Lecture 24
Concurrent Programming

21st November 2002

Today we continue looking at reducing blocking and synchronization.

Example - Synchronous message passing kernel

The synchronization structure of my solution to the synchronous message passing kernel used passing the baton for a single split binary semaphore made up of a global semaphore and a semaphore per selection list that serves for both synchronization and signalling as is standard with the split binary semaphore technique.

```
select (SL):
   P(globalsem);
   for element in SL:
      if Poll(element) {
         V(element.SL.waitsem);
         return;
      }
   for element in SL:
      block(element);
   V(globalsem)
   P(SL.waitsem);
   for element in SL:
      unblock(element);
   V(globalsem);
```

The obvious potential problem is that the global semaphore will prevent two pairs of processes from simultaneously synchronizing even if the the selection lists of the pairs are disjoint. That is, if SL1 and SL2 share some channels (so that they can communicate), and SL3 and SL4 share channels, but channels(SL1+SL2) is disjoint from channels(SL3+SL4), the two communications cannot happen in parallel because of the global lock.

Warning: I have not been able to eliminate the need for the global lock, but the attempt below illustrates a couple of points and techniques that I can share. My attempt is based on using a mutex per selection list and a mutex per channel. Selection lists continue to have a signalling semaphore (SL.waitsem).

```
select (SL):
   acquire SL.mutex
   acquire all the channel locks for SL
   for element in SL:
       if element' = Poll(element) {
         acquire element'.SL.mutex
         acquire element'.SL.channellocks (ones not already held)
         do the transfer
         cleanup element'.SL (remove elements from channels)
         release all channel locks; V(SL.mutex);
         V(element'.SL.waitsem)
         return)
      V(element.SL.mutex)
   for element in SL:
      block(element);
   release all channel locks;
   V(SL.mutex);
   P(SL.waitsem)
   V(SL.mutex)
```

First problem: acquire all the channel locks – we know that the locks must be acquired in the same order everywhere or deadlock is possible. To make this easier we could introduce a lock ordering manager

```
class LOM {
   void sortLocks(Semaphore[] locks) {...}
   void runWithLocks(Runnable op, Semaphore[] locks) {
      sortLocks(locks);
      int lastlocked = -1;
      InterruptedException caught = null;

      try {
        for (int i=0; i<locks.length; i++) {
           locks[i].acquire();
           lastlocked = i;
        }
        op.run();
      }
      catch (InterruptedException ie) {
         caught = ie;
```

```
            }
            finally {
               for (int j = lastlocked; j>=0; --j) {
                  locks[j].release();
               }
               if (caught != null) throw caught;
            }
         }
      }
```

Notice the care that has been placed on dealing with InterruptedException, but also notice that the code does not allow use of any user-defined exceptions. In general, exceptions require great care when used in conjunction with synchronization. As we observed early on in talking about synchronized methods, Java automatically releases the locks for synchronized methods when the synchronized block is exited for any reason. But when we build our own synchronization objects, such as semaphores, we no longer have this benefit. We have to make sure that whenever we hold a semaphore a finally block will release it if an exception occurs.

This raises the second ugly interaction of synchronization and exceptions which also pertains to synchronized methods: the lock may be released, but does the data structure it protects satisfy its invariant? In general not. We need explicit exception handlers to put the data structure into a consistent state before allowing the exception to propagate.

Returning now to our example, unfortunately the ordered lock manager technique won't help us enough. Although two processes that are successfully communicating won't deadlock even though they acquire the two SL.mutex variables in opposite order (why?), a third process can cause a deadlock:

Process1 SL1	Process2 SL2	Process3 SL3
c1	c1	c2
c2		

Process 1 enqueues events on all three channels and releases the SL1 lock.

Process 2 acquires the SL2 lock and the c1 lock and the SL1 lock; Process3 acquires the SL3 lock and the c2 lock; now Process3 needs the SL1 lock and the c1 before releasing any locks; Process 2 needs the c2 lock before releasing any locks. – Deadlock.

Possible solution? *lock backoff* – if the lock is not available within a certain period of time, release all locks, undo all work done so far, wait awhile, and try again. How long should the timeout be? How long should the process wait before trying again?