UNIT-V

FILE SYSTEM MANAGEMENT

File System Interface and Operations -Access methods, Directory Structure, Protection, File System Structure, Allocation methods, Free-space Management. Usage of open, create, read, write, close, lseek, stat, ioctl system calls.

File Concept:

- A file is a named collection of related information that is recorded on secondary storage.
- Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary.
- Files may be free form, such as text files. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user.

File Attributes:

- A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as example .c
- Some systems differentiate between uppercase and lowercase characters in names, whereas other systems do not.
- When a file is named, it becomes independent of the process, the user, and even the system that created it. For instance, one user might create the file example .c, and another user might edit that file by specifying its name.

A file's attributes vary from one operating system to another but typically consist of these:

- **Name:** The symbolic file name is the only information kept in human readable form.
- **Identifier:** This unique tag, usually a number, 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**: This information is a pointer to a device and to the location of the file on that device.
- **Size:** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection:** Access-control information determines who can do reading, writing, executing, and so on
- **Time, date, and user identification**:. This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring

File Operations:

- A file is an abstract data type .To define a file properly, we need to consider the operations that can be performed on files.
- The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.

Let's examine what the operating system must do to perform each of these six basic file operations.

- **Creating a file:** Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.
- **Writing a file:** To write a file, we make a system call specifying both the name of the file and the information to be written to the file.
  - Given the name of the file, the system searches the directory to find the file's location. The system must keep a write pointer to the location in the file where the next write is to take place.
  - The write pointer must be updated whenever a write occurs.
- **Reading a file:** To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put.
  - Again, the directory is searched for the associated entry, and the system needs to keep a read pointer to the location in the file where the next read is to take place.
  - Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process
  - Both the read and write operations use this same pointer, saving space and reducing system complexity.
- **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/0. This file operation is also known as files seek.
- **Deleting a file:** To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.
- **Truncating a file:** The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged –except for file length-but lets the file be reset to length zero and its file space released.

- Most Operating systems require that files be opened before access and closed after all access is complete.
- Normally, the programmer must open and close files explicitly, but some rare systems open the file automatically at first access. Information about currently open files is stored in an open **file table**, containing for **example:**

Several pieces of information are associated with an open file.

File pointer:

- 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.
- This pointer is unique to each process operating on the file and therefore must be kept separate from the on-disk file attributes.

File-open count:

- As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table.
- Because multiple processes may have opened a file, the system must wait for the last file to close before removing the open-file table entry.
- The file-open counter tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.

Disk location of the file:

- Most file operations require the system to modify data within the file.
- The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.

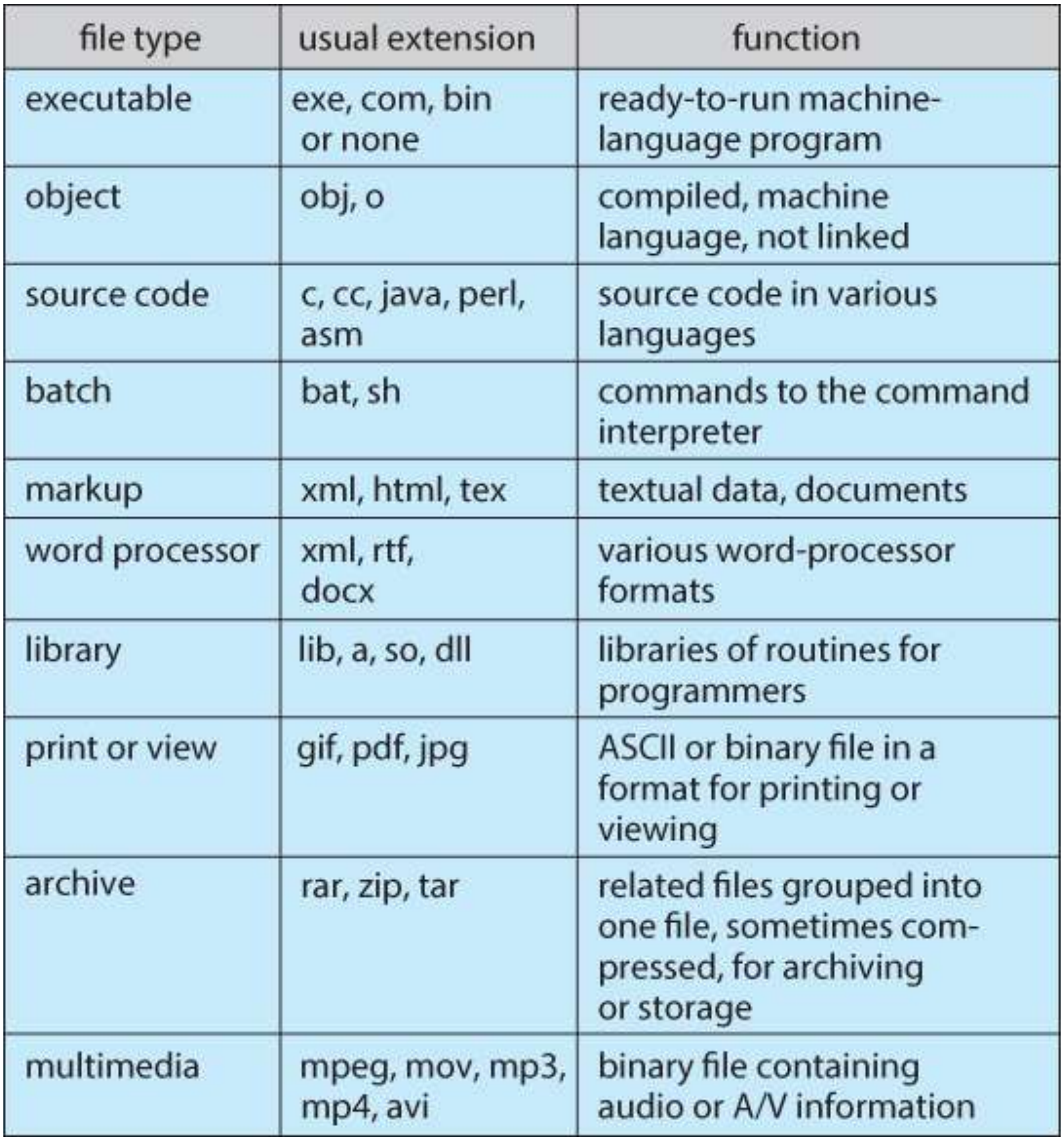
Access rights:

- Each process opens a file in an access mode.
- This information is stored on the per-process table so the operating system can allow or deny subsequent I/0 requests.

- Some operating systems provide facilities for **locking an open file** (or sections of a file). File locks allow one process to lock a file and prevent other processes from gaining access to it.
- File locks are useful for files that are shared by several processes.
- For example, a system log file that can be accessed and modified by a number of processes in the system.

File Types:

- File type refers to the ability of the operating system to distinguish different types of file such as text files source files and binary files etc. Many operating systems support many types of files. Operating system like MS-DOS and UNIX have the following types of files −
- Windows (and some other systems) use special file extensions to indicate the type of each file:



file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, 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

Figure 4-A: Common file types.

File Access Methods

- Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files. While others support many access methods, and choosing the right one for a particular application is a major design problem.

1). Sequential Access:

- The simplest access method is **sequential access**. Information in the file is processed in order, one record after the other. This mode of access is by far the most common.
- Examples: Editors, compilers.

A sequential access file supports a few operations:

- read next - read a record and advance the tape to the next position.
- write next - write a record and advance the tape to the next position.
- rewind
- skip n records - May or may not be supported. N may be limited to positive numbers, or may be limited to +/- 1.

