OS MAY 2024 Answers

Q1

a) What are the various objectives and functions of Operating Systems?

The primary **objectives of an Operating System (OS)** are to efficiently manage hardware resources, provide an environment for executing applications, and ensure that users can interact with the computer in a controlled and convenient manner.

Objectives of an Operating System

1. **Resource Management:** Ensures efficient allocation and use of system resources like CPU, memory, storage, and input/output devices.
2. **Convenience:** Provides a user-friendly interface to make interaction with hardware easier for users and developers.
3. **Reliability:** Ensures the system operates without failures and maintains data integrity.
4. **Efficiency:** Optimizes system performance by minimizing resource wastage and processing time.
5. **Security:** Protects the system and user data from unauthorized access and threats.

Functions of an Operating System

1. **Process Management:**
   - Handles process creation, scheduling, and termination.
   - Manages synchronization and communication between processes (IPC).

2. **Memory Management:**
   - Allocates and deallocates memory to programs.
   - Uses techniques like paging and segmentation to optimize memory usage.

3. **File System Management:**
   - Manages files and directories for storing and retrieving data.
   - Provides functionalities like file creation, deletion, and access permissions.

4. **Security and Protection:**
   - Restricts unauthorized access to resources.
   - Implements mechanisms like authentication and encryption.

b) Differentiate between process and threads.

Aspect	Process	Thread
Definition	Independent program in execution with its own memory space.	Smallest unit of execution within a process.
Memory	Separate memory spaces (address spaces).	Shared memory space within the same process.
Creation	Relatively expensive, involves allocating memory and resources.	Less expensive, only requires resources for stack and registers.
Resource Sharing	Do not share resources directly; use IPC mechanisms.	Directly share resources like memory and file handles.
Isolation	Isolated from each other; a crash in one doesn't affect others.	Not isolated; a crash in one can affect the entire process.
Context Switching	More expensive, involves switching the entire memory space.	Less expensive, shared memory space.
Communication	Requires IPC mechanisms (e.g., pipes, message queues).	Simpler and faster, as threads can directly access shared resources.
Examples	Running applications like web browsers, text editors.	Multiple tabs in a web browser, background tasks.

c) Explain Race condition with example.

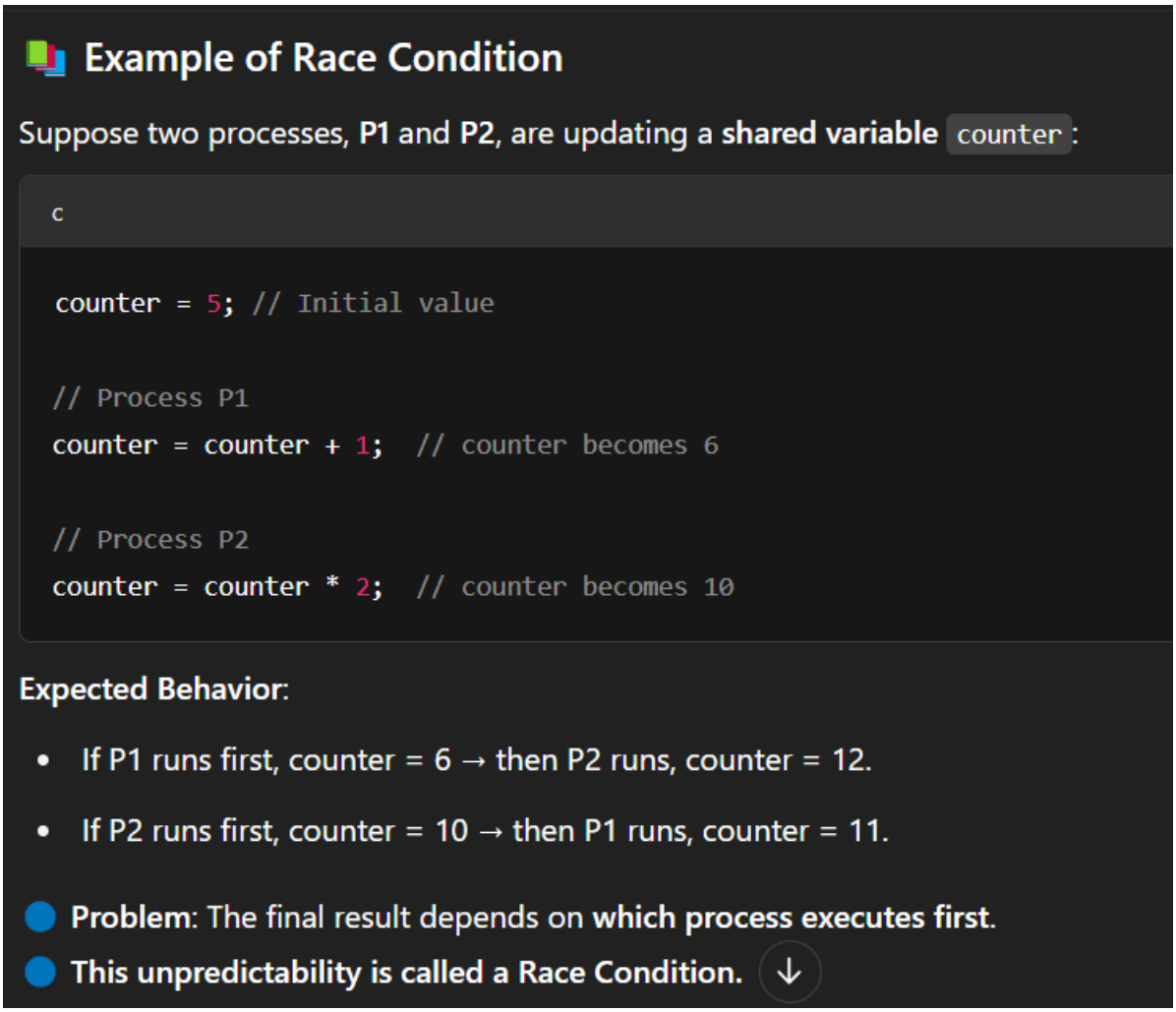
A **Race Condition** occurs when **two or more processes** access shared data **at the same time**, and the **final outcome** depends on the **order** in which the processes are scheduled. It leads to **inconsistent or incorrect results** if not handled properly.

