Module 2: File Systems and Management

In the previous module, we emphasized that a computer system processes and stores information. Usually, during processing computers need to frequently access primary memory for instructions and data. However, the primary memory can be used only for only temporary storage of information. This is so because the primary memory of a computer system is volatile. The volatility is evinced by the fact that when we switch off the power the information stored in the primary memory is lost. The secondary memory, on the other hand, is non-volatile. This means that once the user has finished his current activity on a computer and shut down his system, the information on disks (or any other form of secondary memory) is still available for a later access. The non-volatility of the memory enables the disks to store information indefinitely. Note that this information can also be made available online all the time. Users think of all such information as files. As a matter of fact, while working on a computer system a user is continually engaged in managing or using his files in one way or another. OS provides support for such management through a file system. File system is *the* software which empowers users and applications to organize and manage their files. The organization and management of files may involve access, updates and several other file operations. In this chapter our focus shall be on organization and management of files.

2.1 What Are Files?

Suppose we are developing an application program. A program, which we prepare, is a file. Later we may compile this program file and get an object code or an executable. The executable is also a file. In other words, the output from a compiler may be an object code file or an executable file. When we store images from a web page we get an image file. If we store some music in digital format it is an audio file. So, in almost every situation we are engaged in using a file. In addition, we saw in the previous module that files are central to our view of communication with IO devices. So let us now ask again:

What is a file?

Irrespective of the content any organized information is a file.

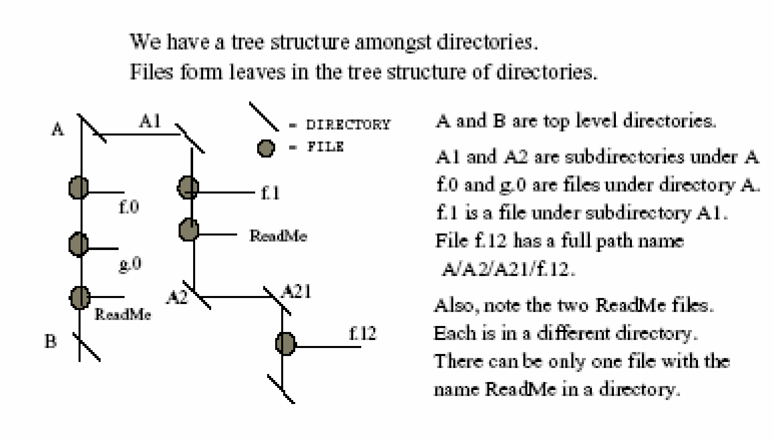
So be it a telephone numbers list or a program or an executable code or a web image or a data logged from an instrument we think of it always as a file. This formlessness and disassociation from content was emphasized first in Unix. The formlessness essentially means that files are arbitrary bit (or byte) streams. Formlessness in Unix follows from the basic design principle: keep it simple. The main advantage to a user is flexibility in organizing files. In addition, it also makes it easy to design a file system. A file system is that software which allows users and applications to organize their files. The organization of information may involve access, updates and movement of information between devices. Later in this module we shall examine the user view of organizing files and the system view of managing the files of users and applications. We shall first look at the user view of files.

User's view of files: The very first need of a user is to be able to access some file he has stored in a non-volatile memory for an on-line access. Also, the file system should be able to locate the file sought by the user. This is achieved by associating an identification for a file i.e. a file must have a name. The name helps the user to identify the file. The file name also helps the file system to locate the file being sought by the user.

Let us consider the organization of my files for the Compilers course and the Operating Systems course on the web. Clearly, all files in compilers course have a set of pages that are related. Also, the pages of the OS system course are related. It is, therefore, natural to think of organizing the files of individual courses together. In other words, we would like to see that a file system supports grouping of related files. In addition, we would like that all such groups be put together under some general category (like COURSES).

This is essentially like making one file folder for the compilers course pages and other one for the OS course pages. Both these folders could be placed within another folder, say COURSES. This is precisely how MAC OS defines its folders. In Unix, each such group, with related files in it, is called a directory. So the COURSES directory may have subdirectories OS and COMPILERS to get a hierarchical file organization. All modern OSs support such a hierarchical file organization. In Figure 2.1 we show a hierarchy of files. It must be noted that within a directory each file must have a distinct name. For instance, I tend to have ReadMe file in directories to give me the information on what is in each directory. At most there can be only one file with the name “ReadMe" in a directory. However, every subdirectory under this directory may also have its own ReadMe file. Unix emphasizes disassociation with content and form. So file names can be assigned any way.

Some systems, however, require specific name extensions to identify file type. MSDOS identifies executable files with a .COM or .EXE file name extension. Software systems like C or Pascal compilers expect file name extensions of .c or .p (or .pas) respectively. In Section 2.1.1 and others we see some common considerations in associating a file name extension to define a file type.



We have a tree structure amongst directories.
Files form leaves in the tree structure of directories.

A and B are top level directories.
A1 and A2 are subdirectories under A
f.0 and g.0 are files under directory A.
f.1 is a file under subdirectory A1.
File f.12 has a full path name
A/A2/A21/f.12.
Also, note the two ReadMe files.
Each is in a different directory.
There can be only one file with the
name ReadMe in a directory.

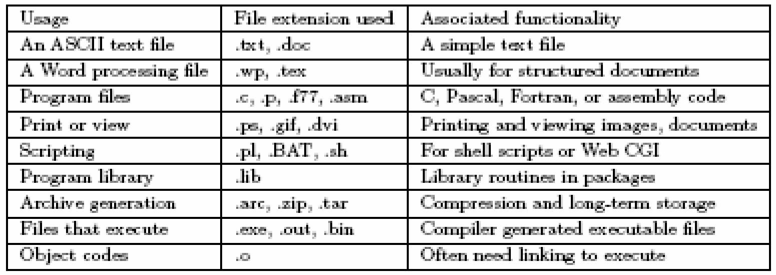
Figure 2.1: Directory and file organisation.

