

▣ Define Process Synchronization problem. Illustrate how a binary semaphore can be used to implement mutual exclusion among n process.

→ Process Synchronization: is a mechanism in OS used to manage the execution of multiple processes that access shared resources.

→ Its main purpose is to ensure data consistency, prevent race conditions and avoid deadlocks in multiprocess environment.

Process Synchronization Problems: The process synchronization problem arises when multiple processes access shared resources (memory, files, I/O devices) concurrently without proper coordination, this can lead to:

- (i) race conditions

- (ii) Data inconsistency

- (iii) Deadlocks or starvation

(i) Race conditions: Two or more processes modify shared data simultaneously, causing unpredictable results.

(ii) Data Inconsistency: Updates may be lost or corrupted.

(iii) Deadlocks or Starvation: Improper resource handling can freeze or delay processes indefinitely.

Binary Semaphore for mutual Exclusion:

A binary semaphore is a synchronization primitive that can have only two values: 0 and 1. It provides two atomic operations.

(i) wait(s): If the semaphore is 1, set it to 0 and proceed. If it is 0, block the process (or inhibit it, becomes driver) semaphore becomes 0 and proceed.

(ii) Signal(s): Set s to 1, wake up a blocked process.

Mutual Exclusion among processes using

a binary semaphore:

Let s be a binary semaphore initialized to 1. Each process $P_i (i = 1 \text{ to } n)$ must

execute the following structure to access its critical ~~section~~ section.

Pseudocode:

```
// Initialization :  $s = 1$ 
Process  $P_i$ :
do {
    wait( $s$ ) ; // Entry section: acquire lock
    // Critical section // access shared resources
    signal( $s$ ) ; // Exit section: release lock
    // Remainder section
} while (true);
```

Working process:

1. Initial state: $s = 1$ (unlocked)

2. Process P_i requests Entry:

→ Calls wait(s), which automatically checks s :

* If $s == 1$ it sets $s = 0$ and enters the critical

section

* If $s == 0$, the process is blocked until s becomes

1. no more waiting at entry here. $s = 1$

3. Inside Critical Section:

→ Only one process can be in the critical section at a time because s is now 0, blocking all other processes.

4. Exiting Critical section: The process calls signal(s), which sets $s=1$ and wakes up one of the blocked process.

5. Woken Process:

→ The awakened process then acquires the lock ($s \geq 0$) and enters the critical section.

Example (Processes: P1 & P2):

→ Initially $s=1$

→ Suppose P1 calls wait(s): since $s=1$, it sets $s=0$ and enters the critical section.

→ Now P2 calls wait(s): since $s=0$, it is blocked.

→ When P1 exits, it calls signal(s), setting $s=1$ and waking P2.

→ P2 now acquires the lock and sets $s=0$, and enters the critical section.

This ensures mutual exclusion for any number of processes (n).

Ex

Process-	Allocation			max			Available		
	A	B	C	A	B	C	A	B	C
P ₀	0	1	0	7	5	3	3	3	2
P ₁	2	0	0	3	2	2			
P ₂	3	0	2	9	0	2			
P ₃	2	1	1	2	2	2			
P ₄	0	0	2	4	3	3			

(i) Need = max - allocation

	A	B	C
P ₀	7	4	3
P ₁	1	2	2
P ₂	6	0	0
P ₃	0	1	1
P ₄	4	3	1

(ii) What is the content of the matrix need of the system in a safe state? If so, show a safe order in which the process can execute.

(ii) Safe state: A system is in a safe state if there exists at least one sequence of all processes such that each process can get

its needed resources, execute completely and then release its resources without causing a deadlock.

→ Banker's algo checks for safe states to ensure the system avoids deadlocks.

Unsafe state:

A system is in unsafe state if no such safe sequence exists.

→ In this system state, system may

transition to a deadlock.

We try to find process whose need $\leq$ available

(i)

Processes	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
✓ P ₀	0	1	0	7	5	3	7	4	3	3	3	2
✓ P ₁	2	0	0	3	2	2	1	2	2	5	3	2
P ₂	3	0	2	9	0	2	6	0	0	7	4	3
✓ P ₃	2	1	1	2	2	2	0	1	1	7	5	3
P ₄	0	0	2	4	3	3	4	3	1	10	5	5

10 5 7

$$(i) P_5 \rightarrow \text{sum of need} \leq \text{Available} \\ (1, 1, 1) \leq (3, 3, 2) \\ 2 \leq 8 \text{ (True)}$$

$$(ii) P_1 \rightarrow \text{sum of need} \leq \text{available} \\ (1, 2, 2) \leq (3, 3, 2) \\ 5 \leq 8 \text{ (True)}$$

$$(iii) P_3 \rightarrow 2 \leq 8 (5, 3, 2) \\ 2 \leq 10 \text{ (True)}$$

$$(iv) P_0 \rightarrow (7, 4, 3) \leq (7, 4, 3) \text{ (True)}$$

$$(v) P_4 \rightarrow (4, 3, 1) \leq (7, 5, 3) \text{ (True)}$$

$$(vi) P_2 \rightarrow (6, 0, 0) \leq (7, 5, 3) \text{ (True)}$$

$$(vii) P_4 \rightarrow (4, 3, 1) \leq (10, 5, 5) \text{ (True)}$$

The system is in a safe state and safe sequence will be: $P_1 \rightarrow P_3 \rightarrow P_0 \rightarrow P_2 \rightarrow P_4$

(iii) If a request from process P_0 arrives for $(0, 2, 0)$ can the request be granted immediately?

→

(i) Check if request $\leq$ need for P_0

$$(0, 2, 0) \leq (7, 4, 3) \text{ (Yes)}$$

(ii) Check request $\leq$ available for P_0

$$(0, 2, 0) \leq (3, 3, 2) \text{ (Yes)}$$

Now, we must recheck safe state

Processes	Allocation	max	need	available
	A B C	A B C	A B C	A B C
P_0	0 3 0	7 5 3	7 2 3	3 1 2
P_1	2 0 0	3 2 2	1 2 2	5 2 3
P_2	3 0 2	9 0 2	6 0 0	7 2 3
P_3	2 1 1	2 2 2	0 1 1	7 5 3
P_4	0 0 2	4 3 3	4 3 1	10 5 5
				10 5 7

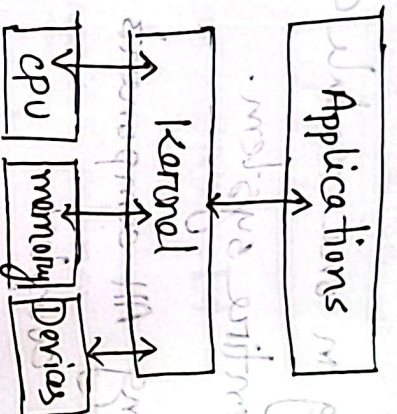
Safe seq: $P_3 \rightarrow P_1 \rightarrow P_0 \rightarrow P_2 \rightarrow P_4$

Still Safe.

So, request can be granted immediately.

Kernel in Operating System:

Kernel: A kernel is the core part of an operating system. It acts as a bridge between software applications and hardware of the computer.



Types of kernel:

1. Monolithic kernel: is a single large process running entirely in a single large space (kernel space). All OS services like memory management, process scheduling, device drivers etc. run in kernel mode.

Advantages:

(i) High performance: since everything runs in the same space, there's less overhead from context switching.

(ii) Faster system calls: Direct communication

within kernel space.

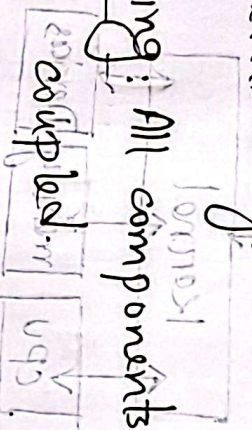
(iii) Efficient I/O operations:

Disadvantages:

(i) Poor modularity: Hard to maintain and update

(ii) Less secure: A bug in one module can crash the entire system.

(iii) Difficult debugging: All components are tightly



Example: Linux, unix, MS-DOS

Microkernel: A microkernel includes only the most essential components like low level address space management, thread management and IPC.

Advantages:

Modular design: Easier to update or replace components.

More secure and stable: Fewer services in user kernel space reduces crash risks.

→ Easier debugging and maintenance.

Disadvantages:

Performance: More context switching and IPC can slow things down.
Complex communication: Requires robust messaging systems.

Example: Minix, QNX, L4.

(iii) Hybrid kernel: combines aspects of monolithic and microkernels.

Advantages:

- Better performance than microkernel
- More stability and modularity than monolithic.
- Flexible architecture.

Disadvantages:

- Difficult to design and manage
- May carry drawbacks from both architectures.

Example: macOS.

(iv) Nano Kernel: A minimal kernel that provides only the basic CPU scheduling and hardware abstraction.

Advantages:

- Very small footprint
- Good for ~~external~~-time systems
- Low power/resource consumption

Disadvantages:

- Limited functionality
- Relies heavily on external services

(v) Exo Kernel: A minimal kernel that gives applications more direct access to hardware while keeping the security and resource allocation in check.

Advantages:

- Maximum flexibility and efficiency for applications
- Low overhead from abstraction layers.

Disadvantages:

- Complex to develop applications
- Lack of standard APIs.

Functions of kernel:

Process management: Scheduling and execution of process

Memory management: Allocation and deallocation of memory space, managing virtual memory, handling memory protection.

Device management: managing I/O devices, providing an interface for hardware devices and handling device driven communication.

File system: Managing file operations and providing

file system interface to applications.

Resource management: Allocating and deallocating resources as needed.

Security and access control: Enforcing access control policies (user permission, authentication).

Inter-process communication: Facilitating communication between processes by providing mechanisms (message passing, shared memory).

Working Principle of a kernel:

(i) System Boot-up:

- The BIOS runs first
- Bootloader loads the kernel into main memory (RAM).
- kernel initializes the system (CPU, memory etc).

(ii) Switch to kernel mode:

- system switches from BIOS to kernel mode
- kernel starts running

(iii) Initialize core components:

The kernel sets up:

Scheduler → manages which process runs next
memory manager → Allocates RAM.

File system → Manages data on storage devices.

Device drivers → Interface with keyboard, mouse etc.

(iv) Process Creation and management:

(v) Handling system calls:

- Applications can't access hardware directly instead, they ask the kernel to do it for

them through system calls.

(v) Memory management:

(iv) Shutdown and Restart:

When the system shuts down

→ The kernel stops all processes

→ Syncs file system.

→ Power off or reboots the system safely.

PCB

A process control block is a data structure used by the operating system to keep track of the process execution & information and manage execution.

