6.1 File – Concept and Basic Operations

What is a File?

- A **file** is an organized collection of data or information, stored on secondary storage (like hard disk, SSD, CD).
- Think of it as a "container" for programs, documents, images, audio, videos, etc.
- The **operating system (OS)** acts as a mediator so users see files as logical entities, not a complex pattern of binary data on storage hardware.

Why Abstract File Storage?

- Storage devices have different physical characteristics (magnetic, optical, SSD).
- The OS provides a common file model for **user convenience**, regardless of hardware.

File Types – Name, Extension:

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

File Attributes

Files have various properties to help manage and secure them. Some main attributes:

Attribute	**Description**
Name	Human-readable file identifier (e.g., project.docx).
Identifier	Unique system-internal number (not visible to user) used by the OS.
Type/Extension	File format/type indicator (text, image, executable, etc.), usually by extension.
Location	Pointer to disk location (e.g., start block/cluster number).
Size	Shows the current length of the file in bytes or KB, MB, etc.
Protection	Who can read, write, execute the file (permissions/ACL).
Timestamps	Creation, last modified, accessed dates—helpful for security and management.
Owner/ID	User who owns the file (for multi-user systems).

Example Directory Entry (like a row in a table):

Name	FileID	Type	Location	Size	Owner	Created	Access Rights
notes.txt	10145	.txt	3320	14KB	Ansh	1/1/25	rw-r--r--

File Operations

The OS provides a set of system calls for users/programs to manipulate files.
Let’s discuss the major file operations:

1. Creating a File

- Space is reserved on disk.
- Directory updated with a new entry (with default attributes).
- **Example:** Saving a new Word document.

2. Writing to a File

- Data is written from program/user memory to the file.
- System keeps a **write pointer** marking where to store next data.

- Pointer updated as data is written.

3. Reading from a File

- Data is transferred from the file (disk) to main memory.
- Uses a **read pointer** to keep track of next data position.

4. Repositioning (Seeking)

- Changing the file pointer to a specific location in the file.
- Allows random access to different parts (e.g., seek to byte 50).
- Example: Music players skipping to a time in a song.

5. Deleting a File

- Directory entry is located and removed.
- Disk space released for future use.

6. Truncating a File

- Clears file contents but keeps its attributes and directory entry.
- File size becomes zero, allocated blocks freed.

Extra Operations

- **Append**: Add data to file end (useful for logs).
- **Rename**: Change the file’s name (directory entry updated).
- **Copy**: Duplicate file contents and attributes under a new name/location.

Open and Close

- **Open**: Loads file attributes from disk into a system structure (file control block) in RAM for quick access.
- **Close**: Updates info (e.g., file size, last modified time) and releases file control block from RAM to save resources.

File Types

- Files have types/extensions (.txt, .exe, .mp3) so the OS knows how to handle them.
- Some OSes (like Windows) use extensions heavily; others (e.g., Unix/Linux) rely more on headers or metadata.
- **Example:** Double-clicking .jpg opens an image viewer, .docx opens Word.

6.2 File Access Methods

How do programs actually use/read/write files? Two main ways:

A. Sequential Access

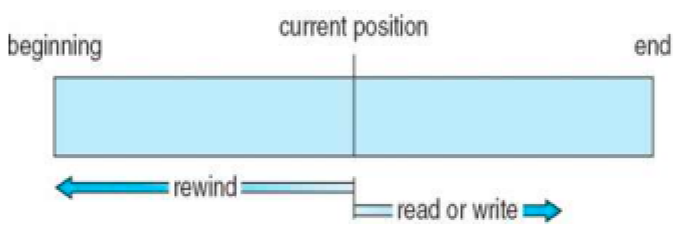
- Read or write operations proceed strictly **one after the other** as if reading a book from start to finish.
- Most common method: text editors, compilers, etc.
- **Operations:**
  - read next, write next (move pointer forward)
  - reset (go to file start)
- **Example:** Reading an ebook, watching a video.

