The UNIX File System

⇒ The UNIX operating system is built around the concept of a file system which is used to store all of the information that constitutes the long-term state of the system.

⇒ This state includes the operating system kernel itself, the executable files for the commands supported by the operating system, configuration information, temporary work files, user data, and various special files that are used to give controlled access to system hardware and operating system functions.

⇒ A **file system** is a logical method for organising and storing large amounts of data/information in a way which makes it easy manage.

⇒ The **file** is the smallest unit in which information is stored (is device which can store the information, data, music (mp3 files), picture, movie, sound, etc…).

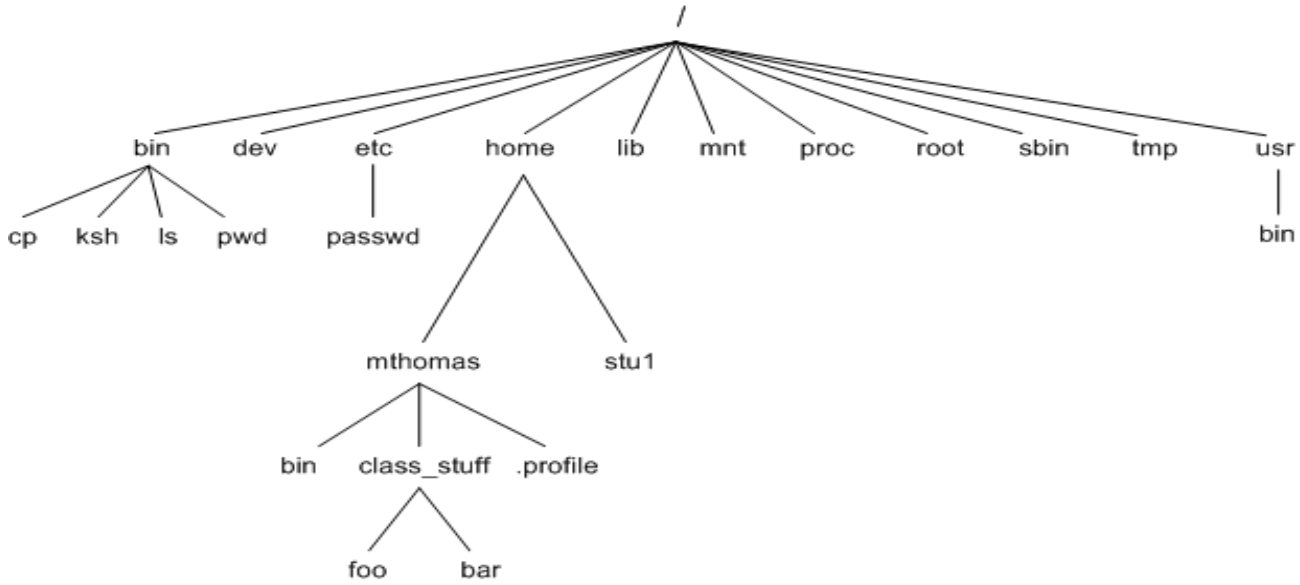
⇒ All of the files in the UNIX file system are organized into a **multi-leveled hierarchy** called a directory tree.

⇒ A family tree is an example of a hierarchical structure that represents how the UNIX file system is organized.

⇒ The UNIX file system might also be envisioned as an inverted tree or the root system of plant.

⇒ At the very top of the file system is single directory called "root" which is represented by a / (slash).All other files are "descendents" of root.

⇒ The number of levels is largely arbitrary, although most UNIX systems share some organizational similarities.



⇒ The following system files (i.e. directories) are present in most UNIX file systems:

- bin - short for binaries, this is the directory where many commonly used executable commands reside
- dev - contains device specific files
- etc - contains system configuration files
- home - contains user directories and files
- lib - contains all library files

- mnt - contains device files related to mounted devices
- proc - contains files related to system processes
- root - the root users' home directory (note this is different than /)
- sbin - system binary files reside here. If there is no sbin directory on your system, these files most likely reside in etc
- tmp - storage for temporary files which are periodically removed from the filesystem
- usr - also contains executable commands

⇒ Every item stored in a UNIX file system belongs to one of six types:

