Review: Event Programming – Analogy

- Once upon a time (before cell-phones)
- Ann calls Bank of America to verify her account...
- ... she is put on hold...
- ... Mary, her roommate, waits for an important phone call

- Is there a better solution?
- Yes! Ann can ask the clerk to call her back ;-) 

- Event programming:
  - Non-blocking I/O call: returns immediately after I/O call
  - Program periodically checks or interrupted when I/O completes

Review: Another Concurrent Program Example

- Two threads, A and B, compete with each other
  - One tries to increment a shared counter
  - The other tries to decrement the counter

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 0;</td>
<td>i = 0;</td>
</tr>
<tr>
<td>while (i &lt; 10)</td>
<td>while (i &gt; -10)</td>
</tr>
<tr>
<td>i = i + 1;</td>
<td>i = i - 1;</td>
</tr>
<tr>
<td>printf(&quot;A wins!&quot;);</td>
<td>printf(&quot;B wins!&quot;);</td>
</tr>
</tbody>
</table>

- Assume that memory loads and stores are atomic, but incrementing and decrementing are not atomic
- Who wins?
- Is it guaranteed that someone wins? Why or why not?
- What if both threads have their own CPU running at same speed? Is it guaranteed that it goes on forever?

Goals for Today

- Synchronization
- Hardware Support for Synchronization
- Higher-level Synchronization Abstractions
  - Semaphores

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated by Kubiatowicz.
Motivation: “Too much milk”
• Great thing about OS’s – analogy between problems in OS and problems in real life
  – Help you understand real life problems better
  – But, computers are much stupider than people
• Example: People need to coordinate:

<table>
<thead>
<tr>
<th>Time</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Look in Fridge. Out of milk</td>
<td></td>
</tr>
<tr>
<td>3:05</td>
<td>Leave for store</td>
<td></td>
</tr>
<tr>
<td>3:10</td>
<td>Arrive at store</td>
<td>Look in Fridge. Out of milk</td>
</tr>
<tr>
<td>3:15</td>
<td>Buy milk</td>
<td>Leave for store</td>
</tr>
<tr>
<td>3:20</td>
<td>Arrive home, put milk away</td>
<td>Arrive at store</td>
</tr>
<tr>
<td>3:25</td>
<td>Buy milk</td>
<td></td>
</tr>
<tr>
<td>3:30</td>
<td>Arrive home, put milk away</td>
<td></td>
</tr>
</tbody>
</table>

Definitions
• Synchronization: using atomic operations to ensure cooperation between threads
  – For now, only loads and stores are atomic
  – We’ll show it’s hard to build anything useful with only reads and writes
• Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  – One thread excludes the other while doing its task
• Critical Section: piece of code that only one thread can execute at once
  – Critical section is the result of mutual exclusion
  – Critical section and mutual exclusion are two ways of describing the same thing

More Definitions
• Lock: prevents someone from doing something
  – Lock before entering critical section and before accessing shared data
  – Unlock when leaving, after accessing shared data
  – Wait if locked
  » Important idea: all synchronization involves waiting
• For example: fix the milk problem by putting a key on the refrigerator
  – Lock it and take key if you are going to go buy milk
  – Fixes too much: roommate angry if only wants orange juice
  – Of Course – We don’t know how to make a lock yet

Too Much Milk: Correctness Properties
• Need to be careful about correctness of concurrent programs, since non-deterministic
  – Always write down behavior first
  – Impulse is to start coding first, then when it doesn’t work, pull hair out
  – Instead, think first, then code
• What are the correctness properties for the “Too much milk” problem?
  – Never more than one person buys
  – Someone buys if needed
• Restrict ourselves to use only atomic load and store operations as building blocks
Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if note (wait)

- Suppose a computer tries this (remember, only memory read/write are atomic):

```c
if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
    remove note;
  }
}
```

- Result?

