Chapter III

File System on UNIX

Files System Concept

The file system is a strategy for storing and organizing files on disk. UNIX treats everything, which is a sequence of bytes, stored in system as a file and every file as a sequence of bytes. For example, magnetic tapes, mail messages, characters typed on keyboard, line printer output, data flowing in pipes. Thus, there are many kinds of files on a UNIX system, all files are organized in a hierarchical structure starting with the root directory, which may in turn contain files and other directories as mentioned in Chapter I. So, the file system can be classified as follows (Valley, 1991):

1. Ordinary Files

This type of file is used for storing data. An ordinary file is always a part of a larger structure called a filesystem --the system dynamically manages disk space, automatically determining where to store a file on disk and allocating disk space for the file as needed-- and can exist only on certain kinds of devices, mainly disk drives. Ordinary files are used for both permanent and temporary storage of information. The file can be organized for either sequential or random access.

2. Directories

A directory is a file like any other file. It is used for grouping files together into meaningful categories. It can contain multiple occurrences of any file type, including other directories. The file name of a directory consists of from one to fourteen characters, only the null character () and the slash (/) cannot appear in a file name, all other characters are allowed including spaces, control characters, and non-ASCII codes.

All directories always have two file names which cannot be written (or removed) --the "." and ".." file names, called dot and dot-dot, respectively. These file names are written when the directory is created.

The "." file is itself a directory. It is a link to the directory that contains it. As a result, a reference to the "." file is always a reference to the current directory. While, the ".." file is also a directory. It is a link to the directory containing the current directory, often called the parent directory.

As mentioned above (see figure 1.1), the UNIX file system stores a hierarchical structure or tree-like structure where each non-terminate node corresponds to a directory. The top of this tree is a single directory conventionally called the "root directory" --"/". A pathname that does not begin with "/" character is called a "relative pathname" and gives the route to the file relative to a user's "current working directory"

3. Special Files

The last type of files can be separated into three categories; character-special, block-special, and pipes. Character-special and block-special files are peripheral devices. While, pipes are a real-time coupling between the output of one program and the input of another, allowing data to be passed immediately from program to program without the need to temporarily store it on an external medium.

A block-special file is any device that transfers data in fixed-size blocks and can access any data block in roughly the same amount of time as any other such as disk and diskette drives. In contrast, character-special files require data transfer in units of a byte or a few bytes rather than blocks or sectors. Furthermore, there is often no way to set or determine device positioning. These devices are able to read but not write, or to write but not read. Included in this large class of devices are terminals, magnetic tape drives, modems, mice, bar-code readers, plotters, page scanners, and printers. However, disk and diskette drives may also support a character device interface in addition to the required block device interface.

The terminal, for example, often responds to control character sequences that move the cursor and therefore allow data to be placed at arbitrary positions on the display.

In UNIX system, most devices are listed in the "/dev" directory or in one of its subdirectories.

The following list shows the Standard UNIX Directories :

The root (/) Directory
This directory contains all other files and directories. It is called root because all other directories branch from it like the branches and leaves of a tree.

The "/bin" Directory
The "/bin" directory contains executable program files.

The "/dev" Directory
The "/dev" directory is reserved for block-special and character-special files.

The "/etc" Directory

This directory should contain miscellaneous files needed for system operation, administration, and maintenance.

The "/lib" Directory

The "/lib" directory contains the C compilation system, including the compiler itself and the function libraries.

The "/tmp" and "/usr/tmp" Directories

These provide space for temporary files and a repository for short-lives files created during the execution of program and must have read, write, and search permission for all general users.

The "/usr" Directory

It is used for storing users' home directories.

The "/usr/bin" Directory

This directory contains all nonessential executable files that were shipped with the system and are available to the general users.

The "/usr/lib" Directory

The "/usr/lib" directory is used by all application packages installed in the system that need a system global directory.

The "/usr/include" Directory

This directory usually contains the header files used with the C standard library.