1. **Ordinary files**
   - Ordinary files can contain text, data, or program information.
   - Files cannot contain other files or directories.
   - Unlike other operating systems, UNIX filenames are not broken into a name part and an extension part (although extensions are still frequently used as a means to classify files). Instead they can contain any keyboard character except for '/' and be up to 256 characters long (note however that characters such as *,?,# and & have special meaning in most shells and should not therefore be used in filenames). Putting spaces in filenames also makes them difficult to manipulate - rather use the underscore '_'.
2. **Directories**
   - Directories are containers or folders that hold files, and other directories.
3. **Special Files/Devices**
   - To provide applications with easy access to hardware devices, UNIX allows them to be used in much the same way as ordinary files.
   - There are two types of devices in UNIX - **block-oriented** devices which transfer data in blocks (e.g. hard disks) and **character-oriented** devices that transfer data on a byte-by-byte basis (e.g. modems and dumb terminals).
4. **Links**
   - A link is a pointer to another file.
   - **Links** are special files enabling several names (*links*) to be associated to a single and same file.
   - The system makes it possible to have several instances of the same file in several places in the tree structure without needing to copy it, which in particular helps to ensure maximum coherence and save on disk space.
   - There are two types of links –
     - A**hard link/physical link**
     - A soft link/symbolic link

Hard Link/Physical link

- Represent an alternative name for a file.
- So, when a file has two physical links, the deletion of one or other of these links does not lead to the deletion of the file.
- More specifically, while there is at least one physical link remaining, the file is not deleted.
- On the other hand, when all physical links for the same file are deleted the file is too.
- Please note however, that it is only possible to create physical links within the single and same file system.
- Physical links are created using the *In* (without the option -*n*) command according to the following syntax:

ln name-of-real-file name-of-physical-link

Soft Link/Symbolic Link

- Provides an indirect pointer or shortcut to a file.
- A soft link is implemented as a directory file entry containing a pathname.
- In the event that a symbolic link is deleted, the file which it points to is not deleted.
- Symbolic links are created using the *In -s* command according to the following syntax:

  ln -s name-of-real-file name-of-symbolic-link

5. **Virtual files**
   - Do not really exist because they only exist in the memory.
   - These files, located in particular in the /proc directory contain information about the system (processor, memory, hard disks, processes, etc.);
6. **Pipes**
   - UNIX allows you to link commands together using a pipe. The pipe acts a temporary file which only exists to hold data from one command until it is read by another
   - For example, to pipe the output from one command into another command:

     who | wc –l

   - This command will tell you how many users are currently logged into the system.
   - The standard output from the "**who**" command is a list of all the users currently logged into the system.
   - This output is piped into the wc command as its standard input.
   - Used with the -l option this command counts the numbers of lines in the standard input and displays the result on its standard output - your terminal.

File Names

⇒ UNIX permits file names to use most characters, but avoid spaces, tabs and characters that have a special meaning to the shell, such as:

& ; ( ) | ? \ ' " ` [ ] { } <> $ - ! /

⇒ Case Sensitivity: uppercase and lowercase are not the same! These are three different files:

NOVEMBER November november

⇒ Length: can be up to 256 characters

⇒ Extensions: may be used to identify types of files

libc.a - ***archive, library file***
program.c - ***C language source file***
alpha2.f - ***Fortran source file***
xwd2ps.o - ***Object/executable code***
mygames.Z - ***Compressed file***

⇒ Hidden Files: have names that begin with a dot (.) For example:

.cshrc .login .mailrc .mwmrc

⇒ Uniqueness: as children in a family, no two files with the same parent directory can have the same name. Files located in separate directories can have identical names.

⇒ **Reserved Filenames:**

/ - ***the root directory (slash)***
. - ***current directory (period)***

.. ***- parent directory (double period)***
~ ***- your home directory (tilde)***

Specifying multiple filenames

⇒ Multiple filenames can be specified using special pattern-matching characters. The rules are:
- '?' matches any single character in that position in the filename.
- '*' matches zero or more characters in the filename. A '*' on its own will match all files. '*.*' matches all files with containing a '.'.
- Characters enclosed in square brackets ('[' and ']') will match any filename that has one of those characters in that position.
- A list of comma separated strings enclosed in curly braces ("{" and "}") will be expanded as a Cartesian product with the surrounding characters.