Too Much Milk: Solution #1 ½

- Clearly the Note is not quite blocking enough
  - Let’s try to fix this by placing note first
- Another try at previous solution:

```c
leave Note;
if (noMilk) {
  if (noNote) {
    buy milk;
  }
}
remove note;
```

- What happens here?
  - Well, with human, probably nothing bad
  - With computer: no one ever buys milk

Too Much Milk Solution #2

- How about labeled notes?
  - Now we can leave note before checking

- Algorithm looks like this:

```c
Thread A
leave note A;
if (noMilk) {
  if (noNote B) {
    buy Milk;
  }
}
remove note A;

Thread B
leave note B;
if (noMilk) {
  if (noNote A) {
    buy Milk;
  }
}
remove note B;
```

- Does this work?
### Too Much Milk Solution #2

- Possible for neither thread to buy milk!

  **Thread A**
  ```java
  leave note A;
  if (noNote A) {
    if (noMilk) {
      buy Milk;
    }
  }
  if (noNote B) {
    if (noMilk) {
      buy Milk;
      ...
    }
  }
  remove note B;
  ```

  **Thread B**
  ```java
  leave note B;
  if (noNote A) {
    if (noMilk) {
      buy Milk;
    }
  }
  ```

- Really insidious:
  - Unlikely that this would happen, but will at worse possible time

### Review: Too Much Milk Solution #3

- Here is a possible two-note solution:

  **Thread A**
  ```java
  leave note A;
  while (note B) \X
  do nothing;
  if (noNote A) \Y
  if (noMilk) {
    buy milk;
  }
  if (noMilk) {
    buy milk;
  }
  remove note A;
  remove note B;
  ```

  **Thread B**
  ```java
  leave note B;
  if (noNote A) {
    if (noMilk) {
      buy Milk;
    }
  }
  ```

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit

  - At X:
    - if no note B, safe for A to buy,
    - otherwise wait to find out what will happen

  - At Y:
    - if no note A, safe for B to buy
    - Otherwise, A is either buying or waiting for B to quit

### Solution #3 discussion

- Our solution protects a single "Critical-Section" piece of code for each thread:
  ```java
  if (noMilk) {
    buy milk;
  }
  ```

- Solution #3 works, but it’s really unsatisfactory
  - Really complex – even for this simple an example
    - Hard to convince yourself that this really works
  - A’s code is different from B’s – what if lots of threads?
  - Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - This is called “busy-waiting”

- There’s a better way
  - Have hardware provide better (higher-level) primitives than atomic load and store
  - Build even higher-level programming abstractions on this new hardware support
High-Level Picture

- The abstraction of threads is good:
  - Maintains sequential execution model
  - Allows simple parallelism to overlap I/O and computation
- Unfortunately, still too complicated to access state shared between threads
  - Consider “too much milk” example
    - Implementing a concurrent program with only loads and stores would be tricky and error-prone
- Today, we’ll start implementing higher-level operations on top of atomic operations provided by hardware
  - Develop a “synchronization toolbox”
  - Explore some common programming paradigms

Too Much Milk: Solution #4

- Suppose we have some sort of implementation of a lock (more in a moment).
  - Lock.Acquire() – wait until lock is free, then grab
  - Lock.Release() – unlock, waking up anyone waiting
- These must be atomic operations – if two threads are waiting for the lock and both see it’s free, only one succeeds to grab the lock

- Then, our milk problem is easy:
  - milklock.Acquire();
  - if (nomilk)
    - buy milk;
  - milklock.Release();

- Once again, section of code between Acquire() and Release() called a "Critical Section"

How to implement Locks?