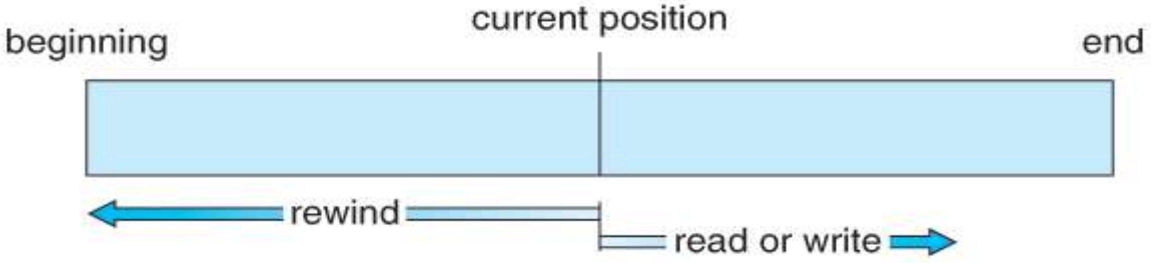


Figure 4-B: Sequential-access file.

2). Direct Access ((or relative access / Random access) :

- Here, a file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.
- For direct access, 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.
- Direct-access files are of great use for immediate access to large amounts of information.
- For the direct-access method, the file operations must be modified to include the block number as a parameter.
- Thus, we have read (n) rather than read next ( ), and write (n) rather than write next ( ).

3) Other Access Methods:

- Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file.
- The index, like an index in the back of a book, 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.

Directory and Disk Structure

- A storage device can be used in its entirety for a file system.
- For example, a disk can be partitioned into quarters, and each quarter can hold a separate file system.
- Any entity containing a file system is generally known as a **volume**. The volume may be a subset of a device, a whole device, or multiple devices linked together into a **RAID** ((redundant array of independent disks) set.

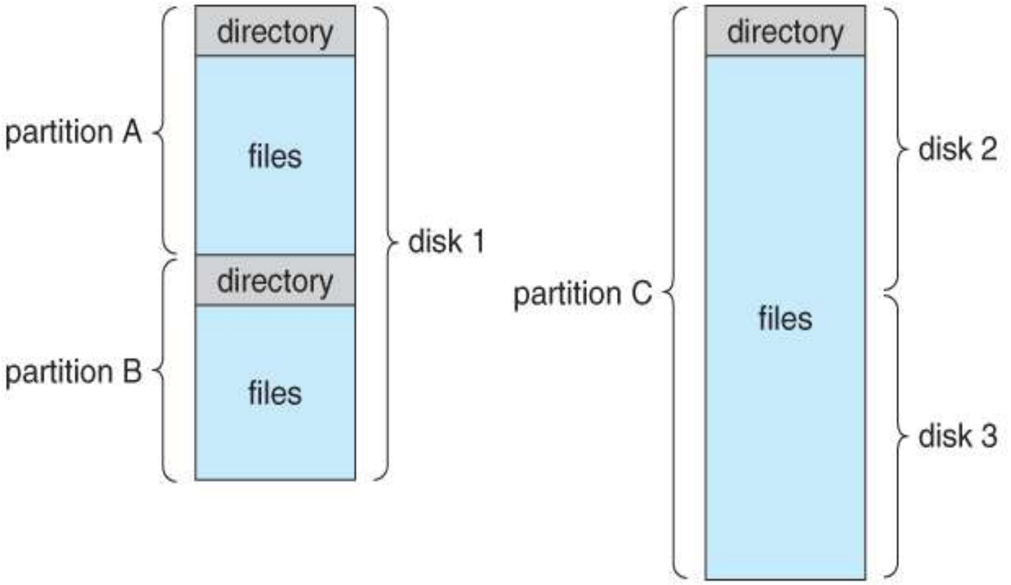


Figure 4-C: A typical file-system organization.

1. **Directory Overview:**

- The directory can be viewed as a symbol table that translates file names into their directory entries.
- When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:
  - **Search for a file**. We need to search a directory structure to find the entry for a particular file.
  - **Create a file**. New files need to be created and added to the directory.
  - **Delete a file**. When a file is no longer needed, we want to be able to 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**. Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes.
  - **Traverse the file system**. We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files to magnetic tape.

2. Single-Level Directory:

- The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand (Figure 4-D).

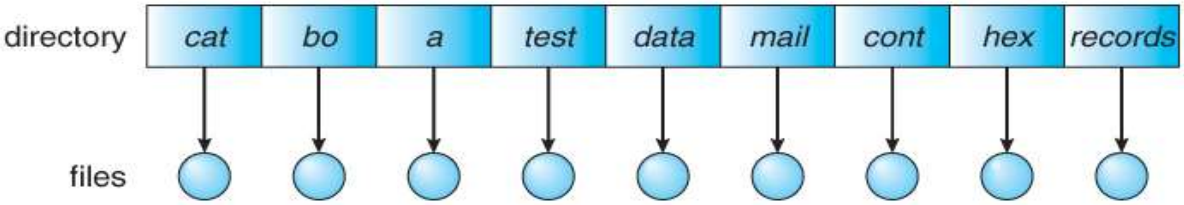


Figure 4-D: Single-level directory.

- When the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names.
- 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 not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system.
- A single-level directory often leads to confusion of file names among different users. The standard solution is to create a separate directory for each user.

3. Two-Level Directory:

- In the two-level directory structure, each user has his/her own **user file directory (UFD)**. The UFDs have similar structures, but each lists only the files of a single user.
- A **master file directory (MFD)** is used to keep track of each user's directory, and must be maintained when users are added to or removed from the system (Figure 4-E).

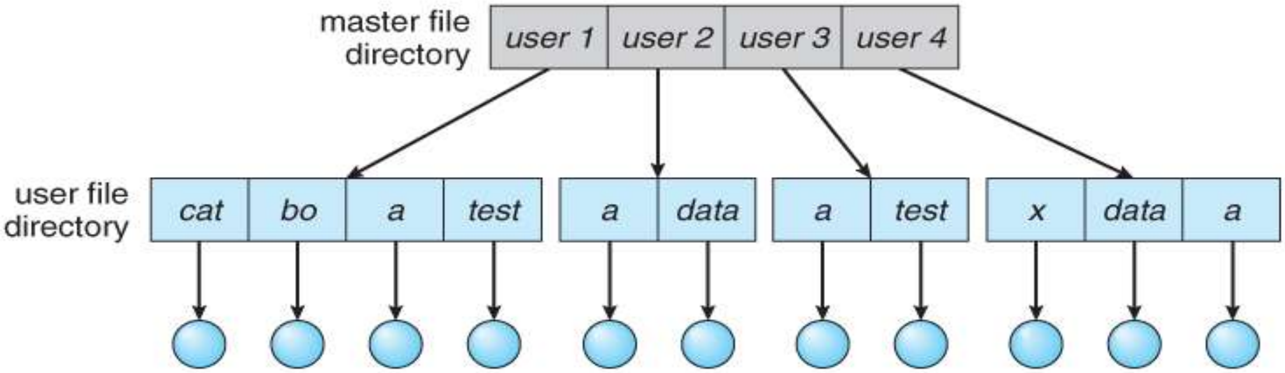


Figure 4-E: Two-level directory structure.

- The two-level directory structure solves the name-collision problem.
- It still has disadvantages- This structure effectively isolates one user from another.

- Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to access one another's files.
- Some systems simply do not allow local user files to be accessed by other users. If access is to be permitted, one user must have the ability to name a file in another user's directory.
- To name a particular file uniquely in a two-level directory, we must give both the user name and the file name.
- Thus, a user name and a file name define a **path name**. Every file in the system has a path name.

4. Tree-Structured Directories:

- This is extension of Two-Level Directory. This generalization allows users to create their own subdirectories and to organize their files accordingly.
- A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique **path name**.
- Path names can be of two types: absolute and relative. An **absolute path name** begins at the root and follows a path down to the specified file, giving the directory names on the path. A **relative path name** defines a path from the current directory
- 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). Special system calls are used to create and delete directories.

