USN					
Subject:	**OPERATING SYSTEM**			**Sub Code:**	**21CS44**
Date:	**11-08-2023**	**Duration:**	**90 Min’s**	**Sem/Sec:**	**IV Sem-A,AIML**

Note: Answer any Five Full Questions	Marks	CO	RBT
1a.What is Critical Section Problem? If you are to provide a solution for Critical Section Problem,explain the requirements that you have to satisfy.	[5]	2	L3

Critical Section Problem: [2]

The critical section cannot be executed by more than one process at the same time; operating system faces the difficulties in allowing and disallowing the processes from entering the critical section.

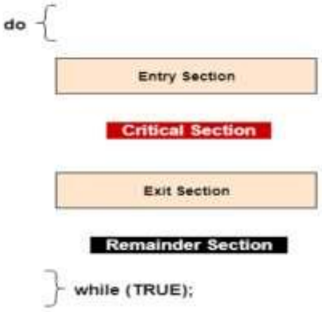
The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise.

A solution to the problem must satisfy the following 3 requirements: [3]

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.



1b. What is Semaphore? What are its types? Explain how it has to be implemented to solve the problem of Process Synchronization.	[5]	2	L2

What is Semaphore? What are its types?[2.5]

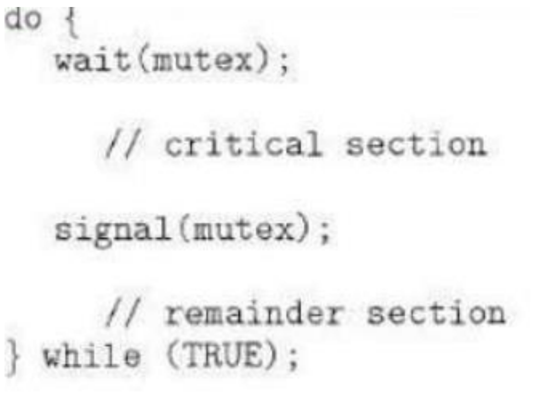
Counting Semaphore • The value of a semaphore can range over an unrestricted domain

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



```
do {
   wait(mutex);

      // critical section

   signal(mutex);

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

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

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 [2.5]

- 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

and the following statements in process P2

- 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:

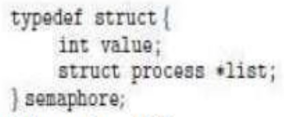
→ When a process executes the wait() and finds that the semaphore-value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself.

→ 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().

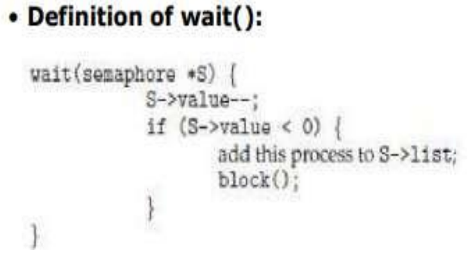
- We assume 2 simple operations: → block() suspends the process that invokes it.

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

- We define a semaphore as follows:
-

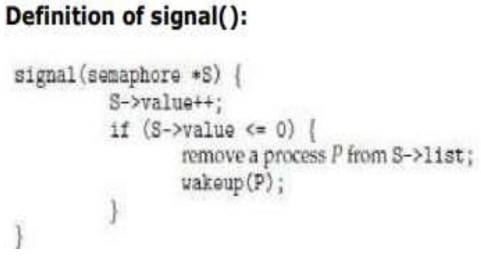


