CS162
Operating Systems and Systems Programming
Lecture 8

Readers-Writers
Language Support for Synchronization

September 24, 2007

Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

Review: 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() {
   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;
}
```


Review: 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
                                     sleep
sleep return <--context switch--
enable ints
   .
   .
```


Review: Locks using test&set

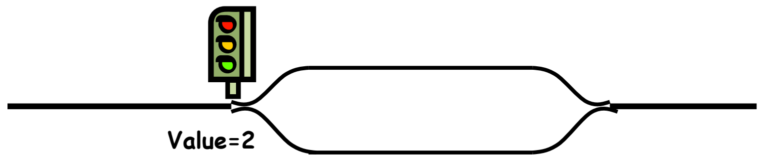
- Can we build test&set locks without busy-waiting?
  - Can't entirely, but can minimize!
  - Idea: only busy-wait to atomically check lock value

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

Acquire() {
   // 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;
   }
}

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;
   }
   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?

Review: Semaphores

- 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
  - Only time can set integer directly is at initialization time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



Goals for Today

- Continue with Synchronization Abstractions
  - Monitors and condition variables
- Readers-Writers problem and solutoin
- Language Support for Synchronization

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.

Review: 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?
  - Yes! Can cause deadlock:

```
Producer(item) {
  mutex.P();        // Wait until buffer free
  emptyBuffers.P(); // Could wait forever!
  Enqueue(item);
  mutex.V();
  fullBuffers.V(); // Tell consumers more coke
}
```

- 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, but:
  - They are confusing because they are dual purpose:
    - » Both mutual exclusion and scheduling constraints
    - » Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious
  - 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
  - Use of Monitors is a programming paradigm
  - Some languages like Java provide monitors in the language
- 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

Simple Monitor Example (version 1)

- Here is an (infinite) synchronized queue

```
Lock lock;
Queue queue;

AddToQueue(item) {
   lock.Acquire();        // Lock shared data
   queue.enqueue(item);   // Add item
   lock.Release();        // Release Lock
}

RemoveFromQueue() {
   lock.Acquire();        // Lock shared data
   item = queue.dequeue();// Get next item or null
   lock.Release();        // Release Lock
   return(item);          // Might return null
}
```


Condition Variables

- How do we change the RemoveFromQueue() routine to wait until something is on the queue?
  - Could do this by keeping a count of the number of things on the queue (with semaphores), but error prone
- **Condition Variable**: a queue of threads waiting for something *inside* a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section
- Operations:
  - `Wait(&lock)`: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  - `Signal()`: Wake up one waiter, if any
  - `Broadcast()`: Wake up all waiters
- Rule: Must hold lock when doing condition variable ops!
  - In Birrell paper, he says can perform signal() outside of lock – IGNORE HIM (this is only an optimization)

Complete Monitor Example (with condition variable)

- 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);
}
```

Mesa vs. Hoare monitors

- Need to be careful about precise definition of signal and wait. Consider a piece of our dequeue code:

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

  - Why didn't we do this?