Structure of PCB:

Pointer
Process state
Process Number
Process Counter
Registers
Memory limit
List of open files

Pointer → It's a stack pointer that is required to be saved when the process is switched from one state to another to retain the current position of the process.

Process state → It stores the respective state of the process.

Process number → Every process is assigned a unique ID known as PID.

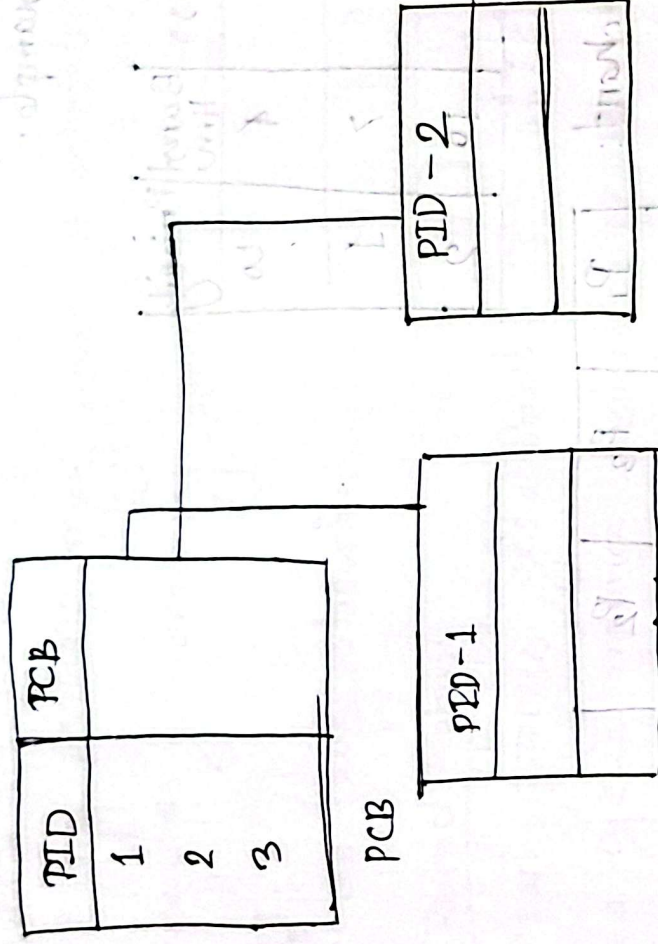
Program counter → Keeps track of the address of the next instruction to be executed in a program.

Register: storage location within the CPU used to temporarily hold data instructions, address or intermediate results during processing.

Memory limits → Contains the information about memory management system.

List of open files → This information includes the list of files opened for a process.

Process table:



Questions [Basic to deadlock]

Which scheduling algorithm could result in starvation? Justify your answer.

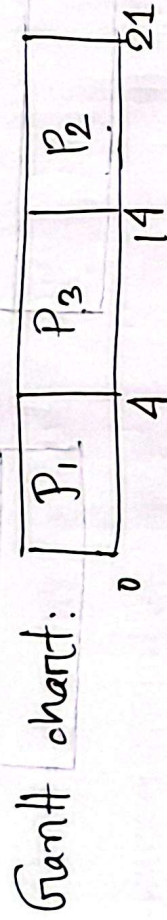
starvation happens when a process waits indefinitely to get CPU time because other processes are continuously getting scheduled before it.

→ This problem arises in heavily loaded systems where resources are always occupied by higher priority tasks, leaving some

processes starved

for example:

Process	Burst time	Priority
P ₁	4	0
P ₂	7	1
P ₃	10	2



As we see in the above example process P₁ has higher priority than other processes getting CPU earlier. If one process has priority less priority, this can lead to indefinite waiting for the process for CPU which has priority, which leads to starvation.

Analysis for each algorithm:

(i) FCFS: Definition: Processes are scheduled in the order of arrival.

→ So, no starvation.

(ii) SJF: Long processes may never get scheduled if short processes get arriving - specially in non-preemptive.

SJF.

→ Can cause starvation.

(iii) Round Robin:

→ Each process gets a fixed time slice (quantum) in a cyclic manner, ensures fairness - every process will get CPU time eventually.

→ No starvation.

(iv) Priority scheduling:

→ Low priority processes may never execute if high priority ones keep coming.

→ So, it can cause starvation.

∴ Scheduling algorithms can cause starvation.
are: SJF (Non-preemptive), priority scheduling.

Final (21-22)

1a) Discuss the relationship between hardware, OS, system programs, and application programs.

→

Hardware: Physical components of computer (CPU, RAM, disk, keyboard etc)

→ Hardware is the base layer
→ It does the actual computing but can't do much without software.

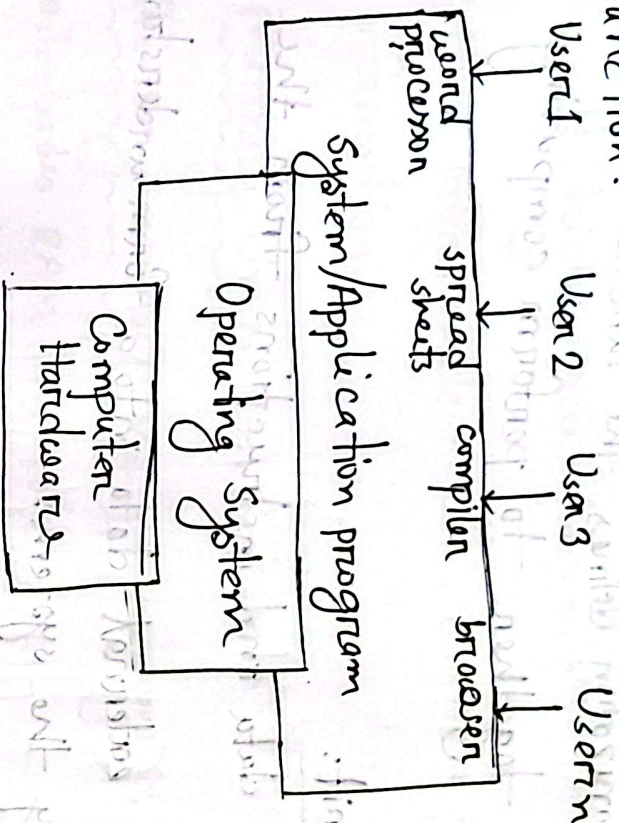
OS: Software that manages hardware and provides services to the other software
→ The OS sits on top of the hardware
→ It controls and manages access to hardware resources

System Programs: Software that supports the operating system and hardware interaction (compiler, file manager)

→ sit above the OS
→ Help manage and run the system efficiently.

(iv) Application Program: software for user tasks.

→ may rely on system programs and the OS to function.



Example: (MS word to write document)

→ User interact with ms word
→ MS requests to save file → calls the system Program
 (this is responsible for file manage-ment)

→ system program passes the request to the OS

→ OS communicates the hardware to write the file.

→ file saved to software with storing

program itself - software can't exist

2) What are the main components in a logical organization of a computer system?

→ Logical organization defines the functional blocks that work together to perform computing tasks.

(I) Input unit:

- receive data and instructions from the user
- Converts external data into a form understandable by the system.
- keyboard, mouse, scanner etc.

(II) CPU: The "brain" of the computer - logically divided into:

1. ALU (Arithmetic Logic Unit):

Performs all arithmetic (addition, subtraction) and logical operations (AND, OR, NOT)

2. Control Unit:

- Directs the instruction of the processor
- Fetches instructions from memory.

decodes them, tells the ALU, memory and I/O devices what to do.

3. Registers:

- small, high speed storage location within the CPU.
- Hold intermediate results, counter and control information.

(iii) Memory Unit: Divided into:

1. Primary memory (main memory):

- Includes RAM and ROM
- stores data and instructions currently being used.
- Fast but volatile.

2. Secondary memory:

- Includes hard drives, SSDs, USB drives etc.

→ Used for long term storage

- slower but non-volatile.

(iv) Output Unit:

- Presents the processed data to the user.
- Converts binary results into human readable form.

of form: primitive ni yod 20 int 6000 data factor (ii)

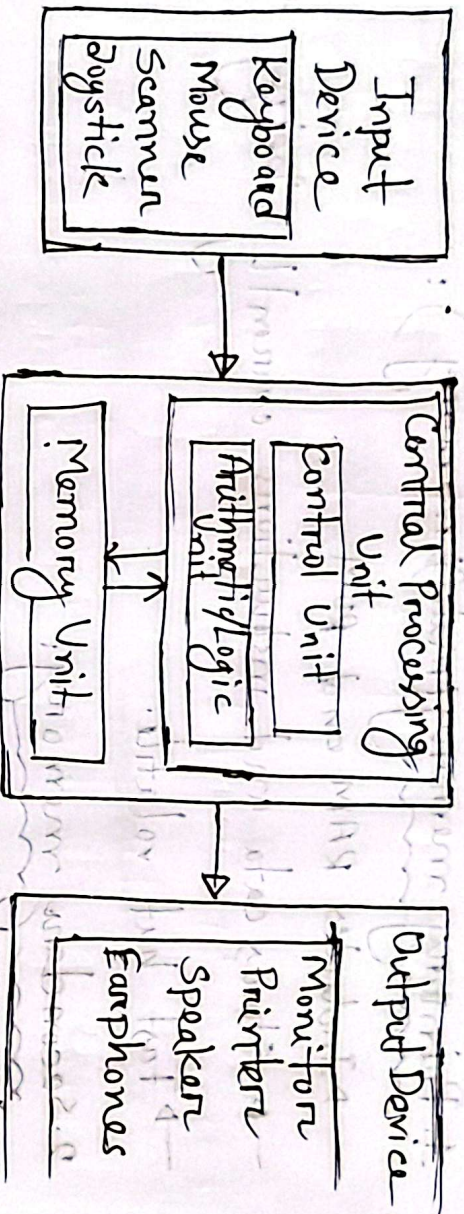
→ Monitor, printer etc.

(v) System Interconnections (Buses):

Buses are communication systems that transfer data between components.

Data buses: Carries data

Address : memory address
Control Bus : control signal.



(d)

An office employee rely on computers for tasks like writing reports, sending emails, and managing databases.

An employee turns on computer. But there's problem (ii)

OS installed just the hardware.

(i) What challenges he/she face when using a computer without OS.

(ii) What role does the OS play in providing access to applications and files.

(1) Without an operating system the employee would face serious limitations. They are:

1. No user interface: No desktop, icons, menus or command line to interact with.

2. Can't run applications: Programs like MS word, email clients, databases need the OS to run.

3. No file management: Can't save, open or manage files without a file system provided by the OS.

4. No hardware Control: Keyboard, mouse, monitor, printer won't function properly without OS drivers.

5. No multitasking: Can't run or switch between multiple programs.

6. No network access: Can't send or receive emails - OS handles network protocols and drivers.

(ii) ~~Role of~~ Role of OS in providing access to applications:

→ The Operating System manages the execution of application programs by loading them into memory

and assigning necessary system resources like CPU time, memory and I/O devices.

