UNIT-V

FILE SYSTEM INTERFACE AND OPERATIONS

DEFINITION OF A FILE

A **file** is a **named collection of related information** that is stored on **secondary storage devices** such as hard disks or SSDs. It acts as the **smallest unit of logical storage** that the operating system can manage.

- Data **must be stored within files** to be written to secondary storage; we cannot directly write raw data.
- A file may contain **program code** or **data**.
- Data files can be of different types: **numeric**, **alphanumeric**, or **binary**.
- The **creator of the file** decides its purpose and what kind of content it holds.
- A file can store a wide range of content including:
    - Source code or executable programs
    - Text or numeric data
    - Photos, music, and video

Importance of Files in Secondary Storage

Files allow structured and permanent storage of information on secondary storage. Without files, the operating system would have no organized way of accessing, naming, or protecting stored data.

Types of Data Stored in Files

Depending on the usage, files can store:

- Source programs or compiled executables
- Structured or unstructured data
- Multimedia content like images, audio, or videos
- Configuration data or logs

The nature of data helps define the **file structure** and its behavior during read/write operations.

FILE TYPES

Each file has a structure that depends on its specific **type**:

- **Text File**: A sequence of characters organized into lines.
- **Source File**: A sequence of **functions**, where each function includes **declarations** and **executable statements**.
- **Executable File**: A series of **code sections** designed to be loaded into memory by the OS loader and executed.

FILE ATTRIBUTES

The operating system stores a number of attributes to manage and access files properly. These include:

- **Name**: Human-readable name used to identify the file.
- **Identifier**: A unique system-generated number used internally by the OS to refer to the file.
- **Type**: Indicates the file format (e.g., text, binary, executable).
- **Location**: A pointer to the actual storage location on the device.
- **Size**: Current size of the file, and possibly the maximum allowed size.
- **Protection**: Specifies the allowed operations on the file (read, write, execute) and who can perform them.
- **Time, Date, and User Identification**: Metadata including creation time, last modified time, and user information for security and monitoring.

Role of Directory Structure in File Management

The directory maintains the structure and access path for files on secondary storage:

- A **directory entry** includes the file's **name** and its **unique identifier**.
- This identifier helps locate other file attributes efficiently.
- Storing all directory information may require **more than a kilobyte per file**, especially in large systems.

FILE OPERATIONS

The operating system supports several fundamental operations on files. These are implemented using system calls:

1. **Creating a File**
    - System call: create ()

    - Checks for available space and creates a new directory entry if successful.
2. **Repositioning Within a File (Seek)**
    - Updates the **current-file-position pointer** to a specified value.
    - Does not require actual I/O; only internal pointer movement.
3. **Deleting a File**
    - System call: delete ()
    - Searches for the file in the directory, removes its entry, and releases the occupied storage space.
4. **Truncating a File**
    - Erases all the file contents but **retains file attributes**.
    - Resets the file length to zero.
5. **Writing to a File**
    - System call: write ()
    - Needs the file name and data to write.
    - The system locates the file and performs the write at the current position.
6. **Reading from a File**
    - System call: read ()
    - Needs the file name and location to store data.
    - Locates the file and reads from it.

Use of File Pointers During Operations

Whenever a file is being read or written by a process, the OS uses a **Current-File-Position Pointer**.

- This pointer is maintained per process.
- It is used by both read() and write() operations to determine where the next I/O should occur.

File Opening and Closing Mechanism

To access a file, it must first be opened using open() and then closed after use with close().

- The operating system uses an **Open-File Table** to track active files.
- When a file is opened:
    - An entry is made in the table.
- When a file is closed:
    - The corresponding entry is removed from the table.

Internal Tables Used for File Management

The OS uses **two levels of tables** to manage open files:

1. **Per-Process Table**
    - Maintained individually for each process.
    - Tracks files opened by the process.
    - Each entry points to the system-wide open-file table.
2. **System-Wide Open-File Table**
    - Maintains info that is independent of specific processes.
    - Stores data like file location on disk, access permissions, file size, and last modification time.
    - Once a file is opened by any process, an entry is made here.
    - If another process opens the same file, it adds an entry to its per-process table pointing to this system-wide table.

Information Maintained for Open Files

When a file is open, the system keeps the following data:

- **File Pointer**
    - Unique for each process.
    - Tracks the current I/O position.
    - Stored separately from file attributes on disk.
- **File-Open Count**
    - Counts how many processes have the file open.
    - Decrements with each close() call.
    - When it reaches zero, the file's entry is removed from the open-file table.
- **Disk Location of File**
    - Maintained in memory for faster access during I/O operations.
    - Avoids repetitive disk reads.
- **Access Rights**
    - Specifies the allowed operations (read, write, execute) for the process.
    - Stored in the per-process table.

Internal File Structure

Managing the internal organization of files is an important function of the operating system. This involves how data is stored, accessed, and aligned with the physical hardware characteristics such as disk blocks.

Locating an Offset Within a File

- Locating a particular **offset** (position) inside a file can be a challenging task for the operating system.
- This complexity arises because files are logically divided into **records**, but the disk hardware accesses data in fixed-sized units called **blocks**.

Disk Block Size and Disk I/O

- Every disk system has a **block size** that is well defined and is typically the size of a sector.
- Disk input/output operations are always performed in units of one block, also called a **physical record**.
- All disk blocks are the same fixed size.
- This means all reading and writing to disk happens in chunks of this block size.

The Mismatch Between Logical Records and Physical Blocks

- Logical records represent the meaningful units of data from the user's perspective, and their size may vary.
- The physical block size, however, is fixed by the hardware.
- Often, the size of logical records does **not** match the size of physical disk blocks exactly.
- This mismatch creates a challenge for storing data efficiently and for accessing logical records correctly.

Packing Logical Records into Physical Blocks

- To address the mismatch, **packing** is used to fit several logical records inside one physical block.
- This way, the system can store multiple smaller logical records together inside a fixed-size block.
- Packing improves the use of disk space and aligns logical data storage with the physical storage constraints.

Example: UNIX File System Model

- UNIX treats all files as a **stream of bytes**, rather than fixed-size records.
- Each byte can be accessed individually via its offset from the beginning of the file.
- This simple model allows flexible handling of data and avoids complications caused by fixed record sizes.

Factors Affecting the Number of Logical Records Per Block

The number of logical records stored in each physical block depends on several factors:

1. **Logical Record Size** – Size of each individual data unit as defined by the user or application.
2. **Physical Block Size** – The fixed size of blocks used by the disk hardware for I/O.
3. **Packing Technique** – The method or algorithm used to combine logical records into physical blocks.

