

# SOLUTION

**Institute of Business Administration  
MIS & CS Department  
Operating Systems, Fall Semester 2003  
BCS IV  
Third Hourly Test  
November 3, 2003**

**Time Allowed: One Hour**

**Total Marks: 100**

## ***Instructions***

- a. Attempt all questions.
  - b. Maximum/Total Marks are 100.
  - c. Time allowed is 1 hour.
  - d. Do NOT write any thing on the Question Paper except your name. Provide your answers on the answer sheet provided for this purpose.
- 

## ***Question 1: State whether True (T) or False(F) (40 Marks)***

1. Simple Paging results in no internal fragmentation. (F)
2. In a uniprocessor machine concurrent processes cannot be overlapped, they can only be interleaved. (T)
3. Semaphore is an operating system level mechanism to provide concurrency. (T)
4. The circular wait condition can be prevented by defining a non-linear ordering of resource types. (F)
5. Three conditions of policy that must be present for a deadlock to be possible are (1) Mutual Exclusion (2) No Preemption and (3) Circular Waiting (F)
6. In a Deadlock Detection Strategy we do not grant an incremental resource request to a process if this allocation might lead to a deadlock. (F)
7. A modified form of paging is used for UNIX kernel memory allocation. (F)
8. A semaphore may be initialized to any negative value. (F)
9. A relative address is a particular example of a physical address in which the address is expressed as a location relative to some known point, usually the beginning of a program. (F)
10. Simple paging is similar to dynamic partitioning. (F)
11. With the use of a special machine instruction to enforce mutual exclusion deadlock is not possible. (F)
12. The smaller the page size, the less the amount of internal fragmentation. (T)
13. If the page size is small then the page fault rate should be low. (T)
14. The signal operation decrements the semaphore value. (F)
15. With demand paging, pages other than the one demanded by the page fault are also brought in. (F)
16. The use of a special machine instruction to enforce mutual exclusion has the disadvantage that it employs busy waiting. (T)
17. The most popular virtual memory scheme is demand paging. (T)
18. I/O Channels are an example of a consumable resource. (F)
19. There is no single effective strategy that can deal with all types of deadlock. (T)

## SOLUTION

20. Deadlock may be a consequence of the mutual exclusion enforcement. (T)

### Question 2:

(15 Marks)

Suppose the page table for the process currently executing on the processor looks like the following. All numbers are decimal, everything is numbered starting from zero, and all addresses are memory byte addresses. The page size is 1024 bytes.

<i>Virtual Page Number</i>	<i>Valid Bit</i>	<i>Reference Bit</i>	<i>Modify bit</i>	<i>Page Frame Number</i>
0	1	1	0	4
1	1	1	1	7
2	0	0	0	-
3	1	0	1	2
4	0	0	0	-
5	1	0	0	0
6	1	1	0	1

What physical addresses, if any, would each of the following virtual addresses correspond to? Do not try to handle page faults, if any.

1. 1552
2. 3221
3. 6499

### Answer:

1. 1552

Virtual Address 1552  $\Rightarrow$  Virtual Page Number =  $\text{floor}(1552 / 1024) = 1$

Virtual Address 1552  $\Rightarrow$  Offset =  $1552 - 1 * 1024 = 528$

From the given Page Table Virtual Page 1  $\rightarrow$  Page Frame 7

Physical Address = Frame Starting Address + Offset =  $7 * 1024 + 528 = 7696$

2. 3221

Virtual Address 3221  $\Rightarrow$  Virtual Page Number =  $\text{floor}(3221 / 1024) = 3$

Virtual Address 3221  $\Rightarrow$  Offset =  $3221 - 3 * 1024 = 149$

From the given Page Table Virtual Page 3  $\rightarrow$  Page Frame 2

Physical Address = Frame Starting Address + Offset =  $2 * 1024 + 149 = 2197$

## SOLUTION

3. 6449

Virtual Address 6449  $\Rightarrow$  Virtual Page Number =  $\text{floor}(6449 / 1024) = 6$

Virtual Address 3221  $\Rightarrow$  Offset =  $6449 - 6 * 1024 = 355$

From the given Page Table Virtual Page 6  $\rightarrow$  Page Frame 1

Physical Address = Frame Starting Address + Offset =  $1 * 1024 + 355 = 1379$

### Question 3:

(15 Marks)

A process has five page frames allocated to it. All the following numbers are decimal, and every thing is numbered starting from zero. The time of last loading of a page into each page frame, the time of last access to the page in each page frame, the virtual page number in each page frame, and the referenced (R) and modified (M) bits for each page frame are as shown in the following table. The times are in clock ticks from the process start time 0 to event.

<i>Virtual Page Number</i>	<i>Page Frame</i>	<i>Time Loaded</i>	<i>Time Referenced</i>	<i>R bit</i>	<i>M bit</i>
2	0	50	148	1	0
1	1	130	142	1	0
0	2	24	144	0	0
3	3	20	140	0	1
5	4	80	130	0	1

A page fault to virtual page 4 has occurred. Which page frame will have its contents replaced for each of the following memory management policies? Explain why in each case.

