UNIT-V

FILE SYSTEM INTERFACE

FILE CONCEPT

A file is a named collection of related information that is recorded on secondary storage. · A file is the smallest allotment of logical secondary storage (i.e.) data cannot be written to secondary storage unless they are within a file.

· Files represent programs and data. Data files may be numeric, alphanumeric or binary. · The information in a file is defined by its creator.

· Different types of information may be stored in a file such as source or executable programs, numeric or text data, photos, music, video and so on.

A file structure depends on its type:

· **Text file** is a sequence of characters organized into lines.

· **Source file** is a sequence of functions, each of which is further organized as declarations followed by executable statements.

· **Executable file** is a series of code sections that the loader can bring into memory and execute.

File Attributes

A file is referred to by its name. The following are the list of file attributes: · **Name**: The symbolic file name is the only information kept in human-readable form. · **Identifier**: This is a unique number that identifies the file within the file system. It is the non-human-readable name for the file.

· **Type**: This information is needed for systems that support different types of files. · **Location**: It is a pointer to a device and to the location of the file on that device. · **Size**: The current size of the file and the maximum size are included in this attribute. · **Protection**: It is access-control information determines who can do reading, writing, executing and so on.

· **Time, Date and User Identification**: This information kept for creation, last modification and last use. These data can be useful for protection, security and usage monitoring.

Directory structure keeps information about all files. It resides on secondary storage. · A directory entry consists of the file's name and its unique identifier. · The identifier locates the other file attributes.

· It may take more than a kilobyte to record this information for each file.

File Operations

There are 6 basic operations performed on file and corresponding System call are: 1. **Creating a file: create( )** system call is used to create a file. To create a file Operating system checks whether there is enough space in the system. If yes, then a new entry will be made in the directory structure.

2. **Repositioning within a file**. The directory is searched for the appropriate entry and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a **File-Seek**.

3. **Deleting a file: delete( )** system call is used to delete a file. To deete a file, we search the directory for the named file. If we found the associated directory entry, we release all file space and erase the directory entry.
4. **Truncating a file**. The user erases all the contents of a file but keep its attributes. The length of the file will be reset to zero.
5. **Writing a file**. write( ) system call is used to write a file. It specifies both the name of the file and the information to be written to the file. The system searches the filename in the directory to find the file's location.
6. **Reading a file**. read( ) system call is used to read from a file. It specifies the name of the file and where the next block of the file should be put. The directory is searched for the associated entry.

Note: A process is usually either reading from or writing to a file, the current operation location can be kept as a per-process **Current-File-Position Pointer**. Both the read and write operations use this same pointer.

open( ) and close( ) system calls

· System calls open( ) and close( ) are used to open and to close a file respectively. · **OS** maintains information about all open files in **Open-File Table.**

· When a file operation is requested, the file is indexed into Open File Table. · When a file is closed by a process then the OS removes its entry from the open-file table.

Operating system uses **2-levels** of Internal tables:

1. **Per-Process Table** The per-process table tracks all files that a process has open. This table stores information regarding the process's use of the file. Each entry in the per process table in turn points to a system-wide open-file table.
2. **System-Wide Table:** It contains process-independent information, such as the location of the file on disk, access dates and file size. Once a file has been opened by one process, the system-wide table includes an entry for the file. When another process executes an open( ) call, a new entry is added to the process's open-file table pointing to the appropriate entry in the system-wide table.

An open file is associated with following information:

· **File pointer:** This pointer is unique to each process operating on the file. It must be kept separate from the on-disk file attributes. On systems that do not include a file offset as part of the read( ) and write( ) system calls, the system must track the last read– write location as a current-file-position pointer.

· **File-open count:** The open-file table also has an **open count** associated with each file to indicate how many processes have opened that file. Each close( ) decreases the open count. When the open count reaches zero, the file is no longer in use and the file's entry is removed from the open-file table.

· **Disk location of the file**: Most file operations require the system to modify data several times within the file. The information needed to locate the file on disk is kept in main memory so that the system does not have to read it from disk for each operation.

· **Access rights**: Each process opens a file in an access mode. This information is

File Types

· File types are generally included as part of file name. The file name is split into two parts: a name and an extension usually separated by a dot.

· The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.

Example: resume.docx, server.c and ReaderThread.cpp.

The below table shows the common file types:

File Type	Extension	Function
executable	exe, com, bin or none	ready-to-run machinelanguage program
Object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	Bat,sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf,docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Internal File Structure

· Locating an offset within a file can be complicated for the operating system. · Disk systems have a well-defined block size determined by the size of a sector. · All disk I/O is performed in units of one block (physical record). All blocks are the same size.

Problem: It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may have different lengths.
Solution: Packing a number of logical records into physical blocks solves this problem.

Example: The UNIX operating system defines all files to be streams of bytes. Each byte is individually addressable by its offset from the beginning of the file. · Logical

record size, physical block size and packing technique determine how many logical records are in each physical block.

· Packing can be done either by the user's application program or by the operating system. **Note:** All file systems suffer from internal fragmentation. The larger the block size, the greater the internal fragmentation.

ACCESS METHODS

Files store information. When a file is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways: 1. Sequential Access

2. Direct Access

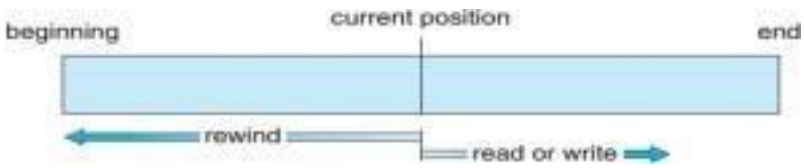
3. Indexed Access

Sequential Access

Information in the file is processed in order, one record after the other record. Example: editors and compilers usually access files in sequential order. Reads and writes make up the bulk of operations on a file:

· **read_next( )** operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.

· **write_next( )** operation appends to the end of the file and advances to the end of the newly written material.



Sequential access is based on a tape model of a file and works on sequential-access devices.

Direct Access or Relative Access

· In direct access method, the file is viewed as a numbered sequence of blocks or records. · There are no restrictions on the order of reading or writing for a direct-access file. · Thus, we may read block 14, then read block 53 and then write block 7.

· A file is made up of fixed-length **logical records** that allow programs to read and write records rapidly in no particular order.

· The direct-access method is based on a disk model of a file, since disks allow random access to any file block.

