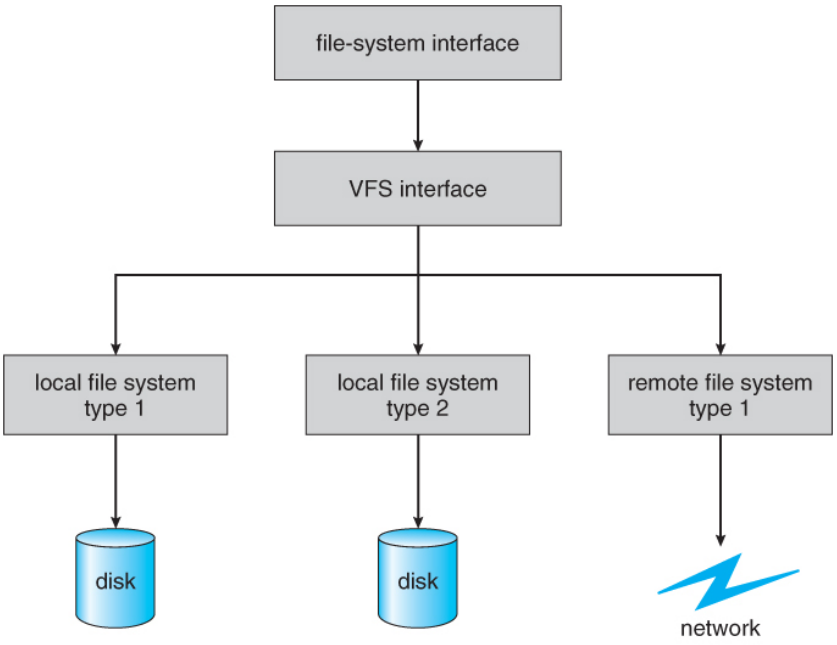
File System in Operating System (OS)

What is a File System?

- A **file system** is a method used by an operating system to **store, organize, manage, and retrieve data** on storage devices like hard disks, SSDs, USB drives, etc.
- It provides a logical way to name, store, access, and protect files so users and programs can easily work with data.

Objectives of a File System

- Efficient **storage and retrieval** of data
- Provide **data sharing** among users and applications
- Ensure **data protection and security**
- Support **backup and recovery**
- Maintain **consistency and reliability** of stored data



Basic Concepts of File System

1. **File:**

   A file is a collection of related data stored on secondary storage with a name, size, type, and permissions.

2. **Directory:**

   A directory is a special file that stores information about other files and directories, forming a hierarchical structure.

3. **Path:**

   A path specifies the location of a file in the directory hierarchy (absolute or relative path).

File Attributes (Metadata):

Each file has attributes such as:

- File name
- File type
- Size
- Location
- Owner
- Permissions (read, write, execute)
- Creation and modification time

File Operations:

The operating system provides several file operations:

- **Create** – Create a new file
- **Open** – Access a file
- **Read** – Read data from a file
- **Write** – Write data into a file
- **Close** – Close an opened file
- **Delete** – Remove a file
- **Rename** – Change file name

File Access Methods:

1. **Sequential Access:**

   Data is accessed in order from beginning to end (e.g., text files).

2. **Direct (Random) Access:**

   Data can be accessed at any position directly (e.g., databases).

3. **Indexed Access:**

   Uses an index to locate data blocks efficiently.

Directory Structures:

- **Single-level Directory** – One directory for all users.
- **Two-level Directory** – Separate directory for each user.
- **Tree-structured Directory** – Hierarchical structure (most common).
- **Acyclic Graph Directory** – Allows shared files/directories.
- **General Graph Directory** – Supports cycles (complex to manage).

File System Structure:

A typical file system contains:

- **Boot Control Block** – Used to boot the OS.
- **Super Block** – Stores file system metadata.
- **File Control Block (FCB) / Inode** – Stores file attributes.
- **Data Blocks** – Actual file contents.

