

Permissions and Rights in Linux

BY CAROLYN GILLAY & PATRICIA SULLIVAN

Linux, as an operating system based on Unix, is designed as a multiuser system; that is, many users can work with the system at one time, making security paramount. To provide this security, every user must be a valid user. You are considered a valid user if you can successfully log on, which means the system administrator (super user) has created an account for you with a user name and password giving you access to the system as an approved user. The kind of access you have is also determined by the system administrator and is dependent on the groups you belong to and the assignment of permissions to you and the groups you belong to.

In a secure system, you

do not want to allow an individual user to modify, add, or delete files of another user. In fact, you may not even want an individual user to be able to read another user's files. If you are using Linux on a computer as a stand-alone operating system, obviously, you will not have several users logged at a time. Nonetheless, you are still bound by the architecture of a multiuser system that provides security for all users.

To provide this security, and the ability to manage users and their files, Linux/Unix assigns each file and directory permissions. Permissions are granted by the owner of the files and directories. Every file and directory

owned by someone, usually the user that created them. Although permissions could be granted on a user-by-user basis, this would be logistically difficult to manage. Hence, permissions are usually granted to groups. A group is a set of users to whom the owner can grant permissions.

Permissions refer to the way a file can be used by an individual user. There are three file permissions: read (r), write (w) and execute (x). A read permission gives you the right to look at a file's contents. A write permission gives you the right to alter or delete a file. An execute permission gives you the right to execute a program. For directories, the permissions are the same—read/write/execute—but have different meaning. A read permission gives you the right to list the contents of a directory.

In a secure system, you do not want to allow an individual user to modify, add, or delete files of another user. In fact, you may not even want an individual user to be able to read another user's files.

A write permission gives you the right to add or remove files in that directory. You may not separate these permissions, i.e., you cannot give rights to add files to a directory but deny permission to remove a file from that directory. An execute permission gives you the right to list information about files in that directory. Normally in a directory, both read and execute permissions are paired and either both are given or denied to a user.

Whenever a file or directory is created, the operating system assigns a set of default permissions. Many utility programs also assign permissions based on certain default criteria. To determine who gets what permissions, Linux allows three levels of permissions: owner, group and other. The owner is generally the user who created the file. Each user also belongs to a default group, and that group is also assigned to every file the user creates. Many groups can be created, however, and users can belong to multiple groups. Thus, you have three types of permissions (r, w, x) that can be assigned to three groups (owner, group, other) for a total of

nine access bits.

One user, the super user, has special privileges because the super user, in a sense, owns everything. The super user is generally the creator of new groups, and the super user has the power to change file ownerships and bypass permissions that the actual owner of the file or directory may have set. The super user also uses his root account to perform administrative functions such as maintenance of the system. There are also other accounts on the system that are not intended for human interaction at all. These types of accounts are used generally by system daemons, which access files on the system through a specific system user ID other than root or an ordinary user account. In addition to regular files and directories, there are also special files such as devices and sockets. A device is some unit of hardware inside or outside the system unit capable of providing input or receiving output or both. Sockets provide a method for communication between a client program and a server program in a network. A socket is defined as “the endpoint in a connection.”

Sockets are created and used with a set of programming requests sometimes called the sockets application programming interface. The most common sockets, API, is the Berkeley Unix interface for sockets. Sockets can also be used for communication between processes within the same computer.

File Systems

These permissions and rights apply only if you are working with a secure file system such as the native **ext2** file system for Linux. If you are using a file system such as FAT, there are no available permissions or file ownership, even though those permissions are displayed. To test ownership and permissions, you want to create a floppy disk that has the Linux file system, **ext2**, on it. This means you must place the Linux file system on a newly formatted floppy disk. There are two parts to creating a file system on any device. First it is formatted, which is considered a low-level format (laying of the track and sector information), and second, the file system of choice is placed on the disk.