· Databases are direct access type. When a query concerning a particular **subject** arrives, we compute which block contains the **answer** and then read that block directly to provide the desired information.

Example: Airline-reservation system

· We might store all the information about a particular flight 713 in the block identified by the flight number.

· The number of available seats for flight 713 is stored in block 713 of the reservation file. · To store information about a larger set, such as people, we might compute a

hash function on the people's names to determine a block to read and search.

File operation in Direct Access Method

read(n) and write(n) are the read and write operation performed in Direct Access method where **n** represent the block number.

The block number provided by the user to the operating system is a **Relative Block Number**. · A relative block number is an index relative to the beginning of the file. · Thus, the first relative block of the file is 0, the next is 1 and so on.

· Relative block numbers allows the **OS** to decide where the file should be placed and helps to prevent user from accessing portions of the file system that may not be part of its file.

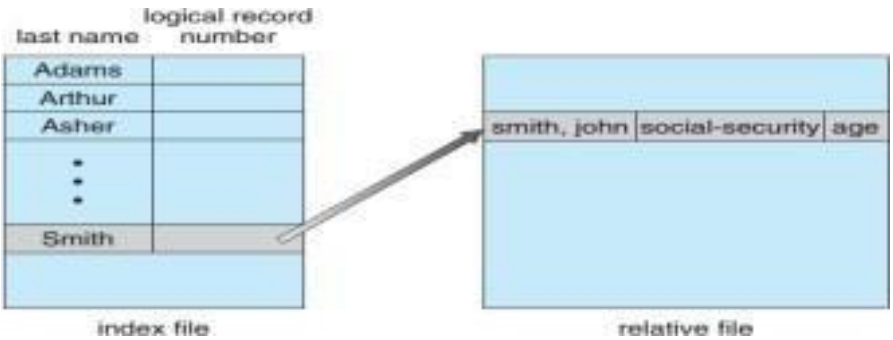
Indexed Access

The **index** is like an index in the back of a book that contains pointers to the various blocks. · To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.

· To find a record we can make a binary search of the index. This search helps us to know exactly which block contains the desired record and access that block.

· This structure allows us to search a large file doing little I/O.

· With large files, the index file itself may become too large to be kept in memory. · One solution is to create an index for the index file. The primary index file contains pointers to secondary index files, which point to the actual data items.



DIRECTORY STRUCTURE

Files are stored on random-access storage devices such as Hard-disks, Optical-disks and Solid-state disks.

· A storage device can be used for a file system. It can be subdivided for finer-grained control.

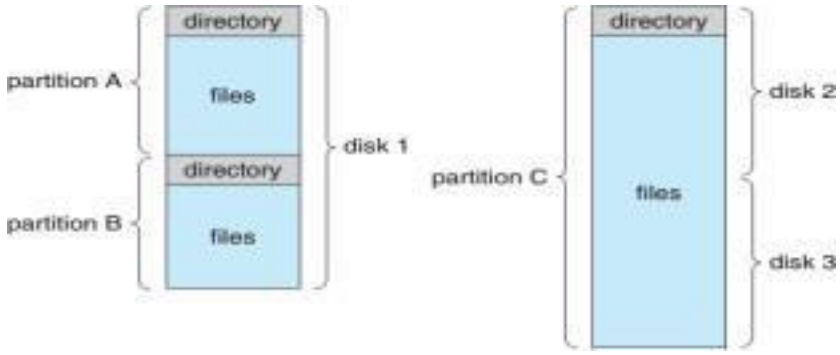
· **Ex:** A disk can be **partitioned** into quarters. Each quarter can hold a separate file system. · Partitioning is useful for limiting the sizes of individual file systems, putting multiple file system types on the same device or leaving part of the device available for other uses, such as swap space or unformatted (raw) disk space.

· A file system can be created on each of these disk partitions. Any entity containing a file system is generally known as a **Volume**.

· Volume may be a subset of a device, a whole device. Each volume can be thought of as a virtual disk.

· Volumes can also store multiple operating systems. Volumes allow a system to boot and run more than one operating system.

· Each volume contains a file system maintains information about the files in the system. · This information is kept in entries in a **Device directory** or **Volume table of contents**. · The device directory (**directory**) records information such as name, location, size and type for all files on that volume.
The below figure shows the typical file system organization:



Storage Structure in Solaris OS

The file systems of computers can be extensive. Even within a file system, it is useful to segregate files into groups and manage those groups. This organization involves the use of directories.
Consider the types of file systems in the Solaris Operating system:
· **Tmpfs** is a ¯temporary‖ file system that is created in volatile main memory and has its contents erased if the system reboots or crashes
· **objfs** is a ¯virtual‖ file system that gives debuggers access to kernel symbols · **ctfs** is a virtual file system that maintains ¯contract‖ information to manage which processes start when the system boots and must continue to run during operation · **lofs** is a ¯loop back‖ file system that allows one file system to be accessed in place of another file system.
· **procfs** is a virtual file system that presents information on all processes as a file system · **ufs, zfs** are general-purpose file systems.

Operations on Directory

Different operations performed on a directory are:
· **Search for a file**. This operation searches a directory structure to find the entry for a particular file. It finds all files whose names match with a particular pattern. · **Create a file**. When a new file is created its entry is added to the directory. · **Delete a file**. When a file is no longer needed, we can remove it from the directory. · **List a directory**. We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
· **Rename a file**. A file can be renamed, when the contents or use of the file changes (i.e.) csec.txt to cse.txt or cse.txt to cse.c file etc.
· **Traverse the file system**. We may wish to access every directory and every file within a directory structure.

LOGICAL STRUCTURE OF A DIRECTORY

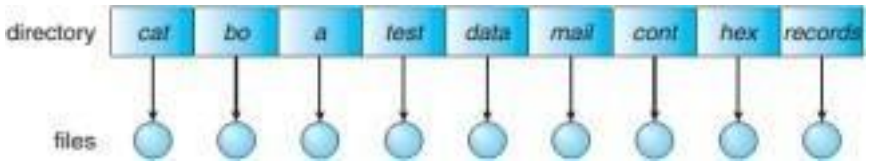
The different directory structures are:

1. Single Level Directory
2. Two-Level Directory

3. Tree Structured Directory
4. Acyclic-Graph Directories
5. General Graph Directory

Single-Level Directory

In Single-Level Directory structure, all files are contained in the same directory.