Disk Space Allocation Methods:

1. **Contiguous Allocation:**

   Files are stored in consecutive blocks (fast but causes fragmentation).

2. **Linked Allocation:**

   Each block points to the next block (no fragmentation, slower access).

3. **Indexed Allocation:**

   Uses an index block for pointers (efficient and flexible).

Free Space Management:

- Bit Vector
- Linked List
- Grouping
- Counting

File Protection and Security:

- Access control using **permissions.**
- User authentication.
- Encryption
- Backup mechanisms.

Types of File Systems:

- **FAT / FAT32** – Simple, used in USB drives.
- **NTFS** – Windows file system with security features.
- **ext3 / ext4** – Linux file systems.
- **APFS** – Apple file system.

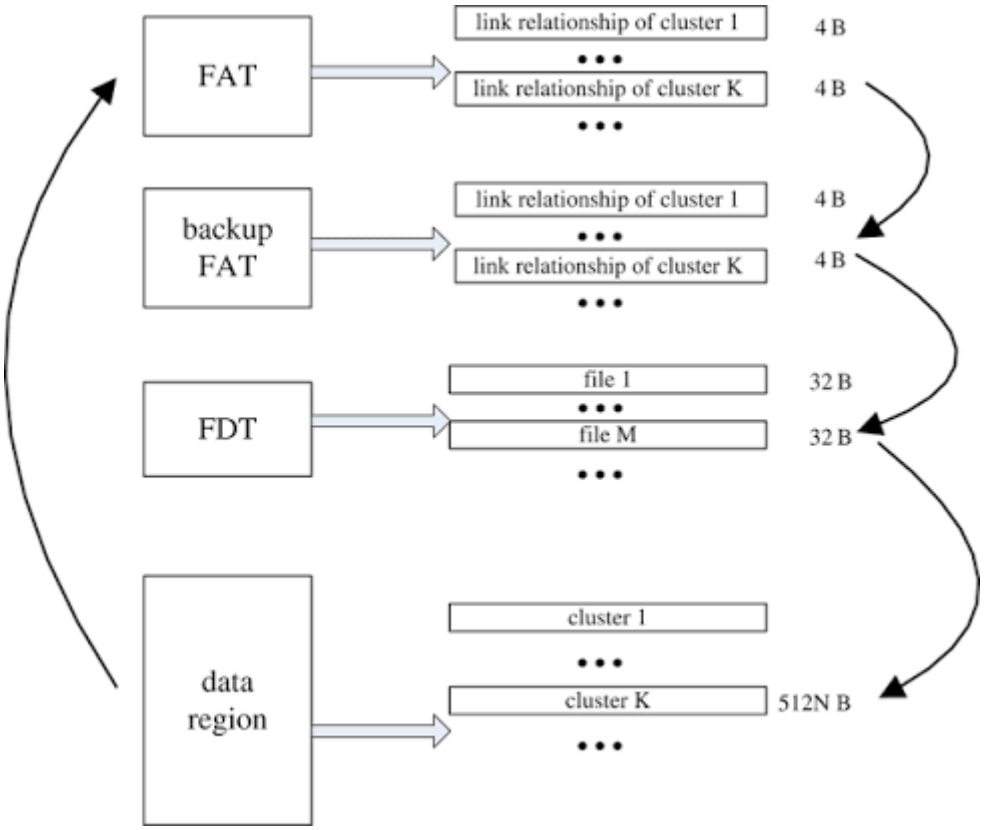
Advantages of File System:

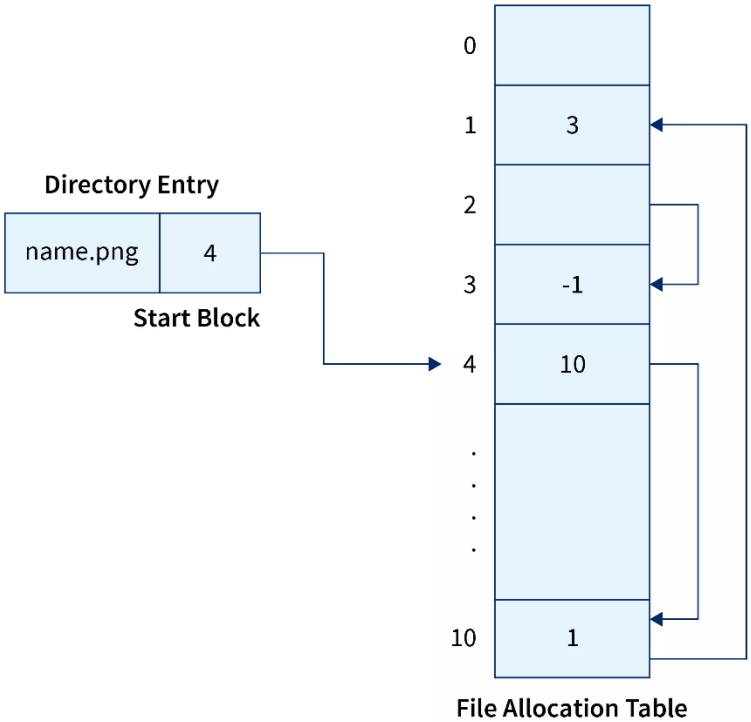
- Organized data storage.
- Easy data access.
- Data security and integrity.
- Efficient memory usage.

FAT (FILE ALLOCATION TABLE)

What is FAT?

- **FAT (File Allocation Table)** is a **simple and widely used file system** developed by Microsoft.
- It uses a **table (FAT)** to keep track of the allocation status of disk blocks (clusters) for files.





Basic Idea of FAT

- The disk is divided into **clusters** (group of sectors).
- A **File Allocation Table** stores entries for each cluster.
- Each FAT entry tells:
    - Which cluster comes **next** in a file
    - Or whether the cluster is **free**
    - Or marks **end of file (EOF)**

Structure of FAT File System

1. **Boot Sector**
    - Contains information needed to boot the system
    - Stores file system parameters
2. **File Allocation Table (FAT)**
    - Central data structure
    - Keeps track of cluster usage and links
3. **Root Directory**
    - Stores file and directory metadata
    - File name, size, starting cluster, attributes
4. **Data Area**
    - Actual storage area where file contents are stored

Working of FAT (Step-by-Step)

1. When a file is created, free clusters are allocated.
2. The starting cluster number is stored in the directory entry.
3. FAT entries form a **linked list** of clusters for that file.
4. EOF marker indicates the end of the file.
5. During file access, OS follows the cluster chain using FAT.

FAT Entry Values

- **0** → Free cluster
- **Cluster number** → Next cluster in file
- **EOF marker** → End of file
- **Bad cluster marker** → Unusable cluster

File Operations in FAT

- **Create** – Allocate clusters and update FAT
- **Read** – Follow FAT chain sequentially
- **Write** – Add clusters and update FAT entries
- **Delete** – Mark clusters as free in FAT

Types of FAT

Type	Cluster Size	Max Volume Size	Usage
FAT12	12-bit	~32 MB	Floppy disks
FAT16	16-bit	~2 GB	Old Windows systems
FAT32	32-bit	~2 TB	USB drives, memory cards

Advantages of FAT

- Simple and easy to implement
- Low overhead
- High compatibility across OS
- Suitable for small storage devices

Disadvantages of FAT

- No file-level security
- No journaling (risk of corruption)
- Inefficient for large files
- Fragmentation problem
- Slower access for large files (sequential traversal)

Applications of FAT

- USB flash drives
- Memory cards
- Embedded systems
- Legacy systems

FAT vs Modern File Systems (Quick View)

Feature	FAT	NTFS / ext4
Security	No	Yes
Journaling	No	Yes
Large file support	Limited	High
Reliability	Low	High

EXT4 (Fourth Extended File System)

What is EXT4?

