Unit 5 : File Management (6 Hrs)

File management is a critical component of an operating system, responsible for organizing, storing, retrieving, and managing files and directories on secondary storage. It provides a logical view of data to users, abstracting the complexities of physical storage.

5.1. File Overview

A **file** is a named collection of related information that is recorded on secondary storage (e.g., hard drives, SSDs). From a user's perspective, a file is the smallest logical unit of storage.

5.1.1. File Naming

Files are given names for user convenience. File naming conventions vary across operating systems:

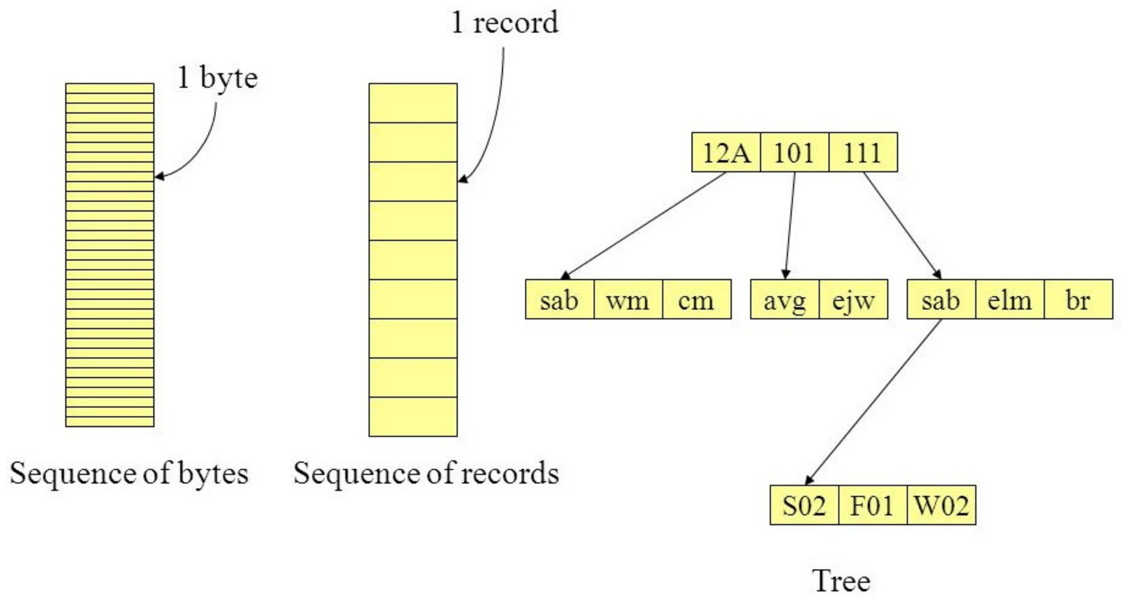
- **Case Sensitivity:**
  - **Use Case/Example:** In Linux, `report.txt` and `Report.txt` are treated as two distinct files. In Windows, these names refer to the same file.
- **Length Limits:**
  - **Use Case/Example:** Older systems like MS-DOS enforced an "8.3" format (8 characters for the name, 3 for the extension), so a file named `my_annual_report.document` would have to be truncated to something like `myannual.doc`. Modern systems (e.g., Windows NTFS, Linux ext4) support hundreds of characters.
- **Allowed Characters:**
  - **Use Case/Example:** Characters like `/` (Linux/UNIX path separator), `\` (Windows path separator), `*` (wildcard), `?` (wildcard), `<>`, `|`, `"` are typically forbidden in filenames because they have special meaning to the shell or file system.
- **Extensions:**
  - **Use Case/Example:** `.txt` indicates a plain text file, `.pdf` indicates an Adobe Portable Document Format file, `.exe` indicates a Windows executable program, `.jpg` indicates a JPEG image.

5.1.2. File Structure

File structure refers to how a file is organized internally. Common structures include:

- **Unstructured Sequence of Bytes:** The most common model, used by UNIX/Linux and Windows. The OS views the file as a flat sequence of bytes, and the interpretation is left to the application.
  - **Advantages:** Simple, flexible, no overhead from the OS enforcing structure.
  - **Disadvantages:** Applications must define and manage their own internal structure.
  - **Use Case/Example:** A `.mp3` audio file. The OS simply stores its bytes. It's the music player application that understands the MP3 header, audio frames, and how to decode them. Similarly, a `.docx` file is just a sequence of bytes to the OS; Microsoft Word interprets its complex XML-based internal structure.

- **Simple Record Structure:** A file is a sequence of fixed-length or variable-length records. Used in older mainframe systems for database-like applications.
    - **Use Case/Example:** A file containing customer records, where each record has a fixed format like `CustomerID (5 chars) | Name (20 chars) | Balance (10 chars)`. An application could easily read the 5th record by calculating its offset.
- **Complex Structures:** Trees, indices, or databases, often managed by a database management system (DBMS) built on top of the OS's unstructured byte stream.
    - **Use Case/Example:** A MySQL database file (e.g., `.ibd` file for InnoDB). The OS sees it as a large binary file, but the MySQL server software internally manages it as a complex B-tree or other database structure to store tables, indices, and rows.



5.1.3. File Types

File types categorize files based on their content and purpose. The OS or applications use file types to determine how to handle a file.

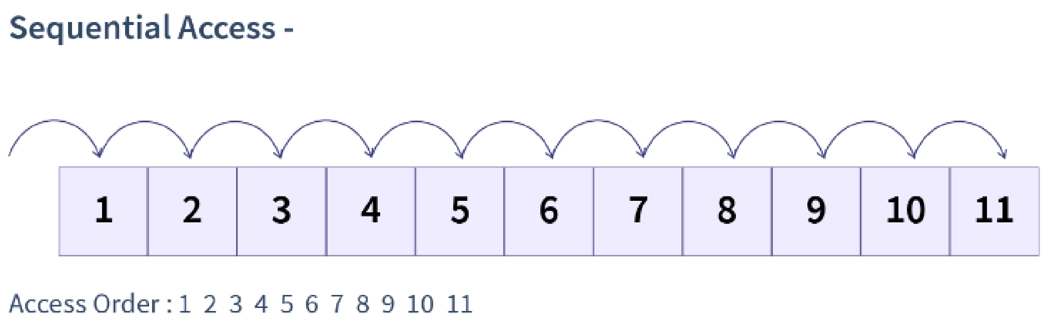
- **Regular Files:** Contain user information.
    - **Use Case/Example:** `document.docx` (text/document), `photo.jpg` (image), `program.exe` (executable), `log.txt` (data file).
- **Directories:** System files that store information about other files and directories.
    - **Use Case/Example:** `/home/user/documents/` is a directory that contains other files (`report.pdf`, `notes.txt`) and potentially subdirectories (`projects/`).
- **Character Special Files:** Used for character-oriented I/O devices.
    - **Use Case/Example:** In Linux, `/dev/ttyS0` for a serial port or `/dev/urandom` for a source of random data. Data is read/written character by character.
- **Block Special Files:** Used for block-oriented I/O devices.

