Postgraduate course, Covenant University Ota, Nigeria

Oyelade, O. J., PhD | ola.oyelade@covenantuniversity.edu.ng

Isewon, I. | itunu.isewon@covenantuniversity.edu.ng

Module Two - Filesystem

2.1   Objectives

In this module, we will covers:

The Linux filesystem and directory structure.
File and directory handling commands.
How to make symbolic and hard links.
How wildcard filename expansion works.
What argument quoting is and when it should be used.

2.2   The Linux Filesystem

The Linux operating system is built around the concept of a filesystem 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 such as:
    - configuration information,
    - temporary workfiles,
    - user data, and
    - various special files that are used to give controlled access to system hardware and operating system functions.

Every item stored in a Linux filesystem belongs to one of four types:

1. **Ordinary files**

    - Ordinary files can contain text, data, or program information.

- They can contain any keyboard character except for '/' and be up to 256 characters long (note 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. **Devices**

- To provide applications with easy access to hardware devices, Linux allows them to be used in much the same way as ordinary files.
- There are two types of devices in Linux:
    - **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. There are two types of links - a **hard link** to a file is indistinguishable from the file itself. A **soft link** (or symbolic link) provides an indirect pointer or shortcut to a file. A soft link is implemented as a directory file entry containing a pathname.

2.3 Typical UNIX Directory Structure

- The Linux filesystem is laid out as a hierarchical tree structure which is anchored at a special top-level directory known as the root (designated by a slash '/').
- Because of the tree structure, a directory can have many child directories, but only one parent directory.
- Fig. 2.1 illustrates this layout.
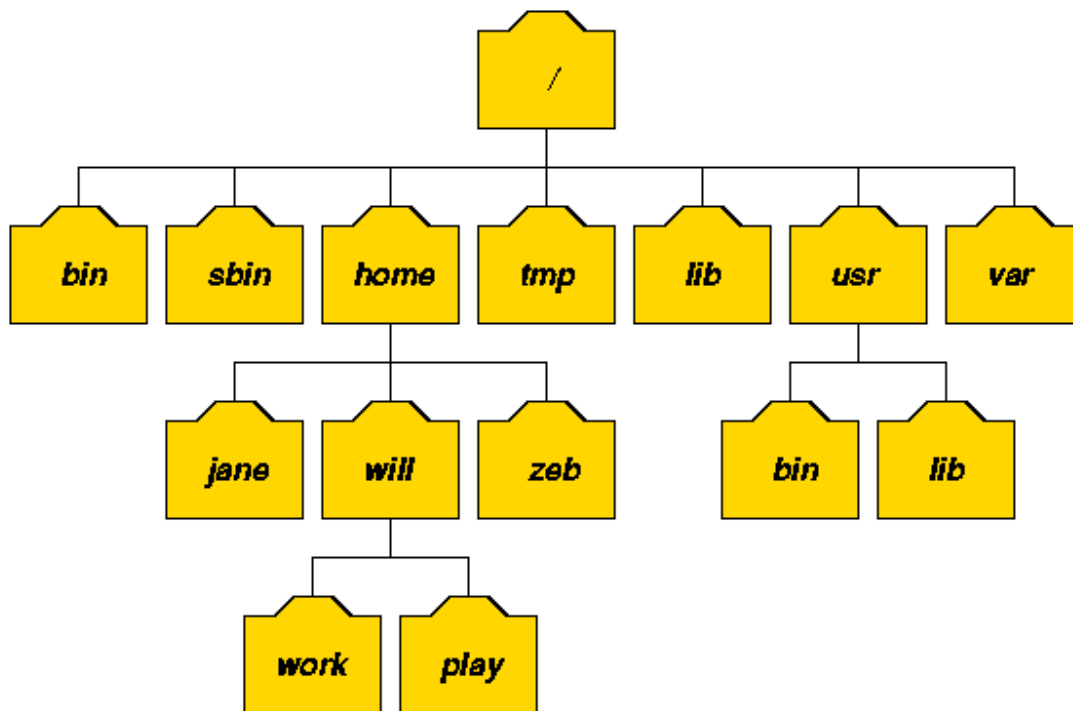
Fig. 2.1: Part of a typical UNIX filesystem tree

- To specify a location in the directory hierarchy, we must specify a path through the tree.
- The path to a location can be defined by an absolute path from the root /, or as a relative path from the current working directory.
- To specify a path, each directory along the route from the source to the destination must be included in the path, with each directory in the sequence being separated by a slash.
- To help with the specification of relative paths, Linux provides the shorthand **"."** for the current directory and **".."** for the parent directory.
    - o For example, the absolute path to the directory "play" is /home/will/play, while the relative path to this directory from "zeb" is ../will/play.

