Lecture 23
Concurrent Programming

20th November 2003

Today we continue our tour of techniques for 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:
      enqueue(element);
   V(globalsem)
   P(SL.waitsem);
   for element in SL:
      dequeue(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.

My attempt to solve this problem 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):
   # by acquiring all the channel locks for SL we ensure that no
   # other process can affect the decision of this process to
   # enqueue its elements for waiting, if that is what it decides to do
   SL.completed = false
   acquire all the channel locks for SL
   for each element in SL {
       if ((element' = Poll(element))!= null) {
         # there is a matching element on the channel
         # note double-checked locking variant
         acquire element'.SL.mutex
         # need to hold element'.SL.mutex for the following
         # so that no other process can successfuly synchronize
         # with element'.SL
         if element'.SL.completed:
            release element'.SL.mutex
            continue
         element'.SL.completed = true
         release element'.SL.mutex
         transfer data between element and element'
         V(element'.SL.waitsem)
       }
   }
   # did not find any uncompleted matching elements
   for element in SL {
      enqueue(element);
   }
   # all elements have been enqueued on their channels
   # and none has partnered in a communication
   release all channel locks;
   # note the importance of this being a semaphore:
   # V(SL.waitsem) may already have been performed
   P(SL.waitsem)
   # when woken exactly one element has passed data
   # and SL.completed is true
   # therefore no other process will do any other actions on SL
   for each element in SL {
      if this element was the one that was selected {
         res = element
      }
      lock element.channel.mutex
      dequeue element from element.channel
      unlock element.channel.mutex
```

```
    }
    # discuss value of defensive programming here
    return res
```

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.