⇒ For example:
1. ??? matches all three-character filenames.
2. ?ell?matches any five-character filenames with 'ell' in the middle.
3. he* matches any filename beginning with 'he'.
4. [m-z]*[a-l] matches any filename that begins with a letter from 'm' to 'z' and ends in a letter from 'a' to 'l'.
5. {/usr,}{/bin,/lib}/file expands to /usr/bin/file /usr/lib/file /bin/file and /lib/file.

⇒ **Note** that the UNIX shell performs these expansions (including any filename matching) on a command's arguments *before* the command is executed.

inodes

⇒ UNIX directories and files don't really have names.
⇒ They are numbered, using node numbers called **inodes**, **vnodes**, or even **gnodes** (depending on the version of UNIX).
⇒ You won't find the name of a particular file or directory in or near the file or directory itself.
⇒ All the name-to-number mappings of files and directories are stored in the parent directories.
⇒ For each file or directory, a *link count* keeps track of how many parent directories contain a name-number mapping for each node. When a link count goes to zero, no directory points to the node and UNIX is free to reclaim the disk space.
⇒ UNIX permits all files to have many name-to-number mappings. So, a file may appear to have several different "names" (UNIX calls them "links"); that is, several names that all map to the same node number (and thus to the same file). Or, the file may have the same "name"; but, that name may appear in different directories.
⇒ Anyone can create a link to any file to which they have access. They don't need to be able to read or write the file itself to make the link; they only need write permission on the directory in which the name-to-number map (the name, or "link") is being created.
⇒ Directories are not allowed to have many name-to-number mappings. Each directory name-to-number map is allowed to appear in exactly one parent directory and no more.
⇒ This restriction means that every directory has only one "name". It prevents loops and cycles in the file system tree. (Many things are simpler if the tree has no cycles.)
⇒ Since a parent directory may have many sub-directories, and since the name "**..**" (dot dot) in every one of those sub-directories is a map to the node number of the parent directory, the

link count of the parent directory is increased by one for every sub-directory the parent contains.

⇒ Every directory also contains the name "**.**" (dot), a map to the directory itself, so the smallest link count of any Unix directory is 2: one for the map in the parent directory that gives the directory its "name", and one for the dot map in the directory itself.

⇒ Finally an**inode** is handle to a file and contains the following information:

- file ownership indication
- file type (e.g., regular, directory, special device, pipes, etc.)
- file access permissions. May have setuid (sticky) bit set.
- time of last access, and modification
- number of links (aliases) to the file
- pointers to the data blocks for the file
- size of the file in bytes (for regular files), major and minor device numbers for special devices.

⇒ An integral number of inodes fits in a single data block.

⇒ Information the inode does not contain:

- path (short or full) name of file

For example, when a user issues *open(``/etc/passwd'', ...)* the kernel performs the following operations:

1. Because the file name is a full path name, find the inode of the root directory (found in superblock) and search the corresponding file for the entry ``etc''
2. when the entry ``etc'' is found, fetch its corresponding inode and check that it is of type directory
3. scan the file associated with ``/etc'' looking for ``passwd''
4. Finally, fetch the inode associated with *passwd*'s directory entry, verify that it is a regular file, and start accessing the file.

Note: What would the system do when opening ``/dev/tty01''?

⇒ Eventually, the system would find the inode corresponding to the device, and note that its file type was ``special''.

⇒ Thus, it would extract the major/minor device number pair from the length field of the inode, and use the device number as an index into the device switch table.

Example

⇒ Suppose the **root** directory has node number #2. Here is a small part of a Unix file system tree, showing hypothetical node numbers:

Node #2	
. (dot)	**2**
.. (dot dot)	**2**
home	**123**
bin	**555**

Node #555	
. (dot)	**555**
.. (dot dot)	**2**
rm	**546**
ls	**984**

Node #123	
. (dot)	**123**
.. (dot dot)	**2**

usr	654

cp	333
ln	333
mv	333

ian	111
stud0002	755
stud0001	883
stud0003	221

⇒ Note how one directory (#555) has three name-to-number maps for the same node.

⇒ All three names (cp, ln, mv) refer to the same node number, in this case a file containing an executable program. (This program looks at its name and behaves differently depending on which name you use to call it.)

Node #111	
. (dot)	111
.. (dot dot)	123
.profile	334
.login	335
.logout	433

Node #333
Disk blocks *for the* *cp / ln / mv* *file* *(link count: 3)*

Node #335
Disk blocks *for the* *.login* *file* *(link count: 1)*

Example

⇒ Here are two shell programs that are linked into different directories under different names.

