Concurrent Programming
Lecture 9

26th September 2002

Please cross out all timer example material in the Lecture 8 notes. Some of the code is wrong.

Timer Example - part 1 - covering condition

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

   procedure delay(int interval) {
      int waketime = tod+interval;
      while (tod<waketime) wait(check);
   }
   procedure tick() {
      tod = tod+1;
      signal_all(check);
   }
}
```

Timer Example - part 2 - more precise covering condition

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().waketime < 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.

Monitors in pthreads

The pthreads (portable threads) library for many *nix systems provides user-space threads and methods for synchronizing them. Section 4.6 discusses the thread manipulation primitives and the semaphore operations in pthreads. Today we'll talk about the monitor-like operations. The necessary declarations are in pthread.h so

```
#include <pthread.h>
```

in each program module using thread primitives. Since pthread objects are outside the language the programmer must explicitly initialize them using pthread_mutex_init and pthread_cond_init. Depending on how they are initialized, pthread mutexes can either allow multiple acquisition by a single thread (recursive), deadlock on an attempted second acquisition (fast, default), or indicate an error on a second acquisition (error-checking). A thread that acquires a lock multiple times must release it the same number of times.

Warning about recursive acquisition (also applies to Java): good monitor programming relies on monitor invariants – on entry to monitor code you may assume that the invariant holds; on exit (either just before a wait, or before return from the procedure) ensure that it holds. Recursive monitor locks make it very easy to break this programming paradigm. *Every* procedure call is potentially a point where you must ensure the invariant holds (or argue why that

procedure call will not lead to recursive entry). If recursive entry is not allowed, your program deadlocks immediately. If it is allowed, your data structures may become corrupted but the program continues running.

mutex operations

pthread_mutex_{init,lock,trylock,unlock,destroy}

condition operations

pthread_cond_{init,wait,timedwait,signal,broadcast,destroy}

man pages

Manual pages are available on most eecs linux machines for the pthread library.

```
man pthread_{cond,mutex}_init
```

Timer Example - part 3 - pthreads

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:

```
PTHREADMONITOR Timer {
   int tod = 0;
   OrderedQueue q;
   pthread_mutex_t mutex;
   pthread_mutex_init(&mutex, NULL);
   pthread_cond_init(&check, NULL);

   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, &mutex);
      }
      pthread_mutex_unlock(&mutex);
```

```
    }
    procedure tick() {
       tod = tod+1;
       pthread_mutex_lock(&mutex);
       while q.head().waketime < tod {
          QueueElement qe = q.removeHead();
          pthread_cond_signal qe.cond
       }
       pthread_mutex_unlock(&mutex);
    }
}
```

Monitors in Java

In Java there is a monitor implicitly associated with every object, that is anything that descends from java.lang.Object. The monitor of course has a mutex and it has a single condition variable. The fact that there is only one condition variable encourages programming with covering conditions as in our Timer examples 1 and 2.

The mutex for an object is acquired using a synchronized statement or synchronized method. The former is second is defined in terms of the first, so we'll look at the synchronized statement first.

```
synchronized(<object>) { stmts }
```

means acquire object's lock, execute stmts and release the lock. To create a synchronized method, prepend the word synchronized to a normal method definition.

```
synchronized int foo(int x) { stmts }
```

is equivalent to

```
int foo(int x) { synchronized (this) {stmts}}
```

Conditions

The condition operations in Java are object.wait(), object.notify(), object.notifyAll(). There also timed versions of wait. Condition operations may *only* be used within a synchronized method or statement holding the corresponding object mutex.

Documentation

Documentation for the synchronization primitives is found with the java.lang.Object documentation.

Timer Example - part 4 - Java

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

```
class Timer ... {
   volatile int tod = 0;
   OrderedQueue q = new...;
   delay(int interval) {
      int waketime;
      QueueElement qe;
      synchronized (this) {
         wakeTime = tod+interval;
         qe = new QueueElement(waketime);
         q.insert(qe);
      }
      synchronized (qe) {
         while (tod<waketime) wait();
      }
   }
   synchronized procedure tick() {
      tod = tod+1;
      while q.head().waketime < tod {
         QueueElement qe = q.removeHead();
         synchronized(qe) {qe.notify();}
      }
   }
}
```

Work focus for the next week

Exam study. Material from chapters 1 through 5 covered in lectures, notes, and homework problems. Midterm exam is next Thursday, October 3, during class period.