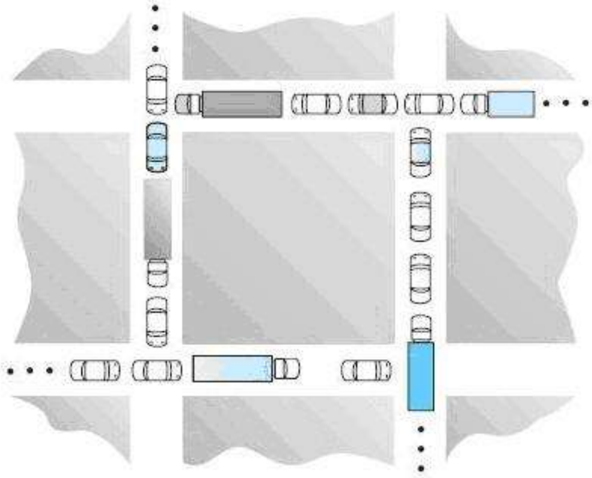
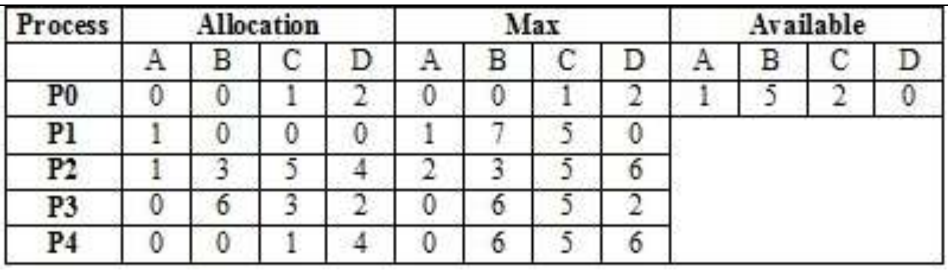
USN

Internal Assessment Test 2 – August 2022

Sub:	**Operating Systems**				Sub Code:	**18CS43**	Branch:	**CSE**	
Date:	**04/08/22**	Duration:	**90 minutes**	Max Marks: **50**	Sem / Sec:	**IV / A, B, C**		**OBE**	

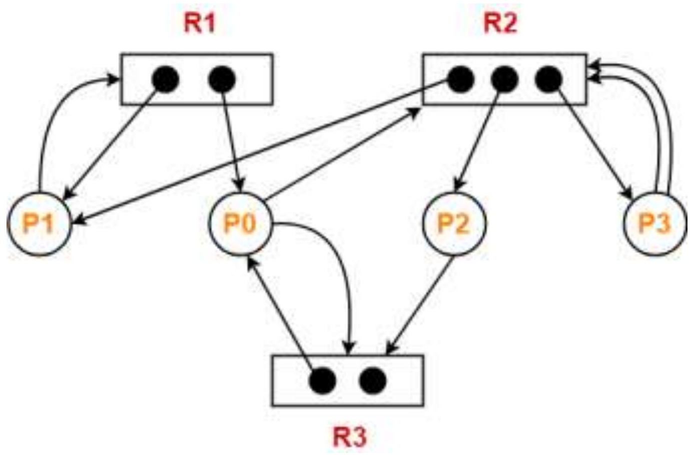
<table>
<tr><th colspan="3"><u>Answer any FIVE FULL Questions</u></th><th>MARKS</th><th>CO</th><th>RBT</th></tr>
<tr><td rowspan="2">1</td><td>a</td><td>What is Critical Section Problem? Draw the general structure of a process with critical section. If you are to provide a solution for Critical Section Problem, explain the requirements that you have to satisfy.</td><td>[5]</td><td>2</td><td>L3</td></tr>
<tr><td>b</td><td>What is Semaphore? What are its types? Explain how it has to be implemented to solve the problem of Process Synchronization.</td><td>[5]</td><td>2</td><td>L2</td></tr>
<tr><td rowspan="2">2</td><td>a</td><td>How Semaphores provide solution for Readers Writers Problem</td><td>[5]</td><td>2</td><td>L2</td></tr>
<tr><td>b</td><td>Help the Dining Philosophers to solve the problem of synchronization using Monitors.</td><td>[5]</td><td>2</td><td>L2</td></tr>
<tr><td rowspan="2">3</td><td>a</td><td>Consider the traffic deadlock depicted in the figure.



What is a deadlock? Show that the four necessary conditions for deadlock indeed hold in this example.</td><td>[5]</td><td>2</td><td>L3</td></tr>
<tr><td>b</td><td>Draw and Justify Resource Allocation Graph
(i) With Deadlock
(ii) With Cycle but No Deadlock</td><td>[5]</td><td>2</td><td>L2</td></tr>
<tr><td>4</td><td>a</td><td>

<table>
<tr><th>Process</th><th colspan="4">Allocation</th><th colspan="4">Max</th><th colspan="4">Available</th></tr>
<tr><td></td><td>A</td><td>B</td><td>C</td><td>D</td><td>A</td><td>B</td><td>C</td><td>D</td><td>A</td><td>B</td><td>C</td><td>D</td></tr>
<tr><td>P0</td><td>0</td><td>0</td><td>1</td><td>2</td><td>0</td><td>0</td><td>1</td><td>2</td><td>1</td><td>5</td><td>2</td><td>0</td></tr>
<tr><td>P1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>7</td><td>5</td><td>0</td><td></td><td></td><td></td><td></td></tr>
<tr><td>P2</td><td>1</td><td>3</td><td>5</td><td>4</td><td>2</td><td>3</td><td>5</td><td>6</td><td></td><td></td><td></td><td></td></tr>
<tr><td>P3</td><td>0</td><td>6</td><td>3</td><td>2</td><td>0</td><td>6</td><td>5</td><td>2</td><td></td><td></td><td></td><td></td></tr>
<tr><td>P4</td><td>0</td><td>0</td><td>1</td><td>4</td><td>0</td><td>6</td><td>5</td><td>6</td><td></td><td></td><td></td><td></td></tr>
</table>