⇒ The only way you can tell which names point to the same program files is by looking at the **inode** numbers using the "**-i**" option to **ls**:

```
$ls -i /sbin/sh /usr/bin/sh
   136724 /sbin/sh        279208 /usr/bin/sh

$ncheck -i 279208,136724

/dev/dsk/c0t3d0s0:
279208  /usr/lib/rsh
136724  /sbin/jsh
136724  /sbin/sh
279208  /usr/bin/jsh
279208  /usr/bin/sh
```

⇒ The **ncheck** command is usable only by the Super User. It finds all pathnames that lead to a particular **inode**.

File Attributes and Permissions

Now that we have a basic understanding of how filesystems work, we'll turn our attention to understanding how filesystems influence the security of a Unix system. Nearly all of this discussion will be concerned with the metadata that a filesystem contains?the filenames, permissions, timestamps, and access control attributes.

You can use the ls command to list all of the files in a directory. For instance, to list all the files in your current directory, type:

⇒ % **ls**
⇒ instructions invoice letter more-stuff notes stats
⇒ %
⇒ Actually, ls alone won't list all of the files. Files and directories beginning with a dot (".") are hidden from the ls command but are shown if you use ls -a:
⇒ % **ls -a**
⇒ . .. .indent instructions invoice letter notes more-stuff stats
⇒ %
⇒ The entries for "." and ".." refer to the current directory and its parent directory, respectively. The file *.indent* is a hidden file. If you use ls -A instead of ls -a, you'll see hidden files, but "." and ".." will not be shown.
⇒ You can get a more detailed listing by using the ls -lF command:
⇒ % **ls -lF**
⇒ total 161
⇒ -rw-r--r-- 1 sian user 505 Feb 9 13:19 instructions
⇒ -rw-r--r-- 1 sian user 3159 Feb 9 13:14 invoice
⇒ -rw-r--r-- 1 sian user 6318 Feb 9 13:14 letter
⇒ -rw------- 1 sian user 15897 Feb 9 13:20 more-stuff
⇒ -rw-r----- 1 sian biochem 4320 Feb 9 13:20 notes
⇒ -rwxr-xr-x 1 sian user 122880 Feb 9 13:26 stats*
⇒ %
⇒ The first line of output generated by the ls command (total 161 in the example above) indicates the number of KBs taken up by the files in the directory. Each of the other lines of output contains the fields, from left to right, as described in Table 6-2.

Table 6-2. ls output

Field contents	Meaning
-	The file's type; for regular files, this field is always a dash
rw-r?r?	The file's permissions
1	The number of "hard" links to the file; the number of "names" for the file
sian	The name of the file's owner

user	The name of the file's group
505	The file's size, in bytes
Feb 9 13:19	The file's modification time
instructions	The file's name

File Permissions

The file permissions on each line of the ls listing tell you what the file is and what kind of file access (that is, the ability to read, write, or execute) is granted to various users on your system.

Here are two examples of file permissions:

-rw-------
drwxr-xr-x

The first character of the file's mode field indicates the type of file (described in Table 6-4).

Table 6-4. File types

Contents	Meaning
-	Plain file
d	Directory
D	Solaris door construct
c	Character device (tty or printer)
b	Block device (usually disk or CD-ROM)
l	Symbolic link (BSD or SVR4)
s	Socket (BSD or SVR4)
= or p	FIFO (System V, Linux)

The next nine characters taken in groups of three indicate who on your computer can do what with the file. There are three kinds of permissions:

r

Permission to read

w

Permission to write

x

Permission to execute

Similarly, there are three classes of permissions:

Owner

The file's owner

Group

Users who are in the file's group

Other

Everybody else on the system (except the superuser)

What is the file command in UNIX?

The file command determines the file type of a file. It reports the file type in human readable format (e.g. ‘ASCII text’) or MIME type (e.g. ‘text/plain; charset=us-ascii’). As filenames in UNIX can be entirely independent of file type file can be a useful command to determine how to view or work with a file.