Why Race Condition Happens?

- **Processes share the same data or resource** (like a variable, file, or memory).
- **No proper synchronization** between processes.
- **Order of execution is unpredictable**.

Race conditions can be prevented using synchronization mechanisms like:

1. **Mutex (Mutual Exclusion):** Allows only one process to access the shared resource at a time.
2. **Semaphores:** Used to signal and control access to shared resources.
3. **Critical Sections:** Code segments where shared resources are accessed are protected, ensuring only one process executes them at a time.



d) What is Demand Paging? What are its advantages?

Demand paging is a memory management technique used in modern operating systems where pages of data (from a process) are only loaded into the main memory (RAM) when they are required during program execution. It helps to optimize memory usage by avoiding the loading of unnecessary pages.

How Demand Paging Works

1. **Page Table**:
   - Each process has a page table that keeps track of the mapping between the logical (virtual) addresses and physical addresses.
   - The page table entries include a *Present/Absent* bit, indicating whether the page is in physical memory.

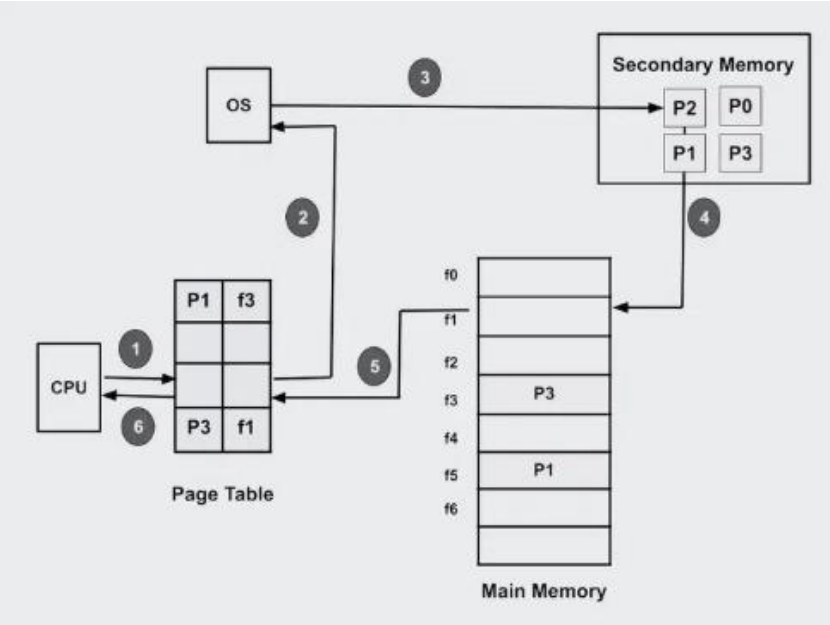
2. **Page Fault**:
   - When a process tries to access a page that is not currently in memory (the Present bit is "absent"), the system raises an interrupt called a **Page Fault**.
   - The operating system then loads the required page from secondary storage (e.g., a hard disk or SSD) into physical memory.

3. **Execution Resumes**:
   - Once the page is loaded into memory, the process resumes execution at the point where it was interrupted.

Advantages of Demand Paging

1. **Efficient Memory Usage**:
   - Only the necessary pages are loaded into memory, which reduces memory wastage.

2. **Supports Large Programs**:
   - Programs larger than the available physical memory can execute because pages are loaded on demand.



Example:
Assume a program has 10 pages, but only pages 1, 3, and 5 are needed at the beginning. With demand paging, only these pages are loaded, saving memory and improving performance.

e) What are features of Mobile and Real Time Operating Systems?

