CS 372 — Fall 1996	**Midterm #1 Solution**
Introduction to Operating Systems	**October 23, 1996**

Problem 1 (10 pts)

For each of the following programs, list values possible for x if the program terminates. If the program terminates sometimes but not always, state it, and list the values for the cases in which the program terminates.. And, if the program never terminates, state it.

You may assume that variables are initialized as follows:

x: integer init 0

y: integer init 23

m: general semaphore, init 1

ml: general semaphore, init 0

mx: general semaphore, init 1

1. **cobegin** $x := x + 1 \parallel x := y + 1$ **coend**

   A: 1, 24, 25

2. **cobegin**
   $P(m); x := x + 1; V(m)$
   $\parallel$
   $P(m); x := y + 1; V(m)$
   coend

   A: 24, 25

3. **cobegin**
   $P(ml); x := x + 1; V(m)$
   $\parallel$
   $P(m); x := y + 1; V(ml)$
   coend

   A: 25

4. **cobegin**
   $V(ml); x := x + 1; P(m)$
   $\parallel$
   $V(ml); x := y + 1; P(ml)$
   coend

   A: 1, 24, 25

5. **cobegin**
$P(mx); P(m); x := x + 1; V(mx); V(m)$
$\|$
$P(m); P(mx); x := y + 1; V(m); V(mx)$
coend

A: Sometimes deadlocks
<u>24, 25</u>

Problem 2 (12 pts)

Simulate the traffic of two intersecting one-way streets using semaphore operations. In particular, the following rules should be satisfied:

- Only one car can be crossing the intersection at any time.
- When a car reaches the intersection of two one-way streets and there are no cars approaching from the other street, it should be allowed to cross.
- When cars are arriving from both directions, they should alternate to prevent infinite postponement in either direction.

You should give the implementation of two procedures, *EnterIntersection* and *ExitIntersection*, that take the direction as a parameter. These procedures would be called by *car processes* when attempting to enter the intersection and when exiting the intersection. Be sure to state what kind of semaphore you are using (binary or general) and the fairness property you are assuming (i.e. how do the semaphores choose which blocked process gets unblocked as a result of a V operation).

Answer: Let the variables *nNS* and *nEW* represent the number of cars going north-south and east-west respectively waiting to cross the intersection and *in* be the number of cars in the intersection. Cars wait on the semaphore *cNS* and *cEW* depending on their direction of travel. Both of these are general semaphores. In fact, to model a physical system they must be strictly FIFO queues. The semaphore *mutex* can be a binary semaphore.

```
type Dir = {NS, EW};
var nNS, nEW, in: int ← 0;
var cNS, cEW: semaphore ← 0,0;
var mutex: binary semaphore ← 1;

procedure EnterIntersection[d:Dir]
    begin
        P_b(mutex);
        if in = 0 then
            in ← 1;
            V_b(mutex)
        else if in > 0 then
            if d = NS then
                nNS ← nNS + 1;
                V_b(mutex);
                P(cNS)
            else if d = EW then
                nEW ← nEW + 1;
                V_b(mutex);
                P(cEW)
            fi
        fi
    end

    procedure ExitIntersection[d:Dir]
```

```
begin
    P_b(mutex);
    if d = NS then
        if nEW > 0 then
            nEW leftarrow nEW - 1;
            V(cEW)
        else if nEW = 0 and nNS > 0 then
            nNS ← nNS - 1;
            V(cNS)
        else if nEW = 0 and nNS =0 then
            in ← 0;
        fi
    else if d = EW then
        if nNS > 0 then
            nNS leftarrow nNS - 1;
            V(cNS)
        else if nNS = 0 and nEW > 0 then
            nEW ← nEW - 1;
            V(cEW)
        else if nEW = 0 and nNS =0 then
            in ← 0;
        fi
    fi
end
```

Problem 3 (9 pts)

Consider the following four potential sources of overhead in a context switch.