2.1.1 File Types and Operations

Many OSs, particularly those used in personal computers, tend to use a file type information within a name. Even Unix software support systems use standard file extension names, even though Unix as an OS does not require this. Most PC-based OSs associate file types with specific applications that generate them. For instance, a database generating program will leave explicit information with a file descriptor that it has been generated by a certain database program. A file descriptor is kept within the file structure and is often used by the file system software to help OS provide file management services. MAC OS usually stores this information in its resource fork which is a part of its file descriptors.

This is done to let OS display the icons of the application environment in which this file was created. These icons are important for PC users. The icons offer the operational clues as well. In Windows, for instance, if a file has been created using *notepad* or *word* or has been stored from the browser, a corresponding give away icon appears. In fact, the OS assigns it a file type. If the icon has an Adobe sign on it and we double click on it the acrobat reader opens it right away. Of course, if we choose to open any of the files differently, the OS provides us that as a choice (often using the right button).

For a user the extension in the name of a file helps to identify the file type. When a user has a very large number of files, it is very helpful to know the type of a file from its name extensions. In Table 2.1, we have many commonly used file name extensions. PDP-11 machines, on which Unix was originally designed, used an octal 0407 as a magic number to identify its executable files. This number actually was a machine executable jump instruction which would simply set the program counter to fetch the first executable instruction in the file. Modern systems use many magic numbers to identify which application created or will execute a certain file.



Usage	File extension used	Associated functionality
An ASCII text file	.txt, .doc	A simple text file
A Word processing file	.wp, .tex	Usually for structured documents
Program files	.c, .p, .f77, .asm	C, Pascal, Fortran, or assembly code
Print or view	.ps, .gif, .dvi	Printing and viewing images, documents
Scripting	.pl, .BAT, .sh	For shell scripts or Web CGI
Program library	.lib	Library routines in packages
Archive generation	.arc, .zip, .tar	Compression and long-term storage
Files that execute	.exe, .out, .bin	Compiler generated executable files
Object codes	.o	Often need linking to execute

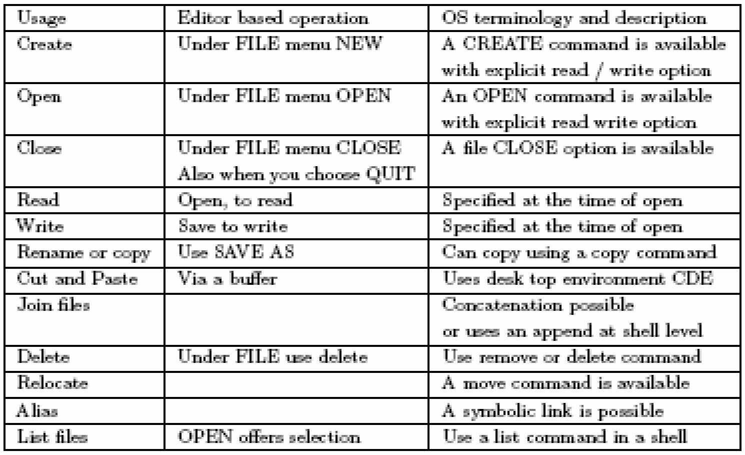
Table 2.1: File extension and its context of use.

In addition to the file types, a file system must have many other pieces of information that are important. For instance, a file system must know at which location a file is placed in the disk, it should know its size, when was it created, i.e. date and time of creation.

In addition, it should know who owns the files and who else may be permitted access to *read, write* or *execute*. We shall next dwell upon these operational issues.

File operations: As we observed earlier, a file is any organized information. So at that level of abstraction it should be possible for us to have some logical view of files, no matter how these may be stored. Note that the files are stored within the secondary storage. This is a physical view of a file. A file system (as a layer of software) provides a logical view of files to a user or to an application. Yet, at another level the file system offers the physical view to the OS. This means that the OS gets all the information it needs to physically locate, access, and do other file based operations whenever needed. Purely from an operational point of view, a user should be able to create a file. We will also assume that the creator owns the file. In that case he may wish to save or store this file. He should be able to read the contents of the file or even write into this file. Note that a user needs the write capability to update a file. He may wish to display or rename or append this file. He may even wish to make another copy or even delete this file. He may even wish to operate with two or more files. This may entail cut or copy from one file and paste information on the other.

Other management operations are like indicating who else has an authorization of an access to *read* or *write* or *execute* this file. In addition, a user should be able to move this file between his directories. For all of these operations the OS provides the services. These services may even be obtained from within an application like mail or a utility such as an editor. Unix provides a visual editor vi for ASCII file editing. It also provides another editor *sed* for stream editing. MAC OS and PCs provide a range of editors like SimpleText.



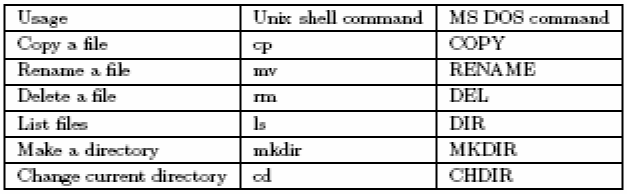
Usage	Editor based operation	OS terminology and description
Create	Under FILE menu NEW	A CREATE command is available with explicit read / write option
Open	Under FILE menu OPEN	An OPEN command is available with explicit read write option
Close	Under FILE menu CLOSE Also when you choose QUIT	A file CLOSE option is available
Read	Open, to read	Specified at the time of open
Write	Save to write	Specified at the time of open
Rename or copy	Use SAVE AS	Can copy using a copy command
Cut and Paste	Via a buffer	Uses desk top environment CDE
Join files		Concatenation possible or uses an append at shell level
Delete	Under FILE use delete	Use remove or delete command
Relocate		A move command is available
Alias		A symbolic link is possible
List files	OPEN offers selection	Use a list command in a shell

Table 2.2: File operations.

With multimedia capabilities now with PCs we have editors for audio and video files too. These often employ MIDI capabilities. MAC OS has Claris works (or Apple works) and MSDOS-based systems have Office 2000 suite of packaged applications which provide the needed file oriented services. See Table 2.2 for a summary of common file operations.