Responsibility for Packing

- Packing can be done by the **user's application program**, which arranges records before writing.
- Alternatively, the **operating system** can handle packing automatically.
- Regardless of who performs packing, the goal is to efficiently utilize disk space and align logical data storage with physical disk blocks.

Internal Fragmentation in File Systems

- Internal fragmentation refers to wasted space inside an allocated block that is not used for actual data.
- All file systems experience some level of internal fragmentation because logical records rarely fill blocks exactly.
- The **larger the block size**, the **more internal fragmentation** occurs, since it is more likely that some space in the block remains unused.
- Therefore, choosing block size involves balancing efficient disk I/O and minimizing wasted space.

FILE ACCESS METHODS

Access Methods

When a file is accessed to read or write data, different methods can be used based on how the data needs to be retrieved or stored. The three main file access methods are:

1. **Sequential Access**
2. **Direct Access**
3. **Indexed Access**

Each method has its own way of locating and retrieving information from a file, with specific advantages and disadvantages.

Sequential Access

In sequential access, data in the file is processed **one record after another** in a fixed order.

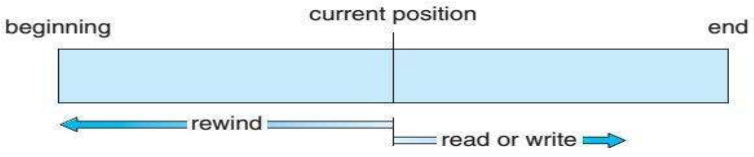
- **How it works:**
  - The file pointer starts at the beginning and moves forward as data is read or written sequentially.
  - Typical operations include:
    - **read_next( )**: Reads the next record and moves the pointer forward.
    - **write_next( )**: Appends data at the end and advances the pointer.
- **Common Uses:**
  - Used in applications like text editors and compilers where files are processed in a linear fashion.

Advantages

- Simple to implement and manage.
- Efficient when processing large amounts of data sequentially.
- Minimal overhead in maintaining file pointers.

Disadvantages

- Inefficient for accessing data randomly or searching for specific records.
- To access a record near the end, all preceding data must be read first.
- Not suitable for databases or applications needing fast random access.



Direct Access (Relative Access)

Direct access views the file as a collection of **fixed-length records or blocks** that can be accessed in any order.

- **How it works:**
  - Files are divided into numbered blocks.
  - Any block can be read or written directly using its block number (relative block number).
  - Operations:
    - **read(n)**: Reads block number *n*.
    - **write(n)**: Writes to block number *n*.
  - Common in disk-based file systems where random access is possible.
- **Example Use Case:**
  - Airline reservation systems where records for specific flights or customers are accessed directly.

Advantages

- Fast access to specific records without reading the entire file.
- Suitable for applications requiring random retrieval like databases.
- Efficient use of disk space when records are fixed-length.

Disadvantages

- Requires fixed-length records, which may lead to wasted space if records vary in size.
- More complex file management compared to sequential access.
  - User must know or compute the correct block number for access.

Indexed Access

Indexed access uses an **index structure** to map keys or record identifiers to their physical location in the file.

- **How it works:**
  - An index file contains pointers to blocks in the main data file.
  - Searching the index (often by binary search) identifies the block containing the desired record.
  - The data block is then accessed directly.

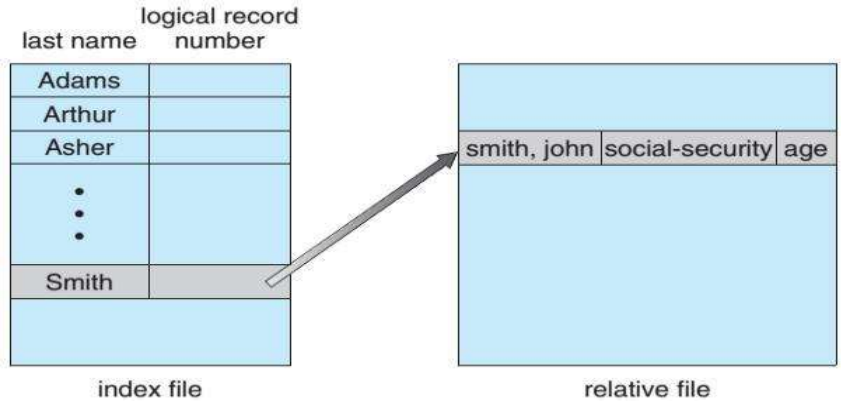
- **Handling Large Files:**
    - For very large files, multi-level indexes are used.
    - A primary index points to secondary indexes, which point to actual data blocks.

Advantages

- Allows very fast searching and retrieval of records.
- Efficient for large files where direct sequential or direct access would be slow.
- Reduces the number of disk I/O operations needed.

Disadvantages

- Maintaining the index adds overhead, especially for insertions and deletions.
- Index files themselves can become large, requiring additional storage and memory.
    - Slightly more complex to implement and manage compared to other methods.



DIRECTORY STRUCTURE

Directory Structure and File System Organization

Files are stored on random-access storage devices such as Hard Disks, Optical Disks, and Solid-State Drives (SSDs). These storage devices form the physical basis for file systems.

Partitioning of Storage Devices

A storage device can be divided into smaller parts called **partitions** to allow more precise control and management. For example, a hard disk can be divided into four quarters, with each quarter able to hold a separate file system.

- Partitioning serves several purposes:
    - It limits the size of individual file systems to manage space better.
    - It allows multiple types of file systems to exist on the same physical device.

- It reserves some space on the device for other uses, like swap space or raw (unformatted) space.

Volumes and File Systems

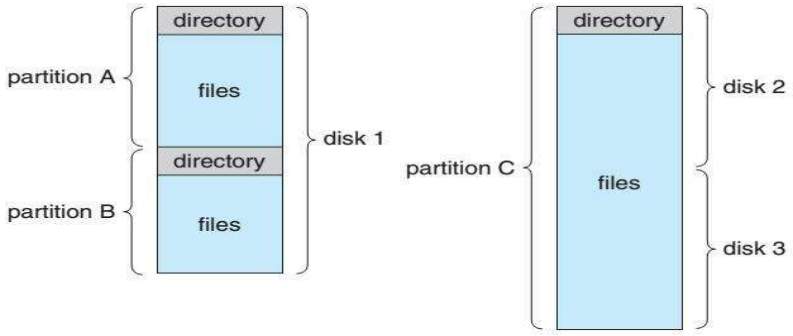
A **volume** is any entity (such as a partition or a whole device) that contains a file system. Each volume acts like a virtual disk to the operating system.

- Volumes can store multiple operating systems, enabling a computer to boot and run different OSs from the same physical device.
- Each volume maintains a **file system** that manages all the files stored within it.