A. Procedure call and return.

B. Trap to operating system and return.

C. Save and restore general-purpose registers.

D. Save and restore memory-mapping context.

For each of the following context switches, list all of the above four sources of overhead that apply.

1. Switch from one user-level thread to another user-level thread in the same process.

   Answer: A, C.

2. Switch from one kernel thread to another kernel-thread in the same process.

   Answer: (A), B, C.

3. Switch from one kernel thread to another kernel-thread in a different process.

   Answer: (A), B, C, D.

Problem 4 (15 pts)

1. (5 pts) What is the difference between a binary semaphore and a condition variable (as in a monitor)? Describe a scenario in which the two behave differently.

   Answer: A binary semaphore has memory, while a condition variable hasn't. Here is a scenario that shows the difference:

   (a) Let s be a binary semaphore. If process p executes $V_b(s)$ followed by $P_b(s)$, p does not block.

   (b) Let c be a condition variable. If process p executes c.signal followed by c.wait, p blocks.

2. (10 pts) Consider the following implementation of the general semaphore operations using binary semaphores:

```
count: integer := 0;
s: integer := {initial value of the semaphore};
mutex: binary semaphore := 1;
block: binary semaphore := 0;

P(s):  Pb(mutex);
       if s = 0 then
           count := count + 1;
           Vb(mutex);
           Pb(block)
       else /* s > 0 */
           s := s - 1;
           Vb(mutex);
       fi

V(s):  Pb(mutex);
       if count = 0 then
           s:= s + 1;
           Vb(mutex);
       else /* count > 0 */
           count := count - 1;
           Vb(block);
           Vb(mutex);
       fi
```

   This implementation suffers from a liveness problem. Trace an execution which ends with $s > 0$, *count* $= 0$ and a process waiting on the semaphore *block*. Why is this a liveness problem?

   Answer: The best way to attack a problem like this is to observe if it is possible for two or more $\mathbf{V}_b$ operations to be done on a semaphore before the corresponding two $\mathbf{P}_b$ operations are done. This cannot occur with *mutex*, for example, since all $\mathbf{P}_b$(mutex) and $\mathbf{V}_b$(mutex) alternate. The problem *can* occur with *block*.

   Consider s initially zero. Let one process execute a $\mathbf{P}$(s) to the point just after the V_b(mutex) and before P_b(block), so *count* $= 1$. Let another process do the same so *count* $= 2$ with s still zero.

   Let a process do two $\mathbf{V}$ operations, so s is still zero, *count* is zero and *block* is 1.

   Let the first process run. It will set *block* to zero and leave the $\mathbf{P}$ implementation. Let the second process run. It will block on $\mathbf{P}_b$(block) with *count* and s zero. Let the first process do a $\mathbf{V}$, and *count* will remain zero with $s = 1$ and the second process is still blocked.

This is a liveness problem. The second process will never terminate the **P** operation even given an unbounded number of **V** operations.

Problem 5 (7 pts)

1. (2 pts) What are the 4 necessary conditions for deadlock?

   Answer:

   (a) hold and wait

   (b) circular wait

   (c) mutual exclusion

   (d) no preemption

2. (5 pts) On the island of *Deadlokia*, all streets are one-way and have two lanes. Some streets are one-way from north to south, and all others are one-way from east to west. The only way to reach Deadlokia is via a one-way bridge, which deposits vehicles in the northeast corner of the island on a north-south street. The only way to leave Deadlokia is via a one way bridge that is reached from a north-south street in the southwest corner of the island.

   Deadlokia is scenic but extremely dangerous for tourists. Therefore, tourists never stop and leave their vehicles—they simply drive onto the island, drive around on the island (obeying the directionality of the one-way street and bridges), and drive off the island.

   Traffic on Deadlokia is chaotic, but Deadlokians like it as it is. They even refused in a referendum to adopt the following law:

   > "It is illegal for a vehicle to enter an intersection unless a way exists for that vehicle to exit the intersection."

   claiming that it would take creativity out of their driving experience.