Answer the following questions using the banker's algorithm:
(i) What is the content of the Matrix Need?
(ii) Is the system in a safe state?
(iii) If a request (0,4,2,0) from process P1 be granted immediately?</td><td>[5]</td><td>2</td><td>L3</td></tr>
</table>

	b	Consider the resource allocation graph in the figure-  Find if the system is in a deadlock state otherwise find a safe sequence.	[5]	2	L3
5	a	Explain the various steps of Address Binding with neat diagram (3) Differentiate Internal and External Fragmentation. (2)	[5]	3	L2
	b	Illustrate Contiguous Memory Allocation with example.	[5]	3	L2
6	a	Elucidate Paging as a Memory Management Scheme	[5]	3	L2
	b	What are Translation Load aside Buffer? Explain TLB in detail with a simple paging system with neat diagram.	[5]	3	L2

CI **CCI** **HoD**

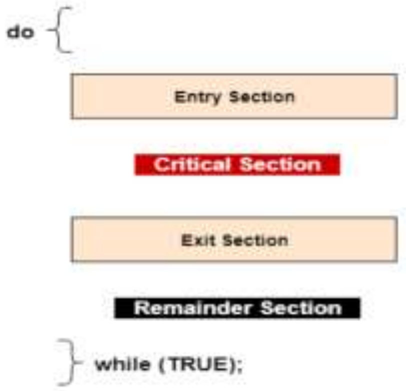
--All the Best--

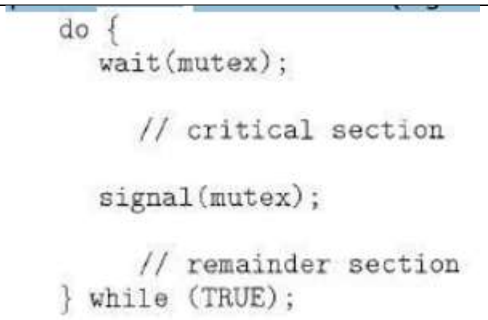
CO-PO Mapping																		
Course Outcomes		**Modules covered**	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**	**PSO1**	**PSO2**	**PSO3**	**PSO4**
CO1	Describe the Operating System Structure and Services.	**1**	3	-	-	-	-	-	-	-	-	-	-	3	-	2	-	-
CO2	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	-	-
CO 5	Elucidate the Protection features in Operating System and case study in Linux OS.	**5**	3	2	2	-	-	-	-	-	-	-	-	3	-	2	-	-

USN

Internal Assessment Test 2 – August 2022

Sub:	**Operating Systems**				Sub Code:	**18CS43**	Branch:	**CSE**
Date:	**04/08/22**	Duration:	**90 minutes**	Max Marks:	**50**	Sem / Sec:	**IV / A, B, C**	**OBE**

		Answer any FIVE FULL Questions	**MARKS**	**CO**	**RBT**
1	a	**What is Critical Section Problem? Draw the general structure of a process with critical section. If you are to provide a solution for Critical Section Problem, explain the requirements that you have to satisfy.** 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, and 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 after a process has made a request to enter its critical-section and before the request is granted.  do { Entry Section Critical Section Exit Section Remainder Section } while (TRUE);	[5]	2	L3
	b	**What is Semaphore? What are its types? Explain how it has to be implemented to solve the problem of Process Synchronization.** Counting Semaphore • The value of a semaphore can range over an unrestricted domain 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. 1) 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 (Figure 3.9).	[5]	2	L2



```
do {
   wait(mutex);

      // critical section

   signal(mutex);

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

semaphores 2) Use of counting semaphores

• Counting semaphores can be used to control access to a given resource consisting of a finite number o£ 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.

2) Solving synchronization problems

• Semaphores can also be used to solve synchronization problems.

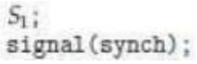
• For example, consider 2 concurrently running-processes:

Suppose we require that S2 be executed only after S1 has completed.

We can implement this scheme readily

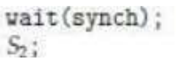
by letting P1 and P2 share a common semaphore synch initialized to 0,

and by inserting the following statements in process P1



```
S1;
signal(synch);
```

and the following statements in process P2



```
wait(synch);
S2;
```

• Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal (synch), which is after statement S1 has been executed.

Semaphore Implementation

• Main disadvantage of semaphore:

→ Busy waiting.

• Busy waiting: While a process is in its critical-section, any other process that tries to enter its critical-section must loop continuously in the entry code.

• Busy waiting wastes CPU cycles that some other process might be able to use productively.

• This type of semaphore is also called a spinlock (because the process "spins" while waiting for the lock).

• To overcome busy waiting, we can modify the definition of the wait() and signal() as follows:

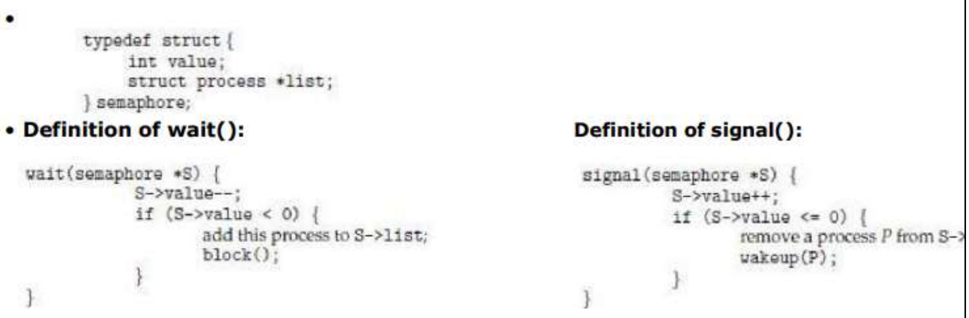
→ When a process executes the wait() and finds that the semaphore-value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself.

→ A process that is blocked (waiting on a semaphore S) should be restarted when some other process executes a signal(). The process is restarted by a wakeup().

• We assume 2 simple operations: → block() suspends the process that invokes it.

→ wakeup(P) resumes the execution of a blocked process P.

• We define a semaphore as follows:



```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

• **Definition of wait():**

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

Definition of signal():

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->
                wakeup(P);
        }
}
```

2 a

How Semaphores provide solution for Readers Writers Problem

The reader processes share the following data structures:

semaphore mutex, wrt;

int readcount;

The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0.

The semaphore wrt is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the object.

The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

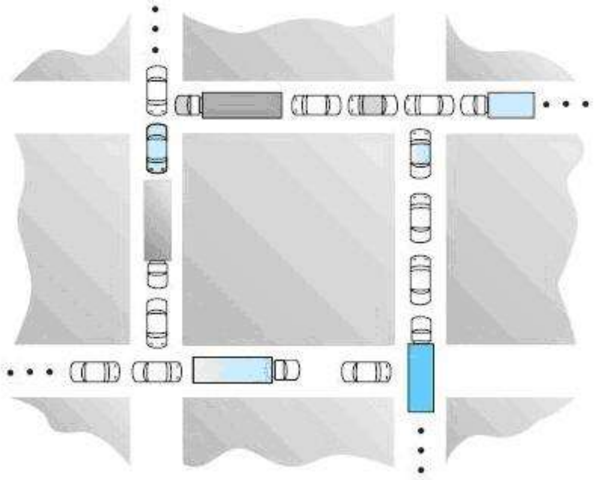
If a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n-1 readers are queued on mutex.

[5] 2 L2

		When a writer executes signal (wrt), we may resume the execution of either the waiting readers or a single waiting writer
		//Writer Process
		do
		{ wait(wrt); // writing is performed
		signal (wrt) ,-
		}while (TRUE);
		//Reader Process do {
		wait(mutex); readcount + + ;
		if (readcount == 1) wait(wrt);
		signal(mutex); // reading is performed wait (mutex) ,- readcount--;
		if (readcount == 0) signal(wrt);
		signal(mutex); }
		while (TRUE);			

	b	**Help the Dining Philosophers to solve the problem of synchronization using Monitors.**
		This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:
		enum {thinking, hungry, eating} state[5];
		thinking: State when philosopher does not need chopsticks
		hungry: State when philosopher needs chopsticks, but didn't obtain them
		eating: State when philosopher needs chopsticks, and has obtained them
		Philosopher i can set the variable state[i] = eating only if her two neighbours are not eating:
		(state[(i+4) °/» 5] != eating) and (state[(i+1) % 5] != eating).
		We also need to declare condition self [5] where philosopher i can wait when she is hungry but is unable to obtain the chopsticks she needs.
		The following is the solution for each philosopher. Each philosopher i must invoke the operations pickup () and putdownO in the following sequence:
		dp.pickup(i); //eat
		dp.putdown(i);
		The monitor implementation is as follows
		monitor dp
		enum {THINKING, HUNGRY, EATING}state [5]
		condition self [5] ;
		void pickup(int i)
		{
		state [i] = HUNGRY;
		test (i) ;
			[5]	2	L2

		if (state [i] != EATING)
