SNS COLLEGE OF TECHNOLOGY

(An Autonomous Institution)
Approved by AICTE, New Delhi, Affiliated to Anna University, Chennai
Accredited by NAAC-UGC with 'A++' Grade (Cycle III) &
Accredited by NBA (B.E - CSE, EEE, ECE, Mech&B.Tech.IT)
COIMBATORE-641 035, TAMIL NADU

B.E/B.Tech- Internal Assessment – III

Academic Year 2024-2025 (Even Semester)

Fourth Semester

23CST202–Operating Systems

ANSWER KEY

1. Illustrate with an example how a linked allocation method works in file storage.
In the linked allocation method, each file is stored as a linked list of disk blocks located anywhere on the disk. Each block contains the actual data and a pointer to the next block in the sequence. For example, a file might be stored in blocks 4 → 10 → 22 → 17, with each block pointing to the next. This approach eliminates external fragmentation and is efficient for sequential access. However, it is inefficient for random access since blocks must be accessed sequentially.

2. Compare and contrast single-level and hierarchical directory structures.
A single-level directory structure places all files in one directory, making it simple to implement but problematic as file names must be unique. It becomes inefficient as the number of files grows, especially in multi-user systems. In contrast, a hierarchical directory uses a tree structure with directories and subdirectories, allowing better file organization. Users can group related files and avoid name conflicts. Hierarchical systems scale well and support complex pathnames.

3. Differentiate between RAID Level 0 and RAID Level 1 in terms of data redundancy and performance.
RAID Level 0 uses data striping, distributing data across multiple disks for increased performance, but it offers no data redundancy—if one disk fails, all data is lost. RAID Level 1 mirrors data on two or more disks, ensuring data redundancy by keeping exact copies. While RAID 0 provides better speed, RAID 1 provides high fault tolerance at the cost of storage efficiency. RAID 1 is preferred for reliability, while RAID 0 suits performance-focused systems. Storage cost is higher in RAID 1 due to duplication.

4. A 300 GB disk uses a file descriptor with 8 direct block addresses, 1 indirect, and 1 doubly indirect block address. With 128-byte disk blocks and 8-byte addresses, what is the maximum file size?
Given a 128-byte block size and 8-byte address size, each block can store 16 addresses. The file descriptor has 8 direct block pointers (8 × 128 = 1024 bytes), one indirect block (16 × 128 = 2048 bytes), and one doubly indirect block (16 × 16 × 128 = 32,768 bytes). Adding all together, the maximum file size supported is 1024 + 2048 + 32,768 = **35,840 bytes**. Despite the 300 GB disk, the file size is limited by the descriptor structure.

5. Describe how the NTFS file system manages disk space and file metadata.
NTFS uses a Master File Table (MFT) to store metadata and manage disk space efficiently. Each file or directory is represented by a record in the MFT, which includes attributes such as file name, size, timestamps, and permissions. NTFS uses clusters (groups of sectors) for disk allocation and maintains a bitmap to track free and used space. For small files, data can be stored directly within the MFT record, reducing fragmentation. NTFS also supports advanced features like journaling, encryption, and compression.

6 (a)Analyze the different file allocation methods (contiguous, linked, and indexed). Discuss their advantages, disadvantages, and the type of applications where each is most suitable

Efficient file allocation is a fundamental aspect of operating system design, particularly in the management of secondary storage such as hard disks and solid-state drives. The goal of file allocation is to ensure that data is stored and retrieved efficiently, reliably, and with minimal overhead. The method used to allocate space for files directly influences system performance, fragmentation, data integrity, and ease of file management. Three primary

methods of file allocation are commonly used: **Contiguous Allocation**, **Linked Allocation**, and **Indexed Allocation**. Each of these methods has its own strengths and weaknesses and is best suited to specific types of applications and usage scenarios.

1. Contiguous Allocation

Mechanism

Contiguous allocation is the simplest file allocation technique. When a file is created, the operating system allocates a sequence of consecutive blocks on the disk to store the file. The starting block number and the length of the file (in blocks) are stored in the file allocation table (FAT) or the directory structure. For instance, if a file needs five blocks and starts at block 10, it will occupy blocks 10 to 14.

Advantages

- **Fast Access**: Since blocks are located next to each other, this method provides excellent performance for both sequential and direct access.
- **Simple Implementation**: Its straightforward design makes it easy to implement and maintain.
- **Minimal Overhead**: Unlike other allocation methods, it doesn't require additional metadata (like pointers or index blocks) beyond the base and length.

Disadvantages

- **External Fragmentation**: Over time, as files are created and deleted, free space becomes fragmented. Large contiguous blocks may become rare, leading to allocation failures even when enough total space exists.
- **Difficult File Growth**: If a file needs to grow beyond its initially allocated space and the following blocks are occupied, it must be relocated entirely, which is inefficient.
- **Wasted Space**: Allocating more space than needed initially to accommodate growth can result in unused storage.