Directory Structure within a Volume

The file system keeps information about all files in the volume through a **directory structure**.

- The **directory** or **volume table of contents** records details about each file, including:
  - File name
  - Location of the file on the device
  - Size of the file
  - File type
- The directory acts like an index or a catalog of files on the volume, enabling the operating system to quickly locate and manage files.



Operations on Directory

A directory plays a crucial role in managing files within a file system. Various operations can be performed on a directory to maintain and organize files efficiently. These operations include:

Searching for a File

- This operation involves looking through the directory structure to find an entry corresponding to a specific file.

- The search can locate files by matching names exactly or by matching a pattern (such as wildcard characters).
- It is essential for file access, as the directory provides the link between the file name and its metadata.

Creating a File

- When a new file is created, an entry for that file is added to the directory.
- This entry contains important details such as the file name, location, size, and type.
- Creating a file involves updating the directory to reflect the existence of the new file.

Deleting a File

- When a file is no longer required, its entry can be removed from the directory.
- Deleting the file frees up the space used by the file on the storage device and removes its metadata from the directory.
- Proper deletion ensures that the file system remains clean and does not waste resources.

Listing a Directory

- This operation lists all files contained in a directory.
- It displays the directory entries for each file, including details like file name, size, and type.
- Listing helps users or programs to view and manage the files stored in a particular directory.

Renaming a File

- Files can be renamed to reflect changes in their content or usage.
- For example, changing a filename from csec.txt to cse.txt or from cse.txt to cse.c involves updating the directory entry.
- Renaming changes the symbolic name of the file without altering the actual data stored.

Traversing the File System

- Traversal involves accessing every directory and every file within the directory hierarchy.
    - This operation is useful for tasks like backups, file searches across directories, or displaying a full directory tree structure.

.

DIRECTORY STRUCTURE

Directories organize and manage files on a storage system. Different directory structures have been designed to address various needs of users and systems. The common directory structures are:

1. **Single-Level Directory**
2. **Two-Level Directory**
3. **Tree-Structured Directory**
4. **Acyclic-Graph Directory**

1. Single-Level Directory

In a **Single-Level Directory**, all files are stored within a single directory.

- Every file shares the same directory space and must have a unique name.
- This structure is simple but has major limitations when the number of files grows or multiple users share the system.
- Users must remember unique filenames across the entire system, which becomes difficult as file count increases.
- Typically, a user may have hundreds of files, making management challenging.

Advantages

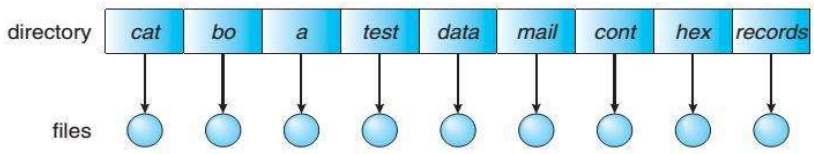
- Simple to implement and understand.
- Suitable for very small systems or single-user environments.

Disadvantages

- Filename uniqueness is required globally, which is restrictive.
- Difficult to manage when many files exist.
- Not suitable for multi-user systems due to naming conflicts.

Example

- Early personal computer systems or very basic file systems.



2. Two-Level Directory

The **Two-Level Directory** structure introduces separation between users:

- Each user has their own **User File Directory (UFD)** containing only their files.
- The system maintains a **Master File Directory (MFD)** indexing each user's UFD by username or account number.
- When a user logs in, only their UFD is searched, so multiple users can have files with the same name without conflicts.

Advantages

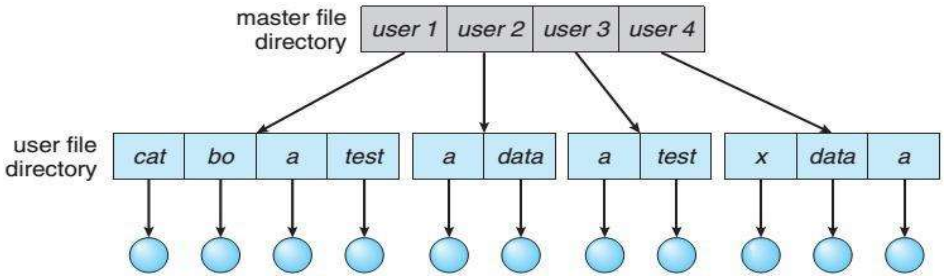
- Solves the name collision problem by isolating users' files.
- Users can have files with the same name in different UFDs.
- Easier to manage user files separately.

Disadvantages

- Isolates users completely, which is a problem if users want to share or cooperate on files.
- No direct sharing of files between users is possible.

Operating System Example

- Some early multi-user operating systems with simple directory management.



Tree-Structured Directories

The **Tree-Structured Directory** allows directories to contain files and subdirectories, creating a hierarchical tree.

- The root directory is at the top, and every file or subdirectory has a unique path name from the root.
- A directory entry marks whether it points to a file or a subdirectory.
- Each process has a **current directory** to simplify file access.
- Users can specify **absolute** or **relative** path names to access files.

Absolute vs Relative Pathnames

- **Absolute Pathname**: Starts at the root and specifies the complete path.
- **Relative Pathname**: Starts from the current directory.

Directory Deletion

- Empty directories can be deleted simply by removing their entry.
- If the directory contains files or subdirectories, either:
    1. The system refuses deletion until empty (user deletes contents first).
    2. The system deletes the directory and all its contents recursively (e.g., UNIX rm command).

Advantages

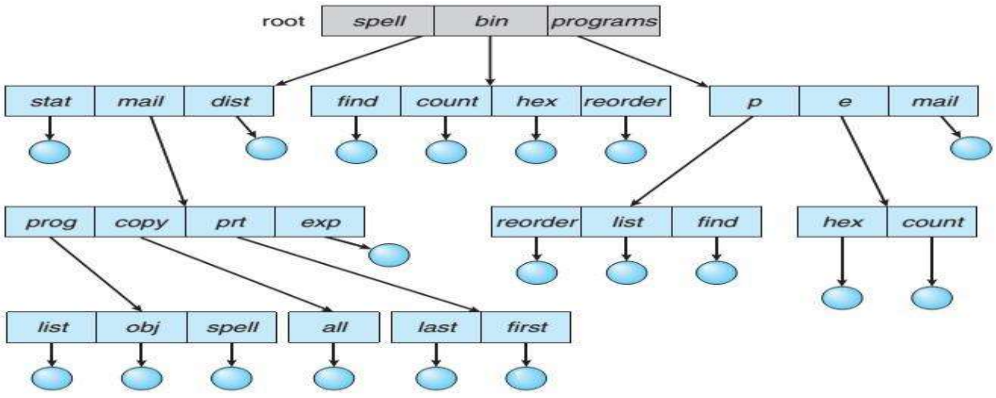
- Supports organization of files in a logical hierarchy.
- Enables user cooperation by allowing access to other users' files through pathnames.
- Simplifies file management for users.