		 self [i] .wait() ;
		}
		void putdown(int i)
		{
		state til = THINKING;
		test((i + 4) % 5} ;
		
		 test((i + 1) % 5) ;
		 }
		void test(int i)
		{
		if ((state [(i + 4) % 5] != EATING) && (state [i] == HUNGRY) && (state [(i + 1) % 5] != EATING))
		 {
		state [i] = EATING;
		self [i] .signal() ;
		}
		}
		initialization-code ()
		{
		 for (int i = 0; i < 5; i++)
		 state [i] = THINKING;
		 }
		}			

3	a	**Consider the traffic deadlock depicted in the figure.**  What is a deadlock? Show that the four necessary conditions for deadlock indeed hold in this example. In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; and if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. Characteristics (or	[5]	2	L3

Necessary conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. Mutual exclusion. At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and wait. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. No preemption. Resources cannot be preempted. That is, a resource can be released only voluntarily by the process holding it, after that process has completed its task. 4. Circular wait. A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, •••, Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

Methods to handle deadlocks: Prevention, Avoidance, Detect and recovery

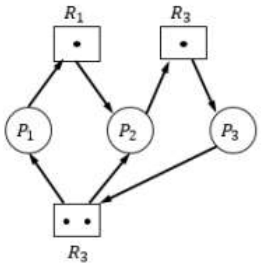
b

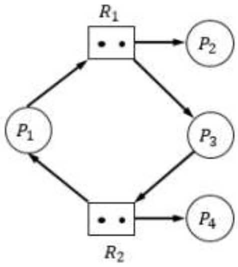
Draw and Justify Resource Allocation Graph

(i) With Deadlock

(ii) With Cycle but No Deadlock

Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes: P = {P1, P2,…, Pn}, the set consisting of all the active processes in the system, and R = {R1, R2, … Rm}, the set consisting of all resource types in the system.

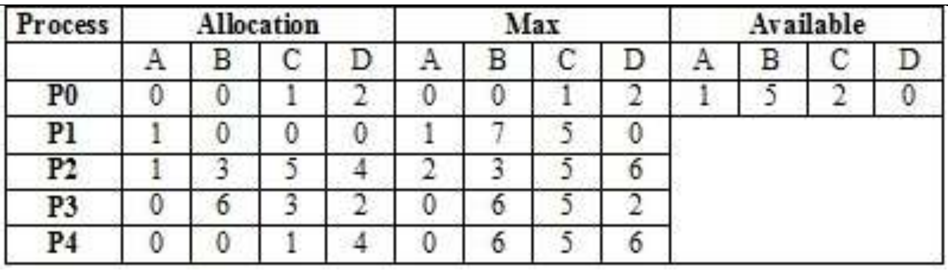




Take example on the left. Here all the resources are part of a cycle. From this, we learn that the system is in a deadlocked state. Take example on the right. Here, even though all the resources are occupied by all the processes, not all resources are part of a cycle. Hence, no deadlock.

[5] 2 L2

4 a



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

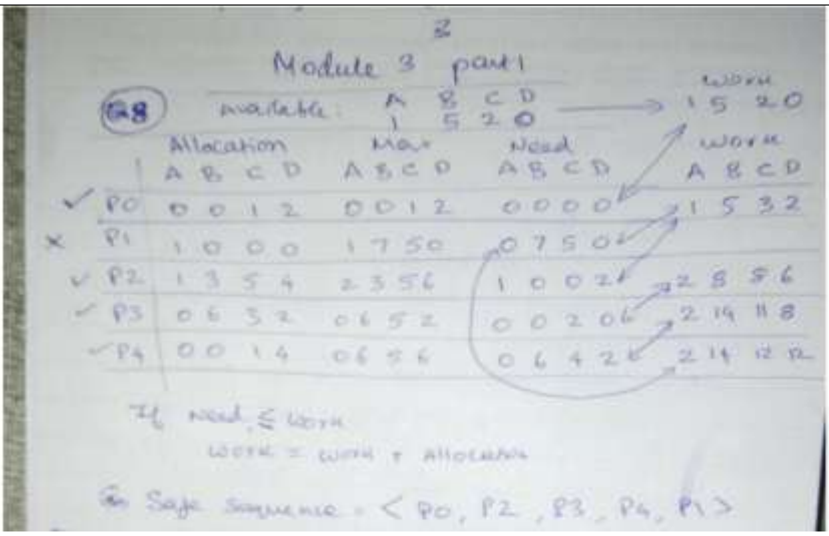
Answer the following questions using the banker's algorithm:

(i) **What is the content of the Matrix Need?**

(ii) **Is the system in a safe state?**