→ It ensures multitasking - allowing users to run multiple applications at once.

→ The OS provides APIs (Application Programming Interfaces) that help applications interact with hardware without knowing the hardware details.

→ Ensures that one application does not interfere with others.

Role of operating system in providing access

to files:

→ The OS provides a file system to organize and manage data stored on storage devices like hard drives, SSDs, etc.

→ It allows users and applications to create, read, write, rename and delete files easily.

→ It maintains directories or folders - keeps track of file locations and manages

metadata.

→ The OS also enforces file access permissions to ensure data security and user privacy.

2)c)

Process	Burst time	Priority
P ₁	5	3
P ₂	3	2
P ₃	9	4
P ₄	2	1
P ₅	6	2

(i) Preemptive priority scheduling (large priority num = higher priority and RR (quantum = 5))

(ii) Minimum avg waiting time in every scheduling algorithm.

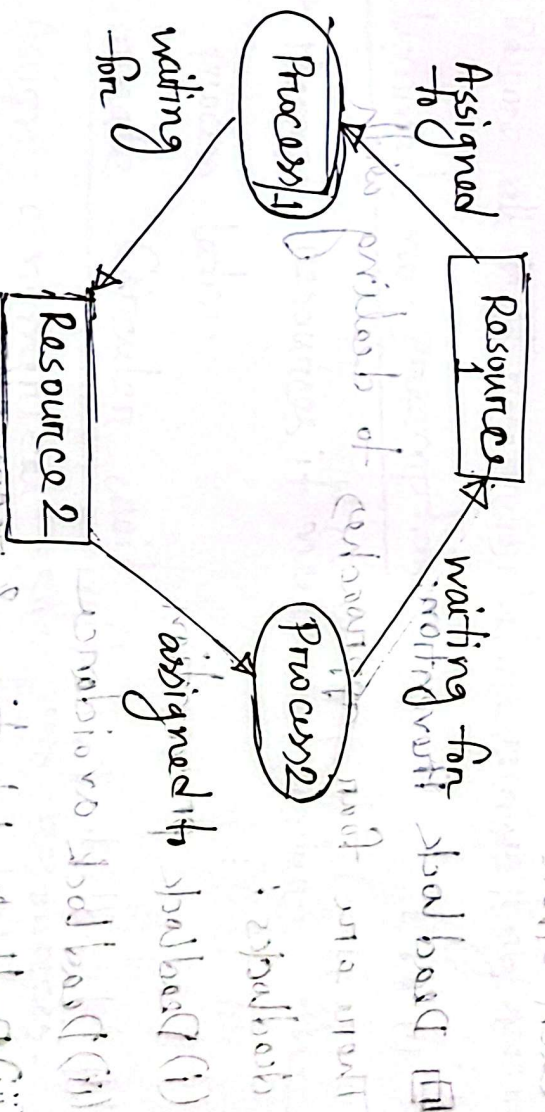
3a) What is deadlock? Discuss various strategies used to prevent it from occurring.

→ Deadlock: A deadlock is a situation in competing environment where a set of processes get permanently stuck because each process is waiting for a resource held by another process and none of them can proceed.

Example:

→ P_1 holds resource R_1 and requests R_2

→ P_2 holds resource R_2 and requests R_1 .
Neither process can proceed causing a deadlock.



Deadlock can arise if following four conditions hold simultaneously:

(i) Mutual Exclusion: Only one process can use a resource at any given time.

(ii) Hold & Wait: A process is holding one resource at a time and is waiting to acquire other resources held by some other processes.

(iii) No Preemption: A resource can't be taken from a process unless it releases the resource.

(iv) Circular wait: Set of processes waiting for each other in a circular fashion.

Deadlock Prevention:

There are four approaches to dealing with deadlocks:

- (i) Deadlock prevention
- (ii) Deadlock avoidance
- (iii) Deadlock detection & recovery
- (iv) Deadlock ignorance

(i) Deadlock Prevention: is a method used to ensure that different processes can run smoothly without getting stuck waiting for each other forever.

To prevent deadlock, we can eliminate any of these conditions (pre-mutual exclusion, hold & wait, no preemption, circular wait) that causes deadlock.

1. Eliminate mutual exclusion:

→ Use shareable resources ^{state} whenever possible (like read-only files)

2. Eliminate hold & wait:

→ Request all resources before a process starts.
→ Release all held resources before requesting new ones.

3. Eliminate no-preemption:

→ Preempt resources if needed, restart the process later.

4. Eliminate Circular wait:

→ Assign a numerical order to resources.

→ Process can ~~at~~ only request resources in increasing order.

(ii) Deadlock Prevention

Avoidance:

is a method where

The OS makes decisions dynamically to ensure the system never enters an unsafe state. This is usually done by two algorithms like:

(1) Banker's algorithm:

→ Used when multiple instances of resources exist.

→ It simulates allocation and ensures that the system remains in a safe state before granting resources.

(2) Resource Allocation graph Algo:

→ Used when each resource has only one instance.

→ It checks for cycles in a graph to avoid unsafe state.

(iii) Deadlock Detection & Recovery:

Deadlock detection is a method that periodically checks for circular waits and resolves

by aborting, and restarting processes, releasing their resources

Deadlock recovery is the process of handling a deadlock after it has occurred. It involves:

(i) Process termination: Aborting one or more deadlocked processes to break the cycle.

(ii) Resource Preemption: Temporarily taking resources from some processes and reallocating them to others.

(iii) Rollback one of the processes to an earlier safe state.

(iv) Deadlock Ignorance:

→ OS acts like the deadlock never occurs and completely ignores it even if the deadlock occurs.

→ Only applies if deadlock occurs very rarely.

notes on deadlock A: deadlock prevention

steps to avoid a deadlock

1. request must not increase the number of instances of a resource beyond the maximum number of instances of that resource.

(b) Illustrate how resource allocation graph can be utilized to identify deadlocks in a system.

→ RAG: is a visual way to understand how resources are assigned in an operating system.

→ RAG shows which resource are allocated and which are requested by the processes

→ It helps to visualize deadlocks more clearly than tables

Graph Components:

→ Processes ($P_1, P_2, \dots, P_n$): represented as circles

→ Resources ($R_1, R_2, \dots, R_n$): represented as squares

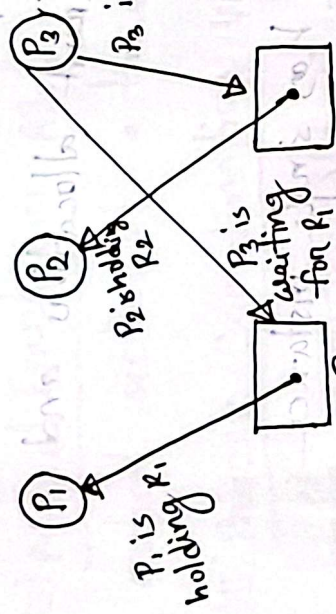
→ Request edge: from process to resource ($P \rightarrow R$)

→ Assignment edge: from resource to process ($R \rightarrow P$)

Identifying Deadlocks: A deadlock occurs when there is a circular wait - i.e., a cycle in the graph where:

- Each process holds at least one resource
- Waiting for another resource held by another process in the cycle.

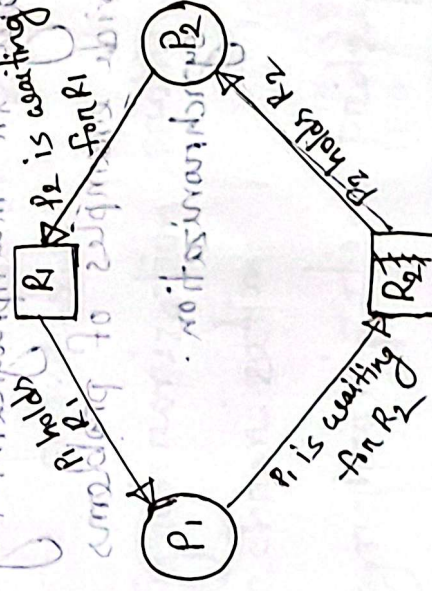
Example-1: (Without deadlock)



- Process P_1 and P_2 acquiring resources R_1 and R_2 while process P_3 is waiting to acquire both resources.

- There is no deadlock because there is no circular dependency.

Example-2: (Deadlock situation):



→ This creates a cycle and since each resource

has only one instance, this is a deadlock.

→ P₁ holds R₁ and requests R₂

→ P₂ holds R₂ and requests R₁

Detecting Deadlock With RAG:

→ Build graph using current allocation and requests

→ Search for cycles:

→ If the system has single instance resources, a cycle implies deadlock.

→ If multiple instances exist a cycle may imply deadlock.

3(c) → thread after

4(a) What is process synchronization and why

it is necessary in multiprogramming environ-

ment? Provide examples of problems caused

by lack of synchronization.

→

III Importance of Process Synchronization in multiprogramming environment:

(i) Preventing Race Conditions: Ensures processes don't access shared data at the same time avoiding inconsistent result.

(ii) Mutual Exclusion: Allows only one process in the critical section at a time.

(iii) Process Coordination: Let's processes wait for specific conditions (e.g., producer-consumer)

(iv) Deadlock Prevention: Avoids circular wait and indefinite blocking by using proper resource handling.

(v) Safe Communication: Ensures data/messages between processes are sent/received and processed in order.

(vi) Fairness: Prevents starvation by giving all processes fair access to resources.

IV Process Synchronization problem caused by lack of synchronization:

The process synchronization problem arises when

multiple processes access shared resources (memory, files, I/O devices) concurrently without proper coordination. This can lead to

(I) Data inconsistency

(II) Race condition

(III) Deadlock

(I) Data Inconsistency: When two or more processes access and modify data simultaneously, the final result can become inconsistent or incorrect.

Example: Two processes P_1 and P_2 are updating a shared counter n .

$\text{int } n = 5;$ // shared variable

P_1 and P_2 code executed by both P_1 and P_2

Here, P_1 reads $n = 5$

$\rightarrow P_2$ reads $n = 5$

$\rightarrow P_1$ computes $n+1 = 6$ stores it

→ P_2 computes $n+1=6$, stores it again
Final value of n is 6, but it should be 7.
This is an inconsistency caused by concurrent access without synchronization.

(ii) Race Condition: occurs when two or more processes race to perform operations on shared data and outcome depends on the timing or sequence of their execution.