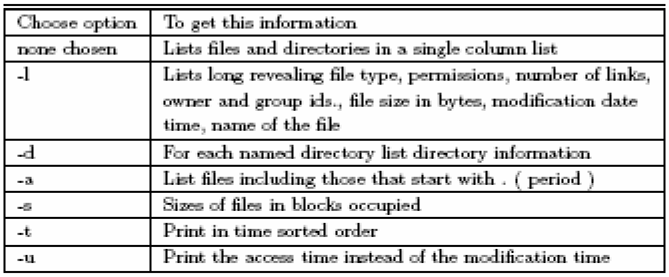
For illustration of many of the basic operations and introduction of shell commands we shall assume that we are dealing with ASCII text files. One may need information on file sizes. More particularly, one may wish to determine the number of lines, words or characters in a file. For such requirements, a shell may have a suite of word counting programs. When there are many files, one often needs longer file names. Often file names may bear a common stem to help us categorize them. For instance, I tend to use “prog” as a prefix to identify my program text files. A programmer derives considerable support through use of regular expressions within file names. Use of regular expressions enhances programmer productivity in checking or accessing file names. For instance, prog* will mean all files prefixed with stem prog, while my file? may mean all the files with prefix my file followed by at most one character within the current directory. Now that we have seen the file operations, we move on to services. Table 2.3 gives a brief description of the file-oriented services that are made available in a Unix OS. There are similar MS DOS commands. It is a very rewarding experience to try these commands and use regular expression operators like ? and * in conjunction with these commands.

Later we shall discuss some of these commands and other file-related issues in greater depth. Unix, as also the MS environment, allows users to manage the organization of their files. A command which helps viewing current status of files is the *ls* command in Unix (or the *dir* command in MS environment). This command is very versatile. It helps immensely to know various facets and usage options available under the *ls* command.



Usage	Unix shell command	MS DOS command
Copy a file	cp	COPY
Rename a file	mv	RENAME
Delete a file	rm	DEL
List files	ls	DIR
Make a directory	mkdir	MKDIR
Change current directory	cd	CHDIR

Table 2.3: File oriented services.



Choose option	To get this information
none chosen	Lists files and directories in a single column list
-l	Lists long revealing file type, permissions, number of links, owner and group ids., file size in bytes, modification date time, name of the file
-d	For each named directory list directory information
-a	List files including those that start with . (period)
-s	Sizes of files in blocks occupied
-t	Print in time sorted order
-u	Print the access time instead of the modification time

Table 2.4: Unix *ls* command options.

The *ls* command: Unix's *ls* command which lists files and subdirectories in a directory is very revealing. It has many options that offer a wealth of information. It also offers an insight in to what is going on with the files i.e. how the file system is updating the information about files in “inode” which is a short form for an index node in Unix. We shall learn more about inode in Section 2.4. In fact, it is very rewarding to study *ls* command in all its details. Table 2.4 summarizes some of the options and their effects.

Using regular expressions: Most operating systems allow use of regular expression operators in conjunction with the commands. This affords enormous flexibility in usage of a command. For instance, one may input a partial pattern and complete the rest by a * or a ? operator. This not only saves on typing but also helps you when you are searching a file after a long time gap and you do not remember the exact file names completely. Suppose a directory has files with names like Comp_page_1.gif, Comp_page_2.gif and Comp_page_1.ps and Comp_page_2.ps. Suppose you wish to list files for page_2. Use a partial name like *ls* C*p*2 or even *2* in *ls* command. We next illustrate the use of operator ?. For instance, use of *ls* my file? in *ls* command will list all files in the current directory with prefix my file followed by at most one character.

Besides these operators, there are command options that make a command structure very flexible. One useful option is to always use the *-i* option with the *rm* command in Unix. A *rm -i* my files* will interrogate a user for each file with prefix my file for a possible removal. This is very useful, as by itself *rm* my file* will remove all the files without any further prompts and this can be very dangerous. A powerful command option within the *rm* command is to use a *-r* option. This results in recursive removal, which means it removes all the files that are linked within a directory tree. It would remove files in the current, as well as, subdirectories all the way down. One should be careful in choosing the options, particularly for remove or delete commands, as information may be lost irretrievably.

It often happens that we may need to use a file in more than one context. For instance, we may need a file in two projects. If each project is in a separate directory then we have two possible solutions. One is to keep two copies, one in each directory or to create a symbolic link and keep one copy. If we keep two unrelated copies we have the problem of consistency because a change in one is not reflected in the other. The symbolic link helps to alleviate this problem. Unix provides the *ln* command to generate a link anywhere regardless of directory locations with the following structure and interpretation: *ln* fileName pseudonym.

Now fileName file has an alias in pseudonym too. Note that the two directories which share a file link should be in the same disk partition. Later, in the chapter on security, we shall observe how this simple facility may also become a security hazard.

2.2 File Access Rights

After defining a fairly wide range of possible operations on files we shall now look at the file system which supports all these services on behalf of the OS. In the preamble of this chapter we defined a file system as that software which allows users and applications to organize and manage their files. The organization of information may involve access, updates, and movement of information between devices. Our first major concern is access.

Access permissions: Typically a file may be accessed to read or write or execute.

The usage is determined usually by the context in which the file is created. For instance, a city bus timetable file is created by a transport authority for the benefit of its customers.

So this file may be accessed by all members of public. While they can access it for a read operation, they cannot write into it. An associated file may be available to the supervisor who assigns duties to drivers. He can, not only read but also write in to the files that assign drivers to bus routes. The management at the transport authority can read, write and even execute some files that generate the bus schedules. In other words, a file system must manage access by checking the access rights of users. In general, access is managed by keeping access rights information for each file in a file system.

Who can access files?: Unix recognizes three categories of users of files, e.g. user (usually the user who created it and owns it), the group, and others. The owner may be a person or a program (usually an application or a system-based utility). The notion of “group” comes from software engineering and denotes a team effort. The basic concept is that users in a group may share files for a common project. Group members often need to share files to support each other's activity. Others has the connotation of public usage as in the example above. Unix organizes access as a three bit information for each i.e. owner, group, and others. So the access rights are defined by 9 bits as *rwx rwx rwx* respectively for owner, group and others. The *rwx* can be defined as an octal number too. If all bits are set then we have a pattern 111 111 111 (or 777 in octal) which means the owner has read, write, and execute rights, and the group to which he belongs has also read, write and execute rights, and others have read, write and execute rights as well. A pattern of 111 110 100 (or 764 octal, also denoted as *rwx rw- r--*) means the owner has read, write, and execute permissions; the group has read and write permissions but no execute permission and others have only the read permission. Note that Unix group permissions are for all or none. Windows 2000 and NTFS permit a greater degree of refinement on a group of users. Linux allows individual users to make up groups.