Advantages:

- Simple to implement, efficient for certain use-cases.

Disadvantages:

- Cannot skip/jump directly to desired data (must process all previous data).



B. Direct Access (Random Access)

- File is divided into **fixed-size blocks** or records.
- Any block can be read/written *independently* using its position (number), without traversing from start.
- Used in **databases**, some large files, where speed is crucial.

Operations:

- `read n` (fetch block n)
- `write n`
- `seek n` (move pointer to block n)

Example: Jumping to a specific scene in a movie file; retrieving the 12th record in a database.

File Allocation Methods: How Files Use Disk Space

Allocation = how the operating system assigns disk blocks to files.

A. Contiguous Allocation

- Stores each file in a single, continuous sequence of blocks.
- Maintains starting block and file length.
- **Example:** If a file starts at block 20 and is 5 blocks long, it occupies 20, 21, 22, 23, 24.

Directory Entry:

- Start Block
- Length

Advantages:

- Supports both **sequential and direct access** (jump to offset easily).
- Fastest for sequential operations (all blocks adjacent).

Disadvantages:

- Suffers from **external fragmentation**—free space gets split into small holes, making it hard to find large blocks for big/new/growing files.

- Requires knowing file size in advance or reserving extra space.
- Might need *disk compaction* (repacking files) to consolidate space, which is slow.

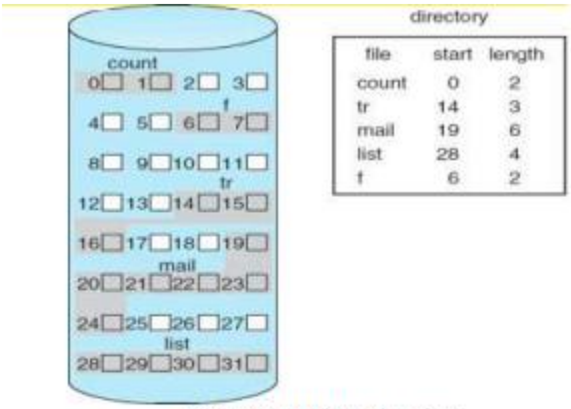


Fig. Contiguous file allocation

B. Linked Allocation

- Each file is a **linked list of disk blocks** scattered anywhere on the disk.
- Each block contains data + pointer to the next block.
- Directory stores a pointer to first (and possibly last) block.

Advantages:

- No external fragmentation; any free block can be reused.
- Ideal for *growing* files.
- Suited for **sequential access**.

Disadvantages:

- Not suitable for direct/random access; must follow pointers from the start block.
- Pointer overhead (few bytes per block wasted for linkage).
- Reliability risk: pointer corruption breaks file continuity.

Example:

- A file consists of blocks: 10 → 26 → 56 → 11 → 80

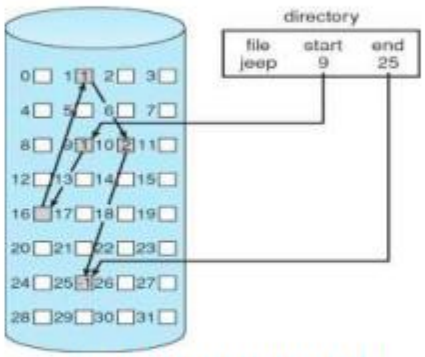


Fig. :-Linked allocation method

C. Indexed Allocation

- Each file has an **index block** (table) containing pointers to all its disk blocks.
- Directory entry contains address of the index block.
- To access block i, look up entry i in the index.
- Supports large files and both direct/sequential access.

Advantages:

- No external fragmentation.
- Supports **direct access** efficiently.
- Easy to expand files (by adding entries to index).

Disadvantages:

- Index block needs to be in RAM or loaded for every access.
- Small files waste space due to index block overhead; very large files might require multi-level indexes.