Disadvantages

- Requires users to understand and manage directory paths.
- Slightly more complex to implement than simpler structures.

Operating System Example

- Most modern operating systems, including UNIX, Linux, Windows.



Acyclic-Graph Directories

An **Acyclic-Graph Directory** extends the tree structure by allowing files and subdirectories to be shared between directories.

- Unlike trees, this structure allows multiple directory entries to point to the same file or subdirectory.
- Sharing is enabled through **links**, which are pointers to files or directories.
- Sharing means changes in one place reflect everywhere the file or directory is linked.

How Links Work

- Links store the path to the real file or directory.
- When accessed, the link is resolved to locate the actual file.
- The system avoids cycles when traversing by ignoring links that would cause them.

Advantages

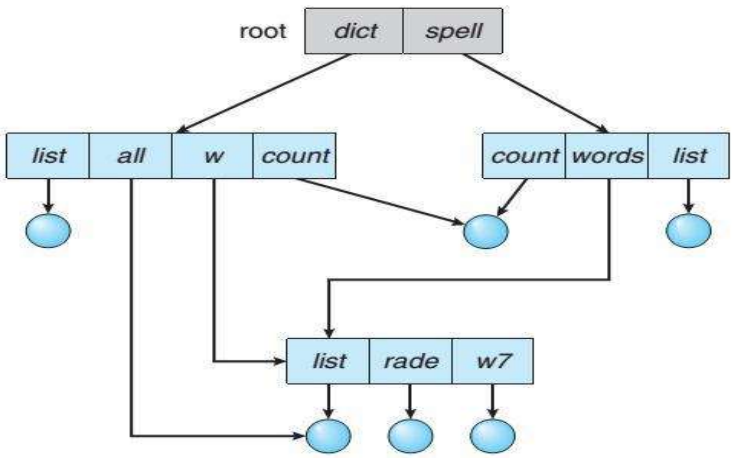
- Supports file and directory sharing, improving collaboration and storage efficiency.
- Changes in shared files are reflected system-wide instantly.

Disadvantages

- Files may have multiple pathnames, leading to complexity in file management.
- Deletion is complicated; deleting one link might leave "dangling" pointers elsewhere.
- Extra care is needed to manage file references to prevent errors.

Operating System Example

- UNIX and Linux implement links (hard and symbolic links) supporting acyclic graph directories.



FILE PROTECTION

Protection in operating systems controls access to files and directories to prevent unauthorized users from reading, modifying, or executing them. UNIX-like systems implement protection by assigning permission bits that specify allowed operations.

User Categories

There are **three categories** of users for each file/directory:

1. **Owner**: The user who created the file or directory.
2. **Group**: A set of users who share similar access rights.
3. **Universe (Others)**: All other users outside the owner and group.

Permission Bits Breakdown

UNIX permissions are stored in **7 bits**, but conceptually broken down as:

Permission Type	Number of Bits	Description
Read	4 bits	Permission to read the file or list the directory contents
Write	2 bits	Permission to modify the file or add/remove files in the directory
Execute	1 bit	Permission to run the file as a program or access the directory

Explanation of Bit Counts

- **Read (4 bits)**: Read permission requires more bits because it applies differently depending on the file or directory context (e.g., reading file content vs listing directory contents).
- **Write (2 bits)**: Write permission involves modifying file contents or directory entries.
- **Execute (1 bit)**: Execute permission means running the file or entering the directory.

File vs Directory Symbol

The first character in the permission string shows the **type** of the object:

Symbol	Meaning
-	Regular File
d	Directory

Permission String Format: A typical permission string looks like this:

drwxr-xr--

Breaking it down:

Position	Description
d	Indicates it’s a **directory**
rwx	Owner permissions: read, write, execute
r-x	Group permissions: read, no write, execute
r--	Universe permissions: read only

Bit-Level Representation

Let's analyze the permission bits for owner, group, and universe, considering the breakdown:

Owner Permissions (rwx)

- Read = 4 bits: 1111 (full read permission bits set)
- Write = 2 bits: 11 (full write permission bits set)
- Execute = 1 bit: 1 (execute allowed)

Together owner bits = 1111111 (7 bits)

Group Permissions (r-x)

- Read = 4 bits: 1111 (read allowed)
- Write = 2 bits: 00 (write not allowed)
- Execute = 1 bit: 1 (execute allowed)

Group bits = 1111001

Universe Permissions (r--)

- Read = 4 bits: 1111 (read allowed)
- Write = 2 bits: 00 (write not allowed)
- Execute = 1 bit: 0 (execute not allowed)

Universe bits = 1111000

Examples with Explanation

Permission String	Type	Owner Bits	Group Bits	Universe Bits	Meaning
-rw-r--r--	File	1111100	1111000	1111000	Owner can read/write, others read only
drwxr-xr-x	Dir	1111111	1111001	1111001	Owner full access, others read and execute
-r--------	File	1111000	0000000	0000000	Owner read only, others no access

FILE ALLOCATION METHODS

To store files on a disk, an operating system must allocate disk space efficiently and reliably. Three main allocation strategies are widely used:

1. **Contiguous Allocation**
2. **Linked Allocation**
3. **Indexed Allocation**

1. Contiguous Allocation

Concept:

- Files are stored in **contiguous blocks** on the disk — all blocks of a file are sequentially placed one after another.
- The directory entry stores:
  - The **starting block address** (e.g., block number b).
  - The **length** of the file in blocks (e.g., n blocks).
- Example: If a file is 5 blocks long starting at block 100, the file occupies blocks 100, 101, 102, 103, and 104.

Access Method:

- **Sequential Access**: Read blocks one by one from start to end.
- **Direct Access**: Access block i by calculating b + i directly — no need to traverse or search.
- Head movement is minimal when reading sequentially because the blocks are physically adjacent.