2.3 File Access and Security Concerns

The owner of a file can alter the permissions using the *chmod* command in Unix. The commonly used format is *chmod* octalPattern fileName which results in assigning the permission interpreted from the octalPattern to the file named fileName. There are other alternatives to *chmod* command like *chmod* changePattern fileName where changePattern may be of the form *go-rw* to denote withdrawal of read write permission from group and others. Anyone can view all the currently applicable access rights using a *ls* command in Unix with *-l* option. This command lists all the files and subdirectories of the current directory with associated access permissions.

Security concerns: Access permissions are the most elementary and constitute a fairly effective form of security measure in a standalone single user system. In a system which may be connected in a network this can get very complex. We shall for now regard the access control as our first line of security. On a PC which is a single-user system there is no security as such as anyone with an access to the PC has access to all the files.

Windows 2000 and XP systems do permit access restriction amongst all the users of the system. These may have users with system administrator privileges. In Unix too, the super-user (root) has access to all the files of all the users. So there is a need for securing files for individual users. Some systems provide security by having a password for files. However, an enhanced level of security is provided by encryption of important files. Most systems provide some form of encryption facility. A user may use his own encryption key to encrypt his file. When someone else accesses an encrypted file he sees a garbled file which has no pattern or meaning. Unix provides a crypt command to encrypt files.

The format of the crypt command is:

crypt EncryptionKey < inputFileName > outputFileName

The EncryptionKey provides a symmetric key, so that you can use the same key to retrieve the old file (simply reverse the roles of inputFileName and outputFileName) In Section 2.4 we briefly mention about audit trails which are usually maintained in syslog files in Unix systems. In a chapter on security we shall discuss these issues in detail. So far we have dealt with the logical view of a file. Next, we shall address the issues involved in storage and management of files.

2.4 File Storage Management

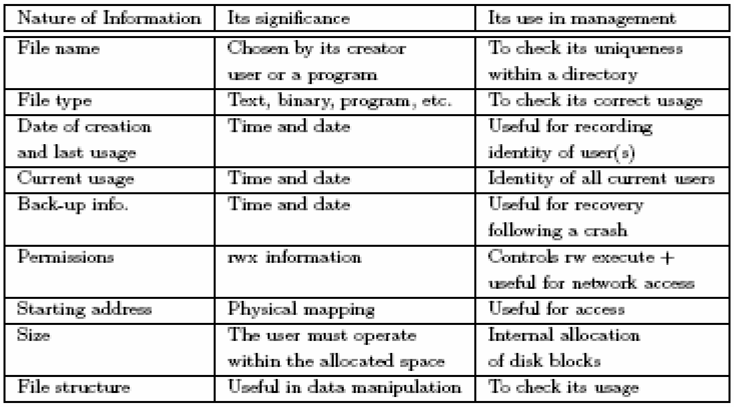
An operating system needs to maintain several pieces of information that can assist in management of files. For instance, it is important to record when the file was last used and by whom. Also, which are the current processes (recall a process is a program in execution) accessing a particular file. This helps in management of access. One of the important files from the system point of view is the audit trail which indicates who accessed when and did what. As mentioned earlier, these trails are maintained in syslog files under Unix. Audit trail is very useful in recovering from a system crash. It also is useful to detect un-authorized accesses to the system. There is an emerging area within the security community which looks up the audit trails for clues to determine the identity of an intruder.

In Table 2.5 we list the kind of information which may be needed to perform proper file management. While Unix emphasizes formlessness, it recognizes four basic file types internally. These are ordinary, directory, special, and named. Ordinary files are those that are created by users, programs or utilities. Directory is a file type that organizes files hierarchically, and the system views them differently from ordinary files. All IO communications are conducted as communications to and from special files. For the present we need not concern ourselves with named files. Unix maintains much of this information in a data structure called inode which is a short form for an index node. All file management operations in Unix are controlled and maintained by the information in the inode structure.

We shall now briefly study the structure of inode.

2.4.1 Inode in Unix

In Table 2.6 we describe typical inode contents. Typically, it offers all the information about access rights, file size, its date of creation, usage and modification. All this information is useful for the management in terms of allocation of physical space, securing information from malicious usage and providing services for legitimate user needs to support applications.



Nature of Information	Its significance	Its use in management
File name	Chosen by its creator user or a program	To check its uniqueness within a directory
File type	Text, binary, program, etc.	To check its correct usage
Date of creation and last usage	Time and date	Useful for recording identity of user(s)
Current usage	Time and date	Identity of all current users
Back-up info.	Time and date	Useful for recovery following a crash
Permissions	rwx information	Controls rw execute + useful for network access
Starting address	Physical mapping	Useful for access
Size	The user must operate within the allocated space	Internal allocation of disk blocks
File structure	Useful in data manipulation	To check its usage

Table 2.5: Information required for management of files.

Typically, a disk shall have inode tables which point to data blocks. In Figure 2.2 we show how a disk may have data and inode tables organized. We also show how a typical Unix-based system provides for a label on the disk.

