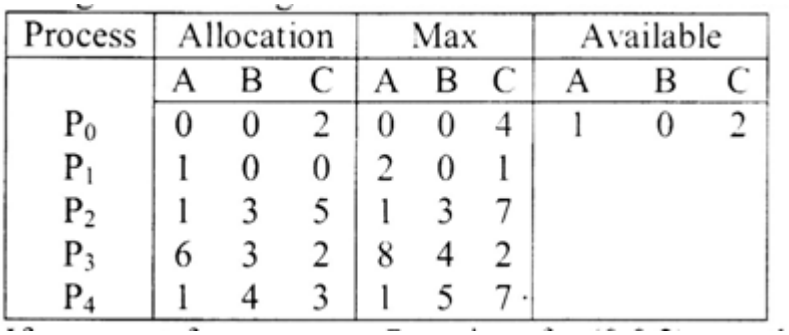
USN | | | | | | | | | | |

Internal Assessment Test 1 – December 2023

Sub:	**Operating Systems**					Sub Code:	**BCS303**	Branch:	**AIML**
Date:	**19/01/24**	Duration:	**90 minutes**	Max Marks:	**50**	Sem/Sec:	**III -A,B,C**		**OBE**

		Answer any FIVE FULL Questions	**MARKS**	**CO**	**RBT**
1		Define Semaphores and Discuss How Readers & Writers Problem can be solved using Semaphores.?	[10]	3	L2
2		Define bankers Algorithm & Using bankers Algorithm determine whether the following system is in safe state?  If a request from process p2 arrives for (0,0,2) can the request granted immediately.?	[10]	3	L3
3	a	Discuss how Dining Philosophers problem solved using Semaphores with code?	[5]	3	L2
	b	Define deadlock. What are the necessary conditions for deadlock to occur?	[5]	3	L1
4		What is Critical section problem? What are the requirements for the solution to critical section problem? Explain Peterson's Solution.	[10]	3	L2
5		What are the principles behind paging? Explain its operation & Structure, clearly indicating their purpose in one line each	[10]	4	L2
6		Explain Segmentation in detail with hardware structures and suitable example?	[10]	4	L1

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	0	2	0	0	4	1	0	2
P_1	1	0	0	2	0	1			
P_2	1	3	5	1	3	7			
P_3	6	3	2	8	4	2			
P_4	1	4	3	1	5	7			

CI **CCI** **HoD**

----------All the Best----------

CO-PO Mapping

Course Outcomes		Modules covered	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3	PSO4
CO1	Describe the Operating System Structure and Services.	**1**	3	-	-	-	-	-	-	-	-	-	-	3	-	2	-	-
CO2	Summarize the Process Management concepts like Processes, Threads, CPU Scheduling, Process Synchronization and Deadlocks	**1, 2**	3	2	2	-	-	-	-	-	-	-	-	3	-	2	-	-
CO3	Interpret the Memory Management concepts with respect to Main Memory and Virtual Memory.	**3, 4**	3	2	2	-	-	-	-	-	-	-	-	3	-	2	-	-
CO4	Discuss the Storage Management concepts like File-System Interface, File-System Implementation and Mass-Storage Structure	**4, 5**	3	2	2	-	-	-	-	-	-	-	-	3	-	2	-	-
CO5	Elucidate the Protection features in Operating System and case study in Linux OS.	**5**	3	2	2	-	-	-	-	-	-	-	-	3	-	2	-	-

1) Define Semaphores and Discuss How Readers & Writers Problem can be solved using Semaphores.? CO-3 L2

SEMAPHORES 5 marks

• A *semaphore* is a synchronization-tool.

• It used to control access to shared-variables so that only one process may at any point in time
change the value of the shared-variable.

• A semaphore(S) is an integer-variable that is accessed only through 2 atomic-operations:
1. wait() and
2. signal().

• wait() is termed P ("to test").
signal() is termed V ("to increment").

• **Definition of wait():**

```
wait(S) {
    while (S <= 0)
      ; // busy wait
    S--;
}
```

Definition of signal():

```
signal(S) {
    S++;
}
```

• When one process modifies the semaphore-value, no other process can simultaneously modify that same semaphore-value.