· A single-level directory has significant limitations that when the number of files increases or when the system has more than one users, all files are in the same directory, they must have unique names.

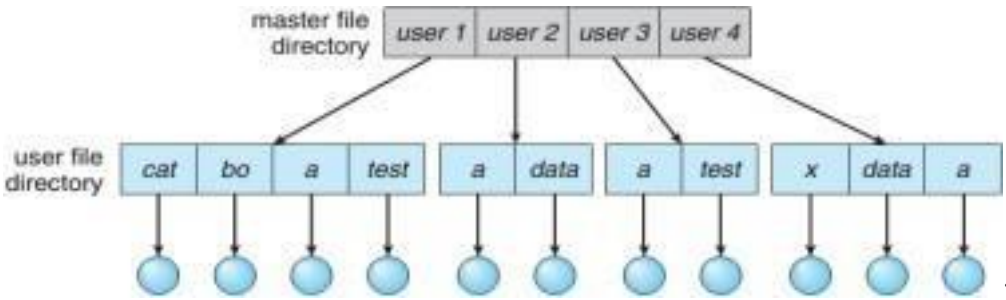
· Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.

· It is common for a user to have hundreds of files on one computer system. · Keeping track of so many files is a difficult task.

Two-Level Directory

· In the two-level directory structure, each user has his own **user file directory (UFD)**. · Each UFD lists only the files of a single user.

· When a user job starts or a user logs in, the system's **Master File Directory (MFD)** is searched. MFD is indexed by user name or account number and each entry points to the UFD for that user.



· When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with same name as long as all the file names within each UFD are unique.

· To create a file for a user, the operating system searches only that user's UFD to check whether another file of that name exists.

· To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

· This way the two-level directory structure solves the name-collision problem. · To name a particular file uniquely in a two-level directory, we must give both the user name and the file name.

A two-level directory can be thought of as a tree or an inverted tree, of height 2. · The root of the tree is the MFD.

· MFD's direct descendants are the UFDs.

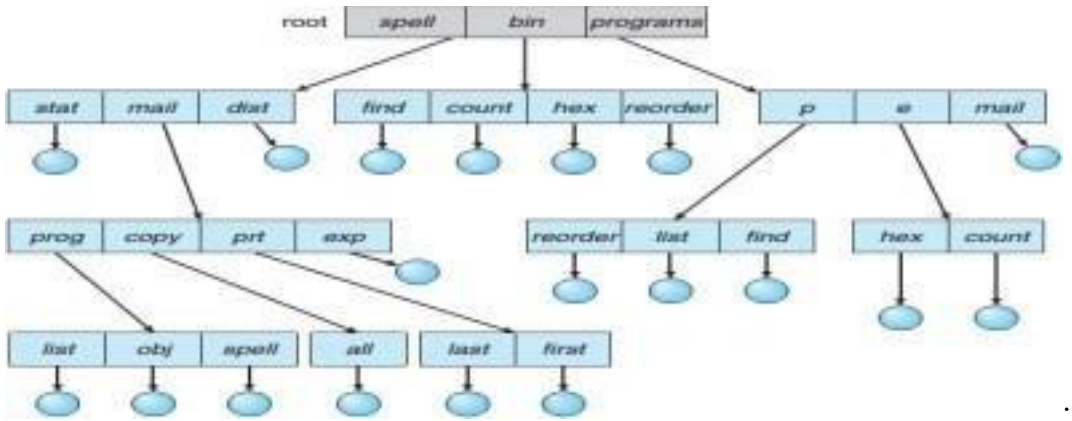
· The descendants of the UFDs are the files.
· The files are the leaves of the tree.

Specifying a user name and a file name defines a path in the tree from the root (MFD) to a leaf (a file).
· A user name and a file name define a **path name**.
· Every file in the system has a path name.
· To name a file uniquely, a user must know the path name of desired file.

The user directories themselves must be created and deleted as necessary. · A special system program is run with the appropriate user name and account information. · The program creates a new UFD and adds an entry in the MFD.
· The execution of this program might be restricted to system administrators.

Disadvantages:
· This structure effectively isolates one user from another.
· Isolation is an advantage when the users are completely independent but it is a disadvantage when the users want to cooperate on some task and to access others files. · Some systems simply do not allow one local user files to be accessed by other users.

Tree-Structured Directories
Tree Structure allows users to create their own subdirectories and to organize their files accordingly.



· The tree has a root directory and every file in the system has a unique path name. · A directory (or) subdirectory contains a set of files or subdirectories. · A directory is simply another file, but it is treated in a special way. All directories have the same internal format.
· One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).

Each process has a current directory. The **current directory** should contain most of the files that are of current interest to the process.
· When reference is made to a file, the current directory is searched.
· If a file is needed that is not in the current directory, then the user must specify a path name or change the current directory to the directory holding that file.
· To change directories, a system call is provided that takes a directory name as a parameter and uses it to redefine the current directory.

· Thus, the user can change the current directory whenever the user wants.

Path names can be of **2-types:** Absolute and Relative path name.

1. An **absolute path name** begins at the root and follows a path down to the specified file, giving the directory names on the path.
2. A **relative path name** defines a path from the current directory.

Example: If the current directory is **root/spell/mail** then the relative path name **prt/first** refers to the same file as does the absolute path name **root/spell/mail/prt/first.**

Deletion of Directory

If a directory is empty, its entry will be deleted form corresponding the directory. If the directory to be deleted is not empty but contains several files or subdirectories then one of the two approaches can be followed:

1. Some systems will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory.

2. When a request is made to delete a directory, all the directory's files and subdirectories are also to be deleted. Example: UNIX rm command used for this purpose. **Note:** With a tree-structured directory system, users can be allowed to access the files of other users. Example: user B can access a file of user A by specifying its path names.