```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```



- **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->list;
                    wakeup(P);
            }
}
```

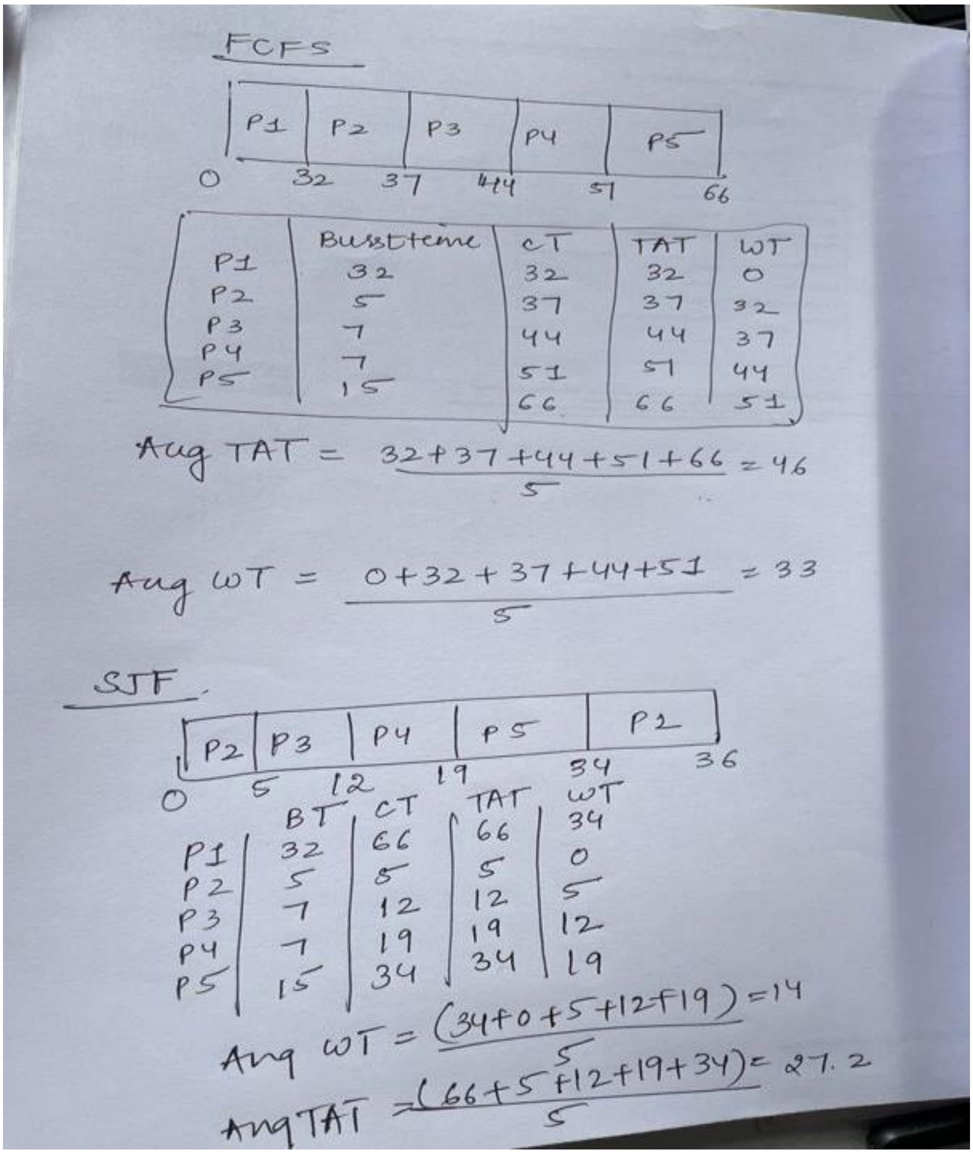
2 a. Consider the set of given process with the Burst time	[5]	2	L2

Process	**Burst Time**
P1	**32**
P2	**5**
P3	**7**
P4	**7**
P5	**15**

Calculate the Average Waiting Time and Turn Around Time for Shortest Job First (SJF) and First Come First Serve (FCFS)

FCFS **[2.5]**

SJF **[2.5]**



FCFS

	P1	P2	P3	P4	P5	
0	32	37	44	51	66	

	Busstteme	cT	TAT	WT
P1	32	32	32	0
P2	5	37	37	32
P3	7	44	44	37
P4	7	51	51	44
P5	15	66	66	51

Aug TAT = $\frac{32+37+44+51+66}{5} = 46$

Aug WT = $\frac{0+32+37+44+51}{5} = 33$

SJF.

	P2	P3	P4	P5	P2
0	5	12	19	34	36

	BT	CT	TAT	WT
P1	32	66	66	34
P2	5	5	5	0
P3	7	12	12	5
P4	7	19	19	12
P5	15	34	34	19

Avg WT = $\frac{(34+0+5+12+19)}{5} = 14$

AvgTAT = $\frac{(66+5+12+19+34)}{5} = 27.2$

2b.Help the Dining Philosophers to solve the problem of synchronization using monitor.	[5]	2	L2

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
L2if (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;
}
}
```