- **EXT4 (Fourth Extended File System)** is a **modern Linux file system**, designed as an improvement over ext3 and ext2.
- It supports **large files, high performance, reliability, and journaling**, making it the **default file system for most Linux distributions**.

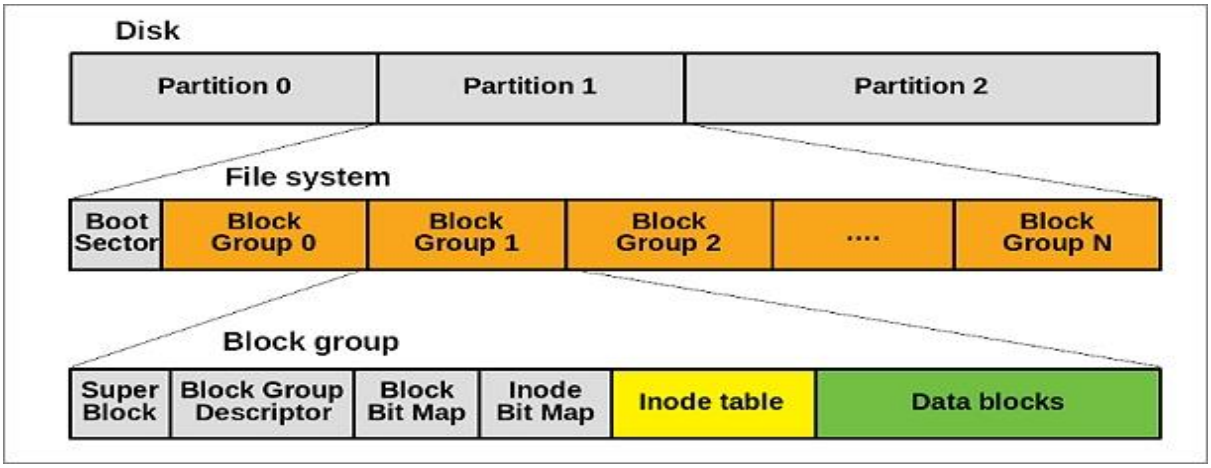
Evolution of EXT File Systems

File System	Features
ext2	No journaling
ext3	Journaling added
ext4	Extents, large file support, faster performance

Key Features of EXT4

- **Journaling support** (metadata and optional data journaling)
- **Extents-based allocation** instead of block mapping
- Supports **very large files and volumes**
- **Delayed allocation** for better performance
- **Faster file system check (fsck)**
- Backward compatible with ext3 and ext2

Structure of EXT4 File System:



1. Disk

- The physical storage device is divided into **partitions**:
  - **Partition 0**, **Partition 1**, **Partition 2**
- Each partition can have its own file system (e.g., EXT4, FAT, NTFS).

2. File System Layout

Inside a partition, the file system is divided into **Block Groups**:

- **Boot Sector**
  - Contains code and information to boot the system (if the partition is bootable).
- **Block Groups (0, 1, 2, ..., N)**
  - The partition is split into multiple **block groups** for better performance and organization.

- This helps reduce disk seek time because metadata and data are stored close together.

3. Block Group Structure

Each **Block Group** contains:

1. **Superblock**
   - Stores overall information about the file system: size, number of blocks, number of inodes, block size, etc.
   - Contains file system information
   - Critical for mounting the file system.
2. **Block Group Descriptor**
   - Contains details about the block group: locations of bitmaps, inode tables, and free space.
3. **Block Bitmap**
   - A map showing which data blocks in this group are free or used.
   - tracks used/free data blocks
4. **Inode Bitmap**
   - A map showing which inodes are free or in use.
   - tracks used/free inodes.
5. **Inode Table**
   - Stores **inodes** (data structures containing file metadata: size, ownership, timestamps, and pointers to data blocks).
   - stores file metadata
6. **Data Blocks**
   - Where the **actual file contents** are stored(store actual file contents).