(iii) **If a request (0,4,2,0) from process P1 be granted immediately**?

[5] 2 L3



2

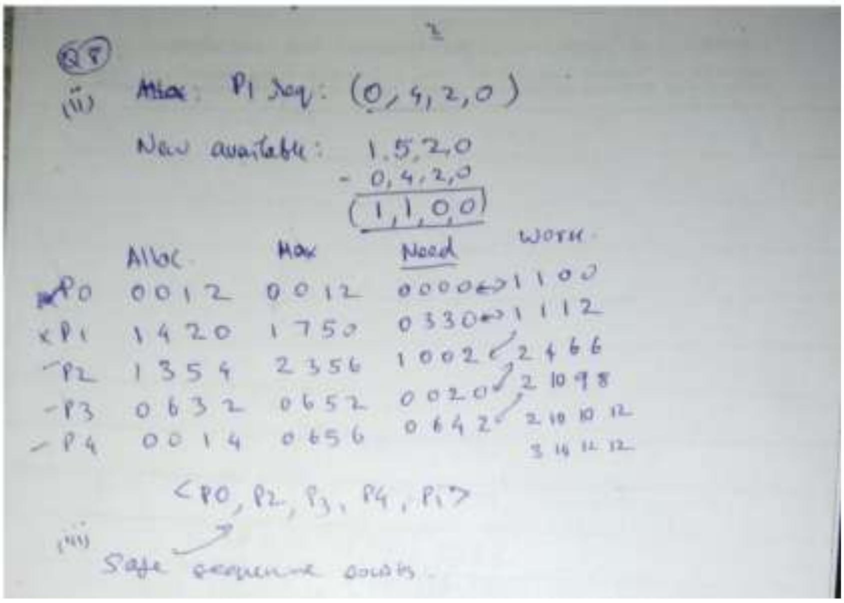
Module 3 part 1

Q8 available: A B C D
1 5 2 0 → Work 1 5 2 0

	Allocation A B C D	Max A B C D	Need A B C D	Work A B C D
✓ P0	0 0 1 2	0 0 1 2	0 0 0 0	1 5 3 2
× P1	1 0 0 0	1 7 5 0	0 7 5 0	
✓ P2	1 3 5 4	2 3 5 6	1 0 0 2	2 8 5 6
✓ P3	0 6 3 2	0 6 5 2	0 0 2 0	2 14 11 8
✓ P4	0 0 1 4	0 6 5 6	0 6 4 2	2 14 12 12

If Need ≤ Work
Work = Work + Allocation

Safe sequence = < P0, P2, P3, P4, P1 >



2

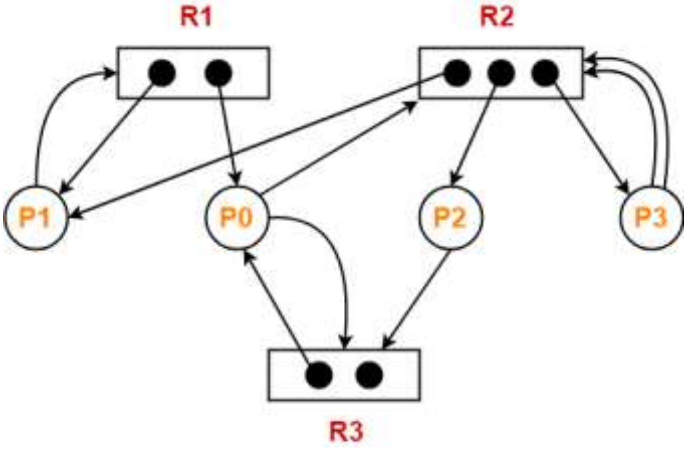
Q8
(ii) Max: P1 req: (0, 4, 2, 0)

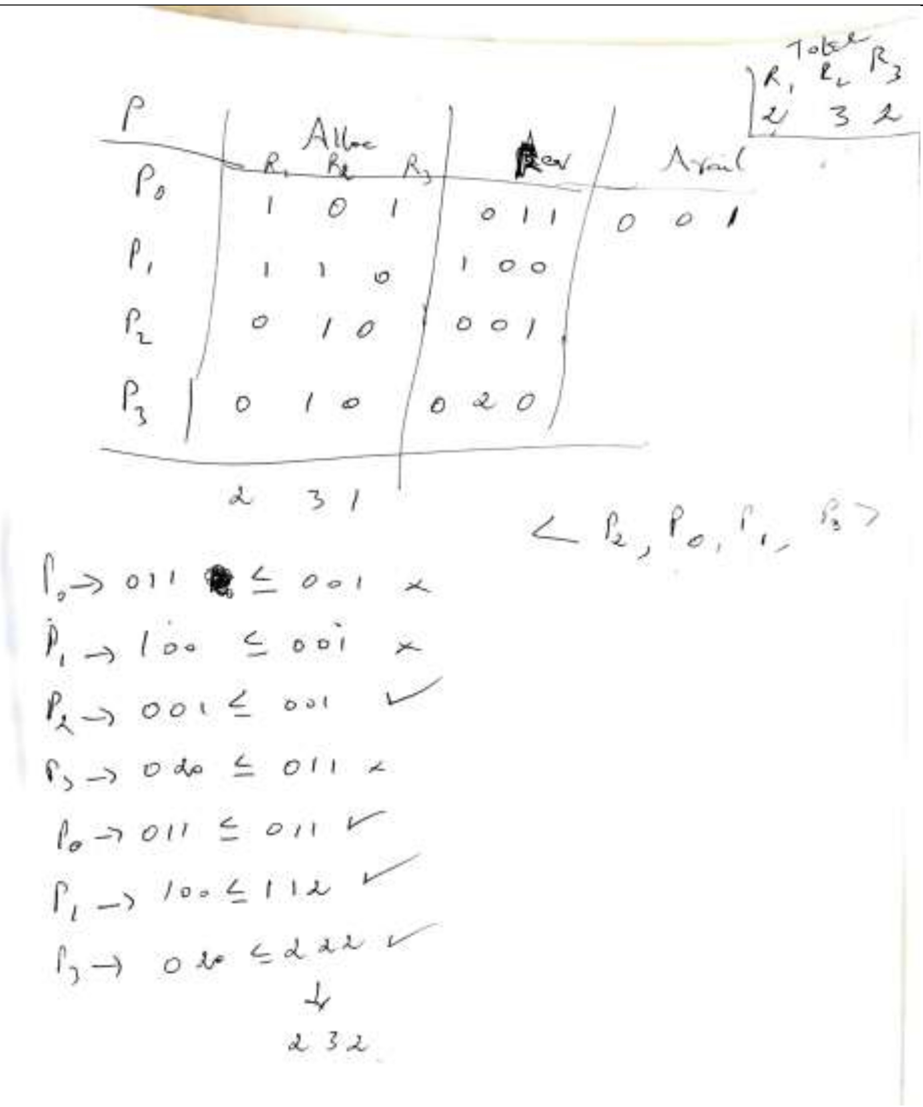
New available: 1, 5, 2, 0
− 0, 4, 2, 0
$\boxed{(1, 1, 0, 0)}$

	Alloc	Max	Need	Work
P0	0 0 1 2	0 0 1 2	0 0 0 0	1 1 0 0
× P1	1 4 2 0	1 7 5 0	0 3 3 0	1 1 1 2
✓ P2	1 3 5 4	2 3 5 6	1 0 0 2	2 4 6 6
✓ P3	0 6 3 2	0 6 5 2	0 0 2 0	2 10 9 8
✓ P4	0 0 1 4	0 6 5 6	0 6 4 2	2 10 10 12
				3 14 12 12

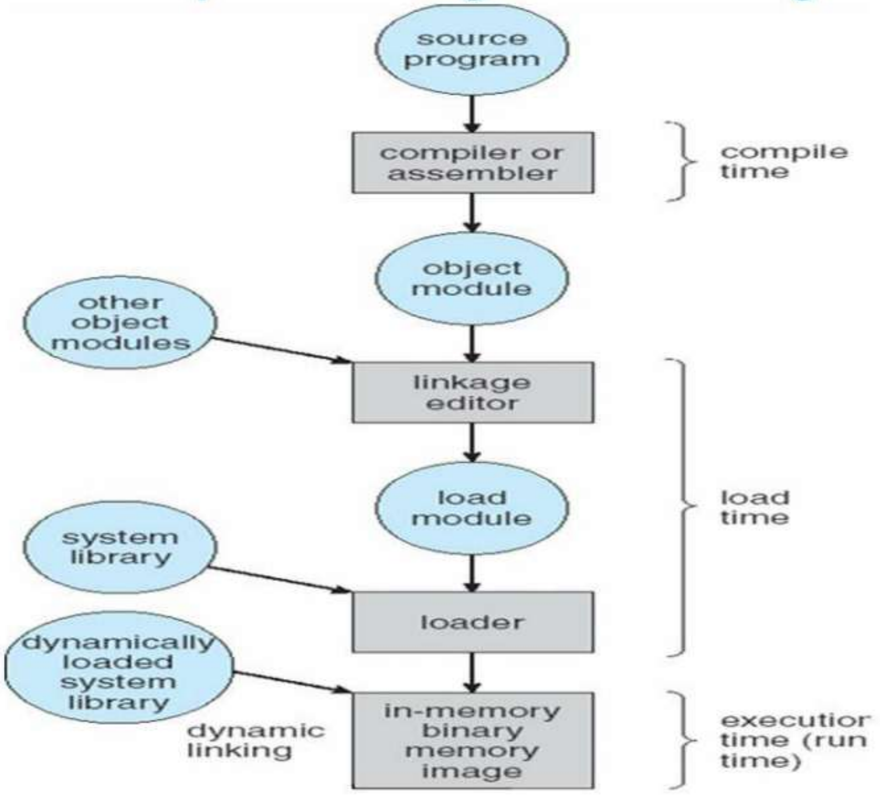
< P0, P2, P3, P4, P1 >

(iii) Safe sequence exists.

	b	Consider the resource allocation graph in the figure-  Find if the system is in a deadlock state otherwise find a safe sequence.	[5]	2	L3

<table>
<tr><td></td><td></td><td>
<table>
<tr><th>P</th><th colspan="3">Alloc</th><th colspan="3">Req</th><th colspan="3">Avail</th></tr>
<tr><td></td><td>R1</td><td>R2</td><td>R3</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>P0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>P1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td></td><td></td><td></td></tr>
<tr><td>P2</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td></td><td></td><td></td></tr>
<tr><td>P3</td><td>0</td><td>1</td><td>0</td><td>0</td><td>2</td><td>0</td><td></td><td></td><td></td></tr>
<tr><td></td><td>2</td><td>3</td><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>
<table>
<tr><th colspan="3">Total</th></tr>
<tr><td>R1</td><td>R2</td><td>R3</td></tr>
<tr><td>2</td><td>3</td><td>2</td></tr>
</table>
&lt; P2, P0, P1, P3 &gt;
P0 → 011 ≤ 001 ×
P1 → 100 ≤ 001 ×
P2 → 001 ≤ 001 ✓
P3 → 020 ≤ 011 ×
P0 → 011 ≤ 011 ✓
P1 → 100 ≤ 112 ✓
P3 → 020 ≤ 222 ✓
↓
232</td><td></td><td></td><td></td></tr>
<tr><td>5</td><td>a</td><td><b>Explain the various steps of Address Binding with neat diagram (3)</b>