- Lock: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - Important idea: all synchronization involves waiting
    - Should sleep if waiting for a long time
- Atomic Load/Store: get solution like Milk #3
  - Pretty complex and error prone
- Hardware Lock instruction
  - Is this a good idea?
  - What about putting a task to sleep?
    - How do you handle the interface between the hardware and scheduler?
  - Complexity?
    - Each feature makes hardware more complex and slow

Naïve use of Interrupt Enable/Disable

- How can we build multi-instruction atomic operations?
  - Recall: dispatcher gets control in two ways.
    - Internal: Thread does something to relinquish the CPU
    - External: Interrupts cause dispatcher to take CPU
- On a uniprocessor, can avoid context-switching by:
  - Avoiding internal events (although virtual memory tricky)
  - Preventing external events by disabling interrupts
- Consequently, naïve Implementation of locks:
  - LockAcquire { disable Ints; }
  - LockRelease { enable Ints; }

- Problems with this approach:
  - Can’t let user do this! Consider following:
    - LockAcquire();
    - While(TRUE) {};
  - Real-Time system—no guarantees on timing!
    - Critical Sections might be arbitrarily long
  - What happens with I/O or other important events?
    - "Reactor about to meltdown. Help?"
**Better Implementation of Locks by Disabling Interrupts**

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```c
int value = FREE;

Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
        // Enable interrupts?
    } else {
        value = BUSY;
        enable interrupts;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if (anyone on wait queue) {
        take thread off wait queue
        Place on ready queue;
    } else {
        value = FREE;
        enable interrupts;
    }
    enable interrupts;
}
```

**New Lock Implementation: Discussion**

- Why do we need to disable interrupts at all?
  - Avoid interruption between checking and setting lock value
  - Otherwise two threads could think that they both have lock

```c
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
        // Enable interrupts?
    } else {
        value = BUSY;
        enable interrupts;
    }
    enable interrupts;
}
```

- Note: unlike previous solution, the critical section (inside `Acquire()`) is very short
  - User of lock can take as long as they like in their own critical section: doesn’t impact global machine behavior
  - Critical interrupts taken in time

**Interrupt re-enable in going to sleep**

- What about re-enabling ints when going to sleep?
  - Before putting thread on the wait queue?
    - Release can check the queue and not wake up thread
  - After putting the thread on the wait queue
    - Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
    - Misses wakeup and still holds lock (deadlock!)
  - Want to put it after `sleep()`. But — how?
How to Re-enable After Sleep()?

- Since ints are disabled when you call sleep:
  - Responsibility of the next thread to re-enable ints
  - When the sleeping thread wakes up, returns to acquire and re-enables interrupts

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>disable int</td>
<td>sleep</td>
</tr>
<tr>
<td>context switch</td>
<td></td>
</tr>
<tr>
<td>sleep return</td>
<td></td>
</tr>
<tr>
<td>enable int</td>
<td></td>
</tr>
<tr>
<td>context switch</td>
<td></td>
</tr>
<tr>
<td>sleep return</td>
<td></td>
</tr>
</tbody>
</table>

Atomic Read-Modify-Write instructions

- Problems with previous solution:
  - Can’t give lock implementation to users
  - Doesn’t work well on multiprocessor
    - Disabling interrupts on all processors requires messages and would be very time consuming
- Alternative: atomic instruction sequences
  - These instructions read a value from memory and write a new value atomically
  - Hardware is responsible for implementing this correctly
    - on both uniprocessors (not too hard)
    - and multiprocessors (requires help from cache coherence protocol)
  - Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors

Examples of Read-Modify-Write

- `test&set (address) { /* most architectures */
  result = M[address];
  M[address] = 1;
  return result;
}

- `swap (address, register) { /* x86 */
  temp = M[address];
  M[address] = register;
  register = temp;
}

- `compare&swap (address, reg1, reg2) { /* 68000 */
  if (reg1 == M[address]) {
    M[address] = reg2;
    return success;
  } else {
    return failure;
  }
}

Implementing Locks with test&set

- Simple solution:
  ```
  int value = 0; // Free
  Acquire() {
    while (test&set(value));
  }
  Release() {
    value = 0;
  }
  ```

  - Simple explanation:
    - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits.
    - If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues
    - When we set value = 0, someone else can get lock
Problem: Busy-Waiting for Lock

- Positives for this solution
  - Machine can receive interrupts
  - User code can use this lock
  - Works on a multiprocessor

- Negatives
  - This is very inefficient because the busy-waiting thread will consume cycles waiting
  - Waiting thread may take cycles away from thread holding lock (no one wins!)
  - Priority Inversion: If busy-waiting thread has higher priority than thread holding lock ⇒ no progress!
- Priority Inversion problem with original Martian rover
- For semaphores and monitors, waiting thread may wait for an arbitrary length of time!
  - Thus even if busy-waiting was OK for locks, definitely not ok for other primitives
  - Homework/exam solutions should not have busy-waiting!

Better Locks using test&set

- Can we build test&set locks without busy-waiting?
  - Can’t entirely, but can minimize!
  - Idea: only busy-wait to atomically check lock value