Advantages:

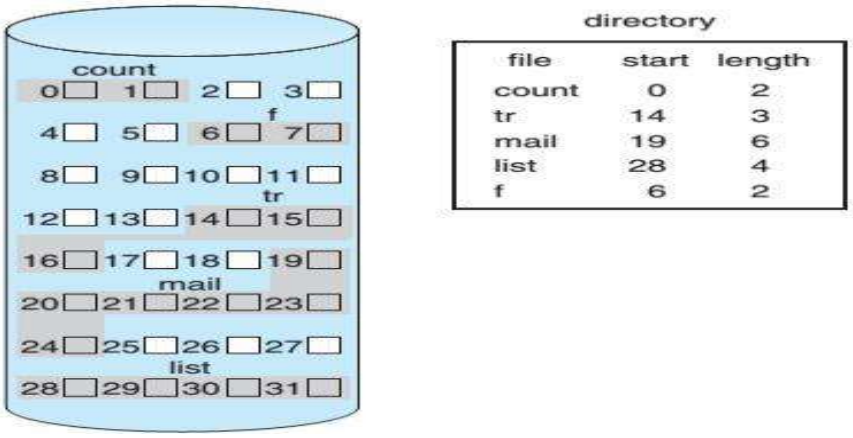
- Very simple to implement.
- Excellent for both sequential and random access — direct block addressing means quick reads/writes.
- Minimal metadata needed (only starting block and length).

Disadvantages:

- **External Fragmentation**: Over time, as files are created and deleted, free space becomes fragmented into small chunks scattered across the disk.
- Finding a large enough contiguous space for a new file or file extension becomes difficult.
- If space is insufficient or fragmented, file creation or extension fails unless compaction (rearranging files) is performed.
- **Compaction** is costly, especially on large disks, as it involves moving files to create a large free block.
- User/program must often estimate file size beforehand; underestimating wastes space or causes file growth failure.

OS Implementations:

- Simple file systems in early operating systems and some embedded systems.
- Early versions of FAT file systems use this concept partially.



2. Linked Allocation

Concept:

- Files are stored as a **linked list of disk blocks** scattered anywhere on the disk.
- Each block contains:
  - Data portion.

    - Pointer to the next block of the file.
- Directory entry stores the pointer to the **first block** of the file.
- No contiguous allocation needed — blocks can be anywhere.

Access Method:

- **Sequential Access**: Follow pointers from the first block through the chain.
- **Direct Access**: Inefficient, because to reach the ith block, OS must traverse through i-1 pointers sequentially.

Advantages:

- No external fragmentation because blocks can be anywhere on the disk.
- File size can grow dynamically without preallocating space.
- Simple to allocate new blocks by linking them to the end of the file.
- Free-space management is straightforward.

Disadvantages:

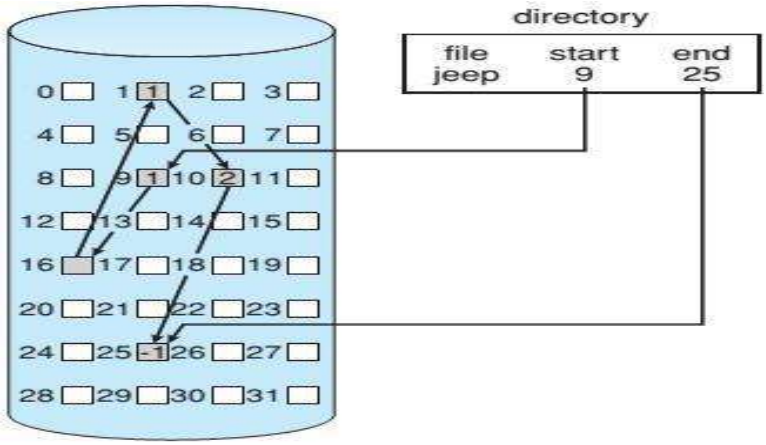
- Pointer overhead: Each block stores a pointer which uses some space (e.g., 4 bytes out of 512 bytes per block).
- Inefficient for direct/random access due to sequential traversal of pointers.
- Reliability issues: Corrupted or lost pointers may lead to loss of data or cross-linking errors.
- Disk head may need to move frequently, causing poor performance in random access.

Mitigation:

- **Clustering**: Allocate multiple blocks together as a cluster (e.g., 4 blocks). This reduces pointer overhead and improves throughput but increases **internal fragmentation** (wasted space within clusters).

OS Implementations:

- MS-DOS and early Windows FAT file systems use **File Allocation Table (FAT)** which is a variant of linked allocation.
    - Simple file systems in embedded devices.



File Allocation Table (FAT)

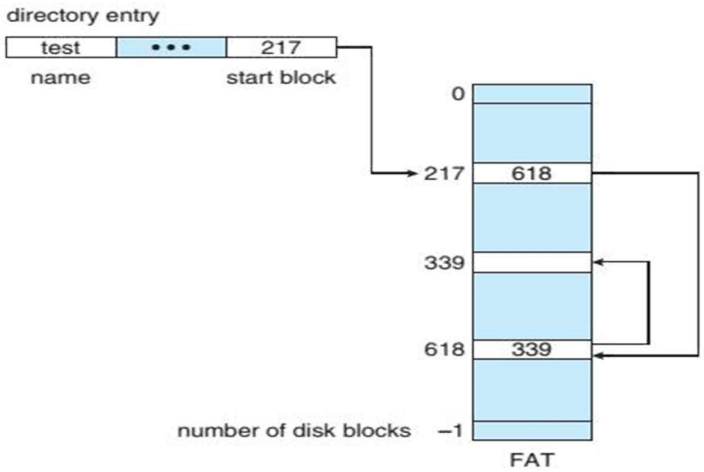
- FAT stores the linked list of blocks in a **table in memory** instead of in disk blocks.
- The FAT contains one entry per disk block.
- Directory entry points to the first block.
- FAT entries contain the address of the next block.
- End-of-file is marked with a special value.
- Allocating a block updates the FAT instead of modifying pointers on disk.

Advantages of FAT:

- Faster than disk-based linked pointers because FAT can be cached.
- Improves random access somewhat but still requires reading FAT entries.

Disadvantages of FAT:

- Large FAT tables consume memory.
    - Disk head must move between data and FAT locations, causing overhead.



Indexed Allocation

Concept:

- Each file has an **index block** (or inode in UNIX) that contains an array of pointers.
- Each pointer in the index block points directly to a data block of the file.
- Directory entry points to the index block.
- Supports **direct access** efficiently because OS can directly use the index block to get any data block.

Access Method:

- Direct access by reading the index block and then the desired data block.
- Sequential access by traversing pointers in the index block.

Advantages:

- No external fragmentation because blocks are allocated independently.
- Efficient direct access to any block.
- File size can grow dynamically; pointers are added to index block as needed.
- Metadata and data are separate, improving organization.

Disadvantages:

- Wasted space if the index block is large but file is small (pointer overhead).
- Fixed-size index blocks limit maximum file size unless advanced techniques are used.
- Accessing large files requires complex indexing schemes.