Acyclic-Graph Directories

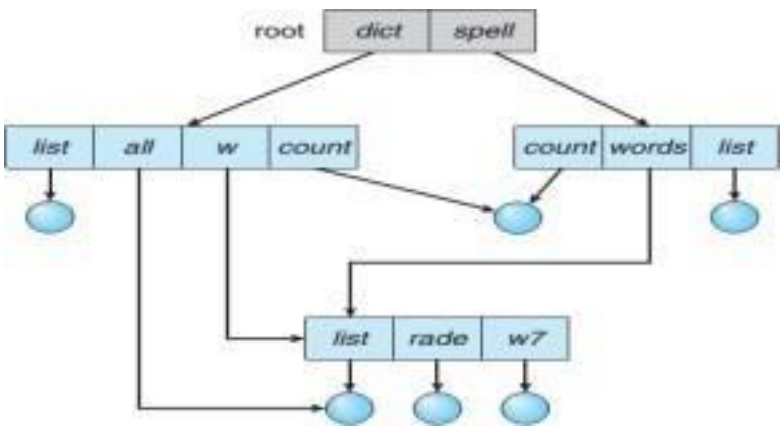
The acyclic graph is a natural generalization of the tree-structured directory scheme. · A tree structure prohibits the sharing of files or directories.

· An **acyclic graph** is a graph with no cycles allows directories to share subdirectories and files. The same file or subdirectory may be in two different directories.

· With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other.

· Sharing is particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.

· UNIX implements Shared files and subdirectories by creating a new directory entry called a Link. A **link** is effectively a pointer to another file or subdirectory.



Example: A link may be implemented as an absolute or a relative path

name. · When a reference to a file is made, we search the directory.

· If the directory entry is marked as a link, then the name of the real file is included in the link information.

· We **resolve** the link by using that path name to locate the real file.

· Links are easily identified by their format in the directory entry and Links are effectively indirect pointers.

· The operating system ignores these links when traversing directory trees to preserve the acyclic structure of the system.

Problems with Acyclic-Graph Directories

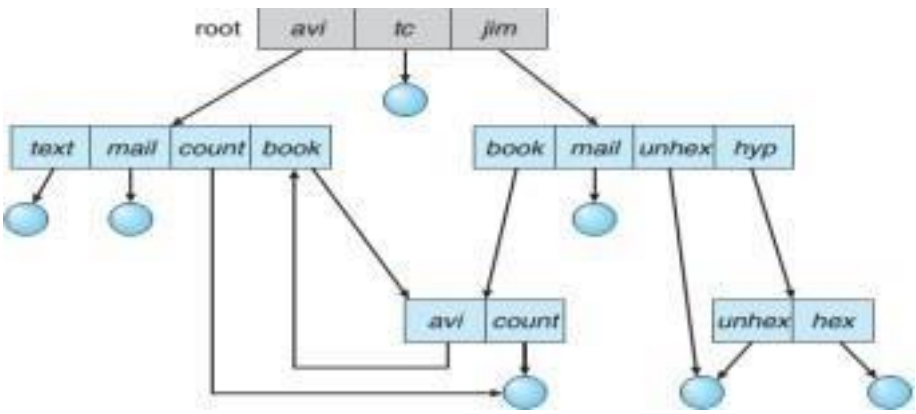
1. A file may now have multiple absolute path names. Consequently, distinct file names may refer to the same file.
2. Deletion of shared file is problematic. Because more than one user is using the file if one user deletes a shared file, it may leave dangling pointers to non-existence file for other users.

General Graph Directory

General Graph Directory structure is an acyclic graph with Cycles.

· A problem with using an acyclic-graph structure is ensuring that there are no cycles.

· In tree-structure directory we can add new files and subdirectories to an existing tree structured directory preserves the tree-structured nature but if we add links, the tree structure is destroyed, resulting in a simple graph structure.



· The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file. · If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance.

· **Problem:** A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating.

· **Solution:** we can limit arbitrarily the number of directories that will be accessed during a search.

FILE-SYSTEM MOUNTING

A file system must be mounted before it can be available to processes on the system.

· The directory structure may be built out of multiple volumes, which must be mounted to make them available within the file-system name space.

· When a file system is mounting, the operating system is given the name of the

device and the **Mount point.**

· The mount point is the location within the file structure where the file system is to be attached. In general a mount point is an empty directory.

· Example: On a UNIX system, a file system containing a user's **home** directories might be mounted as /home, then to access the directory structure within that file system, we could precede the directory names with /home, as in **/home/jane**.

· After mounting, the operating system verifies that the device contains a valid file system. · Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point.

Consider the above file system, the triangles represent subtrees of directories. · Figure (a) shows existing systems and Figure (b) shows Unmounted volume residing on /device/dsk.

· The last figure shows the mounting of the volume residing on **/device/dsk** over **/users**.

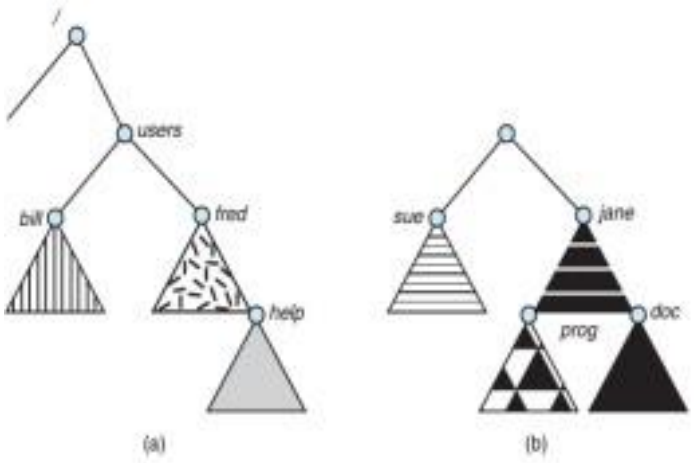


Figure 11.14 File system. (a) Existing system. (b) Unmounted volume.

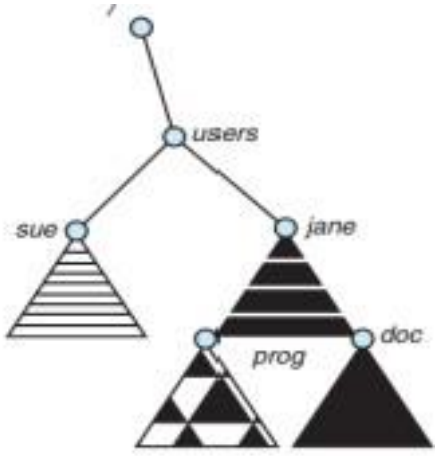


Figure 11.15 Mount point.

Mounting in Windows Operating System

· The Microsoft Windows family of operating systems maintains an extended two-level directory structure, with devices and volumes assigned drive letters.

· Volumes have a general graph directory structure associated with the drive letter.