The format command

for a floppy disk is **fdformat [options] device**. The available option is **-n**, which is “no verify.” This option will disable the verification that is performed after the format. You must know the actual name of the device, sometimes called the raw device file name. Usually, the floppy device name is **/dev/fd0** or **/dev/fd1**. In order to format a floppy disk, it must not be mounted. If the disk is mounted, Linux considers the device busy and cannot format it. You may also specify a code for the type of drive (**d**—low-density 5¼-inch disk; **D**—low-density 3½-inch disk; **h**—high-density 5¼-inch disk; **H**—high-density 3½-inch disk) and the size of the disk (360, 720, 1200, 1440). Thus, you could key in the command as **fdformat /dev/fd0d360** to format a 5¼-inch low-density floppy disk or **fdformat /dev/fd0H1440** to format a 3½-inch high-density disk. However, since most floppy disks today are 3½-inch high-density disks, you do not need to add the type or size information. Furthermore, since most people purchase preformatted disks, you do not have to low-level format the disk.

To create a file system, you use the **mkfs** command. A file system is what determines how file are named, stored, and retrieved on a device. Every file system has its own **mkfs** command associated with it. Thus, for instance, for a MS-DOS file system, the command is **mkfs.msdos** and for the native Linux file system, **ext2**, the command is **mkfs.ext2**. The command **mke2fs** is equivalent to **mkfs.ext2**. The command **mkfs, fs-type** tells **mkfs** to execute the correct version of **mkfs**. If you do not have the front-end program, **mkfs**, then you directly use **mke2fs** or **mkfs.ext2**. Major file systems are listed in Table 1—Major Linux File System Types.

Type	File System	Description
ext2	Second Extended file system	Native Linux file system. NFS Network File System Allows access to remote files on a network.
umsdos	UMSDOS File System	Installs Linux on an MS-DOS partition. msdos DOSO-FAT File System Accesses MS-DOS files.
vfat	VFAT File System	Accesses Windows 95/98 files. ntfs NT File System Accesses Windows NT files.
iso9660	ISO9660 File System	Used by most CD-ROMs.
hfs	Apple Mac File System	Accesses files from Apple Macintosh.
ncpfs	Novell File System	Access files from a Novell server.
smbfs	SMB File System	Accesses files from a Windows for Workgroups or Windows NT server.

Table 1—Major Linux File System Types

The syntax for the command is: **mkfs -t type device blocks**. Type is the type of file system you wish to create, as listed in Table 1. Device is the name of the device. This must be a “real” device name such as **/dev/fd0**. You cannot use the **/mnt/floppy** name. Blocks refers to the

size of the file system in 1024-byte blocks. Thus, using the above command to format a 3½-inch floppy disk in the Linux file system, you would issue the command as **mkfs -t ext2 /dev/fd0 1440**. If you wanted the 3½-inch floppy disk to be formatted with the MS-DOS file system, you would issue the command as **mkfs -t msdos /dev/fd0 1440**. Once the disk is formatted, and has a file system on it, it can be mounted.

Disk Use

The space available on disk is always of concern. Eventually, no matter what size your disk, you will run out of room. In addition, Linux runs most efficiently with at least 5 to 30 per cent of the disk space free in each type of file system. The minimum amount of disk space you should leave free is machine-dependent. Using the maximum amount of disk space will degrade performance and, obviously, if you fill a disk, you will not be able to add any more files to it.

The reason you need free disk space is that when Linux processes run, they generate a number of housekeeping files to track what it is doing. Furthermore, in any operating system, to maintain the file system and perform other system-related tasks, there is overhead. Overhead are the needed resources, (usually processing time or storage space), that are used for purposes incidental to, but necessary to the main one. In a retail business, for instance, overhead consists of items like the cost of heating and cooling the building and the cost of electricity and supplies. None of these items has anything directly to do with selling merchandise, but the business cannot be run without the overhead items. It is the “cost” of doing business. The analogy works with computers—there is a “cost” for doing business of running and maintaining the system, which includes the file system. There are also always cost trade-offs. For instance, if you keep a program running all the time, you save the overhead costs of loading and initializing the program for each transaction. But by running the program all the time, you now have introduced a space overhead consideration. For instance, in Linux, a directory can hold any number of files, it is generally a good use of space to keep the number of files in any single directory reasonably small. Not only can too many files in a directory degrade performance, but, the directory file increases in size. This is true because deleting files does shrinks size of the directory file.

Two programs that can help you monitor disk space are **du** (disk usage) and **df** (disk free). The **du** command reports how much space is used by a directory, including its subdirectories, or a specific file. It displays the number of blocks that are occupied by the directory or file.