Suitable Applications

- **Multimedia Streaming**: Audio and video files benefit from fast sequential access.
- **Read-Only Media**: Systems where files are written once and never modified (e.g., CD-ROMs, firmware storage).
- **Static File Systems**: Embedded or appliance-based OSes where file sizes are predictable and don’t change frequently.

2. Linked Allocation

Mechanism

In linked allocation, each file is stored as a linked list of disk blocks. Each block contains data and a pointer to the next block. Only the starting block address is stored in the directory entry. Unlike contiguous allocation, blocks can be scattered throughout the disk.

For example, if a file occupies block 9, which points to block 16, which then points to block 25, and so on, the file can be read by following the chain of pointers from the starting block.

Advantages

- **No External Fragmentation**: Blocks can be located anywhere on the disk, so there is no need to find a large contiguous space.
- **Dynamic File Size Management**: Files can grow or shrink as needed without reallocation or movement.

- **Efficient Use of Disk Space**: The system can utilize all available disk blocks without worrying about block contiguity.

Disadvantages

- **Poor Random Access**: To read the nth block, the system must traverse all n-1 blocks sequentially, making random access operations slow.
- **Pointer Overhead**: Each block must store a pointer, reducing the usable space for actual data.
- **Reliability Risks**: If a pointer is corrupted, the rest of the file may become inaccessible. Additionally, recovering a damaged chain is difficult.
- **Increased Disk Seeks**: Since blocks are scattered, the read/write head may need to move frequently, increasing access time.

Suitable Applications

- **Log Files and Transaction Histories**: Where data is primarily appended and rarely accessed randomly.
- **Archival Systems**: Where read/write performance is secondary to efficient storage.
- **File Systems in Simple or Embedded OSes**: Where minimal metadata and small file sizes are the norm.

3. Indexed Allocation

Mechanism

Indexed allocation uses a special block, known as an index block, which contains pointers to all the blocks used by the file. The directory entry for the file stores the address of this index block. The file blocks themselves can be scattered anywhere on the disk, and the index keeps track of their order.

If a file has blocks at 2, 15, 26, and 33, the index block will contain these addresses. For large files, multi-level indexing (e.g., as in UNIX inodes) or indirect blocks may be used.

Advantages

- **Supports Both Sequential and Random Access**: Direct access to any block is possible via the index.
- **No External Fragmentation**: Like linked allocation, blocks can be anywhere.
- **Efficient for Small and Large Files**: Small files need only one index block, while larger ones can use multi-level indexing.

Disadvantages

- **Metadata Overhead**: Extra storage is needed for the index block(s).
- **Complex Implementation**: Managing multi-level indexes and dealing with file growth is more complicated.
- **Wasted Index Space for Small Files**: Small files may use only a few pointers in the index block, wasting space.

Suitable Applications

- **General-Purpose Operating Systems**: UNIX, Linux, and Windows use indexed allocation due to its versatility.
- **Databases and Applications Requiring Random Access**: Where efficient block-level access is critical.
- **Large File Support Systems**: Cloud storage backends, digital archives.

Comparison Table

Feature	Contiguous Allocation	Linked Allocation	Indexed Allocation
Access Speed	Very fast (sequential & direct)	Slow for random access	Fast (random & sequential)
Fragmentation	External	None	None
File Growth	Difficult	Easy	Easy
Metadata Overhead	Low	Moderate (per block)	High (index block)
Suitable For	Media, Read-only files	Logs, sequential data	OS, databases, large files

Hybrid Systems and Real-World Implementations

Many modern file systems use **hybrid approaches** to leverage the strengths of different allocation techniques. For example:

- **UNIX/Linux Inodes**: Use a combination of direct, single indirect, double indirect, and triple indirect blocks, blending indexed allocation with multi-level indexing. Small files use direct pointers, while large files scale using indirect ones.
- **FAT (File Allocation Table)**: A variation of linked allocation where block pointers are stored in a separate FAT structure instead of inside each data block. This reduces data block overhead but increases seek times due to frequent FAT access.
- **NTFS (New Technology File System)**: Uses a form of indexed allocation with Master File Table (MFT) entries that can store both metadata and direct pointers to data. It adapts based on file size—small files may be entirely stored within the MFT entry.

These implementations highlight the practical trade-offs between performance, space efficiency, and complexity in real-world systems.

6.(b) Evaluate the various directory structures (single-level, two-level, tree-structured, acyclic graph, and general graph). Discuss how they impact file sharing, access efficiency, and protection in a multi-user environment.

In an operating system, the directory structure plays a vital role in organizing files and managing access in a systematic manner. As systems scale and evolve to support multiple users, applications, and security requirements, directory structures must ensure efficient access, secure file management, and flexible file sharing. The directory not only provides a way to locate files but also determines how users interact with the file system.

Several directory structures are used in modern file systems, each with different capabilities in terms of scalability, protection, and usability. These include the **single-level directory**, **two-level directory**, **tree-structured directory**, **acyclic graph directory**, and **general graph directory**. Each structure offers distinct trade-offs, which directly affect file access efficiency, file sharing capabilities, and user-level protections, especially in multi-user environments.