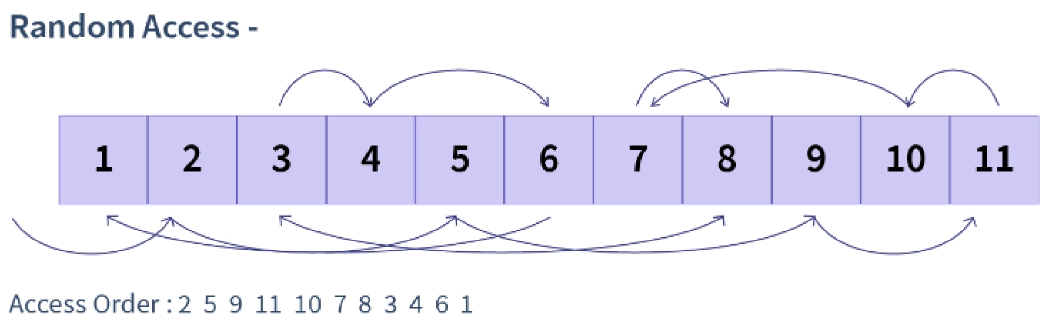
    - **Use Case/Example:** In Linux, `/dev/sda` for a hard disk drive or `/dev/cdrom` for a CD-ROM drive. Data is read/written in fixed-size blocks.
- **Pipes/FIFOs:** Used for inter-process communication.
    - **Use Case/Example:** A named pipe in Linux, created with `mkfifo my_pipe`. Process A writes to `my_pipe`, and Process B reads from `my_pipe` to communicate.
- **Sockets:** Used for network communication.
    - **Use Case/Example:** A Unix domain socket file like `/var/run/mysqld/mysqld.sock` which allows a local client to communicate with a MySQL server process.

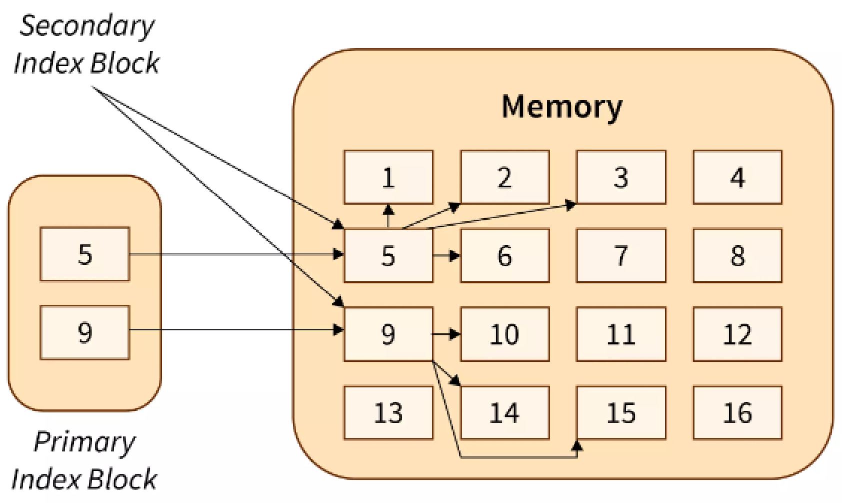
5.1.4. File Access

File access methods define how information within a file can be read or written.

- **Sequential Access:** Data is read/written in order, from the beginning to the end.
    - **Advantage:** Simple to implement.
    - **Disadvantage:** Inefficient for random lookups.
    - **Use Case/Example:** Reading a `.log` file line by line, streaming an audio file, or processing transactions from a batch file. To read the 100th line, all 99 preceding lines must be read first.
- **Random Access (Direct Access):** Data can be read/written at any arbitrary position within the file. Requires seeking to a specific byte offset. Essential for databases.
    - **Advantage:** Fast access to specific records/data.
    - **Disadvantage:** More complex to implement.
    - **Use Case/Example:** Accessing a specific record in a database file by its record ID. An application can directly jump to the calculated byte offset of that record without reading previous records. For instance, in a student database, directly retrieving student ID 500 without scanning all records from 1 to 499.
- **Indexed Sequential Access:** A hybrid approach. Files are organized sequentially, but an index is maintained to allow faster jumps to specific sections, after which sequential reading can continue.
    - **Use Case/Example:** An old ISAM (Indexed Sequential Access Method) file system for large data records. One might use an index to quickly jump to the start of records for "Students with last names starting with 'S'", then read sequentially from there.







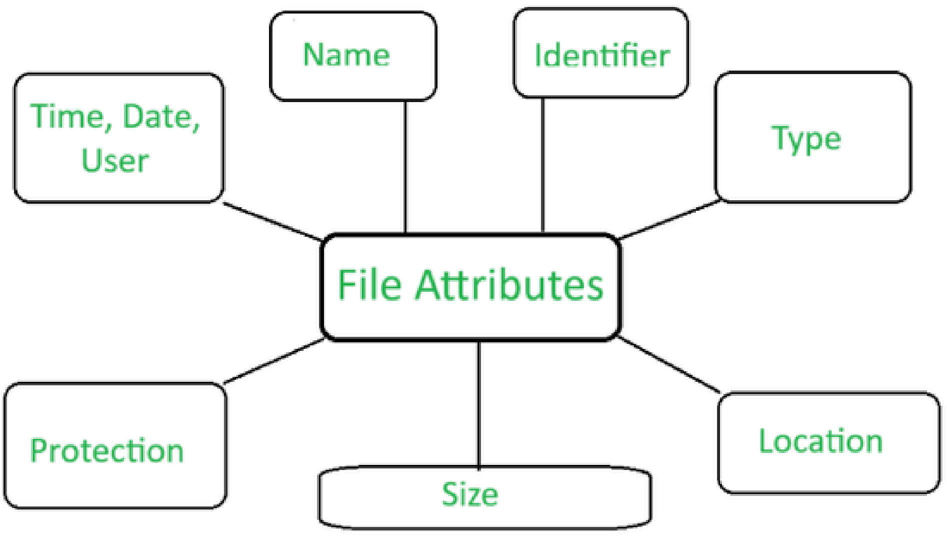
Indexed Sequential Access Of File

5.1.5. File Attributes

File attributes are metadata associated with a file, providing information about it.

- **Name:** The symbolic name for the file.
  - **Use Case/Example:** `meeting_notes.txt` .
- **Type:** Indicates the file's format or purpose.

    - **Use Case/Example:** `.pdf` for Portable Document Format, allowing the OS to open it with a PDF viewer.
- **Location:** Pointer to the file's starting location on disk.
    - **Use Case/Example:** For a file, this might be a pointer to its first data block or its inode on the disk.
- **Size:** Current size of the file in bytes, blocks, or records.
    - **Use Case/Example:** A `report.pdf` file might have a size attribute of `2,548,760 bytes`.
- **Protection (Permissions):** Controls who can read, write, or execute the file.
    - **Use Case/Example:** In Linux, permissions like `rwx` (read, write, execute) for the owner, `r--` for the group, and `---` for others (`-rwxrw----`). A user trying to write to a file without write permission will receive an "Access Denied" error.
- **Time, Date, User Identification:** Time of creation, last modification, last access, and the user ID of the creator/owner.
    - **Use Case/Example:** A file created by `userA` on `2024-01-15 10:00:00` and last modified by `userB` on `2024-06-10 15:30:00`. This helps in version control or system auditing.
- **Read-only/Archive/Hidden/System flags:** Specific flags for file behavior.
    - **Use Case/Example:** Setting the "Read-only" flag on `important_config.ini` prevents accidental modification. The "Hidden" flag on system files (e.g., in Windows) prevents them from cluttering user views.