• Also, in the case of wait(S), following 2 operations must be executed without interruption:
1. Testing of S(S<=0) and
2. Modification of S (S--)

• There are 2 types: 1) Binary Semaphore 2) Counting Semaphore.

1) BINARY SEMAPHORE

• The value of a semaphore can range only between 0 and 1.

• On some systems, binary semaphores are known as *mutex locks*, as they are locks that provide mutual-exclusion.

Solution for Critical-section Problem using Binary Semaphores

➢Binary semaphores can be used to solve the critical-section problem for multiple processes.

➢The 'n' processes share a semaphore *mutex* initialized to 1

```
do {
   wait(mutex);

      // critical section

   signal(mutex);

      // remainder section
} while (TRUE);
```

2) COUNTING SEMAPHORE

• The value of a semaphore can range over an unrestricted domain

Use of Counting Semaphores

➢Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.

➢The semaphore is initialized to the number of resources available.

➢Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).

➢When a process releases a resource, it performs a signal() operation (incrementing the count).

➢When the count for the semaphore goes to 0, all resources are being used.

➢After that, processes that wish to use a resource will block until the count becomes greater than 0.

READERS-WRITERS PROBLEM 5 marks

• A data set is shared among a number of concurrent processes.
• **Readers** are processes which want to only read the database (DB).
Writers are processes which want to update (i.e. to read & write) the DB.
• Problem:

➢Obviously, if 2 readers can access the shared-DB simultaneously without any problems.

➢However, if a writer & other process (either a reader or a writer) access the shared-DB simultaneously, problems may arise.

Solution:

➢The writers must have exclusive access to the shared-DB while writing to the DB.

• **Shared-data**

where,

➢*mutex* is used to ensure mutual-exclusion when the variable *readcount* is updated.

➢*wrt* is common to both reader and writer processes.

wrt is used as a mutual-exclusion semaphore for the writers.
wrt is also used by the first/last reader that enters/exits the critical-section.

➢*readcount* counts no. of processes currently reading the object.

