UNIT I

CHAPTER 1
LINUX OPERATING SYSTEM

1.1 Linux Operating System Concepts and Architecture

The Linux kernel is composed of five main subsystems that communicate using procedure calls. The architecture of the kernel is one of the reasons that Linux has been successfully adopted by many users. In particular, the Linux kernel architecture was designed to support a large number of volunteer developers. Further, the subsystems that are most likely to need enhancements were architected to easily support extensibility.

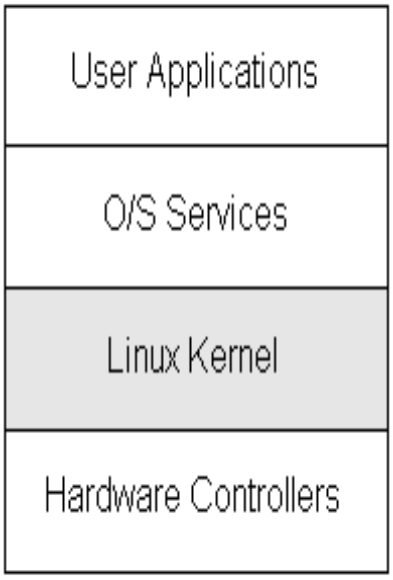


Figure 1.1: *Decomposition of Linux System into Major Subsystems*

The Linux operating system is composed of four major subsystems:

1. User Applications -- the set of applications in use on a particular Linux system will be different depending on what the computer system is used for, but typical examples include a word-processing application and a web-browser.

2. O/S Services -- these are services that are typically considered part of the operating system (a windowing system, command shell, etc.); also, the programming interface to the kernel (compiler tool and library) is included in this subsystem.

3. Linux Kernel -- this is the main area of interest in this paper; the kernel abstracts and mediates access to the hardware resources, including the CPU.

4. Hardware Controllers -- this subsystem is comprised of all the possible physical devices in a Linux installation; for example, the CPU, memory hardware, hard disks, and network hardware are all members of this subsystem

1.2 Over view of the Linux Kernel

The Linux kernel is composed of five main subsystems:

1. The Process Scheduler (SCHED) is responsible for controlling process access to the CPU. The scheduler enforces a policy that ensures that processes will have fair access to the CPU, while ensuring that necessary hardware actions are performed by the kernel on time.

2. The Memory Manager (MM) permits multiple process to securely share the machine's main memory system. In addition, the memory manager supports virtual memory that allows Linux to support processes that use more memory than is available in the system. Unused

memory is swapped out to persistent storage using the file system then swapped back in when it is needed.

3. The Virtual File System (VFS) abstracts the details of the variety of hardware devices by presenting a common file interface to all devices. In addition, the VFS supports several file system formats that are compatible with other operating systems.

4. The Network Interface (NET) provides access to several networking standards and a variety of network hardware.

5. The Inter-Process Communication (IPC) subsystem supports several mechanisms for process-to-process communication on a single Linux system.

1.3 User space and Kernel space

System memory in Linux can be divided into two distinct regions: *kernel space* and *user space*. Kernel space is where the *kernel* (i.e., the core of the operating system) *executes* (i.e., runs) and provides its *services*. User space is that set of memory locations in which *user processes* (i.e., everything other than the kernel) run.

Memory consists of *RAM* (random access memory) cells, whose contents can be *accessed* (i.e., read and written to) at extremely high speeds but are retained only temporarily (i.e., while in use or, at most, while the power supply remains on). Its purpose is to hold programs and data that are currently in use and thereby serve as a high speed intermediary between the CPU (central processing unit) and the much slower *storage*, which most commonly consists of one or more hard disk drives (HDDs).

A *process* is an executing instance of a program. One of the roles of the kernel is to manage individual user processes within this space and to prevent them from interfering with each other.

