Module 3 - Files, directory administration and permissions

By Juan Medina

jmedina@collin.edu

COLLIN COLLEGE

File Types

Most file-system implementations define seven types of files.

Symbol	Meaning
-	Regular file
d	Directory
l	Symbolic Link
c	Character device File / Special File
s	Local Domain Sockets
p	Named Pipes (FIFOs)
b	Block Device

You can determine the type of an existing file with **ls -l** The first character of the ls output encodes the type.

```
$ ls -l
-rw-rw-r--. 1 jmedinar jmedinar     0 May 16 10:29  apple
drwxr-xr-x. 9 jmedinar jmedinar  4096 Jul  1 17:10  Documents
```

Regular files

Regular files consist of a series of bytes; file-systems impose no structure on their contents. Text files, data files, executable programs, and shared libraries are all stored as regular files. Both sequential access and random access are allowed.

Directories

A directory contains named references to other files. The special entries “.” and “..” refer to the directory itself and to its parent directory; they may not be removed. A file’s name is stored within its parent directory, not with the file itself. In fact, more than one directory can refer to a file at one time, and the references can have different names. Such an arrangement creates the illusion that a file exists in more than one place at the same time. These are called “hard links”

Character and block device files

Device files let programs communicate with the system's hardware and peripherals. The kernel loads) driver software for each of the system's devices. This software takes care of the messy details of managing each device so that the kernel proper can remain relatively abstract and hardware independent. Device drivers present a standard communication interface that looks like a regular file. When the file-system is given a request that refers to a character or block device file, it simply passes the request to the appropriate device driver. It's important to distinguish device files from device drivers, however.

Local domain sockets

Sockets are connections between processes that allow processes to communicate. Linux defines several kinds of sockets, most of which involve the use of a network. Local domain sockets are accessible only from the `localhost` and are referred to through a file-system object rather than a network port. They are sometimes known as "Linux domain sockets."

Named pipes

Like local domain sockets, named pipes allow communication between two processes running on the same host. They're also known as "FIFO files" "first-in, first-out". You can create named pipes with `mknod` and remove them with rm. Named pipes and local domain sockets serve similar purposes, and the fact that both exist is essentially a historical artifact. Neither of them would exist if Linux were designed today; network sockets would stand-in for both.

Symbolic links

A symbolic or "soft" link points to a file by name. When the kernel comes upon a symbolic link in the course of looking up a pathname, it redirects its attention to the pathname stored as the contents of the link. The difference between hard links
and symbolic links is that a hard link is a direct reference, whereas a symbolic link is a reference by name. Symbolic links are distinct from the files they point to.

What is really a file?

For most users and for most common system administration tasks. It is enough to accept that files and directories are ordered in a tree-like structure. The computer, however, doesn't understand a thing about trees or tree-structures.

Every partition has its own file system. In a file system, a file is represented by an ***inode***, that is basically an ID or a serial number that provides information about the actual data that makes up the file:

- Owner and group owner of the file.
- File type (regular, directory, ...)
- Permissions on the file
- Date and time of creation, last read, and change.
- Date and time this information has been changed in the inode.
- Number of links to this file
- File size

- An address defining the actual location of the file data.

Every partition has its own set of inodes; throughout a system with multiple partitions, files with the same inode number can exist.

The only information not included in an inode is the file name and directory. These are stored in the special directory files. By comparing file names and inode numbers, the system can make up a tree-structure that the user understands. You can display inode numbers using the -i option to ls.

```
[jmedinar@localhost ~]$ ls -i /etc/passwd
789070 /etc/passwd
```

And you can see the inode information with the stat command

```
[jmedinar@localhost ~]$ stat /etc/passwd
  File: /etc/passwd
  Size: 2662          Blocks: 8          IO Block: 4096   regular file
Device: fd00h/64768d Inode: 789070      Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Context: system_u:object_r:passwd_file_t:s0
Access: 2020-12-20 12:46:01.542750112 -0600
Modify: 2020-11-18 17:12:30.816245363 -0600
Change: 2020-11-18 17:12:30.845245797 -0600
 Birth: 2020-11-18 17:12:30.816245363 -0600
```