1. Single-Level Directory

Structure Overview

In a single-level directory structure, all files are contained within the same directory. Every user and application accesses this common pool of files, and each file must have a unique name. This is the simplest form of directory management and is typically only suitable for systems with a very limited number of files or users.

Impact on Access Efficiency

Accessing files is relatively efficient because the directory is small and flat—searching requires minimal traversal. However, as the number of files increases, the single-level directory becomes inefficient. Linear search through

an increasingly large list slows down file lookup operations. Moreover, without subdirectories, organizing files logically becomes difficult, reducing usability.

Impact on File Sharing

File sharing is inherently straightforward in this structure because all users access the same directory. However, this creates potential for conflicts due to the lack of namespace separation. For example, if two users create files with the same name, the system cannot distinguish between them. This severely limits its applicability in multi-user systems.

Impact on Protection

Single-level directories offer minimal protection. Since there is no user-specific separation, enforcing permissions on a per-user basis is nearly impossible. All users have equal access to all files unless the operating system provides access control outside the directory system. In multi-user environments, this model is insecure and unsuitable.

Use Cases

- Simple, embedded systems.
- Early-generation operating systems.
- Educational or demo systems where user isolation is not critical.

2. Two-Level Directory

Structure Overview

The two-level directory structure introduces a user-level separation. The system has a master file directory (MFD) that contains a separate user file directory (UFD) for each user. Each UFD maintains the files created by the corresponding user. While users cannot interfere with each other’s file names, each file must still be uniquely named within a user's directory.

Impact on Access Efficiency

Access efficiency improves as each user searches within their own directory. Since the user directories are typically smaller, file lookup is faster compared to a single-level system. This model supports better organization and scaling by separating user files into distinct namespaces.

Impact on File Sharing

Two-level structures isolate users effectively but hinder file sharing. Since each user has an independent directory, explicit mechanisms must be introduced to allow shared access. This can be achieved by creating links or symbolic pointers, but such mechanisms are not inherent to the two-level structure.

Impact on Protection

This model provides improved protection compared to the single-level structure. User directories are isolated, which allows access control at the directory level. Operating systems can enforce permission checks when users attempt to access files outside their own directory, supporting basic multi-user security.

Use Cases

- Educational or small-scale multi-user systems.
- Systems requiring basic isolation but minimal file sharing.
- Early multi-user UNIX-like systems.

3. Tree-Structured Directory

Structure Overview

The tree-structured directory extends the two-level structure by allowing directories to contain subdirectories. This results in a hierarchical structure rooted at a base directory. Users and system administrators can create directories to organize files logically, grouping them by project, type, or access control requirements.

Impact on Access Efficiency

This structure significantly improves organization and scalability. Files are grouped hierarchically, enabling efficient search and navigation. Directory paths (e.g., /home/user/docs/report.txt) clearly specify file locations. Searching can be optimized using techniques like hash tables or balanced trees within each directory.

Impact on File Sharing

Tree-structured directories support limited file sharing. Although users can navigate and access files in other directories (with appropriate permissions), there is no built-in mechanism for multiple users to share a single file seamlessly. Links and symbolic references can be used for sharing, but care must be taken to ensure consistency and prevent cycles.

Impact on Protection

Tree structures support robust access control mechanisms. Permissions can be enforced at each directory and file level, allowing detailed control over who can read, write, or execute a file. This is especially important in a multi-user environment, where access control lists (ACLs) or user/group/others permission schemes (e.g., in UNIX) are implemented effectively.

Use Cases

- Modern desktop and server operating systems (Windows, macOS, Linux).
- Systems requiring strong file organization and moderate file sharing.
- Enterprise environments with multiple users and applications.

4. Acyclic Graph Directory

Structure Overview

The acyclic graph structure generalizes the tree structure by allowing directories and files to have multiple parent directories. This supports shared subdirectories or files among users. The system avoids cycles, ensuring that directory traversal does not result in infinite loops. Hard links and symbolic links are common mechanisms used here.

Impact on Access Efficiency

Access remains efficient due to the hierarchical nature, though the presence of shared files or directories may increase the complexity of managing paths and references. File access times can remain low if the system maintains efficient lookup tables and caches frequently accessed links.

Impact on File Sharing

File sharing is greatly enhanced. Multiple users can access a common file without duplicating it, ensuring consistency and reducing storage overhead. For example, a shared library can be linked in multiple users' directories. Users can reference shared data without copying it, which improves collaboration and resource usage.

Impact on Protection

Protection becomes more complex due to shared access. If a file is linked in multiple locations, modifying it through one path affects all users referencing that file. The system must implement strict permission and locking mechanisms to prevent unauthorized modifications or race conditions. Fine-grained access control lists are typically used.

Use Cases

- UNIX/Linux systems with symbolic links and shared libraries.
- Multi-user systems requiring efficient collaboration.
- Networked file systems and shared workspace environments.

5. General Graph Directory

Structure Overview

The general graph structure removes the restriction on cycles, allowing directories and files to be linked arbitrarily, including cyclic paths. This creates a complex web of relationships among files and directories. Such flexibility provides the greatest power and sharing capability but also introduces significant management challenges.