Example:

int balance = 100; // shared variable

temp = balance // P_1

temp = 60

balance = temp // balance = 40

temp = balance; // P_2 [temp = 100 old value because P_1 hasn't

temp = temp - 50
updated yet]

balance = temp // balance = 50

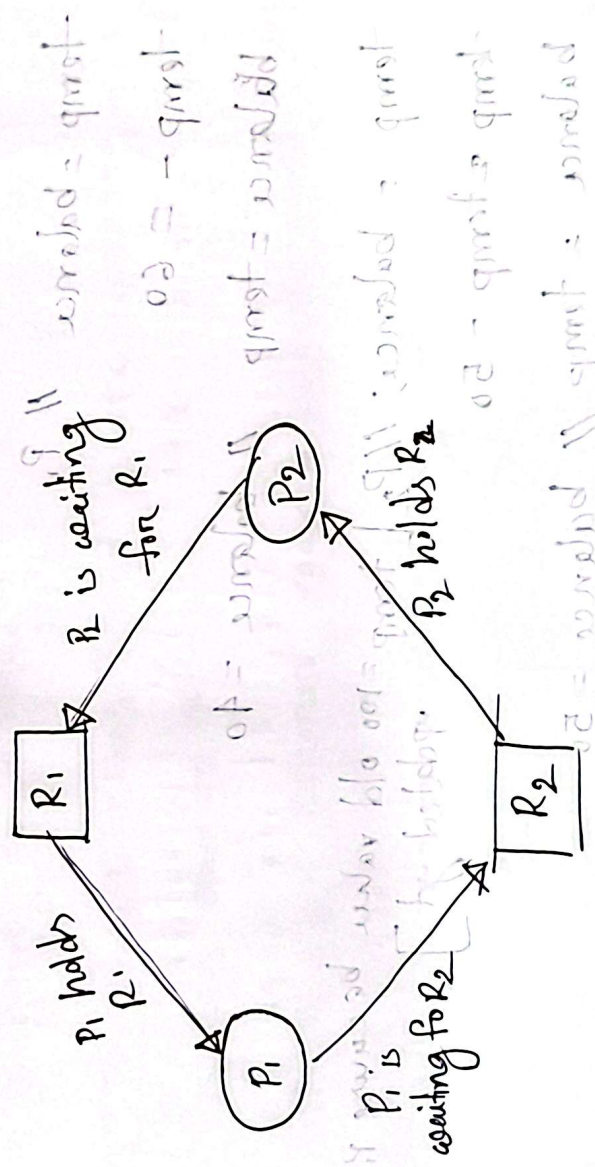
Final balance becomes 50 but it should be
 $10 - 60 - 50 = -10$ or system should prevent overdraw.

Here, updates overlap - a classic race condition.

(iv) Critical section: A critical section is a part of the code where a process accesses shared resources.

Only one process should enter the critical section at a time to avoid conflicts.

(v) Deadlock: A deadlock occurs when two or more processes are waiting for each other to release resources but none of them ever do. As a result, they get stucked.



Here:
→ P1 holds R1 and waiting for R2
→ P2 holds R2 and waiting for R1

forms a cycle - classic deadlock.

(b) Describe how semaphores are used for process synchronization. Illustrate with an example of the producer-consumer problem.

→ Semaphore: A semaphore is simply a variable used to control access to a shared resource by multiple processes in a concurrent system.

Types of semaphores:

(i) counting semaphore: Used when multiple resources instances of a resource exist.

(ii) Binary Semaphore: Special case of binary semaphore with only two values: 0 and 1

A semaphore works using two fundamental operations:
(i) wait (P operation/down)

→ Decreases the semaphore value.

→ If value becomes negative, the process is blocked until the resource becomes available.

(ii) signal: (v operation/up)

→ Increases the semaphore value.

→ If there are waiting processes, one of them gets unblocked.

wait (Semaphore S) {

while (S == 0);

S = S - 1; // acquire resource

}

// Critical section

signal (Semaphore S) {

S = S + 1; // release resource

}

How Semaphore are used for PS:

To achieve synchronization, every critical section of code is surrounded by two operations - wait and signal.

Example: Two processes P₁ and P₂ sharing a Semaphore S, initialized to 1.

State-1: Both processes are in their non

critical section and $s=1$.

state-2: P_1 enters critical section. Calls wait

$s=0$ and P_2 continues in non critical section.

state-3: If P_2 wants to enter, it can't proceed

since $s=0$. It must wait until $s>0$.

state-4: When P_1 finishes, it performs signal(s), making

$s=1$, now P_2 enters in critical section and

sets $s=0$ again.

▣ Solution of Producer-Consumer Problem using sema-

-phore:

(5)(i) Identify and explain four necessary condition for deadlock. Which of these are satisfied in the above scenario.

→ Deadlock can arise if the following four conditions hold simultaneously:

1. Mutual Exclusion: Only one process can use a resource at any given time. i.e. the resources are non shareable.
2. Hold and Wait: A process is holding at least one ~~resource~~ ^{resource} at a time and is waiting to acquire other resources held by some other process.
3. No preemption: A resource can't be taken from a process unless the process releases the resource.
4. Circular Wait: Set of processes are waiting for each other in a circular function.

In given scenario, printer A and printer B are non shareable. So, it satisfies the mutual exclusion condition of deadlock. Also

Mutual Exclusion $\rightarrow$ satisfied because each printer is non-sharable. (i)(e)

Hold & Wait $\rightarrow$ satisfies because Alice holds printer A and waits to print B; Bob holds printer B and waits for printer A.

No preemption $\rightarrow$ satisfied because the system doesn't preempt resources. W has a printer.

Circular wait $\rightarrow$ satisfied because a set of processes are waiting for each other in a circular chain.

Here, all four conditions are satisfied, hence a deadlock has occurred.

Two E waiting for A printing resources waiting for resources but not waiting for A to print resources.

all A should be waiting for resources

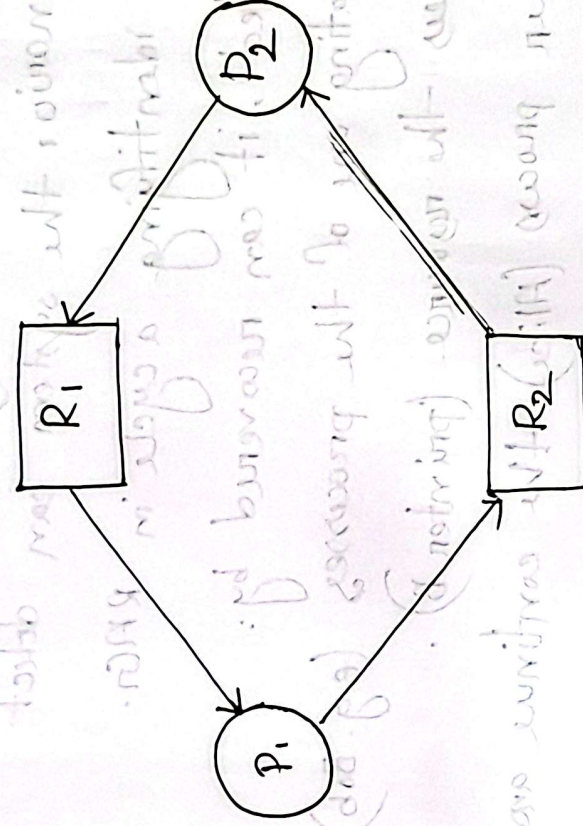
(ii) Draw RAG to represent the above scenario.
Based on your graph, explain how a cycle implies a deadlock.

→ RAG: Resource Allocation Graph

processes: P_1 and P_2

↓ Alice
↓ Bob

Resources: R_1 (Printer A), R_2 (Printer B)



→ P_1 holds R_1 and requests R_2

→ P_2 holds R_2 and requests R_1

No process in the cycle can proceed - they are

all waiting on each other.

The graph forms a cycle: $P_1 \rightarrow P_2 \rightarrow P_1$

The presence of a cycle in a RAG indicates a deadlock.

(iii) Describe how a deadlock detection and recovery approach would handle this deadlock.

Deadlock detection and recovery: prev

In this scenario, the system can detect deadlock by identifying a cycle in RAG.

Once detected, it can be recovered by:

→ Terminating one of the processes (e.g. Bob), which frees the resource (printer B).

→ The other process (Alice) can then continue and complete its task.

(iv) Modify the system such that it uses the Banker's algo. Would the system have entered the deadlock in that case? Explain ensuring a safe state with example.

8)c. Explain different types of CPU schedulers.

→ Process Scheduler: are fundamental components of an operating system responsible for deciding the order in which processes are executed by the CPU.

There are three types of process scheduler.

(1) Long term scheduler (Job Scheduler): LTS loads a process from disk to main memory for execution. The new process to the 'ready state'.
→ It mainly moves processes from job arrival to ready queue.

→ It controls the degree of Multi-programming.

→ LTS make a careful selection of both I/O and CPU bound processes. It increases efficiency by maintaining a balance between the two.

→ Slowest among the three, that's why called long-term.

(1) Short-term or CPU scheduler: CPU scheduler

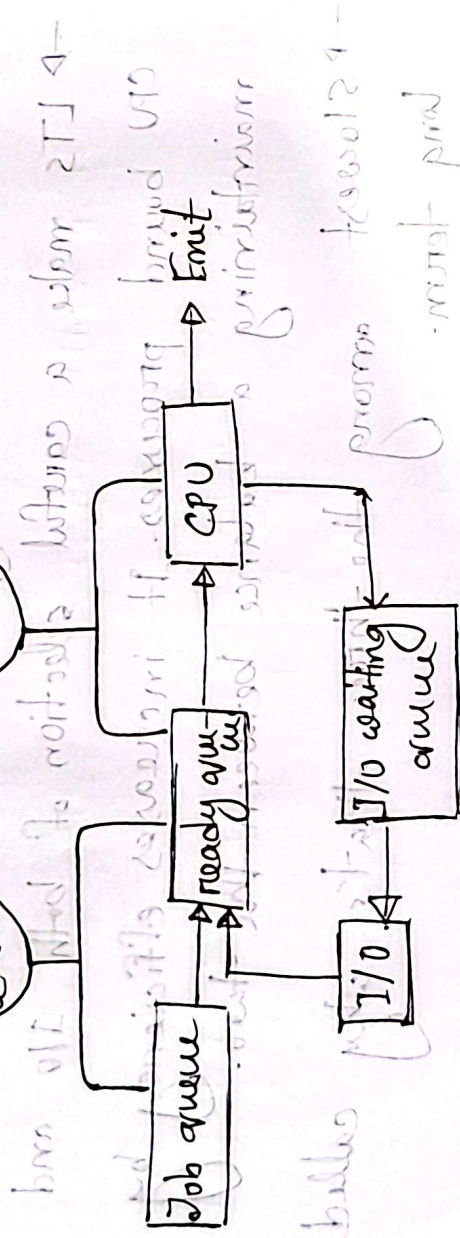
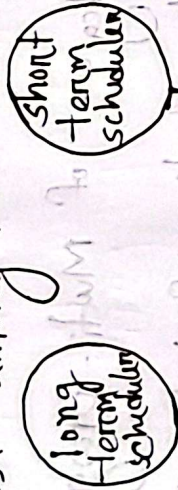
is responsible for selecting one process from the ready state for running (or assigning CPU to it).