1. FIFO
2. LRU
3. Clock

### Answer:

1. FIFO

Page Frame 3 will have its contents replaced because it has been loaded for the longest time. It was loaded at 20 ticks before any other Page Frame.

2. LRU

Page Frame 4 will have its contents replaced because it was referenced at 130 ticks. All other Page Frames have been referenced since then.

3. Clock

Page Frame 2 will have its contents replaced because it has both R and M bits set to 0. In the Clock Algorithm the first such frame encountered is the one whose contents are replaced.

# SOLUTION

## Question 4:

(15 Marks)

Consider the following snapshot of a system. There are no current outstanding queued unsatisfied request.

### Available

R1	R2	R3	R4
2	1	0	0

Process	Current Allocation				Maximum Demand				Still Needs			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	0	0	1	2	0	0	1	2				
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	6				
P4	2	3	5	4	4	3	5	6				
P5	0	3	3	2	0	6	5	2				

1. Compute what each process still might request, i.e., Still Needs columns.
2. Is this system currently in a safe state or an unsafe state? Why?
3. If a request from P3 arrives for (0, 1, 0, 0), can that request be safely granted immediately? In what state (deadlocked, safe, unsafe) would immediately granting that whole request leave the system? Which processes, if any, are or may become deadlocked if this whole request is granted immediately?

### Answer:

1. Still Needs columns are filled out in the figure below:

### Available

R1	R2	R3	R4
2	1	0	0

Process	Current Allocation				Maximum Demand				Still Needs			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	0	0	1	2	0	0	1	2	0	0	0	0
P2	2	0	0	0	2	7	5	0	0	7	5	0
P3	0	0	3	4	6	6	5	6	6	6	2	2
P4	2	3	5	4	4	3	5	6	2	0	0	2
P5	0	3	3	2	0	6	5	2	0	3	2	0

## SOLUTION

2. The system is currently in a safe state because P1 can run to completion as it has all the resources it needs. Once P1 completes it releases 1 unit of R3 and 2 units of R4. Now P4 can be granted 2 units of R1 and 2 units of R4 and so it will have all the resources it would need to run to completion. Once P4 completes it releases 4 units of R1, 3 units of R2, 5 units of R3 and 6 units of R4. This allows P5 to complete. Once P5 is complete it releases 6 units of R2, 5 units of R3 and 2 units of R4. This would allow P2 to complete and then P3 can complete at the end. So there is at least one sequence of events that does not lead to a deadlock hence the system is in a safe state. This is further explained below.

P1 can complete straight away because it has all its required resources and therefore needs (0, 0, 0, 0). After P1 completes we have the following situation:

### Available

R1	R2	R3	R4
2	1	1	2

Process	Current Allocation				Maximum Demand				Still Needs			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P2	2	0	0	0	2	7	5	0	0	7	5	0
P3	0	0	3	4	6	6	5	6	6	6	2	2
P4	2	3	5	4	4	3	5	6	2	0	0	2
P5	0	3	3	2	0	6	5	2	0	3	2	0

Now P4 can be allocated its requirement (2, 0, 0, 2) and that would allow it to complete. After P4 completes we have the situation as shown below:

### Available

R1	R2	R3	R4
4	4	6	6

Process	Current Allocation				Maximum Demand				Still Needs			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P2	2	0	0	0	2	7	5	0	0	7	5	0
P3	0	0	3	4	6	6	5	6	6	6	2	2
P5	0	3	3	2	0	6	5	2	0	3	2	0

Now P5 can be allocated its requirement (0, 3, 2, 0) and would be able to complete. When it completes we have the situation as shown below:

### Available

R1	R2	R3	R4
4	9	9	8

## SOLUTION

<i>Process</i>	<i>Current Allocation</i>				<i>Maximum Demand</i>				<i>Still Needs</i>			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P2	2	0	0	0	2	7	5	0	0	7	5	0
P3	0	0	3	4	6	6	5	6	6	6	2	2

Now P2 can be allocated (0, 7, 5, 0) and therefore it can complete and after it completes we have the situation as shown in the figure below:

*Available*

R1	R2	R3	R4
6	9	9	8

<i>Process</i>	<i>Current Allocation</i>				<i>Maximum Demand</i>				<i>Still Needs</i>			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P3	0	0	3	4	6	6	5	6	6	6	2	2

Now P3 can be allocated (6, 6, 2, 2) and so it can also complete.

