The File System

The file system is a hierarchical system following the *file system standard*. It is a hierarchical system based off of UNIX standards that contains not just stored data for the user, but the operating system itself. The file system contains all of the files that the operating system requires to work.

6.1 File System Roadmap

If you navigate to the root directory `/`:

```
cd / && ls -la
```

> dr-xr-xr-x. 20 root root 4096 Aug 15 2019 .
>
> dr-xr-xr-x. 20 root root 4096 Aug 15 2019 ..
>
> drwxr-xr-x 4 root root 76 Aug 15 2019 accounts
>
> drwxr-xr-x 7 root root 89 Aug 12 2016 admiral_archive
>
> -rw-r--r-- 1 root root 0 Jan 14 2015 .autorelabel
>
> lrwxrwxrwx 1 root root 7 Jul 27 2018 bin -> usr/bin
>
> dr-xr-xr-x. 5 root root 4096 Apr 24 21:08 boot
>
> drwxr-xr-x 19 root root 3340 Apr 24 20:49 dev
>
> drwxr-xr-x. 100 root root 8192 Apr 28 16:36 etc
>
> drwx--x--x 1358 root root 32768 Jun 28 01:41 home
>
> lrwxrwxrwx 1 root root 7 Jul 27 2018 lib -> usr/lib
>
> lrwxrwxrwx 1 root root 9 Jul 27 2018 lib64 -> usr/lib64
>
> drwxr-xr-x. 2 root root 6 Apr 10 2018 media
>
> drwxr-xr-x. 2 root root 6 Apr 10 2018 mnt
>
> drwxr-xr-x 9 root root 85 Feb 2 2015 nsr
>
> drwxr-xr-x. 3 root root 16 Apr 28 16:35 opt
>
> dr-xr-xr-x 228 root root 0 Apr 24 20:49 proc
>
> dr-xr-x---. 5 root root 4096 Apr 28 20:48 root
>
> drwxr-xr-x 34 root root 1060 Jun 28 19:44 run

> lrwxrwxrwx 1 root root 8 Jul 27 2018 sbin -> usr/sbin
>
> drwxr-xr-x. 2 root root 6 Apr 10 2018 srv
>
> dr-xr-xr-x 13 root root 0 Jun 3 00:16 sys
>
> drwxrwxrwt. 49 root root 4096 Jun 28 18:14 tmp
>
> drwxr-xr-x. 13 root root 4096 Jul 27 2018 usr
>
> drwxr-xr-x. 20 root root 4096 Jul 27 2018 var

You'll see a number of directories there. Table 6.1 (pp. 179) has many default directories and their descriptions, here are a few:

- `bin/` - this directory holds many of the commands we use. Remember this directory is on the shell's path to find commands.
- `sbin/` - System commands
- `boot/` - Commands required for booting the system up.
- `etc/` - System configurations / data / etc.
- `home/` - Specific user directories for
- `mnt/` - generic mounting points for the file system and its devices.

Depending on the type of system you're running on, you may see a number of directories that don't come with default implementations, or you may see other directories maintained by the superuser (e.g. the `admiral_archive`).

6.2 File Types

The types of files that exist in the file tree that we've worked with so far are just directories and ordinary files. There are a few more. When you're in a directory, and you type: `ls -l`, the very first `-` is the type of file: All the file types (and their symbols) are:

- `-` **Ordinary Files:** Files containing text or executable binaries
- `d` **Directories**: Files that contain the names and addresses of other files
- `c` **Character** or `b` **Block Special Files**: These represent entire drives or partitions
- `l` **Symbolic Links**: These files are pointers that act as a link to another space in the file hierarchy
- `s` **Sockets**: Files that are used for inter-process communication
- `p` **Named Pipes**: Another interprocess communication tool.

Ordinary Files

Ordinary files are any files that are plain text, have some encoding, or act as any means of storage. These types of files are stored with file names that typically have some extensions to help the operating system know how to work with these files.

File extensions are nothing more than an additional string after a `.` in the name that tells the operating system the most likely programs that the file can interact with. For example, if you attempt to open a `.pdf` file with a text editor, you'll see gibberish. If you change the name of a `.pdf` file to have an extension `.txt`, the default program required to open it will be some text editor. If you instead still open the file with a `.pdf` reader, the fill will work just the same. Extensions are there only to help out the operating system (and to help us keep our files organized).