Impact on Access Efficiency

Access efficiency can degrade in general graph structures due to potential cycles and the need for traversal detection. Systems must implement mechanisms such as reference counting, garbage collection, or visited-node tracking to avoid infinite loops and ensure correct traversal during operations like searching or deletion.

Impact on File Sharing

General graphs offer maximum file sharing flexibility. Files and directories can be freely shared and organized in various ways. This supports advanced use cases such as versioning, backups, or software dependency trees. However, the complexity of managing links, references, and permissions grows with the structure's flexibility.

Impact on Protection

Protection is challenging in general graph structures. Since files can be accessed through many paths, ensuring consistent permission enforcement requires sophisticated access control and audit mechanisms. Deletion becomes non-trivial, as the system must determine whether other links to a file or directory exist.

Use Cases

- Complex software development environments (e.g., package managers with dependency graphs).
- Version control systems (e.g., Git, where commits form a directed graph).
- Operating systems with advanced symbolic linking and virtualization.

Comparison Table

Feature	**Single-Level**	**Two-Level**	**Tree-Structured**	**Acyclic Graph**	**General Graph**
User Separation	None	Basic	Full	Full	Full
File Sharing	Difficult	Limited	Moderate	Easy	Very Easy
Access Efficiency	Fast (few files)	Moderate	High	High	Moderate

Feature	Single-Level	Two-Level	Tree-Structured	Acyclic Graph	General Graph
Protection Support	Poor	Basic	Strong	Complex	Very Complex
Namespace Organization	Flat	Per-user	Hierarchical	Hierarchical + Links	Arbitrary
Cycles Allowed	No	No	No	No	Yes
Use in Modern Systems	Rare	Limited	Common	Very Common	Specialized

Impact in Multi-User Environments

File Sharing

In multi-user environments, file sharing is a critical requirement. Single and two-level directories provide minimal support. Tree-structured directories improve the situation by allowing users to access each other's directories with permissions. Acyclic and general graph structures offer the most flexibility, allowing shared libraries, collaborative files, and centralized resources to be accessed by many users without duplication.

Access Efficiency

Tree-structured and acyclic graph models strike a balance between efficient access and structural organization. The introduction of links allows frequently used files to be easily accessible from multiple paths. However, general graph structures may suffer from efficiency issues unless sophisticated caching and traversal strategies are implemented.

Protection and Security

Security becomes progressively more challenging as directory structures become more complex. Single-level directories cannot enforce per-user protection. Two-level systems support basic isolation. Tree structures support robust security policies through permission hierarchies. Acyclic and general graphs require advanced mechanisms like ACLs, role-based access control, and encryption to ensure secure file sharing and prevent unauthorized access.

7.(a) Critically evaluate various disk scheduling algorithms such as FCFS, SSTF, LOOK, and C-SCAN. Compare their performance in terms of seek time and system responsiveness using suitable examples.

Disk scheduling determines the order in which read/write requests to a disk are served, affecting performance metrics such as seek time, throughput, and system responsiveness. Modern disks have mechanical parts, so minimizing head movement (seek time) is essential. Several algorithms exist, each with strengths and trade-offs.

First-Come-First-Serve (FCFS) is the simplest scheduling algorithm. Requests are handled in the order they arrive. This approach is easy to implement and fair, ensuring no starvation. However, it often results in **high average seek time**, especially if requests are scattered across the disk. For instance, if the queue contains requests at cylinders 10, 70, 20, 90, 30 (starting at head position 50), FCFS will move to 10, then 70, then 20, causing unnecessary back-and-forth head movement. Despite its fairness, FCFS is inefficient and unsuitable for systems requiring high throughput.

Shortest Seek Time First (SSTF) selects the request closest to the current head position. This reduces seek time significantly by minimizing movement. Using the above example, SSTF would go from 50 to 30, then 20, 10, 70, and 90. This results in reduced total seek time. However, SSTF may lead to **starvation** of far-off requests if closer requests keep arriving. While SSTF improves performance over FCFS, it can cause delays for requests at the disk's ends. SSTF is suitable for moderate-load systems with balanced access needs.

LOOK is an improvement over SCAN (elevator algorithm). It scans in one direction, servicing requests until none remain in that direction, then reverses. Unlike SCAN, it doesn't go to the disk's end if there are no requests. LOOK reduces unnecessary movement and prevents starvation. For example, with head at 50 and requests at 10,

30, 70, and 90, LOOK moves from 50 → 70 → 90, then reverses to 30 → 10. This method balances efficiency and fairness and is suitable for general-purpose operating systems.

C-SCAN (Circular SCAN) treats the disk as a circular list. The head moves in one direction (e.g., from inner to outer tracks), servicing requests, then jumps back to the beginning and starts again. This provides **uniform wait times**, especially for newly arriving requests, avoiding the reversal delay in LOOK. It's predictable and avoids starvation, making it ideal for systems needing fairness, such as shared servers. The drawback is slightly more seek time due to the jump back, but the consistent performance offsets this.