Files & Directory Information

All users in UNIX system will be assigned a login name and password with their own "home directory", where they are placed at login, for entering to the system. When users log into the system, they will be verified by requesting their login and password. While a user login is known as login-id, the system actually recognizes it as a number, called user-id or uid. Moreover, users are assigned a group identification, or group-id, which users are classified as part of a user group. All ordinary users are placed in a single group called "other". Therefore, user access to the file system will be determined by the permissions granted to user uid and group-id.

The file "/etc/passwd" contains all the login information about each user separated by colons login-id, uid, group-id, login-directory and shell are all contained in "/etc/password.

UNIX is a multi-user system, therefore, permissions are concerned with protecting resources from damage by unauthorized users. However, users must have privileges such as being allowed to create and manipulate their own files under their

"home directory". When a file or a directory is created, every attempt to access it is checked against every the file's or directory's permissions. If the check fails, the access is prohibited. Besides permissions, when files and directories are created the kernel allocates an inode to contain a description of the file. The inode is filled in with all the information needed to manage and protect the file. This information can be seen by using the "ls" command. Furthermore, there are three times contained in the inode. First, the time which the contents of the file were last modified (written). Second, the time which the file was last used (read and execute). Finally, the time which the inode itself was last changed.

Permissions

Permissions are simply means by which users are granted access to the system resources. Every file on UNIX system has a set of permissions for access. However, there is a special user on every UNIX system called super-user, who can read or modify any file on the system --this user is celled "root". The file permissions are stored in the inode of the file, in a field called the file mode.

As mentioned in chapter I, there are three kinds of permissions for each file. Moreover, different permissions can apply to different people. Permissions can be represented in a standard symbolic form, as used by the "ls" command. This notation represents permissions as a string of ten characters. Next to the first bit, the first group of three characters describes permission for the file owner, the second for the file's group and the third for the "anyone else" or "other" category.

- r w x r - x r - -

Concerning the first bit, if it is a directory, the first "-" will be replaced with "**d**". If it is "-', it indicates that this is an ordinary file.

For the other nine bytes, Haviland and Salama describe that there are three types of permissions within each of these triplets of permissions. Firstly, read permission on a directory means that the appropriate class of users is able to list the names of files and sub-directories contained within the directory. This does not mean that the users are able to read the information contained in the files themselves -- permission for that is controlled by the access permissions of the individual files. Concerning a file, this "r" permission means that user can read the content of the file.

Secondly, write permission on a directory enables a user to create new files and remove existing files in the directory. It does not permit a user to modify the contents of existing files unless the individual file permission is granted. It would, however, be possible to remove an existing file and create a new one. For a file, this "w" permission means that a user can write anything into it.

Thirdly, execute permission, also called search permission on a directory, allows a user to move into the directory or change to other directory. In addition, to open a file, or execute a program, a user must have execute permission on all directories leading to the file as specified in the file's pathname. Regarding a file, this "x" permission means that user can execute that file.

With regard to "w" permission, even if "group" and "other" users allow you to write a file, the system will not allow you to change its permission bits. Nevertheless, please remember that only the owner of a file or a directory may change the permissions on a file or a directory by using the utilities which are provided by UNIX.

System Calls and Library Subroutines

Haviland and Salama explained that

System calls are in fact the software developer's passport into the UNIX kernel. The kernel is a single piece of software which is permanently memory-resident and deals with a UNIX system's process scheduling and I/O control. In essence, the kernel is that part of UNIX which qualifies as an operating system proper. All user processes and all file system accesses will be resource, monitored and controlled by the kernel.

System calls are invoked in the same way a programmer would call an ordinary C subroutine or function. The essential difference between a subroutine and a system call is that when a program calls a subroutine the code executed is always part of the final object program, even if it was linked in from a library; with a system call the major part of the code executed is actually part of the kernel itself and not be calling program. In other words the calling program is making process and kernel is usually achieved via a software interrupt mechanism.

UNIX provides facilities not directly offered by the many system calls which make up the file access primitives. It also provides the Standard I/O routines ultimately used by the system call interface for manipulating and working with both files and directories. The following system calls and subroutines will present some system calls which are very useful to programmers for controlling both files and directories.

```
#include <stdio.h>
int     fopen (const char *path, const char *type);
extern  FILE *stdin, * stdout, * stderr;
```

Opens the file named by path for operations of type "type". A FILE block and a stream buffer are allocated using "malloc". The stream files stdin, stdout, and stderr are always open and may be used without an explicit fopen request.