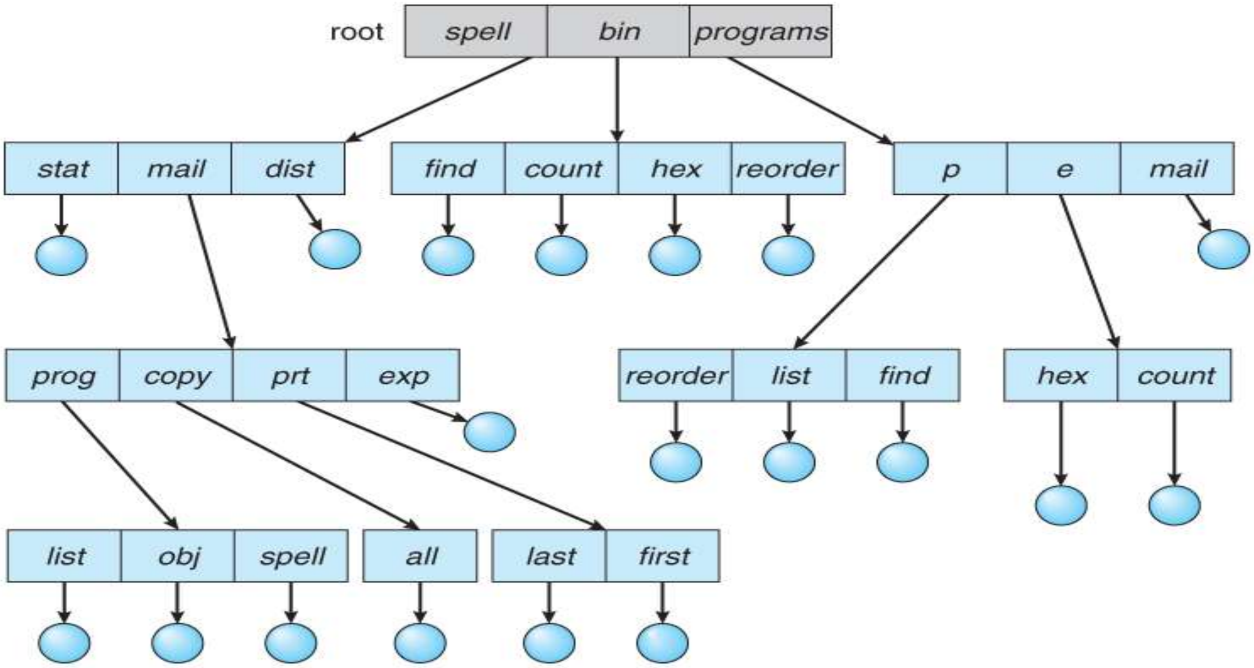


Figure 4-F: Tree-structured directory structure.

5. Acyclic-Graph Directories:

- When the same files need to be accessed in more than one place in the directory structure (e.g. because they are being shared by more than one user / process), it can be useful to provide an acyclic-graph structure. (Note the directed arcs from parent to child.).
- **An acyclic graph —that is, a graph with no cycles**—allows directories to share subdirectories and files (Figure 4-G).
- The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.
- Shared files and subdirectories can be implemented in several ways. Common way, exemplified by many of the UNIX systems, is to create a new directory entry called a **link**.
- **A link** is effectively a pointer to another file or subdirectory

NOTE: It is important to note that a shared file (or directory) is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. 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.

- **A hard link** (usually just called a link) involves multiple directory entries that both refer to the same file. Hard links are only valid for ordinary files in the same file system.
- **A symbolic link** that involves a special file, containing information about where to find the linked file. Symbolic links may be used to link directories and/or files in other file systems, as well as ordinary files in the current file system.

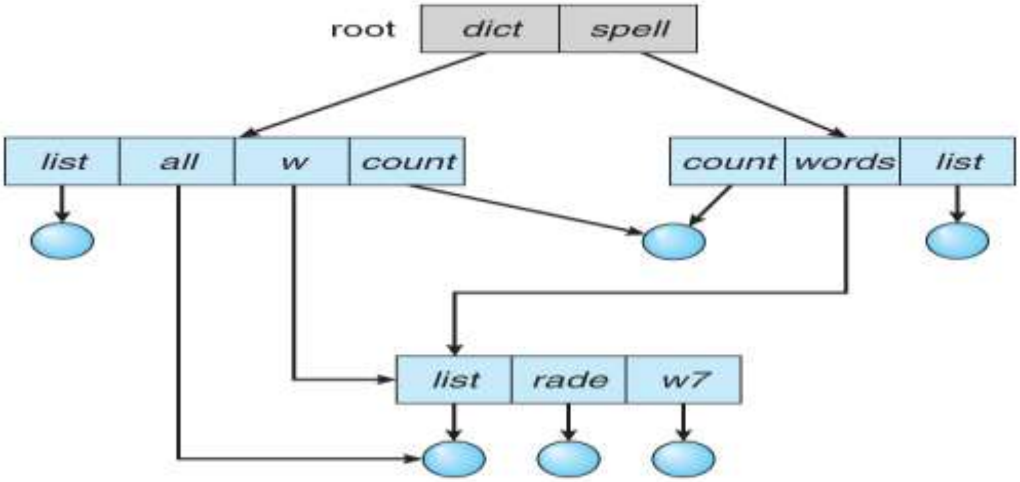


Figure 4-G: Acyclic-graph directory structure

File-System Mounting

- The basic idea behind mounting file systems is to combine multiple file systems into one large tree structure.
- The mount procedure is straightforward. The operating system is given the name of the device and the mount point—the location within the file structure where the file system is to be attached.
- **To illustrate file mounting**, consider the file system depicted in Figure 4-H, where the triangles represent subtrees of directories that are of interest.
- Figure 4-H (a) shows an existing file system, while Figure 4-H (b) shows a un mounted volume residing on /device/dsk.

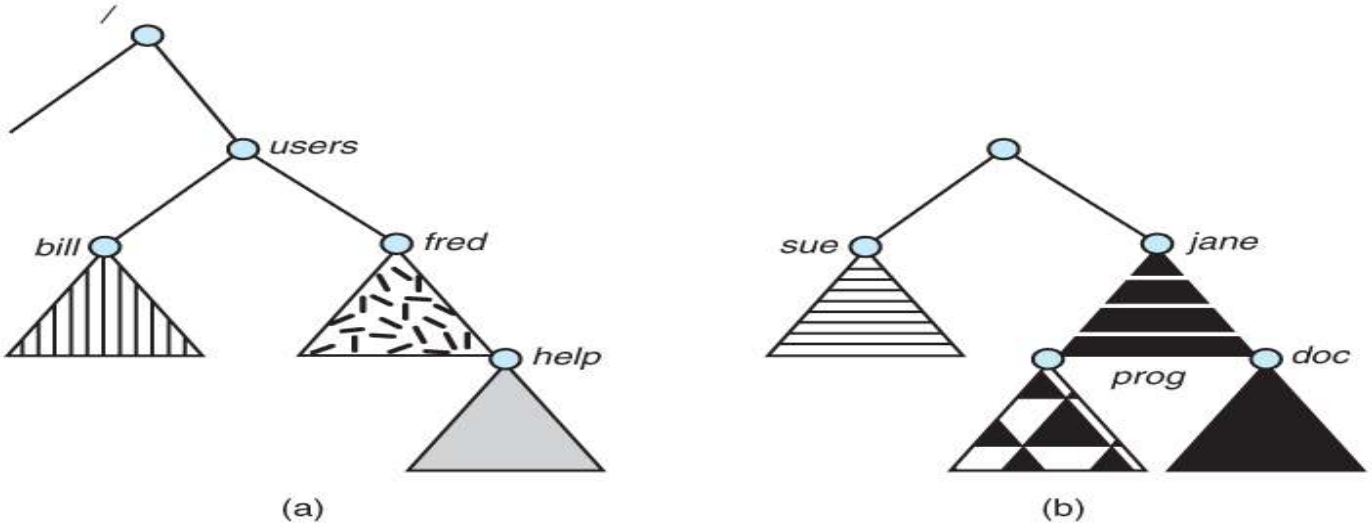


Figure 4-H: File system. (a) Existing system. (b) Un mounted volume.