The syntax for **du** is **du [options] [file]**. Some useful options include:

- a All—displays the count for all files, not just directories.
- b Bytes—displays the size in bytes.
- c Total—displays a grand total.

- h Human readable—displays output in bytes, kilobytes or megabytes.
- k Kilobytes—displays output in bytes.
- m Megabytes—displays output in bytes.
- s Summarize—Displays a total.

The syntax for `df` is **`df [options] [file]`**. Some useful options include:

- a All—displays all file systems including those with 0 blocks.
- h Human readable—displays output in bytes, kilobytes or megabytes.
- i Inodes—displays inode information instead of block usage.
- k Kilobytes—displays output in block sizes of 1024.
- m Megabytes—displays output in block sizes of 1048576.

Groups

Permissions, file ownership, and groups are dependent on one another, and different types of permissions can be assigned to the different levels of groups. When a new user account is created, such as **student**, that account is assigned a numeric user id (UID) as well. In Red Hat Linux, the default is to use a number greater than 500 that is greater than all other existing user IDs.

When Red Hat Linux creates a new user account, it also creates a new group for every new user and, by default, every user belongs to that group. That group is assigned to every file the user creates. The group name has the same name as the user name. If the user name were **student**, then that user's group name is also **student**, and user **student** would belong to the group **student**. Red Hat Linux also assigns a numeric value to that group (GID). The default is to use the smallest number greater than 500 that is greater than all other group ids. Thus, if **cgillay** were the first user, the **cgillay**'s user id would be 500 and there would be a group **cgillay** with a group id of 500. If **student** were the second user, the user id would be 501 and **student** would belong to the **student** group with a group id of 501. Supplementary groups can be created and users can belong to multiple groups.

All the information about users and groups are kept in two files. User information is stored in the `/etc/passwd` file. Group information is kept in the `/etc/group` file. Only the super user (root) may create users and groups. These files can be edited directly by the root user.

There are several commands to manage groups and ownership of files and directories. These include:

- `chgrp` Change the group of a file or directory. Only the owner or root can use this command. The syntax is `chgrp [option] newgroup file`, where `newgroup` is either a group name or group ID number. Useful options include `-c`, which reports when a change is made; `-R`, which operates recursively on files and directories and `-v`, which displays the results of the command.

- groups Allows you to see to which groups you belong.
- chown Change the owner of a file. Only root may use this command.

Permissions

There are three permission types—read, write and execute. When these permissions are applied to files, they grant the user the rights to:

- read (r) Allowed to look at the file’s contents.
- write (w) Allowed to change the contents of the file or delete the file.
- execute (x) Allowed to run the file as a program, if it is, of course, a program.

When these permissions are applied to directories, they grant the user the rights to:

- read (r) Allowed to list the contents of a directory.
- write (w) Allowed to add or remove files from a directory.
- execute (x) Allowed to access files and subdirectories, which is the ability to change into this directory with **cd**. Within a directory structure, normally **r** and **x** are paired.

When a user creates a file or a directory, the system normally assigns a default set of permissions. The user can change these default permissions for files and directories he has created and assign different levels of permissions to the different types of users. Thus, the elements that make up the security system for files and directories in Linux can be summarized as follows: there are three levels of permission (read, write, and execute), and those permissions may be assigned to three types of users (user, group, and other). If you look at a file, **april.txt**, using the **ls-l** command, you would see the display

```
-rwxr-x--- 1 cgillay student 86 Jan 21 10:30 april.txt
```

Using what you have learned, you can see the breakdown of permissions and ownership as exemplified in Table 2—Ownership and Permissions.

File Type	Owner Permissions	Group Permissions	Other Permissions	Owner	Group
-	rwx	r-x	---	cgillay	student
Ordinary file	The owner can read, write and execute the file	The group can read and execute the file.	All others have no access to the file.	The owner is cgillay.	The group that can access the file is student. Student has the permissions specified under group.

Table 2—Ownership and Permissions

There are some differences when you look at directory entries, one that you might create and a home directory. A typical directory entry might be

```
drwxr-xr-x 2 cgillay cgillay 4086 Jan 21 10:30 cgillay
```

The breakdown of the listing is shown in Table 3—Directory Ownership and Permissions.

