Concurrency lecture 2

Hardware help 6.3

Semaphores 6.4

Classic problems

Hardware solutions

Test and Set instructions

Or equivalent atomic instructions

> they appear indivisible - once started no other process can interfere until completed

It works like this:

testAndSet(lockVariable)

> returns the current value of the lockVariable
>
> and sets the lockVariable to true

With this our lock can become

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

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

Disabling context switches

Of course if we only had a single processor we can save ourselves a lot of bother simply by disabling context switches. Either by disabling interrupts or setting a flag before we enter the critical section which stops context switching occurring.

We don't do this on multiprocessors because it is too time consuming and there are more elegant solutions.

Getting out of the spin

This spin lock is both simple and works, but not perfectly. Ignoring the wasted processor cycles for a moment we don't have any way of guaranteeing that a process doesn't wait forever for the lock.

How could that happen?

It might just be very unlucky. We really want to ensure some sense of fairness in our OS.

Fairness

Without priorities:

Each thread shouldn't have to wait while another thread gets access to the resource more than once.

Each thread should get access before any other thread which requests it later.

But with priorities:

Threads with higher priorities - should they get prior access to resources? give pros and cons

Makes the priority mechanism more effective.

But can lead to indefinite postponement.

Priority mechanism can still work when selecting next runnable thread.

It is also not usually nice to use spin locks especially if the resource is not going to be available for a while. Instead we prefer to put our threads to sleep.

So we are forced to construct some form of queueing associated with our lock and also a way of putting waiting processes onto this queue and waking them up when it is their turn to move on from the lock.

The advantages are:

no waste of CPU cycles

possibly frees pages for other processes

orderly handling of all waiting threads

know how many threads are waiting for this resource

It is subtle, however, what could go wrong with the following?

```
suspend(resource_x):
  enqueue(thisThread, queue_x);
  reschedule;     // another thread can now run
```

- like yield but the current thread is now waiting rather than runnable

```
awaken(resource_x):
  first = dequeue(queue_x);
  makeRunnable(first);
```

and our lock and unlock are:

```
lock(resource_x):
  if  (testAndSet(lock_x))
    suspend(resource_x);

unlock(resource_x):
  if not empty(queue_x)
    awaken(resource_x);
  else
    lock_x = false;
```

It is possible to get indefinite postponement of unlucky threads, caught between the testAndSet and the suspend.

We need mutual exclusion on the lock variable in order to make the mutual exclusion work.

Actually we can do this with a spin lock because the lock variable is only held for a very short period of time.

Edsger Dijkstra and Semaphores (1965)

A semaphore is an integer count, two indivisible operations and an initialization.

S a semaphore - the indivisible or *atomic* operations are:

```
V(S):
  S = S + 1;
P(S):
  wait until S > 0;
  S = S - 1;
```

The count tells how many of a certain resource are available.

If the semaphore is initialized to 1 this is known as a binary semaphore and works just like a simple lock.

To get a resource the thread calls P on the semaphore.

To return the resource the thread calls V.

Rather than calling the operations P and V (a little hard to remember which is which) we will call them *wait* and *signal*.

To solve the problems we had earlier with spin locks it is common to implement wait and signal like this:

```
signal(S):
  if anything waiting on S then
    start the first process on the S queue
  else
    S = S + 1;
wait(S):
  if S < 1 then
    put this process on the S queue
  else
    S = S - 1;
```

another common alternative is:

```
signal(S):
  S = S + 1;
  if S < 1 then
    start the first process on the S queue;

wait(S):
  S = S - 1;
  if S < 0 then
    put this process on the S queue;
```

S now keeps count of either the number of resources currently available or the number of threads waiting for this resource.

How are the complex semaphore operations made atomic?

One way is with a simple spin lock using a TestAndSet instruction.

Write a Java class which implements a semaphore.

```
public class Semaphore {
  private int count;

  public Semaphore(int count) {
    this.count = count;
  }

  public synchronized void semSignal() {
    count++;
    if (count < 1)      // unnecessary
      notify();
  }
```

```
  public synchronized void semWait() {
    count--;
    if (count < 0)      // not good
      try  {
        wait();
      }
      catch (InterruptedException e) {
        System.err.println("semaphore  wait  interrupted");
      }
  }

}
```

Classic problems

You have already seen the producer/consumer problem.

Lets see how we could solve this with semaphores:

```
program  ProducerConsumerRelationship;

var
  numberDeposited : semaphore;
  numberReceived : semaphore;
  numberBuffer  :  integer;

procedure  producerProcess;
var
  nextResult  :  integer;
begin
  while true do begin
    calculate(nextResult);
    wait(numberReceived);
    numberBuffer := nextResult;
    signal(numberDeposited)
```

```
  end
end;

procedure consumerProcess;
var
  nextResult : integer;
begin
  while true do begin
    wait(numberDeposited);
    nextResult := numberBuffer;
    signal(numberReceived);
    use(nextResult)
  end
end;

begin
  semaphoreInitialize(numberDeposited, ?);
  semaphoreInitialize(numberReceived, ?);
  cobegin
    producerProcess;
    consumerProcess;
  coend
end.
```

What should the semaphores be initialized to?

The cobegin and coend pseudo instructions mean that any statements in here can be executed in parallel. You can think of them as starting up to separate threads.

Rewrite the above as a Java program.