Initialization

mutex = 1, wrt = 1, readcount = 0

Writer Process:

```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

Reader Process:

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

The readers-writers problem and its solutions are used to provide **reader-writer locks** on some systems.

• The mode of lock needs to be specified:

1. read mode

➢When a process wishes to read shared-data, it requests the lock in *read mode*.

2. write mode

➢When a process wishes to modify shared-data, it requests the lock in *write mode*.

• Multiple processes are permitted to concurrently acquire a lock in read mode, but only one process may acquire the lock for writing.
• These locks are most useful in the following situations:
1. In applications where it is easy to identify
→ which processes only read shared-data and
→ which threads only write shared-data.
2. In applications that have more readers than writers.

2) Define bankers Algorithm & Using bankers Algorithm determine whether the following system is in safe state? CO-3- L3 10 Marks

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	0	2	0	0	4	1	0	2
P_1	1	0	0	2	0	1			
P_2	1	3	5	1	3	7			
P_3	6	3	2	8	4	2			
P_4	1	4	3	1	5	7			

If a request from process p2 arrives for (0,0,2) can the request granted immediately.?

Ans:

Solution (i): 2 Marks

• The content of the matrix *Need* is given by
Need = Max - Allocation
• So, the content of Need Matrix is:

	NEED
	A B C
P0	0 0 2
P1	1 0 1
P2	0 0 2
P3	2 1 0
P4	0 1 4

Solution (ii):3 Marks

• Applying the Safety algorithm on the given system,
Step 1: Initialization
Work = Available i.e. Work =1 0 2
……P0……..P1……..P2……..P3……P4…..
Finish = | false | false | false | false | false |
Step 2: For i=0

Finish[P0] = false and Need[P0]<=Work i.e. (0 0 2)<=(1 0 2) □ true

So P0 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P0] =(1 0 2)+(0 0 2)=(1 0 4)
……P0……..P1……..P2……..P3……P4….
Finish = | true | false | false | false | false |
Step 2: For i=1

Finish[P1] = false and Need[P1]<=Work i.e. (1 0 1)<=(1 0 4) □ true

So P1 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P1] =(1 0 4)+(1 0 0)=(2 0 4)
…….P0…….P1……P2……..P3……..P4…
Finish = | true | true | false | false | false |
Step 2: For i=2

Finish[P2] = false and Need[P2]<=Work i.e. (0 0 2)<=(2 0 4) □ true

So P2 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P2] =(2 0 4)+(1 3 5)=(3 3 9)
……P0…….P1……..P2……..P3……P4….
Finish = | true | true | true | false | false |

	Allocation	**Max**	**Available**
	A B C	**A B C**	**A B C**
P0	0 0 2	0 0 4	1 0 2

P1	1 0 0	2 0 1
P2	1 3 5	1 3 7
P3	6 3 2	8 4 2
P4	1 4 3	1 5 7

Need

	A B C
P0	0 0 2
P1	1 0 1
P2	0 0 2
P3	2 1 0
P4	0 1 4

Step 2: For i=3

Finish[P3] = false and Need[P3]<=Work i.e. (2 1 0)<=(3 3 9) □ true

So P3 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P3] = (3 3 9)+(6 3 2)=(9 6 11)
…..P0…….P1……P2……..P3…….P4….
Finish = | true | true | true | true | false |
Step 2: For i=4

Finish[P4] = false and Need[P4]<=Work i.e. (0 1 4)<=(9 6 11) □ true

So P4 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P4] =(9 6 11)+(1 4 3)=(10 10 14)
……P0…….P1……..P2……P3…….P4….
Finish = | true | true | true | true | true |
Step 4: Finish[Pi] = true for 0<=i<=4
Hence, the system is currently in a safe state.
The safe sequence is <P0, P1, P2, P3, P4>.
Conclusion: Yes, the system is currently in a safe state.

Solution (iii): 2 Marks

P2 requests (0 0 2) i.e. Request[P2]=0 0 2

• To decide whether the request is granted, we use Resource Request algorithm.