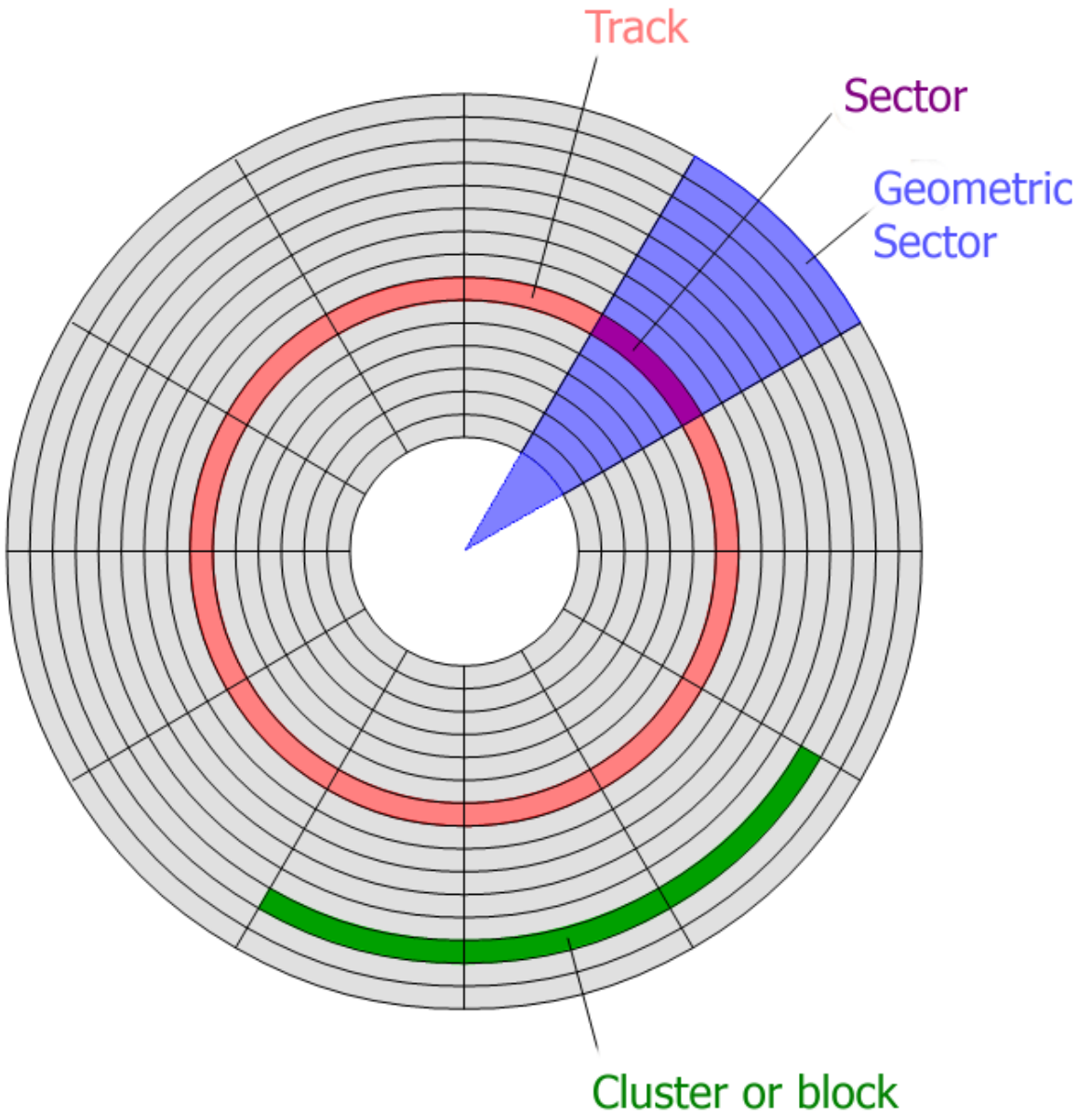
FileSystems

It is the way a computer organizes data in media like USB, HardDrive, DVD... etc. You can think of it as a database that contains the physical location of every piece of data stored... It is basically a huge index!

There are multiple types:

- FAT (Windows)
- NTFS (Windows)
- exFAT (Windows)
- HFS+ (iOS)
- ext2, 3 and 4 (Linux)
- JFS (AIX)
- NFS (Network FileSystem)
- GPFS (General Parallel FileSystem)

They all keep the data in "sectors" that are managed in "blocks" and the main differences between them are the block sizes, sector arrangement, fragmentation and writing and reading speed. Some have redundancy and fault-tolerance but most don't!. Some also support shared-disks arrays and in-network file transfer.


