ECS150 Winter 2004
Homework Assignment 3
Due: Wednesday, 10 March 2004, 4:30PM , HW box, EUII

Question 1: Tanenbaum, Chapter 2-35

Question 2: Tanenbaum, Chapter 2-36

Question 3: Jurassic Park consists of a dinosaur museum and a park for safari riding. There are m passengers and n single passenger cars. Passengers wander around the museum for a while, then line up to take a ride in a safari car. When a car is available, it loads the one passenger it can hold and rides around the park for a random amount of time. If the n cars are all out riding passengers around, then a passenger who wants to ride waits; if a car is ready to load but there are no waiting passengers, then the car waits. Use semaphores to synchronize the m passenger processes and the n car processes.

Here is skeleton code; it is flawed:

```
Car-avail: semaphore := 0
Car-taken: semaphore := 0
Car-filled: semaphore := 0
Passenger-released : semaphore := 0
```

Passenger process code

```
Co-begin(I := 1 to num-passengers)
Do true ->
#wander around museum until ready to take a ride
P(car-avail);
V(car-taken);
P(passenger-released)
Od
Co-end
```

Car process code

```
Do true ->
V(car-avail)
P(car-taken)
V(car-filled)


Travel around the park until ride is done
V(passenger-released
Od
Co-end
```

Find the flaw. Also show a timing diagram that indicates the sequence of actions for several interleavings.

Question 4: Consider the following third variation to the readers and writers problem.

Symmetric version: When a reader is active, new readers may start immediately. When a writer finishes, a new writer has priority, if one is waiting. In other words, once we have started reading, we keep reading until there are no readers left. Similarly, once we have started writing, all pending writers are allowed to run.

Show a solution using monitors. Show a timing diagram. Is it possible for a reader (or writer) to starve?

Question 5: Write a monitor that implements an “alarm clock” which enables a calling process to delay itself for a specified number of time units (ticks). You may assume the existence of a real hardware clock, which invokes a procedure “tick”, that is part of your monitor, at regular intervals. Your monitor will contain some global variables and some conditions (for you to decide on) and two procedures: “delay” (called by a process to set the alarm AND possibly block itself), and “tick” (called by the hardware each time it generates a new tick); you can consider the hardware} to be a process.

Part a: Assume there is only one process (in addition to the hardware), and delay has a single argument:
delay(integer n). Write the code for the process (yes, this is intended to be simple)
and for the monitor (the major effort will be in writing the code for the procedures
delay and tick after you
declare the globals and conditions).

Part b: Assume a fixed number of processes, “nprocs”, any of which can set the alarm at any time;
thus delay will have two arguments: delay(process p, integer n); for convenience the process type is really integer.
Note: This case can be tricky. Remember that
a process calling signal must have this action
be the last thing it does before exiting the monitor.
And, you will want to wake up all processes whose sleeping time is up at the occurrence of a tick,
but within a procedure invocation (say “tick”) only one process can be signaled at a time.

Question 6: Tanenbaum, Chapter 3, Number 21

Question 7: Tanenbaum, Chapter 4, Number 2

Question 8: Tanenbaum, Chapter 4, Number 4

Question 9: Tanenbaum, Chapter 4, Number 7

Question 10: Tanenbaum, Chapter 4, Number 8

Question 11: Tanenbaum, Chapter 4, Number 10

Question 12:

This question concerns the *Buddy System,* an approach to partition main memory for the purpose of allocating contiguous areas of main memory to processes. In a buddy system, memory blocks are available of size 2^K, L <= K <= U, where

- 2^U = smallest block that is allocated
- 2^U = largest block that is allocated; generally, 2^U is the size of the entire memory available for allocation.

To begin, the entire space available for allocation is treated as a single block of size 2^U. If a request of size s such that $2^{U-1} < s \le 2^U$ is made, then the entire block is allocated. Otherwise, the block is split into two equal buddies of size 2^{U-1}. If $2^{U-2} < s \le 2^{U-1}$, then the request is allocated to one of the two buddies. Otherwise, one of the buddies is split in half again. This process continues until the smallest block greater than or equal to s is generated and allocated to the request. At any time, the buddy system maintains a list of holes (unallocated blocks) of each size 2^i. A hole may be removed from the (i+1) list by splitting it in half to create two buddies of size in the I list. Whenever a pair of buddies on the I list both become unallocated, they are removed from the list and coalesced into a single block on the (i+1) list. Here is a question:

A megabyte block of memory is allocated using the buddy system. Show the results of the following sequence in a figure that shows the block of memory allocated at each step, the unallocated blocks, and the buddies:

A: request 70k
B: request 35k
C: request 80k
release A
D: request 60k
Release B
Release D
Release C

Question 13: . Here is a question about low level synchronization. Instead of an atomic test-and-set instruction, a computer could provide an atomic instruction TestAndInc which sets the new value of a parameter to one greater than its old value, as define below:

```
atomic function TestAndInc(var lock: integer) : integer
```

```
begin
  TestAndInc := lock;
  Lock++;
End
```

a) A student from Berkeley uses TestAndInc to implement the critical section problem as follows:

```
Var L : lock := 0 #initialization of L
Cobegin(i:= 0 to N)
   #process 1


   #process i
   while TestAndInc (L) > 0
      do null;
   critical section for process i
   L:= 0;

   #remainder of processes
Coend
```

a) Does this solution provide mutual exclusion? Explain.

b) Is it possible for L to overflow: Explain. How might this impact your answer to a), depending on your assumptions on the effect of overflow?

c) Is deadlock possible? Explain.
d) A student from Davis comes up with a better solution. Here is a sketch of it:

```
var L : lock := 0 #initialization of L
Cobegin(i:= 0 to N)
   #process i
   while TestAndInc (L) > 0
      do op(L);
   critical section for process I
   op(L);
Coend
```

What would you pick for the definition of op to provide mutual exclusion and to eliminate the possibility of overflow?

Question 14: Consider the following algorithm, called algorithm X, which might or might not be a solution to the mutual exclusion problem. In evaluating it (unless otherwise stated) make the same assumptions discussed for Petersen's algorithm: all instructions are atomic, but there can be a context switch between processes subsequent to any instruction execution.

```
var flag: array}[0..1]  of} boolean, initially false
var} turn := 0 (must be 0 or 1)

 cobegin(i:= 0  to 1)
Process i
 do true ->
```

```
 negotiate to enter critical section
  flag[i] := true
  while turn not = i
       do begin
               while flag[1-i]  do skip
               turn := i
          end

critical section for process i


leave critical section
flag[i] := false

rest of process i


od
```

A. This part is easy. Assume Process 1 successfully executes the code following "negotiate to enter critical section" and is in its critical section. Now Process 2 comes along and executes its code "negotiate to enter critical section." Show that Process 2 will not get into its critical section. Show that when Process 1 completes the code following "leave critical section" Process 2 will get into its critical section.

B. This part is more difficult. Determine the correctness of algorithm X as a "solution" to the critical section problem. If it is incorrect, show an example of instruction interleaving (what I have been calling a timing chart) where mutual exclusion is not achieved. If it is a correct solution, argue for its correctness. Is deadlock possible for this algorithm? Explain.