Step 1: Request[P2]<=Need[P2] i.e. (0 0 2) <= (1 3 7) □ true.

Step 2: Request[P2]<=Available i.e. (0 0 2) <= (1 0 2) □ true.

Step 3: Available = Available – Request[P2] = (1 0 2) - (0 0 2)= (1 0 0)
Allocation[P2] = Allocation[P2] + Request[P2] = (1 3 5) + (0 0 2)= (1 3 7)
Need[P2] = Need[P2] – Request[P2] = (0 0 2) - (0 0 2)= (0 0 0)

• We arrive at the following new system state:

	Allocation	**Max**	**Available**
	A B C	**A B C**	**A B C**
P0	0 0 2	0 0 4	1 0 0
P1	1 0 0	2 0 1	
P2	1 3 7	1 3 7	
P3	6 3 2	8 4 2	
P4	1 4 3	1 5 7	

• The content of the matrix *Need* is given by
Need = Max - Allocation
• So, the content of Need Matrix is:

	NEED
	A B C
P0	0 0 2
P1	1 0 1
P2	0 0 0
P3	2 1 0
P4	0 1 4

Solution (IV): 3 Marks

• To determine whether this new system state is safe, we again execute Safety algorithm.

Step 1: Initialization

Work = Available i.e. Work =2 3 0

....P0…….P1……..P2…….P3…..P4…..

Finish = | false | false | false | false | false |

Step 2: For i=0

Finish[P0] = false and Need[P0]<=Work i.e. (0 0 2)<=(2 3 0) false

So P0 must wait.

Step 2: For i=1

Finish[P1] = false and Need[P1]<=Work i.e. (1 0 1)<=(2 3 0) false

So P1 must wait.

	Allocation	**Max**	**Available**
	A B C	**A B C**	**A B C**
P0	0 0 2	0 0 4	1 0 0
P1	1 0 0	2 0 1	
P2	1 3 7	1 3 7	
P3	6 3 2	8 4 2	
P4	1 4 3	1 5 7	

Need

	A B C
P0	0 0 2
P1	1 0 1
P2	0 0 0
P3	2 1 0
P4	0 1 4

1-11

Step 2: For i=2

Finish[P2] = false and Need[P2]<=Work i.e. (0 0 0)<=(2 3 0) true

So P2 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P2] =(1 0 0)+(1 3 7)=(2 3 7)

.....P0…….P1……..P2…..P3……..P4….

Finish = | false | false | true | false | false |

Step 2: For i=3

Finish[P3] = false and Need[P3]<=Work i.e. (2 1 0)<=(2 3 7) true

So P3 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P3] = (2 3 7)+(6 3 2)=(8 6 9)

....P0……..P1…….P2…….P3……..P4…

Finish = | false | false | true | true | false |

Step 2: For i=4

Finish[P4] = false and Need[P4]<=Work i.e. (0 1 4)<=(8 6 9) true

So P4 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P4] =(8 6 9)+(0 1 4)=(8 7 13)

....P0……..P1…….P2…….P3……..P4…

Finish = | false | false | true | true | true |

Step 2: For i=0

Finish[P0] = false and Need[P0]<=Work i.e. (0 0 2)<=(8 7 13) true

So P0 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P0] =(8 7 13)+(0 0 2)=(8 7 15)

....P0……..P1…….P2…….P3…….P4…

Finish = | true | false | true | true | true |

Step 2: For i=1

Finish[P1] = false and Need[P1]<=Work i.e. (1 0 1)<=(8 7 15) true

So P1 must be kept in safe sequence.

Step 3: Work = Work + Allocation[P1] =(8 7 15)+(1 0 0)=(9 7 15)

....P0……..P1…….P2……P3…….P4…

Finish = | true | true | true | true | true |

Step 4: Finish[Pi] = true for 0<=i<=4

Hence, the system is in a safe state.
The safe sequence is <P2, P3, P4, P0, P1>.

Conclusion: Since the system is in safe sate, the request can be granted.

3) Discuss how Dining Philosophers problem solved using Semaphores with code?CO-3 L2 5 Marks

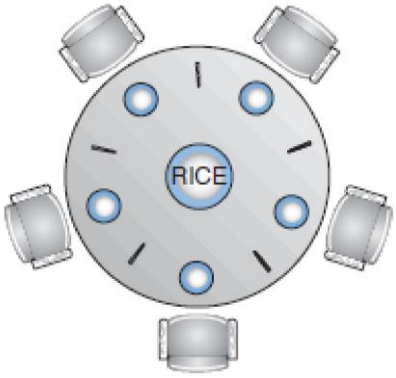
DINING PHILOSOPHERS PROBLEM

Philosophers spend their lives alternating thinking and eating

Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl

Need both to eat, then release both when done

• Problem Objective: To allocate several resources among several processes in a deadlock-free & starvation-free manner



Shared data

- Bowl of rice (data set)
- Semaphore chopstick [5] initialized to 1

The structure of Philosopher *i*:

```
do {
        wait ( chopstick[i] );
            wait ( chopStick[ (i + 1) % 5] );

                // eat

            signal ( chopstick[i] );
            signal (chopstick[ (i + 1) % 5] );

             // think
} while (TRUE);
```

philosopher either thinks or eats

Thinks - when a philosopher thinks, does not interact with his colleagues.

Eats- When a philosopher gets hungry he tries to pick up the two forks that are closest to him(left & right).A philosopher may pick up only one fork at a time.

One cannot pick up a fork that is already in the hand of a neighbour.

when a hungry philosopher has both his forks.when he has finished eating, he puts down both of his forks and starts thinking again.

One simple solution is to represent each fork /chopstick with a semaphore

A philosopher tries to grab a fork/chopstick by executing a wait() operation on that semaphore.

He releases his fork/ chopsticks by executing the signal () operation on the appropriate semaphores.

the shared data are semaphore chopstick[5];//array of 5 chopsticks

where all the elements of chopstick are initailized to 1.

binary semaphore(1 or 0) 1- free, 0-currently being held by other processes.

The structure of Philosopher i:

```
do {
   wait (chopstick[i] );
        wait (chopStick[ (i + 1) % 5] );

            //  eat

        signal (chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

          //  think

} while (TRUE);
```

Although this solution guarantees that no two neighbors are eating simultaneously ,it could still create a deadlock.

suppose that all five philosophers become hungry simultaneously and each grabs their left chopstick will now be equal to 0.

when each philosopher tries to grab his right chopstick, he will be delayed forever.leads to deadlock

some possible remedies to avoid deadlocks:

allow at most four philosophers to be sitting simultaneously at the table.

allow a philosopher to pick up his chopsticks only if both chopsticks are available(to do this he must pick them up in a critical section).

use an asymmetric solution , that is an odd philosopher picks up first his left chopstick and then his right chopstick and then his left chopstick.

Incorrect use of semaphore operations:

signal (mutex) …. wait (mutex)

wait (mutex) … wait (mutex)

Omitting of wait (mutex) or signal (mutex) (or both)

Deadlock and starvation are possible.

3b. Define deadlock. What are the necessary conditions for deadlock to occur? CO-3,L2 5 marks

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.
Waiting for something for infinite time, in which there is no progress for waiting processes.
Processes waits for one another's action indefinitely.
example: you can't get job without experience and experience can only come when you do some job.so we end up in a deadlock.

p1 requires resources r1, & r2. same for p2 also for execution.

System consists of resources
Resource types R1, R2, . . ., Rm
CPU cycles, memory space, I/O devices
Each resource type Ri has Wi instances.
Each process utilizes a resource as follows:
request
use
release
Multiprogramming os- Several processes, finite no of resources.
competing for these number of resources fine, now it's sometime may be one process p1. process->request a resources.if this resource is not available and waiting time leads to no progress.then its deadlock.