Working

- When you save a file:
  - An **inode** is created in the inode table containing metadata and pointers to data blocks.
  - The **data blocks** store the file's actual content.
  - The block and inode bitmaps are updated to reflect the allocation.
- The **block group design** ensures that inodes and data blocks for related files are kept close together, improving access speed.

Extents in EXT4:

- An **extent** is a group of **contiguous blocks**
- Reduces fragmentation

- Improves large file performance
- Stores (starting block + length) instead of individual block pointers

Journaling in EXT4:

- Maintains a **journal** to record changes before committing them
- Helps in **crash recovery**
- Journaling modes:
  - **Journal mode** – Data and metadata journaled
  - **Ordered mode** (default) – Metadata journaled, data written first
  - **Writeback mode** – Metadata journaled only

File Operations in EXT4:

- **Create** – Allocate inode and extents
- **Read** – Direct access using extents
- **Write** – Delayed allocation improves block placement
- **Delete** – Deallocate inode and blocks

Limits Supported by EXT4:

Feature	Limit
Max file size	~16 TB
Max volume size	~1 EB
Max filename length	255 characters
Block size	1 KB – 4 KB

Advantages of EXT4:

- High performance
- Strong reliability
- Fast recovery after crashes
- Supports large storage devices
- Efficient space utilization

Disadvantages of EXT4:

- Limited native support in Windows
- No built-in data compression
- No snapshot feature (needs LVM)

INODES

What is an Inode?

An **inode (Index Node)** is a **data structure used by Unix/Linux file systems** (such as ext2, ext3, ext4) to **store metadata about a file**. It does **not store the file name or file data**, but contains all information needed to locate and manage the file.