Track
Sector
Geometric
Sector
Cluster or block

File Attributes

Under the traditional Linux file-system model, every file has a set of nine permission bits that control who can read, write, and execute the contents of the file. Together with three other bits that primarily affect the operation of executable programs, these bits constitute the file's "mode."

The twelve mode bits are stored together with four bits of file-type information. The four file-type bits are set when the file is first created and cannot be changed, but the file's owner and the superuser can modify the twelve mode bits with the chmod (change mode) command.

```
[jmedinar@localhost ~]$ ls -l
-rw-rw-r--. 1 jmedinar jmedinar 23631 Mar 2 11:49 dirlist-bin.txt
```

Permissions & Ownership

Linux is a not only multitasking system but also a multi-user system. What exactly does this mean? It means that more than one person can be using the computer at the same time. This is a feature that is deeply embedded in the design of the operating system also requires a method to protect the users from each other. After all, the actions of one user could not be allowed to crash the computer, nor could one user interfere with the files belonging to another user.

Linux divides authorization into 2 levels.

1. Ownership
2. Permission

Ownership

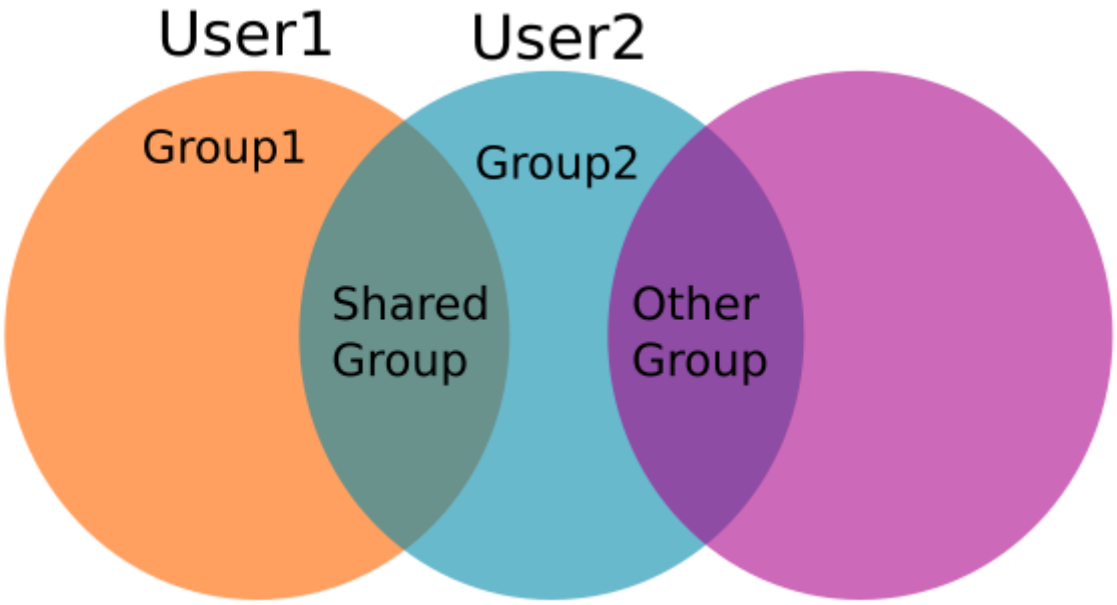
Every file and directory on your Linux system is assigned 3 types of persons that can access it.

User: A user is the owner of the file. By default, the person who created a file becomes its owner. Hence, a user is also sometimes called an owner.

Group: A user- group can contain multiple users. All users belonging to a group will have the same access permissions to the file.

Other: Any other user who has access to a file. This person has neither created the file, nor he belongs to a user-group who could own the file.

In the following example `User1` is a member of the groups `Group1` and `Shared Group`, and `User2` is a member of `Group2`, `Shared Group`, and `Other group`. When User1 and User2 need to collaborate, the files should be associated with the group `Shared Group` and group permissions should allow the desired access.