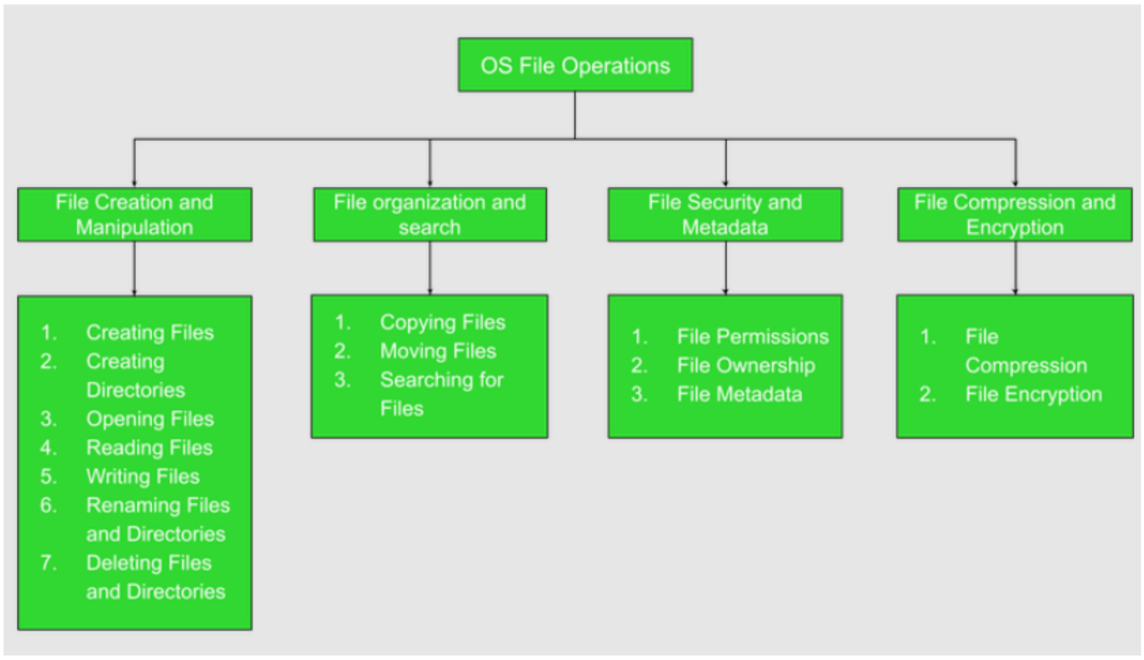


5.1.6. File Operations

Operating systems provide system calls for users and applications to perform operations on files:

- **Create:** Creates a new file, allocates space, and adds an entry to the directory.

    - **Use Case/Example:** `touch newfile.txt` (Linux) or `New-Item newfile.txt` (PowerShell) creates an empty file.
- **Delete:** Removes a file, frees its space, and removes its directory entry.
    - **Use Case/Example:** `rm old_report.pdf` (Linux) or deleting a file via File Explorer (Windows).
- **Open:** Prepares a file for use. The OS brings file attributes and metadata into memory and returns a file handle/descriptor.
    - **Use Case/Example:** A text editor `opens` a `.txt` file before allowing editing. The `open()` system call in C returns a file descriptor (an integer).
- **Close:** Releases resources associated with an open file, writes back metadata if dirty.
    - **Use Case/Example:** When a program finishes writing to a file, it calls `close()` to ensure all buffered data is written to disk and resources are released.
- **Read:** Reads data from a file into a buffer.
    - **Use Case/Example:** A web browser `reads` an HTML file to display a webpage. The `read()` system call takes a file descriptor, a buffer, and a size.
- **Write:** Writes data from a buffer into a file.
    - **Use Case/Example:** A word processor `writes` the contents of a document to a file when saving. The `write()` system call takes a file descriptor, a buffer, and a size.
- **Seek:** Changes the current read/write position (file pointer) within a file (for random access).
    - **Use Case/Example:** In a video player, seeking to the middle of a movie involves a `seek` operation to a specific byte offset in the video file. The `lseek()` system call in UNIX.
- **Truncate:** Deletes the contents of a file but keeps its attributes.
    - **Use Case/Example:** `truncate -s 0 large_log.txt` (Linux) empties a log file without deleting the file itself.
- **Rename:** Changes the name of a file.
    - **Use Case/Example:** `mv old_name.txt new_name.txt` (Linux) or "Rename" option in File Explorer (Windows).

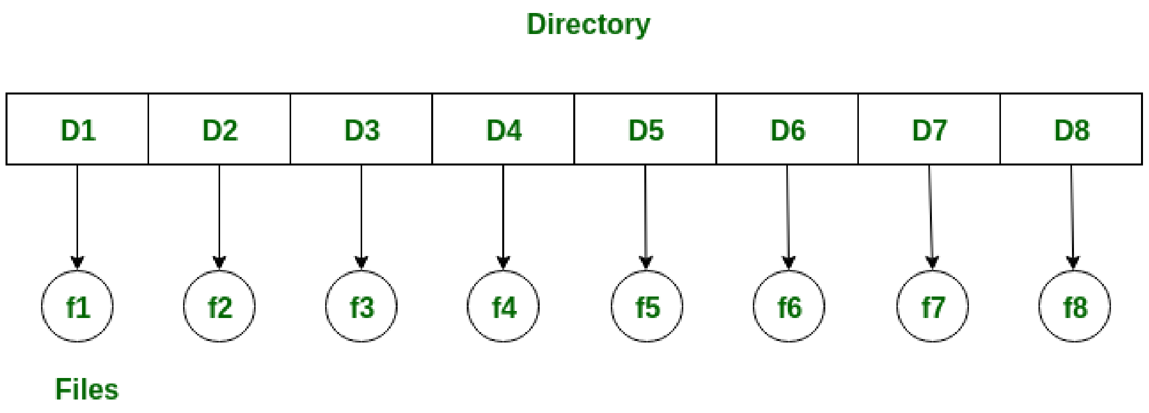


5.1.7. Directory Systems

Directory systems organize files in a hierarchical structure, making it easier for users to locate and manage them.

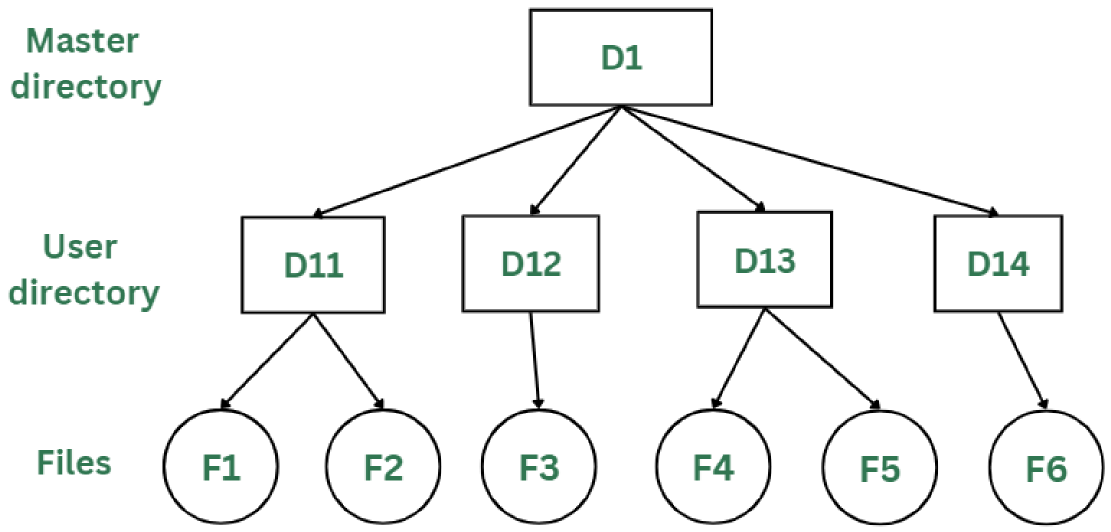
a) Single-Level Directory System:

- **Structure:** All files are contained in a single directory.
- **Advantages:** Simple to implement.
- **Disadvantages:**
  - **Naming Conflicts:** All files must have unique names across the entire system.
  - **Scalability Issues:** Hard to manage a large number of files.
  - **No User Isolation:** No way to group files by user or project.
- **Use Case/Example:** Early personal computer systems like CP/M, where all files were in the "root" or main directory of a floppy disk. If user A created `report.txt`, user B could not create another file also named `report.txt`.



b) Two-Level Directory System:

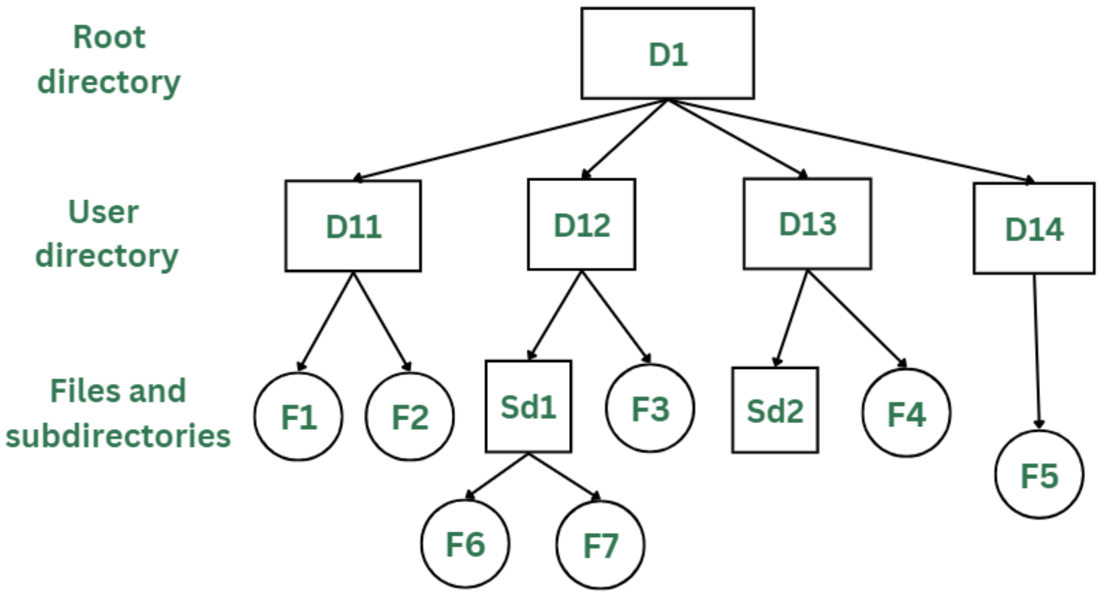
- **Structure:** Each user has a separate directory (User File Directory - UFD) within a master file directory (MFD).
- **Advantages:**
  - **Solves Naming Conflicts:** Users can have files with the same name.
  - **User Isolation:** Each user has a private space.
- **Disadvantages:**
  - **No Sharing:** Difficult to share files between users without copying.
  - Still limited for grouping files within a single user's space.
- **Use Case/Example:** Used in some early multi-user mainframes. If User1 has `report.txt` in their UFD, and User2 also has `report.txt` in their UFD, there's no conflict. However, User1 cannot easily access User2's `report.txt`.



c) Hierarchical (Tree-Structured) Directory System:

- **Structure:** The most common model today. Directories can contain both files and other subdirectories, forming a tree-like structure. There is a single root directory.
- **Advantages:**
  - **No Naming Conflicts:** Files with the same name can exist in different directories.
  - **Logical Grouping:** Files can be organized intuitively by project, type, or user.
  - **Easy Sharing:** Files can be shared by referring to their pathnames.
- **Disadvantages:**
  - More complex to implement.
  - Searching for a file can take longer if the path is deep.
- **Use Case/Example:** All modern operating systems like UNIX/Linux (`/home/user/documents/projects/os_project/`) and Windows

(`C:\Users\Username\Documents\Projects\OS_Project\`). This allows for highly organized file storage.



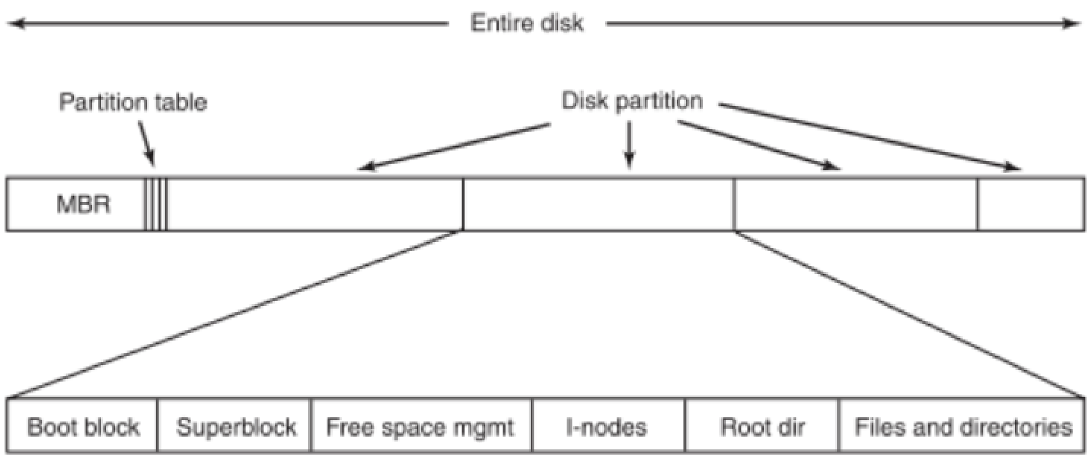
5.1.8. File System Layout

The file system layout describes how the file system is organized on a disk. A disk is divided into sectors, which are grouped into blocks.

- **Partition Table:** At the beginning of the disk, defines how the disk is divided into partitions.
    - **Use Case/Example:** On a typical hard drive, the partition table might specify that 100 GB is allocated for the Windows C: drive, and another 200 GB for a Linux partition.
- **Boot Block:** Contains a small program to bootstrap the operating system.
    - **Use Case/Example:** When a computer starts, the BIOS/UEFI reads the boot block from the active partition to load the initial bootloader, which then loads the full OS.
- **Superblock:** Contains key information about the file system (e.g., total blocks, free block count, inode table size, block size, file system state). Crucial for mounting and checking the file system.
    - **Use Case/Example:** When Linux mounts an `ext4` partition, it reads the superblock to understand the file system's overall structure and status. If the superblock is corrupted, the file system cannot be read.
- **Free Block Management Area:** Stores information about free (available) disk blocks.
    - **Use Case/Example:** A bitmap indicating which blocks are available for new file data.
- **Inode List (for UNIX-like systems):** A table of inodes, each describing a file (metadata, pointers to data blocks).
    - **Use Case/Example:** When the OS needs to access a file in an `ext4` file system, it first finds its inode number, then uses that number to locate

the inode in the inode list to get the file's attributes and data block pointers.