→ STS must select a new process for the CPU frequently to avoid starvation.

→ It uses different scheduling algorithms to balance the allocation of CPU time.

→ Its main objective is to make the best use of CPU.

→ Fastest among the three.

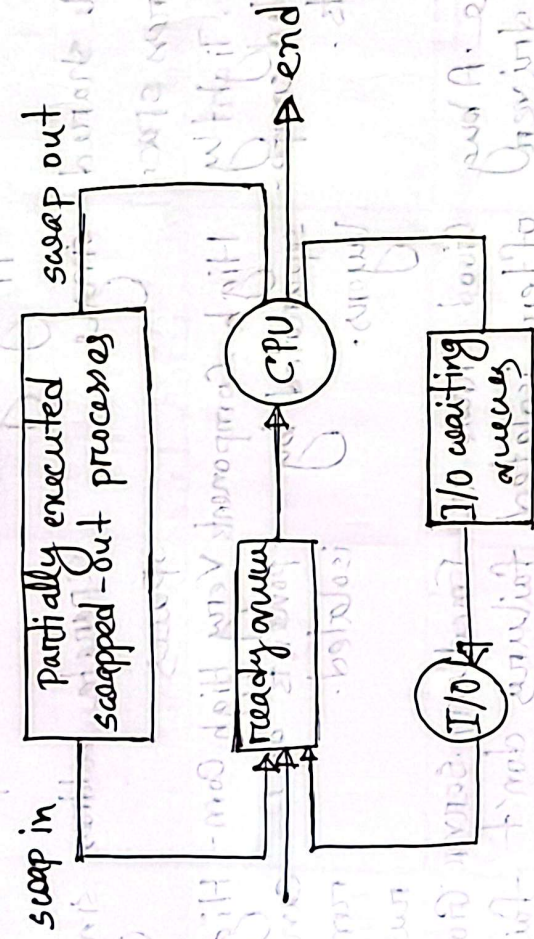


(iii) Medium term scheduler: is responsible for moving a process from memory to disk.

- It reduces the degree of multiprogramming
- A running process may become suspended if it makes an I/O request. suspended process is

moved to the secondary storage.

- It brings processes back into memory and pick up right where it left off.
- It is faster than long term and slower than short term.



Medium-term scheduler

Final (20-21)

1(b) Compare monolithic, layered, microkernel and modular operating system structures. How does each structure organize components and what are the main advantages and disadvantages? Provide example.

Characteristic	Monolithic	Layered	Microkernel	Modular
Kernel size	Large (All OS services reside in it).	Variable. The core kernel may be large, but the overall structure is divided.	very small (only essential services - cs)	Large core kernel that can expand dynamically.
2. component location	All services are in kernel space.	Components are assigned to strict, non-overlapping layers;	Essential services in kernel space, non-essentials are in user space.	core services in kernel space. New services loaded as kernel modules.
3. Address space	Single shared address space.	single.	separate address spaces.	single
4. Modularity	Low. Tightly coupled components.	High. Components separated by layers.	Very High. Components are isolated.	High. New features can be added or removed at runtime.
Fault isolation	None. A bug in a driver can crash the entire OS.	Good. Errors are often isolated to the layer where they occur.	Excellent. Server failures don't crash the minimal core kernel.	Good. Module failure might crash only that module.

Organization of components:

1. Monolithic structure:

- All OS components reside in a single large program running entirely in kernel mode.
- Share same address space
- Communication is straight forward.

2. Layered Structure:

- Organizes components into a strict hierarchy.

3. Microkernel Structure:

- Is a tiny core that enforces protection
- Handles IPC.

4. Modular Structure:

- Keeps the basic kernel core small but allows for dynamic loading.

advantages:

flexibility: System features can be ^{added} ~~loaded~~ in on the fly without a reboot.

Good Performance: Core services operate with monolithic.

efficiency.

Better resource use: Only necessary modules are loaded into memory.

Disadvantages:

Less structured: Less formal interface control.

~~Potential~~ → Less potential for complexity.

(c) Explain the key differences between batch operating systems multiprogramming operating system and multitasking operating systems. Specially describe how each system manages processes, CPU utilization and user interaction. Provide examples of scenarios on use cases where each type of operating system should be ~~more~~ most effective.

☐ Differences in OS structures:

Feature	Batch OS	Multiprogramming OS	Multitasking OS
1. Primary goals	Maximum throughput (no. of jobs finished)	Maximum CPU utilization and throughput.	Maximum user responsiveness and CPU utilization.
2. Process management	Jobs are grouped into batches and executed sequentially without user intervention.	The OS keeps multiple jobs in memory. When one job waits for I/O, the CPU switches to another ready job.	The OS rapidly switches the CPU between multiple processes, giving each a short burst of time.
3. CPU Utilization	Low. CPU is idle when the currently running jobs perform I/O operations.	High. CPU is almost always busy working on different jobs while others wait.	High. The CPU is continuously utilized by switching among interactive processes.
4. Job scheduling	Simple FIFO within a batch.	Uses job scheduling (long-term) and CPU scheduling (short-term) to select which job runs next.	Uses CPU scheduling with a time sharing mechanism (eg. round robin)

— under doc background differe nces
— ut and timelike doc no id pities

• ubi up
griq shov ed naitaditu upo cosimiam t—

Effective scenarios and use cases:

(I) OS:

- Idle for long running jobs that don't require any human interaction once they begin.
- High throughput for similar non-interactive tasks.
- Eliminates setup time between jobs.

Examples:

Billing and payroll systems: Generating monthly

payrolls for thousands of customers.

scientific simulations: running large-scale

weather models or complex mathematical

calculations.

(II) Multiprogramming OS:

- Foundational OS concept for environments with multiple background jobs where waiting for one job shouldn't leave the CPU idle.
- Maximizes CPU utilization by overlapping

I/O wait time with the execution time of another processes.

Examples:

Mainframe systems: Used in data centers

Early Servers: Running a database query and a file backup concurrently with a computation job.

3. Multitasking OS:

- Essential for interactive systems where a single user runs several programs simultaneously.
- Provides the illusion of parallel execution

Example:

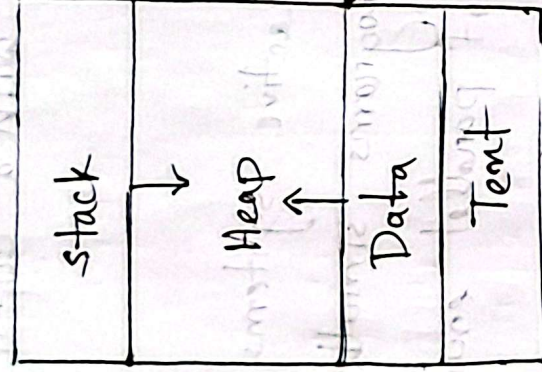
Personal Computers: Running a web browser, word processor and music player at the same time.

2a) What is process in an OS and how does it differ from a program?

→ Process: A process is an instance of a program in execution. For example, when we write a program in C or C++

and compile it, the compiler creates binary code. The original code and binary code are programs. When we actually run the binary code, it becomes a process.

How a process looks in memory:



Text section: contains executable instructions.

It is typically a read only section.

Stack: Contains temporary data, such as function parameters.

Data section: Contains the global variables.

Heap section: Dynamically memory allocated.

to process during its run time.

Attributes of process:

1. P-ID: A unique number assigned to each process so the OS can identify it.
2. Process state: Shows the current status of process (running, waiting)
3. Priority and other CPU scheduling information: Data that helps the OS decide which process should run next.
4. I/O information: Information about I/O devices the process is using.
5. File Descriptors: Info about open files and network connection.
6. Accounting info: tracks how long the process has run.
7. Memory management info: Details about the memory space allocated to the process.
 (Process memory management for OS necessary)
 (Process memory management)

□ Differences between program and process:

Program	Process
1. A passive set of instructions 2. Static 3. Stored in disk 4. Unlimited lifetime 5. It has no associated OS control structure. 6. A single program can generate multiple independent processes	1. An active instance of a program. 2. Dynamic 3. Stored in RAM 4. Limited lifetime 5. Managed by the PCB, which stores all its state info. 6. Each process is a unique instance of execution, with its own resources and states.

2) a) Process Synchronization? Why is it necessary?

and what problems can arise if processes are not properly synchronized?

② → previous.

(c) Outline the lifecycle of process, describing each state and the events that trigger transitions between these states.

→ process states with figure (previous)

Process states transitions:

(i) New → Ready (Event triggered by: Admitted)

The OS has successfully completed initial setup for the process and moved it from secondary storage to main memory.

(ii) Ready → Running (Event triggered by: dispatcher)

The OS process scheduler selects the process from the ready queue and allocates the CPU to it.

(iii) Running → Ready (Event triggered by: Timer interrupt)

The process's allocated time-slice expires or a higher priority process is now available. The OS performs a context switch to preempt the current process.

(iv) Running → waiting (I/O or event wait)

The process initiates an action that requires waiting (e.g., waiting for user input)

(iv) Waiting → ready: (I/O or event completion)

The external event the process was waiting for (e.g., input received) has completed. The process is moved back to ready queue.

(v) Running → terminated: (Exit)

The process finishes executing its last instruction or terminated by the OS due to an error (abnormal exit/killed).



3a) Process scheduling? The role of scheduler in managing processes. How do different scheduling algorithms affect process performance and CPU utilization.



Process Scheduling is the activity of the OS that decides which process in the ready queue should be executed next by the CPU. Since the CPU can execute one process at a time, the OS uses

scheduling to ensure efficient execution, maximize CPU utilization and provide a responsive system.

Q The role of scheduler in managing processes:

Definition → prev

Key functions of the scheduler:

Decision Making: Decides which process to be executed next.

State transition management: Manages the movement of processes between different states. (ready, running)

Context switching: When a process is selected to run the scheduler initiates a context switch. Which involves saving the state of the currently running process and loading the saved state of the process to be run next.

Q Types of scheduler and their function:
→ previous

Q How scheduling algos affect process performance and CPU utilization:

Q Scheduling algos are the rules the scheduler uses to make its decisions.
(1) Maximize CPU utilization

(II) Maximize throughput

(III) Minimize turnaround time

(IV) " waiting time

(V) " response time

Major scheduling algos and their effect.

