SOLUTION SET **SOLUTION SET**

Spring 2009 CS4410 Prelim 1
This exam has 4 questions, Q1-Q4, on 7 pages. 100 points.

Q1. Banker's Algorithm

(a) [10 points] Briefly describe two drawbacks of the Banker's algorithm for deadlock avoidance.

1. *Users need to estimate their maximum need; if these estimates are too high, the algorithm can be very conservative about granting requests.*

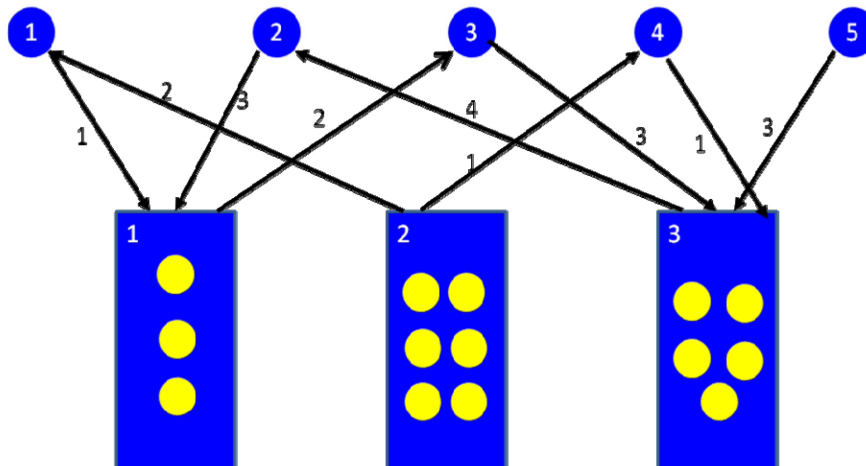
2. *Users need to know what resources they will require, at the start of the execution.*

(b) [10 points] Suppose that in some system, at some point in time, there are *U* units of resource Rj available. Now process P_i requests r<U units of resource R_j. In class we discussed the idea that the Banker's Algorithm can be understood in terms of resource-wait graphs (and saw that this is more intuitive than the matrix representation). What resource-wait graph will the Banker's Algorithm build? How does the concept of graph reduction on this graph relate to the notion of a "safe state"?

The resource-wait graph will show each resource and process in the system, with the number of units represented as circles inside a box representing the corresponding resource. Labeled edges denote resources already assigned, or requested. This graph represents the "current state".

The Banker's Algorithm will now add an edge from Pi to Rj and label it r (the state if the request were granted now). It then computes the graph reduction algorithm on this resource graph. If the graph is reducible, than the BA grants the request now. If not it makes Pi wait until some future state is reached in which the request can safely be granted.

(c) [10 points] For the resource-wait graph shown below, either prove that a deadlock exists or give a reduction order proving that the system is not currently deadlocked.



Answer:

The graph has an irreducible subgraph. We can remove (reduce) P1 and P4, but he requests by P2 and P3 have formed a deadlock (and P5 is stuck too).

Q2. Processes and Threads

(a) [5 points] When a process creates a new process using the fork() operation, which of the following are shared between the parent process and the child process?

i. Stack
ii. Heap
iii. Shared memory segments
iv. Page table

The parent and child share only the shared memory segments. Everything else is duplicated, but the child has its own versions (changes it makes won't be visible to the parent, and vice-versa).

(b) [5 points] Which of the following components of program state are shared across threads in a multithreaded process?

i. Register values
i. Heap memory
ii. Global variables
iii. Stack memory
iv. Page table

Threads share the heap, global memory and the page table. They have private register values and private stack segments.

Q3. Process Synchronization

We learned that TestAndSet instructions are often used to implement CSEnter and CSExit, and in particular that some real systems use the following code:

```
Boolean         mutex = FALSE;

public static void CSEnter(int threadid)
{
        while(TestAndSet(mutex) == TRUE)
                continue;
}

Public static void CSExit(int threadid)
{
        Mutex = FALSE;
}
```

(a) [10 points] What would happen if a recursive procedure called this version of CSEnter()/CSExit()? Assume that the call to CSEnter() is the first thing done and it occurs at every level of the recursion, and that the call to CSExit() is always called right before returning.

The code would go into an infinite loop on the second call (the recursion call) to CSEnter. This is because mutex will be TRUE as a result of the prior call.

(b) [10 points] Write a version of CSEnter/CSExit for which recursion will work correctly.

```
Boolean        mutex = FALSE;
int            cur_thread_id = 0, recursion_cnt = 0;

public static void CSEnter(int threadid)
{
        while(TestAndSet(mutex) == TRUE && cur_thread_id != threadid)
                continue;
        cur_thread_id = threaded;
        ++recursion_cnt;
}

Public static void CSExit(int threadid)
{
        If(--recursion_cnt > 0)
                return;
        cur_thread_id = 0;        // Must occur BEFORE setting mutex = FALSE;
        mutex = FALSE;
}
```

(c) [10 points] Suppose that you had a choice between implementing CSEnter/CSExit using TestAndSet (as above) or the Bakery Algorithm on a true multicore platform. Which would you pick and why?

I would use the TestAndSet solution. We know that the CPU vendor designed this instruction for this kind of synchronization and will have implemented it in a way that can't be fooled by caching or prefetching/instruction-reordering/pipelining. Bakery Algorithm is a nice tool for thinking about concurrency but could generate incorrect code on hardware with those kinds of features.

(d) [10 points] With the advent of multicore hardware, many multithreaded programs are suddenly being run on true parallel hardware, for the first time ever. There are a lot of reports of programs that *slow down* when given two cores, relative to how they run on one core – with no change at all to the code. Give some reasons that could explain such an experience.

With true multicore platforms, lock contention can be a big source of overhead. To get the full benefit of multithreading you often need to tune your code carefully.

Multicore systems can also have issues with caching (e.g. if two threads touch the same parts of memory), or with the TLB (if two threads use the same page table).

(e) [10 points] Here's the correct code for ReadersAndWriters synchronization in Java:

```
public class ReadersNWriters {
   int NReaders = 0, NWriters = 0;
   Object CanBegin = new Object();

   public synchronized void BeginWrite()
   {
      synchronized(CanBegin) {
         if(NWriters == 1 || NReaders > 0)
            CanBegin.Wait();
         NWriters = 1;
      }
   }
   public void EndWrite()
   {
      synchronized(CanBegin) {
         NWriters = 0;
         CanBegin.Notify();
      }
   }

   public synchronized void BeginRead()
   {
      synchronized(CanBegin) {
         if(NWriters == 1)
            CanBegin.Wait();
         ++NReaders;
      }
   }

   public void EndRead()
   {
      synchronized(CanBegin) {
         if(--NReaders == 0)
            CanBegin.Notify();
      }
   }

}
```

Notice that the EndRead and EndWrite methods don't have the synchronized keyword on them. Would this code still be correct if these methods were modified (e.g. public void **synchronized** EndRead())? If so, explain why; if not, explain what goes wrong.

The code would deadlock. As written, the "synchronized" keyword on the BeginRead and BeginWrite methods were being used to ensure that at most a single thread is in one of those two procedures. That thread could block on the wait operation, waiting for someone to call EndRead or EndWrite. But when it blocks, it won't release the synchronized lock, so if we also synchronize EndRead and EndWrite, threads currently reading or writing can't exit the critical section.

Q4. [10 points] List the four necessary conditions for deadlock

1. *Mutual Exclusion.*

2. *Hold and Wait*

3. *No preemption*

4. *Cyclic wait*