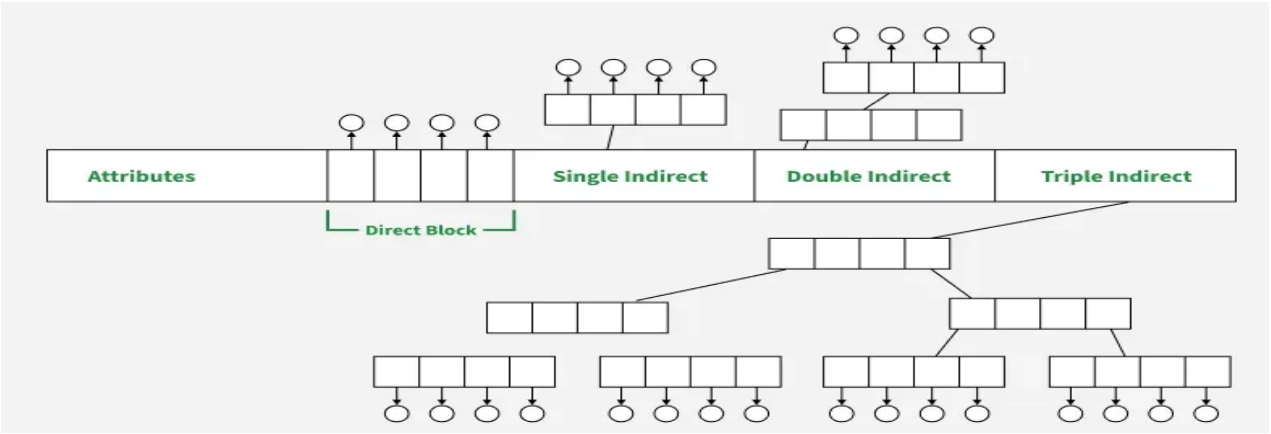
Information Stored in an Inode

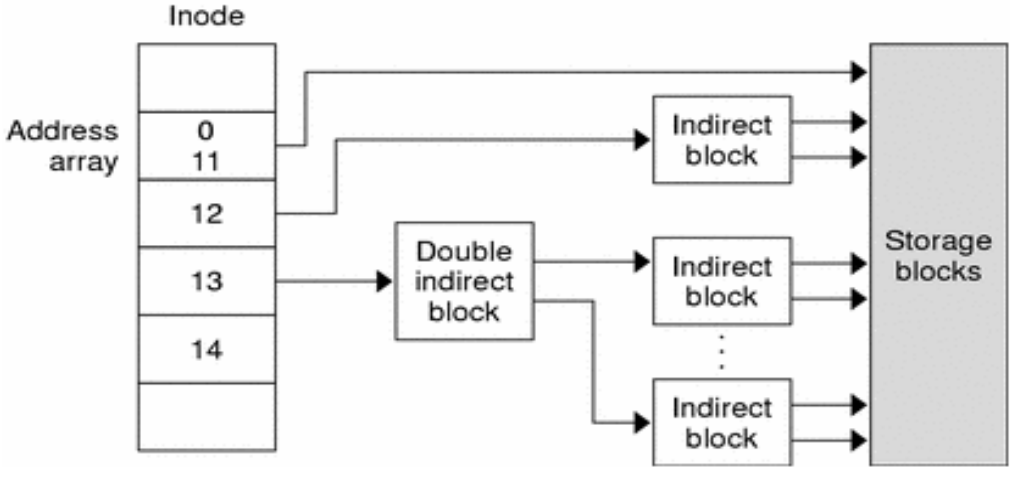
Each inode contains:

- File type (regular file, directory, etc.)
- File size
- Owner (User ID, Group ID)
- Permissions (read, write, execute)
- Timestamps (creation, modification, access)
- Number of links (hard links)
- Pointers to data blocks (or extents in ext4)

What is NOT Stored in an Inode?

- File name
- Directory name





a)Direct Block

- **0 to 11 → Direct pointers**
  - Each points **directly to a storage block** containing file data.
  - Which point directly to the actual data blocks that hold the file's content.
  - First-level pointers that allow the file system to quickly access the data
  - If the file is small (fits within these 12 blocks), no indirect pointers are needed.

b)Single Indirect Block

. **12 → Single indirect pointer**

- Points to an **Indirect block**.
- The indirect block contains addresses of another data blocks.
- Useful when file is larger than the space covered by direct pointers.

c) Double indirect pointer

13 → Double indirect pointer

- Points to a **Double indirect block**, which contains addresses of **Indirect blocks**, and those indirect blocks point to actual data blocks.
- When a file becomes even larger, the **double indirect block** is used
- Expands capacity further.

d) Double indirect blocks

- **14 → Triple indirect pointer**
  - Points to a block that contains addresses of **Double indirect blocks**, which then lead to **Indirect blocks**, and finally to data blocks.
  - Used for **very large files**.
  - This multi-layered indirection system allows the file system to address massive files, even when the file size exceeds the typical limits of direct and indirect addressing.

Working of Inode:

1. When a file is created, an inode is allocated and filled with metadata about the file.
2. The filesystem stores the inode in a table (called the inode table), which is a list of all inodes in the filesystem.
3. The directory entry holds the mapping between the filename and the inode number.

4. When you access a file, the system retrieves the inode by its number, loads the metadata, and uses the pointers to access the data blocks where the file's content resides.

- **Small file** : Uses direct blocks (fast access).
- **Medium file** : Uses single indirect pointer when direct blocks are full.
- **Large file** : Uses double indirect pointer for even more blocks.
- **Huge file** : Uses triple indirect pointer.

Example

Suppose a file is created named report.txt:

- Inode number: **34567**
- Inode metadata:
  - Size: 2048 bytes
  - Permissions: rw- r-- r--
  - Owner: UID 1000
  - Data block pointers: [#500, #501]

Role of Inodes in File Access

1. User requests a file by name.
2. OS searches the directory to get the **inode number**.
3. Inode is accessed using inode number.
4. Inode provides pointers to data blocks.
5. File data is read or written.

