Chapter: 2. UNIX Files

2.1 Introduction

In UNIX everything can be treated as files. Hence files are building blocks of UNIX operating system. When you execute a command in UNIX, the UNIX kernel fetches the corresponding executable file from a file system, loads its instruction text to memory and creates a process to execute the command.

2.2 UNIX / POSIX file Types

The different types of files available in UNIX / POSIX are:

- Regular files. Example: All .exe files, C, C++, PDF Document files.
- Directory files. Example: Folders in Windows.
- Device files. Example: Floppy, CDROM, Printer.
- FIFO files. Example: Pipes.
- Link Files (only in UNIX) Example: alias names of a file, Shortcuts in Windows..

2.2.1 Regular files

It is a text or binary file. For Example: All exe files, text, Document files and program files. Created by using the commands vi, vim , emacs or by using cat.
For example using cat:

Syntax: $cat > filename

```
$ cat > newf
Hello, this is the file created by using the command cat.
^d                    // terminates input from keyboard
$
```

We can delete the file(s) by using the command *rm.*

Syntax: $ rm file(s)
For example: $rm newf

2.2.2 Directory file

A *directory* is a file that contains information to associate other files with names; these associations are called *links* or *directory entries*. Sometimes, people speak of "files in a directory", but in reality, a directory only contains pointers to files, not the files themselves. Hence is like a folder in windows that contains other files, including other subdirectories. *mkdir* is the command used for creating the directory.

Syntax: $mkdir directory_name(s)
For example: $mkdir newdir

rmdir is a command, which is used to delete a directory(s). To remove the directory that directory should be empty.

Syntax: $rmdir directory_name(s)
For example: $rmdir newdir

2.2.3 Device files

Actually these are physical devices such as printers, tapes, floppy devices CDROMs, hard disks and terminals.
There are two types of device files: Block and Character device files.

1. **Block Device files:** A physical device that transmits block of data at a time.
   For example: floppy devices CDROMs, hard disks.
2. **Character Device files:** A physical device that transmits data character by character.
   For example: Line printers, modems etc.

mknod is the command used for creating the device files.
Syntax: # mknod Device_Filename DeviceFile_Type Major_ Number Minor_Number
Example1: # mknod /dev/cdev c 120 20
This creates character device file called *cdev* with major number 120 and minor number 20.
Example2: # mknod /dev/bdev b 225 15
This creates block device file called *bdev* with major number 225 and minor number 15

A major device number is an index to a kernel table that contains the addresses of all device driver functions known to the system.

A minor device number is an integer value to be passed as an argument to a device driver function when it is called. The minor device number tells the device driver function what actual physical device it is taking to and the input output buffering scheme to be used for data transfer.

2.2.4 FIFO file

It is a special pipe device file, which provides a temporary buffer for two or more processes to communicate by writing data to and reading data from the buffer. The data in the buffer is accessed in a by a first-in-first-out manner, hence the file is called a FIFO. These are having name hence they called named pipes.

mkfifo is the command used for creating a *fifo* file(s).
$mkfifo /usr/fifo1
or by using *mknod* with the option *p*
#mknod /usr/fifo1 p
rm is the command used for removing fifo files.

2.2.5 Link files

Links are nothing but file names. UNIX allows a file to have more than one name and yet maintain single copy on the disk. Changes made to one link are visible in all other links. Hence links just like reference variables in C++. These are not supported by POSIX.
There are two types of Links (1) hard links (2) soft links.

Hard Links : It is a UNIX path or file name, by default files are having only one hard link.
ln is the command used for creating hard links.
Syntax: $ln existing_file_name link_file_name
For example: *$ ln abc abcl*

After this file user may refer to the same file by either *abc* or *abcl*.

```
$ls –li abc abcl
167 -rw-r--r-- 2 karan karan 50 DEC 10 12:25 abc
167 -rw-r--r-- 2 karan karan 50 DEC 10 12:25 abcl
```

For each *ln*, on the same file the hard link count is incremented by 1, and inode numbers are same for all the links. Links are ordinary files, remove links by using *rm* command, each time hard link count is decremented by 1.