3a.What is a deadlock? Explain the four necessary conditions for deadlock .

[5] 2 L3

Deadlock: **[2]**

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 Necessary conditions) **[3]**

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

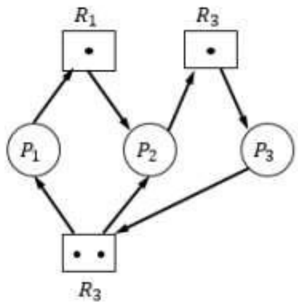
3b.Draw and Justify Resource Allocation Graph

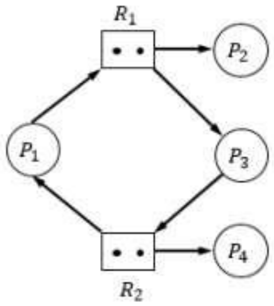
(i) With Deadlock **[2.5]**

(ii) **With Cycle but No Deadlock** **[2.5]**

[5] 2 L2

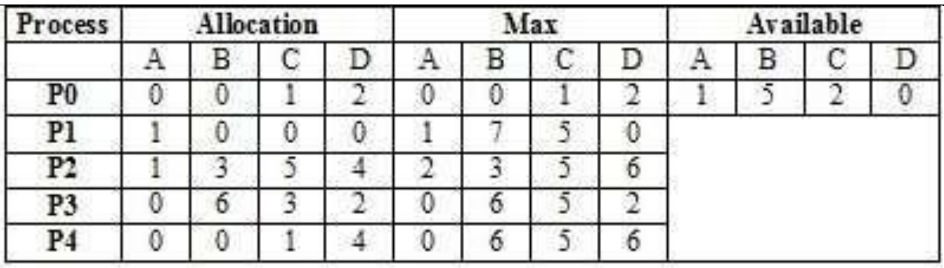
Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes: P = {P1, P2,…, Pn}, the set consisting of all the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

4



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:

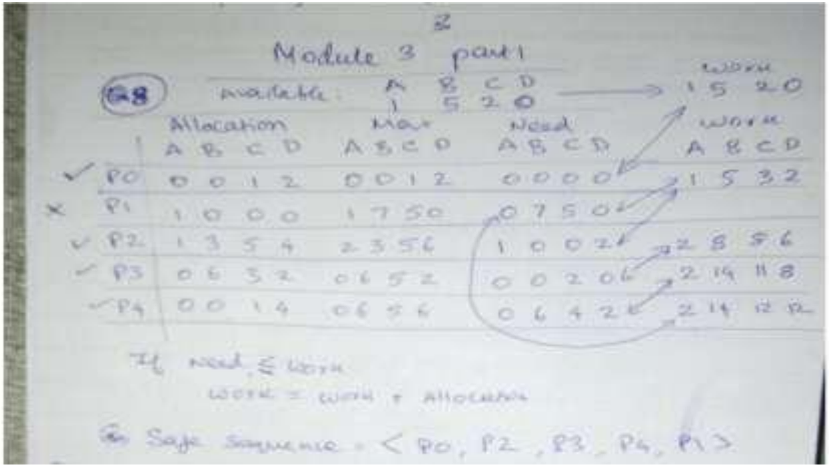
[5] 2 L3

(i) **What is the content of the Matrix Need? [2]**

(ii) **Is the system in a safe state? [2]**

(iii) **If a request (0,4,2,0) from process P1 be granted immediately**? [1]