Mobile Operating Systems (e.g., Android, iOS):

- **User Interface (UI):** Optimized for touchscreens.
- **Multitasking:** Handles multiple apps running concurrently, with focus on efficient memory management.
- **Connectivity:** Seamless integration with cellular, Wi-Fi, Bluetooth, GPS, etc.
- **Power Management:** Designed to conserve battery life with features like app sleep modes.
- **Security:** Built-in encryption, app sandboxing, and secure boot processes.
- **App Ecosystem:** Supports a large range of apps and offers stores for distribution (Google Play, App Store).

Real-Time Operating Systems (RTOS):

- **Task Scheduling:** Prioritizes tasks based on urgency (e.g., priority scheduling or rate-monotonic).
- **Minimal Latency:** Ensures minimal delay in processing real-time tasks.
- **Resource Management:** Provides strict control over memory, CPU, and I/O resources.
- **Concurrency:** Supports multitasking with high precision for time-sensitive operations.

Q2

a) Give the explanation of necessary conditions for deadlock. Explain how a resource allocation graph determines a deadlock.

Necessary Conditions for Deadlock
For a deadlock to occur, the following **four necessary conditions** (known as **Coffman Conditions**) must hold simultaneously:

1. **Mutual Exclusion**:
    - At least one resource must be held in a non-shareable mode (i.e., only one process can use the resource at a time).
2. **Hold and Wait**:
    - A process holding at least one resource is waiting to acquire additional resources that are currently held by other processes.
3. **No Preemption**:
    - Resources cannot be forcibly taken away from a process. A process must voluntarily release its resources.
4. **Circular Wait**:
    - There exists a circular chain of processes, where each process in the chain is waiting for a resource held by the next process in the chain.

Example
Imagine two processes, P1 and P2, and two resources, R1 and R2:

- **P1** holds **R1** and waits for **R2** (held by **P2**).
- **P2** holds **R2** and waits for **R1** (held by **P1**). This creates a circular wait, leading to deadlock.

A **Resource Allocation Graph (RAG)** is a graphical representation used in operating systems to model the allocation of resources to processes and to detect the presence of a **deadlock**.

Components of a Resource Allocation Graph

1. **Processes (P):** Represented by circles (e.g., P1, P2).
2. **Resources (R):** Represented by rectangles (e.g., R1, R2). Each resource has multiple instances depicted as small dots inside the rectangle.
3. **Edges:**

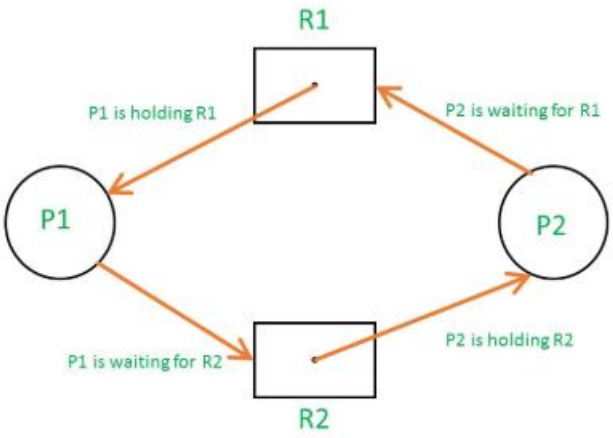
- **Request Edge:** A directed edge from a process to a resource (P → R). This indicates that the process is waiting for the resource.
- **Assignment Edge:** A directed edge from a resource to a process (R → P). This indicates that the resource is allocated to the process.

Detecting Deadlock

A **deadlock** occurs when processes form a cycle in the RAG, and all processes in the cycle are waiting for resources held by other processes in the cycle.

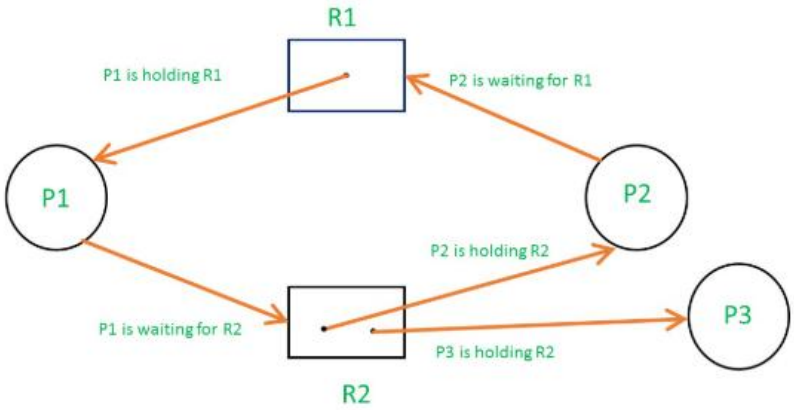
1. **Cycle in the Graph:**

- **Single Instance per Resource:** A cycle implies a deadlock. (Example below)