Kernel space can be accessed by user processes only through the use of *system calls*. System calls are requests in a Unix-like operating system by an *active process* for a service performed by the kernel, such as *input/output* (I/O) or process creation. An active process is a process that is currently progressing in the CPU, as contrasted with a process that is waiting for its next turn in the CPU. I/O is any program, operation or device that transfers data to or from a CPU and to or from a peripheral device (such as disk drives, keyboards, mice and printers).

1.4 Processes and Demons

A ***process*** is an *executing* (i.e., running) instance of a *program*. Processes are also frequently referred to as *tasks*.

A program is an *executable file* that is held in *storage*. Storage refers to devices or media that can retain data for relatively long periods of time (e.g., years or even decades), such as hard disk drives (HDDs), optical disks and magnetic tape. This contrasts with *memory*, whose contents can be accessed (i.e., read and written to) at extremely high speeds but which are retained only temporarily (i.e., while in use or only as long as the power supply remains on).

A program is a passive entity until it is launched, and a process can be thought of as a program in action. Processes are dynamic entities in that they are constantly changing as their machine code instructions are executed by the CPU. Each process consists of (1) *system resources* that are allocated to it, (2) a section of memory, (3) security attributes (such as its *owner* and its set of *permissions*) and (4) the processor *state*.

An alternative definition of a process is the execution *context* of a running program, i.e., all of the activity in the current *time slot* in the CPU. A time slot, also called a *time slice* or a *quantum*, is the length of time that each process is permitted to run in the CPU until it is *preempted* (i.e., replaced) by another process in a *time sharing* operating system.

A ***daemon*** is a type of program on Unix-like operating systems that runs unobtrusively in the background, rather than under the direct control of a user, waiting to be activated by the occurrence of a specific event or condition.

Unix-like systems typically run numerous daemons, mainly to accommodate requests for services from other computers on a network, but also to respond to other programs and to hardware activity. Examples of actions or conditions that can trigger daemons into activity are a specific time or date, passage of a specified time interval, a file landing in a particular directory, receipt of an e-mail or a Web request made through a particular communication line. It is not necessary that the perpetrator of the action or condition be aware that a daemon is *listening*, although programs frequently will perform an action only because they are aware that they will implicitly arouse a daemon.

Daemons are usually instantiated as *processes*. A process is an *executing* (i.e., running) instance of a program. Processes are managed by the *kernel* (i.e., the core of the operating system), which assigns each a unique *process identification number* (PID).
There are three basic types of processes in Linux: interactive, batch and daemon. Interactive processes are run interactively by a user at the *command line* (i.e., all-text mode). Batch processes are submitted from a queue of processes and are not associated with the command line; they are well suited for performing recurring tasks when system usage is otherwise low.
Daemons are recognized by the system as any processes whose *parent process* has a PID of one, which always represents the process *init*. init is always the first process that is started when a Linux computer is *booted up* (i.e., started), and it remains on the system until the computer is turned off. init *adopts* any process whose *parent* process *dies* (i.e., terminates) without waiting for the *child* process's status. Thus, the common method for launching a daemon involves *forking* (i.e., dividing) once or twice, and making the parent (and grandparent) processes die while the child (or grandchild) process begins performing its normal function.