To summarize, FCFS is fair but inefficient; SSTF is fast but may starve requests; LOOK offers a good balance between seek time and fairness; and C-SCAN provides uniform performance at the cost of extra travel. **C-SCAN or LOOK** are preferred in modern OSes for fairness and efficiency, while **SSTF** is used when performance is critical and starvation is manageable.

7(b) Explain the strategies used in disk management and swap-space management. Compare various disk scheduling algorithms and justify which algorithm would be most suitable for a real-time operating system. Additionally, design a memory management plan incorporating swap-space for a system running multiple large applications concurrently.

1. Disk Management Strategies

1.1 Disk Partitioning

Disk partitioning divides a physical disk into logical segments, each acting as a separate volume. Partitions help segregate the OS, user data, and swap space, enhancing security and manageability. Partitioning strategies include:

- **Primary and extended partitions**: Used in legacy BIOS systems.
- **GUID Partition Table (GPT)**: Supports larger disks and more partitions.
- **Logical volumes**: Created using Logical Volume Manager (LVM) for flexible resizing and snapshotting.

1.2 Disk Formatting

Formatting involves preparing a disk with a file system (e.g., NTFS, ext4). This process creates structures like boot blocks, superblocks, inodes, and free space maps. File systems organize files and manage metadata, access rights, and directory structures.

1.3 Free Space Management

Free space can be tracked using:

- **Bitmaps**: A sequence of bits where each bit represents a block's allocation status.
- **Linked lists**: Free blocks linked together, making traversal easier but slower.
- **Grouping**: Stores the addresses of free blocks in groups for faster allocation.

1.4 Disk Caching

Disk caching stores frequently accessed disk blocks in main memory, reducing access time. Techniques such as write-through and write-back control when data is written back to disk.

1.5 Disk Reliability and Fault Tolerance

Modern systems employ redundancy (e.g., RAID) and error detection (e.g., checksums, journaling) to prevent data loss. Disk scanning and predictive failure analysis (SMART) also enhance reliability.

2. Swap-Space Management Strategies

2.1 Purpose of Swap-Space

Swap-space, or virtual memory backing store, is used when physical RAM is insufficient to hold all active processes. It enables a process to run as if it has more memory than physically available, albeit with a performance trade-off.

2.2 Swap-Space Allocation Policies

Swap-space can be allocated using:

- **Preallocation**: Swap space is reserved for each process at start. This guarantees availability but wastes space if unused.
- **Demand allocation**: Space is allocated as needed. This is space-efficient but may lead to allocation failures under pressure.

2.3 Swap-Space Location

Swap-space may reside:

- **On a dedicated disk partition**: Faster due to lower overhead.
- **In a regular file**: More flexible but incurs file system overhead.
- **On SSDs or NVMe devices**: Offers faster swapping compared to HDDs.

2.4 Swapping Techniques

- **Full process swapping**: Entire processes are moved to and from disk.
- **Page-level swapping**: Only inactive memory pages are swapped. Modern systems use this with demand paging.

2.5 Swap Management Tools

Operating systems use tools like swapon, swapoff, and vmstat in Linux to manage and monitor swap usage. Swappiness parameters can be tuned to control how aggressively the kernel uses swap space.

3. Disk Scheduling Algorithms

Disk scheduling determines the order in which disk I/O requests are serviced. An efficient scheduler improves throughput, minimizes latency, and enhances system responsiveness.

3.1 FCFS (First-Come, First-Served)

- **Algorithm**: Requests are processed in the order they arrive.
- **Pros**: Simple, fair.
- **Cons**: Poor average seek time, long wait times if distant requests precede nearby ones.
- **Use case**: Suitable for small, non-critical systems.

Example: If requests are at cylinders 10, 40, 20, and 60, the disk head will traverse 10→40→20→60, which causes unnecessary movement.

3.2 SSTF (Shortest Seek Time First)

- **Algorithm**: Services the request closest to the current head position.
- **Pros**: Minimizes seek time.
- **Cons**: Can cause starvation of distant requests.
- **Use case**: General-purpose systems with moderate load.

Example: For head at cylinder 30 with requests at 20, 35, and 60, SSTF will go 30→35→20→60.

3.3 SCAN (Elevator Algorithm)

- **Algorithm**: Head moves in one direction, servicing all requests until the end, then reverses.
- **Pros**: Fair and better performance than FCFS.
- **Cons**: Edge requests may wait longer.
- **Use case**: Suitable for workloads with uniform request distribution.

3.4 C-SCAN (Circular SCAN)

- **Algorithm**: Like SCAN, but only services in one direction, jumping to the beginning at the end.
- **Pros**: More uniform wait times than SCAN.
- **Cons**: Slightly higher average seek than SSTF.
- **Use case**: Time-sharing systems.

3.5 LOOK and C-LOOK

- **LOOK**: Like SCAN, but reverses at the last request instead of disk end.
- **C-LOOK**: Like C-SCAN, but jumps to the lowest request.
- **Pros**: Optimized head movement.
- **Use case**: Systems needing balance between performance and fairness.