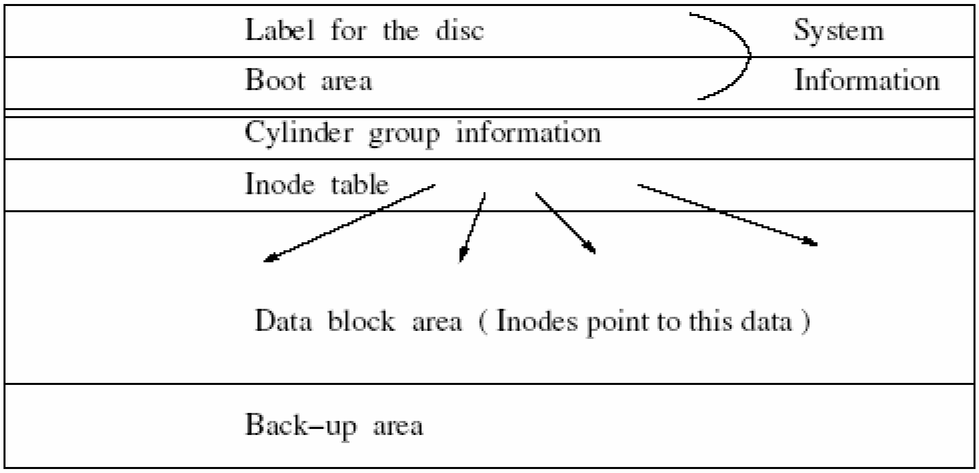
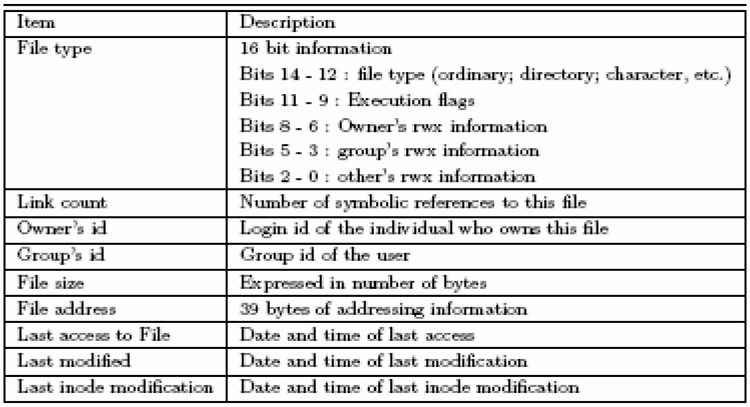

Label for the disc
Boot area
System Information
Cylinder group information
Inode table
Data block area (Inodes point to this data)
Back–up area

Figure 2.2: Organisation of inodes.



Item	Description
File type	16 bit information Bits 14 - 12 : file type (ordinary; directory; character, etc.) Bits 11 - 9 : Execution flags Bits 8 - 6 : Owner's rwx information Bits 5 - 3 : group's rwx information Bits 2 - 0 : other's rwx information
Link count	Number of symbolic references to this file
Owner's id	Login id of the individual who owns this file
Group's id	Group id of the user
File size	Expressed in number of bytes
File address	39 bytes of addressing information
Last access to File	Date and time of last access
Last modified	Date and time of last modification
Last inode modification	Date and time of last inode modification

Table 2.6: Inode structure in Unix.

2.4.2 File Control Blocks

In MS environment the counterpart of inode is FCB, which is a short form for File Control Block. The FCBs store file name, location of secondary storage, length of file in bytes, date and time of its creation, last access, etc. One clear advantage MS has over Unix is that it usually maintains file type by noting which application created it. It uses extension names like *doc, txt, dll,* etc. to identify how the file was created. Of course, notepad may be used to open any file (one can make sense out of it when it is a text file). Also, as we will see later (in Sections 2.6 and 2.7), MS environment uses a simple chain of clusters which is easy to manage files.

2.5 The Root File System

At this stage it would be worthwhile to think about the organization and management of files in the root file system. When an OS is installed initially, it creates a root file system. The OS not only ensures, but also specifies how the system and user files shall be distributed for space allocation on the disk storage. Almost always the root file system has a directory tree structure. This is just like the users file organization which we studied earlier in Figure 2.1. In OSs with Unix flavors the root of the root file system is a directory. The root is identified by the directory `/'. In MS environment it is identified by `n'. The root file system has several subdirectories. OS creates disk partitions to allocate files for specific usages. A certain disk partition may have system files and some others may have other user files or utilities. The system files are usually programs that are executable with .bin in Unix and .EXE extension in MS environment.

Under Unix the following convention is commonly employed.

- Subdirectory usr contain shareable binaries. These may be used both by users and the system. Usually these are used in read-only mode.
- Under subdirectories bin (found at any level of directory hierarchy) there are executables. For instance, the Unix commands are under /usr/bin. Clearly, these are shareable executables.
- Subdirectory sbin contains some binaries for system use. These files are used during boot time and on power-on.
- Subdirectories named lib anywhere usually contain libraries. A lib subdirectory may appear at many places. For example, as we explain a little later the graphics

library which supports the graphics user interface (GUI) uses the X11 graphics library, and there shall be a lib subdirectory under directory X11.

- Subdirectory etc contains the host related files. It usually has many subdirectories to store device, internet and configuration related information. Subdirectory hosts stores internet addresses of hosts machines which may access this host. Similarly, config subdirectory maintains system configuration information and inet subdirectory maintains internet configuration related information. Under subdirectory dev, we have all the IO device related utilities.
- Subdirectories mnt contain the device mount information (in Linux).
- Subdirectories tmp contain temporary files created during file operation. When you use an editor the OS maintains a file in this directory keeping track of all the edits. Clearly this is its temporary residence.
- Subdirectories var contain files which have variable data. For instance, mail and system log information keeps varying over time. It may also have subdirectories for spools. Spools are temporary storages. For instance, a file given away for printing may be spooled to be picked up by the printer. Even mails may be spooled temporarily.
- All X related file support is under a special directory X11. One finds all X11 library files under a lib directory within this directory.
- A user with name u name would find that his files are under /home/u name. This is also the home directory for the user u name.
- Subdirectories include contain the C header include files.
- A subdirectory marked as yp (a short form for yellow pages) has network information. Essentially, it provides a database support for network operations.

One major advantage of the root file system is that the system knows exactly where to look for some specific routines. For instance, when we give a command, the system looks for a path starting from the root directory and looks for an executable file with the command name specified (usually to find it under one of the bin directories). Users can customize their operational environment by providing a definition for an environment variable PATH which guides the sequence in which the OS searches for the commands. Unix, as also the MS environment, allows users to manage the organization of their files.

One of the commands which helps to view the current status of files is the *ls* command in Unix or the command dir in MS environment.