For Example after remove the file *abc* the hard link count is decremented by 1 as shown below.

```
$rm abc
$ls –li abcl
167 -rw-r--r-- 1 karan karan 50 DEC 10 12:25 abcl
```

Symbolic Links : Symbolic links are called soft links. These are created in the same manner as hard links, but it requires –s option to the *ln* command. These are just like shortcuts in windows.

Syntax: *$ln -s existing_file_name link_file_name*
For example: $ln –s abc abcl

After this file user may refer to the same file by either *abc* or *abcl*. But the hard link count is remains same and inode numbers, sizes are different. The size of the link file is number of characters in the original file name or path name.

```
$ ls –li abc abcl
267 -rw-r--r--  1  karan  karan  50  DEC  10  1:25  abc
260 lrwxrwxrwx  1  karan  karan  3   DEC  10  1:25  abcl → abc
```

Note: *After remove the file abc in the above example the association is broken between abc and abcl. After wards suppose if you create the file with name abc, the abc will get new inode number hence there is no link at all even after creating the file with the original name. Hence, both the files are entirely different in case of hard links.*
However, in case of soft links if you delete the file abc, the link file abcl still references to the non-existing file abc. After creating the new file with the same i.e. abc the link file references to this new version of the file.

2.2.5.1 Limitations of Hard links

1. Users cannot create hard links for directories unless they have super user privileges.
2. Users cannot create hard links on a file system that references files on a different system.

2.2.5.2 Differences between Hard links and Symbolic Links

Hard Link	Soft Links
1. Do not creates new inode.	1. Creates new inode.
2.Cannot link directories unless super user privileges.	2. Can link directories.
3. Cannot link file across file systems.	3. Can link files across file systems.
4. Increase the hard link count by one.	4. Does not change the hard link count.
5. Always refer to the old file only, means hard links can be broken by removal of one or more links.	5.Always reference to the latest version of the files to which they link.

2.3 The UNIX and POSIX File System

File System: Is a hierarchical arrangement of directories and files. Everything starts in the directory called *root*, denoted by '/'. Each intermediate node in a file system tree is a directory file. The leaf nodes of a file system tree are either empty directory or other types of files.

File Names: In order to open a connection to a file, or to perform other operations such as deleting a file, you need some way to refer to the file. Nearly all files have names that are strings. These strings are called *file names*. You specify the file name to say which file you want to open or operate on. The only two characters cannot be appearing in a file names are the slash character (/) & the NULL character. The slash separates the file names that form a pathname & the NULL character terminates a pathname. Hence, the legal file names characters are: alphabets (A - Z, a - z), digits (0 - 9) and underscore (_).

Two file names are automatically created after creating new directory are called:**.** (dot) and **. .** (dot-dot) refers to the current and parent directory respectively. Some UNIX systems restrict a file name to 14 characters, but some versions extended this limit to 255 characters.

Hence, file name may not exceed NAME_MAX characters and the total number of characters in a path name may not be exceeding PATH_MAX in UNIX. In POSIX _POSIX_NAME_MAX and _POSIX_PATH_MAX respectively.

Path Name: A sequence of zero or more filenames separated by slashes and optionally starting with a slash (/) character forms a path name. A path name that begins with a slash called an absolute pathname (Example: */home/karan/citech/cse/xyz.c*), otherwise it called as relative path name (Example: $cd .. , $ ls . , $ cat cse/xyz.c).

Working Directory: Each process has associated with it a directory, called its *current working directory* or simply *working directory*, which is used in the resolution of relative file names. When you log in and begin a new session, your working directory is initially set to the home directory associated with your login account in the system user database. Users can change the working directory using shell commands like *cd*.

Home Directory: When we log in, the working directory is set to home directory, called as Login directory.

More Examples for path name:

`/usr`	the file named *usr*, in the root directory.
`/a/b`	the file named *b*, in the directory named *a* in the root directory.
`a`	the file named *a*, in the current working directory.
`/a/./b`	this is the same as `/a/b`.
`./a`	the file named *a*, in the current working directory.
`../a`	the file named *a*, in the parent directory of the current working directory.

File Table: The UNIX kernel has a file table that keeps track of all opened files in the system. This table stores the file pointer, access modes(r-read, w-write, rw-read write) and reference count.

File Descriptor Table: When a user executes a command, process is created by the kernel to carryout the command execution. Each process has its own data structure, which, among other things is a file descriptor table. The file descriptor table has OPEN_MAX entries and it records all opened files. This table holds the integer value, known as the file descriptor of the file opened by the process.