4. Disk Scheduling for Real-Time Operating Systems (RTOS)

Requirements for RTOS

Real-time systems prioritize **predictability** and **bounded latency** over average performance. They must guarantee worst-case execution times and meet deadlines for critical tasks.

Why FCFS or Deadline Scheduling is Preferred

- **FCFS**: Though inefficient in average seek time, it ensures predictability, which is vital for real-time applications.
- **Earliest Deadline First (EDF)**: In RTOS, EDF schedules disk requests based on deadline constraints, ensuring tasks meet their timing requirements.
- **SSTF, SCAN, C-SCAN**: These may cause starvation or unpredictable latencies, making them unsuitable.

---

5. Memory Management Plan with Swap-Space for Large Applications

In systems running multiple large applications—such as data analytics, simulation software, or integrated development environments—efficient memory management is essential to avoid resource exhaustion.

5.1 Objectives

- Maximize memory utilization.
- Minimize swapping overhead.
- Ensure responsiveness under load.
- Support concurrency across applications.

5.2 Components of the Plan

a. Paging with Demand Paging

- Divide memory into fixed-size pages and map them to frames in RAM.
- Load pages only when accessed, reducing memory footprint.
- Use page replacement algorithms (e.g., LRU, CLOCK) to decide which pages to evict.

b. Swap-Space Allocation Strategy

- Use a dedicated swap partition for speed and predictability.
- Dynamically allocate swap pages using a bitmap allocator.
- Monitor free swap space and trigger low-memory warnings when thresholds are crossed.

c. Working Set Model

- Maintain a "working set" (set of active pages) for each application.
- Prioritize keeping the working set in RAM to minimize page faults.
- Dynamically adjust the working set size based on usage patterns.

d. Priority-Based Memory Allocation

- Assign priorities to applications (e.g., foreground vs. background).
- Allocate more memory to high-priority apps, demoting background apps to swap if necessary.
- Use groups or memory limits to cap usage per application.

e. Swap-Aware Scheduler Integration

- The CPU scheduler should consider swap usage when scheduling.
- Avoid scheduling tasks with high swap thrashing.
- Integrate I/O scheduling to prioritize swapping I/O during idle CPU cycles.

5.3 Multi-Application Memory Example

Assume a system with:

- 16 GB RAM
- 32 GB dedicated swap partition
- 5 applications: A (video editing), B (IDE), C (browser), D (database), E (backup utility)

Plan Execution:

1. Application A and D are high-priority; they receive the largest RAM share.
2. Applications C and E are background; swap their inactive pages aggressively.
3. LRU or CLOCK algorithm ensures frequently accessed pages remain in RAM.
4. The swap partition manages overflow from applications C and E.
5. SSDs are preferred for swap devices to reduce latency.

8 (a) Designing a file system for cloud or mobile operating systems involves unique challenges: constrained resources, remote access, high concurrency, and the need for strong data protection. A robust design must address file access methods, directory structures, mounting, sharing, and security. Performance must be optimized using smart storage strategies including disk scheduling, RAID, and swap-space management.

1. Design Considerations for Modern File Systems

File systems designed for traditional desktop OSs often assume direct control over local storage devices. However, cloud and mobile environments introduce unique challenges such as intermittent connectivity, distributed storage, multiple client types, and security concerns. Therefore, a robust file system must meet the following criteria:

- **Scalability** to handle petabytes of data and millions of users.
- **Fault tolerance** to avoid data loss from hardware or network failures.
- **Low latency and high throughput** for user responsiveness.
- **Efficient metadata handling** for billions of small files.
- **Access control and encryption** for data privacy.

2. File Access Methods

A versatile file system must support multiple file access methods to accommodate various application requirements:

2.1 Sequential Access

In mobile systems, streaming media and document viewing are common use cases that benefit from sequential access. This method reads data in order and minimizes seek time.

2.2 Direct (Random) Access

Direct access is essential in cloud environments where large databases and analytics workloads access specific file segments. It allows fast access to individual blocks without scanning the whole file.

2.3 Indexed Access

For cloud-based applications like search engines or document storage systems, indexed access supports fast retrieval using an index structure (e.g., B-tree). It is particularly useful in NoSQL database storage engines or structured object storage.

2.4 Cloud-Native Access (Object Storage)

Cloud systems use object-based storage access, where files are treated as objects identified by unique keys. This decouples the data from physical location and allows scalable, distributed data access.

3. Directory Structure Design

3.1 Tree-Based Directory Structure

The proposed file system uses a tree-based directory structure with support for symbolic links. This hierarchical structure allows easy navigation, simplifies namespace organization, and supports nested directories for logical grouping.

3.2 Metadata Separation

To support billions of files efficiently, metadata is decoupled from the file content and stored in distributed metadata servers. This allows faster updates and concurrent access without file locking.

3.3 Caching and Synchronization

On mobile devices, caching of directory contents allows offline access. Sync algorithms like rsync or delta encoding are used to update only the changed parts of a file or directory, reducing bandwidth usage.

