Synchronization with Monitors — Today

What's wrong with Semaphores?

Semaphores are a huge step up from the equivalent load/store implementation, but have the following drawbacks.

- They are essentially shared global variables.
- There is no linguistic connection between the semaphore and the data to which the semaphore controls access.
- Access to semaphores can come from anywhere in a program.
- They serve two purposes, mutual exclusion and scheduling constraints.
- There is no control or guarantee of proper usage.

A Monitor:

- is similar to a C++ class that ties the data, operations, and in particular, the synchronization operations all together,
- Unlike classes,
  - monitors guarantee mutual exclusion, i.e., only one thread may execute a given monitor method at a time.
  - monitors require all data to be private.

Monitors

More formally,

A Monitor defines a *lock* and zero or more *condition variables* for managing concurrent access to shared data.

- The monitor uses the *lock* to insure that only a single thread is active in the monitor at any instance.
- The *lock* also provides mutual exclusion for shared data.
- *Condition variables* enable threads to go to sleep inside of critical sections, by releasing their lock at the same time it puts the thread to sleep.

A monitor operation:

- Encapsulates the shared data you want to protect.
- Acquires the mutex at the start.
- Operates on the shared data.
- Temporarily releases the mutex if it can't complete.
- Reacquires the mutex when it can continue.
- Releases the mutex at the end.

Implementing Monitors in Java:

It is simple to turn a Java class into a monitor:

- Make all the data private
- Make all methods synchronized (or at least the non-private ones)

```
class Queue {
  private ...;  // queue data

  public void synchronized add( Object item ) {
    put item on queue;
  }

  public Object synchronized remove() {
    if queue not empty {
       remove item;
       return item;
    }
    else {
      ???
    }
  }
}
```


Condition Variables

How can we change remove() to wait until something is on the queue?

- Logically, we want to go to sleep inside of the critical section, but if we hold on to the lock and sleep, then other threads cannot access the shared queue, add an item to it, and wake up the sleeping thread. The thread could sleep forever.
- **Solution:** condition variables enable a thread to sleep inside a critical section, by atomically releasing the Lock at the same time as the thread is put to sleep.

A Condition variable: is a queue of threads waiting for something inside a critical section. Condition variables support three operations:

1. Wait(Lock lock) - atomic (release lock, go to sleep), when the process wakes up it re-acquires lock.
2. Signal() - wake up waiting thread, if one exists. Otherwise, it does nothing.
3. Broadcast() - wake up all waiting threads

Rule: thread must hold the lock when doing condition variable operations.

Condition Variables in Java

- Use wait() to give up the lock
- Use notify() to signal that the condition a thread is waiting on is satisfied.
- Use notifyAll() to wake up all waiting threads.
- Effectively one condition variable per object.

```
class Queue {
  private ...;   // queue data

  public void synchronized Add( Object item ) {
    put item on queue;
    notify ();
  }

  public Object synchronized Remove() {
    while queue is empty {
      wait ();
    }
    remove item;
    return item;
  }
}
```


Mesa versus Hoare Monitors

What should happen when signal() is called?

- If there are no waiting threads, the signaler continues and the signal is effectively lost (unlike what happens with semaphores).
- If there is a waiting thread:
  - One of the threads can execute, the other must wait or we will have two threads active in the monitor concurrently.

Mesa-style: (Java, and most real operating systems)

- The thread that signals keeps the lock (and thus the processor).
- The waiting thread waits for the lock.

Hoare-style: (most textbooks)

- The thread that signals gives up the lock and the waiting thread gets the lock.
- When the thread that was waiting and is now executing exits or waits again, it releases the lock back to the signaling thread.

Mesa versus Hoare Monitors (cont.)

The synchronized queuing example above works for either style of monitor, but we can simplify it for Hoare-style semantics:

- With Mesa-style, the waiting thread may need to wait again after it is awakened, because some other thread could grab the lock and remove the item before it gets to run.
- With Hoare-style, we can change the 'while' in Remove to an 'if' because the waiting thread runs immediately after an item is added to the queue.

```
class Queue {
  private ...;   // queue data

  public void synchronized add( Object item ) {
    put item on queue;
    notify ();
  }

  public Object synchronized remove() {
    if queue is empty {
      wait ();
    }
    remove item;
    return item;
  }
}
```


Java Readers/Writers

```
class ReaderWriter {
  private int numReaders = 0;
  private int numWriters = 0;

  private synchronized void prepareToRead () {
    while ( numWriters > 0 ) wait ();
    numReaders++;
  }
  private synchronized void doneReading () {
    numReaders--;
    if ( numReaders == 0 ) notify ();
  }
  public void someReadMethod () {
    prepareToRead ();
    <do the reading>
    doneReading ();
  }
  private void prepareToWrite () {
    numWriters++;
    while ( numReaders != 0 ) wait ();
  }
  private void doneWriting () {
    numWriters--;
    notify ();
  }
  public synchronized void someWriteMethod (...) {
    prepareToWrite ();
    <do the writing>
    doneWriting ();
  }
}
```

Monitors in C++

Monitors in C++ are more complicated:

- No synchronization keyword, so the class must provide the lock, initialize, acquire and release it correctly.

Example:

```
class Queue {
  public:
     Add();
     Remove();
  private
     Lock lock;
     ...             // all the queue data();
}
Queue::Add() {
  lock->Acquire();          // lock before using data
  put item on queue;        // ok to access shared data
  condition->Signal();
  lock->Release();           // unlock after access
}
Queue::Remove() {
  lock->Acquire();          // lock before using data
  while queue is empty
    condition->Wait(lock); // release lock & sleep
  remove item from queue;
  lock->Release();          // unlock after access
  return item;
}
```

Notice that the programmer of the monitor must make the call to lock→Acquire() before performing condition→Wait().

C++ Bounded Buffer using Hoare-style condition variables

```
class BBMonitor {
   public:  void Append(item); void Remove(item);
   private:
     item buffer[N];
     int  lastpointer, count;
     Condition full, empty;

}
BBMonitor::BBMonitor {
   count = 0;
   lastpointer = 0;
}
BBMonitor::Append(item){
  lock->Acquire();
  if (count == N) empty->Wait(lock);
  buffer[lastpointer] = item;
  lastpointer = lastpointer + 1 mod N;
  count += 1;
  full->Signal();
  lock->Release();
}
BBMonitor::Remove(item){
  lock->Acquire();
  if (count == 0) full->Wait(lock);
  item = buffer[(lastpointer-count) mod N];
  count = count-1;
  empty->Signal();
  lock->Release();
}
```

Summary

- Monitor wraps operations with a mutex.
- Condition variables release mutex temporarily.
- C++ does not provide a monitor construct, but monitors can be implemented by following the monitor rules for acquiring and releasing locks.

Next Time

- Comparing monitors and semaphores
- In class monitor problem for credit.