Concurrent Programming
Lecture 10

25th September 2003

Monitors & P/V

Notion of a process being "not runnable": implicit in much of what we have said about P/V and monitors is the notion that a process may be in one of several *scheduling states* at any given time. Typical states: *running, runnable* (but not running), *waitP, waitML, waitCV.* The *scheduler* decides at each moment how to allocate the CPU (CPUs) to the running and runnable processes. Thus processes in any of the wait* states do not use any CPU resources, though they do occupy memory.

Synchronization operations (P, V, monitor entry/exit, condition wait/signal) cause processes to change their scheduling states. Some state changes occur because of the action of the affected process (transitions from running to wait*), while transitions out of wait* states are caused by other processes (or the passing of time).

Using pthreads primitives to simulate monitors

... which might also be entitled: what do monitor constructs compile to?

```
Monitor M {
   int x = 47;
   condition c;
   int P(int y) {
      while (x<0) wait(c);
      x = x-y;
      if (x<=0) notify(c);
   }
   int Q(int y) {
      while (x>0) wait(c);
      x = x+y;
```

```
      if (x>=0) notify(c);
   }
}
```

Here, P is a monitor procedure, x is part of the protected state of the monitor. Suppose we want to "compile" this code to use the pthread primitives. We'll need a mutex to provide mutual exclusion between P and Q and a pthread condition variable corresponding to c. In C using pthreads our code would be

```
#include <pthread.h>
static int x = 47;
static pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t c = ?

int P(int y) {
   pthread_mutex_lock(&m);
   while(x<0) pthread_cond_wait(&c, &m);
   x = x-y;
   if (x<=0) pthread_cond_signal(&c);
   pthread_mutex_unlock(&m);
}

int Q(int y) {
   pthread_mutex_lock(&m);
   while(x>0) pthread_cond_wait(&c, &m);
   x = x+y;
   if (x>=0) pthread_cond_signal(&c);
   pthread_mutex_unlock(&m);
}
```

Why does pthread_cond_wait take both a cond and a mutex as a parameter? Need to describe more fully what mutex_lock and cond_wait do.

```
mutex_lock(m) {
   while(TAS(m->lock)) {
      joinWaiters(m->waitSet);
      enterBlockedState()
   }
}
mutex_unlock(m) {
   if (m->waitSet not empty) {
      pick one or more processes from m->waitSet and put them in
      ready state
   }
   m->lock = 0
```

```
}
cond_wait(c,m) {
   joinWaiters(c->waitSet);
   mutex_unlock(m);
   enterBlockedState();
   mutex_lock(m);
}
signal(c) {
   // why does java insist the the object mutex be held
   // by signal's caller?
   // pthreads does not.
   if (c->waitSet not empty) {
      pick one or more processes from c->waitSet and put them in
      ready state
   }
}
```

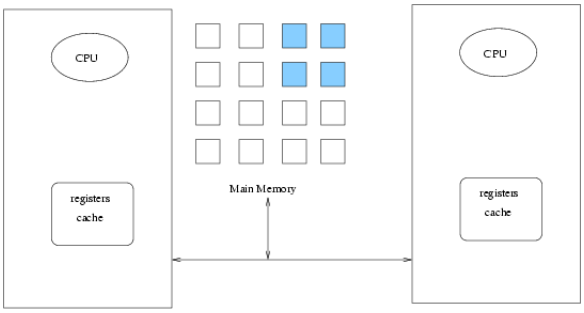
Timed wait: I mentioned but didn't go into details the notion of a timed wait() operation. In java it looks like wait(millis) or wait(millis,nanos). A process calling a timed wait operation waits for *either* the amount of time to pass *or* some other process to signal. Notice that if both occur in quick succession you can't necessarily predict which will cause the waiting process to wake up. In Java, there isn't any way for the process to tell which occurred, anyway. (In pthreads the awakened thread *can* tell whether or not the wait timed out).

Interrupted wait: another way for a wait to finish in Java is for the waiting thread to have its interrupt() method called. If this happens, as soon as the thread reacquires the monitor lock it will throw the InterruptedException. I am not a big fan of using interrupt(), or its closest analog in C, kill() for thread communication. There are usually better ways to do things.

Java

Java synchronized methods use the recursive locking model (also called re-entrant locking): if a thread holds an object's lock, it will successfully enter other synchronized methods of the object without blocking.

Java's memory model is closer to that of a multiprocessor machine than a uniprocessor.



Even on a uniprocessor, modern machines (and compilers) have a great deal of freedom to reorder loads and stores. Consider for example:

```
class SetCheck {
   private int a = 0;
   private long b= 0;

   void set () {
      a = 1;
      b = -1;
   }

   boolean check () {
      return ((b==0) ||
              (b==-1 && a==1));
   }
}
```

In set(), the compiler and memory system may store b before a, the high word of b then a then the low word of b, or even might never store the values in the main memory locations of a and b.

In a single-threaded program you are guaranteed that the order of loads and stores from memory seen by your program will be what you expect: loads following stores in the program text will see the result of the store; loads preceding stores will not see the result of the store. This is called *as-if-serial* semantics.

In a multithreaded program no such guarantee is made, unless synchronized methods and the volatile declaration are used. For synchronized methods, java will ensure that between the exit of one synchronized method and entering another using the same lock, all stores of the first are pushed to main memory. So conventional use of monitors to provide atomic actions also ensures memory consistency.

The main other way under programmer control to ensure that changes are visible is to declare a field volatile.

```
volatile int foo;
```

Assignments to volatile variables will be pushed to main memory prior to the thread performing any other load or store operations. And each time a volatile variable is read, it is fetched from main memory.

Finally, as a thread terminates all variables that it has written are pushed to main memory.

The Java memory system is complicated and there are a good many rules about what can happen when – but the only safe course is to assume that you must use volatile declaration or a synchronized block to access any data that is shared between threads.

If you refer to the timer example part 4 from last time you'll see that we used both synchronized and volatile. The tod variable was written in a synchronized block holding one monitor lock and read while holding a different lock, so it had to be declared volatile.

Current Java implementations probably do not use such a weak memory model so it may be very hard to cause failures of due to unexpected ordering of memory operations. But the language definition requires programs to be written with this level of care to ensure that they work in all language implementations, current and future.

One further remark on Java monitors: be sure to distinguish the monitor corresponding to a Java object (class instance) from the static monitor associated with the class, itself. It's important to keep straight which one you are using to protect what data.

- Synchronized static methods acquire the static lock;
- synchronized non-static methods acquire the per-object lock.
- with synchronized statements you can acquire whatever lock you wish (even one on another object or class).
- beware inheritance: synchronized(getClass()) {...} may well acquire a different static lock than you intended. (If *this* is a member of a subclass, and synchronized(getClass()) ... occurs in a method of a superclass, the static lock acquired is that of the subclass, *not* the superclass).