Review: A 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? Could be either
- 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?

Review: Hand Simulating Multiprocessor Example

- Inner loop looks like this:
  - Thread A
    - r1=0 load r1, M[i]
    - r1=1 add r1, r1, 1
    - M[i]=1 store r1, M[i]
  - Thread B
    - r1=0 load r1, M[i]
    - r1=-1 sub r1, r1, 1
    - M[i]=-1 store r1, M[i]

- Hand Simulation:
  - And we’re off. A gets off to an early start
  - B says “hmph, better go fast” and tries really hard
  - A goes ahead and writes “1”
  - B goes and writes “-1”
  - A says “HUH??? I could have sworn I put a 1 there”

- Could this happen on a uniprocessor?
  - Yes! Unlikely, but if you depending on it not happening,
    it will and your system will break...

Review: Too Much Milk Solution #3

- Here is a possible two-note solution:
  - Thread A
    - leave note A;
    - while (note B) { if (noNote A) {
      - do nothing;
      - if (noMilk) {
        - buy milk;
        - if (noMilk) {
          - buy milk;
        } else {
          - remove note B;
          - remove note A;
        }
      }
    }
  - Thread B
    - leave note B;
    - while (note B) { if (noNote A) {
      - do nothing;
      - if (noMilk) {
        - buy milk;
        - if (noMilk) {
          - buy milk;
        } else {
          - remove note B;
          - remove note A;
        }
      }
    }

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit
- At A:
  - if no note B, safe for A to buy,
  - otherwise wait to find out what will happen
- At B:
  - if no note A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit
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 implement higher-level operations on top of atomic operations provided by hardware
  - Develop a “synchronization toolbox”
  - Explore some common programming paradigms

Where are we going with synchronization?
• We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

- Programs
- Shared Programs

<table>
<thead>
<tr>
<th>Higher-level API</th>
<th>Locks</th>
<th>Semaphores</th>
<th>Monitors</th>
<th>Send/Receive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>Load/Store</td>
<td>Disable Ints</td>
<td>Test&amp;Set</td>
<td>Comp&amp;Swap</td>
</tr>
</tbody>
</table>

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
• Atomic Load/Store: get solution like Milk #3
  - Looked at this last lecture
  - Pretty complex and error prone
• Hardware Lock instruction
  - Is this a good idea?
  - Complexity?
  » Done in the Intel 432
  » Each feature makes hardware more complex and slow
  - What about putting task to sleep?
  » How do you handle the interface between the hardware and scheduler?

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;
}

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

- 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?

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

- 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?

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;
}
- 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;
  }
}
- load-linked&store conditional(&address) {
  /* R4000, alpha */
  loop:
  li r1, M[address];
  movi r2, 1; /* Can do arbitrary comp */
  sc r2, M[address];
  beqz r2, loop;
}

Implementing Locks with test&set

- Another flawed, but simple solution:
  int value = 0; // Free
  Acquire() {
    while (test&set(value)); // while busy
  }
  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
- Busy-Waiting: thread consumes cycles while waiting

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!

Higher-level Primitives than Locks

- Goal of last couple of lectures:
  - 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
- Synchronization is a way of coordinating multiple
  concurrent activities that are using share state
  - This lecture and the next presents a couple of ways of
    structuring the sharing
Semaphores

• Semaphores are a kind of generalized lock
  - 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
  - The book uses the terms: acquire() and release() for P() and V() respectively.

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();
    }
    ```

Producer-consumer with a bounded buffer

• Problem Definition
  - Producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization to coordinate producer/consumer
• Don't want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty
• Example: Coke machine
  - Producer can put limited number of cokes in machine
  - Consumer can't take cokes out if machine is empty
Correctness constraints for solution

- **Correctness Constraints:**
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)

- **Remember why we need mutual exclusion**
  - Because computers are stupid
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine

- **General rule of thumb:** Use a separate semaphore for each constraint
  - Semaphore fullBuffers; // consumer’s constraint
  - Semaphore emptyBuffers; // producer’s constraint
  - Semaphore mutex; // mutual exclusion

Full Solution to Bounded Buffer

```java
Semaphore fullBuffer = 0; // Initially, no coke
Semaphore emptyBuffers = numBuffers; // Initially, num empty slots
Semaphore mutex = 1; // No one using machine

Producer(item) {
    emptyBuffers.P(); // Wait until space
    mutex.P(); // Wait until buffer free
    Enqueue(item);
    mutex.V();
    fullBuffers.V(); // Tell consumers there is more coke
    mutex.V();
}

Consumer() {
    fullBuffers.P(); // Check if there’s a coke
    mutex.P(); // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V(); // tell producer need more
    return item;
}
```

Discussion about Solution

- **Why asymmetry?**
  - Producer does: emptyBuffer.P(), fullBuffer.V()
  - Consumer does: fullBuffer.P(), emptyBuffer.V()

- **Is order of P’s important?**
  - Yes! Can cause deadlock

- **Is order of V’s important?**
  - No, except that it might affect scheduling efficiency

- **What if we have 2 producers or 2 consumers?**
  - Do we need to change anything?

Motivation for Monitors and Condition Variables

- **Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores**
  - Problem is that semaphores are dual purpose:
    - They are used for both mutex and scheduling constraints
    - Example: the fact that flipping of P’s in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?

- **Cleaner idea:** Use *locks* for mutual exclusion and *condition variables* for scheduling constraints

- **Definition:** Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
  - Some languages like Java provide this natively
  - Most others use actual locks and condition variables
Monitor with Condition Variables

- **Lock**: the lock provides mutual exclusion to shared data
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free
- **Condition Variable**: a queue of threads waiting for something inside a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section

Simple Monitor Example

- Here is an (infinite) synchronized queue

```java
Lock lock; // Get Lock
Condition dataready; // Signal any waiters
Queue queue;

AddToQueue(item) {
    lock.Acquire(); // Get Lock
    queue.enqueue(item); // Add item
    dataready.signal(); // Signal any waiters
    lock.Release(); // Release Lock
}

RemoveFromQueue() {
    lock.Acquire(); // Get Lock
    while (queue.isEmpty()) { // If nothing, sleep
        dataready.wait(&lock);
    }
    item = queue.dequeue(); // Get next item
    lock.Release(); // Release Lock
    return(item);
}
```

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, swap, comp&swap, load-locked/store conditional
- **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
- **Talked about Semaphores, Monitors, and Condition Variables**
  - Higher level constructs that are harder to "screw up"