Handling Large Files in Indexed Allocation

a) Linked Index Blocks

- Index blocks linked together when more pointers are needed.
- Each block contains pointers to data blocks or next index block.

b) Multilevel Indexing

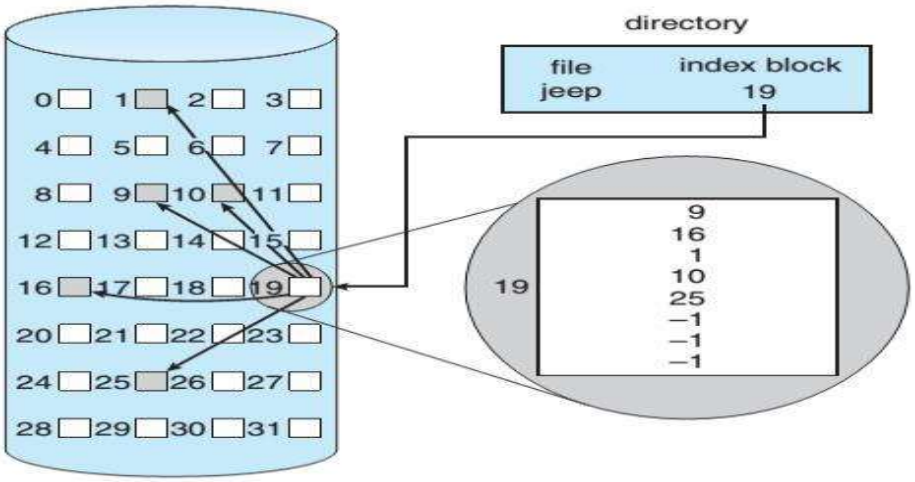
- First-level index block points to second-level index blocks.
- Second-level blocks point to actual data blocks.
- Can extend to multiple levels (triple, quadruple indirect blocks) for very large files.
- Example: UNIX uses 4KB blocks with 4-byte pointers — 1024 pointers per block.
- Two-level indexing can address over a million data blocks (~4GB file).

c) Combined Scheme (UNIX Inode Structure)

- Stores first 12 pointers as direct pointers to data blocks.
- Next three pointers are indirect:
  - Single indirect: points to a block of pointers.
  - Double indirect: points to a block of single indirect pointers.
  - Triple indirect: points to double indirect blocks.
- Efficient for small files (direct blocks) and scalable for large files (indirect blocks).

OS Implementations:

- UNIX and UNIX-like systems (Linux, BSD).
- NTFS (New Technology File System) uses a form of indexed allocation with extents and B-trees.
- Modern file systems like ext4 use advanced variants of this scheme.



FILE SYSTEM STRUCTURE – LAYERED ARCHITECTURE

What is a File System?

A **file system** is a method used by an operating system to **organize, store, retrieve, and manage files** on storage devices like hard disks, SSDs, or USB drives.

To manage files efficiently, the file system is structured in **layers** — each layer has a specific role in handling file operations.

Layered Structure of a File System

The file system works like a pipeline — each layer performs its part and passes the request down to the next layer.

Here are the layers from **top (user level)** to **bottom (hardware level)**:

1. Application Program

- **What it is**: This is the program written or used by the user (like Notepad, Word, C/C++ programs).
- **What it does**: Requests file operations like read, write, open, delete.
- **Example**: A C program writing output to result.txt.

User interacts with files through programs.

2. Logical File System

- **What it does**:
  - Checks if the requested file exists.
  - Collects the **logical block number** (used to locate the file inside the file system).
  - Handles **file name to inode mapping**, access control, and protection.
- **Example**: Finds that result.txt starts at logical block 105.

Acts like the file manager who finds where the file lives.

3. File Organization Module

- **What it does**:
  - Translates the **logical block number** to a **physical block number** (exact location on the disk).
  - Handles block allocation, free space management, and keeps track of file data blocks.
- **Example**: Maps logical block 105 → physical block 3200 on the disk.

Like converting a library's catalog number into the actual shelf location.

4. Basic File System

- **What it does**:
  - Issues commands to I/O control (like read/write).
  - Handles raw block I/O operations using the physical block number.
- **Example**: Issues command “read block 3200”.

It gives direct commands based on the translated location.

5. I/O Control Layer

- **What it does**:
  - Receives the I/O commands (read/write/print).
  - Converts them into **device-specific instructions**.
  - Uses device drivers to talk to hardware.
- **Example**: Sends "read block 3200" to the disk controller using the device driver.

Like an interpreter who knows how to talk to different devices.

6. Devices

- **What it is**:
  - The actual **hardware** that stores the files.
  - Can be hard drives (HDD), solid-state drives (SSD), USBs, etc.
- **What it does**:
  - Executes the operation (like read/write) on the physical storage.

The physical disk where your data is actually stored or retrieved.

FREE-SPACE MANAGEMENT

Disk space is a limited resource. As files are created, modified, and deleted, disk blocks get allocated and freed dynamically. Efficient management of free space on the disk is critical to ensure smooth operation and maximum utilization of storage.

Why Free-Space Management is Important?

- When a file is deleted or truncated, the disk blocks it occupied become free.
- These free blocks must be tracked so they can be reused for new files or for file extensions.
- Without proper free-space management, the system would not know which disk blocks are available, leading to wasted space or errors when allocating new files.

What is a Free-Space List?

- The **Free-Space List** is a data structure maintained by the operating system to keep track of all free disk blocks.
- It records all disk blocks **not currently allocated** to any file or directory.
- Whenever a new file is created or extended, the OS refers to the free-space list to find and allocate the required blocks.
- When a file is deleted, the freed blocks are returned back to the free-space list for future reuse.

Methods of Free-Space Management

There are several ways to manage the free space efficiently, depending on the size of the disk and the OS design:

1. Bit Vector (Bit Map)

- The free-space list is represented as a **bit vector (bit map)** — a simple array of bits.
- Each bit corresponds to one disk block.
- If the bit is **1**, the corresponding disk block is **free**.
- If the bit is **0**, the block is **allocated**.

Example:

Suppose the disk blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free, and the rest are allocated. The free-space bit vector might look like:

Block#: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 ... 27
Bits: 0 0 1 1 1 1 0 0 1 1 1 1 1 1 ... 1

- The bit vector efficiently shows which blocks are free.
- Searching for the first free block or a sequence of free blocks is fast.

Advantages:

- Simple and easy to implement.
- Very efficient to check free blocks.
- Good for relatively small to medium-sized disks.

Disadvantages:

- For very large disks, the bit vector can become huge.
- For example, a 1 TB disk with 4 KB blocks has about 256 million blocks.
- The bit vector would require 256 million bits ≈ 32 million bytes ≈ 256 MB of memory.
- Keeping such large bit vectors in main memory is expensive or impractical.