Inodes in EXT4

- EXT4 uses **extents instead of block pointers**
- An extent stores:
  - Starting block number
  - Length (number of contiguous blocks)
- Reduces fragmentation and improves performance

Inode Number

- Each file has a **unique inode number**
- Inode numbers are assigned when the file is created
- Can be viewed using**:**

  ls -i filename

Hard Link and Inodes

- Hard links share the **same inode number**

- Link count in inode increases
- File data is deleted only when link count becomes zero

Inode Table

- All inodes are stored in an **inode table**
- Size of inode table is fixed at file system creation
- Limits the **maximum number of files** in a file system

Advantages of Inodes

- Efficient file access
- Supports large files
- Enables hard links
- Improves file system reliability

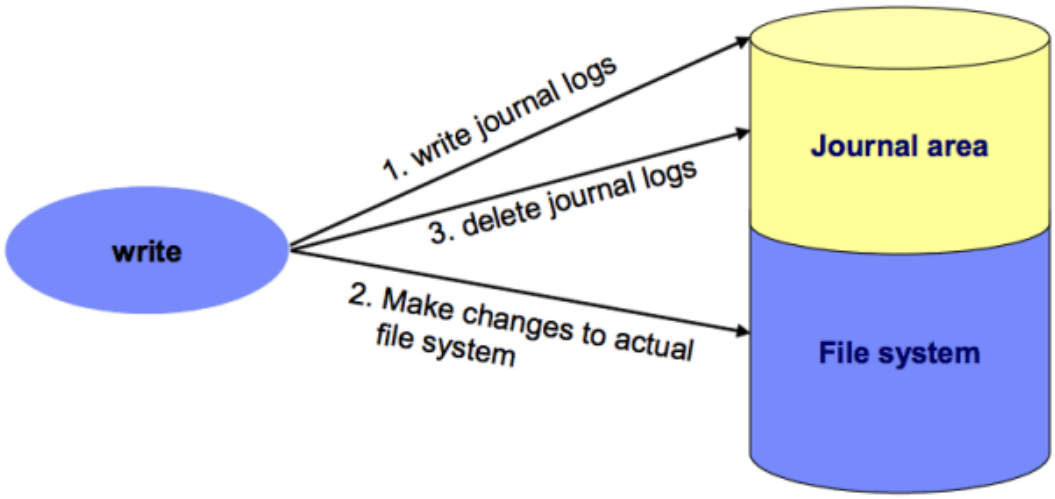
Limitations of Inodes

- Fixed number of inodes (inode exhaustion)
- Cannot store file names directly

JOURNALING

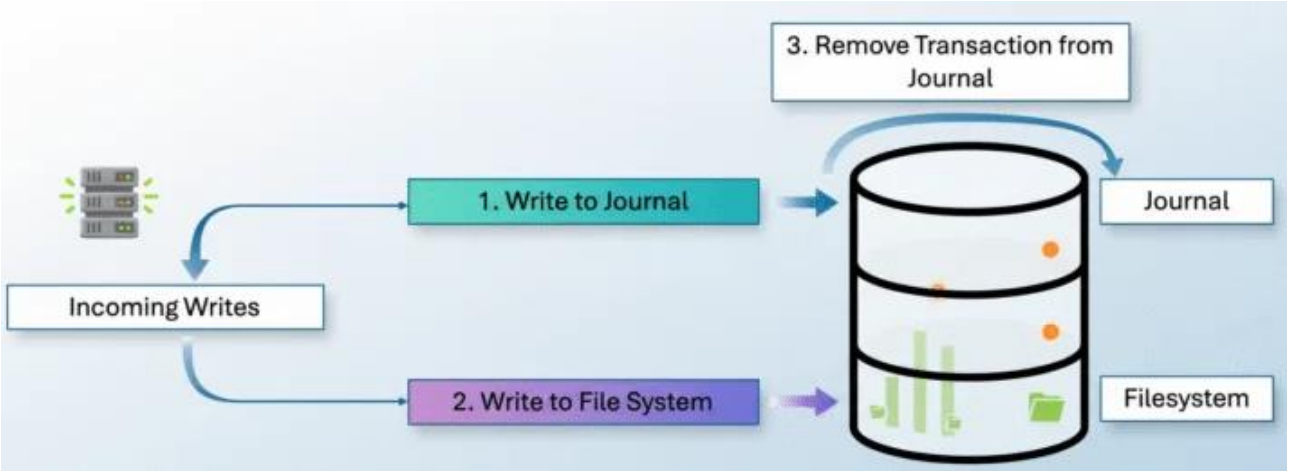
What is Journaling?