   A poll taken by the Deadlokian Tourist Office shows that visitors to the island live in constant fear of a traffic deadlock (a *gridlock*), and you have been hired to study the situation. Is gridlock possible on Deadlokia? If so, explain how one might arise. If not, indicate which of the 4 necessary conditions for deadlock does not hold, and briefly explain why.

   Answer Gridlock is not possible on Deadlokia, since the pattern of one way streets guarantees that the "circular waiting" condition does not hold.

Problem 6 (8 pts)

- True or false. A thread cannot read the stack of another thread, even if that other thread is in the same process.

  Answer: *False*. It may be argued that a thread should not read the stack of another process, but there is nothing that prevents it from doing so.

- True or false. An operating system that does not support multithreading has no use for semaphores.

  Answer: *False*. HOCA is an example of an operating system that does not support multithreading but uses semaphores

- True or false. There are problems that can be solved with a monitor, but not with a binary semaphore.

  Answer: *False*. As we have seen in class, a monitor can be implemented using general semaphores, and a general semaphore can be implemented using binary semaphores. Hence, any problem that can be solved using monitors can be solved using binary semaphores.

- True or false. It never makes sense to use busy-waiting to implement a mutex on a uniprocessor. Briefly explain why.

  Answer: *True*. There is no reason why a process should be wasting busy-waiting cycles that could be used by another process. I also accepted the answer *False*, however, if accompanied by a reasonable argument, such as: "It makes sense to use busy waiting if the cost of busy waiting is less than the cost of context switch". In general, however, determining if the cost of busy waiting is less than the cost of a context switch may be impossible.

Problem 7 (9 pts)

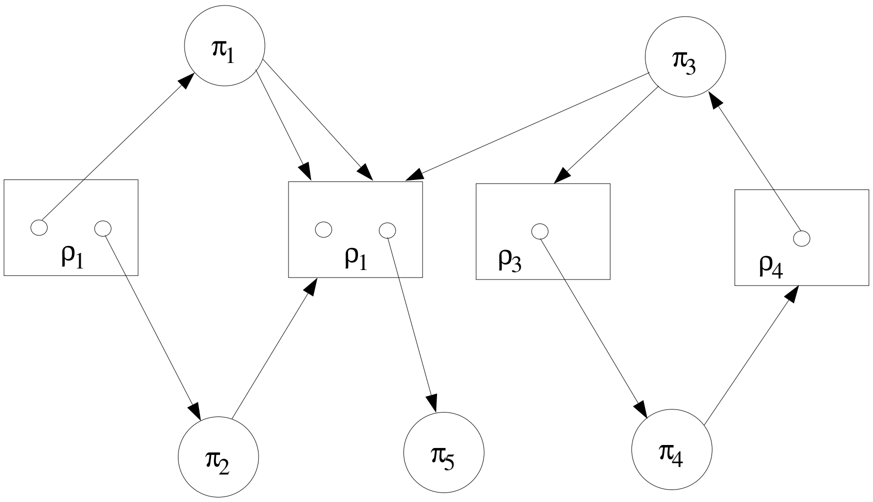

Figure 1: A reusable resource graph.

Consider the resource graph in Figure 1.

1. Which processes are blocked?

   Answer: A process π_i is blocked if for all ρ_j:

   $$|(\pi_i, \rho_j)| + \sum_k |(\rho_j, \pi_k)| > t_j$$

   where t_j is the number of resources in ρ_j. Hence the blocked processes are π_1, π_3, and π_4.

2. Which processes are deadlocked?

   Answer: To determine which processes are deadlock, it is sufficient to perform reduction steps on the given graph until no reductions are possible. All non-isolated nodes correspond to deadlocked processes. In this example, π_3 and π_4 are deadlocked.

3. Is the state a deadlock state?

   Answer: Yes, since there are deadlocked processes.

If you are using reductions to determine your answers, show your intermediate steps.