2. Linked List

- Free blocks are linked together forming a list.
- The system maintains a pointer to the **first free block**.
- Each free block contains a pointer to the **next free block**.
- The pointer to the first free block is stored in a known location on disk and cached in memory.

Example:

Given free blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27:

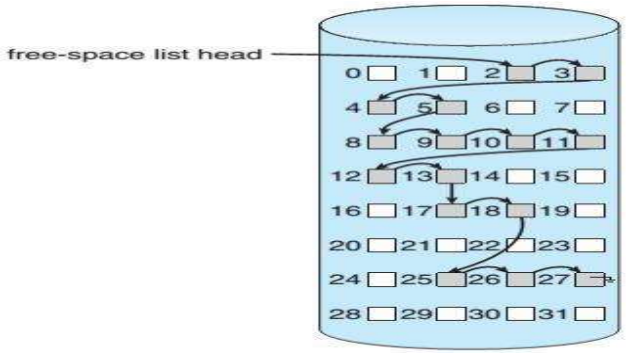
- The OS stores a pointer to block 2 (first free block).
- Block 2 contains a pointer to block 3.
- Block 3 points to block 4, and so on.

Advantages:

- Space-efficient as only pointers are stored in free blocks themselves.
- No large bit map required.

Disadvantages:

- To traverse the entire free space, the OS needs to read each block sequentially.
- This causes many disk I/O operations, which is slow.
- Not efficient for quickly finding large amounts of free space.



3. Grouping

- An improvement over simple linked list.
- The first free block stores the addresses of **n free blocks**.
- The last address in the block points to another block containing **n addresses** of free blocks.
- This creates a hierarchy or group of free blocks, allowing faster access.

How it works:

- The OS can find multiple free blocks at once by reading one block.
- Reduces disk I/O compared to simple linked lists.

Advantages:

- Faster access to free blocks.
- Less disk I/O compared to the simple linked list method.

Disadvantages:

- Slightly more complex to implement.
- Still requires sequential reading of groups.

4. Counting

- Used especially when free space is allocated in **contiguous blocks** (e.g., contiguous allocation or clustering).
- Instead of listing all free blocks individually, this method records:
    - The **starting block address** of a free space segment.
    - The **number of contiguous free blocks** following it.

Example:

- If blocks 100 to 110 are free contiguously, this is stored as (start = 100, count = 11).
- Similarly, (start = 200, count = 5) for another group of 5 contiguous free blocks.

Advantages:

- Very compact representation of free space.
- Efficient for allocating or freeing large contiguous spaces.

Disadvantages:

- Not suitable for fragmented disks with many small scattered free blocks.

5. Space Maps

- Used in advanced file systems (e.g., ZFS).
- Space maps are a **log-structured** representation of free and allocated blocks.
- They record changes (allocations and frees) incrementally in logs.
- This approach allows efficient tracking of free space with minimal overhead.
- Helps in fast recovery and consistency checks.

SYSTEM CALLS FOR FILE MANAGEMENT

In an Operating System, a **system call** is a way for a program (usually written in C/C++) to **request services from the kernel**. These services could be related to:

- Creating or opening a file
- Reading or writing data

- Creating a directory
- Getting file information, and more

System calls work as a **bridge between user-level applications and the core of the OS (kernel)**.

Why do we need system calls?

When you write a C program to open a file using `open()` or create a directory using `mkdir()`, your program can't directly communicate with the hardware. So, it makes a **system call**, which is a secure and predefined way to ask the kernel to perform the task.

File System Calls vs Directory System Calls

Category	File System Calls	Directory System Calls
Works on	Individual files (data/documents etc.)	Directories (folders)
Examples	`open()`, `read()`, `write()`, `lseek()`	`opendir()`, `mkdir()`, `readdir()`, `rmdir()`
Purpose	Manage contents inside files	Manage file structures and navigation

Introduction to System Calls

System calls are the fundamental interface between a user application and the operating system (kernel). They allow user programs to request services from the OS such as file handling, process control, and device management. File and Directory System Calls specifically handle operations like creating, opening, modifying, deleting, reading, and writing to files and directories.

System calls execute in kernel mode, which provides privileged access to system resources. These calls ensure controlled, secure access to hardware and file systems.

Common Header Files:

- `<unistd.h>` – For basic I/O calls like `read`, `write`, `close`, `lseek`
- `<fcntl.h>` – For file status flags like `O_CREAT`, `O_RDWR`
- `<sys/types.h>` and `<sys/stat.h>` – For defining data types and file metadata structure
- `<dirent.h>` – For directory handling functions like `opendir`, `readdir`, etc.

FILE MANAGEMENT SYSTEM CALLS

1. `creat()`

- **Purpose:** Creates a new file or truncates the file if it already exists.
- **Syntax:**

```
int creat(const char *pathname, mode_t mode);
```

- **Parameters:**
  - `pathname`: Full or relative path of the file to be created.
  - `mode`: Permission bits (e.g., 0644) set at creation, affected by current `umask()`.
- **Return:** File descriptor on success; -1 on failure.
- **Example:**

```
int fd = creat("example.txt", 0644);
```

- **Note:** Same as `open(path, O_CREAT | O_WRONLY | O_TRUNC, mode)`.

2. `open()`

- **Purpose:** Opens an existing file or creates a new file.
- **Syntax:**

```
int open(const char *pathname, int flags, mode_t mode);
```

- **Flags:**
  - `O_RDONLY`, `O_WRONLY`, `O_RDWR` — Access modes
  - `O_CREAT` — Create file if not exists
  - `O_TRUNC` — Truncate file to 0 length
  - `O_APPEND` — Append data to end of file
- **Returns:** File descriptor or -1 on failure.
- **Example:**

```
int fd = open("data.txt", O_CREAT | O_WRONLY, 0644);
```

3. `close()`

- **Purpose:** Closes an opened file descriptor.
- **Syntax:**

```
int close(int fd);
```

- **Returns:** 0 on success, -1 on failure.

4. `read()`

- **Purpose:** Reads bytes from a file into a buffer.
- **Syntax:**

```
ssize_t read(int fd, void *buf, size_t count);
```

- **Parameters:**
  - `fd`: File descriptor
  - `buf`: Pointer to memory buffer
  - `count`: Number of bytes to read
- **Returns:** Number of bytes read, 0 on EOF, -1 on error.