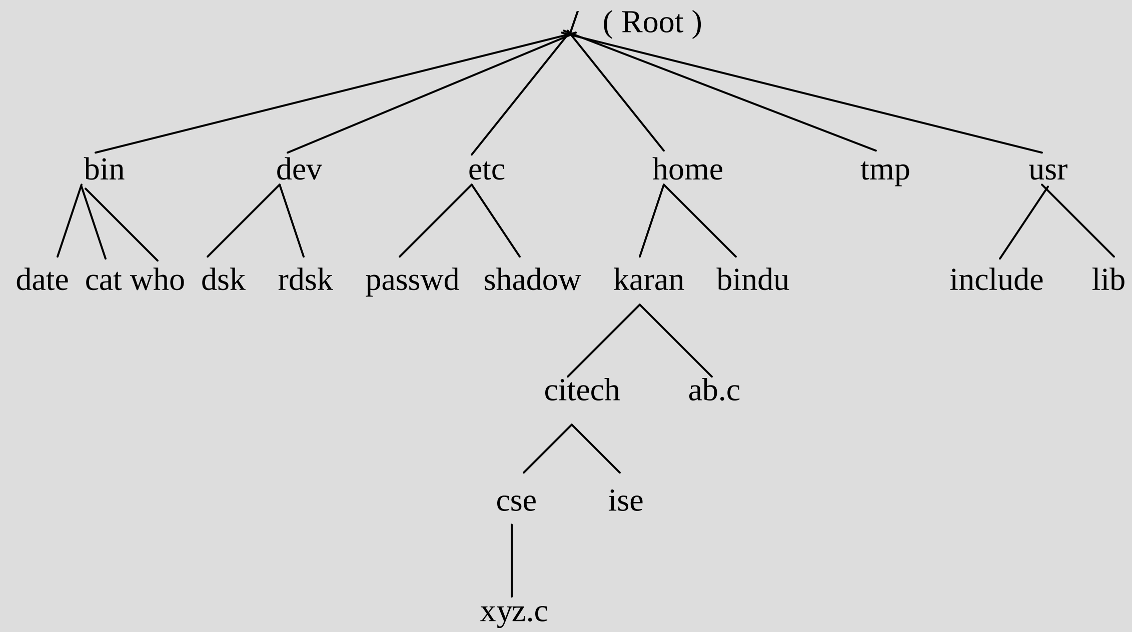

Figure: **The UNIX File System**

/	:	Root directory
/bin	:	Stores all most all the commands like cat, date, who, cp, etc.
/dev	:	Stores all the character and device files.
/etc	:	Stores system administrative commands and files.
/etc/group	:	Stores all group information.
/etc/passwd	:	Stores all users' information.
/etc/shadow	:	Stores user passwords.
/home	:	Stores all the user/login names.
/tmp	:	Stores temporary files created by the programs.
/usr/include	:	Stores standard header files.
/usr/lib	:	Stores standard libraries.

2.4 The UNIX and POSIX Attributes

Both UNIX and POSIX.1 maintain a common set of attributes for each file in file system. Almost all the attributes of file are listed by using *ls* command with –l option. As shown below: When the user *karan* invoke the command from his home directory the *ls* command displays the following output.

```
$ls –li *
File type and Permissions   Owner Name     File Size In bytes           File Name

125   –rw-r-xr--     1     karan     karan     25    DEC  25  10:40   ab.c
150   drw-r-xr--     2     karan     karan     40    DEC  20  11:20   citech

Inode Number     Links           Group Name   Last Modification Date & Time
```

Attribute	Meaning
1. Inode Number	An identification number of the file.
2. File Type	Type of File.
3. Permissions	The file access permission for Owner Group and Others.
4. Hard Links	Number of hard links of a file.
5. Owner Name	The file Owner or User Name.
6. Group Name	The group to which User or Owner Belongs to.
7. File Size	The file size in bytes.
8. Last Access Time	The time, the file was last accessed.
9. Last Modification Time	The time, the file was last modified.
10.Last Change Time	The time, the file Access Permission, Owner, Group or Hard link count was last changed.
11. File System Id	The file system id where the file is stored.

