CS162
Operating Systems and Systems Programming
Lecture 7

Mutual Exclusion, Semaphores, Monitors, and Condition Variables

September 22, 2008

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

Review: Synchronization problem with Threads

- One thread per transaction, each running:

```
Deposit(acctId, amount) {
  acct = GetAccount(actId); /* May use disk I/O */
  acct->balance += amount;
  StoreAccount(acct);       /* Involves disk I/O */
}
```

- Unfortunately, shared state can get corrupted:

Thread 1	Thread 2
load r1, acct->balance	
	load r1, acct->balance
	add r1, amount2
	store r1, acct->balance
add r1, amount1	
store r1, acct->balance	

- **Atomic Operation**: an operation that always runs to completion or not at all
    - It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle

---

Review: Too Much Milk Solution #3

- Here is a possible two-note solution:

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

Review: Solution #3 discussion

- Our solution protects a single "Critical-Section" piece of code for each thread:

```
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

Goals for Today

- **Hardware Support for Synchronization**
- **Higher-level Synchronization Abstractions**
  - **Semaphores, monitors, and condition variables**
- **Programming paradigms for concurrent programs**

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

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**

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"**
- **Of course, you can make this even simpler: suppose you are out of ice cream instead of milk**
  - **Skip the test since you always need more ice cream.**

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
  - **Looked at this last lecture**
  - **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?**
    - » Done in the Intel 432
    - » 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**

```
int value = FREE;

Acquire() {                          Release() {
   disable interrupts;                  disable interrupts;
   if (value == BUSY) {                 if (anyone on wait queue) {
      put thread on wait queue;            take thread off wait queue
      Go to sleep();                       Place on ready queue;
      // Enable interrupts?             } else {
   } else {                               value = FREE;
      value = BUSY;                     }
   }                                    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

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

- **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?**

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

Enable Position →
Enable Position →
Enable Position →

Administrivia

- **My office hours: MW 2:30-3:30**
  - My faculty page through the department was wrong…
  - Does this work for people?
- **First Design Document due Wednesday 9/24**
  - Subsequently need to schedule design review with TA (through web form)
  - Note that most of the design document grade comes from first version (some from final version)
- **Design doc contents:**
  - Architecture, correctness constraints, algorithms, pseudocode, testing strategy, and test case types
- **Subversion accounts should be set up today or tomorrow**
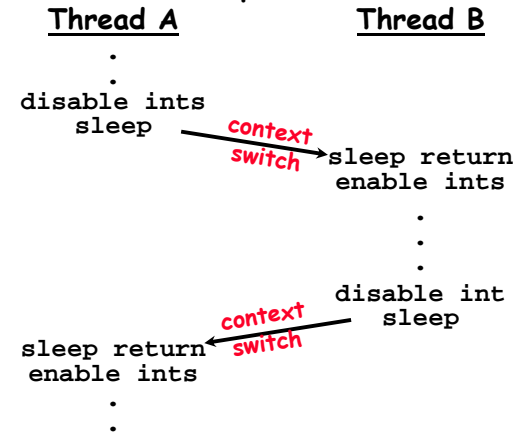 - Turns out that the TAs need to configure things

How to Re-enable After Sleep()?

- **In Nachos, 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

```
        Thread A                Thread B
            .
            .
  disable ints
     sleep  ──context──►
            switch  sleep return
                    enable ints
                            .
                            .
                            .
                    disable int
        context──►     sleep
        switch
  sleep return◄──
   enable ints
       .
       .
```

Interrupt disable and enable across context switches

- **An important point about structuring code:**
  - In Nachos code you will see lots of comments about assumptions made concerning when interrupts disabled
  - This is an example of where modifications to and assumptions about program state can't be localized within a small body of code
  - In these cases it is possible for your program to eventually "acquire" bugs as people modify code
- **Other cases where this will be a concern?**
  - What about exceptions that occur after lock is acquired? Who releases the lock?

```
        mylock.acquire();
        a = b / 0;
        mylock.release()
```

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;
      }
  }
  ```
- ```
  load-linked&store conditional(&address) {
      /* R4000, alpha */
      loop:
          ll 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!**

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**

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

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

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

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 shared 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**

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:**
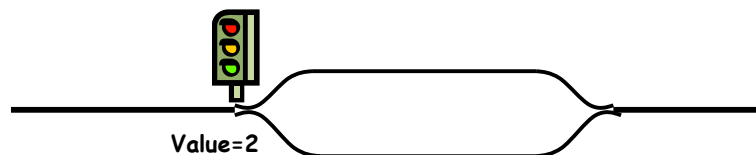
Value=2

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 terminiate:**
    ```
    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 1: GCC compiler
  - cpp | cc1 | cc2 | as | ld
- Example 2: 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

```
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
}
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?**

- **Is order of V's important?**

- **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
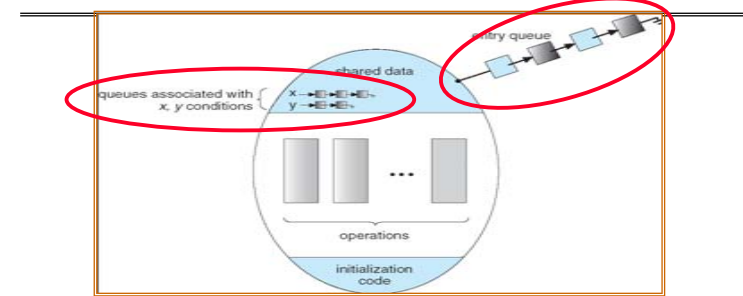
- **Lock: the lock provides mutual exclusion to shared data**
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free
- **Condition Variable: a queue of threads waiting for something *inside* a critical section**
  - Key idea: 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**

```
Lock lock;
Condition dataready;
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()) {
      dataready.wait(&lock); // If nothing, sleep
   }
   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-linked/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"