2.6 Block-based File Organization

Recall we observed in chapter 1 that disks are bulk data transfer devices (as opposed to character devices like a keyboard). So data transfer takes place from disks in blocks as large as 512 or 1024 bytes at a time. Any file which a user generates (or retrieves), therefore, moves in blocks. Each operating system has its own block management policy. We shall study the general principles underlying allocation policies. These policies map each linear byte stream into disk blocks. We consider a very simple case where we need to support a file system on one disk. Note a policy on storage management can heavily influence the performance of a file system (which in turn affects the throughput of an OS). File Storage allocation policy: Let us assume we know apriori the sizes of files before their creation. So this information can always be given to OS before a file is created. Consequently, the OS can simply make space available. In such a situation it is possible to follow a pre-allocation policy: find a suitable starting block so that the file can be accommodated in a contiguous sequence of disk blocks. A simple solution would be to allocate a sequence of contiguous blocks as shown in Figure 2.3.

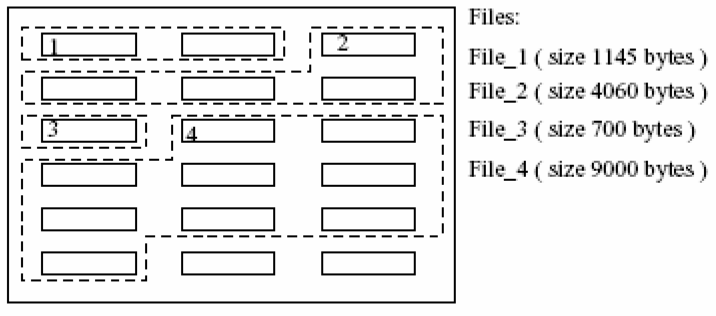

1
2
3
4
Files:
File_1 (size 1145 bytes)
File_2 (size 4060 bytes)
File_3 (size 700 bytes)
File_4 (size 9000 bytes)

Figure 2.3: Contiguous allocation.

The numbers 1, 2, 3 and 4 denote the starting blocks for the four files. One clear advantage of such a policy is that the retrieval of information is very fast. However, note that pre-allocation policy requires apriori knowledge. Also, it is a static policy. Often users' needs develop over time and files undergo changes. Therefore, we need a dynamic policy.

Chained list Allocation : There are two reasons why a dynamic block allocation policy is needed. The first is that in most cases it is not possible to know apriori the size of a file being created. The second is that there are some files that already exist and it is not easy to find contiguous regions. For instance, even though there may be enough space in the disk, yet it may not be possible to find a single large enough chunk to accommodate an incoming file. Also, users' needs evolve and a file during its lifetime undergoes changes. Contiguous blocks leave no room for such changes. That is because there may be already allocated files occupying the contiguous space.

In a dynamic situation, a list of free blocks is maintained. Allocation is made as the need arises. We may even allocate one block at a time from a free space list. The OS maintains a chain of free blocks and allocates next free block in the chain to an incoming file. This way the finally allocated files may be located at various positions on the disk. The obvious overhead is the maintenance of chained links. But then we now have a dynamically allocated disk space. An example is shown in Figure 2.4.

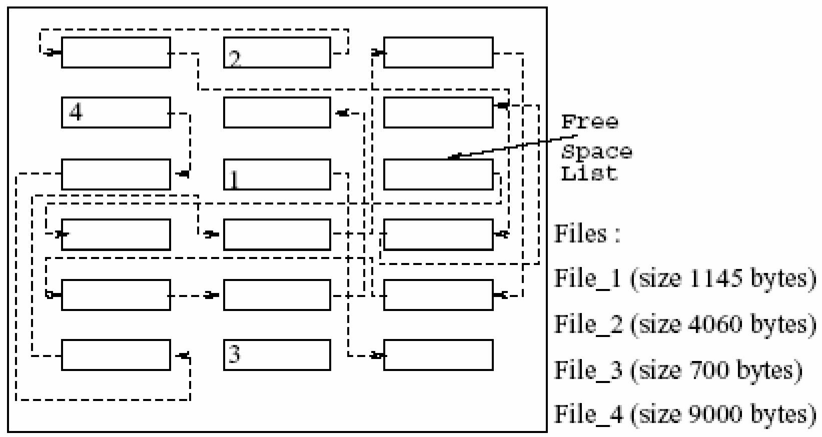


Figure 2.4: Chained allocation.

Chained list allocation does not require apriori size information. Also, it is a dynamic allocation method. However, it has one major disadvantage: random access to blocks is not possible.

Indexed allocation: In an indexed allocation we maintain an index table for each file in its very first block. Thus it is possible to obtain the address information for each of the blocks with only one level of indirection, i.e. from the index. This has the advantage that there is a direct access to every block of the file. This means we truly operate in the direct access mode at the block level.

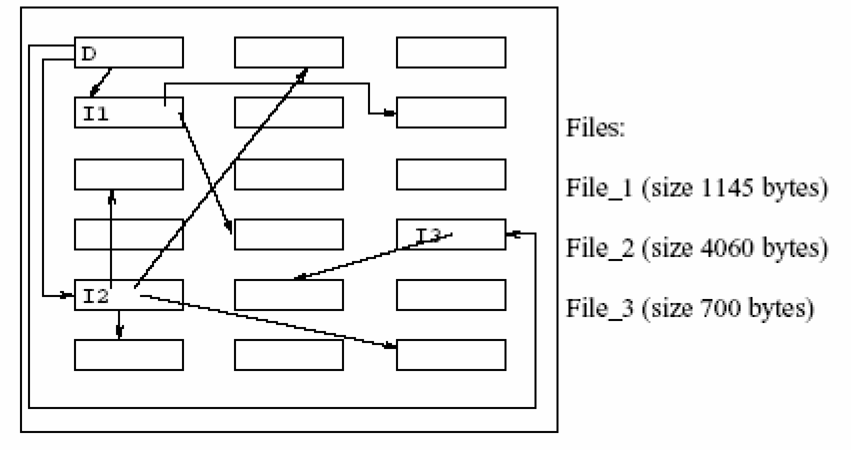


Figure 2.5: Indexed allocation.

In Figure 2.5 we see that File-2 occupies four blocks. Suppose we use a block I2 to store the starting addresses of these four blocks, then from this index we can access any of the four parts of this file. In a chained list arrangement we would have to traverse the links. In Figure 2.5 we have also shown D to denote the file's current directory. All files have their own index blocks. In terms of storage the overhead of storing the indices is more than the overhead of storing the links in the chained list arrangements. However, the speed of access compensates for the extra overhead.