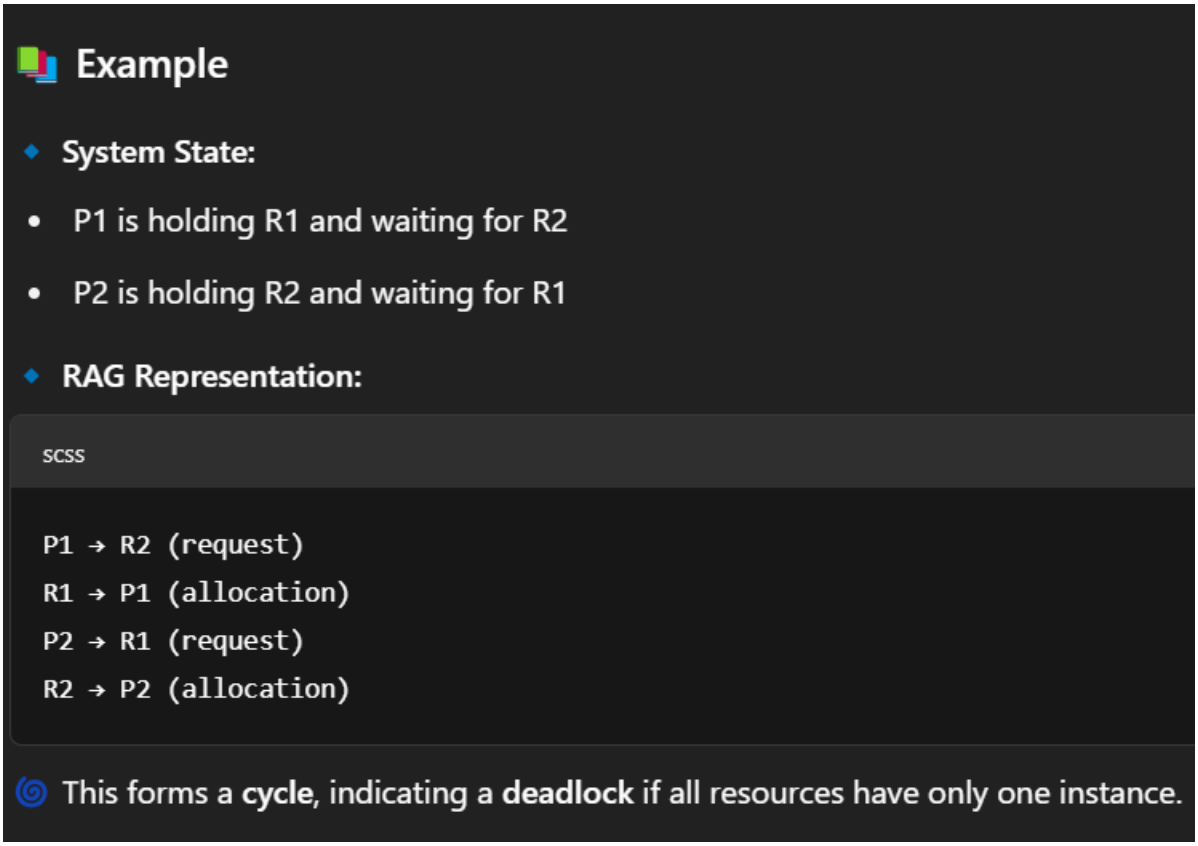


SINGLE INSTANCE RESOURCE TYPE WITH DEADLOCK

- **Multiple Instances per Resource:** A cycle *may* or *may not* indicate a deadlock. Further checks are needed to determine if the cycle can be broken.



MULTI INSTANCES WITHOUT DEADLOCK



Resolving Deadlock

To resolve a deadlock, the operating system can:

1. **Terminate a process** in the cycle to break the deadlock.
2. **Preempt resources** from some processes and reallocate them.
3. Use **deadlock prevention** techniques, such as ensuring at least one of the four necessary conditions for deadlock (mutual exclusion, hold and wait, no preemption, circular wait) does not occur.

Q3

a) Explain RAID Level in Details

RAID (Redundant Array of Independent Disks)

RAID is a **data storage virtualization technology** that combines multiple physical disk drives into a single unit to:

- Improve **performance**
- Provide **data redundancy** (fault tolerance)

It is commonly used in servers, data centers, and enterprise systems.

RAID 0 – Striping
How it works:
Data is **split (striped)** across two or more disks. No redundancy.

- **Performance**: Excellent (parallel read/write)
- **Fault Tolerance**: None (if one disk fails, all data is lost)
- **Storage Efficiency**: 100% (all disk space usable)

Minimum Disks: 2
Use Case: Gaming, video editing (where speed is important, not data safety)

RAID 1 – Mirroring
How it works:
Each disk has an **exact copy** (mirror) of the data.