- At this point, only the files on the existing file system can be accessed. Figure 4-I shows the effects of mounting the volume residing on /device/dsk over /users.

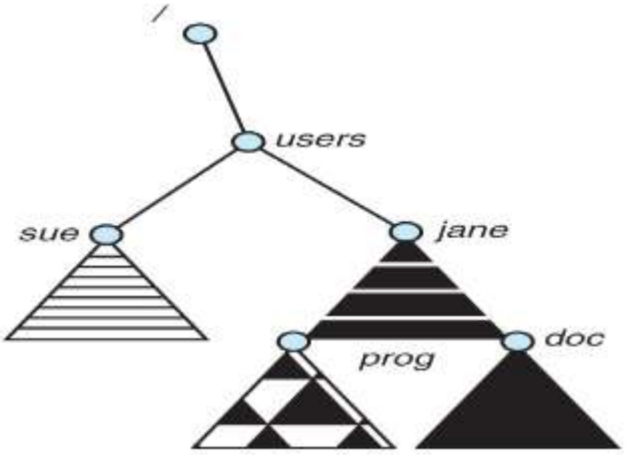


Figure 4-I: Mount point

Protection

- When information is stored in a computer system, we want to keep it safe from physical damage (the issue of reliability) and improper access (the issue of protection).

❖ Types of Access:

- The need to protect files is a direct result of the ability to access files.
- Systems that do not permit access to the files of other users do not need protection.
- Thus, we could provide complete protection by prohibiting access.
- Protection mechanisms provide controlled access by limiting the types of file access that can be made.
  - **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.

❖ Access Control:

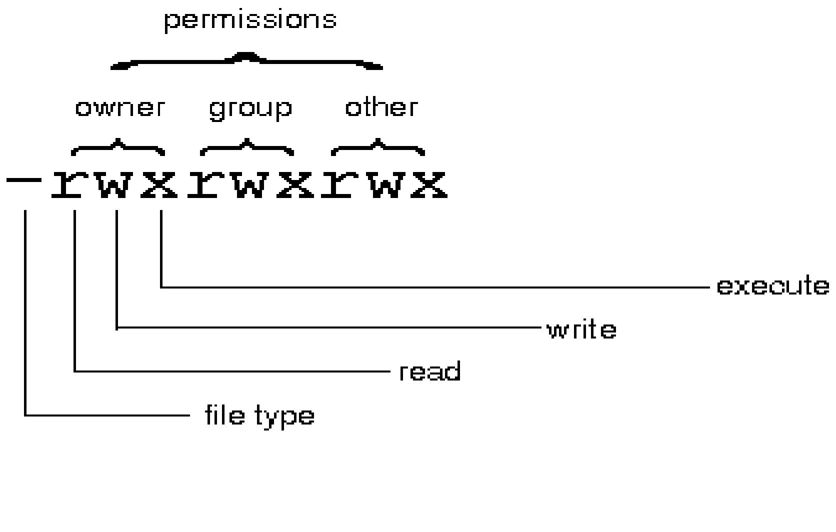
- The most common approach to the protection problem is to make access dependent on the identity of the user. Different users may need different types of access to a file or directory.
- One approach is to have **Access Control Lists, ACL**, which specify exactly what access is allowed or denied for specific users or groups.
- The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access.
- <u>This technique has two undesirable consequences:</u>
  - Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
  - The directory entry, previously of fixed size, now must be of variable size, resulting in more complicated space management.
- These problems can be resolved by use of a condensed version of the access list.
- To condense the length of the access-control list, 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 group, or work group.
  - **Others/ Universe**. All other users in the system constitute the universe.

- To illustrate, consider a person, Sara, who is writing a new book. She has hired three graduate students (Jim, Dawn, and Jill) to help with the project. The text of the book is kept in a file named book.tex.

  The protection associated with this file is as follows:

  - Sara should be able to invoke all operations on the file.
  - Jim, Dawn, and Jill should be able only to read and write the file; they should not be allowed to delete the file.
  - All other users should be able to read, but not write, the file. (Sara is interested in letting as many people as possible read the text so that she can obtain feedback.)
- To achieve such protection, we must create a new group—say, text— with members Jim, Dawn, and Jill. The name of the group, text, must then be associated with the file book.tex, and the access rights must be set in accordance with the policy we have outlined.
- Now consider a visitor to whom Sara would like to grant temporary access to Chapter 1. The visitor cannot be added to the text group because that would give him access to all chapters. Because a file can be in only one group, Sara cannot add another group to Chapter 1.With the addition of access-control-list functionality, though, the visitor can be added to the access control list of Chapter 1.

- UNIX uses a set of 9 access control bits, in three groups of three. These correspond to R, W, and X permissions for each of the **Owner, Group, and Others**. (See "man chmod" for full details.). The RWX bits control the following privileges for ordinary files and directories:



File-System Structure

- Disks provide most of the secondary storage on which file systems are maintained. Two characteristics make them convenient for this purpose:
  1. A disk can be rewritten in place; 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.
- Disks are usually accessed in physical **blocks**, rather than a byte at a time. Block sizes may range from 512 bytes to 4K or larger.
- File systems organize storage on disk drives, and can be viewed as a layered design:

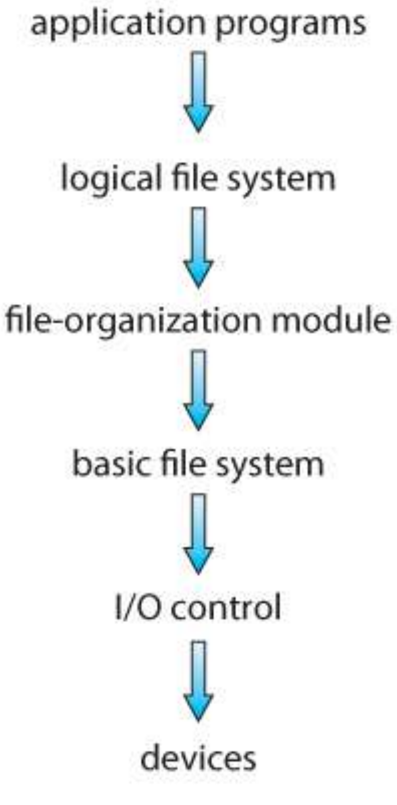


Figure 4-J: Layered file system structure.

- At the lowest layer are the physical devices, consisting of the magnetic media, motors & controls, and the electronics connected to them and controlling them. Modern disk put more and more of the electronic controls directly on the disk drive itself, leaving relatively little work for the disk controller card to perform.

- **I/O Control** consists of **device drivers**, special software programs (often written in assembly) which communicate with the devices by reading and writing special codes directly to and from memory addresses corresponding to the controller card's registers. Each controller card (device) on a system has a different set of addresses (registers, a.k.a. ports) that it listens to, and a unique set of command codes and results codes that it understands.

- The **basic file system** level works directly with the device drivers in terms of retrieving and storing raw blocks of data, without any consideration for what is in each block. Depending on the system, blocks may be referred to with a single block number, or with head-sector-cylinder combinations.

- The **file organization module** knows about files and their logical blocks, and how they map to physical blocks on the disk. In addition to translating from logical to physical blocks, the file organization module also maintains the list of free blocks, and allocates free blocks to files as needed.