- **Root Directory:** The top-level directory of the file system.
  - **Use Case/Example:** In Linux, the `/` directory; in Windows, the root of a drive like `C:\`.
- **Data Blocks:** The actual storage area for file data and directory contents.
  - **Use Case/Example:** The blocks where the actual content of `my_document.txt` or `photo.jpg` is stored.



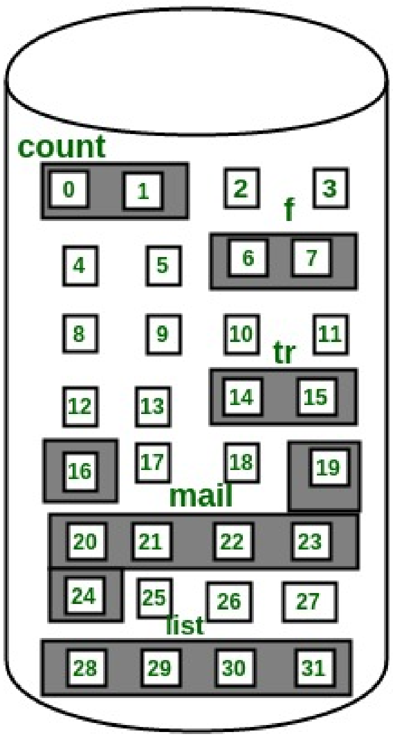
5.2. Implementing Files (Disk Space Allocation)

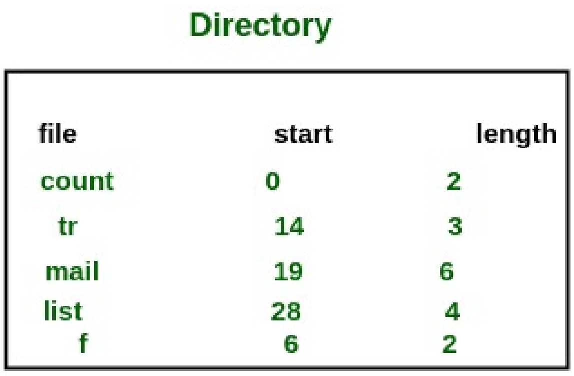
This section details how file data is stored on disk and how the OS manages the disk blocks allocated to a file.

5.2.1. Contiguous Allocation

- **Explanation:** Each file occupies a set of contiguous (adjacent) blocks on the disk. The directory entry for the file stores the starting block address and the length (number of blocks) of the file.
- **Advantages:**
  - **Simple:** Easy to implement.
  - **Excellent Read Performance:** Minimal head movement for sequential access; fast random access (start_block + offset).
- **Disadvantages:**
  - **External Fragmentation:** As files are created and deleted, holes of varying sizes appear in memory, making it difficult to find a contiguous block large enough for new files.
  - **Difficulty with File Growth:** If a file needs to grow, it might require relocating the entire file if adjacent space is not available.
  - **Pre-allocation:** Requires knowing the maximum file size in advance or frequent defragmentation.
- **Numerical Example/Use Case:**
  - Consider a disk with blocks 0-9.
  - File A (3 blocks): allocated blocks 0, 1, 2. Directory entry: `File A: Start=0, Length=3`.

    - File B (2 blocks): allocated blocks 3, 4. Directory entry: `File B: Start=3, Length=2`.
    - Later, File A is deleted, freeing blocks 0, 1, 2.
    - Now, if a new File C (4 blocks) arrives, it cannot be allocated contiguously because the largest free contiguous space is 3 blocks (0,1,2). Even if blocks 5, 6, 7, 8 are free, they might not be consecutive if File B is still there, leading to external fragmentation.



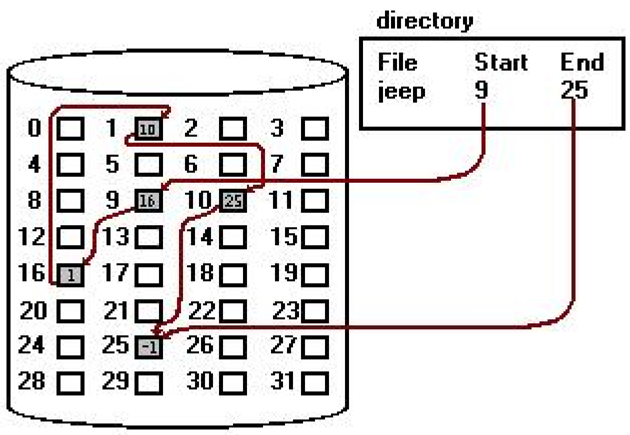

Directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

5.2.2. Linked List Allocation

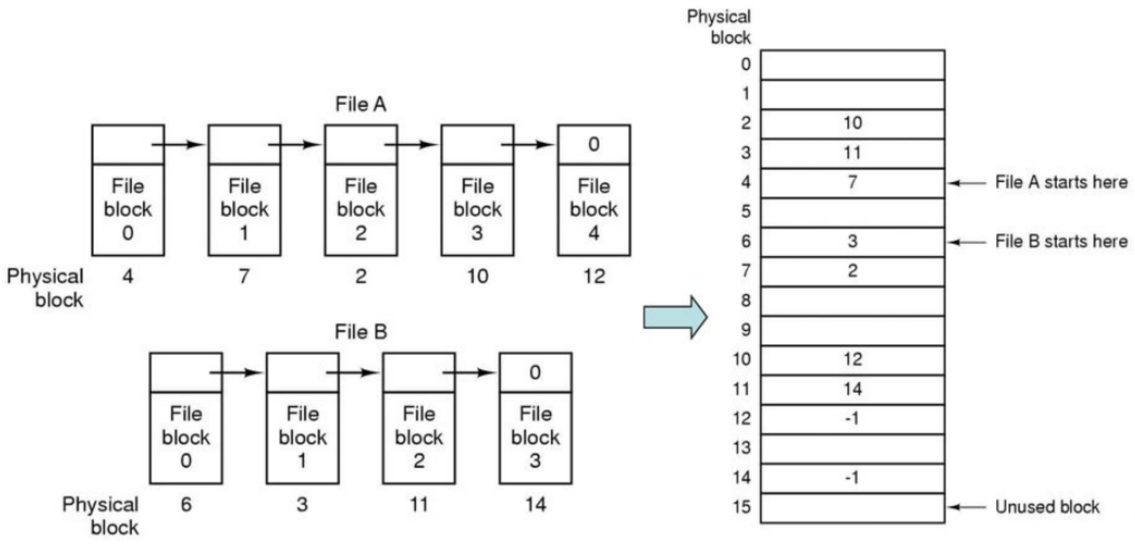
- **Explanation:** Each file is stored as a linked list of disk blocks. Each block contains a pointer to the next block in the file. The directory entry for the file stores the starting block address and optionally the ending block address.
- **Advantages:**
    - **No External Fragmentation:** Files can use any available block; holes are automatically linked into the free list.
    - **Flexible File Growth:** Files can grow dynamically by adding more blocks anywhere on the disk.
- **Disadvantages:**
    - **Poor Random Access Performance:** To reach a specific block (e.g., block N), the system must traverse $N-1$ blocks sequentially.
    - **Space Overhead:** Each block needs to store a pointer to the next block, reducing the actual data storage per block.
    - **Reliability Issues:** A single broken pointer can lead to the loss of the rest of the file.
- **Numerical Example/Use Case:**
    - Assume blocks are 512 bytes. A pointer takes 4 bytes. So, 508 bytes are for data.

