Lecture 7
Synchronisation:
Hardware support, Semaphores, Monitors, and Condition Variables

April 28, 2009

Léon Mugwaneza

http://www.it.uu.se/edu/course/homepage/os/vt09
Synchronization problem with Threads

- ATM server: one thread per transaction, each running:

  \[
  \text{Deposit}(\text{acctId}, \text{amount}) \begin{cases}
  \text{acct} = \text{GetAccount}(\text{acctId}); /* May use disk I/O */ \\
  \text{acct} -> \text{balance} += \text{amount}; \\
  \text{StoreAccount}(\text{acct}); /* Involves disk I/O */ 
\end{cases}
\]

- Unfortunately, shared state can get corrupted:

  Thread 1
  \[
  \begin{align*}
  &\text{load r1, acct} -> \text{balance} \\
  &\text{add r1, amount1} \\
  &\text{store r1, acct} -> \text{balance}
  \end{align*}
  \]

  Thread 2
  \[
  \begin{align*}
  &\text{load r1, acct} -> \text{balance} \\
  &\text{add r1, amount2} \\
  &\text{store r1, acct} -> \text{balance}
  \end{align*}
  \]

- 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
  - \text{Acct} -> \text{value} += \text{amount} is not Atomic, load and store are atomic (on architectures)

- Race Condition: outcome depends on thread interleaving
Goals for Today

• 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

HttpServletRequest{HttpServletRequest} Today, we’ll implement higher-level operations on top of atomic operations provided by hardware
  – Develop a “synchronization toolbox”
  – Explore some common programming paradigms

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne, others from Kubiatowicz - CS162 ©UCB Fall 2007 (University of California at Berkeley)
Too Much Milk: Solution #4

• Suppose we have some sort of implementation of a lock
  – 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 is free, only one succeeds to grab the lock

• Then, our too much milk problem is easy:
  ```java
  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
  – Only Critical Sections need to be accessed in a synchronized way
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 Too Much Milk #3
  - 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 a 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 (what?)
    » 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; }

  ➔ LockAcquire and LockRelease are system calls (in OS kernel)
  – Why?
    ➔ Remember how a process calls a procedure in OS kernel?

• 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) {
        // calling thread goes to sleep
        put thread on wait queue
        update thread state
        call scheduler
    } else {
        value = BUSY;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if(anyone on wait queue) {
        //resume one waiting thread
        take a thread off wait queue
        place thread on ready queue
    } else {
        value = FREE;
    }
    enable interrupts;
}
```

• Locks are provided by the OS (like unix pipes - see lab1)
  – Acquire and Release are system calls, we also need system calls to create and close ("kill") locks
  – Need 1 wait queue and 1 lock variable per lock
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) {
        // calling thread goes to sleep
        put thread on wait queue
        update thread state
        call scheduler
      } else {
        value = BUSY;
      }
    }
    enable interrupts;
    ```

• Note: unlike previous solution, the critical section (inside Acquire()) is very short
  – Users 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!
What if a thread does not release lock?

• What about exceptions that occur after lock is acquired?

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

– Who releases the lock?

• What if thread terminates without releasing the lock?

– Who releases the lock?

• Os releases lock when threads terminates or error causes thread abortion

– Are there errors which do not cause process abortion?

» Yes (eg. page fault - see later)
Atomic Read-Modify-Write instructions

• Problems with previous solution:
  – Can’t give lock implementation to users
  – Doesn’t work well on multiprocessor or multi-core CPU
    » Disabling interrupts on all processors/cores 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/multi-core (requires help from cache coherence protocol)
  – Unlike disabling interrupts, can be used on uniprocessors, multiprocessors, and multi-core CPUs
Examples of Read-Modify-Write

- test\&set (address, register) \{ /* most architectures */
  /* actual inst. slightly ≠, 68000: TAS, INTEL : BTS */
  register = M[address];
  M[address] = 1;
\}

- swap (address, register) \{
  temp = M[address];
  M[address] = register;
  register = temp;
\}

- compare\&swap (address, reg1, reg2) \{ /* 68000, Sparc */
  if (reg1 == M[address]) {
    M[address] = reg2;
    set bit of CCR=1 //CCR is Condition Code Register
  } else { set bit of CCR=0 }
\}