If you try to read the file `/etc/shadow` as a regular user you may face a problem because this file belongs to the root user.

```
[jmedinar@localhost ~]$ file /etc/shadow
/etc/shadow: regular file, no read permission
[jmedinar@localhost ~]$ less /etc/shadow
/etc/shadow: Permission denied
```

In the Linux security model, a user may own files and directories. When a user owns a file or directory, the user has control over its access. Users can, in turn, belong to a group consisting of one or more users who are given access to files and directories by their owners. In addition to granting access to a group, an owner may also grant some set of access rights to everybody, which in Unix terms is referred to as others. To find out information about your identity, use the id command:

```
[jmedinar@localhost ~]$ id
uid=1000(jmedinar) gid=1000(jmedinar)
groups=1000(jmedinar),10(wheel),971(vboxusers),973(docker)
```

When user accounts are created, users are assigned a number called a `user ID` or `uid` which is then, for the sake of the humans, mapped to a username. The user is assigned a primary `group ID` or `gid` and may belong to any number of additional groups.

User accounts are defined in the `/etc/passwd` file and groups are defined in the `/etc/group` file. When user accounts and groups are created, these files are modified along with `/etc/shadow` which holds information about the user's password.

The chown Command

A newly created file is owned by the user who creates that file. By default, new files have a group ownership that is the primary group of the user creating the file.

Only root can change the user that owns a file. Group ownership, however, can be set by root or by the file's owner. root can grant file ownership to any group, but regular users can make a group the owner of a file only if they are a member of that group.

File ownership can be changed with the chown (change owner) command.

```
1 [root@localhost ~]# chown student test_file
```

chown can be used with the -R option to recursively change the ownership of an entire directory tree.

```
1 [root@host ~]# chown -R student test_dir
```

The chown command can also be used to change group ownership of a file by preceding the group name with a colon (:).

```
1 [root@host ~]# chown :admins test_dir
```

The chown command can also be used to change both owner and group at the same time by using the owner:group syntax.

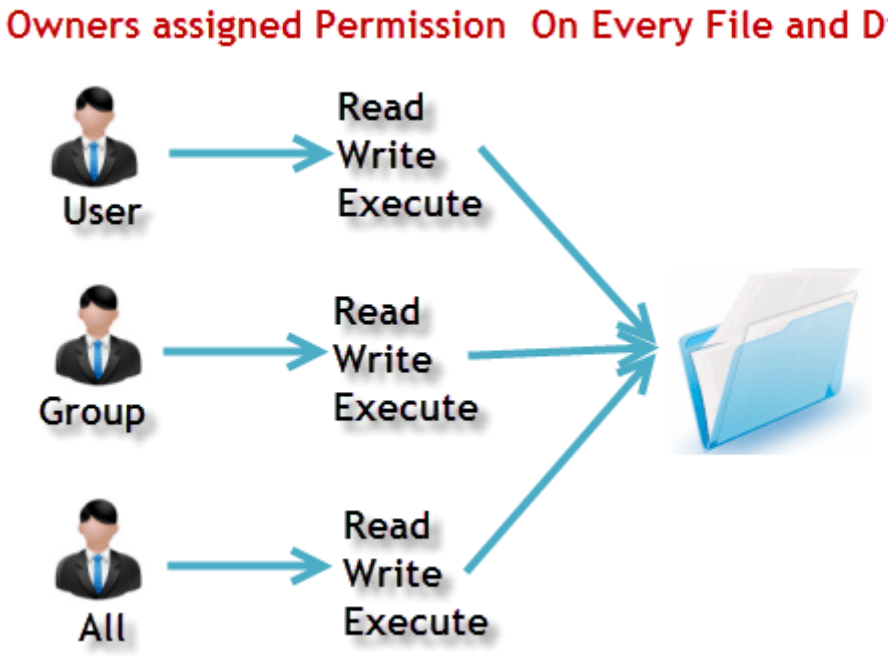
```
1 [root@host ~]# chown visitor:guests test_dir
```

Permissions

File permissions control access to files. The Linux file permissions system is simple but flexible, which makes it easy to understand and apply, yet still able to handle most normal permission cases easily.

Every file and directory in your Linux system has the following 3 permissions defined for all the 3 owners discussed above.

PERMISSION	EFFECT ON FILES	EFFECT ON DIRECTORIES
r (read)	Contents of the file can be read.	Contents of the directory (the file names) can be listed.
w (write)	Contents of the file can be changed.	Any file in the directory can be created or deleted.
x (execute)	Files can be executed as commands.	Contents of the directory can be accessed. (You can change into the directory, read information about its files, and access its files if the files permissions allow it.)



