Concurrency lecture 1

Why a problem? 6.1, 6.2

Software solutions 6.2.1, 6.2.2

Why a problem?

Multiprogramming (multitasking)

several threads/processes running *at the same time*.

using the same resources - accessing the same data structures/objects

Some resources can only be safely used by one thread at a time.

They don't work properly otherwise.

Wasn't a problem before preemptive scheduling.

Multiprocessing makes things even worse.

Race conditions

Any situation where the order of execution of threads can cause different results is known as a *race condition*.

Unless the programmer controls the progression of threads it is impossible to predict the outcome.

Critical sections

Need some way of locking threads out of *critical sections*.

Only one thread is allowed into the critical section at a time.

This is known as mutual exclusion.

Need to ensure:

threads are not kept waiting forever - *starvation*

Starvation can be caused in different ways

Deadlock where a cycle of threads hold locks which other threads in the cycle need

indefinite postponement - priority too low or just unlucky

Software solutions

Not quite as simple as we would like:

We are going to put a simple lock around the critical section of our code.

```
lock
critical  section
unlock
```

Let's say the shared resource used in the critical section is *resource_x* and the boolean lock variable associated with this resource is *lock_x* which has an initial value of false.

```
lock(resource_x);
critical  section  using  resource_x
unlock(resource_x);
```

our first attempt

```
lock(resource_x):
  while  (lock_x)
    no-op;
  lock_x  =  true;

unlock(resource_x):
  lock_x  =  false;
```

no-op just means the thread keeps running - later we will have to put the thread to sleep.

Locks like this are known as *spin-locks* or *busy waits*.

And no, it doesn't work. Why not?

Either a context switch could occur straight after the while test or

the same code could be performed *simultaneously* on a multiprocessor.

A related question

In Java why do wait() and notify() calls have to be inside synchronized methods?

e.g.

```
while (!condition)
  wait();
…
condition = false;
…

…
condition = true;
…
notify();
```

BTW

“Using notify wakes up the one that has been waiting the longest”.

but

“You cannot choose which thread will be notified, so use this form of notify only when you are sure you know which threads are waiting for what at which time”.

> from “The Java Programming Language” by Ken Arnold and James Gosling

Simultaneously

All shared memory multiprocessors that I am aware of don’t allow simultaneous access to the same word of memory. Two writes don’t get interleaved, the hardware allows only one processor access at a time.

Software solutions to locking critical regions require this level of hardware assistance.

Two thread solution based on the textbook - pg 169

There is now an array for each shared resource_x, *flag_x* and a *turn_x*.

```
boolean[] flag_x = new boolean[2];  // both false initially
int turn_x  = 0;

lock: performed by thread i, j is the other thread
  flag_x[i] = true;
  turn_x = j;
  while (flag_x[j] && turn_x = j)
    no-op;

unlock: performed by thread i
  flag_x[i] = false;
```

This works but is severely limited by the fact that it only works for two threads and they need to know about each other.

The general software solution is known as the bakery algorithm - pg 171

Each thread is given a number indicating when it requests the lock.

These are not unique so some other method of ordering e.g. pid is necessary as well.