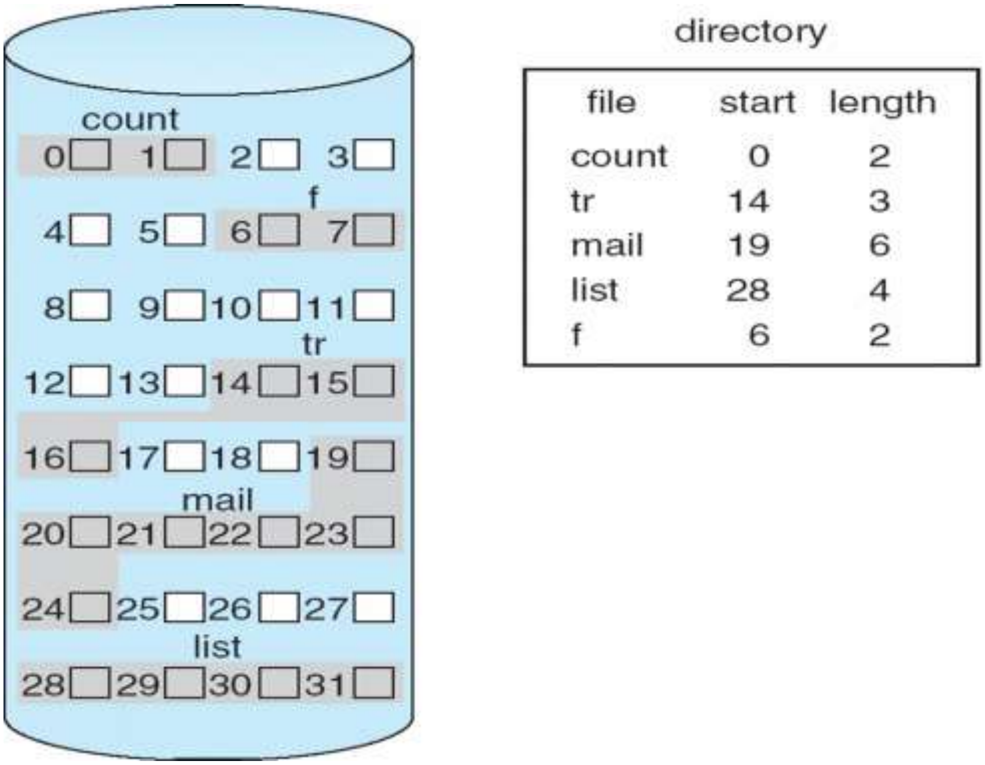
- The **logical file system** deals with all of the meta data associated with a file (UID, GID, mode, dates, etc), i.e. everything about the file except the data itself. This level manages the directory structure and the mapping of file names to **file control blocks, FCBs**, which contain all of the meta data as well as block number information for finding the data on the disk.

FILE Allocation Methods

- ❖ There are three major methods of storing files on disks: contiguous, linked, and indexed.

1. Contiguous Allocation:

- ➢ Contiguous Allocation requires that all blocks of a file be kept together contiguously.
- ➢ Performance is very fast, because reading successive blocks of the same file generally requires no movement of the disk heads.



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

Figure 4-N: Contiguous allocation of disk space.

- ➢ Problems can arise when files grow, or if the exact size of a file is unknown at creation time:
  - Over-estimation of the file's final size **increases external fragmentation** and wastes disk space.
  - If a file grows slowly over a long time period and the total final space must be allocated initially, then a lot of space becomes unusable before the file fills the space.
- ➢ To minimize these drawbacks, some operating systems use a modified contiguous-allocation scheme. Here, a contiguous chunk of space is allocated initially.
- ➢ Then, if that amount proves not to be large enough, another chunk of contiguous space, known as an **extent**, is added. The location of a file’s blocks is then recorded as a location and a block count, plus a link to the first block of the next extent.

2. Linked Allocation:

- **Linked allocation** solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks.
- Linked allocation involves no external fragmentation, does not require pre-known file sizes, and allows files to grow dynamically at any time.
- The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block.
- Another big problem with linked allocation is reliability if a pointer is lost or damaged. Doubly linked lists provide some protection, at the cost of additional overhead and wasted space.
- Allocating **clusters** of blocks reduces the space wasted by pointers, at the cost of internal fragmentation.

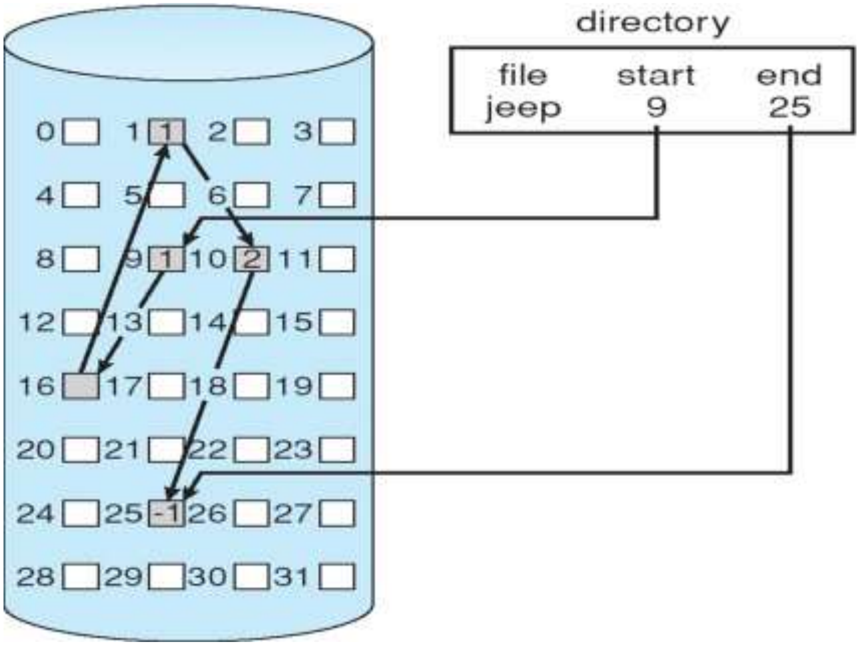


Figure 4-O: Linked allocation of disk space.

3. Indexed Allocation:

- Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation.

However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order

- **Indexed allocation** solves this problem by bringing all the pointers together into one location: the **index block**.

- Each file has its own index block, which is an array of disk-block addresses.
- The i_{th} entry in the index block points to the i_{th} block of the file .Every file must have an index block, so we want the index block to be as small as possible.

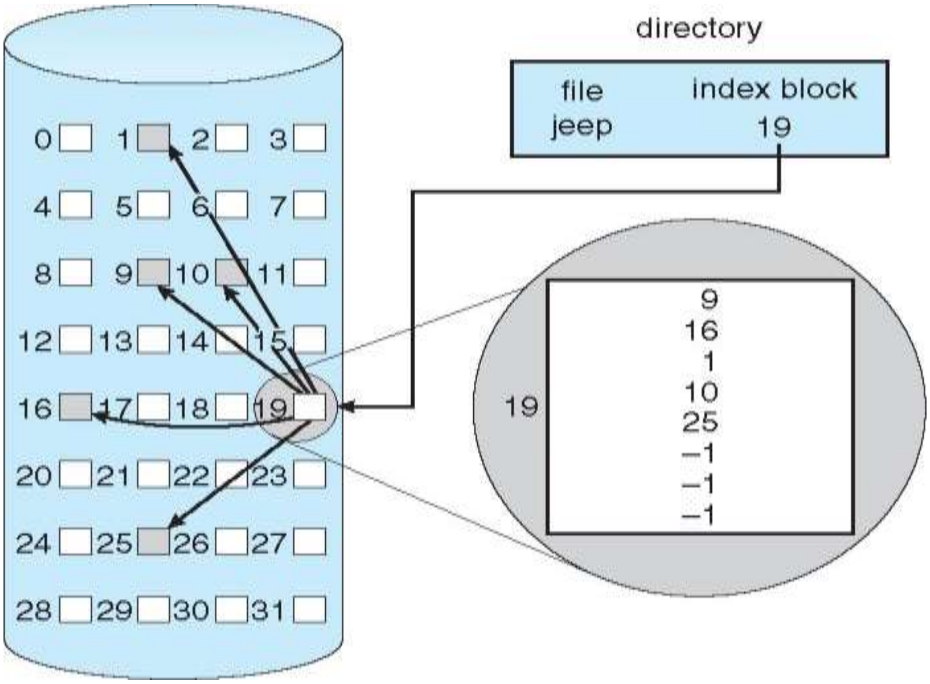


Figure 4-P: Indexed allocation of disk space**.**

- If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue.
- Mechanisms for this purpose include the following:
    - **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.
    - **Multilevel index**.Avariantof 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.
    - **Combined scheme.** This is the scheme used in UNIX inodes, in which the first 12 or so data block pointers are stored directly in the inode, and then singly, doubly, and triply indirect pointers provide access to more data blocks as needed.

