Lecture 8
Synchronisation:
Semaphores, Monitors,
and Condition Variables
(cnt’d - 2)

May 4, 2009

Léon Mugwaneza

http://www.it.uu.se/edu/course/homepage/os/vt09
Goals

• Continue with Synchronization Abstractions
  – Monitors and condition variables
• Language Support for Synchronization

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)
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

• Operations:
  – **Wait()**: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  – **Signal()**: Wake up one waiter, if any
  – Note some monitor definitions have a 3rd operation:
    » **Broadcast()**: Wake up all waiters

• Rule: Must hold lock when doing condition variable operations
Synchronized queue (with condition variable)

- RemoveFromQueue waits until item available

```c
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);
}
```
Mesa vs. Hoare monitors

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

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

– Why didn’t we do this?

```java
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 and CPU to waiter; waiter runs immediately
    » Waiter gives up lock and processor back to signaler when it exits critical section or if it waits again
  – Mesa-style (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
Monitors are language constructs

• Programmer does not have to bother about lock:

```java
Monitor queueMonitor{
    Condition dataready;
    Queue queue; // and other internal state variables
    //init{...
    // internal procedures
    // AddToQueue & RemoveFromQueue are external ops
    AddToQueue(item) {
        queue.enqueue(item); // Add item
        dataready.signal(); // Signal any waiters
    }
    RemoveFromQueue() {
        while (queue.isEmpty()) {
            dataready.wait(); // If nothing, sleep
        }
        item = queue.dequeue(); // Get next item
        return(item);
    }
} // end Monitor queueMonitor
```

→ lock, and system call to lock.Acquire() and lock.Release() will be inserted by the compiler
Bounded Buffer Hoare Style Monitor (1)

• Monitor has 2 external operations:
  – append() and take()

• Monitor needs to hold the buffer:
  – ItemType buffer[num] ; // array of items;

• needs two condition variables:
  – notfull: notfull.signal() indicates that the buffer is not full.
  – notempty: notempty.signal() indicates that the buffer is not empty.

• needs buffer pointers and counts:
  – nextin: points to next item to be appended.
  – nextout: points to next item to be taken.
  – count: holds the number of items in buffer.
Bounded Buffer  Hoare style Monitor (2)

Monitor BoundedBuffer{
    itemType buffer[num];
    int nextin, nextout, count;
    condition notempty, notfull;
    init(){
        nextin =0; nextout=0; count =0 ;
    }
    external op
    appendString(itemType item) {
        if (count==num) notfull.wait();
        buffer[nextin] = item;
        nextin = (nextin+1) % num;
        count++;
        notempty.signal();
    }
    external_op_itemType take(){
        if (count=0) notempty.wait();
        item = buffer[nextout];
        nextout = (nextout+1) % num;
        count--;
        notfull.signal();
        return(item);
    }
} // end Monitor BoundedBuffer

process Producer
    While TRUE {
        <produce itemType v>
        BoundedBuffer.append(v)
    }
end Producer

process Consumer
    While TRUE{
        v=BoundedBuffer.take()
        <consume item v>
    }
end Consumer
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();  
  }
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:

```
Use monitor procedure

Do something that do not need to wait
```

```
Use monitor procedure

Check and/or update state variables
Wait if necessary
```

```
Use monitor procedure

Check and/or update state variables
```
Java Language Support for Synchronization

- Java has explicit support for threads and thread synchronization
- Bank Account example:
  ```java
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 (cont’d)

• Java also has `synchronized` statements:

  ```java
  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:

  ```java
  synchronized (object) {
    ...
    DoFoo();
    ...
  }
  ```

  ```java
  void DoFoo() {
    throw errException;
  }
  ```
Java Language Support for Synchronization (cont’d 2)

- In addition to a lock, every object has a single condition variable associated with it
  - How to wait inside a synchronization method or 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.now();
      if (t2 - t1 > LONG_TIME) checkMachine();
    }
  - Not all Java VMs equivalent!
    » Different scheduling policies, not necessarily preemptive!
Summary

• Semaphores: a non-negative integer value and queue with following operations:
  – Only time can set integer directly is at initialization time
  – $P()$: 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 (Think of this as the signal() operation)

• Monitors: A lock plus one or more condition variables
  – Mesa or Hoare style
  – State variables and mutually exclusive external operations
  – init operation executed once at creation
  – Use condition variables to wait inside critical section
    » Three operations on cond. var: Wait(), Signal(), and Broadcast()

• Language support for synchronization:
  – Java provides synchronized keyword and one condition-variable per object (with wait() and notify())