- **Journaling** is a **file system technique** used to **maintain consistency and recover quickly after crashes**.
- Before making actual changes to the disk, the file system **records intended changes in a journal (log)**.



Why Journaling is Needed

- Prevents **file system corruption**
- Ensures **data consistency**
- Enables **fast recovery** after power failure or system crash
- Reduces time for **fsck (file system check)**



1. Journal Log Area

- Reserved continuous space on disk
- Stores **log entries (transactions)**
- Works in **circular (ring buffer) manner**

2. Transaction:

A **transaction** is a group of related changes written to the journal as a single unit.

Each transaction contains:

- **Transaction Header**
- **Log Records**
- **Commit Record**

3. Transaction Header:

Contains:

- Transaction ID
- Sequence number
- Start marker
- Timestamp

☞ Identifies the beginning of a transaction.

4. Log Records:

Contain the **actual changes** to be made:

- Metadata updates (inode, bitmap changes).
- Data blocks (only in full data journaling).

5. Commit Record:

- Marks the transaction as **complete.**
- Ensures atomicity (all-or-nothing).

☞ Only committed transactions are replayed after crash.

6. Checkpoint Area:

- After journaled data is written to main disk
- Transaction is marked as **checkpointed**
- Space is freed for reuse

7. Journal Superblock:

Stores:

- Journal size
- Start and end pointers
- Journal status
- Sequence number

How Journaling Works (Step-by-Step):

1. File system writes changes to the **journal**.
2. Journal entry is marked as **committed**.
3. Actual changes are written to disk.
4. On crash:
    - If entry is committed → redo operation.
    - If not committed → discard operation.

Types of Journaling:

1. **Metadata Journaling:**
    - Only metadata is logged.
    - Faster, less disk usage.
    - Used by ext3/ext4 (default).
2. **Full Data Journaling:**
    - Both metadata and file data are logged.
    - Very safe but slower.
3. **Ordered Journaling:**
    - Metadata is journaled.
    - Data blocks are written before metadata commit.
    - Default mode in ext4.
4. **Writeback Journaling:**
    - Only metadata journaled.
    - Fastest but least safe.

Journaling Modes in EXT4:

Mode	Description
Journal	Metadata + data
Ordered	Metadata only (safe & fast)
Writeback	Metadata only (fast, less safe)

Advantages of Journaling:

- Fast crash recovery.
- Improved reliability.
- Prevents inconsistent metadata.
- Reduces boot time after crash.

Disadvantages of Journaling:

- Extra disk overhead.
- Slight performance cost.
- Not protection against hardware failure.

File Systems Using Journaling:

- **ext3, ext4** (Linux)
- **NTFS** (Windows)
- **XFS**
- **JFS**
- **ReiserFS**

Journaling vs Non-Journaling

Feature	Journaling FS	Non-Journaling FS
Crash recovery	Fast	Slow
Data safety	High	Low
Performance	Slightly lower	Higher
Example	ext4	FAT