Some daemons are launched via *System V init scripts*, which are *scripts* (i.e., short programs) that are run automatically when the system is booting up. They may either survive for the duration of the session or be regenerated at intervals. Many daemons are now started only as required and by a single daemon, *xinetd* (which has replaced *inetd* in newer systems), rather than running continuously. xinetd, which is referred to as a *TCP/IP super server*, itself is started at boot time, and it listens to the *ports* assigned to the processes listed in the */etc/inetd.conf* or in */etc/xinetd.conf* configuration file. Examples of daemons that it starts include *crond* (which runs scheduled tasks), *ftpd* (file transfer), *lpd* (laser printing), *rlogind* (remote login), *rshd* (remote command execution) and *telnetd* (telnet). In addition to being launched by the operating system and by application programs, some daemons can also be started manually. Examples of commands that launch daemons include *binlogd* (which logs binary events to specified files), *mysqld* (the MySQL databse server) and *apache* (the Apache web server). In many Unix-like operating systems, including Linux, each daemon has a single *script* (i.e., short program) with which it can be terminated, restarted or have its status checked. The handling of these scripts is based on *runlevels*. A runlevel is a configuration or operating state of the system that only allows certain selected processes to exist. Booting into a different runlevel can help solve certain problems, including repairing system errors. The term *daemon* is derived from the daemons of Greek mythology, which were supernatural beings that ranked between gods and mortals and which possessed special knowledge and power[1]. For example, Socrates claimed to have a daemon that gave him warnings and advice but never coerced him into following it. He also claimed that his daemon exhibited greater accuracy than any of the forms of divination practiced at the time.

The word *daemon* was first used in a computer context at the pioneering Project MAC (which later became the MIT Laboratory for Computer Science) using the IBM 7094 in 1963. This

usage was inspired by Maxwell's daemon of physics and thermodynamics, which was an imaginary agent that helped sort molecules of different speeds and worked tirelessly in the background. The term was then used to describe background processes which worked tirelessly to perform system chores. The first computer daemon was a program that automatically made tape backups. After the term was adopted for computer use, it was rationalized as an acronym for *Disk And Execution MONitor*.

1.5 **Process Control**

Under Linux, the *ptrace* system call is supported for process control, and it works as in 4.3BSD. To obtain process and system information, Linux also provides a */proc* filesystem, but with very different semantics. Under Linux, */proc* consists of a number of files providing general system information, such as memory usage, load average, loaded module statistics, and network statistics. These files are generally accessed using *read* and *write* and their contents can be parsed using *scanf*. The */proc* filesystem under Linux also provides a directory entry for each running process, named by process ID, which contains file entries for information such as the command line, links to the current working directory and executable file, open file descriptors, and so forth. The kernel provides all of this information on the fly in response to *read* requests. This implementation is not unlike the */proc* filesystem found in Plan 9, but it does have its drawbacks--for example, for a tool such as *ps* to list a table of information on all running processes, many directories must be traversed and many files opened and read. By comparison, the *kvm* routines used on other UNIX systems read kernel data structures directly with only a few system calls. Obviously, each implementation is so vastly different that porting applications which use them can prove to be a real task. It should be pointed out that the SVR4 */proc* filesystem is a very different beast than that found in Linux, and they may not be used in the same context. Arguably, any program which uses the *kvm* routines or SVR4 */proc* filesystem is not really portable, and those sections of code should be rewritten for each operating system.

Overview of Linux Administration

2.1 **Linux File system**

General A simple description of the UNIX system, also applicable to Linux, is this: "On a UNIX system, everything is a file; if something is not a file, it is a process." This statement is true because there are special files that are more than just files (named pipes and sockets, for instance), but to keep things simple, saying that everything is a file is an acceptable generalization. A Linux system, just like UNIX, makes no difference between a file and a directory, since a directory is just a file containing names of other files. Programs, services, texts, images, and so forth, are all files. Input and output devices, and generally all devices, are considered to be files, according to the system. In order to manage all those files in an orderly fashion, man likes to think of them in an ordered tree-like structure on the hard disk, as we know from MS-DOS (Disk Operating System) for instance. The large branches contain more branches, and the branches at the end contain the tree's leaves or normal files. For now we will use this image of the tree, but we will find out later why this is not a fully accurate image. Sorts of files Most files are just files, called *regular* files; they contain normal data, for example text files, executable files or programs, input for or output from a program and so on. While it is reasonably safe to suppose that everything you encounter on a Linux system is a file, there are some exceptions.

- *Directories*: files that are lists of other files.
- *Special files*: the mechanism used for input and output. Most special files are in /dev.
- *Links*: a system to make a file or directory visible in multiple parts of the system's file tree.