File Type	Owner Permissions	Group Permissions	Other Permissions	Owner	Group
d Directory	rwX The owner can add or remove files from the directory as well as list the contents of the directory and	r-X The group can list the contents of the directory but may not add or delete files in it. The group member can	r-X All others can list the contents of the directory but may not add or delete files in it. Others can change to the directory.	cgillay The owner is cgillay	cgillay The group that can access the directory is cgillay.

Table 3—Directory Ownership and Permissions

A typical home directory entry might be

drwx----- 2 cgillay cgillay 4086 Oct 21 10:30 cgillay

The breakdown of the listing is shown in Table 4—Home Directory Ownership and Permissions.

File Type	Owner Permissions	Group Permissions	Other Permissions	Owner	Group
d Directory	rwX The owner can add or remove files from the directory as well as list the contents of the directory and change to the directory.	--- The group cannot add or delete files in the directory. In fact, they can not even list the contents of the directory nor can they change to the directory.	--- All others cannot add or delete files in the directory. In fact, they cannot even list the contents of the directory nor can they change to the directory.	cgillay The owner is cgillay.	cgillay The group that can access the directory is cgillay.

Table 4—Directory Ownership and Permissions

The major command to manage the permissions on files and directories is **chmod**.

chmod Change permissions for a file or directory. Only the owner or root can use this command. The syntax is **chmod [options] mode[,mode].. file** or **chmod [options] octal-mode file**.

To use the **chmod** command, you use a **+** (plus) to add permissions and a **-** (minus) to remove permissions. An **=** (equal) sets the permissions to exactly what was specified and removes permissions of fields that are unspecified. You may assign those permissions to **u** (user), **g** (group), **o** (other) or **a** (all). You use the same syntax to apply permissions to directories, except that **x** (execute) has a different meaning. In a directory, execute permission means that a user has the right to navigate through the directory structure and change to that directory. Thus, a user could have a file that was readable by all users, but if he placed it in a directory that had no execute privileges, then no user could read the file even though each user had read and write permissions to the file.

11 Octal Permissions

To alter permissions, you have been using symbolic mode. It was symbolic because you were using letters such as `-r`. There is another mode called numeric or absolute mode which causes you to use bits and octal notation. You need to understand both modes because there are some occasions where symbolic mode will not work. You will also often find that absolute mode rather than symbolic mode is used in documentation. Furthermore, at times you may find absolute mode more convenient.

A typical mode contains three characters, matching the three levels of permissions (**user**, **group**, and **other**). In addition, within each level, there are three bits matching to read, write and execute permissions. Figure 1—Absolute Mode demonstrates this.

User			Group			Other		
read	write	execute	read	write	execute	read	write	execute
400	200	100	40	20	10	4	2	1

Figure 1—Absolute Mode

You sum each number in each column to give octal permissions. Thus, for a file called **jan.99**, if you wanted to give read permission to everyone, you would choose the correct number from each column—400 for the user, + 40 for the group, + 4 for everyone else. The command would then read **chmod 444 jan.99**. If you were using symbolic mode, you could issue the command as **chmod =r jan.99**. If you keyed in **chmod +r jan.99**, you would be granting read permission to everyone, but not removing other permissions. Using the **=r** will assign read permission for everyone and at the same time remove all other permissions.

Another way to look at the numeric mode of setting permissions is to assume read permissions are equal to 4, write permissions are equal to 2, execute permissions are equal to 1, and no permissions are equal to 0. You could then build a table and sum the permissions to reach the correct number to use with `chmod`. For instance, if you keyed in **chmod ug=rw,o=r jan.99**, for the file **jan.99** you would be assigning read and write permissions to both user and group and only read permission to others. Table 5—Numeric Permissions gives an example of this.

Permission	Owner	Group	Other
Read	4	4	4
Write	2	2	0
Execute	0	0	0
Total	6	6	4

Table 5—Numeric Permissions

The command could then be issued as **chmod 664 jan.99**. If you use the table in Figure 1, it also will derive the number 664 (read 400 + write 200 for user = 600; read 40 + 20 write for group = 60; and read = 4 for others, which still totals 664). Another example might be if you install a file and want to make it available to everyone to execute to everyone on the system. You want to retain read, write, and execute permissions for the user, but only execute permissions for members of the group and the rest of the world. Symbolically, you could write the command as `chmod u=rwx,g=x,o=x january`, or you could derive the numeric value as shown in Table 6—Numeric Permissions.