3. If a request from P3(0, 1, 0, 0) arrives it cannot be safely granted because after the grant of the request the system would be in an unsafe state. This is shown in the figure below:

*Available*

R1	R2	R3	R4
2	0	0	0

<i>Process</i>	<i>Current Allocation</i>				<i>Maximum Demand</i>				<i>Still Needs</i>			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	0	0	1	2	0	0	1	2	0	0	0	0
P2	2	0	0	0	2	7	5	0	0	7	5	0
P3	0	1	3	4	6	6	5	6	6	5	2	2
P4	2	3	5	4	4	3	5	6	2	0	0	2
P5	0	3	3	2	0	6	5	2	0	3	2	0

*Available after allocation to P3 as  
per its request (0, 1, 0, 0)*

R1	R2	R3	R4
2	0	0	0

# SOLUTION

<i>Process</i>	<i>Current Allocation</i>				<i>Maximum Demand</i>				<i>Still Needs</i>			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	0	0	1	2	0	0	1	2	0	0	0	0
P2	2	0	0	0	2	7	5	0	0	7	5	0
P3	0	2	3	4	6	6	5	6	6	4	2	2
P4	2	3	5	4	4	3	5	6	2	0	0	2
P5	0	3	3	2	0	6	5	2	0	3	2	0

*Available after P1 completes*

R1	R2	R3	R4
2	0	1	2

<i>Process</i>	<i>Current Allocation</i>				<i>Maximum Demand</i>				<i>Still Needs</i>			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P2	2	0	0	0	2	7	5	0	0	7	5	0
P3	0	2	3	4	6	6	5	6	6	4	2	2
P4	2	3	5	4	4	3	5	6	2	0	0	2
P5	0	3	3	2	0	6	5	2	0	3	2	0

*Available after P4 completes*

R1	R2	R3	R4
4	3	6	6

<i>Process</i>	<i>Current Allocation</i>				<i>Maximum Demand</i>				<i>Still Needs</i>			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P2	2	0	0	0	2	7	5	0	0	7	5	0
P3	0	2	3	4	6	6	5	6	6	4	2	2
P5	0	3	3	2	0	6	5	2	0	3	2	0

*Available after P5 completes*

R1	R2	R3	R4
4	6	9	6

## SOLUTION

<i>Process</i>	<i>Current Allocation</i>				<i>Maximum Demand</i>				<i>Still Needs</i>			
	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P2	2	0	0	0	2	7	5	0	0	7	5	0
P3	0	2	3	4	6	6	5	6	6	4	2	2

Now P2 and P3 are left and neither can complete because the required resources for each are not available so they are in a deadlock.

### Question 5:

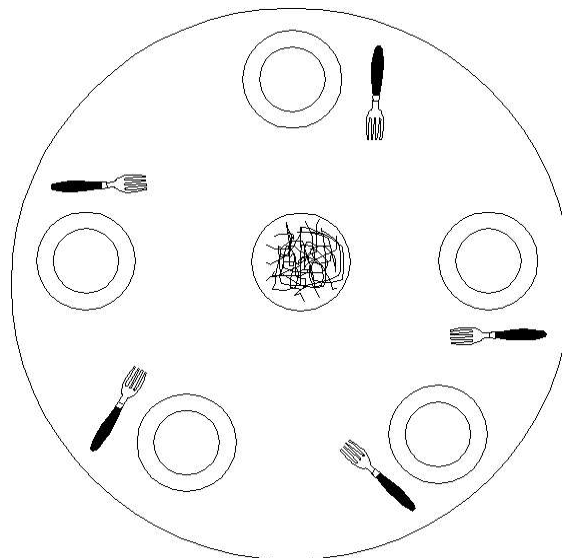
(15 Marks)

Very briefly state the Dining Philosophers' Problem and give one simple solution using semaphores to solve the problem.

### Answer:

This is from the textbook:

There were five philosophers living together. The life of each philosopher consisted principally of thinking and eating, and through years of thought, all of the philosophers had agreed that the only food that contributed to their thinking efforts was spaghetti. The eating arrangement was as shown in the figure below.



The philosophers sat around a round table on which was set a large serving bowl of spaghetti, five plates, one for each philosopher, and five forks. A philosopher wishing to eat would go to his or her assigned place at the table and, using two forks on either side of the plate, take and eat some spaghetti. The problem is this: Devise a ritual (algorithm) that will allow the philosophers to eat. The algorithm must satisfy mutual exclusion (no two philosophers can use the same fork at the same time) while avoiding deadlock and starvation (in this case the term has literal and algorithmic meaning!).

## SOLUTION

One possible solution is to add an attendant who only allows four philosophers at a time into the dining room. With at most four seated philosophers, at least one philosopher will have access to two forks. An algorithm is shown in the figure below:

```
program:    diningphilosophers;
var        fork array[0 ... 4] of semaphore(:=1);
           room: semaphore (:=4);
           i: integer;

procedure philosopher (i: integer)
begin
    repeat
        think;
        wait(room);
        wait(fork[i]);
        wait(fork[(i + 1) mod 5]);
        eat;
        signal(fork[(i + 1) mod 5]);
        signal(fork[i]);
        signal(room);
    forever
end;

begin
    parbegin
        philosopher(0);
        philosopher(1);
        philosopher(2);
        philosopher(3);
        philosopher(4);
    parend
end.
```

-- THE END --