Internal and external Fragmentation: In mapping byte streams to blocks we assumed a block size of 1024 bytes. In our example, a file (File 1) of size 1145 bytes was allocated two blocks. The two blocks together have 2048 bytes capacity. We will fill the first block completely but the second block will be mostly empty. This is because only 121 bytes out of 1024 bytes are used. As the assignment of storage is by blocks in size of 1024 bytes the remaining bytes in the second block can not be used. Such non-utilization of space caused internally (as it is within a file's space) is termed as internal fragmentation. We note that initially the whole disk is a free-space list of connected blocks. After a number of file insertions and deletion or modifications the free-space list becomes smaller in size. This can be explained as follows. For instance, suppose we have a file which was initially spread over 7 blocks. Now after a few edits the file needs only 4 blocks. This space of 3 blocks which got released is now not connected anywhere. It is not connected with the free storage list either. As a result, we end up with a hole of 3 blocks which is not connected anywhere. After many file edits and operations many such holes of various sizes get created. Suppose we now wish to insert a moderately large sized file thinking that adequate space should be still available. Then it may happen that the free space list has shrunk so much that enough space is not available. This may be because there are many unutilized holes in the disk. Such non-utilization, which is outside of file space, is regarded as external fragmentation. A file system, therefore, must periodic all perform an operation to rebuild free storage list by collecting all the unutilized holes and linking them back to free storage list. This process is called compaction. When you boot a system, often the compaction gets done automatically. This is usually a part of file system management check. Some run-time systems, like LISP and Java, support periodic automatic compaction. This is also referred to as run-time garbage collection.

2.7 Policies In Practice

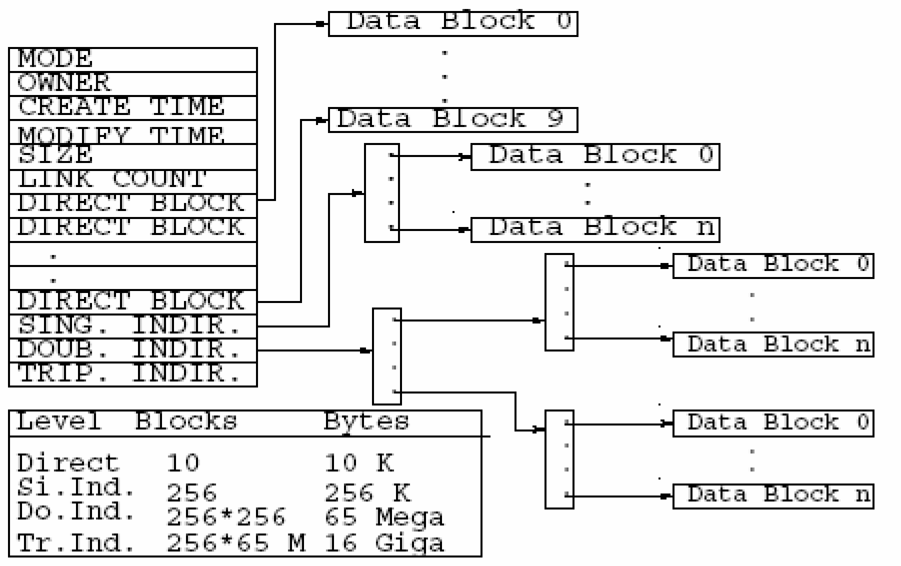
MS DOS and OS2 (the PC-based systems) use a FAT (file allocation table) strategy. FAT is a table that has entries for files for each directory. The file name is used to get the starting address of the first block of a file. Each file block is chain linked to the next block till an EOF (end of file) is stored in some block. MS uses the notion of a cluster in place of blocks, i.e. the concept of cluster in MS is same as that of blocks in Unix. The cluster size is different for different sizes of disks. For instance, for a 256 MB disk the cluster may have a size of 4 KB and for a disk with size of 1 GB it may be 32 KB. The formula used for determining the cluster size in MS environment is disk-size/64K.

FAT was created to keep track of all the file entries. To that extent it also has the information similar to the index node in Unix. Since MS environment uses chained allocation, FAT also maintains a list of “free" block chains. Earlier, the file names under MS DOS were restricted to eight characters and a three letter extension often indicating the file type like BAT or EXE, etc. Usually FAT is stored in the first few blocks of disk space.

An updated version of FAT, called FAT32, is used in Windows 98 and later systems. FAT32 additionally supports longer file names and file compression. File compression may be used to save on storage space for less often used files. Yet another version of the Windows is available under the Windows NT. This file system is called NTFS. Rather than having one FAT in the beginning of disk, the NTFS file system spreads file tables throughout the disks for efficient management. Like FAT32, it also supports long file names and file compression. Windows 2000 uses NTFS. Other characteristics worthy of note are the file access permissions supported by NTFS.

Unix always supported long file names and most Unix based systems such as Solaris and almost all Linux versions automatically compress the files that have not been used for long. Unix uses indexed allocation. Unix was designed to support truly large files. We next describe how large can be large files in Unix.

Unix file sizes: Unix was designed to support large-scale program development with team effort. Within this framework, it supports group access to very large files at

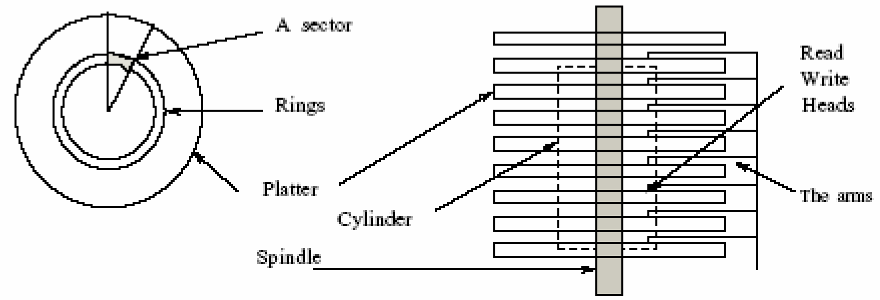


Level	Blocks	Bytes
Direct	10	10 K
Si.Ind.	256	256 K
Do.Ind.	256*256	65 Mega
Tr.Ind.	256*65 M	16 Giga

Figure 2.6: Storage allocation in Unix.

