions

### Advanced Topics

- 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
- At most one lock holder at a time (safety)
- If no one holding, acquire gets lock (progress)
- If all lock holders finish and no higher priority waiters, waiter eventually gets lock (bounded waiting)
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 (Try #4)

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

```c
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
  • 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! Don’t do it even if it’s tempting!
Double Checked Locking

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

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

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

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?
  • Hint: NO 😊

• If we poll `tryget()` in a loop, what happens to a thread calling `tryput()`?
  • Hint: delay!
Condition Variables

- CV::wait(Lock *lock)
  - Atomically release lock and relinquish processor
  - Reacquire the lock when wakened

- CV::signal()
  - Wake up a waiter, if any

- CV::broadcast()
  - Wake up all waiters, if any
Properties of Condition Variables

• CV is memoryless
  • No internal memory except a queue of waiting threads
  • No effect in calling signal/broadcast on empty queue

• wait atomically releases the lock
  • No separation between adding the thread to the waiting queue and releasing the lock

• Re-enabled waiting thread may not run immediately
  • No atomicity between signal/broadcast and the return from wait
Condition Variable Design Pattern

```java
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();

    // 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.release();
    return item;
}

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

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

• Does the $k^{th}$ call to get return the $k^{th}$ item put?
  • Hint: wait must re-acquire the lock after the signaller releases it.
Pre/Post Conditions

- What is state of the bounded buffer at lock acquire?
  - front \leq tail
  - front + MAX \geq tail
- These are also true on return from wait
- And at lock release
- Allows for proof of correctness
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();
    // NO WARNING: signal keeps lock
    // Read/write shared state
    lock.release();
}
Rules for Condition Variables

• **ALWAYS** hold lock when calling `wait`, `signal`, `broadcast`

• **CV::wait MUST** be called in a loop

```cpp
while (needToWait()) {
    condition.wait(&lock);
}
```

And not:

```cpp
if (needToWait()) {
    condition.wait(&lock);
}
```
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

• Add locks to object/module
  • Grab lock on start to every method/procedure
  • Release lock on finish

• If you need to wait
  • Use `while(needToWait()) { condition.wait(lock); }`
  • Do not assume when you wake up, signaler just ran

• If you 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 the beginning of procedure, release at the end
• Always hold lock when using a condition variable
• Always wait in while loop
• Never spin in `sleep()`
Roadmap

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()
  { disable interrupts }
  Lock::release()
  { enable interrupts }
Lock Implementation (Uniprocessor)

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

Lock::release() {
    disableInterrupts();
    if (!waiting.Empty()) {
        next = waiting.remove();
        next->state = READY;
        readyList.add(next);
    } else {
        value = FREE;
    }
    enableInterrupts();
}
What Thread is Currently Running?

- Scheduler needs to know TCB of running thread
  - To suspend and switch to a new thread
  - To check if 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)

- 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 are prevented in hardware

• Examples
  • test_and_set
  • xchgb
  • compare_and_swap

• Any of these can be used for implementing locks and condition variables!
Spinlocks

Spinlock::acquire() {
    while (test_and_set(&lockValue) == BUSY);
}

Spinlock::release() {
    lockValue = FREE;
    memorybarrier();
}
Spinlocks (cont.)

- A spinlock is a lock where the processor waits in a loop for the lock to become free.
- Spinlock has low overhead if lock is held briefly:
  - I.e., less time than a context switch would take.
  - E.g., CPU scheduler for its ready list.
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?

• One spinlock per kernel?
  • Bottleneck!

• One spinlock per lock
  • Per-lock waiting list

• One spinlock for the scheduler ready list
  • Per-core ready list: one spinlock per core
  • Scheduler lock requires interrupts off!
**Lock Implementation, Multiprocessor**

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

```cpp
Lock::release() {
    disableInterrupts();
    spinLock.acquire();
    if (!waiting.Empty()) {
        next = waiting.remove();
        scheduler.makeReady(next);
    } else {
        value = FREE;
    }
    spinLock.release();
    enableInterrupts();
}
```
Sched::suspend(SpinLock *lock) {
    TCB *next;
    oldIPL = setInterrupts(OFF);
    schedSpinLock.acquire();
    spinLock->release();
    myTCB->state = WAITING;
    next = readyList.remove();
    thread_switch(myTCB, next);
    myTCB->state = RUNNING;
    schedSpinLock.release();
    setInterrupts(oldIPL);
}