5. `write()`

- **Purpose:** Writes data from a buffer to a file.
- **Syntax:**

```
ssize_t write(int fd, const void *buf, size_t count);
```

- **Example:**

```
char msg[] = "Hello World";
write(fd, msg, strlen(msg));
```

6. `lseek()`

- **Purpose:** Moves the file read/write offset.
- **Syntax:**

```
off_t lseek(int fd, off_t offset, int whence);
```

- **Parameters:**
  - `fd`: File descriptor
  - `offset`: Number of bytes to move
  - `whence`: Starting point:
    - `SEEK_SET`: From beginning
    - `SEEK_CUR`: From current position
    - `SEEK_END`: From file end
- **Returns:** New offset value or -1 on error.

- **Usage Examples:**

```
lseek(fd, 0, SEEK_SET); // Move to beginning
off_t size = lseek(fd, 0, SEEK_END); // Get file size
```

7. `link()` (Hard Link)

- **Purpose:** Creates another name (alias) for an existing file.
- **Syntax:**

```
int link(const char *existingpath, const char *newpath);
```

- **Properties:**
  - Same inode for both paths
  - Reference count increases
  - File deleted only when last link is removed
- **Limitation:** Must be on the same filesystem

8. `symlink()` (Symbolic Link)

- **Purpose:** Creates a symbolic link to another file.
- **Syntax:**

```
int symlink(const char *target, const char *linkpath);
```

- **Properties:**
  - Different inode
  - Can span across filesystems
  - Stores path as string
  - Breaks if target file is removed

Comparison: `link()` vs `symlink()`

Property	`link()`	`symlink()`
Inode Shared?	Yes	No
Cross-filesystem?	No	Yes
Target Required?	Yes	Not strictly
Broken on target deletion?	No	Yes

10. `stat()`

Purpose:

- The `stat()` system call retrieves detailed information (metadata) about a **file or directory** given its **pathname**.
- It follows symbolic links, meaning if the pathname points to a symlink, `stat()` returns info about the actual file it points to.

Syntax:

```
int stat(const char *pathname, struct stat *statbuf);
```

Parameters:

- `pathname`: Path to the file or directory (string).
- `statbuf`: Pointer to a `struct stat` where metadata will be stored.

What is returned?

- Returns `0` on success.
- Returns `-1` on failure (e.g., if the file does not exist or permission denied).

What info does `stat` fill in `struct stat`?

- File type (regular file, directory, symlink, etc.)
- Permissions (read, write, execute for user/group/others)
- Owner user ID and group ID
- File size in bytes
- Timestamps: last modification (`st_mtime`), last access (`st_atime`), last status change (`st_ctime`)
- Number of hard links
- Device ID (if file is special device)

Example Usage:

```
#include <sys/stat.h>
#include <stdio.h>
int main() {
    struct stat fileStat;
```

```
    if(stat("myfile.txt", &fileStat) == 0) {
        printf("File size: %ld bytes\n", fileStat.st_size);
        printf("File permissions: %o\n", fileStat.st_mode &
0777);
    } else {
        perror("stat error");
    }
    return 0;
}
```

2. `lstat()`

Purpose:

- `lstat()` is similar to `stat()`, but it **does NOT follow symbolic links**.
- If the pathname is a symbolic link, `lstat()` returns info about the **link itself**, not the file it points to.

Syntax:

```
int lstat(const char *pathname, struct stat *statbuf);
```

Parameters:

- Same as `stat()`.

Return:

- `0` on success.
- `-1` on failure.

When to use `lstat()`?

- When you want to **inspect the properties of a symbolic link itself**, such as its length, permissions, or timestamps.
- To distinguish if a file is a symbolic link before deciding how to handle it.

Example:

```
if(lstat("mylink", &fileStat) == 0) {
    if (S_ISLNK(fileStat.st_mode)) {
        printf("It is a symbolic link.\n");
```

```
    }
}
```

3. `fstat()`

Purpose:

- `fstat()` obtains metadata for an **open file descriptor** instead of a pathname.
- Useful when you already have the file open (with `open()`), and want info about it.

Syntax:

```
int fstat(int fd, struct stat *statbuf);
```

Parameters:

- `fd`: Open file descriptor.
- `statbuf`: Pointer to a `struct stat` to be filled.

Return:

- `0` on success.
- `-1` on failure.

When to use `fstat()`?

- When file is already open and you want info without needing its path.
- More efficient in some cases since you avoid resolving pathnames again.

Example:

```
int fd = open("myfile.txt", O_RDONLY);
if (fd != -1) {
    if (fstat(fd, &fileStat) == 0) {
        printf("File size: %ld bytes\n", fileStat.st_size);
    }
    close(fd);
}
```

Summary Table of `stat()`, `lstat()`, and `fstat()`

System Call	Input Type	Follows Symlink?	Info Returned For	Typical Use Case
`stat()`	File pathname	Yes	Actual file (target)	Get file info by path, resolve symlinks
`lstat()`	File pathname	No	The link itself (if symlink)	Get info about symlink itself
`fstat()`	File descriptor (int)	Yes	File opened by descriptor	Get file info when file already open

11. `chmod()`

- **Purpose:** Change file permission bits.

Syntax:

```
int chmod(const char *pathname, mode_t mode);
```

Example:

```
chmod("file.txt", 0755); // rwxr-xr-x
```

12. `chown()`

Purpose: Change file's owner and group.

Syntax:

```
int chown(const char *pathname, uid_t owner, gid_t group);
```

DIRECTORY SYSTEM CALLS

1. `opendir()`

- **Purpose:** Opens a directory stream for reading contents.
- **Syntax:**

```
DIR *opendir(const char *name);
```

- **Returns:** DIR pointer or NULL

2. `readdir()`

- **Purpose:** Reads entries from directory stream one at a time.
- **Syntax:**

```
struct dirent *readdir(DIR *dirp);
```

- **Structure:**

```
struct dirent {
ino_t d_ino;       // Inode number
char d_name[256];  // Filename
};
```

3. `closedir()`

- **Purpose:** Closes the directory stream.
- **Syntax:**

```
int closedir(DIR *dirp);
```

4. `mkdir()`

- **Purpose:** Creates a new directory.
- **Syntax:**

```
int mkdir(const char *pathname, mode_t mode);
```

- **Example:**

```
mkdir("myfolder", 0755);
```

5. `rmdir()`

- **Purpose:** Removes an empty directory.
- **Syntax:**

```
int rmdir(const char *pathname);
```

- **Fails** if directory is not empty.

6. `umask()`

- **Purpose:** Sets the default permission mask for new files.
- **Syntax:**

```
mode_t umask(mode_t mask);
```

- **Example:**

```
umask(022);
creat("new.txt", 0666); // Final permission: 0644
```

`chmod()` vs `umask()`

Feature	`chmod()`	`umask()`
Usage	After file creation	Before file creation
Purpose	Explicit permission change	Mask out default permission bits
Scope	Affects one file	Affects all new files by process