Example:

- Index block = , meaning file uses disk blocks 52, 24, …

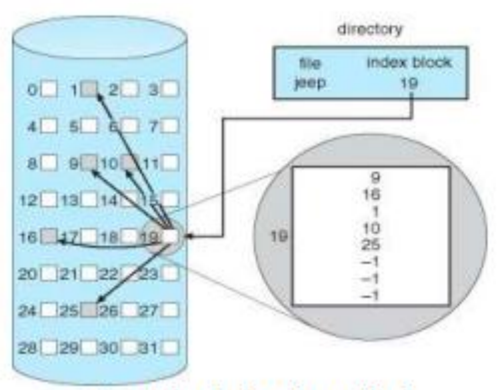


Fig.:- Indexed allocation method

6.3 Directory Structure

What is a Directory?

- Think of it as a folder containing "cards" (entries) describing each file/subdirectory.
- Here's what the directory does:
  - Maps file names to physical info (location, size, attributes).
  - Supports organizing, searching, renaming, traversing, creating/deleting files or subdirectories.

Operations on Directories

- Searching for a file by name.
- Creating new files/directories.
- Deleting files.
- Listing contents of a directory.
- Renaming files.
- Traversing the whole file system.

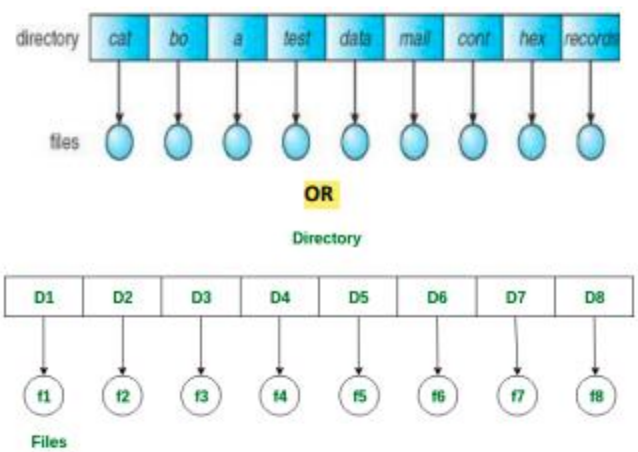
Why Structure Directories?

- **Efficiency:** Faster file search.
- **Naming:** Users can use meaningful names; can avoid conflicts.
- **Grouping:** Related files can be sorted together (music, docs, code).

Types of Directory Implementations

A. Single-Level Directory

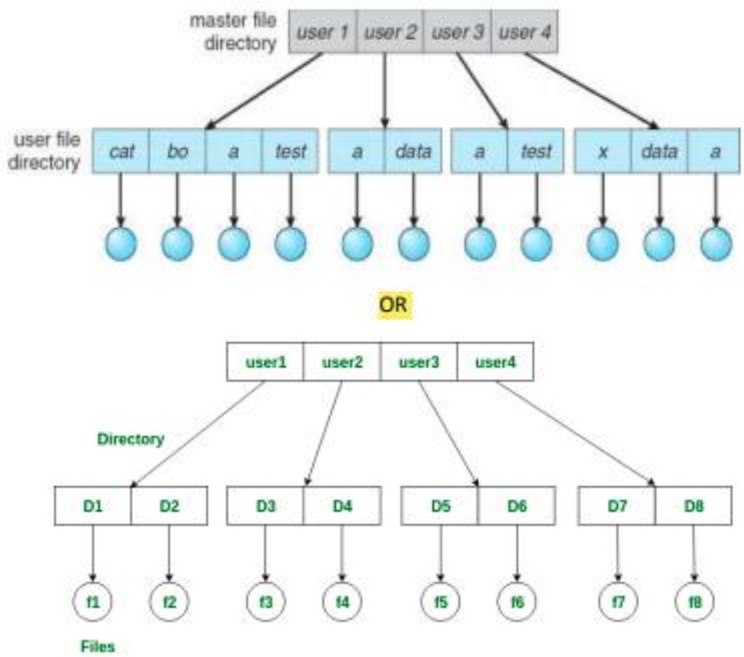
- All files (for all users) kept in the same directory.
- **Pros:** Simple, easy to search small collections.
- **Cons:**
    - Name clash: only one file called “notes.txt” possible.
    - No logical grouping—everything mixed together.
    - Gets messy with many users or files.



