

ECES 338 Midterm #1

Name: _____

Spring'2000
Ozsoyoglu, G.

Recitation Section: _____

The exam is worth 100 points.
Answer all four questions. Question 4 requires more thinking.

(30 points) **1)** Consider the following two-process synchronization algorithm discussed in the class. Note that we have two processes P_i and P_j where $i=1-j$, and $0 \leq i, j \leq 1$.

```
var flag : array [0..1] of boolean; // Initially, both are false.
      turn: 0..1; // Initially, turn is 0.

Pi: repeat
    flag[i] := true;
    (*)----- turn := i;
    (**)----- while (flag[j] and turn=i) do no-op;
                Critical section
                flag[i] := false;
                Remainder section
    until false;
```

This code is different from the original “correct solution” in lines (*) and (**). The original solution had “turn := j;” in (*), and “turn = j” in (**).

(a) Is the mutual exclusion property satisfied? Explain why or why not. If your answer is negative, give a scenario of execution that violates the mutual exclusion property.

(b) Is the progress property satisfied? Explain why or why not. If your answer is negative, give a scenario of execution that violates the progress property.

(c) Is the bounded waiting property satisfied? Explain why or why not. If your answer is negative, give a scenario of execution that violates the bounded waiting property.

(d) Now assume that we change statement (*) above back into the original version, i.e., into “turn := j;”. Now, with this change, does our algorithm satisfy the mutual exclusion property? Explain why or why not. If your answer is negative, give a scenario of execution that violates the mutual exclusion property.

(20 points) 2) Consider the Bakery algorithm given below.

```

Pi: repeat
(**)---choosing[i] := true;
      number[i] := max (number[0], number[1], ..., number[n-1]) +1;
      choosing[i] := false;
      for j:=0 to n-1 do begin
        while choosing[j] do skip;
          while number[j] ≠ 0
(*)-----and (number[j], j) < (number[i], i) do skip;
      end;
      Critical Section
      number[i] := 0;
      Remainder Section
until false;

```

Initially, all choosing values are false, and all number values are zeroes.

(a) Assume that the line (*) is replaced by

```
and (number[j] < number[i]) do skip;
```

Describe a scenario in which the algorithm does not work (i.e., it does not solve the critical section problem). Briefly, explain what goes wrong and why.

(b) Assume that the line (**) is removed. Describe a scenario in which the algorithm does not work (i.e., it does not solve the critical section problem). Briefly, explain what goes wrong and why.

(30 points) 3) Consider the multiple semaphore problem of assignment #3: A multiple semaphore allows the wait and signal primitives to operate on several semaphores simultaneously. It is useful for acquiring and releasing several resources in one atomic operation. Thus, the semantics of **atomic** wait and signal primitives (for two semaphores S and R) can be defined as follows:

```

wait ( $S,R$ ):  while ( $S \leq 0$  or  $R \leq 0$ ) do no-op;
                 $S := S - 1$ ;
                 $R := R - 1$ ;

signal ( $S,R$ ):  $S := S + 1$ ;
                 $R := R + 1$ ;

```

and the following solution (given to you at the web page):

Variables:

```

int cs = 0,  cr = 0,  csr = 0;
nonbinary semaphore S = 0,  R = 0,  SR = 0;

```

WAIT(S,R):

```

{ wait(mutex);
  if (cs < 0 or cr < 0 or csr < 1)
  { // This process must wait.
    csr = csr + 1;
    signal(mutex);
    wait(SR); -----(*)
  }
  else // This process can go into its CS.
  {
    csr = csr + 1;
    signal(mutex);
  }
}

```

SIGNAL(S,R):

```

{
  wait(mutex);
  csr = csr - 1;
  if (cs < 0)
  { signal(S);
    signal(R);
  }
  else signal(S);
  else if (cr < 0)
    signal(R);
  else if (csr < 0)
    signal(SR);
  signal(mutex);
}

```

WAIT(S):

```

{
  wait(mutex);
  cs = cs + 1;
  if (cs < 1 or csr < 0) -----(**)
  { // This process must wait.
    signal(mutex);
    wait(S);
  }
  else // This process can go into its CS.
    signal(mutex);
}

```

SIGNAL(S):

```

{
  wait(mutex);
  cs := cs - 1;
  if (cs < 0)
    signal(S);
  else if (cr = 0 and csr < 0)
    signal(SR);
  signal(mutex);
}

```

- (a) Assume that after the initialization, processes P1, P2, and P3 issue WAIT (S,R), WAIT (S,R), and WAIT(S), in the given time order. As a result, one process succeeds (proceeds), and two are blocked. State what happens in time order, and explicitly specify where each of the blocked processes is blocked.
- (b) Assume that the events described in part (a) occur, and two processes get blocked, one, say P_x, where $1 \leq x \leq 3$, succeeds (proceeds). Later, P_x issues a SIGNAL(S,R)/SIGNAL(S). One more of the blocked processes proceeds with its execution. Which one? Describe what happens.
- (c) What goes wrong if the line (*) is removed? Explain with a scenario.
- (d) What goes wrong if the line (**) is replaced by “if (cs <> 0 or csr <> 0)”? Explain with a scenario.

(20 points) 4) Semaphore primitives **wait** and **signal** are atomic operations, i.e., CPU cannot be interrupted when it is executing a semaphore primitive.

(a) Explain what goes wrong if the atomicity requirement is not followed for the **signal** primitive of non-blocking, binary semaphores (where **signal(S)** is defined as $S := S + 1$;).

(b) Explain what goes wrong if the atomicity requirement is not followed for the **signal(S)** primitive of blocking, non-binary semaphores (given below).

```
signal(S): S.value := S.value + 1;  
    if S.value ≤ 0  
        then begin  
            Remove a process P from S.L;  
            Wakeup(P);  
        end;
```

This page is intentionally left blank.