

(10 pts) (2) Assume that we have the following shared record specification:

```
var V shared record bool A:=False; bool B:=False; bool C:= False; bool D:= False; endrecord
```

Consider the following three separate code fragments, and assume that each of the processes P1, P2 and P3 execute, in the given order, the corresponding critical region statements:

```
P1 → region V when (A and B) do begin A:=True; B:=False; C:= False; D:=True end;  
.....
```

```
P2 → region V when (A or C) do begin A:=True; B:= False; C:=True; D:=False end;  
.....
```

```
P3 → region V when (not D) do begin A:=False; B:= True; C:=True; D:=False end;  
.....
```

Describe, in time order, the executions, blockings, and awakenings of each of the processes, and state the final values of each of the three boolean variables A, B and C.

(15 pts) (3) Consider the “one-cycle” Test-and-Set instruction:

```
function Test-and-Set (bool: target)  
  begin  
    Test-and-Set := target;  
    target := True;  
  end;
```

(a) Assume that there are n processes, P_i , $1 \leq i \leq n$, each having the following format:

```
 $P_i$ : repeat  
  Entry section;  
  Critical Section  $CS_i$ ;  
  Exit section;  
until False;
```

Specify the code for entry and exit sections using the Test-and-Set instruction so that the Mutual Exclusion property for all CS_i executions is satisfied.

(b) Does your solution also satisfy the bounded waiting property for all P_i ? Why or why not?

(15 pts) (4) Consider The Four-of-a-Kind Problem: Develop a solution to the four-of-a-kind problem using critical region statements: There is a deck of 24 cards, split into 6 different kinds, 4 cards of each kind. There are 4 players (processes); each player can hold 4 cards. Between each pair of adjacent (i.e., seated next to each other) players, there is a (possibly empty) pile of cards. Each player behaves according to the following program.

```

while ((hand does not contain four of a kind) and (no one has won))
  begin
    Discard a card into the left-hand pile; Pick up a card from the right-hand pile;
  endwhile;
if hand contains four of a kind then claim victory;

```

There are no ties; when a player has claimed victory, all other players stop. The game begins by dealing four cards to each player and putting two cards on the pile between each pair of adjacent players.

The solution below is incorrect in that it has synchronization problems.

```

var MutualExclusion: shared record bool Winner := False; int array [1..4] PILE := null; endrecord;

```

Procedures used:

```

ThrowLeft( int a, int card);           // Add card into the left pile of the player a.
PickupRight (int a, int array MyCards); //Pick a card from the right pile of player a, and
                                         // add it into the local array MyCards of a.
bool EvaluateHand (int a);             // Evaluate the current hand of player a, and returns True/False.
int ChooseCardToThrow (int array MyCards); //Return one of the values in the array MyCards.

```

Player i: ($1 \leq i \leq 4$)

```

begin bool MyFlag := False; int array MyCards [1..4]; int Card;
repeat
  region MutualExclusion when not Winner
    do { Card := ChooseCardToThrow (MyCards); ThrowLeft (i, Card); PickupRight (i, MyCards);
        Winner := EvaluateHand (i, MyCards)};
    if (Winner) then MyFlag := True;
  until Winner;
if (MyFlag) then Print ("The winner is ", i);
endbegin;

```

Locate the problem and explain it with an example execution scenario. Then fix the error by revising the code.

(20 pts) (5) Consider the following monitor implementation with a single condition variable x (discussed in the class):

Initialization:

semaphore mutex := 1; **semaphore** next := 0; **int** next-count := 0; **semaphore** x-sem := 0; **int** x-count := 0;

For mutual exclusion, each procedure P is implemented as:

```
wait (mutex);  
    Body of P;  
if next-count > 0 then signal(next) else signal (mutex);
```

Each x.wait in a monitor procedure is implemented as:

```
x-count := x-count + 1;  
if next-count > 0 then signal (next) else signal (mutex);  
wait (x-sem);  
x-count := x-count - 1;
```

Each x.signal in a monitor procedure is implemented as:

```
if x-count > 0 then begin  
    next-count := next-count + 1;  
    signal (x-sem);  
    wait (next);  
    next-count := next-count - 1;  
end;
```

(a) Assume that process P_i is waiting on x inside a monitor procedure, and process P_j issues x.signal. Then what happens: Does P_i wait until P_j leaves the monitor, or vice versa? Explain your answer.

(b) Assume that our monitor has two condition variables, namely x and y. Revise the above monitor implementation with semaphores (i.e., give implementations for the mutual exclusion, x.wait, x.signal, y.wait, and y.signal).

(20 pts) (6) Consider the searchers-inserters-deleters problem: Three kinds of processes share access to a singly linked list: searchers, inserters, and deleters. Searchers merely examine the list; hence they can execute concurrently with each other. Inserters add new items to the end of the list; insertions must be mutually exclusive to preclude two inserters from inserting new items at about the same time. However, one insert can proceed in parallel with any number of searches. Finally, deleters remove items from anywhere in the list. At most one deleter process can access the list at a time, and deletion must also be mutually exclusive with searches and insertions. You are to specify a monitor to synchronize searcher, inserter and deleter processes.

Consider the solution below:

```

type SID = monitor
int SCount, SWaitCount, IWaitCount, DWaitCount;   bool SFlag, IFlag, DFlag;   cond S, I, D;

procedure entry SEnter();
if (DFlag) then {SWaitCount++; wait.S;}; SCount++; SFlag := True}

procedure entry SExit();
{SCount--; if (SCount = 0) then { SFlag := False; if (IWaitCount ≠ 0) then { IWaitCount--; signal.I}
else if (DWaitCount ≠ 0) then {DWaitCount--; signal.D} } }

procedure entry IEnter();
if (IFlag and DFlag) then { IWaitCount++; wait.I; IFlag := True}

procedure IExit();
{IFlag := False; if (IWaitCount ≠ 0) then { IWaitCount--; signal.I} else if (DWaitCount ≠ 0)
then {DWaitCount--; signal.D} }

procedure entry DEnter();
if (SFlag and IFlag and DFlag) then { DWaitCount++; wait.D }; DFlag := True}

procedure DExit();
{DFlag := False; if (IWaitCount ≠ 0) then { IWaitCount--; signal.I} else if (DWaitCount ≠ 0)
then {DWaitCount--; signal.D} }

begin           // Monitor Initialization code
  SCount := SWaitCount := IWaitCount := DWaitCount := 0;   SFlag := IFlag := Dflag := False;
End           //End of the monitor

```

Searchers use the monitor as follows: (for inserters or deleters, change S into I or D, respectively).

```

var MySID : SID;
....
MySID.SEnter(); S-LIST(Value); //Critical section code; user-supplied.
MySID.SExit();

```

(a) Is there a starvation possibility for
 (i) searchers? Why or why not?

(ii) inserters? Why or why not?

(iii) deleters? Why or why not?

(b) What goes wrong if a searcher process executes the following sequences of statements?
MySID.SExit(); S-LIST(Value); MySID.SEnter();

Explain with a sample scenario.