· The path to a specific file takes the form of **drive-letter:\path\to\file (**i.e. **F:\dir\f1.txt)**

PROTECTION

The information is stored on the computer system. Protection deals with issue of improper access of information to the illegitimate users.

Protection provides controlled access by limiting the types of file access that can be made. Several different types of operations may be controlled:

· **Read**. Read from the file.

· **Write**. Write or rewrite the file.

· **Execute**. Load the file into memory and execute it.

· **Append**. Write new information at the end of the file.
· **Delete**. Delete the file and free its space for possible reuse.
· **List**. List the name and attributes of the file.
· Renaming, copying and editing the file, may also be controlled.

Access Control

· Different users may need different types of access to a file or directory. · Systems uses A**ccess-control list (ACL) Scheme that** specifies user names and the types of access allowed for each user.
· When a user requests access to a particular file, the operating system checks the access list associated with that file.
· If that user is listed for the requested access, the access is allowed. Otherwise a protection violation occurs and the user job is denied access to the file.

Many systems recognize three classifications of users in connection with each file: · **Owner**: The user who created the file is the owner.
· **Group**: A set of users who are sharing the file and need similar access is a work group. · **Universe**: All other users in the system constitute the universe.

Protection in UNIX

In the UNIX system, groups can be created and modified only by the manager or super user. · With the more limited protection classification, only three fields are needed to define protection.
· Each field is a collection of bits and each bit either allows or prevents the access associated with it.
· The UNIX system defines three fields of 3 bits each–rwx, where **r** controls read access, **w** controls write access and **x** controls execution.
· A separate field is kept for the file owner, for the file's group and for all other users. · In this scheme, 9 bits per file are needed to record protection information.

FILE-SYSTEM STRUCTURE

File systems are maintained on Secondary Storage Disks.
Two reasons for storing file systems on disk are:

1. A disk can be rewritten in place (i.e.) It is possible to read a block from the disk, modify the block and write it back into the same place.
2. A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly and switching from one file to another requires only moving the read-write heads and waiting for the disk to rotate.

I/O transfers between memory and disk are performed in units of **blocks**.
· Each block has one or more sectors.
· Depending on the disk drive, sector size varies from 32 bytes to 4,096 bytes; the usual size is 512 bytes.
· **File systems** provide efficient and convenient access to the disk by allowing data to be stored, located and retrieved easily.

Design issues of File System

1. Defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file and the directory structure for organizing files.
2. Creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

Layered Structured File System

· **I/O control** level consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system.

· **Basic File System** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.

· Each physical block is identified by its numeric disk address

· Example: drive 1, cylinder 73, track 2, sector 10.

· Basic file system layer also manages the memory buffers and caches that hold various file-system, directory and data blocks.

· F**ile-Organization Module** knows about files and their logical blocks, as well as physical blocks.

· The file-organization module includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested. · **Logical File System** manages metadata information. Metadata includes all of the file system structure except the actual data (or) contents of the files.

· Logical File System maintains file structure via File-Control Blocks. A **File-Control Block (FCB)** contains information about the file, including ownership, permissions and location of the file contents. In UNIX FCB is called as an **inode**.

· The logical file system is also responsible for protection.

Advantage: Layered structure minimizes the duplication of code. Code reusability is possible with this structure.

Disadvantage: Layering can introduce more operating-system overhead, which may result in decreased performance.

File systems supported by different Operating systems

1. **UNIX** uses the **UNIX File System (UFS)** is based on Berkeley Fast File System (FFS). 2. **Windows** supports disk file-system formats of **FAT**, **FAT32** and **NTFS** as well as CD ROM and DVD file-system formats.

3. **Standard Linux file system** is known as the **Extended File System**, with the most common versions being ext3 and ext4.

FILE SYSTEM IMPLEMENTATION

Several on-disk and in-memory structures are used to implement a file system. These structures vary depending on the operating system and the file system.

On-Disk Structure

The file system may contain information about how to boot an operating system stored on disk, the total number of blocks, the number and location of free blocks, the directory structure and individual files.

Several On-Disk structure are given below:

Boot Control Block

· A Boot Control Block (per volume) can contain information needed by the system to boot an operating system from that volume.

· If the disk does not contain an operating system, this block can be empty. · It is typically the first block of a volume.

· In UFS, it is called the **Boot Block**. In NTFS, it is the **Partition Boot Sector**.

Volume Control Block

· A Volume Control Block (per volume) contains volume (or) partition details such as the number of blocks in the partition, the size of the blocks, a free-block count and free-block pointers and a free-FCB count and FCB pointers.

· In UFS, this is called a **Super-Block**. In NTFS, it is stored in the **Master File Table**. **Directory Structure**

· A directory structure (per file system) is used to organize the files.

· In UFS, this includes file names and associated inode numbers.

· In NTFS, it is stored in the master file table.

Per-File FCB

· A per-file FCB contains many details about the file.

· It has a unique identifier number to allow association with a directory entry. · In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file.

In-Memory Structure

The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time, updated during file-system operations and discarded at dismount.

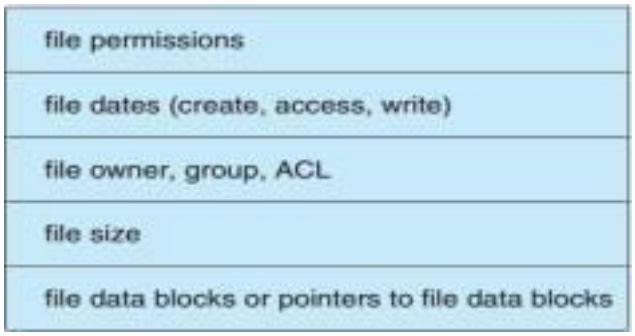
Several in-memory structures are given below:

· An in-memory **mount table** contains information about each mounted volume. · An in-memory directory-structure cache holds the directory information of recently accessed directories. For directories at which volumes are mounted, it can contain a pointer to the volume table.

· The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.

· The **per-process open-file table** contains a pointer to the appropriate entry in the system wide open-file table, as well as other information.