\^ compare and exchange on INTEL X86

- load-linked AND store conditional /* R4000, alpha */
  loop:
    ll r1, addr; //Load-linked: read M[addr] AND Remember addr
    movi r2, 1; // can do arbitrary computations
    sc r2, addr; //Store conditional: try to do M[addr]<-r2
    //store only if addr saved by ll not written meanwhile
    beqz r1, loop; // loop if lock read by ll was busy
Implementing Locks with test&set

• Another flawed, but simple solution:

```c
int value = 0; // Free

Acquire() {
    loop:
        test&set(&value, reg);
        if reg==1 goto loop; // while (busy) loop;
    }

Release() {
    value = 0;
}
```

• 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 to 0 in Release, someone else can get lock

• Busy-Waiting: thread consumes cycles while waiting
  – Also called spinlocking
Problem: Busy-Waiting for Lock

• Positives for this solution
  – Machine can receive interrupts
  – No system call (remember system calls have overhead)
  – Works on a multiprocessor/multi-core

• 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!

• 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
  – Good solutions (exams, labs, …) 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
    loop : test&set(&guard, reg)
      if reg==1 goto loop;
    if (value == BUSY) {
      - perform call to OS to put
        thread on wait queue, **AND**
        set guard to 0
      //OS launches another thread
    } else {
      value = BUSY;
      guard = 0;
    }
}

Release() {
    // Short busy-wait time
    loop: test&set(&guard, reg)
      if reg==1 goto loop;
    if anyone on wait queue {
      - perform call to OS to
        take 1 thread off wait
        queue and place it on
        ready queue;
    } else {
      value = FREE;
    }
    guard = 0;
}
```

• Note: guard variable reset by OS when thread goes to sleep
  – Why can’t we do it just before or just after the syst call to go sleep?
Higher-level Primitives than Locks

• 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
  – We will see 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, a wait queue and supports the following two operations (apart from initialization):
  - P(): an atomic operation that does the following:
    - if value = 0 then sleep
    - else decrement value by 1
    » Course book calls this operation wait()
  - V(): an atomic operation that does the following:
    - if there are any threads sleeping on that semaphore
      - wakeup 1 thread (random choice)
    - else increment value by 1
    » Course book calls this operation signal()
- Etymology: P() stands for “proberen” (to test) and V() stands for “verhogen” (to increment) in Dutch
- DOWN() sometimes used for P(), and UP() for V()

Some implementations allow negative values (P always decrements value by one, and V always increments value by one)
Semaphores are not integers!

- 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 types of Semaphores

- Mutual Exclusion semaphores (initial value = 1)
  - Also called “Binary Semaphore”.
  - Can easily be used for mutual exclusion:
    ```c
    semaphore sem ; // initial value = 1 (by construction)
    sem.P();
    // Critical section goes here
    sem.V();
    ```

- Counting semaphores (initial value can be any integer >=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:
    ```c
    semaphore sem = 0 // initial value = 0
    ThreadJoin {
        sem.P();
    }
    ThreadFinish {
        sem.V();
    }
    ```
  - Counting semaphores can be used to control access to a given resource consisting of a finite number of resources available. Consider a resource with N copies
    » Use a semaphore with initial value N
    » request a copy using P(), release copy using 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 “not very clever”
  – 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 = num; // Initially, num empty slots
Semaphore mutex = 1;       // No one using machine

Producer() {
    While (True) {
        do something else;  // including producing 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() {
    While (True) {
        fullBuffers.P();  // Check if there's a coke
        mutex.P();       // Wait until machine free
        item = Dequeue();
        mutex.V();
        emptyBuffers.V(); // tell producer a slot is free
        do something else;  // including using 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

• Monitor: zero or more condition variables for managing concurrent access to shared data, together with operations that are guaranteed to be mutual exclusive
  – Monitors are language constructs
  – Some languages like Java provide this natively
  – Most others use actual locks and 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
  – Two operations on conditions: `condition.wait()` and `condition.signal()`
  – 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
Example where thread sleeps inside critical section

• An (infinite) synchronized queue

```java
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
        // AND release lock, WILL GET lock BACK at
        // waking up (signal on condition dataready)
    })
    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

• hardware atomicity primitives:
  – Disabling of Interrupts, test&set, swap, comp&swap, load-linked/store conditional

• 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, Monitors, and Condition Variables
  – Higher level constructs that are easier to use