```
* thread one runs in this function */  void *do_work_one(void *param)
{
pthread_mutex_lock(&first_mutex);  pthread_mutex_lock(&second_mutex);
/** * Do some work */  pthread_mutex_unlock(&second_mutex);
pthread_mutex_unlock(&first_mutex);  pthread_exit(0);
}
/* thread two runs in this function */  void *do_work_two(void *param)
{
pthread_mutex_lock(&second_mutex);  pthread_mutex_lock(&first_mutex);
/** * Do some work */  pthread_mutex_unlock(&first_mutex);
pthread_mutex_unlock(&second_mutex);  pthread_exit(0);
```

The four necessary conditions for a deadlock are:
(1) Mutual exclusion (2) Hold-and-wait (3) No preemption (4) Circular wait.

• The mutual exclusion condition holds since only one car can occupy a space in the roadway.
• Hold-and-wait occurs where a car holds onto its place in the roadway while it waits to advance in the roadway.
• A car cannot be removed (i.e. preempted) from its position in the roadway.
• Lastly, there is indeed a circular wait as each car is waiting for a subsequent car to advance.
• The circular wait condition is also easily observed from the graphic.

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set

Example

- System has 2 disk drives
- P_1 and P_2 each hold one disk drive and each needs another one
- Example
- semaphores *A* and *B*, initialized to 1 P_0 P_1

wait (A); wait(B) wait (B); wait(A)

- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSes do not prevent or deal with deadlocks

4) What is Critical section problem? What are the requirements for the solution to critical section problem? Explain Peterson's Solution. CO-3 L2 10 Marks

CRITICAL SECTION PROBLEM : 3 Marks

• **Critical Section** is a segment-of-code in which a process may be
→ changing common variables
→ updating a table or
→ writing a file.
• Each process has a critical-section in which the shared-data is accessed.
• General structure of a typical process has following (Figure 3.1):

1. Entry-section

➢Requests permission to enter the critical-section.

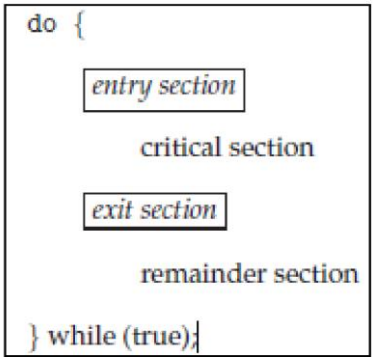
2. Critical-section

➢Mutually exclusive in time i.e. no other process can execute in its critical-section.

3. Exit-section

➢Follows the critical-section.

4. Remainder-section



Problem Statement:

"Ensure that when one process is executing in its critical-section, no other process is to be allowed to execute in its critical-section".

REQUIREMENTS OF SOLUTION TO CRITICAL SECTION PROBLEM : 3 Marks

• A solution to the problem must satisfy the following 3 requirements:

1. Mutual Exclusion

