Deadlock

- A state where a set of processes are **blocked**, waiting for resources held by each other.
- Like two people waiting to get through the same doorway - neither can move until the other steps back.

Example:

- Process A needs resource R1 (printer) and then R2 (scanner).
- Process B needs R2 (scanner) and then R1 (printer).

If process A grabs R1 and B grabs R2, they'll deadlock - A waits for B to release R2, B waits for A to release R1. Neither can proceed.

Deadlock Problem

- A deadlock is a situation in computer systems where two or more processes are blocked forever because they are each waiting for a resource held by another process in the group.

Conditions/Characterization of Deadlock:

- **Mutual exclusion:** Only one process can access a resource at a time.
- **Hold and wait:** A process holding a resource can request another resource while still holding the first one.
- **No preemption:** Once a resource is allocated to a process, it cannot be forcibly taken away.
- **Circular wait:** A set of processes form a chain where each process is waiting for a resource held by the next process in the chain.

Preventing Deadlock

Deadlock prevention ensures a system never enters a deadlocked state. This is achieved by violating at least one of the four conditions necessary for deadlock to occur.

Here's how to prevent deadlock by focusing on the possible violations:

- **Preemption:** If feasible, allow forcible resource removal from a process. For example, a database system might temporarily revoke a table lock from a process if another process urgently needs it. (This can be risky and might cause data inconsistency).

- **Ordered Resource Allocation:** Assign a linear order to resources (e.g., printer > stapler > scanner). Processes must request resources in strictly increasing order. This ensures no circular wait can form. (May lead to inefficient resource utilization).
- **Claim Strategy:** Processes declare their maximum resource needs upfront. The system only grants requests if it can guarantee the process can eventually acquire all its needs. This prevents processes from requesting more than they can ever use. (May lead to delayed process execution).

Deadlock Avoidance

- **Goal:** Prevent deadlocks from happening in the first place.
- **Method:** The system analyzes resource requests and availability to ensure granting a request won't lead to a future deadlock.
    - **Example:** Banker's algorithm predicts if allocating resources to a process will leave the system in a safe state (no deadlocks possible).
- **Advantage:** Proactive approach, guarantees no deadlocks.
- **Disadvantage:** May impose restrictions on resource allocation, reducing system efficiency.

Deadlock Detection and Recovery

- **Goal:** Identify and resolve deadlocks if they occur.
- **Method:** The system periodically checks for cycles in a wait-for graph (processes waiting for resources held by other processes).
    - **Example:** A cycle in the wait-for graph indicates a deadlock. Recovery involves rollback (undoing actions) or terminating a process to break the deadlock.
- **Advantage:** Less restrictive than avoidance, allows for more dynamic resource allocation.
- **Disadvantage:** Deadlocks can cause wasted time and system resources before being detected.

In simpler terms:

- Avoidance: Play it safe, plan resource allocation to avoid deadlocks.
- Detection & Recovery: Let it happen, but have a plan to identify and fix deadlocks when they occur.

File System

What it is: A file system is a method used by an operating system to organize, store, and manage files and folders on a storage device like a hard drive, solid state drive (SSD), or USB flash drive.

What it does:

- Defines how data is stored on the device.
- Keeps track of where files are located.
- Enables users to access and manage files through a hierarchical directory structure (folders within folders).
- Provides metadata (information about the file) such as file size, creation date, and permissions.

Common File Systems:

- **NTFS (New Technology File System):** Used by Windows NT and later versions (Windows 10, 11). Offers features like file permissions, encryption, and journaling.
- **FAT (File Allocation Table):** Used in older MS-DOS systems and some digital cameras. Simpler structure, less secure.
- **HFS+ (Hierarchical File System):** Used by Apple Macintosh computers.
- **ext4 (Fourth Extended Filesystem):** Commonly used on Linux systems. Supports large files and partitions.

Why it's important: Without a file system, data would be stored in a disorganized way, making it difficult to find, access, or manage specific files.

File Concept

- **Function:** A file is a digital container that stores information on a computer.
- **Storage:** Files reside on secondary storage devices like hard drives or solid-state drives (SSDs) for persistent storage, unlike RAM which is volatile.

Attributes of a File:

- **Name:** The unique identifier of the file.
- **Type:** Indicates the format or content type of the file (e.g., text file, executable file).
- **Location:** Physical location on the storage device where the file is stored.
- **Size:** The amount of data the file occupies.
- **Protection:** Permissions specifying who can access and modify the file.
- **Time and Date:** Metadata indicating when the file was created, accessed, and last modified.