4. File System Mounting and Namespace Management

4.1 Mounting Techniques

Cloud-based file systems often use **network-mounted** protocols such as NFS or SMB, or they employ **FUSE-based** mounting to allow user-space file systems.

Mobile operating systems typically **mount file systems as user storage partitions**, with additional application sandboxing to restrict access.

4.2 Namespace Virtualization

To handle user-specific data securely in the cloud, namespace isolation is implemented using:

- Per-user logical namespaces
- Application-specific directories
- Access keys or tokens mapped to virtual paths

This allows multiple tenants or users to operate within the same physical storage infrastructure without data leakage.

5. File Sharing and Synchronization

5.1 Sharing Mechanisms

A robust file system includes:

- **Link-based sharing** (URL with access tokens)
- **Collaborative sharing** (real-time co-editing with locking mechanisms)
- **Access delegation** (temporary read/write privileges)

Cloud file systems (e.g., Google Drive) use these models to allow flexible and secure file access between users and devices.

5.2 Conflict Resolution

To manage concurrent edits in distributed settings:

- Operational transformation (e.g., in collaborative docs)
- Version control and conflict logs
- Timestamp-based last-writer-wins policies

Mobile systems sync changes upon network reconnection using these models to prevent data loss.

6. File Protection and Security

6.1 Access Control

Files are protected using:

- **Discretionary Access Control (DAC)**: Users set permissions (e.g., read/write/execute).
- **Role-Based Access Control (RBAC)**: Common in enterprise cloud file systems.
- **Mandatory Access Control (MAC)**: Used in security-focused mobile OSs (e.g., Android's SEAndroid).

6.2 Encryption

- **At-rest encryption**: Files are stored using AES-256 or similar standards.

- **In-transit encryption**: TLS/SSL ensures secure communication over networks.

6.3 Authentication

Authentication mechanisms include OAuth2, biometrics (mobile), and multi-factor authentication (MFA) for access.

6.4 Audit Logging

Modern systems include audit logs for file access, changes, and sharing events to ensure accountability and compliance.

7. Mass Storage Optimization

7.1 Disk Scheduling Algorithms
a. C-SCAN (Circular SCAN)

Provides uniform wait time for I/O requests by scanning in one direction and jumping to the start. Suitable for multi-tenant systems with consistent loads.

b. Deadline Scheduling

Used in latency-sensitive mobile systems and virtual machines where I/O operations must meet timing constraints.

c. Grouped Scheduling

Batching read/write operations improves throughput. Cloud systems often coalesce I/O operations before writing to disk.

7.2 RAID in Cloud/Mobile Systems
a. RAID in Cloud

RAID is used within data centers to improve performance and fault tolerance.

- **RAID 0**: Used for performance (e.g., caching layers).
- **RAID 1/5/6**: Used in backend storage for redundancy and recovery.

Distributed RAID-like redundancy, such as in Google File System (GFS) and HDFS, replicates files across nodes.

b. RAID in Mobile Devices

Due to limited storage, RAID is not common on individual devices. However, techniques like wear leveling and bad block management on SSDs resemble RAID logic.

7.3 Swap-Space Management
a. Cloud Systems

Use large swap spaces to offload inactive VM memory pages. Dynamic ballooning and memory overcommitment techniques help manage multiple workloads efficiently.

b. Mobile OS

Swap is limited due to power and performance constraints. However, Android introduced **zRAM** to compress memory and swap to a virtual RAM disk, improving multitasking without physical disk I/O.

8. Case Studies of Real-World Systems

8.1 Google File System (GFS)

- Designed for large-scale data-intensive applications.
- Uses large chunk files and replicates them on multiple nodes.
- Metadata is centrally managed for performance.
- Optimized for write-once, read-many scenarios.

Lessons for our design:

- Replication is better than RAID for distributed fault tolerance.
- Separate control (metadata) and data planes improve performance.

8.2 Amazon S3

- Object storage with scalable, key-based access.
- Provides features like versioning, lifecycle rules, and multi-region replication.
- Uses REST APIs for access, making it suitable for mobile/cloud.

Lessons:

- Object storage is better than block/file storage for cloud-native apps.
- Namespace abstraction and access keys improve security and usability.

8.3 Apple iCloud

- Seamlessly integrates file storage with mobile devices.
- Automatically syncs documents and photos across devices.
- Uses content hashes and deduplication to optimize space.

8(a)(i) If 100 libraries are loaded at startup, each requiring one disk access, with a seek time of 10 ms and a rotational speed of 6000 rpm, how long does it take to load all libraries?

To determine the total time required to load 100 libraries, we must consider the **seek time**, **rotational latency**, and **data transfer time** per access. Since the question assumes one disk access per library, we calculate the time per access and multiply it by 100.

1. Understanding Disk Access Time

Disk access time consists of:

1. **Seek time**: Time to move the read/write head to the desired track.
2. **Rotational latency**: Time taken for the desired sector to rotate under the read/write head.
3. **Data transfer time**: Time taken to transfer the data from the disk to memory.