Directories

Directories are files which store the names and addresses of other files. You can view a directory's file contents by accessing it with `vim` or `nano`. For every entry in a directory (i.e. all of the files within), the directories contain the file name and their **i-node**, the file information node. This i-node is essentially the address, in that it points to a table which points to a specific location in memory as to where the file exists.

Special Files

Special files represent i/o devices attached to the system. **Character Special Files**: these files work with byte oriented i/o devices (like a printer), whereas **Block Special Files** work with high speed io devices that transfer data in blocks (like hard drives).

Symbolic Links

It's often times the case that you'll want some file directory to have the same data as another. You can create a "link" that will essentially act as a shortcut that will take you to a completely different path. Say for example, you had a class that you taught every semester. You could link the notes to each directory, but have only on actual notes directory that acted as a source of truth.

Sockets and Named Pipes

These are used for interprocess communication. Often times, as programs and processes get more and more complex, they need to communicate with other processes all the same. It's much easier for multiple processes to tackle a problem than it is for a single one, however, when multiple processes work on a given problem, it's possible that they may get tangled up with resources. By using sockets and named pipes, these systems can work with each other to ensure that they don't get tangle up.

6.3 File Access

We've already seen how to change the mode of a file with `chmod`. So far, though, we've only been changing the modes of ordinary files. Since directories are files too, directories have the same read/write/execute as do the ordinary files.

Read:

Without read permissions, users are unable to see what's in the directory. Curiously, however, depending on the file permissions of what is inside of that directory, you may still be able to access specific files and sub-directories. The permissions of read pertain only to that directory:

```
mkdir unreadableDirectory && cd unreadableDirectory
mkdir readableDirectory
touch readableScript.sh && chmod 700 readableScript.sh
```

Inside `readableScript.sh` we have:

```
#!/bin/bash

echo I am a file within the directory that has no read permissions
```

Now, navigating to the directory above `unreadableDirectory`, we need to remove read permissions:

```
chmod 300 unreadableDirectory
```

Now, if you attempt to read what's in the directory with `ls`, you'll fail:

```
ls unreadableDirectory
```

> ls: cannot open directory unreadableDirectory/: Permission denied

We've only removed the read to the directory, and that's it, we can still access files within it by using their full path:

```
./unreadableDirectory/readableScript.sh
```

> I am a file within the directory that has no read permissions

Additionally, you can also go beyond the directory into readable subdirectories:

```
cd unreadableDirectory/readableDirectory
```

The above will let you navigate to a subdirectory, and you'll still be able to see everything inside of it. These permissions don't cascade down.

Write:

When a directory does not have write permissions, you, naturally, cannot write to it:

```
mkdir unwritableDirectory && cd unwritableDirectory
mkdir subdirectory
cp ../unreadableDirectory/readableScript.sh ./
cd ../
chmod 500 unwritableDirectory && cd unwritableDirectory
```

Since the directory is readable, we can hop into it and see what's in the directory. Being unable to write to a subdirectory means you cannot create new files:

```
touch newFile
```

> touch: cannot touch ‘newFile’: Permission denied

```
mkdir newDirectory
```

> mkdir: cannot create directory ‘newDirectory’: Permission denied

If a file already exists (and you have write permissions), however, you can still edit it. Additionally, subdirectories of an unwritable directory do still allow for you to write to them (assuming they have the proper permissions).

Execute:

When a directory doesn't have execute permissions it blocks the user from executing anything within that directory or on its path:

```
mkdir unexecutableDirectory && cd unexecutableDirectory
mkdir subdirectory
cp ../unreadableDirectory/readableScript.sh ./
cp readableScript.sh subdirectory/cd ../
chmod 600 unexecutableDirectory
```

Now, in removing the execution permissions of the directory, we've ostensibly blocked anyone form getting into the directory whatsoever:

```
cd unexecutableDirectory
```

> -bash: cd: unexecutableDirectory/: Permission denied

We can still read what's in it (while being told we're not allowed to access any of it):

```
ls unexecutableDirectory/
```

> ls: cannot access unexecutableDirectory/subdirectory: Permission denied
>
> ls: cannot access unexecutableDirectory/readableScript.sh: Permission denied
>
> readableScript.sh subdirectory

Finally, if there is a subdirectory within an unexecutable directory, it is completely blocked regardless of its permissions:

```
cd unexecutableDirectory/subdirectory
```

> -bash: cd: unexecutableDirectory/subdirectory: Permission denied

or

```
./unexecutableDirectory/subdirectory/readableScript.sh
```

> -bash: ./unexecutableDirectory/subdirectory/readableScript.sh: Permission denied