The character string, for example, pointed to by "type" must specify one of the following values:

"r" Open the file for reading. Sets the file pointer to the first byte of the file.

"w" Opens the file for writing. Created the file if it does not exist; otherwise truncates it to zero length. Sets the file pointer to the first byte of the file.

```
#include <stdio.h>
int fread(void *ptr,int size,int elems,FILE *stream);
```

Retrieves a number of data elements, each "size" bytes long, from the stream file "stream" and stores them in consecutive elements of the array pointed to by "ptr". Reading stops when "elems" elements have been transferred, EOF is encountered, or an I/O error occurs. If the value of element is 0 or negative, no elements are transferred. One call transfer 65,535 bytes at most.

```
#include <stdio.h>
#include <unistd.h>
int  fseek (FILE *stream, long offset, int origin);
```

Performs a logical seek for the stream file "stream" by setting an internal file pointer to a new byte offset within the file at which the next read or write request will begin transfer of data. The new byte offset is computed as the sum of the signed value of offset and a byte displacement implied by the value of "origin".

```
#include <stdio.h>
long ftell( FILE *stream);
```

Returns the current position of the stream file "stream" as a byte offset from the beginning of the file.

```
#include <stdio.h>
int  fclose (FILE *stream);
```

Closes the stream file pointed to by "stream". The contents of the stream buffer if any, are written to the external file or device.

```
#include <stdio.h>
int  fwrite (const void *ptr, int size, int elems, FILE *stream);
```

Writes successive elements of the array pointed to by "ptr", each "size" bytes long, to the stream file "stream". Writing stops when "elems" elements have been transferred or an I/O error occurs. If the value of "elems" is 0 or negative, no elements are transferred.

int chdir (const char *path);

Sets the current working directory to the given "path". Path names that do not begin with "/" are taken to be relative to the current working directory, as specified by the path argument.

int chmod (const char *path, int modes);

Sets the access permissions of the file named by "path" to the 12 low-order bits of mode. The effective user must be superuser or the owner of the file.

int execl (char *path, char *arg(),..., (char *) 0);

This is the member of the exec family of functions replaces the current process image with a new process image. The initial argument for these functions is the pathname of a file which is to be executed.

char *getlogin (void);

Returns the login name of the current user.

char *getcwd (char *buf, int size);

Stores the pathname of the current directory in the character array pointed to by "buf"; the length of the pathname stored will not exceed "size" bytes.

#include <malloc.h>
void *malloc (size_t size);

Allocates an area of storage at least "size" bytes long, properly aligned for any data type.

int mkdir (const char *path, int mode);

Creates a new directory with a path name equal to null-terminated string pointed to by "path". The access permissions of the new directory are set to the 12 low-order bits of "mode" as modified by the process file creation mask.

void perror (const char *mes);

Writes the error message corresponding to the current value of errno to the standard-error file. The message text is preceded by the null-terminated string pointed to by "msg", which may use to explain the context in which the error occurred.

```
int     rmdir (const char *path);
```

Removes the directory named pointed by "path" if the directory is empty, if it is not a mount point , if it is not the current directory of any process, and if the calling process has write permission in the parent directory. The directory is considered empty when it contains only the "." and ".." entries.

```
#include <string.h>
char    *strcat (const char *s1, const char *s2);
```

Copies the string pointed to by "s2" to the end of the string pointed to by "s1", forming a single unbroken string that is the concatenation of the two original strings.

```
#include <string.h>
char    * strcpy (const char *s1, const char *s2);
```

Copies the null-terminated string pointed to by "s2" to the character array pointed to by "s1".

```
#include <stdio.h>
int     strlen (const char *s)
```

Finds the length of the string pointed to by "s".

```
int     fork (void)
```

Creates a new (child) process by making a copy of the current (parent) process. Both processed then continue execution of the same program.

```
int     system (const char *command);
```

Passed the string contents of the array pointed to by "command" to a new copy of the shell for execution as a command. The shell invoked by "system" is always "/bin/sh" to enhance portability of programs.

```
#include <stdio.h>
int   fprintf (FILE *stream, const char *format, ...);
```

Writes the result string to the stream file stream instead of to standard output.

```
void    exit (int status);
```

Terminates the current process after performing clean up actions, including closing all open files

Shell Command

The UNIX shell is both a command interpreter and a programming language. To manipulate and access files and directories, UNIX provides a shell which has built-in commands.