Fig. 2.2 below shows some typical directories you will find on Linux systems and briefly describes their contents. Note that these subdirectories appear as part of a

logical filesystem, they do not need be present on the same hard disk device; some may even be located on a remote machine and accessed across a network.

Directory	Typical Contents
/	The "root" directory
/bin	Essential low-level system utilities
/usr/bin	Higher-level system utilities and application programs
/sbin	Superuser system utilities (for performing system administration tasks)
/lib	Program libraries (collections of system calls that can be included in programs by a compiler) for low-level system utilities
/usr/lib	Program libraries for higher-level user programs
/tmp	Temporary file storage space (can be used by any user)
/home or /homes	User home directories containing personal file space for each user. Each directory is named after the login of the user.
/etc	UNIX system configuration and information files
/dev	Hardware devices
/proc	A pseudo-filesystem which is used as an interface to the kernel. Includes a sub-directory for each active program (or process).

Fig. 2.2: Typical Linux directories

When you log into Linux, your current working directory is your user home directory. You can refer to your home directory at any time as **"~"** and the home directory of other users as **"~<login>"**. So ~will/play is another way for user jane to specify an absolute path to the directory /homes/will/play. User will may refer to the directory as ~/play.

2.4 Directory and File Handling Commands

This section describes some of the more important directory and file handling commands.

- pwd (print [current] working directory)

---

pwd displays the full absolute path to the your current location in the filesystem.

So

```
$ pwd ↵
/usr/bin
```

implies that /usr/bin is the current working directory.

- ls (list directory)

  ls lists the contents of a directory. If no target directory is given, then the contents of the current working directory are displayed. So, if the current working directory is /,

  ```
  $ ls ↵
   bin   dev  home  mnt   share  usr  var
      boot  etc  lib   proc  sbin   tmp  vol
  ```

  Actually, **ls** doesn't show you *all* the entries in a directory - files and directories that begin with a dot (.) are hidden (this includes the directories '.' and '..' which are always present). The reason for this is that files that begin with a . usually contain important configuration information and should not be changed under normal circumstances. If you want to see all files, ls supports the -a option:

  ```
  $ ls -a ↵
  ```

  Even this listing is not that helpful - there are no hints to properties such as the size, type and ownership of files, just their names. To see more detailed information, use the -l option (long listing), which can be combined with the -a option as follows:

  ```
  $ ls -a -l ↵
    (or, equivalently,)
  $ ls -al ↵
  ```

  Each line of the output looks like this:
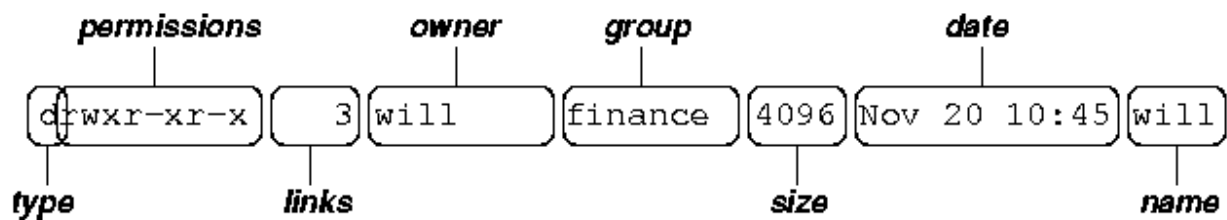
where:

- *type* is a single character which is either 'd' (directory), '-' (ordinary file), 'l' (symbolic link), 'b' (block-oriented device) or 'c' (character-oriented device).
- *permissions* is a set of characters describing access rights. There are 9 permission characters, describing 3 access types given to 3 user categories. The three access types are read ('r'), write ('w') and execute ('x'), and the three users categories are the user who owns the file, users in the group that the file belongs to and other users (the general public). An 'r', 'w' or 'x' character means the corresponding permission is present; a '-' means it is absent.
- *links* refers to the number of filesystem links pointing to the file/directory (see the discussion on hard/soft links in the next section).
- *owner* is usually the user who created the file or directory.
- *group* denotes a collection of users who are allowed to access the file according to the group access rights specified in the permissions field.
- *size* is the length of a file, or the number of bytes used by the operating system to store the list of files in a directory.
- *date* is the date when the file or directory was last modified (written to). The -u option display the time when the file was last accessed (read).
- *name* is the name of the file or directory.

ls supports more options. To find out what they are, type:

```
$ man ls ⏎
```

man is the online Linux user manual, and you can use it to get help with commands and find out about what options are supported. It has

quite a terse style which is often not that helpful, so some users prefer to the use the (non-standard) `info` utility if it is installed:

> `$ info ls` ⏎

- `cd` (change [current working] directory)

  > `$ cd` *path*

  changes your current working directory to *path* (which can be an absolute or a relative path). One of the most common relative paths to use is '..' (i.e. the parent directory of the current directory).

  Used without any target directory

  > `$ cd` ⏎

  resets your current working directory to your home directory (useful if you get lost). If you change into a directory and you subsequently want to return to your original directory, use

  > `$ cd -` ⏎

- `mkdir` (make directory)

  > `$ mkdir` *directory*

  creates a subdirectory called *directory* in the current working directory. You can only create subdirectories in a directory if you have write permission on that directory.

- `rmdir` (remove directory)

  > `$ rmdir` *directory*

  removes the subdirectory *directory* from the current working directory. You can only remove subdirectories if they are completely empty (i.e. of all entries besides the **'.'** and **'..'** directories).

- `cp` (copy)

  `cp` is used to make copies of files or entire directories. To copy files, use:

  > `$ cp` *source-file(s) destination*

  where *source-file(s)* and *destination* specify the source and destination of the copy respectively. The behaviour of cp depends on whether the destination is a file or a directory. If the destination is a file, only one source file is allowed and cp makes a new file called *destination* that has the same contents as the source file. If the destination is a directory, many source files can be specified, each of which will be copied into the destination directory. Section 2.6 will discuss efficient specification of source files using wildcard characters.

  To copy entire directories (including their contents), use a *recursive* copy:

  > `$ cp -rd` *source-directories destination-directory*

  To copy a file (e.g. Hello.txt) in `oyelade directory` into the `oyelade/ade directory` as `Hello.bak:`

  `$ cp /home/oyelade/Hello.txt  /home/oyelade/ade/Hello.bak` ⬅

  > `$ cp -r myfolder/destionation` ⬅

  Here, all folders and files will be copied recursively.

Copying multiple files:

`$ cp  /home/usr/dir/{file1,file2,file3,file4} /home/usr/destination/`
⬅

- `mv` (move/rename)

  `mv` is used to rename files/directories and/or move them from one directory into another. Exactly one source and one destination must be specified:

$ `mv` *source destination*

N.B.: if *destination* is an existing file it will be destroyed and overwritten by *source* (you can use the `-i` option if you would like to be asked for confirmation before a file is overwritten in this way).

- `rm` (remove/delete)

  $ `rm` *target-file(s)*

removes the specified files. Unlike other operating systems, it is almost impossible to recover a deleted file unless you have a backup (there is no recycle bin!) so use this command with care. If you would like to be asked before files are deleted, use the -i option:

```
$ rm -i myfile ⏎
rm: remove 'myfile'?
```

`rm` can also be used to delete directories (along with all of their contents, including any subdirectories they contain). To do this, use the -r option. To avoid rm from asking any questions or giving errors (e.g. if the file doesn't exist) you used the -f (force) option. Extreme care needs to be taken when using this option - consider what would happen if a system administrator was trying to delete user will's home directory and accidentally typed:

```
$ rm -rf / home/will ⏎
```

