Synchronization with Monitors

Drawbacks of Semaphores

- There is no linguistic connection between the semaphore and the data to which the semaphore controls access — they are essentially shared global variables.
- 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:

- An object that enacapsulates the data and their operations, and in particular, the synchronization operations.
- Unlike ordinary objects (such as those defined by classes in C++),
  - 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 *condition variables* for managing concurrent access to shared data.

- The monitor uses the *lock* to ensure that only a single thread is active in the monitor code at any time.
- This *lock* provides mutual exclusion for shared data.
- *Condition variables* enable threads to go to sleep inside 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 (or the relevant data)
- Make all methods synchronized (or at least the relevant 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.

Condition Variables

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: (Nachos, 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 ... 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 ();
  }
}
```

Comparing Semaphores and Monitors

Can we build monitors out of semaphores? After all, semaphores provide atomic operations and queuing. Does the following work?

```
condition->Wait() { semaphore->wait(); }
condition->Signal() { semaphore->signal(); }
```

But condition variables only work inside a lock. If we use semaphores inside a lock, we have may get *deadlock*. Why?

How about this?

```
condition->Wait(Lock *lock) {
   lock->Release();
   semaphore->wait();
   lock->Acquire();
}

condition->Signal() {
  semaphore->signal();
}
```

Comparing Semaphores and Monitors

Condition variables do not have any history, but semaphores do.

- On a condition variable signal, if no one is waiting, the signal is a no-op.
  $\Longrightarrow$ If a thread then does a condition$\rightarrow$Wait, it *waits*.
- On a semaphore signal, if no one is waiting, the value of the semaphore is incremented.
  $\Longrightarrow$ If a thread then does a semaphore$\rightarrow$Wait, the value is decremented and the thread *continues*.

In other words, semaphore$\rightarrow$Wait and Signal are commutative, the result is the same regardless of the order of execution. Condition variables are not, and as a result they must be in a critical section to access state variables and do their job.

...It is possible to implement monitors with semaphores...

Implementing Monitors with Semaphores

```
class Monitor {
  public:
    void ConditionWait();   // Condition Wait
    void ConditionSignal(); // Condition Signal
  private:
    <shared data>;          // data being protected by monitor
    semaphore cvar;         // suspends a thread on a wait
    int waiters;            // number of threads waiting on
                            // a cvar (one for every condition)
    semaphore lock;         // controls entry to monitor
    semaphore next;         // suspends this thread when it
                            //               signals another
    int nextCount;          // number of threads suspended
}                           //                       on next
Monitor::Monitor {
   cvar = 0;    // Nobody waiting on condition variable
   waiters = 0;
   lock = FREE; // Nobody in the monitor
   next = 0;    // Nobody suspended inside the monitor
   nextCount = 0;
}

// Wrapper code for all methods on the shared data
Monitor::someMethod () {
   lock->Wait();        // lock the monitor
   <ops on data and calls to ConditionWait() and ConditionSignal
   if (nextCount > 0)
      next->Signal();   // resume a suspended thread
   else
      lock->Signal();   // allow a new thread into the monitor
}
```

Implementing Monitors with Semaphores

```
Monitor::ConditionWait() {              // Condition Wait
   waiters += 1;
   if (nextCount > 0)
      next->Signal();   // resume a suspended thread
   else
       lock->Signal();  // allow a new thread in the monitor
   cvar->wait();        // wait on the condition
   waiters -= 1;
}
Monitor::ConditionSignal(){             // Condition Signal
   if (waiters > 0) {    // don't signal cvar if nobody is waitin
      nextCount += 1;
      cvar->Signal();          // Semaphore Signal
      next->Wait();            // Semaphore Wait
      nextCount -= 1;
    }
}
```

Is this Hoare semantics or Mesa semantics? What would you change to provide the other semantics?