Given:

- Seek time = 10 ms
- Rotational speed = 6000 rpm
- Number of libraries = 100

Note: Data transfer time is generally very small compared to seek and rotational latency, so we can ignore it unless specified.

2. Rotational Latency Calculation

To compute the **average rotational latency**, we use the formula:

Average Rotational Latency=12×Time for one rotation\text{Average Rotational Latency} = \frac{1}{2} \times \text{Time for one rotation}Average Rotational Latency=21×Time for one rotation

Rotational speed is 6000 rpm, which means the disk completes 6000 revolutions per minute. Convert this to **revolutions per second**:

6000 rpm=600060=100 revolutions per second6000 \, \text{rpm} = \frac{6000}{60} = 100 \, \text{revolutions per second}6000rpm=606000=100revolutions per second

So, time for **one full revolution** is:

1100=0.01 seconds=10 ms\frac{1}{100} = 0.01 \, \text{seconds} = 10 \, \text{ms}1001=0.01seconds=10ms

Thus, average rotational latency:

102=5 ms\frac{10}{2} = 5 \, \text{ms}210=5ms

3. Total Time Per Access

Each access requires:

- Seek time = 10 ms
- Average rotational latency = 5 ms

Total access time per library=10+5=15 ms\text{Total access time per library} = 10 + 5 = 15 \, \text{ms}Total access time per library=10+5=15ms

For 100 libraries:

Total time=100×15=1500 ms=1.5 seconds\text{Total time} = 100 \times 15 = 1500 \, \text{ms} = 1.5 \, \text{seconds}Total time=100×15=1500ms=1.5seconds

Final Answer (i):

The total time to load all 100 libraries is **1.5 seconds** assuming one disk access per library and ignoring data transfer overhead.

8(b)(ii) A FAT-based file system uses 4-byte entries. Given a 100 MB disk and 1 KB data blocks, what is the maximum file size?

To calculate the **maximum file size** supported by a FAT (File Allocation Table)-based file system, we need to understand how FAT works and how it limits file sizes based on the number of data blocks it can address.

1. Key Parameters

- Disk size = 100 MB = 100 × 1024 × 1024 = 104,857,600 bytes
- Block size = 1 KB = 1024 bytes
- FAT entry size = 4 bytes

Each **file** in a FAT system consists of a linked list of **data blocks**, where each block contains a pointer to the next block (via the FAT entry). A 4-byte FAT entry can support a large number of blocks.

2. Total Number of Data Blocks

First, determine how many data blocks are there in the 100 MB disk:

104,857,600 bytes1024 bytes/block=102,400 blocks\frac{104,857,600 \, \text{bytes}}{1024 \, \text{bytes/block}} = 102,400 \, \text{blocks}1024bytes/block104,857,600bytes=102,400blocks

So, the entire disk consists of **102,400 data blocks**.

3. FAT Size (Overhead)

Each block requires a 4-byte FAT entry. The size of the FAT itself is:

102,400×4=409,600 bytes=400 KB102,400 \times 4 = 409,600 \, \text{bytes} = 400 \, \text{KB}102,400×4=409,600bytes=400KB

This FAT must be stored on the disk itself, reducing the space available for actual data. However, for this problem, we are interested in **the maximum size a single file can grow to**, not the total storage available.

4. Maximum File Size Calculation

Each block can hold 1 KB of data. If a single file used **all** the blocks available in the system (assuming FAT can point to all of them), then:

Max file size=Number of addressable blocks×Block size\text{Max file size} = \text{Number of addressable blocks} \times \text{Block size}Max file size=Number of addressable blocks×Block size

The number of blocks that can be addressed by a 4-byte FAT entry = 232=4,294,967,2962^{32} = 4,294,967,296232=4,294,967,296 entries

So in theory, a FAT system with 4-byte entries can support over 4 billion data blocks. But the disk in this case only has **102,400 blocks**, so that is the upper bound.

Max file size=102,400×1024=104,857,600 bytes=100 MB\text{Max file size} = 102,400 \times 1024 = 104,857,600 \, \text{bytes} = 100 \, \text{MB}Max file size=102,400×1024=104,857,600bytes=100MB

Thus, on this 100 MB disk, a single file can span **the entire disk**, assuming no other files are present and FAT overhead is disregarded.

5. Practical Limitations

In practice:

- The FAT itself occupies space on the disk (400 KB).
- There is overhead for root directory, boot sector, and possibly reserved sectors.
- Not all 102,400 blocks will be available for a single file.

However, assuming that all blocks are usable and the FAT overhead is managed separately, the maximum theoretical file size remains **100 MB**.

FAT File System Characteristics

- **Simplicity**: FAT is straightforward to implement, making it ideal for small-scale storage systems, embedded systems, and USB drives.
- **Limitations**: FAT32 (32-bit entries) limits file sizes to 4 GB. In this case, 4-byte (32-bit) entries are more than sufficient for addressing a 100 MB disk.

Final Answer (ii):

The maximum file size supported on a 100 MB disk with 1 KB blocks and 4-byte FAT entries is **100 MB**, as a file can use all available data blocks on the disk.