(instead of `rm -rf /home/will`).

```
$ rm -r myfolder ⏎  To remove a folder
```

- `cat` (catenate/type)

  $ `cat` *target-file(s)*

displays the contents of *target-file(s)* on the screen, one after the other. You can also use it to create files from keyboard input as follows (> is the output redirection operator.

```
$ cat > hello.txt ↵
hello world! ↵
[ctrl-d]
$ ls hello.txt ↵
hello.txt
$ cat hello.txt ↵
hello world!
$
```

- `more` and `less` (catenate with pause)

    ```
    $ more  target-file(s)
    ```

    displays the contents of *target-file(s)* on the screen, pausing at the end of each screenful and asking the user to press a key (useful for long files). It also incorporates a searching facility (press '/' and then type a phrase that you want to look for).

    You can also use `more` to break up the output of commands that produce more than one screenful of output as follows (`|` is the pipe operator, which will be discussed in the next module):

    ```
    $ ls -l | more ↵
    ```

    `less` is just like `more`, except that has a few extra features (such as allowing users to scroll backwards and forwards through the displayed file). `less` not a standard utility, however and may not be present on all Linux systems.

2.5 Making Hard and Soft (Symbolic) Links

Direct (hard) and indirect (soft or symbolic) links from one file or directory to another can be created using the ln command.

```
$ ln filename linkname
```

creates another directory entry for *filename* called *linkname* (i.e. *linkname* is a hard link). Both directory entries appear identical (and both now have a link count of 2). If either *filename* or *linkname* is modified, the change will be reflected in the other file (since they are in fact just two different directory entries pointing to the same file).

```
$ ln -s filename linkname
```

creates a shortcut called *linkname* (i.e. *linkname* is a soft link). The shortcut appears as an entry with a special type ('l'):

```
$ ln -s hello.txt bye.txt ⏎
$ ls -l bye.txt ⏎
lrwxrwxrwx   1 will finance 13 bye.txt -> hello.txt
$
```

The link count of the source file remains unaffected. Notice that the permission bits on a symbolic link are not used (always appearing as rwxrwxrwx). Instead the permissions on the link are determined by the permissions on the target (hello.txt in this case).

2.6 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`.

Exercises Two

1. Try the following command sequence:
   - `cd`
   - `pwd`
   - `ls -al`
   - `cd .`
   - `pwd`   (where did that get you?)
   - `cd ..`
   - `pwd`
   - `ls -al`
   - `cd ..`
   - `pwd`
   - `ls -al`
   - `cd ..`
   - `pwd`   (what happens now)
   - `cd /etc`
   - `ls -al | more`
   - `cat passwd`
   - `cd -`
   - `pwd`
2. Continue to explore the filesystem tree using `cd`, `ls`, `pwd` and `cat`. Look in `/bin`, `/usr/bin`, `/sbin`, `/tmp` and `/boot`. What do you see?
3. Explore `/dev`. Can you identify what devices are available? Which are character-oriented and which are block-oriented? Can you identify your tty (terminal) device (typing `who am i` might help); who is the owner of your tty (use `ls -l`)?

4. Explore /proc. Display the contents of the files interrupts, devices, cpuinfo, meminfo and uptime using cat. Can you see why we say /proc is a pseudo-filesystem which allows access to kernel data structures?
5. Change to the home directory of another user directly, using cd ~username.
6. Change back into your home directory.
7. Make subdirectories called work and play.
8. Delete the subdirectory called work.
9. Copy the file /etc/passwd into your home directory.
10. Move it into the subdirectory play.
11. What is the difference between listing the contents of directory play with ls -l and ls -L?
12. Create a file called hello.txt that contains the words "hello world". Can you use "cp" using "terminal" as the source file to achieve the same effect?
13. Copy hello.txt to terminal. What happens?
14. Imagine you were working on a system and someone accidentally deleted the ls command (/bin/ls). How could you get a list of the files in the current directory? Try it.
15. How would you create and then delete a file called "$SHELL"? Try it.
16. How would you create and then delete a file that begins with the symbol #? Try it.
17. How would you create and then delete a file that begins with the symbol -? Try it.
18. Experiment with the options on the ls command. What do the d, i, R and F options do?