User programs typically refer to memory addresses with symbolic names such as "i", "count", and "average Temperature". These symbolic names must be mapped or bound to physical memory addresses, which typically occurs in several stages:
Compile Time- If it is known at compile time where a program will reside in physical memory, then absolute code can be generated by the compiler, containing actual physical addresses. However if the load address changes at some later time, then the program will have to be recompiled.
Load Time- If the location at which a program will be loaded is not known at compile time, then the compiler must generate relocatable code, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.
o Execution Time- If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time. Figure 8.3 shows the various stages of the binding processes and the units involved in each stage</td><td>[5]</td><td>3</td><td>L2</td></tr>
</table>

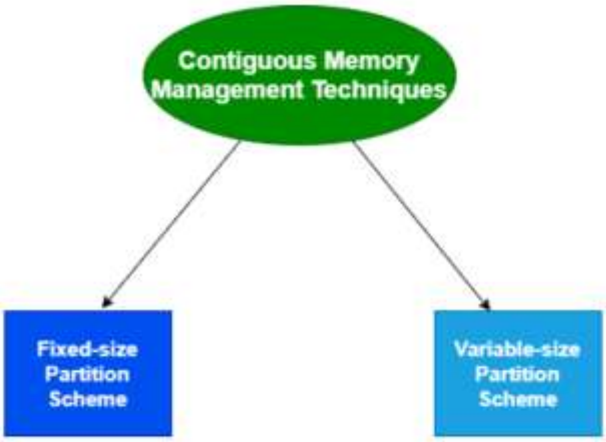
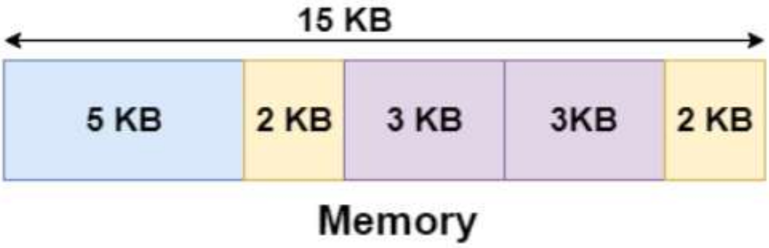
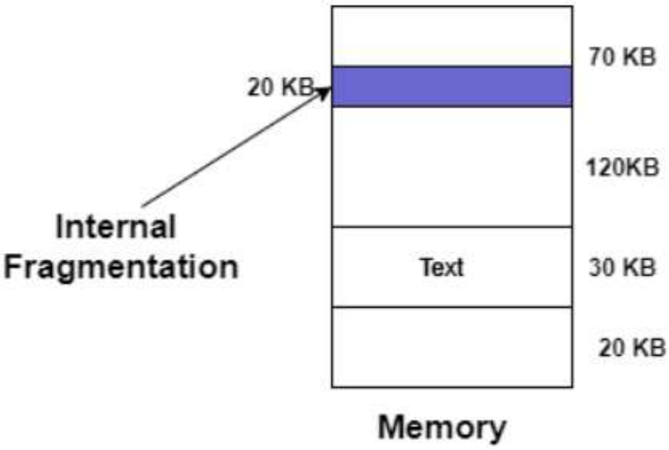
Multistep Processing of a User Program

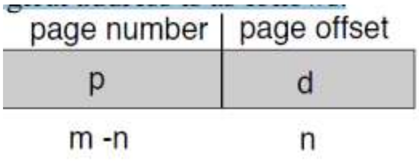


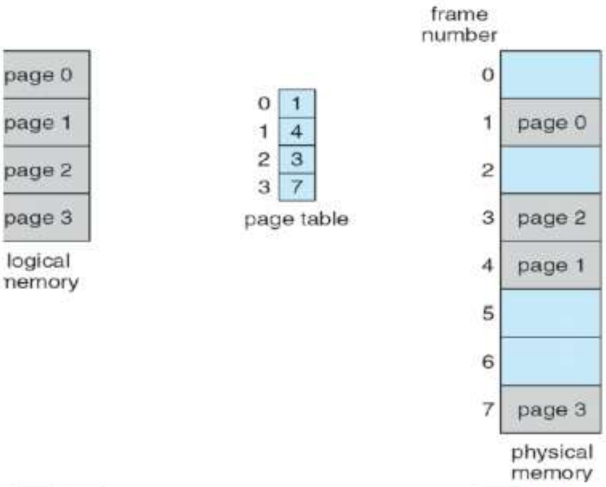
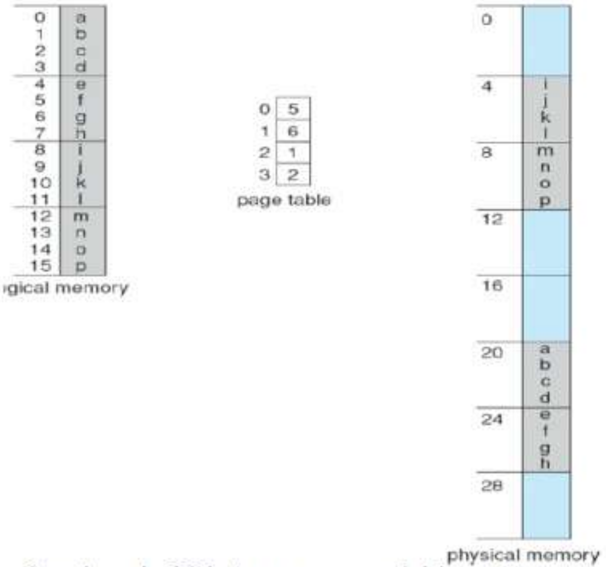
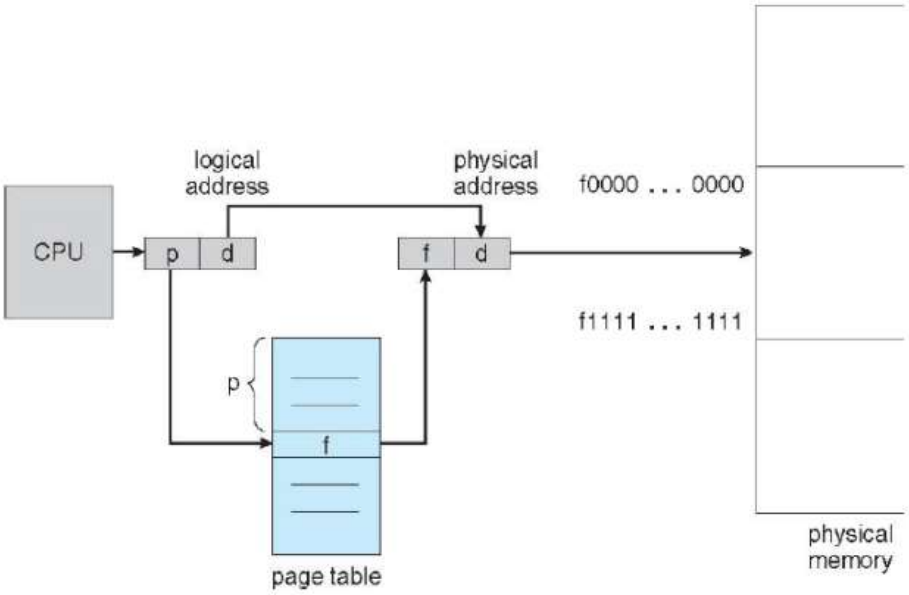
Differentiate Internal and External Fragmentation. (2)

Internal Fragmentation	External Fragmentation
Internal fragmentation is the wasted space within each allocated block because of rounding up from the actual requested allocation to the allocation granularity.	External fragmentation is the various free spaced holes that are generated in either your memory or disk space. External fragmented blocks are available for allocation, but may be too small to be of any use.
It occurs when fixed sized memory blocks are allocated to the processes	It occurs when variable size memory space are allocated to the processes dynamically.
When the memory assigned to the process is slightly larger than the memory requested by the process this creates free space in the allocated block causing internal fragmentation.	When the process is removed from the memory, it creates the free space in the memory causing external fragmentation
Solution: The memory must be partitioned into variable sized blocks and assign the best fit block to the process.	Solution: Compaction, paging and segmentation.