Free-Space Management

- Since disk space is limited, we need to reuse the space from deleted files for new files, if possible.
- To keep track of free disk space, the system maintains a **free-space list**.
- The 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.

1. Bit Vector:

- Frequently, the free-space list is implemented as a **bit map** or **bit vector**. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.
- **For 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

001111001111110001100000011100000 ...

- The main advant: it is fast algorithms exist for quickly finding contiguous blocks of a given size.
- The calculation of the block number is

(number of bits per word) × (number of 0-value words) + offset of first 1 bit.

2. Linked List:

- Another approach to free-space management is to link together all the free disk blocks, 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.

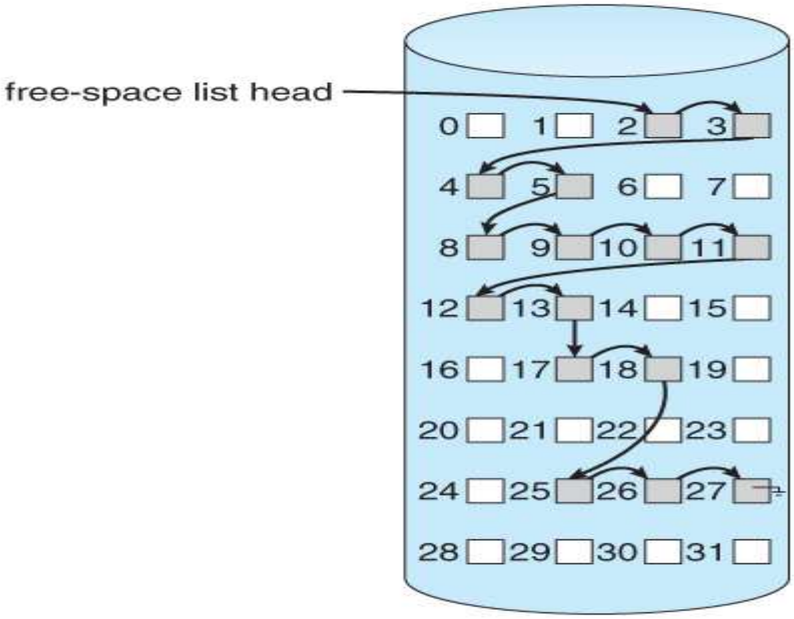


Figure 4-Q: Linked free-space list on disk.

- **For 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 were allocated.
- In this situation, we would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on (Figure 4-Q).
- This scheme is not efficient.

3. **Grouping:**

- A modification of the free-list approach stores the addresses of n free blocks in the first free block.
- The first n−1 of these blocks are actually free. The last block contains the addresses of another n free blocks, and so on.
- The addresses of a large number of free blocks can now be found quickly, unlike the situation when the standard linked-list approach is used.

4. **Counting:**

- When there are multiple contiguous blocks of free space then the system can keep track of the starting address of the group and the number of contiguous free blocks.
- As long as the average length of a contiguous group of free blocks is greater than two this offers a savings in space needed for the free list. (Similar to compression techniques used for graphics images when a group of pixels all the same color is encountered.)

5. **Space Maps :**

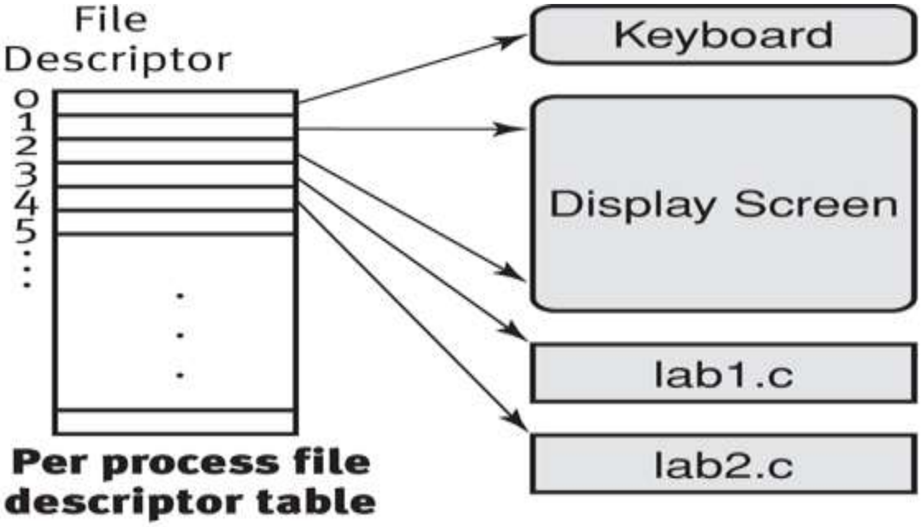
- Sun's **ZFS** file system was designed for HUGE numbers and sizes of files, directories, and even file systems.
- The resulting data structures could be VERY inefficient if not implemented carefully. For example, freeing up a 1 GB file on a 1 TB file system could involve updating thousands of blocks of free list bit maps if the file was spread across the disk.
- ZFS uses a combination of techniques, starting with dividing the disk up into (hundreds of) **metaslabs** of a manageable size, each having their own space map.
- Free blocks are managed using the counting technique, but rather than write the information to a table, it is recorded in a log-structured transaction record. Adjacent free blocks are also coalesced into a larger single free block.
- An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.
- The combination of the in-memory tree and the on-disk log provide for very fast and efficient management of these very large files and free blocks.

System calls for file operations:

File Descriptor: The operating system assigns internally to each opened file a descriptor or an identifier (usually this is a positive integer). When opening or creating a new file the system returns a file descriptor to the process that executed the call. Each application has its own file descriptors.

- By convention, the first three file descriptors are opened at the beginning of each process. The 0 file descriptor identifies the standard input, 1 identifies the standard output and 2 the standard output for errors.
- **File Descriptor table**: File descriptor table is the collection of integer array indices that are file descriptors in which elements are pointers to file table entries. One unique file descriptors table is provided in operating system for each process.

- **File Table Entry:** File table entries is a structure In-memory surrogate for an open file, which is created when process request to opens file and these entries maintains file position.



Standard File Descriptors: When any process starts, then that process file descriptors table's fd(file descriptor) 0, 1, 2 open automatically, (By default) .

- **Read from stdin => read from fd 0** : Whenever we write any character from keyboard, it read from stdin through fd 0 and save to file named /dev/tty.
- **Write to stdout => write to fd 1** : Whenever we see any output to the video screen, it's from the file named /dev/tty and written to stdout in screen through fd 1.
- **Write to stderr => write to fd 2** : We see any error to the video screen, it is also from that file write to stderr in screen through fd 2.

System calls:

1. **create( ):** Used to Create a new empty file.

   Syntax:

   int creat(char *filename, mode_t mode)

Parameters :

- **filename** : name of the file which you want to create
- **mode :** indicates permissions of new file.

Returns :

- return first unused file descriptor (generally 3 when first create use in process beacuse 0, 1, 2 fd are reserved).
- return -1 when error.

How it work in OS

- Create new empty file on disk
- Create file table entry
- Set first unused file descriptor to point to file table entry
- Return file descriptor used, -1 upon failure

Example:

```
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>
int main( )
{
int fd;
printf("\n This will create file");
fd=creat("t1.txt",0777);
printf("\nfd=%d",fd);
if(fd==-1)
{
printf("Error Number %d\n",errno);
}
return 0;
}
```

Output:

```
fd = 3
```

2. **open( ):** Used to Open the file for reading, writing or both.