➢Only one process can be in its critical-section.

2. Progress

➢Only processes that are not in their remainder-section can enter their critical-section.

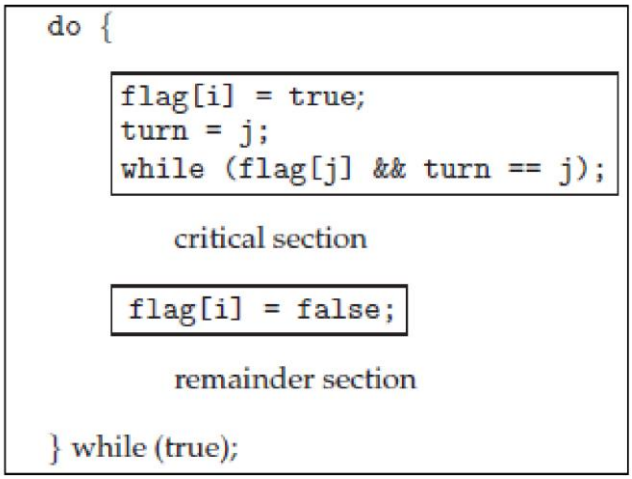
➢The selection of a process cannot be postponed indefinitely.

3. Bounded Waiting

➢There must be a bound on the number of times that other processes are allowed to enter their critical-sections.

PETERSON'S SOLUTION -4 Marks

• This is a classic *software-based solution* to the critical-section problem.
• This is limited to 2 processes.
• The 2 processes alternate execution between
→ critical-sections &
→ remainder-sections.
• The 2 processes share 2 variables (Figure 3.2):
where *turn* = indicates whose turn it is to enter its critical-section.
(i.e., if turn==i, then process Pi is allowed to execute in its critical-section).
flag = used to indicate if a process is ready to enter its critical-section.
(i.e. if flag[i]=true, then Pi is ready to enter its critical-section).



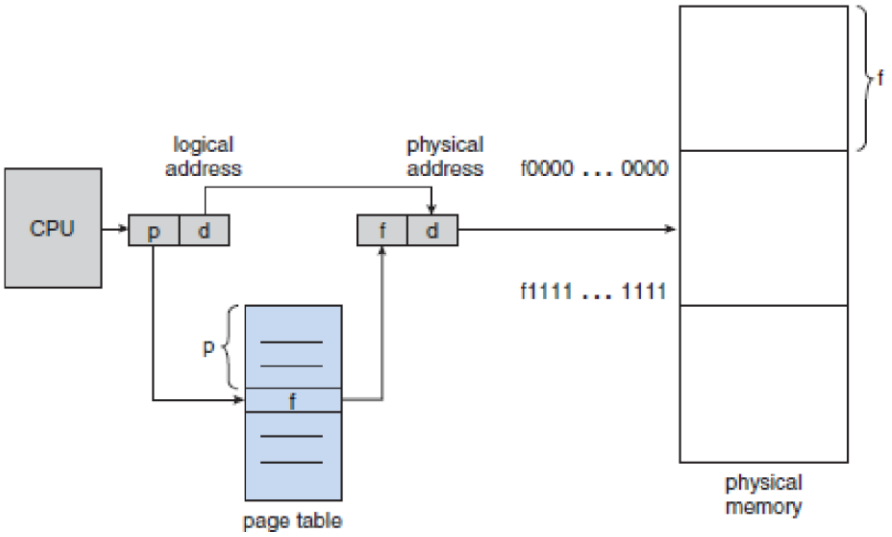
To enter the critical-section,
→ firstly process *Pi* sets *flag[i]* to be true &
→ then sets *turn* to the value j.
• If both processes try to enter at the same time, *turn* will be set to both i and j at roughly the same time.
• The final value of *turn* determines which of the 2 processes is allowed to enter its criticalsection

first.
• To prove that this solution is correct, we show that:
1. Mutual-exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

5) What are the principles behind paging? Explain its operation & Structure, clearly indicating their purpose in one line each.

PAGING -5,diagram-3,Examples -2

• Paging is a memory-management scheme.
• This permits the physical-address space of a process to be non-contiguous.
• Physical-memory is divided into fixed-sized blocks called *frames* (Figure 5.7).
Logical-memory is divided into same-sized blocks called *pages*.
• Any page (from any process) can be placed into any available frame.



A logical address consists of 2 parts:
(1) A page number(p) in which the address resides &
2) An offset(d) from the beginning of that page.

• The page table maps the page number to a frame number, to yield a physical address.
• A physical address consists of 2 parts:
1) The frame number &
2) The offset within that frame.
• If the logical address size is 2^m and the page size is 2^n, then
i) High-order m-n bits of a logical-address designate the page-number and
ii) Low-order n bits designate the page-offset.

Consider the following example, in which a process has 16 bytes of logical memory, mapped in 4 byte pages into 32 bytes of physical memory (Figure 5.9).