(I) FCFS: Processes are executed in the order they arrive

Effect: Simple to implement.

→ Can lead to a long average waiting time

→ Poor turnaround time

→ Lowering CPU utilization as subsequent processes are unnecessarily delayed.

(II) SJF: The process with shortest execution time is

run next.

Effect:

→ Optimal for minimizing cost and lat.

→ Leads to high throughput and excellent CPU utilization.

→ Can cause starvation for very long processes.

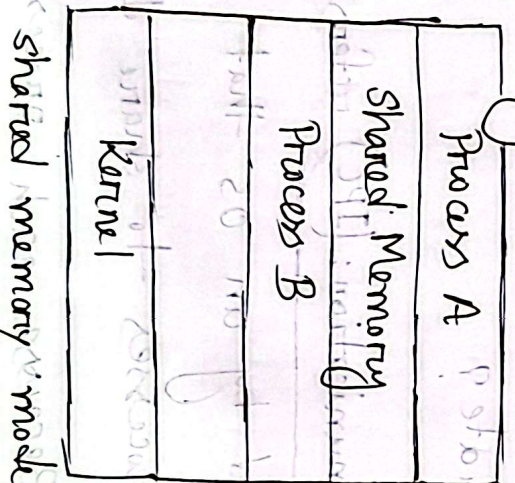
own isolated memory space for security and stability.

IPC is essential for processes to collaborate, share resources and build complex, modular systems.

Common IPC mechanisms and their appropriate scenarios:

IPC mechanisms are generally categorized into two models: Message passing and shared memory.

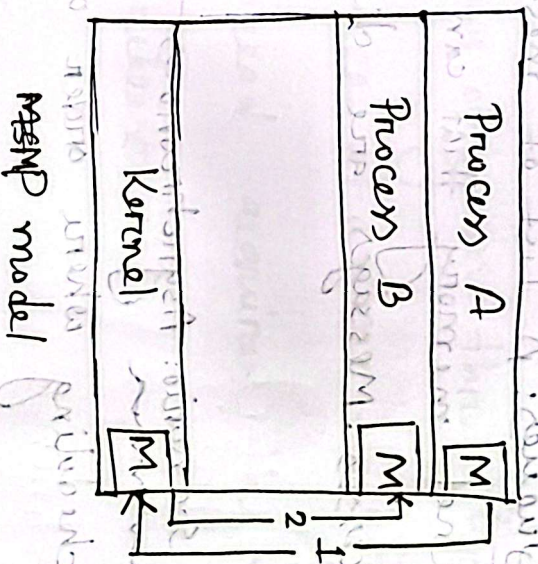
(4) Shared Memory: IPC through shared memory is a method where multiple processes are given access to the same region of memory. This shared memory allows the process to communicate with each other by reading and writing data directly.



Appropriate scenario:

High performance computing, real-time data processing or applications that need to share very large amount of data (e.g. passing a high resolution video frame) between processes on the same machine.

2) Message Passing: IPC through Ms is a method where processes communicate by sending and receiving messages to exchange data. It can be achieved



It can be achieved through different methods:

(i) Pipes (Anonymous or Named)

A simple unidirectional communication channel that connects the output of one process to the

input of another.

Anonymous Pipes: Only worked between related processes (e.g. parent and its child processes)

Named Pipes: Appear as a file in the file system, allowing communication between unrelated processes.

Appropriate Scenario: Simple data streaming or creating shell pipelines.

(ii) Message Queues: A List of messages stored within the OS kernel memory that can be accessed by multiple processes. Messages are discrete - structured data block.

Appropriate Scenario: Asynchronous task coordination and job scheduling where order or message type

is important.

(iii) Network Sockets: An endpoint for sending or receiving data across a network. It supports various protocols like TCP and UDP.

Appropriate scenario: Client-server architecture, microservices or any application that needs to communicate across a network.

c) Synchronization? Why is it important in an OS? Describe common synchronization mechanisms (e.g. mutex, semaphore) and explain how they prevent issues like race condition.

→

Mutex: (Mutual Exclusion Object)

A mutex allows only one thread/process to access a shared resource at a time. It acts like a lock - when one thread acquires the mutex, others must wait until it releases it.

How it works:

(i) Thread A locks the mutex before accessing the shared resource.

(ii) Thread B tries to lock the same Mutex but

must wait until the thread A unlocks it.

(III) After thread A finishes an and unlocks the mutex.

Thread B can lock it and proceed.

(IV) Since only one thread can hold the mutex at any time, simultaneous accesses to shared data is impossible - ensuring mutual exclusion.

Example:

`mutex = lock(m);`

`shared_counter = shared_counter + 1;`

`mutex = unlock(m);`

Here even if two threads try to increment

shared_counter, only one executes the critical section at a time.

RC Prevention:

Synchronization mechanism prevent race conditions by enforcing mutual exclusion over the critical section.

section: M uses self lock of right B has not (1)

1) Prevention using Mutex: Enforces that only one thread and can execute the critical section at a time.
For example,

Thread A acquires the mutex, preventing Thread B from entering the critical section until Thread A releases it. This makes the sequence of updates deterministic, eliminating the race.

2) Prevention using Semaphore: Controls the number of threads that can access a resource.

A binary semaphore is waited on before the critical section and signaled afterward ensuring serial access.

1) a)

Process	Arrival-time	Burst-time
P ₁	0	6
P ₂	2	3
P ₃	4	4
P ₄	6	2

Use the round robin scheduling algorithm with a time quantum of 3 units to answer the following question.

(i) For round robin calculate waiting time, turnaround time and completion time for each process.

→

(i) P₁ → for t=0 to 3 [6-3=3 left]

(ii) P₂ → for t=3 to 6 [3-3=0 left] (completed)

(iii) P₃ → for t=6 to 9 [4-3=1 left]

(iv) P₄ → for t=9 to 11 [completed]

(v) P₁ → for t=11 to 14 [completed]

(vi) P₃ → for t=14 to 15 [completed]

Process	Burst-time	Arrival-time	Completion-time	Turnaround-time	Waiting-time
P ₁	6	0	14	14 - 0 = 14	14 - 6 = 8
P ₂	3	2	6	6 - 2 = 4	4 - 3 = 1
P ₃	4	4	15	15 - 4 = 11	11 - 4 = 7
P ₄	2	6	11	11 - 6 = 5	5 - 2 = 3

(ii) How does the time quantum affect the waiting time and turnaround time compared to other scheduling algo such as FCFS or SJF?

→

The time quantum in RR scheduling decides how long each process runs before being switched out.

Effect of TQ on RR:

(i) Very large: ($\infty \rightarrow \infty$)

→ Each process gets the CPU for a long time before switching

→ The algorithm starts behaving like FCFS.

→ Waiting time and TAT increases specially

for short processes.

(II) Very small time quantum ($\alpha \rightarrow 0$)

→ Processes are switched more often, so every process gets the CPU quickly.

→ Response time improves, but too many context switches cause overhead, which can increase total waiting time slightly.

Comparison to FCFS and SJF.

1. RR vs FCFS:

→ In FCFS, processes are executed in the order they arrive. It is simple to use but can cause the convoy effect, where short processes wait for a long process to finish.

As a result, both waiting time and TAT can be quite high.

→ But RR gives each process a fair share of CPU time, which reduces waiting time.

for later processes and improves response time.

2. RR vs SJF:

→ In SJF the process with the shortest job runs first. This algo provides the lowest avg waiting time and tat but can lead to starvation for long processes if short ones keep arriving.

→ RR avoids starvation by giving each process equal CPU time in a cyclic order. Although RR may not minimize waiting and tat as much as SJF, it offers better fairness specially in time sharing system.

(iii) Compare the performance of RR with a tat of 3 units to a tat of 2 units. Which setting provides better system performance and why?

for time quantum 3 units:

$$\text{Avg waiting time} = \frac{8+1+7+3}{4}$$

$$= 4.75$$

$$\text{Avg TAT} = \frac{14+4+11+5}{4}$$

$$= 8.5$$

For time quantum 2 units:

$P_1 \rightarrow t=0 \text{ to } 2$ [4-2 = 4 units left]

$P_2 \rightarrow t=2 \text{ to } 4$ [3-2 = 1 unit left]

$P_3 \rightarrow t=4 \text{ to } 6$ [4-2 = 2 units left]

$P_4 \rightarrow t=6 \text{ to } 8$ [completed]

$P_1 \rightarrow t=8 \text{ to } 10$ [4-2 = 2 units left]

$P_2 \rightarrow t=10 \text{ to } 11$ [completed]

$P_3 \rightarrow t=11 \text{ to } 13$ [completed]

$P_4 \rightarrow t=13 \text{ to } 15$ [completed]

Process	Burst time	Arrival time	completion time	waiting time	TAT
P_1	6	0	15	15-6=9	15-0=15
P_2	3	2	11	9-3=6	11-2=9
P_3	4	4	13	9-4=5	13-4=9
P_4	2	6	8	2-2=0	8-6=2

$$\text{Avg waiting time} = \frac{9+6+5+0}{4}$$

$$\text{Avg TAT} = \frac{15+9+9+2}{4}$$

$$= 8.75$$

Here, time quantum 3 has the lesser time and avg TAT. So, time quantum 3 units will give better performance.

avg waiting
waiting

Reason: A time quantum of 3 units generally provides better system performance, because it provides good balance between responsiveness and efficiency. A smaller quantum (like 2 units) improves response time but wastes more CPU time on frequent switching, while a longer one reduces overhead and keeps TAT and ~~wait~~ waiting times reasonable.

— , —

exhibits very poor

waiting time

c) (i) CPU Dispatcher: Dispatcher is a special type of program whose work starts after the scheduler. When the scheduler completes its task of selecting process, it is the dispatcher that moves the process to the queue it needs to go to. The following things happen in this function:

(i) switch context: Saves the current process and restores the state of the process to be run next.

(ii) switching to user mode: Makes sure that it runs in the user mode and not kernel mode.

(iii) Jumps to the correct location in the user program from where the program can be restarted.

Advantages:

- Fast process switching
- Efficient CPU time allocation

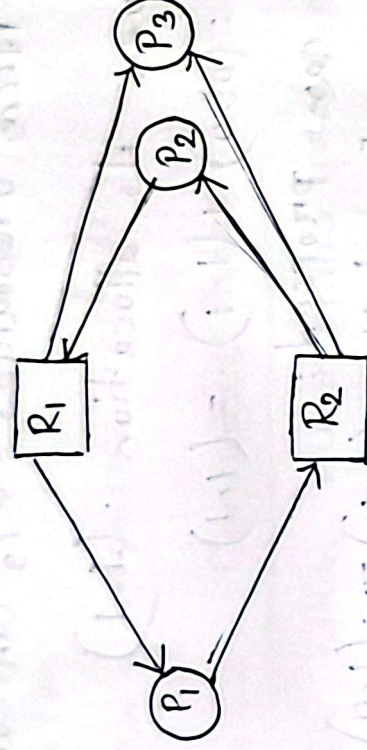
Disadvantages:

- Dispatch latency
- Dependent on scheduler.