Syntax :

```
#include<sys/types.h>
#includ<sys/stat.h>
#include <fcntl.h>
int open (const char* Path, int flags ,[ int mode ]);
```

Parameters:

- This function returns the file descriptor or in case of an error -1.
- The number of arguments that this function can have is two or three. The third argument is used only when creating a new file. When we want to open an existing file only two arguments are used.
- The function returns the smallest available file descriptor. This can be used in the following system calls: *read*, *write*, *lseek* and *close*.
- The effective UID or the effective GID of the process that executes the call has to have read/write rights, based on the value of the argument *flags*.
- The file pointer is places on the first byte in the file. The argument *flags* is formed by a bitwise OR operation made on the constants defined in the *fcntl.h* header.

❖ **flags** : How you like to use.

O_RDONLY	- Opens the file for reading.
O_WRONLY	-Opens the file for writing.
O_RDWR	-The file is opened for reading and writing.
O_APPEND	-It writes successively to the end of the file.
O_CREAT	-The file is created in case it did not already exist.
O_TRUNC	-If the file exists all of its content will be deleted.

- The third argument, *mode*, is a bitwise OR made between a combination of two from the following list:
  - S_IRUSR, S_IWUSR, S_IXUSR

        - Owner: *read*, *write*, *execute*.
    - S_IRGRP, S_IWGRP, S_IXGRP
        - Group: *read*, *write*, *execute*.
    - S_IROTH, S_IWOTH, S_IXOTH
        - Others: *read*, *write*, *execute*.

How it works in OS

- Find existing file on disk
- Create file table entry
- Set first unused file descriptor to point to file table entry
- Return file descriptor used, -1 upon failure

Example:

```
// C program to illustrate
// open system call
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>
extern int errno;
int main( )
{
    // if file does not have in directory
    // then file foo.txt is created.
    int fd = open("foo.txt", O_RDONLY | O_CREAT);

    printf("fd = %d/n", fd);

    if (fd ==-1)
    {
        // print which type of error have in a code
        printf("Error Number % d\n", errno);

        // print program detail "Success or failure"
        perror("Program");
    }
    return 0;
}
```

Output:

```
fd = 3
```

3. read( ):

From the file indicated by the file descriptor fd, the read() function reads cnt bytes of input into the memory area indicated by buf. A successful read() updates the access time for the file.

Syntax in C language

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

Parameters

- **fd:** file descripter
- **buf:** buffer to read data from
- **cnt:** length of buffer

Returns: How many bytes were actually read

- return Number of bytes read on success
- return 0 on reaching end of file
- return -1 on error
- return -1 on signal interrupt

Important points

- **buf** needs to point to a valid memory location with length not smaller than the specified size because of overflow.
- **fd** should be a valid file descriptor returned from open( ) to perform read operation because if fd is NULL then read should generate error.
- **cnt** is the requested number of bytes read, while the return value is the actual number of bytes read. Also, sometimes read system call should read less bytes than cnt.

Example:

```
// read system Call
#include<stdio.h>
#include <fcntl.h>
int main()
{
  int fd, sz;
  char *c = (char *) calloc(100, sizeof(char));

  fd = open("f1.txt", O_RDONLY);
  if (fd = = -1)
```

```
    {
    perror("r1");
    exit(1);
    }
   sz = read(fd, c, 10);
   printf("called read(% d, c, 10).  returned that" " %d bytes  were read.\n", fd, sz);
   c[sz] = '\0';
   printf("Those bytes are as follows: % s\n", c);
   return 0;
 }
Upon execution—
Output:
called read(3, c, 10).  returned that 10 bytes  were read.
Those bytes are as follows:
```

Note: In C, the end of a character string is designated by the `\0` character. This is commonly known as the **null terminator**. Almost all string functions declared in the C library under `<string.h>` use this criteria to check or find the end of a string.

- A text file, on the other hand, will not typically have any `\0` characters in it. So, when reading text from a file, you have to *null-terminate* your character buffer before you then print it.

Calloc ( ) function;

Description

The C library function **void *calloc(size_t nitems, size_t size)** allocates the requested memory and returns a pointer to it. The difference in **malloc** and **calloc** is that malloc does not set the memory to zero where as calloc sets allocated memory to zero.

Declaration

Following is the declaration for calloc() function.

```
void *calloc(size_t nitems, size_t size)
```

Parameters

- **nitems** − This is the number of elements to be allocated.

- **size** − This is the size of elements.

Return Value

This function returns a pointer to the allocated memory, or NULL if the request fails.

Example 2:

Suppose that f1.txt consists of the 5 ASCII characters “Linux”. Then what is the output of the following program?

```
// C program to illustrate
// read system Call
#include<stdio.h>
#include<fcntl.h>

int main( )
{
    char c;
    int fd1 = open("f1.txt", O_RDONLY);
    int fd2 = open("f1.txt", O_RDONLY);
    read(fd1, &c, 1);
    read(fd2, &c, 1);
    printf("c = % c\n", c);
    exit(0);
return 0;
}
```

Output:

```
c = L
```

- The descriptors ***fd1*** and ***fd2*** each have their own open file table entry, so each descriptor has its own file position for ***f1.txt***. Thus, the read from ***fd2*** reads the first byte of ***f1.txt***, and the output is **c = L**, not **c = i.**

4. write( ) :

Writes cnt bytes from buf to the file or socket associated with fd. If cnt is zero, write ( ) simply returns 0 without attempting any other action.

Syntax:

```
#include <fcntl.h>

size_t write (int fd, void* buf, size_t cnt);
```

Parameters

- **fd:** file descripter
- **buf:** buffer to write data to
- **cnt:** length of buffer

Returns: How many bytes were actually written

- return Number of bytes written on success
- return 0 on reaching end of file
- return -1 on error
- return -1 on signal interrupt

Important points

- The file needs to be opened for write operations
- **buf** needs to be at least as long as specified by cnt because if buf size less than the cnt then buf will lead to the overflow condition.
- **cnt** is the requested number of bytes to write, while the return value is the actual number of bytes written. This happens when **fd** have a less number of bytes to write than cnt.
- If write( ) is interrupted by a signal, the effect is one of the following:
    - If write( ) has not written any data yet, it returns -1 and sets errno to EINTR.
    - If write( ) has successfully written some data, it returns the number of bytes it wrote before it was interrupted.

Example 1:

```
// C program to illustrate
// write system Call
#include<stdio.h>
#include <fcntl.h>
Int main( )
{
  int sz;

  int fd = open("f1.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  if (fd ==-1)
  {
    perror("r1");
    exit(1);
  }

  sz = write(fd, "hello linux", strlen("hello linux"));

  printf("called write(%d, \"hello linux\", %d). It returned %d\n", fd, strlen("hello linux"), sz);

  close(fd);
  return 0;
}
```

Output:

```
called write(3, "hello linux\n", 11).  it returned 11
```

- Here, when you see in the file foo.txt after running the code, you get a “hello geeks“. If foo.txt file already have some content in it then write system call overwrite the content and all previous content are deleted and only “hello geeks” content will have in the file.

Example 2:

Print “hello world” from the program without use any printf or cout function.

```
// C program to illustrate
// I/O system Calls
#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include<fcntl.h>

int main (void)
{
    int fd[2];
    char buf1[12] = "hello world";
    char buf2[12];

    // assume foobar.txt is already created
    fd[0] = open("f1.txt", O_RDWR);
    fd[1] = open("f1.txt", O_RDWR);

    write(fd[0], buf1, strlen(buf1));
    write(1, buf2, read(fd[1], buf2, 12));

    close(fd[0]);
    close(fd[1]);

    return 0;
}
```

Output:

```
hello world
```

- In this code, buf1 array’s string ***“hello world”*** is first write in to stdin fd[0] then after that this string write into stdin to buf2 array. After that write into buf2 array to the stdout and print output “***hello world***“.