```
if (queue.isEmpty()) {
   dataready.wait(&lock); // If nothing, sleep
}
item = queue.dequeue();// Get next item
```

- Answer: depends on the type of scheduling
  - Hoare-style (most textbooks):
    - » Signaler gives lock, CPU to waiter; waiter runs immediately
    - » Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again
  - Mesa-style (Nachos, most real operating systems):
    - » Signaler keeps lock and processor
    - » Waiter placed on ready queue with no special priority
    - » Practically, need to check condition again after wait

Administrivia

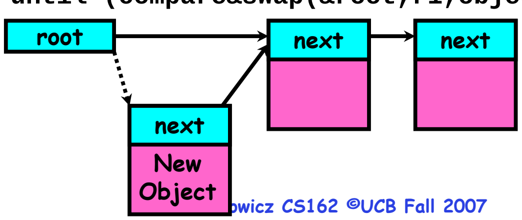
- First design document due *tomorrow*
  - I moved it to allow today's lecture
  - Has to be in by 11:59pm
  - Good luck!
- Design reviews:
  - Everyone must attend! (no exceptions)
  - 2 points off for one missing person
  - 1 additional point off for each additional missing person
  - Penalty for arriving late (plan on arriving 5—10 mins early)
  - Please sign up tomorrow (signup link off announcements)
- What we expect in document/review:
  - Architecture, correctness constraints, algorithms, pseudocode, NO CODE!
  - Important: testing strategy, and test case types

Using of Compare&Swap for queues

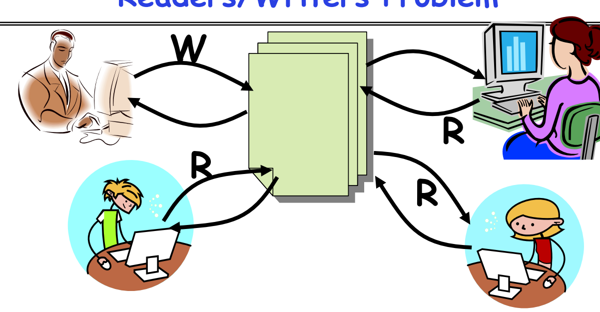
```
• compare&swap (&address, reg1, reg2) { /* 68000 */
     if (reg1 == M[address]) {
        M[address] = reg2;
        return success;
     } else {
        return failure;
     }
  }
```

Here is an atomic add to linked-list function:

```
addToQueue(&object) {
   do {                  // repeat until no conflict
      ld r1, M[root]     // Get ptr to current head
      st r1, M[object]   // Save link in new object
   } until (compare&swap(&root,r1,object));
}
```



Readers/Writers Problem



- Motivation: Consider a shared database
  - Two classes of users:
    - » Readers – never modify database
    - » Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    - » Like to have many readers at the same time
    - » Only one writer at a time

Basic Readers/Writers Solution

- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers or writers
  - Only one thread manipulates state variables at a time
- Basic structure of a solution:
  - Reader()

```
    Wait until no writers
    Access data base
    Check out – wake up a waiting writer
```

  - Writer()

```
    Wait until no active readers or writers
    Access database
    Check out – wake up waiting readers or writer
```

  - State variables (Protected by a lock called "lock"):
    - » int AR: Number of active readers; initially = 0
    - » int WR: Number of waiting readers; initially = 0
    - » int AW: Number of active writers; initially = 0
    - » int WW: Number of waiting writers; initially = 0
    - » Condition okToRead = NIL
    - » Conditioin okToWrite = NIL

Code for a Reader

```
Reader() {
  // First check self into system
  lock.Acquire();
  while ((AW + WW) > 0) { // Is it safe to read?
    WR++;                 // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--;                 // No longer waiting
  }
  AR++;                   // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  AR--;                   // No longer active
  if (AR == 0 && WW > 0)  // No other active readers
    okToWrite.signal();   // Wake up one writer
  lock.Release();
}
```


Code for a Writer

```
Writer() {
  // First check self into system
  lock.Acquire();
  while ((AW + AR) > 0) { // Is it safe to write?
    WW++;                 // No. Active users exist
    okToWrite.wait(&lock);  // Sleep on cond var
    WW--;                 // No longer waiting
  }
  AW++;                   // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  AW--;                   // No longer active
  if (WW > 0){            // Give priority to writers
    okToWrite.signal();   // Wake up one writer
  } else if (WR > 0) {    // Otherwise, wake reader
    okToRead.broadcast(); // Wake all readers
  }
  lock.Release();
}
```


Simulation of Readers/Writers solution

- Consider the following sequence of operators:
  - R1, R2, W1, R3
- On entry, each reader checks the following:

```
while ((AW + WW) > 0) { // Is it safe to read?
  WR++;                 // No. Writers exist
  okToRead.wait(&lock); // Sleep on cond var
  WR--;                 // No longer waiting
}
AR++;                   // Now we are active!
```

- First, R1 comes along:
  AR = 1, WR = 0, AW = 0, WW = 0
- Next, R2 comes along:
  AR = 2, WR = 0, AW = 0, WW = 0
- Now, readers make take a while to access database
  - Situation: Locks released
  - Only AR is non-zero

Simulation(2)

- Next, W1 comes along:

```
while ((AW + AR) > 0) { // Is it safe to write?
  WW++;                 // No. Active users exist
  okToWrite.wait(&lock); // Sleep on cond var
  WW--;                 // No longer waiting
}
AW++;
```

- Can't start because of readers, so go to sleep:
  AR = 2, WR = 0, AW = 0, WW = 1
- Finally, R3 comes along:
  AR = 2, WR = 1, AW = 0, WW = 1
- Now, say that R2 finishes before R1:
  AR = 1, WR = 1, AW = 0, WW = 1
- Finally, last of first two readers (R1) finishes and wakes up writer:

```
if (AR == 0 && WW > 0)  // No other active readers
  okToWrite.signal();   // Wake up one writer
```


Simulation(3)

- When writer wakes up, get:
  AR = 0, WR = 1, AW = 1, WW = 0
- Then, when writer finishes:

```
if (WW > 0){            // Give priority to writers
  okToWrite.signal();   // Wake up one writer
} else if (WR > 0) {    // Otherwise, wake reader
  okToRead.broadcast(); // Wake all readers
}
```

  - Writer wakes up reader, so get:
    AR = 1, WR = 0, AW = 0, WW = 0
- When reader completes, we are finished

Questions

- Can readers starve? Consider Reader() entry code:

```
while ((AW + WW) > 0) { // Is it safe to read?
  WR++;                 // No. Writers exist
  okToRead.wait(&lock); // Sleep on cond var
  WR--;                 // No longer waiting
}
AR++;                   // Now we are active!
```

- What if we erase the condition check in Reader exit?

```
AR--;                   // No longer active
if (AR == 0 && WW > 0)  // No other active readers
  okToWrite.signal();   // Wake up one writer
```

- Further, what if we turn the signal() into broadcast()

```
AR--;                   // No longer active
okToWrite.broadcast();  // Wake up one writer
```

- Finally, what if we use only one condition variable (call it "okToContinue") instead of two separate ones?
  - Both readers and writers sleep on this variable
  - Must use broadcast() instead of signal()

Can we construct Monitors from Semaphores?

- Locking aspect is easy: Just use a mutex
- Can we implement condition variables this way?

```
Wait()   { semaphore.P(); }
Signal() { semaphore.V(); }
```

- Does this work better?

```
Wait(Lock lock) {
   lock.Release();
   semaphore.P();
   lock.Acquire();
}
Signal() { semaphore.V(); }
```

Construction of Monitors from Semaphores (con't)

- Problem with previous try:
  - P and V are commutative – result is the same no matter what order they occur
  - Condition variables are NOT commutative
- Does this fix the problem?

```
Wait(Lock lock) {
    lock.Release();
    semaphore.P();
    lock.Acquire();
}
Signal() {
    if semaphore queue is not empty
        semaphore.V();
}
```

  - Not legal to look at contents of semaphore queue
  - There is a race condition – signaler can slip in after lock release and before waiter executes semaphore.P()
- It is actually possible to do this correctly
  - Complex solution for Hoare scheduling in book
  - Can you come up with simpler Mesa-scheduled solution?

Monitor Conclusion

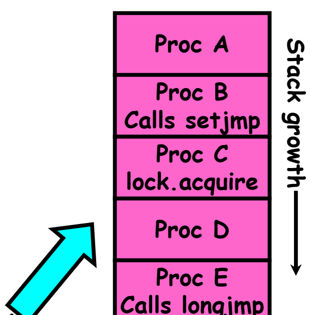
- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed
- Basic structure of monitor-based program:

```
lock
while (need to wait) {       } Check and/or update
   condvar.wait();           }   state variables
}                            }   Wait if necessary
unlock

do something so no need to wait

lock

condvar.signal();            } Check and/or update
                             }   state variables
unlock
```


C-Language Support for Synchronization

- C language: Pretty straightforward synchronization
  - Just make sure you know *all* the code paths out of a critical section

```
int Rtn() {
   lock.acquire();
   …
   if (exception) {
      lock.release();
      return errReturnCode;
   }
   …
   lock.release();
   return OK;
}
```



  - Watch out for setjmp/longjmp!
    - » Can cause a non-local jump out of procedure
    - » In example, procedure E calls longjmp, poping stack back to procedure B
    - » If Procedure C had lock.acquire, problem!

C++ Language Support for Synchronization

- Languages with exceptions like C++
  - Languages that support exceptions are problematic (easy to make a non-local exit without releasing lock)
  - Consider:

```
void Rtn() {
   lock.acquire();
   …
   DoFoo();
   …
   lock.release();
}
void DoFoo() {
   …
   if (exception) throw errException;
   …
}
```

  - Notice that an exception in DoFoo() will exit without releasing the lock

C++ Language Support for Synchronization (con't)

- Must catch all exceptions in critical sections
  - Catch exceptions, release lock, and re-throw exception:

```
void Rtn() {
  lock.acquire();
  try {
    …
    DoFoo();
    …
  } catch (…) {      // catch exception
    lock.release(); // release lock
    throw;          // re-throw the exception
  }
  lock.release();
}
void DoFoo() {
  …
  if (exception) throw errException;
  …
}
```

  - Even Better: auto_ptr<T> facility. See C++ Spec.
    - » Can deallocate/free lock regardless of exit method

Java Language Support for Synchronization

- Java has explicit support for threads and thread synchronization
- Bank Account example:

```
class Account {
  private int balance;
  // object constructor
  public Account (int initialBalance) {
    balance = initialBalance;
  }
  public synchronized int getBalance() {
    return balance;
  }
  public synchronized void deposit(int amount) {
    balance += amount;
  }
}
```

  - Every object has an associated lock which gets automatically acquired and released on entry and exit from a *synchronized* method.

Java Language Support for Synchronization (con't)

- Java also has *synchronized* statements:

```
synchronized (object) {
    …
}
```

  - Since every Java object has an associated lock, this type of statement acquires and releases the object's lock on entry and exit of the body
  - Works properly even with exceptions:

```
synchronized (object) {
  …
  DoFoo();
  …
}
void DoFoo() {
  throw errException;
}
```


Java Language Support for Synchronization (con't 2)

- In addition to a lock, every object has a single condition variable associated with it
  - How to wait inside a synchronization method of block:
    - » `void wait(long timeout); // Wait for timeout`
    - » `void wait(long timeout, int nanoseconds); //variant`
    - » `void wait();`
  - How to signal in a synchronized method or block:
    - » `void notify();    // wakes up oldest waiter`
    - » `void notifyAll(); // like broadcast, wakes everyone`
  - Condition variables can wait for a bounded length of time. This is useful for handling exception cases:

```
t1 = time.now();
while (!ATMRequest()) {
  wait (CHECKPERIOD);
  t2 = time.new();
  if (t2 – t1 > LONG_TIME) checkMachine();
}
```

  - Not all Java VMs equivalent!
    - » Different scheduling policies, not necessarily preemptive!

Summary

- **Semaphores: Like integers with restricted interface**
  - Two operations:
    - » `P()`: Wait if zero; decrement when becomes non-zero
    - » `V()`: Increment and wake a sleeping task (if exists)
    - » Can initialize value to any non-negative value
  - Use separate semaphore for each constraint
- **Monitors: A lock plus one or more condition variables**
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
    - » Three Operations: `Wait()`, `Signal()`, and `Broadcast()`
- **Readers/Writers**
  - Readers can access database when no writers
  - Writers can access database when no readers
  - Only one thread manipulates state variables at a time
- **Language support for synchronization:**
  - Java provides synchronized keyword and one condition-variable per object (with `wait()` and `notify()`)