Q) 3)

Allocation	Request
P1: R1=1 R2=0	P1: R1=0 P2=0
P2: R1=0 R2=1	P2: R1=1 R2=0
P3: R1=1 R2=1	P3: R1=0 R2=0

Draw RAG and determine if the system is in a deadlock state.



The RAG consists of:

- Process nodes: circles (P_1, P_2, P_3)
- Resource nodes: squares (R_1, R_2)
- Total $R_1 = 1(P_1) + 0(P_2) + 1(P_3) = 2$ units

- Total $R_2 = 0(P_1) + 1(P_2) + 1(P_3) = 2$ units
- R_1 available = total - allocated = $2 - 2 = 0$
- $2 - 2 = 0$

Deadlock detection using safety algorithm:

If any process can complete using the initial available resources (0,0):

- $P_1(0,1) \rightarrow$ needs 1 unit of $R_2 \rightarrow$ can't proceed
- ~~$P_2(1,0)$~~ ^{P_2} $P_2(1,0) \rightarrow$ needs 1 unit of $R_2 \rightarrow$ can't proceed
- $P_2(0,1) \rightarrow$ needs 1 unit of $R_1 \rightarrow$ can't proceed
- $P_3(1,1) \rightarrow$ Needs 0 resources $\rightarrow P_3$ can complete

Then, P_3 releases its allocation: (1,1)

$$\text{New available} = (0,0) + (1,1) = (1,1)$$

Now P_1 or P_2 can proceed:

- P_1 can proceed [Available (1,1) $\neq$ (1,0) $\neq$ (2,1)]
- P_2 $\checkmark$ [Available (2,1) + (0,1) $\neq$ (2,2)]

Since a safe execution exists: $P_3 \rightarrow P_1 \rightarrow P_2$

the system is not in a deadlock state.

b) Difference between deadlock prevention, deadlock avoidance and deadlock detection. Provide example of techniques used for each approach.

→ Previous.

Final/18/20:

Classical IPC Problems

Classical IPC problems are standard synchronization problems in OS, used to understand how processes can safely share resources and communicate without causing issues like deadlock, race conditions or starvation.

The main problems include:

(I) Producer-Consumer Problem: It describes the situation when one or more producers generate data items and place them into a shared buffer, while one or more consumers removes data items from that buffer. The buffer has a limited capacity. So, synchronization is required to ensure that.

- A producer doesn't add data when the buffer is full
- A consumer doesn't remove data when the buffer is empty.

Problem statement & solution: (using semaphore)

A fixed-size buffer shared between a producer and a consumer.

- The producer generates an item and places it in the buffer
- The consumer removes an item from the buffer.

The buffer is the critical section. At any moment,

- A producer can't place an item if the buffer is full.

- A consumer can't remove an item if the buffer is empty.

Solution using three semaphores:

(i) Mutex: Ensures mutual exclusion when accessing the buffer

(ii) full: counts the no. of filled slots in the buffer

(iii) empty: counts the no. of empty slots in the buffer.

Semaphore initialization

$\text{mutex} = 1$; // binary semaphore for mutual exclusion
 $\text{full} = 0$; // initially no filled slots
 $\text{empty} = n$; // buffer size

Producer

```
do { // produce an item
    wait(empty); // check for empty slot
    wait(mutex); // Enter critical section
```

// place item in buffer

```
    signal(mutex); // Exit critical section
    signal(full); // Increase no. of filled slots
} while (true);
```

Consumer

```
do {
    wait(full) // check for filled slot
    wait(mutex) // Enter critical section
    // Remove item from buffer
```

```
    signal(mutex) // Exit critical section
```

```
    signal(empty) // Increase number of empty slots
```



```
} while (true);
```

- Empty ensures that producers overfill the buffer
- Full ensures that ^{consumers} ~~producers~~ don't consume from an empty buffer
- Mutex ensures mutual exclusion, so only one process accesses the buffer at a time

(II) Readers-Writers Problem: The reader-writer problem

is a classical synchronization problem in Operating Systems and concurrent programming. It deals with a situation where a shared resource is accessed by multiple threads or process.

Readers: Multiple readers can read the shared data simultaneously without causing inconsistency.

Writers: Only one writer can access the data at a time and no readers are allowed.

while writing.

▣ Solution using Semaphore:

Synchronization variables.

writer: Ensures mutual exclusion while updating

readcnt.

wrt: Control access to the shared data (Used by both readers and writers)

readcnt: No. of readers currently in the critical section.

reader process:

do {

wait(mutex); // Lock before updating readcnt

readcnt ++;

if (readcnt == 1)

wait(wrt);

// First readers blocks writers

signal(mutex);

// Allow other readers in

// Critical section (reading)

wait(mutex);

readcnt --;

if (readcnt == 0)


```
signal(wrt); // Last reader allows writers  
signal(muter); // Unlock  
while (true);
```

Explanation:

→ When a reader enters, it locks mutem for update reader.
→ If it's the first reader, it locks wrt so
writers are blocked.

→ Multiple readers can now read the data simultaneously.
- only.

→ When a reader exists, it decrements readcnt.

→ If it's the last reader, it unlocks wrt & so
writers can proceed.

writers process:

```
do {  
    wait(wrt); // Lock resource  
    // Critical section  
    signal(wrt); // Release resource  
} while (true);
```


Explanation:

- Writer locks the resource to get exclusive access.
- Writer performs writing
- Writer releases the resource after finishing.

(iii) Dining Philosopher Problem: The DPP is a classical synchronization problem introduced by Edsger Dijkstra in 1965. It illustrates the challenges of resource sharing in concurrent programming, such as deadlock, starvation and mutual exclusion.

Problem statement:

- K Philosophers sit around a table.
- Each n alternates between thinking and eating.
- One chopstick between each philosopher.
- A philosopher must pick up two chopsticks to eat.
- Only one philosopher can use a chopstick at a time.

Solution using Semaphore:

Pseudo Pseudocode:

```
semaphore chopstick[5] = {1, 1, 1, 1, 1}
```

```
philosopher(i);
```

```
while (true) {
```

```
    think();
```

```
    wait(chopstick[i]); // pick left chopstick
```

```
    wait(chopstick[(i+1)%5]); // pick right chopstick
```

```
    eat();
```

```
    signal(chopstick[i]); // put left
```

```
    signal(chopstick[(i+1)%5]); // put right
```

Explanation:

→ Each chopstick is a binary semaphore initialized to 1.

→ philosophers spend time thinking.

→ Tries to pick the left; if it's not free, then wait.

→ u u u right

→ signal() puts down both chopsticks, making them available for neighbour.

Issue in this solution:

1. Deadlock: If every philosopher picks up the left, no one can pick up the right one circular wait.

2. Starvation: Some philosophers may never get a chance to eat.

3. Concurrency Control: Must ensure no two adjacent philosophers eat simultaneously.

Solution:

→ 1. Even-numbered philosophers pick up their right chopstick first then left. and Odd-numbered philosophers pick up their left chopstick first, then right.

→ 2. Allow only $N-1$ philosophers to eat at the same time. If n th philosopher wants to eat, it will wait until $(n-1)$ th philosopher finishes eating.

(IV) Sleeping Barber Problem: The SBP is a classic synchronization problem that demonstrates how to manage concurrent process. It models a barbershop scenario with a barber chair and a waiting room, where synchronization mechanisms like semaphores are used to manage access.

problem statement:

- One barber, one barber chair and N waiting chairs.
- If no customer, the barber sleeps in the chair.
- A customer who arrives:

- (I) Wakes the barber if he sleeping.
- (II) Waits if there are free chairs.
- (III) Leaves if no chairs are available.

solution using semaphore:

- (I) customer counts waiting customers.
- (II) Barber signals when the barber is ready.

iii) Mutex ensures mutual exclusion when accessing chairs

Pseudocode:

semaphore Customers = 0;

semaphore Barber = 0;

Mutex Seats = 1;

int FreeSeats = N;

Barber {
 while (true) {

 down(customers); // waits for customers

 down(seats);

 freeSeats++; // one waiting chair

 up(Barber); // Barber ready
 becomes free

 up(seats);

 // cut hair

 }

Waiter { (current waiter) down; // waiter ready

Customer {

 while (true) {

 down(seats);

 if (FreeSeats > 0) {

 down((FreeSeats));


```

freeSeats -- : //take a seat
up(customers) : //notify barber
up(seats) :
down(Barber) : //wait for barber
//get haircut
}else{
up(seats) : //leave if no seat
}
}
}

```

Explanation:

→ Customers counts waiting customers, Barber signals barbers availability, seats (mutex) protects chair count, and freeSeats = N tracks empty chairs.

→ Barber loop: waits (down(customers)) until a customer arrives. Locks seats, frees one chair (freeSeats ++), signals the barber is ready (up(barber)), unlocks seats and cuts hair.

→ Customer loop: Locks seats. If a chair is free ($\text{free seats} > 0$), takes it ($\text{free seat}--$) - notifies $\text{barben}(\text{up}(\text{customers}))$ - unlocks seats, then waits for $\text{barben}(\text{down}(\text{Barben}))$ to get haircut. If no seat is free unlocks seats and leaves.

□ Compare and contrast classical synchronization problem such as the producer consumer, dining philosophers and reader-writer problem.

→ contrast part error

□ Comparison of classical synchronization problem:

Feature	Producer-Consumer Problem	Dining Philosophers Problem	Reader-Writer Problem
1. Scenario	Two types of processes sharing a bounded buffer.	Multiple processes competing for a limited number of shared independent resources.	Two types of processes sharing a single data item.
2. Processes involved	Producers (create data) and consumers (use data)	N philosophers (processes)	readers (only read data) and writers (read and write data)

3. Shared resource	Buffer for holding items.	Chopsticks or forks (resources) for eating.	Data object (e.g. file, variable, database)
4. Synchronization goal	Ensure the buffer doesn't overflow or underflow. Mutual exclusion for buffer access.	Prevent deadlock and ensure fair access to resources so all philosophers can eat.	Allows multiple readers simultaneously but enforces mutual exclusion for writers.
5. Potential problem	Lost updates, busy waiting, data inconsistency.	Deadlock and starvation	Starvation of writers or readers.
6. Solution	Semaphores (e.g., empty, full, mutex)	Semaphores and a state tracking mechanism.	Semaphores and counting variables (e.g., read count for active readers).