- **Performance**: Good for reading
- **Fault Tolerance**: High (can survive one disk failure)
- **Storage Efficiency**: 50% (half the space used for backup)

Minimum Disks: 2
Use Case: Banking, financial data storage, where data safety is critical

RAID 5 – Striping with Parity
How it works:
Data is striped across disks with **parity information** (used to recover data if one disk fails).

- **Performance**: Balanced (good read, moderate write)
- **Fault Tolerance**: Can survive **1 disk failure**

- **Storage Efficiency**: (N-1)/N (one disk used for parity)

Minimum Disks: 3
Use Case: File servers, web servers

RAID 6 – Striping with Double Parity
How it works:
Like RAID 5, but stores **two sets of parity** to survive **2 disk failures**.

- **Performance**: Good read, slower write
- **Fault Tolerance**: Can survive **2 disk failures**
- **Storage Efficiency**: (N-2)/N

Minimum Disks: 4
Use Case: Enterprise systems where data reliability is top priority

RAID 10 (1+0) – Mirroring + Striping
How it works:
Combines RAID 1 and RAID 0. Data is first mirrored, then striped.

- **Performance**: Excellent
- **Fault Tolerance**: High (can survive multiple failures depending on disk pair)
- **Storage Efficiency**: 50%

Minimum Disks: 4
Use Case: High-performance databases, critical applications

b) What is Internal fragmentation?

Internal Fragmentation happens when **fixed-size memory blocks** are allocated to processes, but the process **does not use the entire block**.
The **unused space inside** the allocated block is called **internal fragmentation**.

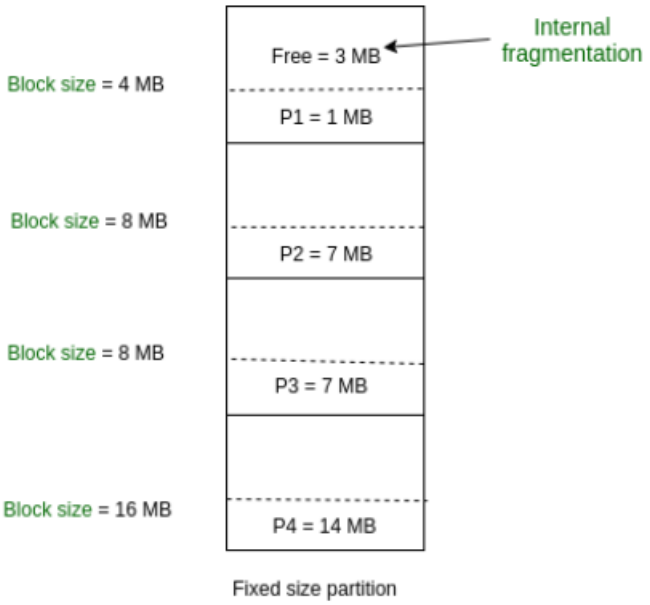
⚙️ **Why Does It Happen?**

- Memory is divided into **fixed-sized blocks**.
- If a process **needs less memory** than the block size, the **leftover space is wasted**.
- This **wasted space cannot be used** by other processes.

📚 **Example**

- Suppose **block size = 8 KB**.
- Process P1 needs only **6 KB**.

- The OS still gives it **one full 8 KB block**.
- Result: **2 KB** inside the block is **wasted** → This is **internal fragmentation**.



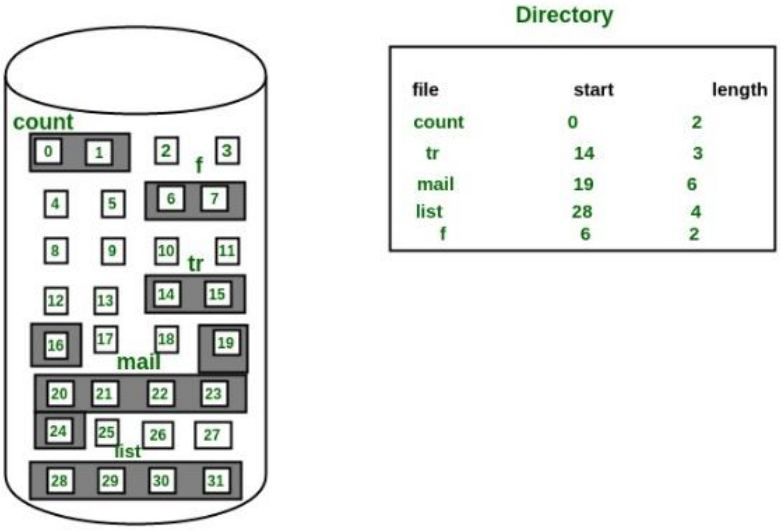
Q4

a) Explain file allocation methods in detail with proper diagram.

File allocation techniques refer to the methods used by operating systems to manage the storage of files on disk. Different allocation methods have their own advantages and drawbacks in terms of performance, fragmentation, and ease of implementation.