B. Two-Level Directory

- Adds a **Master File Directory (MFD)** for the system and a **User File Directory (UFD)** for each user.
- Each username has their own directory containing their files.
- **Pros:**
    - Each user can have files with same names.
    - Easier lookup—search only within user’s UFD.
- **Cons:**

- No sharing/files across users by default.
- Users themselves can’t organize files into further categories (no subdirectories).



C. Tree-Structured Directory

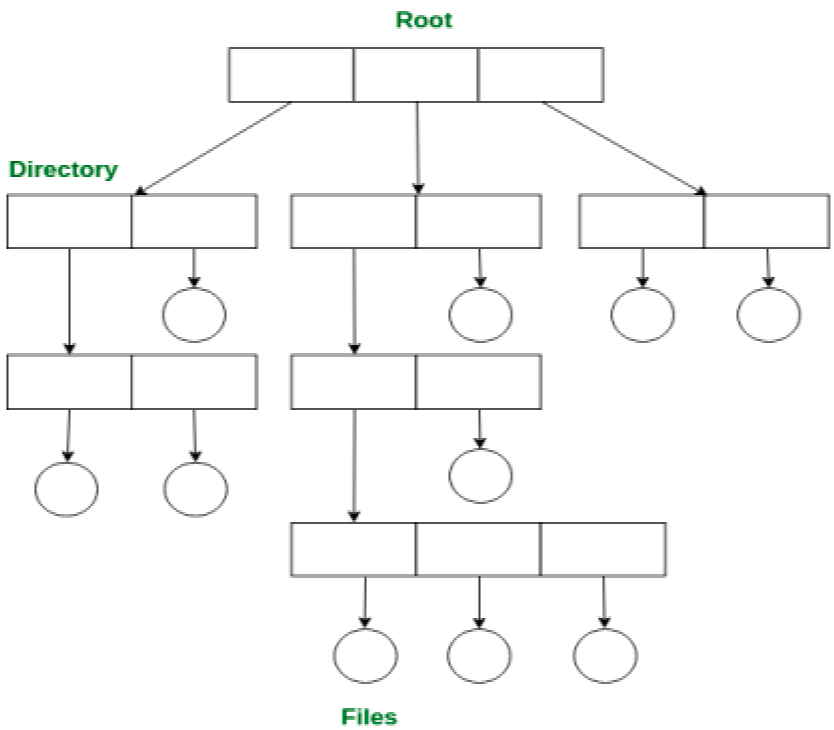
- Hierarchical (like folders in Windows/Unix).
- Subdirectories allowed at any depth—directories can contain files or more directories.
- Root directory at the top.
- Full **paths** indicate file location.
  - Absolute: /users/ansh/docs/project.txt
  - Relative: docs/project.txt (when already in /users/ansh/)
- Supports both absolute and relative addressing.

Pros:

- Scalable for large, complex systems.
- Convenient grouping—music, code, docs, all separated.
- Supports same file name in different subdirectories.

Cons:

- A file can only live in one place (unless using advanced systems like links).
- May require traversing several levels for deep files.



Comparison Table

Feature	Single-Level	Two-Level	Tree-Structured
Hierarchy	None	2 Levels	Many Levels
Name Collisions	Yes (easy)	No (per user)	Very rare (can be grouped)
Grouping	None	By user	By user & subdirectory
File Sharing	No	No	No (directly), possible with links
Search Efficiency	Poor (large)	Good	Good (with full path)
Scalability	Not scalable	Limited	Highly scalable