Buffers hold file-system blocks when they are being read from disk or written to disk. The below figure shows the FCB

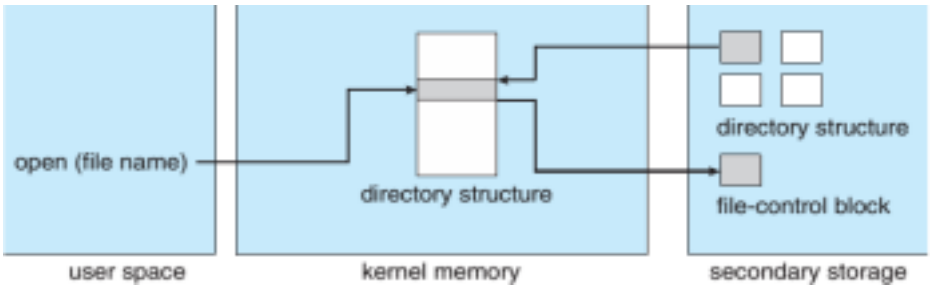


· To create a new file, an application program calls the logical file system. · The logical file system knows the format of the directory structures. · To create a new file, it allocates a new FCB.

· The system then reads the appropriate directory into memory, updates it with the new file name and FCB and writes it back to the disk.

Process of Opening a file

After a file has been created, it can be used for I/O.



· To open a file we use a system call open( ). The open( ) call passes a file name to the logical file system.

· The open( ) system call first searches the system-wide open-file table to see if the file is already in use by another process.

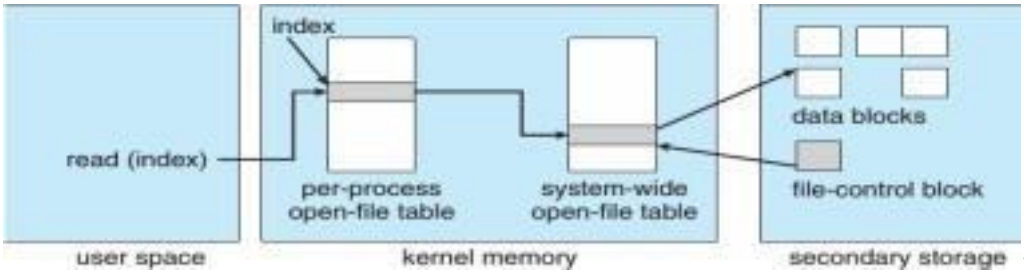
· If the file is open, a per-process open-file table entry is created pointing to the existing system-wide open-file table.

· If the file is not already open, the directory structure is searched for the given file name. · Once the file is found, the FCB is copied into a system-wide open-file table in memory. · This table not only stores the FCB but also tracks the number of processes that have the file open.

Process of Reading a File

· After an entry has been made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields.

· These other fields may include a pointer to the current location in the file for the next read( ) or write( ) operation and the access mode in which the file is open.



· The open( ) call returns a pointer to the appropriate entry in the per-process file-system table. All file operations are then performed via this pointer.

· The file name may not be part of the open-file table, as the system has no use for it once the appropriate FCB is located on disk.

· FCB could be cached to save time on subsequent opens of the same file. The name given to the entry varies.

· UNIX refers to FCB as a **File Descriptor**. Windows refers to FCB as a **File Handle.**

Process of Closing the File

· When a process closes the file, the per-process table entry is removed and the system wide entry's open count is decremented.

· When all users that have opened the file close it, any updated metadata is copied back to the disk-based directory structure and the system-wide open-file table entry is removed.

Partitions and Mounting

· A disk can be sliced into multiple partitions. A partition can be raw or cooked partition. · A partition which does not contain any file system is called raw partition.

· A partition that contains a file system is called as cooked partition.

· UNIX swap space can use a raw partition, since it uses its own format on disk and does not use a file system.

· Boot information can be stored in a separate partition.

· It has its own format, because at boot time the system does not have the file-system code loaded and therefore cannot interpret the file-system format.

· Boot information is a sequential series of blocks, loaded as an image into memory. · Execution of the image starts at a predefined location, such as the first byte. · This **boot loader** knows about the file-system structure to be able to find and load the kernel and start it executing.

· It can contain more than the instructions for how to boot a specific operating system.

· Many systems can be **dual-booted**, allowing us to install multiple operating systems on a single system.

· A boot loader that understands multiple file systems and multiple operating systems can occupy the boot space.

· Once loaded, it can boot one of the operating systems available on the disk. · The disk can have multiple partitions, each containing a different type of file system and a different operating system.

· **Root partition** contains the operating-system kernel. Sometimes other system files are mounted at boot time.

Other volumes can be automatically mounted at boot or manually mounted later, depending on the operating system.

Example 1: Microsoft Windows System mounts each volume in a separate name space, denoted by a letter and a colon.

· To record that a file system is mounted at **F:**, the operating system places a pointer to the file system in a field of the device structure corresponding to **F:**.

· When a process specifies the driver letter, the operating system finds the appropriate file system pointer and traverses the directory structures on that device to find the specified file or directory.

· Later versions of Windows can mount a file system at any point within the existing directory structure.

Example 2: In UNIX based systems, file systems can be mounted at any directory. · Mounting is implemented by setting a flag in the in-memory copy of the inode for that directory.

· The flag indicates that the directory is a mount point. A field then points to an entry in the mount table, indicating which device is mounted there.

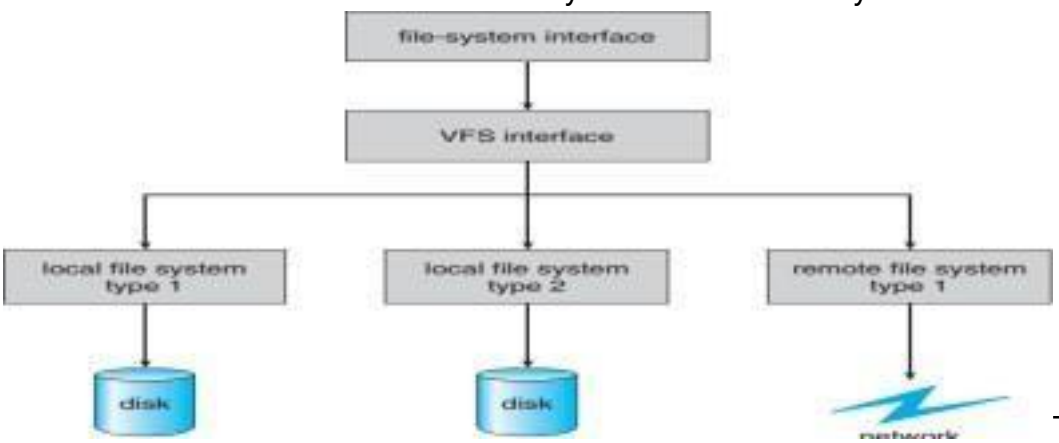
· Mount table entry contains a pointer to the superblock of the file system on that device.