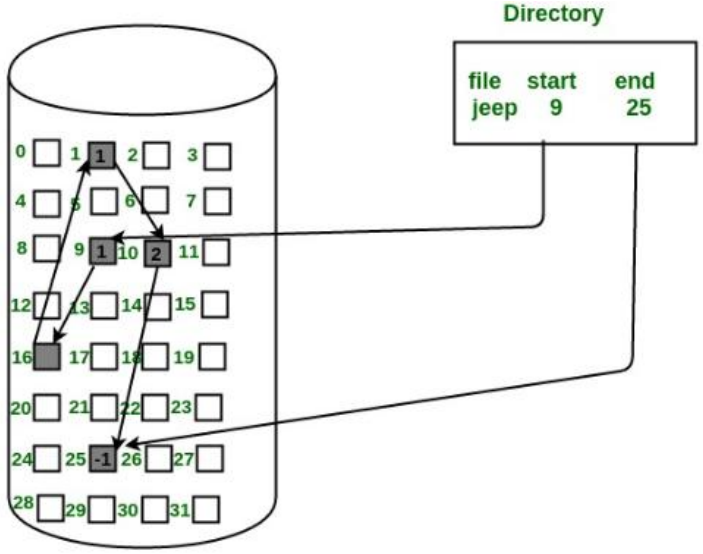
1. Contiguous Allocation

- **Description:** Files are stored in consecutive blocks on the disk.
- **Key Features:** The OS keeps track of the starting block and the length of the file.
- **Advantages:**
  - Simple to implement and manage.
  - Fast read/write access since the disk head doesn't need to move.
- **Drawbacks:**
  - Prone to external fragmentation, making it hard to find continuous free space.



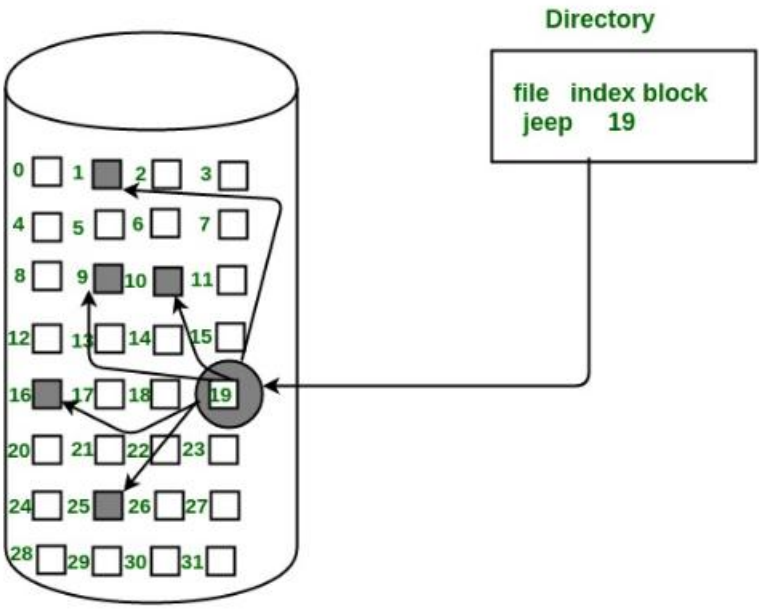
2. Linked Allocation

- **Description:** Files are stored in scattered blocks, and each block contains a pointer to the next block.
- **Key Features:** Only the starting block address is stored in the File Allocation Table (FAT).
- **Advantages:**
  - No external fragmentation.
- **Drawbacks:**
  - Reliability issues if pointers are damaged or corrupted.
  - Requires additional space for pointers.



3. Indexed Allocation

- **Description:** An index block is created, which contains pointers to all blocks used by the file.
- **Key Features:** The index block provides a direct mapping to all file blocks.
- **Advantages:**
  - Fast random access to file blocks.
  - No external fragmentation.
- **Drawbacks:**
  - Requires additional space for the index block.
  - More complex to implement compared to contiguous and linked allocation.



b) What is a thread? How multithreading is beneficial? Compare and contrast different multithreading models.

What is a Thread?

A **thread** is the smallest unit of a process that can be independently scheduled and executed by the CPU. While a process typically has its own memory space and resources, threads within a process share the same memory and resources. This allows threads to execute tasks simultaneously and communicate easily.

Answer to **How multithreading is beneficial** is in **[Q5 > b. OS May 2023 PDF]**

Multithreading Models
Different multithreading models define how user threads are mapped to kernel threads. Here are the primary models:

1. Many-to-One Model

- **Description:** Multiple user threads are mapped to a single kernel thread.
- **Features:**
  - Supported by systems like early versions of Solaris and Green Threads.
- **Advantages:**
  - Efficient and simple because kernel involvement is minimized.
- **Disadvantages:**
  - Only one thread can execute at a time.

2. One-to-One Model

- **Description:** Each user thread is mapped to a kernel thread.
- **Features:**
  - Supported by systems like Windows and Linux.
- **Advantages:**
  - Concurrent execution of threads on multiprocessor systems.
  - No limitation on the number of threads executing.
- **Disadvantages:**
  - Requires significant system resources.