In addition to the above attributes, the UNIX system also stores the Major and Minor numbers for each Device file.

All the above attributes are assigned to the file by the kernel when it is created. Some of these attributes will stay unchanged for the entire life of the file, whereas others may change as the file is being used.

The attributes that are constant for any file are:

- File type
- File Inode number
- File System Id
- Major and Minor numbers

Other attributes are changed by the following UNIX Commands or by using System calls.

Command	System Call	Attributes Changed
chmod	chmod	Changes access permission, last change time.
chown	chown	Changes UID, Last Change time.
chgrp	chown	Changes GID, Last change time.
ln	link	Increase hard link count.
touch	utime	Changes last access time and modification time.
rm	unlink	Decrease the hard link count by one. If the hard link count is zero the file will be permanently removed from the disk.
vi, emacs	Changes file size, last access time and last modification time.	

2.5 Inodes in Unix

The mapping of file name and inode numbers is done via directory files. A directory file contains a list of file names and their respective inode numbers for all files stored in that

Every file system contains a block called inode block, this block contains a table called inode table. All above said attributes of a file and directory are stored in this table as a record (is an individual entry or row in an inode table) except the name of a file or directory. The inode is accessed by a number called an i-number (inode number), it is an unique number in a file system. directory.

Inode Number	File name

Example: The contents of the directory *cse* are as follows.

120	.
140	..
200	xyz.c

2.6 Application Program Interfaces to Files

Both UNIX and POSIX systems provide an application interface similar to files as follows:

1. Files are identified by pathnames.
2. Files must be created before they can be used
   The UNIX commands and corresponding system calls to create the various types of files are as follows:

File type		Command	System call
Regular files		vi, emacs, ex etc	open, create
Directory files		mkdir	mkdir
FIFO files		mkfifo	mkfifo, mknod
Device files		mknod	mknod
Link files	Hard Link	ln	link
	Soft Link	ln -s	symlink

3. Files must be opened before they can be accessed by application programs. UNIX and Posix.1 defines *open* API to open any files. A process may open at most OPEN_MAX or _POSIX_OPEN_MAX files of any types at any one time.
4. The *read* and *write* API can be used to read data from and write data to opened files.
5. File attributes can be queried by the *stat* or *fstat* API.
6. File attributes can be changed by the *chmod*, *chown*, *utime* and *link* API.
7. File hard link can be removed by the *unlink* API.

2.7 UNIX Kernel supports for file / Kernel Data structure for file manipulation

When ever process calls the *open* function call to open a file for read/write, the kernel will resolve the path name to the file inode, if the open call fails and return a -1 to the process.

If however the file inode is accessible to the process, the process is proceed to establish a path from an entry in the file descriptor table, through a file table onto the inode table for the file being opened.

The Steps involved in this process are:

Step 1: The kernel will search the process file descriptor table and look for first unused entry, if an entry is found, that entry will be designated to reference the file. The index to the entry will be returned to the process (via the return value of open API) as the file descriptor of the opened file.

Step 2: The kernel scan the file table in its kernel space to find an unused entry that can be assigned to reference the file.

If an unused entry is found, the following events will occur.

a. The process's file table entry will be set to point to this file table entry.

b. The file table entry will be set to point to the inode table entry where the inode record of the file is stored.
c. The file table entry will contain the current file pointer of the open file. This is an offset from the beginning of he file where the next read or write operation will occur.
d. The file table entry will contain open mode that specifies that the file is open for read-only, write-only or read-write etc. The open mode is specified from the calling process, as an argument to the open function call.
e. The reference count in the file table entry is set to 1. The reference count keeps track of how many file descriptors from any process are referencing the entry.
f. The reference count of the in-memory inode of the file is increased by 1. This count specifies how many file table entries are pointing to that inode.

If either step1 or step 2 fails, the open function will return with a -1 failure status, no file descriptor table or file table entry will be allocated.