Identification: Each file has a unique name to distinguish it from others. File extensions (like .txt, .jpg, .mp3) often indicate the file type and associated program for opening it.

Organization: Files are organized within directories (also called folders) to create a hierarchical structure for easy access and management.

Example: A document named "report.docx" stores your written work. It's a text file with a .docx extension, likely created with Microsoft Word.

File Access Methods

File access methods define how data is read from and written to a file. Here are the three main methods:

1. Sequential Access:

- Simplest method, processes data one record at a time, starting from the beginning.
- Like reading a book, you go from page to page.
    - **Examples:** Reading log files, playing videos.

2. Direct Access:

- Allows jumping directly to any specific location in the file.
- Like jumping to a specific chapter in a book with an index.
    - **Examples:** Database tables, accessing specific songs in a music file.

3. Indexed Sequential Access:

- Combines features of sequential and direct access.
- Uses an index to locate specific records quickly, but also allows sequential processing.
- Like using an index in a book to find a specific topic, then reading sequentially from there.
    - **Examples:** Large databases, file systems with fast search capabilities.

Directory Structure

- **Organization System:** A directory structure is a way to organize files on a computer's hard drive using a hierarchy of folders. Imagine filing cabinets with drawers and sub-drawers.
- **Tree Structure:** Folders (also called directories) are visualized as a tree, with a single root directory at the top and subfolders branching out from it.

Benefits:

- **Easier to find files:** Files grouped by category make them easier to locate.
- **Improved Organization:** Reduces clutter and keeps things tidy.

- **Efficient Management:** Simplifies file management tasks like backup and access control.

Examples:

- Common OS Structure (Windows/Mac):
    - Downloads folder for files retrieved from the internet.
- Project Specific Structure:
    - A web development project might have folders for HTML, CSS, JavaScript files.

Types of Directory Structures

1. Single-Level Directory Structure:

- Simplest structure with only a single directory called the root directory.
- All files reside directly in the root directory.
- Limited organization, becomes messy with many files.
    - Example: Early floppy disks with limited storage.

2. Two-Level Directory Structure:

- Introduces user directories under a root directory.
- Each user gets a dedicated directory for their files.
- Improves organization but lacks hierarchy for complex file systems.
    - Example: Some early multi-user operating systems.

3. Tree-Structured Directory Structure (Most Common):

- Files and subdirectories organized hierarchically, resembling an upside-down tree.
- Root directory at the top, with subdirectories branching out.
- Subdirectories can contain further subdirectories and files.
- Efficient for organizing large numbers of files.
    - Example: Modern operating systems like Windows, macOS, and Linux.

4. Acyclic-Graph Directory Structure (Less Common):

- Similar to tree structure but allows for some looping.
- Limited loops achieved through symbolic links or aliases.
- Enables multiple paths to reach the same file or directory.
- Used in specific scenarios for flexibility, but can introduce complexity.
    - Example: Version control systems like Git.

5. General-Graph Directory Structure (Rare):

- Most complex structure with unrestricted loops and connections.
- Difficult to manage and maintain due to potential inconsistencies.
- Rarely used in practice due to the challenges it presents.

File System Protection

File system protection refers to the methods used to secure files from unauthorized access, modification, or deletion.

- **Goal:** Maintain data confidentiality, integrity, and availability.

Techniques:

1. Access Control:

- **Permissions:** Defines who can access a file (user, group, everyone) and what they can do (read, write, execute).
    - Example: Setting a document as "read-only" for everyone except the owner.
- **Access Control Lists (ACLs):** Granular control specifying access permissions for individual users or groups.
    - Example: An HR department setting ACLs to restrict access to employee salary data.

2. Encryption: Scrambles file content using a key, making it unreadable without authorization.

- Example: Encrypting financial records for secure storage.

3. Auditing: Tracks access attempts and modifications made to files.

- Example: Monitoring login attempts to detect suspicious activity.

Benefits:

- Protects sensitive information (e.g., financial data, personal records).
- Prevents accidental or malicious file modification.
- Ensures data is accessible to authorized users only.

File System Structure:

- **Function:** Organizes files and folders on a storage device (hard drive, USB drive etc.) for efficient storage, retrieval and access.