3. Many-to-Many Model

- **Description:** Multiple user threads are mapped to multiple kernel threads.
- **Features:**
  - Supported by modern systems like Solaris 9 and beyond.
- **Advantages:**
  - Allows the OS to schedule threads efficiently.
  - Utilizes multiprocessor systems well.
- **Disadvantages:**
  - More complex to implement than the other models.

Comparison Table: Multithreading Models

Model	User Threads	Kernel Threads	Multiprocessor Support	Complexity	Overhead
Many-to-One	Many	One	Low	Simple	Minimal
One-to-One	One per thread	One per thread	High	Moderate	High
Many-to-Many	Many	Many	High	Complex	Moderate

Q5

a) Explain paging in detail. Describe how logical address is converted into physical address.

Paging is a memory management scheme that eliminates external fragmentation by dividing **physical and logical memory into fixed-size blocks**:

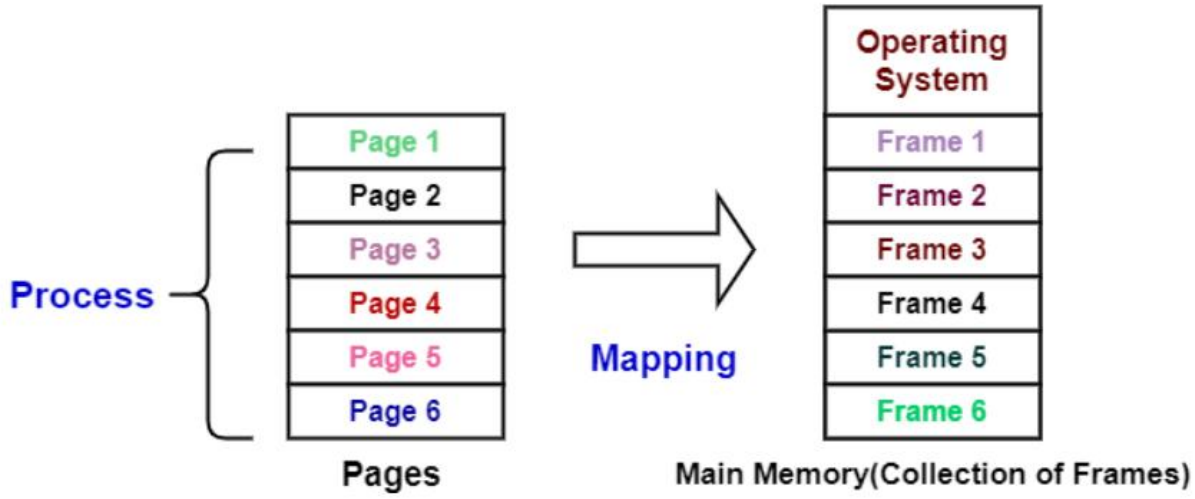
- Logical memory → **Pages**
- Physical memory → **Frames**

Key Points:

- Every logical page maps to a physical frame via a **page table**.
- **No need for contiguous allocation**, improves memory use.

Example:

- Suppose the frame size is 4 KB:
  - Process C (10 KB) is divided into 3 pages (4 KB, 4 KB, 2 KB).
  - These pages are allocated to any available frames, even if they are non-contiguous.



Advantages:

- Eliminates external fragmentation.
- Flexible allocation.

Disadvantages: Can cause internal fragmentation within frames.

Logical to Physical Address Translation

When a program accesses memory, it generates a **logical address**. This address must be translated into a **physical address** to locate data in RAM.

1. Structure of Logical Address

A logical address consists of two parts:

- **Page Number (p)**: Identifies the specific page in the process's logical memory.
- **Page Offset (d)**: Specifies the exact location within that page.

2. Translation Steps

1. **Extract the Page Number**:
    - Divide the logical address by the page size (using integer division) to get the page number **p**.

2. **Locate the Frame Number**:
    - Use the **page table** to map the page number **p** to the corresponding frame number **f** in physical memory.

3. **Compute the Physical Address**:
    - Combine the frame number **f** (from the page table) with the **offset d** (from the logical address) to calculate the physical address.
    - **Physical Address** = (Frame Number × Frame Size) + Offset.

Example:

- Page Size = 1 KB (1024 bytes).
- Logical Address = 2049.
- Page Table:
    - Page 0 → Frame 5.
    - Page 1 → Frame 7.

1. **Page Number (p)** = ⌊2049 ÷ 1024⌋ = 2.
2. **Offset (d)** = 2049 mod 1024 = 1.
3. **Frame Number (f)** = Frame mapped to Page 2 (from the page table) = **Frame 7**.
4. **Physical Address** = (7 × 1024) + 1 = **7169**.