Example: Consider a multiplepartition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate	Example: First-fit and Best-fit strategies. We could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in

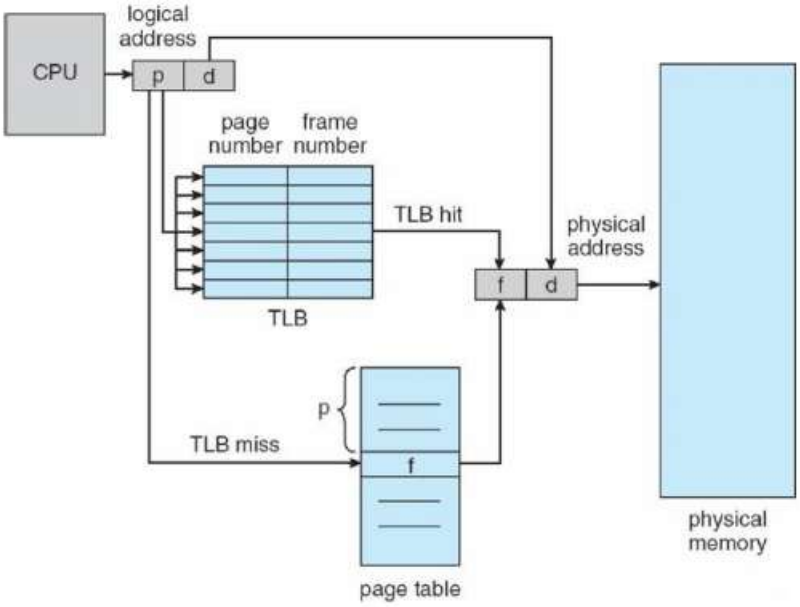
exactly the requested block, we are left with a hole of 2 bytes.	one big free block instead, we might be able to run several more processes.

	b	**Illustrate Contiguous Memory Allocation with example.** In **Contiguous memory allocation** which is a memory management technique, whenever there is a request by the user process for the memory then a single section of the contiguous memory block is given to that process according to its requirement. Contiguous Memory allocation is achieved just by dividing the memory into the **fixed-sized partition.** The memory can be divided either in the **fixed-sized partition or in the variable-sized partition** in order to allocate contiguous space to user processes.  Contiguous Memory Management Techniques Fixed-size Partition Scheme Variable-size Partition Scheme  It is important to note that these partitions are allocated to the processes as they arrive and the partition that is allocated to the arrived process basically depends on the algorithm followed. If there is some wastage inside the partition then it is termed **Internal Fragmentation**.  70 KB 20 KB 120KB Internal Fragmentation Text 30 KB 20 KB Memory	[5]	3	L2

6	a	**Elucidate Paging as a Memory Management Scheme** Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous. Paging avoids the considerable problem of fitting memory chunks of varying sizes onto the backing store. The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames. Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d). The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. If the size of logical address space is 2m and a page size is 2n addressing units (bytes or words), then the high-order m – n bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:  where p is an index into the page table and d is the displacement within the page. Logical address to physical address: As a concrete (although minuscule) example, consider the memory in the Figure below. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 (= (5 x 4) + 0). Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5x4) + 3). Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 (= (6x4) + 0). Logical address 13 maps to physical address 9.	[5]	3	L2