5. close( ):

Tells the operating system you are done with a file descriptor and Close the file which pointed by fd.

Syntax

```
#include <fcntl.h>
 int close(int fd);
```

Parameter

- **fd :**file descriptor

Return

- **0** on success.
- **-1** on error.

How it works in the OS

- Destroy file table entry referenced by element fd of file descriptor table – As long as no other process is pointing to it!
- Set element fd of file descriptor table to **NULL**

Example 1:

```
// C program to illustrate close system Call
#include<stdio.h>
#include <fcntl.h>
int main()
{
    int fd1 = open("f1.txt", O_RDONLY);
    if (fd1 = = -1)
    {
            perror("c1");
            exit(1);
    }
    printf("opened the fd = % d\n", fd1);

    // Using close system Call
    if (close(fd1) = = -1)
    {
            perror("c1");
            exit(1);
    }
    printf("closed the fd.\n");
    return 0;
}
```

Output:

```
opened the fd = 3

closed the fd.
```

Example 2:

```
// C program to illustrate close system Call

#include<stdio.h>

#include<fcntl.h>

int main()
```

```
{
int fd1, fd2;
    // assume that foo.txt is already created
     fd1 = open("foo.txt", O_RDONLY, 0);
    close(fd1);

    // assume that baz.tzt is already created
     fd2 = open("f1.txt", O_RDONLY, 0);

    printf("fd2 = % d\n", fd2);
    exit(0);
    return 0;

}
```

Output:

```
fd2 = 3
```

- Here, In this code first open() returns 3 because when main process created, then fd 0, 1, 2 are already taken by stdin, stdout and stderr. So first unused file descriptor is 3 in file descriptor table. After that in close() system call is free it this 3 file descriptor and then after set 3 file descriptor as null. So when we called second open(), then first unused fd is also 3. So, output of this program is 3.

6. lseek( ):

lseek is a system call that is used to change the location of the read/write pointer of a file descriptor. The location can be set either in absolute or relative terms**.**

Syntax:

```
#include <sys/types.h>
#include <unistd.h>
off_t  lseek(int fd, off_t offset, int whence);
```

Field	Description
int fd	The file descriptor of the pointer that is going to be moved.

off_t offset	The offset of the pointer (measured in bytes).
int whence	The method in which the offset is to be interpreted (relative, absolute, etc.). Legal values for this variable are provided at the end.
return value	Returns the offset of the pointer (in bytes) from the *beginning* of the file. If the return value is -1, then there was an error moving the pointer

Return value:

- Returns the offset of the pointer (in bytes) from the beginning of the file.
- If the return value is -1, then there was an error moving the pointer.

Absolute Path: An *absolute path* is defined as specifying the location of a file or directory from the root directory(/). In other words we can say *absolute path* is a complete path from start of actual filesystem from / directory.

Example:
/home/user/Document/srv.txt
/root/data/dev.jpg
/var/log/messages

All are absolute Path.

Relative Path: *Relative path* is defined as path related to the present working directory(pwd). Suppose I am located in /home/user1 and I want to change directory to /home/user1/Documents. I can use *relative path* concept to change directory to Documents.

Example:
here are two examples for changing directory, 1st by using relative path, 2nd by using absolute path.

```
$ pwd
/home/user1
$cd Documents/  (using relative path)
$pwd
/home/user1/Documents
```

The following is an example using the lseek system call.

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>

int main()
{
    int file=0;
    if((file=open("testfile.txt",O_RDONLY)) < -1)
        return 1;

    char buffer[19];
    if(read(file,buffer,19) != 19)  return 1;
    printf("%s\n",buffer);

    if(lseek(file,10,SEEK_SET) < 0) return 1;

    if(read(file,buffer,19) != 19)  return 1;
    printf("%s\n",buffer);

    return 0;
}
```

The output of the preceding code is:

```
$ cat testfile.txt
This is a test file that will be used
to demonstrate the use of lseek.
$ ./testing
This is a test file
test file that will
```

7. **stat( ):**

- Stat system call is a system call in Linux to check the status of a file such as to check when the file was accessed.

- The stat() system call actually returns file attributes. The file attributes of an inode are basically returned by Stat() function.

Syntax:

```
#include <sys/stat.h>

int stat(const char *path, struct stat *buf)
```

- The return type of the function in int, if the function is executed successfully, 0 is returned if there are any errors, -1 will be returned.
- Here **const char *path** specifies the name of the file. If the path of file is a symbolic link then you need to specify the link instead of file name.
- Then in the function we have a stat structure in which the data or information about the file is stored which uses a pointer named **buf,** which is passed in as a paramteter and filled in during the execution of the call and readable by the user after the call.

Stat structure:

- The stat structure which is defined in <sys/stat.h> header file contains the following fields:

```
struct stat
{
  mode_t        st_mode;
  ino_t         st_ino;
  dev_t         st_dev;
  dev_t         st_rdev;
  nlink_t       st_nlink;
  uid_t         st_uid;
  gid_t         st_gid;
  off_t         st_size;
  struct timspec st_atim;
  struct timspec st_mtim;
  struct timspec st_ctim;
  blksize_t     st_blksize;
  blkcnt_t      st_blocks;
};
```

Description:

1. **st_dev:** It is the ID of device in which we have our file residing currently.
2. **st_rdev:** This field describes that a particular file represents a particular device.
3. **st_ino:** It is the inode number or the serial number of the file. As it is an index number so it should be unique for all files
4. **st_size:** st_size is the size of the file in bytes.
5. **st_atime:** It is the last time or the recent time at which the file was accessed.
6. **st_ctime:** It is the recent time at which the status or the permissions of the file was changed.
7. **st_mtime:** It is the recent time at which the file was modified.
8. **st_blksize:** This field gives the preferred block size for I/O file system which may vary from file to file.

9. **st_blocks:** This field tells the total number of blocks in multiples of 512 bytes.
10. **st_nlink:** This field tells the total number of hard links.
11. **st_uid:** This field indicates the user ID.
12. **st_gid:** This field indicates the group ID.
13. **st_mode:** It indicates the permissions on the file, tells the modes on a file. Following are the flags that should be defined for st_mode field:

Example:

```
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
int main(int argc, char **argv)
{
if(argc!=2)
return 1;

struct stat fileStat;
if(stat(argv[1],&fileStat) < 0)
return 1;

printf("Information for %s\n",argv[1]);
printf("---------------------------\n");
printf("File Size: \t\t%ld bytes\n",fileStat.st_size);
printf("Number of Links: \t%d\n",fileStat.st_nlink);
printf("File inode: \t\t%lu\n",fileStat.st_ino);

printf("File Permissions: \t");
printf( (S_ISDIR(fileStat.st_mode)) ? "d" : "-");
printf( (fileStat.st_mode & S_IRUSR) ? "r" : "-");
printf( (fileStat.st_mode & S_IWUSR) ? "w" : "-");
printf( (fileStat.st_mode & S_IXUSR) ? "x" : "-");
printf( (fileStat.st_mode & S_IRGRP) ? "r" : "-");
printf( (fileStat.st_mode & S_IWGRP) ? "w" : "-");
printf( (fileStat.st_mode & S_IXGRP) ? "x" : "-");
printf( (fileStat.st_mode & S_IROTH) ? "r" : "-");
printf( (fileStat.st_mode & S_IWOTH) ? "w" : "-");
printf( (fileStat.st_mode & S_IXOTH) ? "x" : "-");
printf("\n\n");


return 0;
}
```

Output:

```
student@NNRG310:~/oslab$ cc stat.c

student@NNRG310:~/oslab$ ./a.out read.c

Information for read.c

---------------------------

File Size:              455 bytes

Number of Links:        1

File inode:             794292

File Permissions:       -rw-rw-r--
```