Lecture 8
Concurrent Programming

24th September 2002

Eventual entry using swap to build a linked-list of waiters

Thank you to whoever suggested waiting on something in the predecessor. Combined with an encoding trick it yielded a nice solution.

```
int fin[0..n] = ([n] 0);
int lastProcess = 0;

process CS[i=1 to n] {
   int waitingFor, waitingSub, waitingFin;
   while (true) {
      -- CS entry
      waitingFor = i*2 + ((fin[i]+1) mod 2)
      swap(waitingFor, lastProcess);
      waitingSub = waitingFor/2;
      waitingFin = waitingFor mod 2;
      -- if (waitingSub != i) {
      while (fin[waitingSub] != waitingFin) skip;
      -- }
      CS
      fin[i] = (fin[i]+1) mod 2
      NCS
   }
}
```

Introducing monitors

History – strong ties to Simula class concept which is part of the history of OO languages. Several different definitions for monitors have been put forth, but all are basically similar.

Original idea is to associate both mutual exclusion and condition synchronization with data at the language level.

One way to look at monitors: structured mutual exclusion + efficient waiting.

Mutual exclusion: why not just use binary semaphores? One could, and indeed in the pthreads library for C this is (essentially) the approach taken. But a language with an integrated notion of concurrency can do better: associate lock acquisition with procedure (or block) entry and lock release with procedure (or block) exit. Experience shows that by making lock management correspond to program structure, programmers make fewer mistakes – primarily in not forgetting lock release code. This becomes more important if the language also supports exception handling. Exceptions complicate the possible execution paths, and it is very easy to forget to release a lock on some path.

Example: Java will always release a lock on exit from a synchronized procedure (or block). Countering opinion: in the Mesa programming language invented at Xerox PARC the monitor lock would not be released by an exception unless there was a handler for it in the monitor procedure. The thinking was that since the programmer must ensure that the monitor invariant holds whenever the lock is not held, it was necessary to insist at the language level that the program contain at least an identifiable point where this could be done.

Key steps in using monitor mutual exclusion effectively: identify a data structure that needs to be referenced atomically. Associate a mutex with the data structure (in Java this step is implicit – every object has an associated mutex). Design an invariant for the data structure that holds whenever the monitor lock is not held. The invariant you would use for single-threaded use of the object is a good starting point.

For condition synchronization, monitors provide the so-called condition variable, which really a queue of waiting processes. (In Java, a single condition variable is implicitly associated with each object.) The key operation on a condition variable *and a mutex* (the code in section 5.1.2 doesn't mention the mutex, but it is there as an implicit parameter) is the *wait* operation which joins the queue, releases the lock, and gives up the processor, and re-acquires the monitor lock. Notice how the order of these operations is critical. The other operations on condition variables are *signal*() and *signal_all*() (in some languages they are known as *notify* and *notifyall* or *broadcast*). Following a *signal*() operation, one process (if there are any waiting) is selected from the queue, and made runnable, ready to acquire the monitor lock. *signal_all* makes all the threads in the queue runnable. They thus all compete for the lock.

Also notice that a condition variable captures absolutely nothing about *why* a process is waiting. It is purely a matter of convention between the signaller and the waiter. Therefore it must always be used as follows:

```
{I} -- note mutex is held
while (!B) wait(cv);
```

```
{I ∧ B}
```

Important: as always, the invariant must be proved to be an invariant – you can't make an arbitrary formula an invariant just by saying that it is. However, B is another matter. Following while(!B)...wait, B will always be true using our friend the while rule (provided of course it isn't interfered-with by assignment actions in other processes).

Notice that I say *always* put the wait statement in a while loop. Although the text shows examples guarding a wait with an if, I do not recommend this practice. To do it safely is tricky – and it is too easy for a subsequent programmer to change things so that the code is no longer correct. The resulting bugs are subtle and hard to find. For me, seeing

```
if (...) wait()
```

is a waving red flag indicator of a potential bug. I've seen it several times in production code and it has *always* been wrong.

Example: semaphore implemented using a monitor

```
monitor Sem {
   int s = init;
   cond positive; -- signalled when s>0

   procedure P() {
      {s≥0}
      if (s==0) wait(positive); -- wrong
      while (s==0) {s≥0} wait(positive);
      {s>0}
      s = s-1;
      {s≥0}
   }

   procedure V() {
      {s≥0}
      s = s+1;
      signal(positive);
      {s≥0}
   }
}
```

The book talks about additional condition variable operations *empty()* and *minrank()*. These are not usually found in concurrent programming systems, so solutions such as that in Fig. 5.8 cannot be created quite so simply. Let's look at how we could accomplish the effect of 5.8 without using empty and minrank.

We will have to maintain our own queue of waiting threads and arrange to wake up only the correct thread. A first easy step is to wake up all the threads only if some thread needs to be awakened which is already better than Fig. 5.7.

To do so we'll suppose that we have an unmonitored data structure for queues available with operations to insert and remove queue elements and to examine the head element. Our solution might then look something like this.

```
monitor Timer {
   int tod = 0;
   cond check;
   OrderedQueue q;

   procedure delay(int interval) {
      int waketime = tod+interval;
      QueueElement qe = new QueueElement(waketime);
      q.insert(qe);
      while (tod<waketime) wait(check);
      q.remove(qe);
   }
   procedure tick() {
      tod = tod+1;
      if q.head.val < tod signal_all(check);
   }
}
```

We're not quite there: to succeed we need a scheme that has at most one process waiting on each condition variable, so that a signal can be directed to a particular waiting process. Depending on the programming language this can range from easy to complicated. In pthreads for example, it is straightforward because the pthread_cond_wait operation allows/requires explicit identification of both the mutex and the cond variable. It would look something like this:

```
monitor Timer {
   int tod = 0;
   cond check;
   OrderedQueue q;
   pthread_mutex_t mutex;

   procedure delay(int interval) {
      pthread_cond_t check;
      pthread_mutex_lock(&mutex);
      int waketime = tod+interval;
      QueueElement qe = new QueueElement
                              (waketime, &check);
      q.insert(qe);
```

```
        while (tod<waketime) pthread_cond_wait(&check);
                             -- if instead of while?
    }
    procedure tick() {
        tod = tod+1;
        while q.head().val < tod {
            QueueElement qe = q.removeHead();
            pthread_cond_signal qe.cond
        }
    }
}
```

In Java, however, things become messy. There are two problems: in Java, in order to have n condition variables you must use n objects. Condition variables are in one-one correspondence to objects. Worse, those condvars are also statically paired with the mutexes in those objects and in order to wait on a cond var you must first be holding the corresponding mutex.

Let's revisit this after we talk in more detail about monitors in Java.