- *(Domain) sockets*: a special file type, similar to TCP/IP sockets, providing inter-process networking protected by the file system's access control.
- *Named pipes*: act more or less like sockets and form a way for processes to communicate with each other, without using network socket semantics.

As a user, you only need to deal directly with plain files, executable files, directories and links. The special file types are there for making your system do what you demand from it and are dealt with by system administrators and programmers.

For convenience, the Linux file system is usually thought of in a tree structure. On a standard Linux system you will find the layout generally follows the scheme presented below.

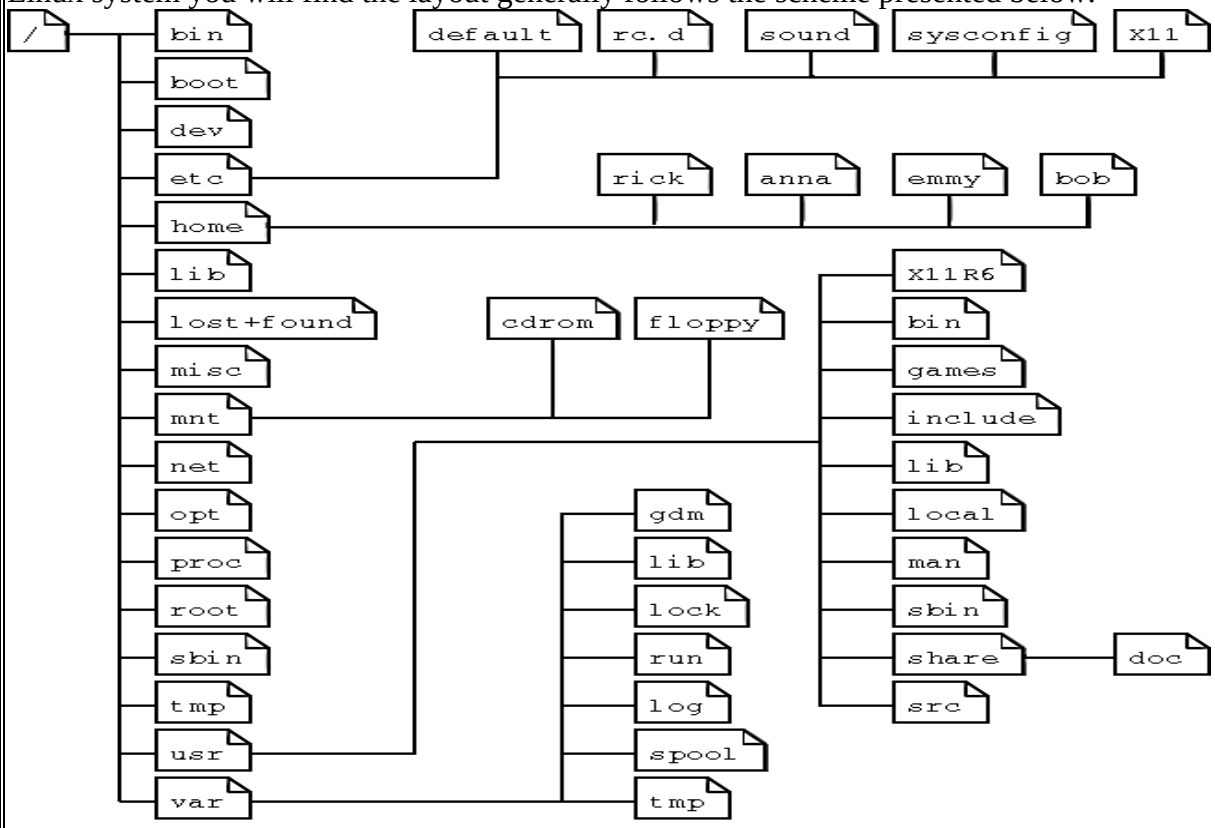


Figure 2-1. Linux file system layout

This is a layout from a RedHat system. Depending on the system admin, the operating system and the mission of the UNIX machine, the structure may vary, and directories may be left out or added at will. The names are not even required; they are only a convention. The tree of the file system starts at the trunk or *slash*, indicated by a forward slash (/). This directory, containing all underlying directories and files, is also called the *root directory* or "the root" of the file system. Directories that are only one level below the root directory are often preceded by a slash, to indicate their position and prevent confusion with other directories that could have the same name. When starting with a new system, it is always a good idea to take a look in the root directory. Let's see what you could run into:

DIRECTORY	CONTENT
/bin	Common programs, shared by the system, the system administrator and the users.
/boot	The startup files and the kernel, vmlinuz. In some recent distributions also grub data. Grub is the GRand Unified Boot loader and is an attempt to get rid of the many different boot-loaders we know today.

DIRECTORY	CONTENT
/dev	Contains references to all the CPU peripheral hardware, which are represented as files with special properties.
/etc	Most important system configuration files are in /etc, this directory contains data similar to those in the Control Panel in Windows
/home	Home directories of the common users.
/initrd	(on some distributions) Information for booting. Do not remove!
/lib	Library files, includes files for all kinds of programs needed by the system and the users.
/lost+found	Every partition has a lost+found in its upper directory. Files that were saved during failures are here.
/misc	For miscellaneous purposes.
/mnt	Standard mount point for external file systems, e.g. a CD-ROM or a digital camera.
/net	Standard mount point for entire remote file systems
/opt	Typically contains extra and third party software.
/proc	A virtual file system containing information about system resources. More information about the meaning of the files in proc is obtained by entering the command **man proc** in a terminal window. The file proc.txt discusses the virtual file system in detail.
/root	The administrative user's home directory. Mind the difference between /, the root directory and /root, the home directory of the *root* user.
/sbin	Programs for use by the system and the system administrator.
/tmp	Temporary space for use by the system, cleaned upon reboot, so don't use this for saving any work!
/usr	Programs, libraries, documentation etc. for all user-related programs.
/var	Storage for all variable files and temporary files created by users, such as log files, the mail queue, the print spooler area, space for temporary storage of files downloaded from the Internet, or to keep an image of a CD before burning it.

Table 2-1. Subdirectories of the root directory

User Group and Resource Management

A *user* is anyone who uses a computer. In this case, we are describing the names which represent those users. It may be Mary or Bill, and they may use the names Dragonlady or Pirate in place of their real name. All that matters is that the computer has a name for each account it creates, and it is this name by which a person gains access to use the computer. Some system services also run using restricted or privileged user accounts.

Managing users is done for the purpose of security by limiting access in certain specific ways. The superuser (root) has complete access to the operating system and its configuration; it is intended for administrative use only. Unprivileged users can use the su and sudo programs for controlled privilege escalation. Any individual may have more than one account, as long as they use a different name for each account they create. Further, there are some reserved names which may not be used such as "root". Users may be grouped together into a "group", and users may be added to an existing group to utilize the privileged access it grants. Linux/Unix

operating systems have the ability to multitask in a manner similar to other operating systems. However, Linux's major difference from other operating systems is its ability to have multiple users. Linux was designed to allow more than one user access to the system at the same time. In order for this multiuser design to work properly, there needs to be a method to protect users from each other. This is where permissions come in to play.

File system Permissions

Read, Write & Execute Permissions

Permissions are the "rights" to act on a file or directory. The basic rights are read, write, and execute. Read - A readable permission allows the contents of the file to be viewed. A read permission on a directory allows you to list the contents of a directory. Write - A write permission on a file allows you to modify the contents of that file. For a directory, the write permission allows you to edit the contents of a directory (e.g. add/delete files). Execute - For a file the executable permission allows you to run the file and execute a program or script. For a directory, the execute permission allows you to change to a different directory and make it your current working directory. Users usually have a default group, but they may belong to several additional groups.