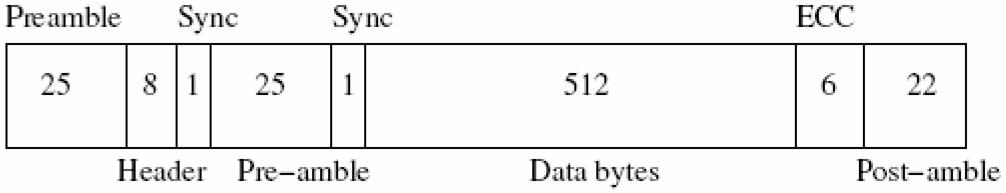
very high speeds. It also has a very flexible organization for files of various sizes. The information about files is stored in two parts. The first part has information about the mode of access, the symbolic links, owner and times of creation and last modification. The second part is a 39 byte area within the inode structure. These 39 bytes are 13, 3 byte address pointers. Of the 39 bytes, first 10 point to the first 10 blocks of a file. If the files are longer then the other 3, 3 byte addresses are used for indirect indexing. So the 11th 3 byte address points to a block that has pointers to real data. In case the file is still larger then the 12th 3 byte address points to an index. This index in turn points to another index table which finally point to data. If the files are still larger then the 13th 3 byte address is used to support a triple indirect indexing. Obviously, Unix employs the indexed allocation. In Figure 2.6 we assume a data block size of 1024 bytes. We show the basic scheme and also show the size of files supported as the levels of indirection increase. **Physical Layout of Information on Media:** In our discussions on file storage and management we have concentrated on logical storage of files. We, however, ignored one very important aspect. And that concerns the physical layout of information on the disk media. Of course, we shall revisit aspects of information map on physical medium later in the chapter on IO device management. For now, we let us examine Figures 2.7 and 2.8 to see how information is stored, read, and written in to a disk.

In Figure 2.7, tracks may be envisaged as rings on a disk platter. Each ring on a platter is capable of storing 1 bit along its width. These 1 bit wide rings are broken into sectors, which serve as blocks. In Section 2.6 we essentially referred to these as blocks.



Note that the rings on the disk platters on the spindle form a cylinder. Since all heads are on a particular ring at the same time, so it is easy to organise information on a cylinder. The information is stored in the sectors that can be identified on the rings. sectors are seperated from each other. All sectors can hold equal amount of information.

Figure 2.7: Information storage organisation on disks.



The numbers are in bytes

Figure 2.8: Information storage in sectors.

This break up into sectors is necessitated because of the physical nature of control required to let the system recognize, where within the tracks blocks begin in a disk. With disks moving at a very high speed, it is not possible to identify individual characters as they are laid out. Only the beginning of a block of information can be detected by hardware control to initiate a stream of bits for either input or output. The read-write heads on the tracks read or write a stream of data along the track in the identified sectors. With multiple disks mounted on a spindle as shown in Figure 2.7, it helps to think of a cylinder formed by tracks that are equidistant from the center. Just imagine a large number of tracks, one above the other, and you begin to see a cylinder. These cylinders can be given contiguous block sequence numbers to store information. In fact, this is desirable because then one can access these blocks in sequence without any additional head movement in a head per track disk. The question of our interest for now is: where is inode (or FAT block) located and how it helps to locate the physical file which is mapped on to sectors on tracks which form cylinders.

2.7.1 Disk Partitions

Disk-partitioning is an important notion. It allows a better management of disk space. The basic idea is rather simple. If you think of a disk as a large space then simply draw some boundaries to keep things in specific areas for specific purposes. In most cases the disk partitions are created at the time the disc is formatted. So a formatted disk has information about the partition size.

In Unix oriented systems, a physical partition of a disk houses a file system. Unix also allows creating a logical partition of disk space which may extend over multiple disk drives. In either case, every partition has its own file system management information.

This information is about the files in that partition which populate the file system. Unix ensures that the partitions for the system kernel and the users files are located in different partitions (or file systems). Unix systems identify specific partitions to store the root file system, usually in root partition. The root partition may also co-locate other system functions with variable storage requirements which we discussed earlier in section 2.5. The user files may be in another file system, usually called home. Under Linux, a proc houses all the executable processes.

Under the Windows system too, a hard disk is partitioned. One interesting conceptual notion is to make each such partition that can be taken as a logical drive. In fact, one may have one drive and by partitioning, a user can make the OS offer a possibility to write into each partition as if it was writing in to a separate drive. There are many third-party tools for personal computer to help users to create partitions on their disks. Yet another use in the PC world is to house two operating system, one in each partition. For instance, using two partitions it is possible to have Linux on one and Windows on another partition in the disk. This gives enormous flexibility of operations. Typically, a 80 GB disk in modern machines may be utilized to house Windows XP and Linux with nearly 40 GB disk available for each.

Yet another associated concept in this context, is the way the disk partitions are mounted on a file system. Clearly, a disk partition, with all its contents, is essentially a set of organized information. It has its own directory structure. Hence, it is a tree by itself. This tree gets connected to some node in the overall tree structure of the file system and forks out. This is precisely what mounting means. The partition is regarded to be mounted in the file system. This basic concept is also carried to the file servers on a network. The network file system may have remote partitions which are mounted on it. It offers seamless file access as if all of the storage was on the local disk. In modern systems, the file servers are located on networks somewhere without the knowledge of the user. From a user's standpoint all that is important to note is that as a user, his files are a part of a large tree structure which is a file system.

2.7.2 Portable storage

There are external media like tapes, disks, and floppies. These storage devices can be physically ported. Most file systems recognize these as on-line files when these are mounted on an IO device like a tape drive or a floppy drive. Unix treats these as special files. PCs and MAC OS recognize these as external files and provide an icon when these are mounted.

In this chapter we have covered considerable ground. Files are the entities that users deal with all the time. Users create files, manage them and seek system support in their file management activity. The discussion here has been to help build up a conceptual basis and leaves much to be covered with respect to specific instructions. For specifics, one should consult manuals. In this very rapidly advancing field, while the concept does not change, the practice does and does at a phenomenal pace.