```c
int guard = 0;
int value = FREE;

Acquire() {
    // Short busy-wait time
    while (test&set(guard));
    if (value == BUSY) {
        put thread on wait queue;
        go to sleep() & guard = 0;
    } else {
        value = BUSY;
        guard = 0;
    }
}
```

```c
Release() {
    // Short busy-wait time
    while (test&set(guard));
    if anyone on wait queue {
        take thread off wait queue
        Place on ready queue;
    } else {
        value = FREE;
    }
}
```

- Note: sleep has to be sure to reset the guard variable
  - Why can’t we do it just before or just after the sleep?

Better Locks using test&set

- Compare to “disable interrupt” solution

```c
int value = FREE;

Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
        // Enable interrupts?
    } else {
        value = BUSY;
    }
} else {
    enable interrupts;
}
```

Higher-level Primitives than Locks

- Synchronization is a way of coordinating multiple concurrent activities that are using shared state
  - This lecture and the next presents a couple of ways of structuring the sharing
  - What is the right abstraction for synchronizing threads that share memory?
  - Want as high a level primitive as possible

- Good primitives and practices important!
  - Since execution is not entirely sequential, really hard to find bugs, since they happen rarely
  - UNIX is pretty stable now, but up until about mid-80s (10 years after started), systems running UNIX would crash every week or so – concurrency bugs
Semaphores

• Semaphores are a kind of generalized locks
  – First defined by Dijkstra in late 60s
  – Main synchronization primitive used in original UNIX

• Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  – P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    » Think of this as the wait() operation
  – V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    » This of this as the signal() operation
  – Note that P() stands for “proberen” (to test) and V() stands for “verhogen” (to increment) in Dutch

Semaphores Like Integers Except

• Semaphores are like integers, except
  – No negative values
  – Only operations allowed are P and V – can’t read or write value, except to set it initially
  – Operations must be atomic
    » Two P’s together can’t decrement value below zero
    » Similarly, thread going to sleep in P won’t miss wakeup from V – even if they both happen at same time

• Semaphore from railway analogy
  – Here is a semaphore initialized to 2 for resource control:

Two Uses of Semaphores

• Mutual Exclusion (initial value = 1)
  – Also called “Binary Semaphore”.
  – Can be used for mutual exclusion:
    semaphore.P();
    // Critical section goes here
    semaphore.V();

• Scheduling Constraints (initial value = 0)
  – Locks are fine for mutual exclusion, but what if you want a thread to wait for something?
  – Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:
    Initial value of semaphore = 0
    ThreadJoin {
      semaphore.P();
    }
    ThreadFinish {
      semaphore.V();
    }

Summary

• Important concept: Atomic Operations
  – An operation that runs to completion or not at all
  – These are the primitives on which to construct various synchronization primitives

• Talked about hardware atomicity primitives:
  – Disabling of Interrupts, test&set

• Showed several constructions of Locks
  – Must be very careful not to waste/tie up machine resources
    » Shouldn’t disable interrupts for long
    » Shouldn’t spin wait for long
  – Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable

• Semaphores: Higher level constructs that are harder to “screw up”