When a user logs in, "login" starts a shell process. A "-" is prefixed to the name of the login shell (-sh), which causes the shell to read commands from the files "/etc/profile" and "$HOME/.profile", if they exist. Moreover, when the shell is first started it opens three files; standard Input(0), standard output(1), standard error(2); and associates each file with a number called a "file descriptor". The following shell commands are either UNIX commands or a built-in shell commands in regard to file and directory access. (Valley, 1991).

pwd

Prints the current working directory

return

Used only within functions; it causes a function to exit to the last command executed.

cat [filename ...]

Writes all its input files as one file to standard output.

chmod mode path

Changes the file access permissions of "path" as directed by the "mode" operand. Only the owner of file or director, or the superuser, can change permissions.

"mode" may be specified as a octal value or as a symbolic expression listed in Table 3.1 and 3.2

Permission	Octal Value
Read by owner	400
Write by owner	200
Execute by owner	100
Read by group	040
Write by group	020
Execute by group	010
Read by others	004
Write by others	002
Execute by others	001

Table 3.1 chmod Octal File Permissions

who		operator		permission	
user	u	add	+	read	r
group	g	take away	-	write	w
other	o	absolute	=	execute	x

Table 3.2 chmod Symbolic Operations

cp filename target

Copies the files specified by "filename" to "target". "target" can be a directory, an existing file, or a new file. You can specify multiple filenames only when "target" is a directory.

If "target" is a directory, then all files are copied into that directory with the same filenames. If target is an existing file, then the input file is copied to it and the mode, owner, group, and links remain intact. If target is a new file, then the input file is copied to it and the mode, owner, and group of the new file remain the same as the old file.

find directory expr

Locates files and directories that match the values specified in "expr". The first argument to "find" must be a staring directory.

ls [opts] [file/directory]

Lists files and directories. Without any options or without a filename specified, "ls" lists all of the files in the current directory. If the file name is specified, only that file is listed. If a directory name is given, then the files in the specified directory are listed.

mkdir directory

Creates a new directory named "directory". The mode of the new directory is set to "777" as modified by the file-creation mask. The "mkdir" command requires that the parent directory have write permission set for the user making the new directory.

mv file(s) target

Moves or renames the specified files to target. The file and target cannot be the same. If the target is a directory, then the named "file(s)" are moved to the directory with their same names. If the target is an existing file, "mv" first determines whether the file can be overwritten. If so, then the file is moved to the target file name. If the target file cannot be written to, then "mv" prints the target file permission mode and asks for a response. "mv" then reads from the standard input and moves the file if permissible. Otherwise, "mv" exits with a nonzero exit status.

rm [opts] file(s)

Removes specified files. Removal of a file requires write permission for the directory ; the file is in, but does not require write or read permission for the file itself.

rmdir directory

Removes the directories specified as arguments to it. The directory must be empty.

du

Summarizes disk usage by displaying the amount disk space used by each file and directory.

df

Summarizes free disk space by displaying the amount of disk space available on the filesystem containing each argument pathname.

compress, gzip ,zip

Compresses or packs the specified files or standard input.

uncompress, gunzip, unzip

Uncompresses or expands or extracts the specified files or standard input.

vi

Standard editor on UNIX system.

pico

Simple text editor in the style of the Pine Composer which displays oriented text editor.

awk

Pattern scanning and processing language. It provides the capability to process the lines of an ASCII format file field-by-field. This utility can be used to alter the format of a file, to extract selected lines from a file, to tabulate numeric information in a file, or to prepare a full-scale report from a sorted data file.