- File X (3 blocks): Directory entry points to Block 10.
- Block 10 contains data + pointer to Block 25.
- Block 25 contains data + pointer to Block 5.
- Block 5 contains data + pointer to NULL (end of file).
- To read the data in Block 5, the system must first read Block 10 to get the pointer to Block 25, then read Block 25 to get the pointer to Block 5. This involves multiple disk I/Os for random access.



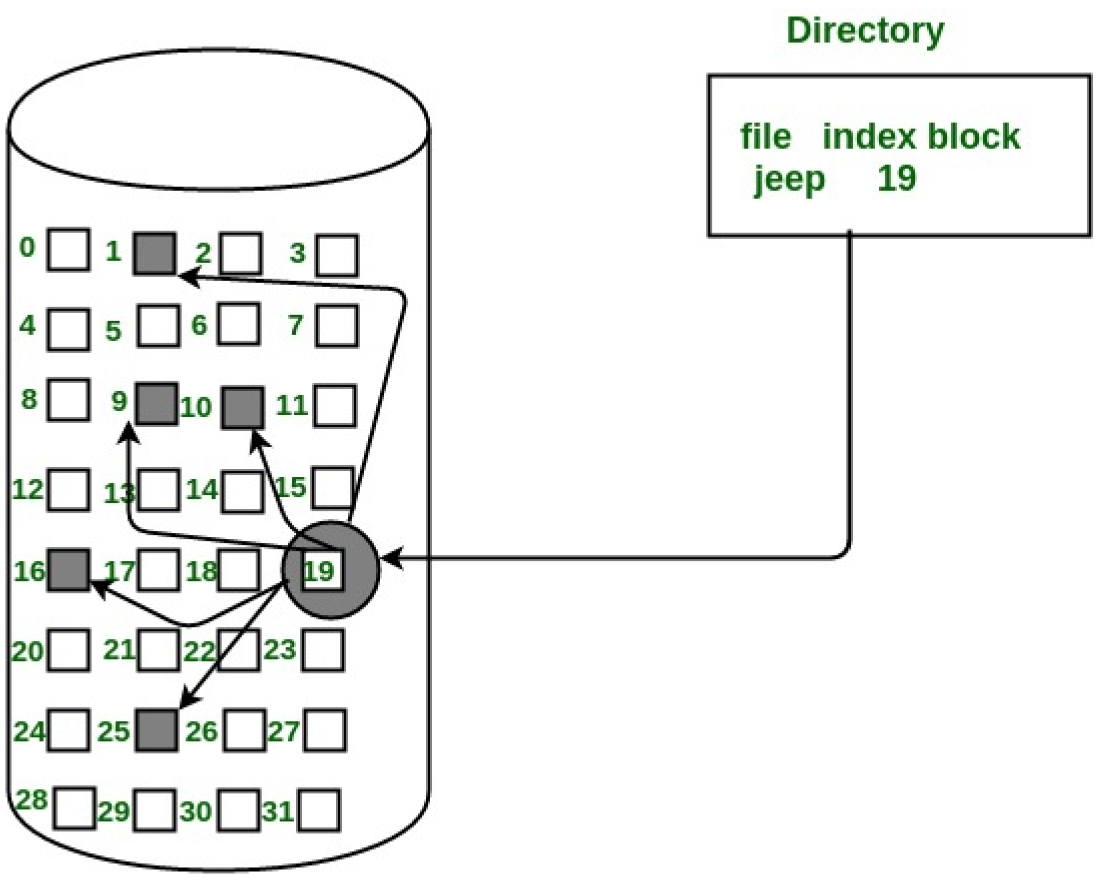
5.2.3. Linked List Allocation using Table in Memory (FAT - File Allocation Table)

- **Explanation:** A variation of linked list allocation where the pointers are stored in a separate table in memory (the File Allocation Table) rather than within the data blocks themselves. Each entry in the FAT corresponds to a disk block.
- **How it works:**
  - The directory entry stores the starting block number of the file.
  - To find the next block, the system consults the FAT entry corresponding to the current block number. The FAT entry contains the block number of the next block.
  - A special value (e.g., -1) indicates the end of a file.
- **Advantages:**
  - **Improved Random Access:** While still not as fast as contiguous, traversing the FAT in memory is much faster than reading blocks from disk.
  - **No External Fragmentation:** Same as simple linked list.
  - **No Pointer Overhead in Data Blocks:** Data blocks are entirely used for data.
- **Disadvantages:**
  - **FAT Size:** The entire FAT must be in memory for efficient operation, which can be very large for large disks with small block sizes.
  - Still sequential traversal for finding arbitrary blocks, though faster than disk traversal.



5.2.4. Inodes (Indexed Allocation)

- **Explanation:** Used predominantly in UNIX-like file systems. Each file has an **inode** (index node) which is a small data structure containing all the metadata about the file (owner, permissions, size, creation time, etc.) and, critically, pointers to the file's data blocks on disk.
- **Structure of an Inode (simplified):**
  - File attributes (owner, permissions, size, timestamps).
  - **Direct Pointers:** A fixed number of pointers (e.g., 12) point directly to the first few data blocks of the file.
  - **Single Indirect Pointer:** Points to a disk block that contains *additional direct pointers* to data blocks.
  - **Double Indirect Pointer:** Points to a disk block that contains pointers to *single indirect blocks*.
  - **Triple Indirect Pointer:** Points to a disk block that contains pointers to *double indirect blocks*.
- **Advantages:**
  - **Efficient Random Access:** For small files, direct pointers provide very fast access. For larger files, a few disk reads can locate any block.
  - **No External Fragmentation:** Files can be scattered across the disk.
  - **Supports Large Files:** Indirect pointers allow very large files to be stored.
  - **Flexible Growth:** Files can grow dynamically.
- **Disadvantages:**
  - **Overhead:** For very small files, the inode itself might be larger than the file data.
  - **Multiple Disk Accesses:** For very large files, multiple disk reads are required to traverse indirect pointers to find a data block (e.g., 3 reads for a triple indirect block before the actual data block).



5.3. Directory Operations, Path Names, Directory Implementation, Shared Files

5.3.1. Directory Operations

Directories are special files that store information about other files and directories. Operations on directories include:

- **Search:** Find an entry for a file.
  - **Use Case/Example:** When a program opens `/home/user/document.txt`, the OS searches the `/` directory for `home`, then `home` for `user`, then `user` for `document.txt`.
- **Create File:** Add a new file entry.
  - **Use Case/Example:** When saving a new file in a word processor, a new entry is added to the relevant directory.
- **Delete File:** Remove a file entry.
  - **Use Case/Example:** Using the `rm` command in Linux removes the directory entry for a file.
- **List Directory:** Display the names of files and subdirectories.
  - **Use Case/Example:** The `ls` command in Linux or `dir` command in Windows lists the contents of a directory.
- **Rename File:** Change a file's name.
  - **Use Case/Example:** Changing `old_report.docx` to `final_report.docx`. This updates the name attribute in the directory entry.