➢We have 4 pages and each page is 4 bytes.

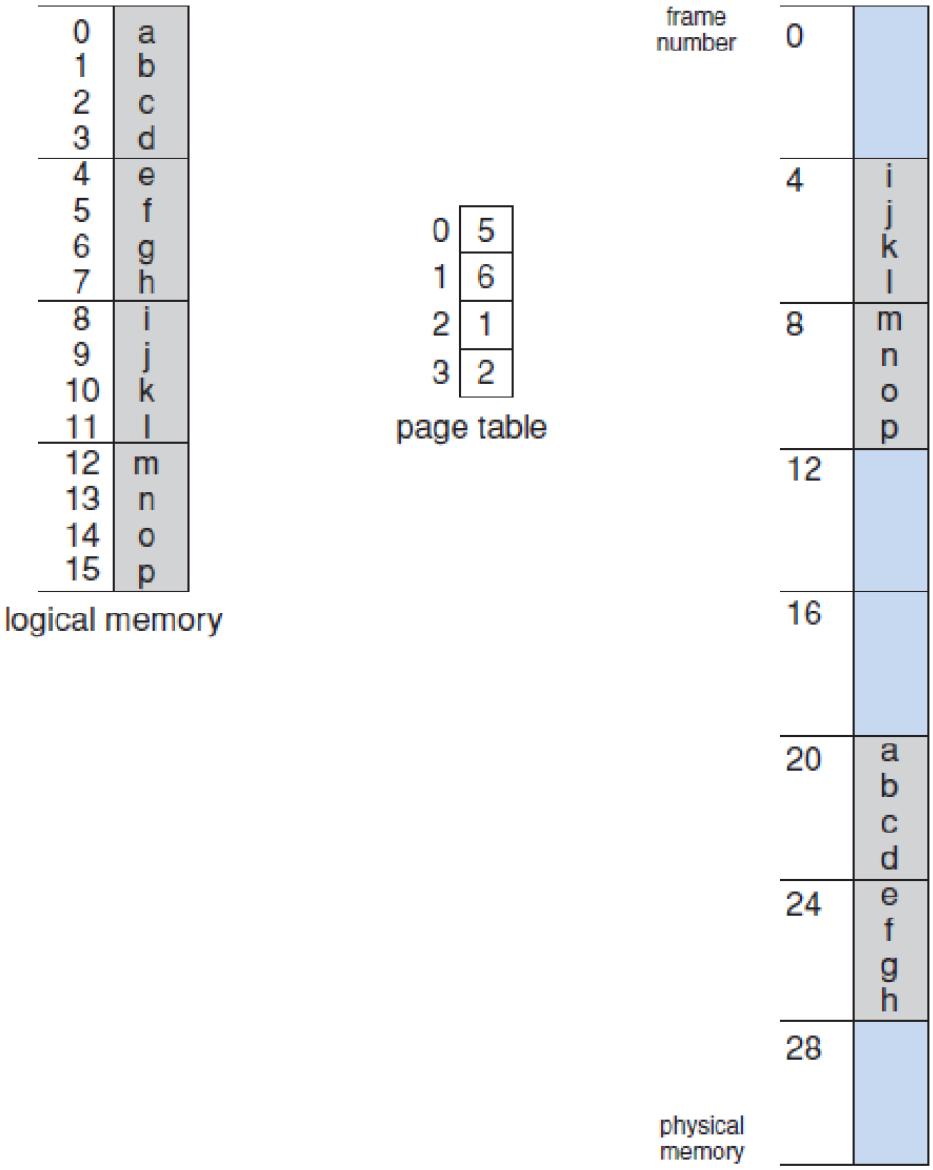
➢Logical address 0 is page 0 and offset 0.

➢Page 0 is in frame 5.

➢Logical address 0 maps to physical address (5x4)+0=20.

➢Logical address 3 (page 0, offset 3) maps to physical address (5x4)+3=23.

➢Logical address 5 (page 0, offset 5) maps to physical address (5×4)+5= 25.



Paging example for a 32-byte memory with 4-byte pages

Page table entries (frame numbers) are typically 32 bit numbers, allowing access to 2^32 physical page frames.

• If those frames are 4 KB in size each, that translates to 16 TB of addressable physical memory.

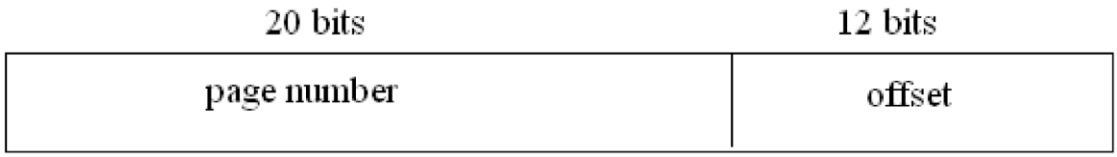
(32 + 12 = 44 bits of physical address space).

• When a process requests memory (e.g. when its code is loaded in from disk), free frames are allocated from a free-frame list, and inserted into that process's page table.

• Since frame size=4 KB, the offset field must contain 12 bits (2_{12} = 4K).

• The remaining 20 bits are page number bits.

• Thus, a logical address is decomposed as shown below.



6) Explain Segmentation in detail with hardware structures and suitable example? CO-3,L2 10 marks

7) SEGMENTATION

• This is a memory-management scheme that supports user-view of memory (Figure 5.17).

• A logical-address space is a collection of segments.

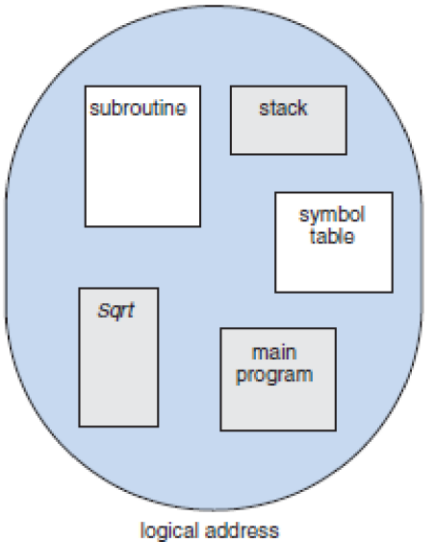
• Each segment has a name and a length.

• The addresses specify both

→ segment-name and

→ offset within the segment.

• Normally, the user-program is compiled, and the compiler automatically constructs segments reflecting the input program.
For ex:
→ The code → Global variables
→ The heap, from which memory is allocated → The stacks used by each thread
→ The standard C library



Hardware support for Segmentation

• Segment-table maps 2 dimensional user-defined addresses into one-dimensional physicaladdresses.
• In the segment-table, each entry has following 2 fields:
1. Segment-base contains starting physical-address where the segment resides in memory.
2. Segment-limit specifies the length of the segment (Figure 5.18).
• A logical-address consists of 2 parts:
1. Segment-number(s) is used as an index to the segment-table .
2. Offset(d) must be between 0 and the segment-limit.
• If offset is not between 0 & segment-limit, then we trap to the OS(logical-addressing attempt beyond end of segment).
• If offset is legal, then it is added to the segment-base to produce the physical-memory address.