Components:

- **Directories (Folders):** Contain files and other subdirectories, creating a hierarchical structure like a tree. (Imagine a filing cabinet with drawers and sub-drawers)
    - **Example:** /home/user/documents/work/ (work folder inside documents folder inside user's home directory)
- **Files:** Store actual data like text, images, videos etc.
- **File Metadata:** Additional information about a file like size, creation date, permissions etc.

Logical vs Physical Structure:

- **Logical Structure:** The tree-like directory structure users see. (The way folders and files are organized for us)
- **Physical Structure:** The way data is actually stored on the storage device (often in scattered locations). The file system manages this behind the scenes.

Directory Implementation in File System

- **Importance:** Directory implementation is a crucial part of file systems as it organizes files and allows users to navigate and access them efficiently.

Data Structures: Different data structures are used for directory implementation, each with its advantages and disadvantages. Here are common approaches:

1. Linear List:

- Simple to implement
- Stores filenames with pointers to their data blocks in a sequential order.
    - **Example:** Imagine a list of entries like "text.txt (points to data block 12)", "image.jpg (points to data block 37)".
- **Drawback:** Searching becomes slow for large directories as you need to scan the entire list.

2. Hash Table:

- Uses a hash function to quickly locate files based on their filenames.
- Faster search compared to linear lists.
    - **Example:** Imagine a hash table with slots for filenames. "text.txt" would be placed in a specific slot based on its hash value.
- **Drawback:** Hash collisions can occur when multiple filenames map to the same slot, requiring additional checks for resolving collisions.

3. B-Trees:

- Offer efficient searching, insertion, and deletion operations even for very large directories.
- Efficient for large-scale directory management with balanced performance for various operations.
- Used in some file systems like Btrfs.
    - **Example:** Imagine a B-Tree structure that keeps directory entries sorted and grouped into nodes. Searching involves traversing the tree based on filenames.
- **Drawback:** More complex to implement compared to simpler data structures.

4. Self-Balancing Trees:

- Similar to B-Trees, but automatically maintain a balanced structure for optimal search performance.
- Dynamically adjust to keep search operations efficient as the directory grows or shrinks.
    - Examples include AVL trees and Red-Black trees.
- **Drawback:** Increased complexity compared to linear lists and require more overhead for maintaining balance.

File Allocation Methods

File allocation methods determine how the operating system stores files on a disk. Each method has its advantages and disadvantages:

1. Contiguous Allocation:

- Stores files in a continuous block of disk space.
- **Advantages:** Fast file access (direct calculation of block addresses), simple to implement.
- **Disadvantages:** Internal fragmentation (wasted space at the end of disk due to non-perfect file size fit), difficult to handle file growth.
    - **Example:** Useful for fixed-size files like system programs.

2. Linked Allocation:

- Stores files in non-contiguous blocks, linked together by pointers in each block.
- **Advantages:** Efficient for file growth (add new blocks as needed), no internal fragmentation.
- **Disadvantages:** Slow file access (need to traverse the linked list), wasted space for pointers.
    - **Example:** Useful for dynamic files that grow or shrink frequently.

3. Indexed Allocation:

- Stores a table (index) containing block addresses for the file.
- **Advantages:** Fast random access (direct lookup in index table), flexible for file growth (update index table).
- **Disadvantages:** Overhead of maintaining the index table, wasted space for small files (index size remains same).
    - **Example:** Used in many modern file systems for balancing access speed and storage efficiency.

Free Space Management

- **What it is:** Free space management is a crucial part of file systems in operating systems. It tracks and allocates unused storage space on a disk drive for efficient storage of files.
- **Why it's important:** - Without proper management, free space can become fragmented, leading to wasted space and slower performance.
    - Efficient allocation ensures enough contiguous space is available for new files.

Techniques for Free Space Management:

1. Bit Vector:

- Uses a bit for each block on the disk.
- 0 = free block, 1 = allocated block.
- Efficient for finding the first free block.
    - Example: A disk with 64 blocks can be represented by 8 bits (1 byte). If blocks 2 and 4 are free, the byte would be 00010010.

2. Linked List:

- Links all free blocks together in a chain.
- Requires searching the list to find a block.
- Less wasted space compared to bit vector.
    - Example: Imagine free blocks 3, 5, and 7. Block 3 points to block 5, block 5 points to block 7, and block 7 has a null pointer signifying the end of the list.

3. Other Techniques:

- **Grouping:** Allocates fixed-size blocks and keeps track of free groups.
- **Counting:** Maintains a counter for the number of free blocks.