- **Traverse Directory:** Navigate through the directory hierarchy.
    - **Use Case/Example:** The `cd` command in Linux or `cd` in Windows changes the current working directory.
- **Create Directory:** Create a new subdirectory.
    - **Use Case/Example:** `mkdir my_project` creates a new subdirectory named `my_project`.
- **Delete Directory:** Remove an empty subdirectory.
    - **Use Case/Example:** `rmdir empty_folder` deletes an empty directory.

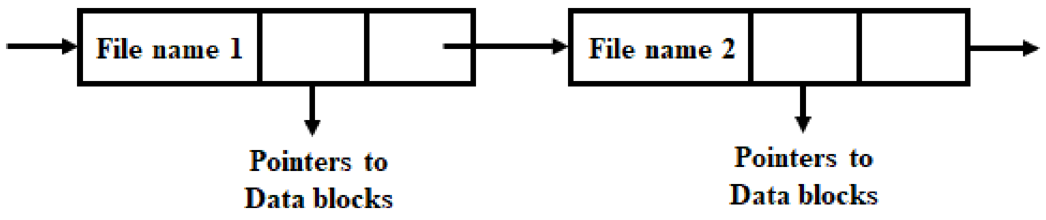
5.3.2. Path Names

- **Path Name:** A string that uniquely identifies a file or directory within a hierarchical file system.
- **Absolute Path Name:** Starts from the root directory and specifies the full path to the file.
    - **Use Case/Example:** In UNIX, `/home/user/documents/report.txt`. In Windows, `C:\Users\User\Documents\report.txt`. This path can be used from anywhere in the file system.
- **Relative Path Name:** Specifies the path relative to the current working directory.
    - **Use Case/Example:** If the current working directory is `/home/user/`, then `documents/report.txt` refers to the same file as the absolute path above. This is convenient for accessing files within the current working area.

5.3.3. Directory Implementation

Directory implementation refers to how the operating system internally stores and manages information about files and directories — essentially how it organizes file metadata and structure. Directories are essentially lists of files and their attributes. Their implementation varies:

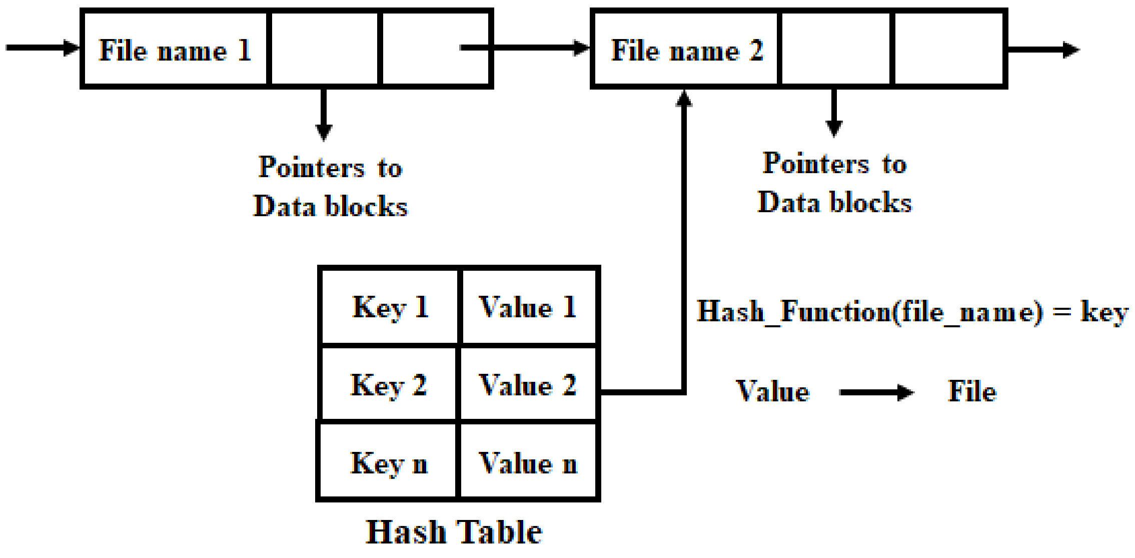
- **Linear List:** A simple list of file names and pointers to their respective data blocks or inodes.
    - **Advantages:** Simple to implement.
    - **Disadvantages:** Slow search for large directories. Deletion can be complex.
    - **Use Case/Example:** In early file systems or very small, simple directories. To find a file, the OS scans this list from beginning to end.



Directory Implementation Using Linear List

- **Hash Table:** Uses a hash function to map file names to entries in a hash table, which then point to the file information.

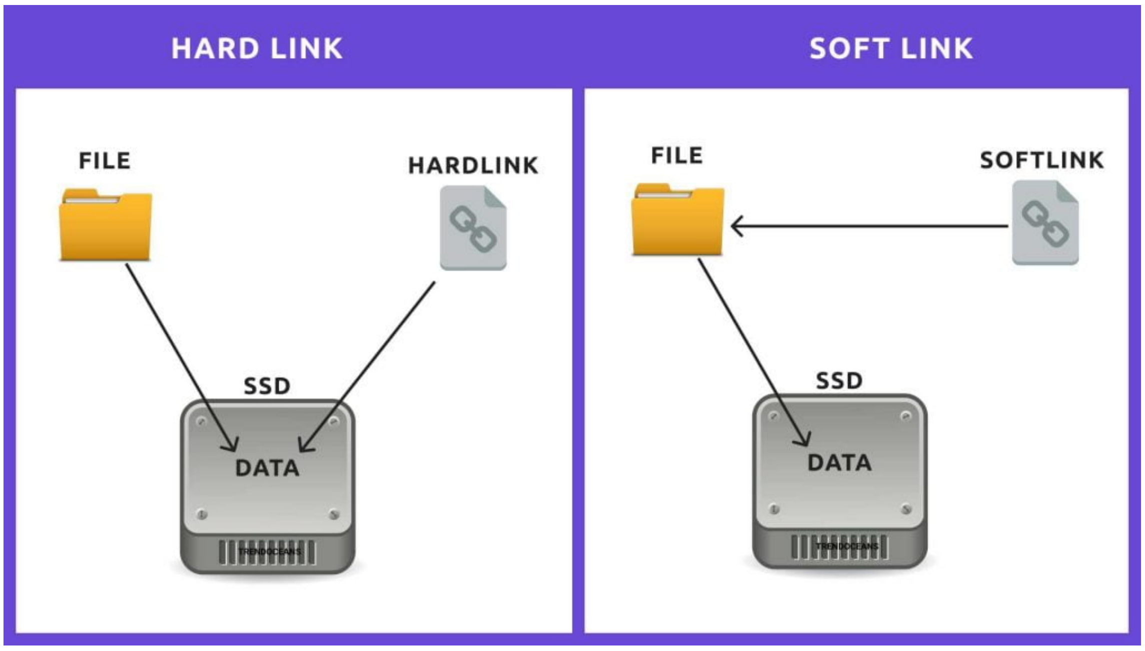
  - **Advantages:** Very fast search and insertion on average.
  - **Disadvantages:** Collisions must be handled. Can be complex to resize.
  - **Use Case/Example:** Some modern file systems might use hashing for large directories to speed up lookups. When searching for "document.txt", its name is hashed to quickly jump to a likely location in the table.