Thus, the logical address 2049 is translated into the physical address 7169.

b) What is semaphore and its types? How the classic synchronization problem - Dining philosopher is solved using semaphores?

A **Semaphore** is a special **integer variable** used for **process synchronization** in an operating system. It helps **control access to shared resources** by multiple processes to **avoid conflicts** like data inconsistency or race conditions.

Semaphores are operated using two basic atomic (unbreakable) operations:

- **wait(S)**: Decreases the semaphore value by 1. If the result is negative, the process **waits** (blocks).
- **signal(S)**: Increases the semaphore value by 1. If other processes are waiting, one of them is **woken up**.

Semaphores help in solving problems like:

- **Mutual exclusion**
- **Deadlocks**
- **Process synchronization**

Types of Semaphores

1. **Binary Semaphore:**
    - Also known as a **mutex**, it can take values 0 or 1.
    - Used for mutual exclusion: A resource is either available (1) or occupied (0).
    - Example: Controlling access to a printer.

2. **Counting Semaphore:**
    - Can take non-negative integer values.
    - Tracks the count of available instances of a resource.
    - Example: Managing access to a pool of threads.

Dining Philosophers Problem using Semaphores

Problem Statement

- Five philosophers sit around a table.
- Each philosopher alternates between **thinking** and **eating**.
- A **fork** is placed between each pair of philosophers.
- To eat, a philosopher needs **both the left and right forks**.

Problems Faced

- **Deadlock**: If all philosophers pick up their left fork at the same time, all will wait for the right fork forever.
- **Starvation**: Some philosophers may never get a chance to eat.

Solution using Semaphores

Approach:

1. Use **one binary semaphore per fork** to ensure mutual exclusion.
2. Use an **additional semaphore (mutex)** to control access to shared data.

```c
semaphore mutex = 1; // For critical section
semaphore forks[5] = {1, 1, 1, 1, 1}; // One for each fork

void philosopher(int i) {
    while (true) {
        think();

        wait(mutex);              // Enter critical section
        wait(forks[i]);           // Pick up left fork
        wait(forks[(i+1)%5]);     // Pick up right fork
        signal(mutex);            // Exit critical section

        eat();

        signal(forks[i]);         // Put down left fork
        signal(forks[(i+1)%5]);   // Put down right fork
    }
}
```

Improved Deadlock-Free Solution:

To prevent deadlock, allow only **4 philosophers to try to pick forks at the same time** using a **counting semaphore** initialized to 4.

```c
semaphore mutex = 1;
semaphore forks[5] = {1, 1, 1, 1, 1};
semaphore room = 4;  // Only 4 philosophers allowed at the table

void philosopher(int i) {
    while (true) {
        think();

        wait(room);                // Limit entry
        wait(forks[i]);
        wait(forks[(i+1)%5]);

        eat();

        signal(forks[i]);
        signal(forks[(i+1)%5]);
        signal(room);              // Leave the room
    }
}
```

Q6

b) What is open-source operating system? What are the design issues of Mobile operating system and Real time operating system?

An **open-source operating system** is a type of OS whose source code is freely available to the public. Developers and users can view, modify, and distribute the source code according to their requirements. This promotes collaboration and innovation within the community.

Examples of Open-Source Operating Systems:

- **Linux (e.g., Ubuntu, Fedora):** One of the most popular open-source operating systems used for servers, desktops, and embedded systems.
- **FreeBSD:** A Unix-like OS often used for networking and internet applications.
- **Android:** Built on the Linux kernel, Android is an open-source mobile operating system.

Design Issues of Mobile Operating Systems

A **Mobile Operating System (Mobile OS)** is software designed specifically for mobile devices like smartphones and tablets. Key design issues include:

1. Resource Constraints:
- Limited memory, CPU power, and battery life compared to desktops.
- Efficient resource utilization is critical to maintaining performance.

2. Security:
- Protection from malware and unauthorized access is crucial.
- Issues like app sandboxing and permission management must be handled.

3. Connectivity:
- Seamless integration with mobile networks (e.g., 4G, Wi-Fi, Bluetooth) is essential.
- Switching between networks without interruptions must be smooth.

4. Power Management:
- Battery efficiency is a major focus.
- OS must optimize background app usage and screen brightness to conserve power.

Design Issues of Real-Time Operating Systems (RTOS)

A **Real-Time Operating System (RTOS)** is designed for applications that require immediate and deterministic response to events (e.g., robotics, aerospace systems). Key design issues include:

1. Determinism:
- The OS must guarantee task completion within a strict deadline.

2. Task Prioritization:
- Tasks are prioritized based on their urgency and importance.
- Preemption ensures higher-priority tasks execute without delay.

3. Resource Sharing:
- Synchronization mechanisms (e.g., semaphores, mutexes) are required to avoid race conditions and deadlocks.

4. Scalability:

- RTOS should efficiently handle systems with a large number of devices or tasks.

5. Minimal Latency:

- The OS must minimize context-switching and interrupt-handling delays.