[2]



2

Module 3 part1

Q8 available: A B C D → work 1 5 2 0

1 5 2 0

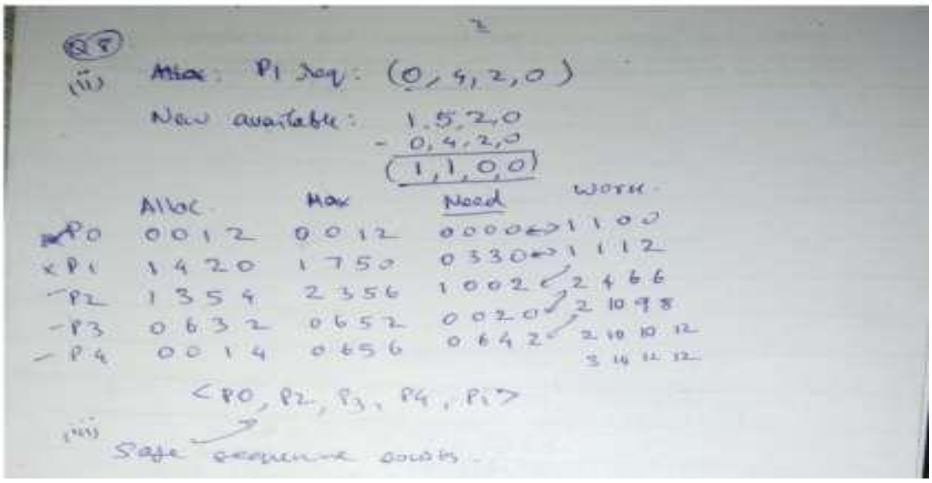
	Allocation A B C D	Max A B C D	Need A B C D	work A B C D
✓ P0	0 0 1 2	0 0 1 2	0 0 0 0	1 5 3 2
✗ P1	1 0 0 0	1 7 5 0	0 7 5 0	
✓ P2	1 3 5 4	2 3 5 6	1 0 0 2	2 8 5 6
✓ P3	0 6 3 2	0 6 5 2	0 0 2 0	2 14 11 8
✓ P4	0 0 1 4	0 6 5 6	0 6 4 2	2 14 12 12

If Need ≤ work

work = work + Allocation

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

[2]



2

Q8

(ii) Max: P1 Req: (0, 4, 2, 0)

New available: 1, 5, 2, 0

\- 0, 4, 2, 0

(1, 1, 0, 0)

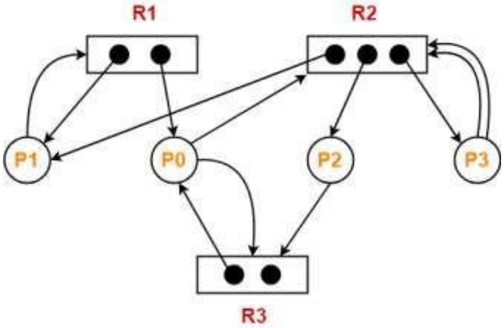
	Alloc	Max	Need	work
✓ P0	0 0 1 2	0 0 1 2	0 0 0 0	1 1 0 0
✗ P1	1 4 2 0	1 7 5 0	0 3 3 0	1 1 1 2
✓ P2	1 3 5 4	2 3 5 6	1 0 0 2	2 4 6 6
✓ P3	0 6 3 2	0 6 5 2	0 0 2 0	2 10 9 8
✓ P4	0 0 1 4	0 6 5 6	0 6 4 2	2 10 10 12
				3 14 11 12