Directory Implementation Using Hash Table

- **B-Tree/B+Tree:** Used for very large directories in some high-performance file systems.
  - **Advantages:** Efficient searching, insertion, and deletion for large datasets.
  - **Disadvantages:** More complex to implement.
  - **Use Case/Example:** Modern file systems like NTFS (Windows) and HFS+ (macOS) use B-trees for very large directories to ensure scalable performance, especially for directories with thousands of files.

5.3.4. Shared Files



1. Hard Link (Default)

- Hard links are used in Linux/Unix systems when you want multiple file names to refer to the same data on disk — without duplicating the content.
- A **hard link** is another name for the same file.
- Both point to the **same inode** (same data on disk).
- If the original file is deleted, the data remains accessible through the hard link.

Syntax:

```
ln source_file link_name
```

Example:

```
ln file1.txt file1_hardlink.txt
```

- Now both files refer to the same data.
- Changes in one reflect in the other.

---

2. Symbolic (Soft) Link

- A **symbolic link** is like a shortcut or pointer.
- It points to the **path** of the original file.
- If the original file is deleted, the symlink becomes **broken**.

Syntax:

```
ln -s source_file link_name
```

Example:

```
ln -s file1.txt file1_symlink.txt
```

- `file1_symlink.txt` points to `file1.txt`.
- One can see it's a symlink using `ls -l`.

- **Comparison Table: Hard Links vs. Symbolic Links**

Feature	Hard Link	Symbolic (Soft) Link
Concept	Multiple directory entries pointing to the same inode/file data.	A file containing the pathname of another file/directory.
Type	Not a separate file type; just another name for the original file.	A distinct file type (a special file).
Deletion Impact	Deleting a hard link only removes one directory entry; file data is removed when the last hard link is gone.	Deleting the original file breaks the symbolic link (dangling pointer).
Cross File Systems	Cannot span across different file systems.	Can span across different file systems.
Link to Directory	Cannot link to directories (to prevent infinite loops).	Can link to directories.
Original File Needed	If original file is moved/deleted, other hard links still work.	If original file is moved/deleted, the link breaks.
Inode Number	Same inode number as the original file.	Different inode number from the original file.

5.4. Free Space Management

Managing free disk blocks is essential for allocating space to new files and reclaiming space from deleted files. Modern OS and file systems primarily use Bitmaps + Extents.

5.4.1. Bitmaps (Bit Vector) [Widely Used Today]

- **Explanation:** A bitmap (or bit vector) is a bit array where each bit corresponds to a disk block. A bit value of `1` typically indicates that the block is allocated (in use), and `0` indicates that it is free.
- **Advantages:**
  - **Simple:** Easy to implement and understand.
  - **Efficient for finding contiguous blocks:** Quickly identify sequences of `0`s for contiguous allocation.
  - **Easy to update:** Changing a bit is fast.
- **Disadvantages:**
  - **Space Overhead:** The bitmap itself can be large for very large disks (e.g., a 1 TB disk with 4 KB blocks needs a 32 MB bitmap, which is (1024 GB * 1024 MB/GB * 1024 KB/MB) / 4 KB/block = 268,435,456 blocks. This would require 268,435,456 bits / 8 bits/byte = 33,554,432 bytes, or 32 MB).

- Must be kept in memory for efficient access, or at least portions of it.

The given instance of disk blocks on the disk in Figure 1 (where green blocks are allocated) can be represented by a bitmap of 16 bits as: **1111000111111001** .

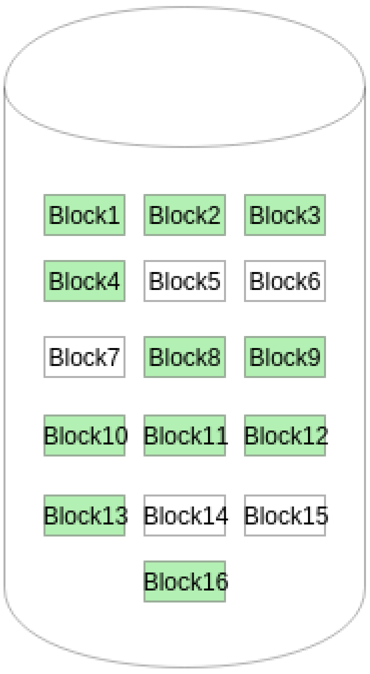


Figure - 1

5.4.2. Linked List (Free List) [Mostly Outdated]

- **Explanation:** All free disk blocks are linked together into a list. The first free block contains a pointer to the second free block, which points to the third, and so on. The OS only needs to store a pointer to the head of this free list.
- **Advantages:**
  - **No Space Overhead in Memory:** Only the pointer to the first free block needs to be in memory. The list itself is distributed across the free blocks on disk.
  - **Simple for allocation:** Just take the head of the list.
- **Disadvantages:**
  - **Poor Performance for Finding Contiguous Blocks:** Difficult and slow to find contiguous blocks for contiguous file allocation.
  - **Reliability Issues:** A single lost pointer can lead to the loss of a large number of free blocks.
  - **Sequential Traversal:** Finding a specific number of free blocks might require traversing a long list on disk.

In Figure-2, the free space list head points to Block 5 which points to Block 6, the next free block and so on. The last free block would contain a null pointer indicating the end of free list.

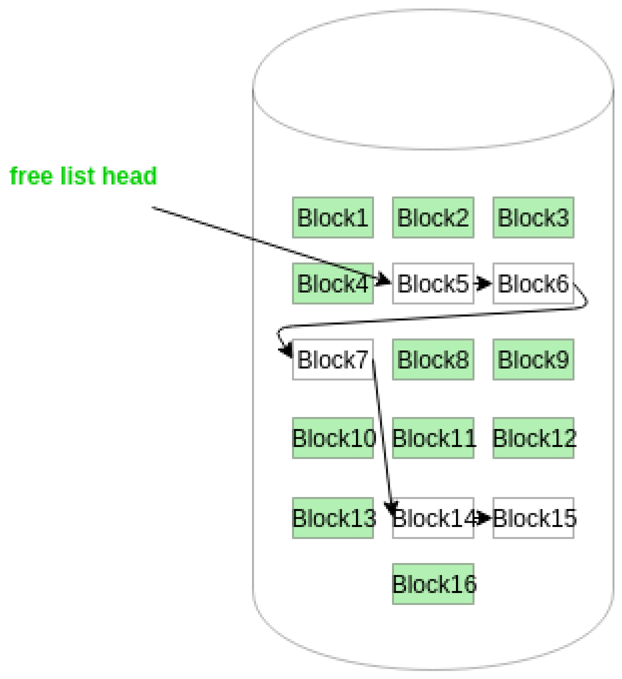

free list head
Block1
Block2
Block3
Block4
Block5
Block6
Block7
Block8
Block9
Block10
Block11
Block12
Block13
Block14
Block15
Block16

Figure - 2