How to determine the file type of a file

To determine the file type of a file pass the name of a file to the file command.The file name along with the file type will be printed to standard output.

```
file file.txt
file.txt: ASCII text
```

To show just the file type pass the -b option.

```
file -b file.txt
ASCII text
```

The file command can be useful as filenames in UNIX bear no relation to their file type. So a file called somefile.csv could actually be a zip file. This can be verified by the file command.

```
file somefile.csv
somefile.csv: Zip archive data, at least v2.0 to extract
```

How to determine the file type of multiple files

The file command can also operate on multiple files and will output a separate line to standard output for each file.

```
file unix-*.md
unix-cat.md:        ASCII text, with very long lines
unix-comm.md:       ASCII text, with very long lines
unix-cut.md:        UTF-8 Unicode text
unix-exit-status.md: ASCII text
unix-file.md:       ASCII text, with very long lines
```

How to view the mime type of a file

To view the mime type of a file rather than the human readable format pass the -i option.

```
file -i file.txt
file.txt: text/plain; charset=us-ascii
```

This can be combined with the -b option to just show the mime type.

```
file -i -b file.txt
text/plain; charset=us-ascii
```

Changing Permissions

To change the file or the directory permissions, you use the **chmod** (change mode) command. There are two ways to use chmod — the symbolic mode and the absolute mode.

Using chmod in Symbolic Mode

The easiest way for a beginner to modify file or directory permissions is to use the symbolic mode. With symbolic permissions you can add, delete, or specify the permission set you want by using the operators in the following table.

Sr.No.	Chmod operator & Description
1	+ Adds the designated permission(s) to a file or directory.
2	- Removes the designated permission(s) from a file or directory.
3	= Sets the designated permission(s).

Here's an example using **testfile**. Running **ls -1** on the testfile shows that the file's permissions are as follows −

```
$ls -l testfile
```

-rwxrwxr-- 1 amrood users 1024 Nov 2 00:10 testfile

Then each example **chmod** command from the preceding table is run on the testfile, followed by **ls –l**, so you can see the permission changes −

```
$chmod o+wx testfile
$ls -l testfile
-rwxrwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod u-x testfile
$ls -l testfile
-rw-rwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod g = rx testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Here's how you can combine these commands on a single line −

```
$chmod o+wx,u-x,g = rx testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Using chmod with Absolute Permissions

The second way to modify permissions with the chmod command is to use a number to specify each set of permissions for the file.

Each permission is assigned a value, as the following table shows, and the total of each set of permissions provides a number for that set.

Number	Octal Permission Representation	Ref
0	No permission	---
1	Execute permission	--x
2	Write permission	-w-
3	Execute and write permission: 1 (execute) + 2 (write) = 3	-wx
4	Read permission	r--
5	Read and execute permission: 4 (read) + 1 (execute) = 5	r-x
6	Read and write permission: 4 (read) + 2 (write) = 6	rw-

7	All permissions: 4 (read) + 2 (write) + 1 (execute) = 7	rwx

Here's an example using the testfile. Running **ls -1** on the testfile shows that the file's permissions are as follows −

```
$ls -l testfile
-rwxrwxr--  1 amrood   users 1024  Nov 2 00:10  testfile
```

Then each example **chmod** command from the preceding table is run on the testfile, followed by **ls –l**, so you can see the permission changes −

```
$ chmod 755 testfile
$ls -l testfile
-rwxr-xr-x  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod 743 testfile
$ls -l testfile
-rwxr---wx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod 043 testfile
$ls -l testfile
----r---wx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Changing Owners and Groups

While creating an account on Unix, it assigns a **owner ID** and a **group ID** to each user. All the permissions mentioned above are also assigned based on the Owner and the Groups.

Two commands are available to change the owner and the group of files −

- **chown** − The **chown** command stands for **"change owner"** and is used to change the owner of a file.
- **chgrp** − The **chgrp** command stands for **"change group"** and is used to change the group of a file.

Changing Ownership

The **chown** command changes the ownership of a file. The basic syntax is as follows −

```
$ chown user filelist
```

The value of the user can be either the **name of a user** on the system or the **user id (uid)** of a user on the system.

The following example will help you understand the concept −

```
$ chown amrood testfile
```

$

Changes the owner of the given file to the user **amrood**.

NOTE − The super user, root, has the unrestricted capability to change the ownership of any file but normal users can change the ownership of only those files that they own.

Changing Group Ownership

The **chgrp** command changes the group ownership of a file. The basic syntax is as follows −

$ chgrp group filelist

The value of group can be the **name of a group** on the system or **the group ID (GID)** of a group on the system.

Following example helps you understand the concept −

$ chgrp special testfile
$