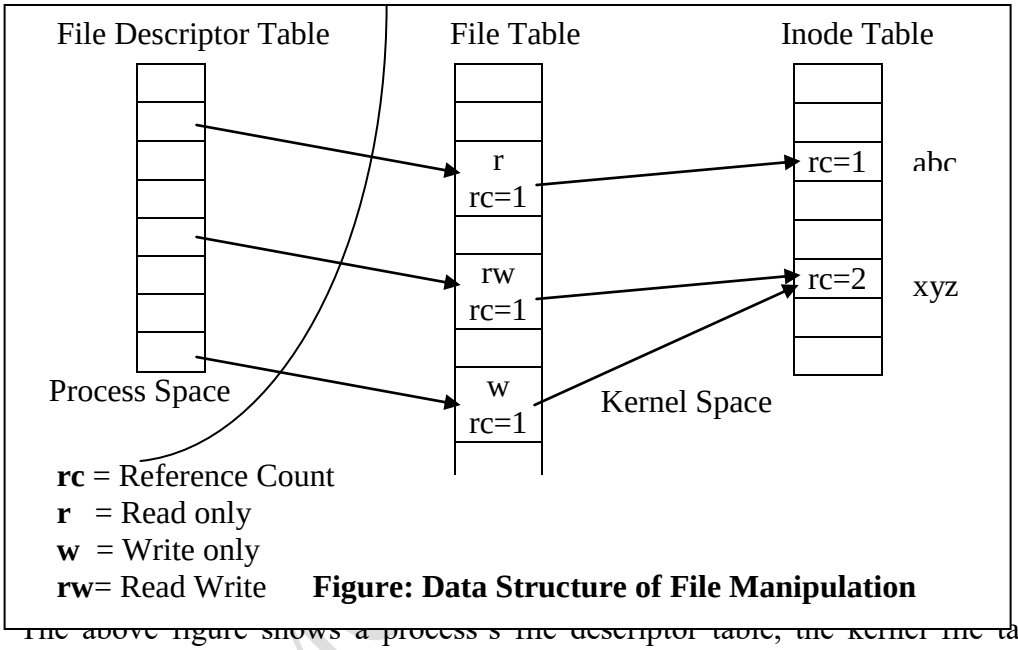

Figure: Data Structure of File Manipulation

The above figure shows a process's file descriptor table, the kernel file table and the inode after the process has opened three files: *abc* for read only, and *xyz* for read- write and *xyz* again for write only.

The reference count value in file table entry is usually 1, but a process may use the *dup (dup2)* function to make multiple file descriptors table entries point to the same file table entry.

The reference count in a file inode table entry is specifies how many file table entries are pointing to the file inode record. If the count is not zero, it means that one or more processes are currently opening the file for access.

Once an open call succeeds, the process can use file descriptor for future reference. Specifically, when the process attempts to read (or write) data from (to the) the file, it will use the file descriptor as the first argument to the read (write) system call. The kernel will use the file descriptor to index the process's file table entry of the opened file; it then checks the file table entry data to make sure that the file is opened with the appropriate mode to allow the requested read (write) operation.

If the read (write) operation is found compatible with the file's open mode, the kernel will use the pointer specified in the file table entry to access the files' inode record and also it will use file pointer stored in the file table entry to determine where the read(or write) operation

should occur in the file. Finally the kernel checks the file type in the inode record and invokes an appropriate driver function to initiate the actual data transfer with a physical device.

When a process calls the function *close* to close an opened file, the following sequence of events will occur.

1. The kernel sets the corresponding file descriptor table entry to be unused.
2. It decrements the reference count in the corresponding file table entry by 1. If the reference count is still non-zero go to step 6.
3. The file table entry is marked as unused.
4. The reference count in the corresponding file inode table entry is set decremented by one. If the count is still non-zero go to step 6.
5. If the hard link count of the inode is not zero, it returns to the caller with a success status otherwise it marks the inode table entry as unused and de- allocates all the physical disk storage of the file.
6. It returns to the caller to the process with 0 (success) statuses.

2.8 Most frequently asked questions

1. What are the different File Types available in POSIX/UNIX, explain commands that can be used to create different files with an example.
2. Explain the different file types available in UNIX / POSIX systems.
3. Explain all the file attributes with an example.
4. Explain Data Structure and Proceedings involved in File Management.
5. Describe Unix Kernel Support for Files with neat diagram.
6. Explain the following tables with respect to Kernel Support provided for files.
   a. File descriptor table. B. File table c. Inode table
7. Explain Directory files and FIFO files with an example.
8. Differentiate between Hard links and Symbolic links with an example.