Sched::makeReady(TCB *thread) {
    oldIPL = setInterrupts(OFF);
    schedSpinLock.acquire();
    readyList.add(thread);
    thread->state = READY;
    schedSpinLock.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 lock
  - If no one is waiting, two instructions to release the lock
- Slow path
  - If lock is BUSY or someone is waiting (see multiproc)
- Two versions: one with interrupts off, one w/o
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

• Recall two ways of application-level concurrency
  • System calls or user-level thread scheduler

• Implement system calls for every lock acquire/release
  • Context switch in the kernel

• Or split implementation into fast and slow path
  • Count field at user level
  • Spinlock and wait list queue at kernel

• Or implement locks in user-level thread library
  • Thread context switch at user level
  • Disable upcalls instead of interrupts
FIFO Bounded Buffer

- 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 (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.release();
    return item;
}

Initially: \textbf{front} = \textbf{tail} = \textbf{numGets} = 0; \textbf{MAX} is buffer capacity \textbf{nextGet} and \textbf{nextPut} are queues of Condition Variables
Semaphores

- **Semaphore** has a non-negative integer value
  - Initial value could be any non-negative integer
  - \( P() \) atomically waits for value to become > 0, then decrements it
  - \( V() \) atomically increments value, then wakes up a waiter if there are any
- **Semaphores** are like integers except
  - Only operations are \( P() \) and \( V() \)
    - Value cannot be directly read
  - Operations are **atomic**
    - If value is 1, two \( P() \)'s will result in value 0 and one waiter
Semaphore (cont.)

• Similar to locks for mutual exclusion
  • Initialize semaphore to 1
  • P is similar to wait and V is similar to release

• Similar to condition variables for general waiting
  • Initialize semaphore to 0
  • P is similar to thread_join and V is similar to thread_exit

• Semaphore has state (its value) whereas locks and condition variables do not have any state
  • cv.signal() without a waiting thread does nothing but V() increments the value
“During system conception it transpired that we used the semaphores in **two completely different ways**. The difference is so marked that, looking back, one wonders whether it was really fair to present the two ways as uses of the very same primitives. On the one hand, we have the semaphores used for **mutual exclusion**, on the other hand, the **private semaphores**.”

From Dijkstra “The structure of the 'THE'-Multiprogramming System” Communications of the ACM v. 11 n. 5 May 1968.)”
Semaphores in Interrupt Handlers

- Kernel and hardware device read/write to shared in-memory data structure
- Hardware updates shared data structure and starts an interrupt handler
- Interrupt handler usually wakes up a waiting thread and returns
- Interrupt handler could us CV and call signal without holding lock (naked notify)
  - If thread hasn’t called wait, then signal gets waisted!
- Common solution is to use semaphores instead
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)

```
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
    }
}
```
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 CV for waiting if readers/ writers are in progress
  - Some state variables
Class RWLock {
    Lock lock;
    CV okToRead;
    CV okToWrite;
    int AW = 0;       //active writers
    int AR = 0;       //active readers
    int WW = 0;       //waiting writers
    int WR = 0;       //waiting readers
    void startRead();
    void doneRead();
    void startWrite();
    void doneWrite();
}
void startRead() {
    lock.acquire();
    while (AW > 0 || WW > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.release();
}

void doneRead() {
    lock.acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.wait(&lock);
    lock.release();
}

void startWrite() {
    lock.acquire();
    while (AW > 0 || AR > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
    lock.release();
}

void doneWrite() {
    lock.acquire();
    AW--;
    if (WW > 0) {
        okToWrite.signal();
    } else if (WR > 0) {
        okToRead.broadcast();
    }
    lock.release();
}
Questions

• 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
```cpp
void startWrite() {
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();
}

void doneWrite() {
lock.acquire();
AW--;
nextToGo++;
if (WW > 0) {
    cv = writers.removeFront();
    cv.signal();
} else if (WR > 0) {
    okToRead.broadcast();
}
lock.release();
}
```
Mesa vs. Hoare Semantics

- Mesa
  - Signal puts waiter on ready list
  - Signaler keeps lock and processor
- Hoare
  - Signal gives processor and lock to waiter
  - When waiter finishes, processor/lock given back to signaler
  - 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.release();
    return item;
}

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

Initially: front = tail = 0; MAX is buffer capacity
empty and full are condition variables
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, gets messages, does operation

- No memory races!
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 {
            // queue put 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()
Acknowledgment

• This lecture is a slightly modified version of the one prepared by Tom Anderson