Permission	Owner	Group	Other
Read	4	0	0
Write	2	0	0
Execute	1	1	1
Total	7	1	1

Table 6—Numeric Permissions

The command would then be **chmod 711 january**. If you use the table in Figure 1, it also will derive the number 711 (read 400 + write 200 + execute 100 for user = 700; execute 10 group = 10; and execute = 1 for others, which still totals 711). As you work with numeric values, you will find common numeric codes that are most often used. Table 7—Commonly Used Absolute Mode.

Absolute Permission	Symbolic Permission	Meaning
600	u=rw,g=,o=	Owner has read and write permission.
644	u=rw,g=r,o=r	Owner has read and write permission. Group and others have read-only permission.
666	u=rw,g=rw,o=rw	Everyone has read and write permissions.
700	u=rwx,g=,o=	Owner has read, write, and execute permissions. If applied to a directory, only the owner has can read and write to the directory. Directories must have the execute bit set.
710	u=rwx,g=x,o=	Owner has read, write, and execute permissions. Group has execute permission.
711	u=rwx,g=x,o=x	Owner has read, write, and execute permissions. Group and other has execute permission. This can be used with directories so that the group or others may not list the contents of a directory but would be able to retrieve a file from the directory if the file name is known.
750	u=rwx,g=rx,o=x	Owner has read, write, and execute permissions. Group has read and execute permission and other has execute permission.
755	u=rwx,g=rx,o=rx	Owner has read, write, and execute permissions. Group and others have read and execute permission. Only the owner can change the contents of the directory, but everyone may view the contents.
777	u=rwx,g=rwx,o=rwx	Everyone has read, write, and execute permissions.

Table 7—Commonly Used Absolute Mode

To further talk about permissions, whenever you create a file or a directory, Linux will assign default permissions to the file or directory. To disclose the default permission, you may use the **umask** command. The **umask** command is a “mask” that determines which permissions are on or off. The **umask** command is in the `/etc/profile` file. You may also place the **umask** command in your shell’s startup file if you want a different default value for what the default permissions are. In bash, the file is called `.bashrc`. If you wanted to find the default permissions assigned to files, you would subtract the value found with **umask** from 666 (read/write permissions for owner, group, and other). If you wanted to find the default permissions assigned to directories, you would subtract the value found with **umask** from 777 (read/write/execute permissions for owner, group, and other). Thus, if you keyed in **umask** and the value 027 were returned, the results would be 640 (read/write permissions for the owner, read permissions for the group, and no permissions for the other). If applied to a directory, the result would be 750 (read/write/execute permission for the owner, read and execute permissions for the group, and no permissions for others). However, when doing the subtraction, for each digit, if any number is less than zero, you use a zero. Table 8—umask Values

umask applied to file	umask applied to directory
666	777
<u>-027</u>	<u>-027</u>
640	750

Table 8—umask Values Permission

Permissions shows these two examples.

There is an even easier way to determine the mask. You can use the **-S** option with **umask**, which will print the symbolic form. The **umask** command allows you to set certain file permissions without having to set individual permissions for every file you create. For security reasons, when you create a file, the execute permission is left off; however, the execute permission is included in the directory permissions. If you want to change your **umask** values, you can use Table 9—umask Values.

umask	File Permissions	Directory Permissions
7	None	None
6	None	Execute (x)
5	Write (w)	Write (w)
4	Write (w)	Write & Execute (wx)
3	Read (r)	Read (r)
2	Read (r)	Read & Execute (rx)
1	Read & Write (rw)	Read & Write (rw)
0	Read & Write (rw)	Read, Write & Execute (rwx)

Table 9—umask Values

Each **umask** value would be placed in the corresponding position—user, group, and other. So if you assigned a umask of 001, files that were created would be assigned read and write permissions for the user, group, and other; and directories that were created would be assigned read, write, and execute permissions for the owner and group, and read and write permissions for others. you would be giving the owner (user) read and write permissions.