Virtual File Systems

The first layer is the file-system interface, based on the open( ), read( ), write( ) and close( ) system calls and also based on file descriptors.

The second layer is called the **virtual file system (VFS)** layer. The VFS layer serves two important functions:

1. It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally.
2. It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure called a **vnode**, that contains a numerical designator for a network-wide unique file. This network-wide uniqueness is required for support of network file systems. The kernel maintains one vnode structure for each active node. A node may be a file or directory.



The

VFS distinguishes local files from remote ones and local files are further distinguished according to their file-system types.

· VFS activates file-system-specific operations to handle local requests according to their file-system types and calls the Network File System (NFS) protocol procedures for remote requests.

· File handles are constructed from the relevant vnodes and are passed as arguments to these procedures.

The third layer implements the file-system type or the remote-file-system protocol.

DIRECTORY IMPLEMENTATION

1. Linear List
2. Hash Table

Linear List

· It maintains linear list of file names with pointers to the data blocks. It is time consuming. · To create a new file, we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory.

· To delete a file, we search the directory for the named file and then release the space allocated to it.

· To reuse the directory entry either we can mark the entry as unused or we can attach it to a list of free directory entries.

Disadvantage: It uses linear search to find a file. Linear search is very slow.

Hash Table

· The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. It decreases the directory search time.

· Insertion and deletion are also very easy to implement.

The major difficulty hash tables are its generally fixed size and Hash tables are dependent on hash function on that size.

Example: Assume that we make a linear-probing hash table that holds 64 entries. · The hash function converts file names into integers from 0 to 63.

· If we try to create a 65th file, we must enlarge the directory hash table to 128 entries. · Hence we need a new hash function that must map file names to the range 0 to 127 and must reorganize the existing directory entries to reflect their new hash-function values.

ALLOCATION METHODS

Three major methods of allocating disk space are in wide use:

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

Contiguous Allocation

Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk. · Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block *b* + 1 after block *b*

normally requires no head movement.

· When head movement is needed the head need only move from one track to the next. · Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block.

· If the file is *n* blocks long and starts at location *b,* then it occupies blocks ***b, b* + 1, *b* + 2, ..., *b* + *n* −** 1.

· The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.

Accessing a file that has been allocated contiguously is easy.

· For sequential access, the file system remembers the disk address of the last block referenced and when necessary, reads the next block.

· For direct access to block ***i*** of a file that starts at block ***b,*** it can immediately access block **b+i**.

· Both sequential and direct access can be supported by contiguous allocation.

Two-Problems with Contiguous Allocation:

1. Finding space for a new file
2. Determining how much space is needed for a file.

Finding space for a new file

· The contiguous-allocation problem occurs in dynamic storage-allocation that involves how to satisfy a request of size *n* from a list of free holes.

· First fit and best fit are the most common strategies used to select a free hole from the set of available holes.

· Both, First fit and Best fit algorithms suffer from the problem of **external fragmentation**. · As files are allocated and deleted, the free disk space is broken into little pieces. · External fragmentation exists whenever free space is broken into chunks. · It becomes a problem when the largest contiguous chunk is insufficient for a request. · The storage is fragmented into a number of holes, none of which is large enough to store the data.

One solution for this problem is **Compaction:**

· Compaction solves external fragmentation by copying an entire file system onto another disk.

· The original disk is then freed completely, creating one large contiguous free space.

· We then copy the files back onto the original disk by allocating contiguous space from this one large hole.

· The cost of compaction is very high when the size of the hard disk is huge. The time taken for compaction will be high as the size of the hard disk increases.

Determining how much space is needed for a file

· When the file is created, the total amount of space it will need must be found and allocated.

· If we allocate too little space to a file, we may find that the file cannot be extended. · Especially with a best-fit allocation strategy, the space on both sides of the file may be in use. Hence, we cannot make the file larger in place.

Two possibilities then exist.

· First, the user program can be terminated, with an appropriate error message. · The user must then allocate more space and run the program again. · These repeated runs may be costly.

· To prevent them, the user will normally overestimate the amount of space needed, resulting in considerable wasted space.

· The other possibility is to find a larger hole, copy the contents of the file to the new space and release the previous space.

· All these are time consuming and system performance will be effected.

Linked Allocation

Linked allocation solves all problems of contiguous allocation.

· With linked allocation, each file is a linked list of disk blocks.

· Disk blocks are scattered anywhere on the disk.

· The directory contains a pointer to the first and last blocks of the file.

Consider the below figure that shows linked list allocation:

· A file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10 and finally block 25.

· Each block contains a pointer to the next block.

· These pointers are not made available to the user.

· Thus, if each block is 512 bytes in size and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

Advantage:

Linked List allocation avoids Compaction

· To create a new file, we simply create a new entry in the directory.

· With linked allocation, each directory entry has a pointer to the first disk block of the

file. · This pointer is initialized to null (the end-of-list pointer value) to signify an empty file. The size field is also set to 0.

· A write to the file causes the free-space management system to find a free block and this new block is written to and is linked to the end of the file.

· To read a file, we simply read blocks by following the pointers from block to block. · There is no external fragmentation with linked allocation and any free block on the free space list can be used to satisfy a request.

· The size of a file need not be declared when the file is created. A file can continue to grow as long as free blocks are available. Hence by Linked List allocation avoids external fragmentation and it avoid need for compact disk space.

Disadvantages:

1. It is inefficient for Direct Access
2. Space for Pointers
3. Reliability

Linked list allocation can be used effectively only for sequential-access files. · To find the **ith** block of a file, we must start at the beginning of that file and follow the pointers until we get to the **ith** block.

· Each access to a pointer requires a disk read and some require a disk seek. · It is inefficient to support a direct-access capability for linked-allocation files. Another disadvantage is the space required for the pointers.

· If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information.

· Each file requires slightly more space than it would otherwise.

Solutions to above problems: **Clustering**

· Cluster is a collection multiple blocks and we allocate clusters rather than blocks. · Let the file system define a cluster as four blocks and operate on the disk only in cluster units. Pointers then use a much smaller percentage of the file's disk space. · This method improves disk throughput and decreases the space needed for block allocation and free-list management.