		 page 0 page 1 page 2 page 3 logical nemory 0 1 1 4 2 3 3 7 page table frame number 0 1 page 0 2 3 page 2 4 page 1 5 6 7 page 3 physical memory  0 a 1 b 2 c 3 d 4 e 5 f 6 g 7 h 8 i 9 j 10 k 11 l 12 m 13 n 14 o 15 p ıgical memory 0 5 1 6 2 1 3 2 page table 0 4 i j k l 8 m n o p 12 16 20 a b c d 24 e f g h 28 physical memory ı=2 and *m*=4 32-byte memory and 4-byte pages  CPU logical address p d physical address f d f0000 ... 0000 f1111 ... 1111 p f page table physical memory			
	b	**What are Translation Load aside Buffer? Explain TLB in detail with a simple paging system with neat diagram.** b What are Translation Load aside Buffer? Explain TLB in detail with a simple system with neat diagram. Translation look-aside buffers (TLBs) are a special, small, fast lookup hardware	[5]	3	L2

The TLB is associative, high-speed memory. Each entry in the TLB consists of tw a key (or tag) and a value. When the associative memory is presented with an it item is compared with all keys simultaneously. If the item is found, the corresp value field is returned. The search is fast; the hardware, however, is expensive. Ty the number of entries in a TLB is small, often numbering between 64 and 1,024. Tl is used with page tables in the following way. The TLB contains only a few of th table entries. When a logical address is generated by the CPU, its page nu presented to the TLB. If the page number is found, its frame number is imm available and is used to access memory. The whole task may take less than 10 longer than it would if an unmapped memory reference were used. If the page nu not in the TLB (known as a TLB miss), a memory reference to the page table r made. When the frame number is obtained, we can use it to access memory. In a we add the page number and frame number to the TLB, so that they will be found on the next reference. If the TLB is already full of entries, the operating syste select one for replacement. Replacement policies range from least recently used (L random. Furthermore, some TLBs allow entries to be wired down, meaning th cannot be removed from the TLB. Typically, TLB entries for kernel code are wired



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	-	-
CO 5	Elucidate the Protection features in Operating System and case study in Linux OS.	**5**	3	2	2	-	-	-	-	-	-	-	-	3	-	2	-	-