<P0, P2, P3, P4, P1>

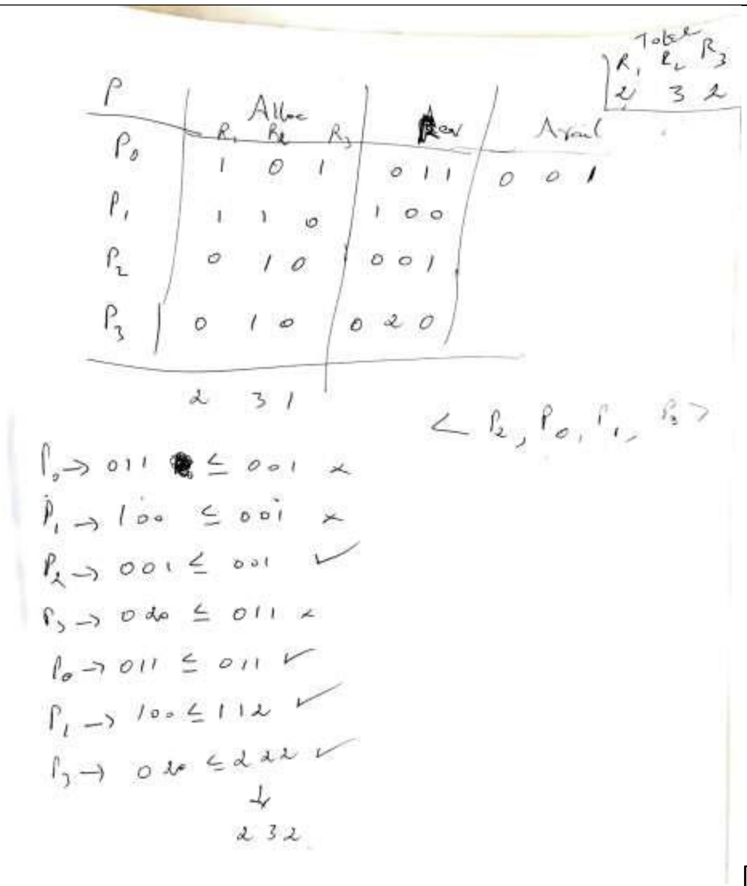
(iii) Safe sequence exists.

4b.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



Total

	R1	R2	R3
	2	3	2

P	Alloc R1 R2 R3	Req	Avail
P0	1 0 1	0 1 1	0 0 1
P1	1 1 0	1 0 0	
P2	0 1 0	0 0 1	
P3	0 1 0	0 2 0	
	2 3 1		

< P2, P0, P1, P3 >

P0 → 011 ≤ 001 ×

P1 → 100 ≤ 001 ×

P2 → 001 ≤ 001 ✓

P3 → 020 ≤ 011 ×

P0 → 011 ≤ 011 ✓

P1 → 100 ≤ 112 ✓

P3 → 020 ≤ 222 ✓

↓

232

[5]

5 a. **Explain the various steps of Address Binding with neat diagram (3)** **Differentiate Internal and External Fragmentation. (2)**	[5]	3	L2

The various steps of Address Binding with neat diagram (3)

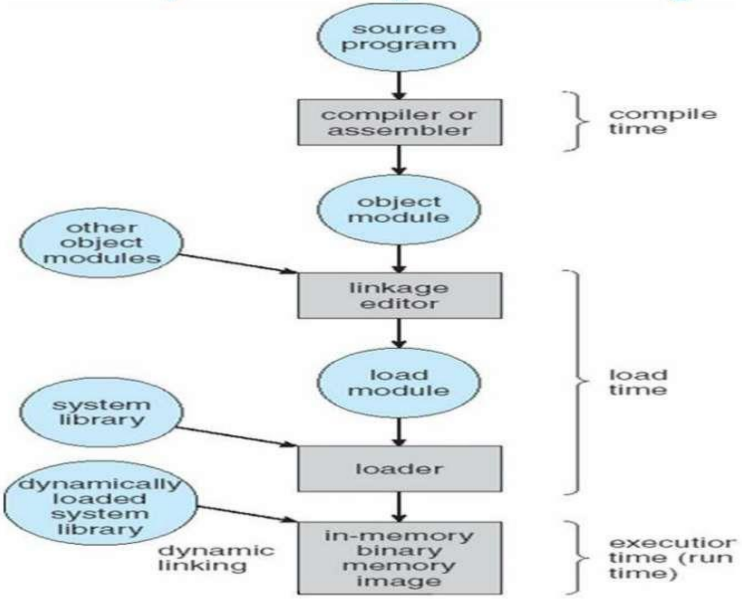
User programs typically refer to memory addresses with symbolic names such as "i", "count", and "average Temperature". These symbolic names must be mapped or bound to physical memory addresses, which typically occurs in several stages:

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 shows the various stages of the binding processes and the units involved in each stage

Multistep Processing of a User Program

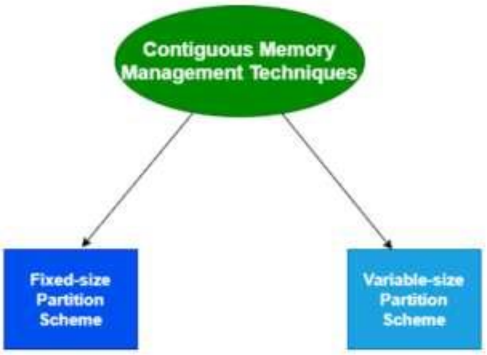
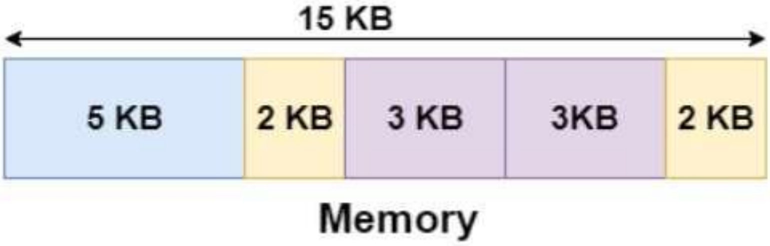
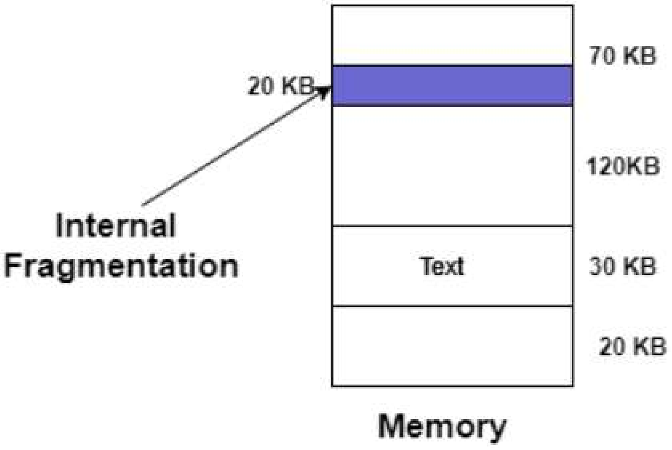


Internal and External Fragmentation. (2)

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.

5 b. illustrate Contiguous Memory Allocation with example.	[5]	3	L2

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.   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**.  			
6. a.Elucidate Paging ?Consider a single level paging scheme. The virtual address space is 4 MB and page size is 4 KB. What is the maximum page table entry size possible such that the entire page table fits well in one page? **Paging : [2]** 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	[5]	3	L3

involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages.

Problem: **[3]**

Number of Pages of Process-Number of pages the process is divided= Process size / Page size= 4 MB / 4 KB= 2^{10}pages

Page Table Size-Let page table entry size = B bytes
Now,Page table size= Number of entries in the page table x Page table entry size= Number of pages the process is divided x Page table entry size= 2^{10}x B bytes
Now,According to the above condition, we must have-2^{10}x B bytes <= 4 KB
2^{10}x B <= 2^{12}
B <= 4
Thus, maximum page table entry size possible = 4 bytes

6 b What are Translation Load aside Buffer? Explain TLB in detail with a simple paging system with neat diagram.

TLB: **[2]**

The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; the hardware, however, is expensive.

Explain TLB in detail with a simple paging system with neat diagram. [3]

Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024. The getable