Final 19-20: 5(a) Suppose that a system is in an unsafe state. Show that it is possible for the

processes to complete their execution without entering a deadlock.

(atob)

-> Unsafe state: An unsafe state in resource allocation, as defined by the Banker's Algorithm, It signifies a system condition where deadlock is possible but not guaranteed.

A system enters an unsafe state when the OS can't guarantee that all processes will be able to complete their execution successfully even if they were granted all their remaining necessary resources.

How processes complete their execution without entering a deadlock?

The main reason why an unsafe state might not immediately lead to a deadlock is that processes might not ^{request} ~~declared~~ their maximum declared resources.

for example, P_1 and P_2 are two processes and a

single resource type R with 10 total units.

Process	Maximum Need	Allocation	Remaining need
P ₁	9	4	5
P ₂	7	2	5

Available resource unit = $10 - (4 + 2)$
 $= 4$ units.

Step-1: Prove the state is unsafe

→ To complete P₁ $(9 - 4) = 5$ units needs. But

currently available is 4. → can't satisfy P₁.

→ To complete P₂ $(7 - 2) = 5$ units need. But

available 4 → can't satisfy P₂.

Since, no processes can be satisfied with the currently available resources no safe sequence exists, and the system is in an unsafe state.

Step-2: Execution without deadlock.

Note, suppose the processes don't request

their maximum need, but instead request less:
→ P₂ finishes its current phase of work and releases the 2 units it holds.

Now, available = $4 + 2 = 6$

→ The available resources (6 units) are sufficient to satisfy P₁'s remaining need (5 units). P₁ gets the resources, completes and releases all 9 units.

→ Since P₂ already completed its and released its resources.

In this scenario, the processes started in an unsafe state but completed their execution without any process being permanently blocked.

b)

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P ₀	0	0	1	2	0	0	1	2	1	5	2	0
P ₁	1	0	0	0	1	7	5	0				
P ₂	1	3	5	4	2	3	5	6				
P ₃	0	6	3	2	0	6	5	2				
P ₄	0	0	1	4	0	6	5	6				

(I) What is the content of the matrix need:

→

Process	Need			
	A	B	C	D
P ₀	0	0	0	0
P ₁	0	7	5	0
P ₂	1	0	0	2
P ₃	0	0	2	0
P ₄	0	6	4	2

(II)

Process	Allocation				Max				Need				Available			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P ₀	0	0	1	2	0	0	1	2	0	0	0	0	1	5	2	0
P ₁	1	0	0	0	1	7	5	0	0	7	5	0	1	5	3	0
P ₂	1	3	5	4	2	3	5	6	1	0	0	2	2	2	8	6
P ₃	0	6	3	2	0	6	5	2	0	0	2	0	3	1	13	10
P ₄	0	0	1	4	0	6	5	6	0	6	4	2	3	14	11	8

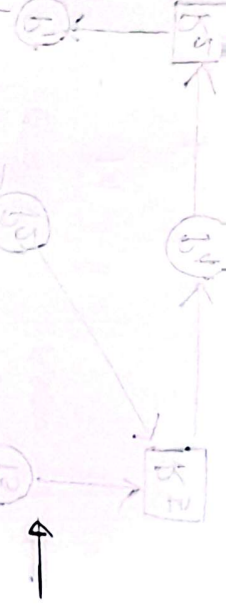
~~P₀ → P₂ → P₃ → P₁ → P₄~~

P₀ → P₂ → P₁ → P₃ → P₄

Available

A	B	C	D
1	5	2	0
1	5	3	2
2	8	8	6

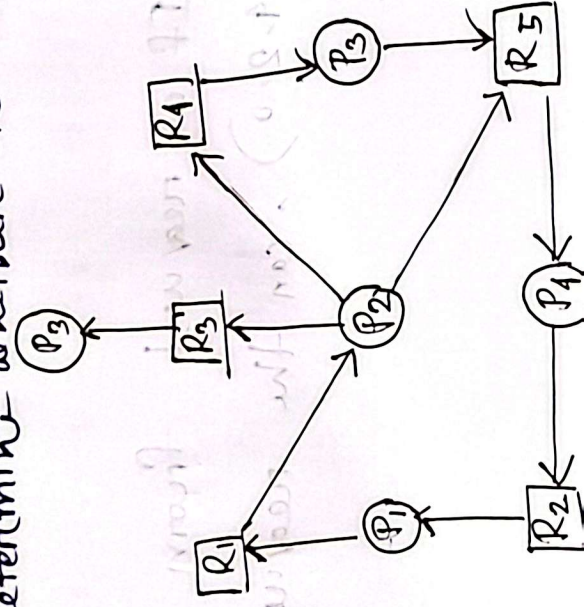
(III) If a request from process P₁ arrives for (0, 4, 2, 0), can the request be granted immediately?



Process	Allocation				Max				Need				Available			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P ₀	0	0	1	2	0	0	1	2	0	0	0	0	1	1	0	0
P ₁	1	4	2	0	1	7	5	0	0	3	3	2	1	1	1	2
P ₂	1	3	5	4	2	3	5	6	1	0	0	2	2	4	6	6
P ₃	0	6	3	2	0	6	5	2	0	0	2	0	3	8	8	6
P ₄	0	0	1	4	0	6	5	6	0	6	4	2	3	14	11	8
													3 14 12 12			

$P_0 \rightarrow P_2 \rightarrow P_1 \rightarrow P_3 \rightarrow P_4$

- Q) List two deadlock detection mechanisms. Construct the wait for graph that corresponds to the following RAG and determine whether or not there is a deadlock.



Deadlock detection mechanisms:

Deadlock detection mechanisms are used in systems that don't prevent or avoid deadlocks. Instead, they allow the system to enter a deadlock state, and then periodically checks for its existence.

(I) Wait-for Graph Algo (for single instance of each resource)

This is a simple algorithm used when each resource type has only one instance.

→ The system maintains a wait-for graph derived directly from the RAG.

→ Deadlock exists if and only if the wait-for graph contains a cycle.

(II) RAG:

RAG is a visual way to understand how resources are assigned in an OS.

→ It uses vectors and matrices (Allocation - request and available).

state that corresponds to a deadlock.

→ Deadlock exists if and only if the algorithm fails to find a sequence where all currently requesting processes can complete.

□ Construction of wait-for graph and determining deadlock:

The wait-for graph is derived from the RAG by removing the resource nodes and connecting processes nodes directly. An edge exists from P_i to P_j ($P_i \rightarrow P_j$).

Here, P_i is waiting for a resource that is currently held by P_j .

Step 1: Analyzing the RAG:

Process

Waiting for

P_1

Held by R_1

P_2

R_4, R_5

Held by

R_3 is held by P_2

R_4 is held by P_3 and

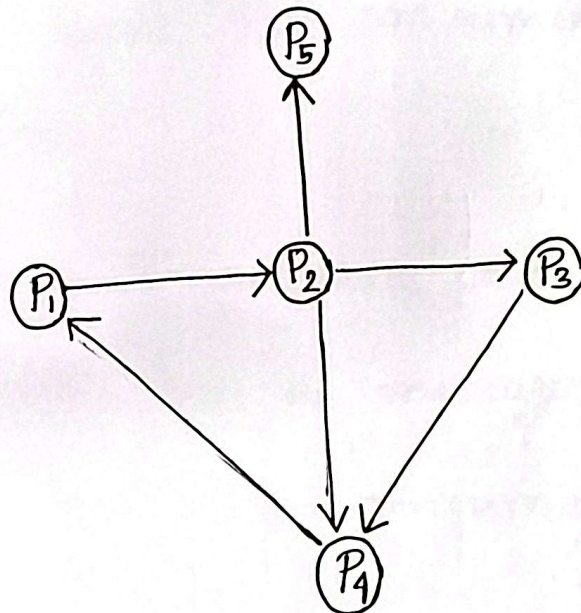
R_5 is held by P_4

P_3	R_5 is held by P_4
P_4	R_2 is held by P_1
P_5	R_3 is held by P_5

nearest for none

step-2: Constructing the wait-for-graph:

$P_1 \rightarrow P_2$
 $P_2 \rightarrow P_5$
 $P_2 \rightarrow P_3$
 $P_2 \rightarrow P_4$
 $P_3 \rightarrow P_4$
 $P_4 \rightarrow P_1$



wait-for graph

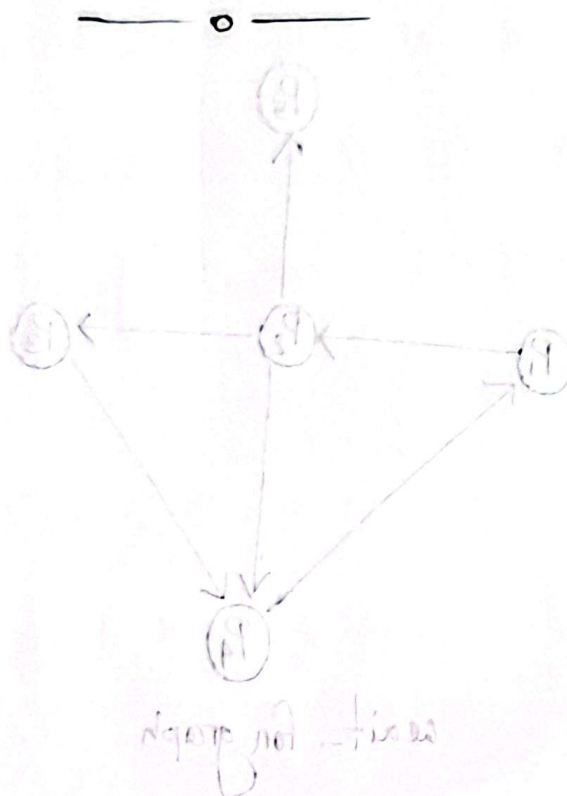
Step-3 Determine deadlock

A deadlock exists if the wait-for graph contains a cycle (A path that returns to the starting node).

Upon inspecting the edges, we find a cycle;

$$P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_1$$

Since, the wait for graph contains a cycle, a deadlock exists in the system.



4) b) Regarding the producer-consumer problem, explain the following term:

(i) Producer process: A producer process is responsible for generating data, items, or messages and inserting them into a shared storage area (the buffer).

- Primary job is to create work or data packages.
- The producer must wait if the shared buffer is full, as there is no space to store new items.
- It should signal the consumer once it has placed an item.

(ii) Consumer Process: A consumer process is responsible for retrieving data, items or messages from the shared buffer and then consuming or processing them.

- Its primary job is to take the produced work and use it.