· Clustering approach leads to the problem of internal fragmentation, because more space is wasted when a cluster is partially full than when a block is partially full.

Reliability issues will be arised

· The files are linked together by pointers scattered all over the disk.

· If a pointer were lost or damaged, this might result in picking up the wrong pointer. · This error could in turn result in linking into the free-space list or into another file.

FILE ALLOCATION TABLE (FAT)

Linked List allocation uses File Allocation Table.

· FAT is very efficient method of disk-space allocation. It was used by the MS-DOS operating system.

· A section of disk at the beginning of each volume contains the table.

· The table has one entry for each disk block and is indexed by block number. · The directory entry contains the block number of the first block of the file. · The table entry indexed by that block number contains the block number of the next block in the file.

· This chain continues until it reaches the last block, which has a special end-of-file value as the table entry.

· An unused block is indicated by a table value of 0. Allocating a new block to a file is to find the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value.

· The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached.

· The disk head must move to the start of the volume to read the FAT and find the location of the block in question, then move to the location of the block itself.

· A benefit is that random-access time is improved, because the disk head can find the location of any block by reading the information in the FAT.

Indexed Allocation

In Indexed allocation, each file has its own index block. An index block is an array of disk block addresses.

· The **i**th entry in the index block points to the **i**th block of the file.

· The directory contains the address of the index block.

· To find and read the **i**th block, we use the pointer in the **i**th index-block entry. · When the file is created, all pointers in the index block are set to null. · When the **i**th block is first written, a block is obtained from the free-space manager and its address is put in the **i**th index-block entry.

· Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more

space.

Indexed allocation suffers from wasted space and Pointer Overhead.

· The pointer overhead of the index block is greater than the pointer overhead of linked allocation.

· Consider we have a file of only one or two blocks.

· With linked allocation, we lose the space of only one pointer per block. · With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-null.

Determining size of the index block is a big issue in Indexed allocation. Several mechanisms are used for this purpose are:

1. Linked Scheme
2. Multilevel Index
3. Combined Scheme

Linked Scheme

· An index block is normally one disk block. Thus, it can be read and written directly by itself.

· To allow for large files, we can link together several index blocks.

· **Example:** An index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses.

· The next address (i.e.) the last word in the index block is null (for a small file) or is a pointer to another index block (for a large file).

Multilevel index

· A variant of linked representation uses a first-level index block to point to a set of second level index blocks, which in turn point to the file blocks.

· To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block.

· This approach could be continued to a third or fourth level, depending on the desired maximum file size.

· With 4,096-byte blocks, we could store 1,024 four-byte pointers in an index block. · Two levels of indexes allow 1,048,576 data blocks and a file size of up to 4 GB.

Combined Scheme

· It is used by the UNIX based file system that keeps the first 15 pointers of the index block in the file's inode.
· The first 12 of these pointers point to **direct blocks** (i.e.) they contain addresses of blocks that contain data of the file.
· Thus, the data for small files of no more than 12 blocks do not need a separate index block.

· If the block size is 4 KB, then up to 48 KB of data can be accessed directly. · The next three pointers point to **Indirect blocks**.
· The first points to a **Single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data.
· The second points to a **Double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. · The last pointer contains the address of a **Triple indirect block**.

FREE-SPACE MANAGEMENT

· Since disk space is limited, we need to reuse the space from deleted files for new files. · To keep track of free disk space, the system maintains a **Free-Space List**. · Free-space list records all free disk blocks, those not allocated to some file or directory. · To create a file, we search the free-space list for the required amount of space and
allocate that space to the new file. This space is then removed from the free-space list. · When a file is deleted, its disk space is added to the free-space list.

The free space can be managed in several ways:
1. Bit Vector
2. Linked List
3. Grouping
4. Counting
5. Space Maps

Bit Vector

The free-space list is implemented as a **Bit map** or **Bit vector**.
Each block is represented by one bit, the bit 1 represents block is free and bit 0 represents block is allocated.

Example: Consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 and 27 are free and the rest of the blocks are allocated. The free-space bit map would be **00111100111111000110000001110000 0**

Advantage: Its relative simplicity and its efficiency in finding the first free block or *n* consecutive free blocks on the disk.

Disadvantage: Bit Vectors are kept in main memory is possible for smaller disks. For larger disks it is not efficient to keep it in Main memory because A 1-TB disk with 4- KB blocks requires 256 MB to store its bit map. So, as the disk size increases, the bit vector size is also increases.

Linked List

All the free disk blocks are linked together by keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block and so on.

· Consider the above figure, that shows the set of free blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 and 27.

· The system would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3 and so on.

· This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.

Grouping

· It stores the addresses of *n* free blocks in the first free block.

· The first *n*– 1 of these blocks are free blocks and the last block contains the addresses of another *n* free blocks and so on.

· Addresses of a large number of free blocks can be found quickly than linked-list method.

Counting

· When space is allocated with the contiguous-allocation algorithm or through clustering, several contiguous blocks may be allocated or freed simultaneously.

· Here we keep the address of the first free block and the number (*n*) of free contiguous blocks that follow the first block.

· Each entry in the free-space list then consists of a disk address and a count. Hence the overall disk entries are small.

Space Maps

Oracle's **ZFS** file system was designed to encompass huge numbers of files, directories and even file systems.

· In its management of free space, ZFS uses a combination of techniques to control the size of data structures and minimize the I/O needed to manage those structures.

· ZFS creates **meta-slabs** to divide the space on the device into chunks of manageable size. · A volume contains hundreds of meta-slabs. Each meta-slab has an associated space map. · ZFS uses the counting algorithm to store information about free blocks. It uses log structured file-system techniques to record them.

· The space map is a log of all block activity such as allocating and freeing, in time order and in counting format.

· When ZFS decides to allocate or free space from a meta-slab, it loads the associated space map into memory in a balanced-tree structure (for very efficient operation), indexed by offset and replays the log into that structure.

· The in-memory space map is then an accurate representation of the allocated and free space in the meta-slab.

· ZFS also condenses the map as much as possible by combining contiguous free blocks into a single entry.

· Finally, the free-space list is updated on disk as part of the transaction-oriented operations of ZFS.

· During the collection and sorting phase, block requests can still occur and ZFS satisfies these requests from the log. In essence, the log plus the balanced tree is the free list