Viewing File Permissions

To view the permissions on a file or directory, issue the command ls -l <directory/file>. Remember to replace the information in the < > with the actual file or directory name. Below is sample output for the ls command: -rw-r--r-- 1 root root 1031 Nov 18 09:22 /etc/passwd

The first ten characters show the access permissions. The first dash (-) indicates the type of file (d for directory, s for special file, and - for a regular file). The next three characters (**rw-**) define the owner's permission to the file. In this example, the file owner has read and write permissions only. The next three characters (**r--**) are the permissions for the members of the same group as the file owner (which in this example is read only). The last three characters (**r--**) show the permissions for all other users and in this example it is read only.

Access permissions and security

The Unix operating system (and likewise, Linux) differs from other computing environments in that it is not only a *multitasking* system but it is also a *multi-user* system as well. The computer would support many users at the same time. In order to make this practical, a method had to be devised to protect the users from each other. After all, you could not allow the actions of one user to crash the computer, nor could you allow one user to interfere with the files belonging to another user. Linux uses the same permissions scheme as Unix. Each file and directory on your system is assigned access rights for the owner of the file, the members of a group of related users, and everybody else. Rights can be assigned to read a file, to write a file, and to execute a file (i.e., run the file as a program). To see the permission settings for a file, we can use the ls command as follows:

[me@linuxbox me]$ ls -l some_file

-rw-rw-r-- 1 me me 1097374 Sep 26 18:48 some_file

We can determine a lot from examining the results of this command:

- The file "some_file" is owned by user "me"
- User "me" has the right to read and write this file
- The file is owned by the group "me"
- Members of the group "me" can also read and write this file

- Everybody else can read this file

Let's try another example. We will look at the bash program which is located in the /bin directory:

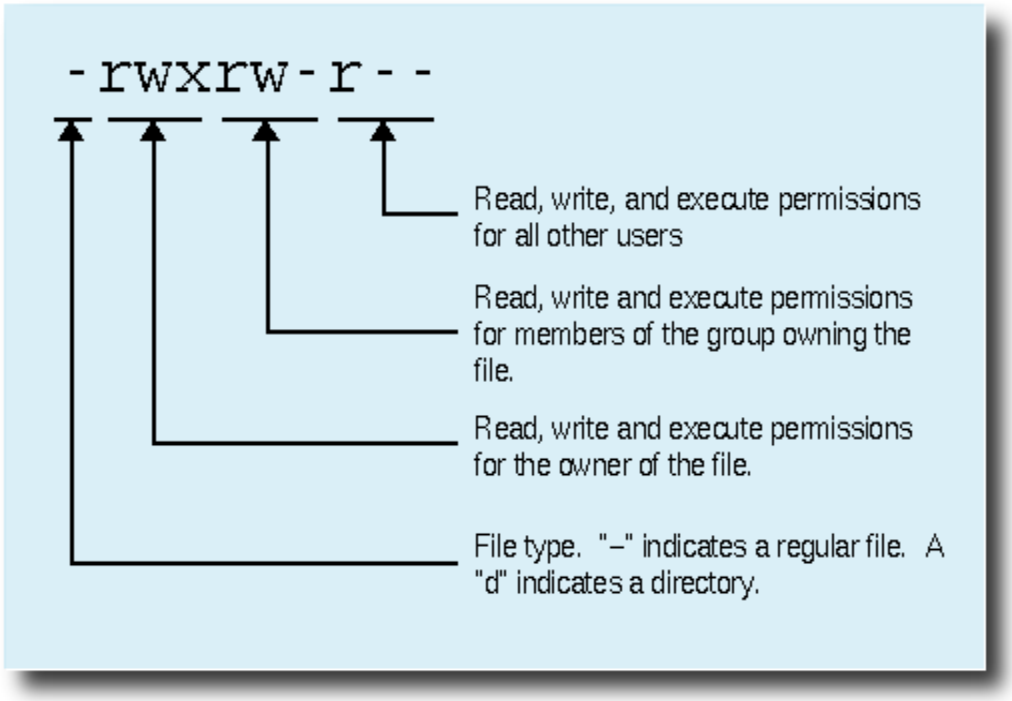
[me@linuxbox me]$ ls -l /bin/bash

-rwxr-xr-x 1 root root 316848 Feb 27 2000 /bin/bash

Here we can see:

- The file "/bin/bash" is owned by user "root"
- The superuser has the right to read, write, and execute this file
- The file is owned by the group "root"
- Members of the group "root" can also read and execute this file
- Everybody else can read and execute this file

In the diagram below, we see how the first portion of the listing is interpreted. It consists of a character indicating the file type, followed by three sets of three characters that convey the reading, writing and execution permission for the owner, group, and everybody else.



chmod

The chmod command is used to change the permissions of a file or directory. To use it, you specify the desired permission settings and the file or files that you wish to modify.

It is easy to think of the permission settings as a series of bits (which is how the computer thinks about them). Here's how it works:

rwx rwx rwx = 111 111 111
rw- rw- rw- = 110 110 110
rwx --- --- = 111 000 000

and so on...

rwx = 111 in binary = 7
rw- = 110 in binary = 6
r-x = 101 in binary = 5

r-- = 100 in binary = 4

Now, if you represent each of the three sets of permissions (owner, group, and other) as a single digit, you have a pretty convenient way of expressing the possible permissions settings. For example, if we wanted to set some_file to have read and write permission for the owner, but wanted to keep the file private from others, we would:

[me@linuxbox me]$ chmod 600 some_file

Here is a table of numbers that covers all the common settings. The ones beginning with "7" are used with programs (since they enable execution) and the rest are for other kinds of files.

Value	*Meaning*
777	*(rwxrwxrwx)* No restrictions on permissions. Anybody may do anything. Generally not a desirable setting.
755	*(rwxr-xr-x)* The file's owner may read, write, and execute the file. All others may read and execute the file. This setting is common for programs that are used by all users.
700	*(rwx------)* The file's owner may read, write, and execute the file. Nobody else has any rights. This setting is useful for programs that only the owner may use and must be kept private from others.
666	*(rw-rw-rw-)* All users may read and write the file.
644	*(rw-r--r--)* The owner may read and write a file, while all others may only read the file. A common setting for data files that everybody may read, but only the owner may change.
600	*(rw-------)* The owner may read and write a file. All others have no rights. A common setting for data files that the owner wants to keep private.

Directory permissions

The chmod command can also be used to control the access permissions for directories. In most ways, the permissions scheme for directories works the same way as they do with files. However, the execution permission is used in a different way. It provides control for access to file listing and other things. Here are some useful settings for directories:

Value	*Meaning*
777	*(rwxrwxrwx)* No restrictions on permissions. Anybody may list files, create new files in the directory and delete files in the directory. Generally not a good setting.
755	*(rwxr-xr-x)* The directory owner has full access. All others may list the directory, but cannot create files nor delete them. This setting is common for directories that you wish to share with other users.
700	*(rwx------)* The directory owner has full access. Nobody else has any rights. This setting is useful for directories that only the owner may use and must be kept private from others.

Common file system commands

Command	**Meaning**
cat file(s)	Send content of file(s) to standard output.
chmod *mode* file(s)	Change access permissions on file(s)
cp sourcefile targetfile	Copy sourcefile to targetfile.
echo *string*	Display a line of text
file filename	Determine file type of filename.
locate *searchstring*	Print all accessible files matching the search pattern.
ls file(s)	Prints directory content.
mkdir newdir	Make a new empty directory.
mv oldfile newfile	Rename or move oldfile.
Pwd	Print the present or current working directory.
rm file	Removes files and directories.
rmdir file	Removes directories.
wc file	Counts lines, words and characters in file.