Note that users normally have both read and execute permissions on read-only directories so that they can list the directory and have full read-only access to its contents. If a user only has read access on a directory, the names of the files in it can be listed, but no other information, including permissions or time stamps, are available, nor can they be accessed. If a user only has execute access on a directory, they cannot list the names of the files in the directory, but if they already know the name of a file that they have permission to read, then they can access the contents of that file by explicitly specifying the file name.

A file may be removed by anyone who has write permission to the directory in which the file resides, regardless of the ownership or permissions on the file itself. This can be overridden with a special permission, the sticky bit, discussed later.

The -l option of the ls command shows more detailed information about file permissions and ownership:

```
1 [jmedinar@localhost ~]$ ls -l /etc/redhat-release
2 lrwxrwxrwx. 1 root root 14 Dec 15 09:38 /etc/redhat-release ->
  fedora-release
```

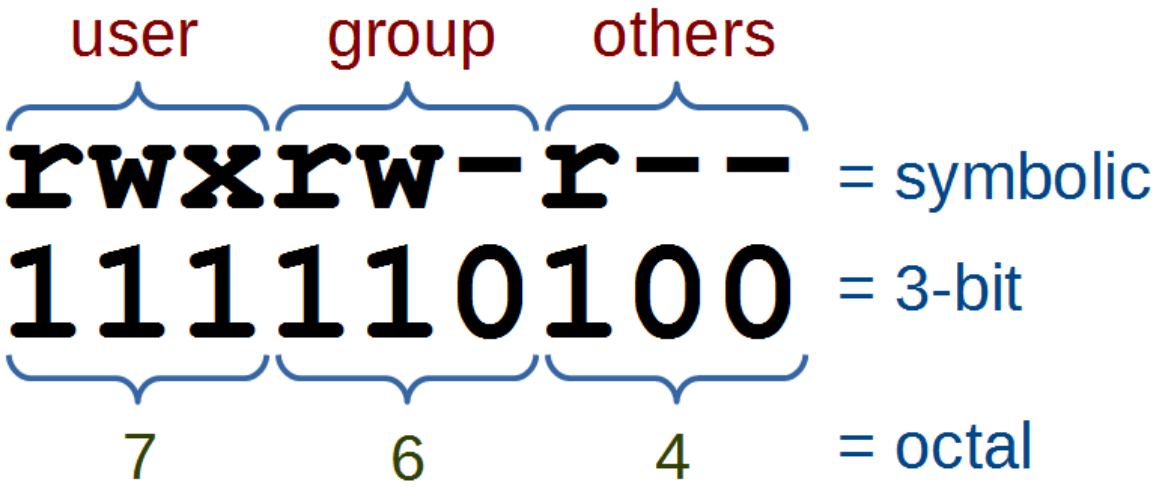
You can use the -d option to to show detailed information about a directory itself, and not its contents.

```
1 [jmedinar@localhost ~]$ ls -ld /home
2 drwxr-xr-x. 4 root root 4096 Jul 27 13:22 /home
```

The first character of the long listing is the file type. You interpret it like this:

- `-` is a regular file.
- `d` is a directory.
- `l` is a soft link.
- Other characters represent hardware devices (b and c) or other special-purpose files (p and s).

The next nine characters are the file permissions. These are in three sets of three characters: permissions that apply to the user that owns the file, the group that owns the file, and all other users.



The characters are pretty easy to remember.

- **r** = read permission
- **w** = write permission
- **x** = execute permission
- **-** = no permission

After the link count, the first name specifies the user that owns the file, and the second name the group that owns the file.

The chmod Command

The command used to change permissions from the command line is chmod, which means "change mode" (permissions are also called the mode of a file). The chmod command takes a permission instruction followed by a list of files or directories to change.

```
1 chmod permissions filename
```

The permission instruction can be issued in two ways:

1. Symbolic method
2. Numeric method

Symbolic Method

The symbolic method of changing file permissions uses letters to represent the different groups of users

User Denotations	
u	user/owner
g	group

User Denotations	
o	other
a	all

It makes use of mathematical notations to modify the file permissions.

Operator	Description
+	Adds a permission to a file or directory
-	Removes the permission
=	Sets the permission and overrides the permissions set earlier.

With the symbolic method, it is not necessary to set a complete new group of permissions. Instead, you can change one or more of the existing permissions.

```
chmod WhoWhatWhich file|directory
```

When using a capital `X` as the permission flag will add execute permission **only if the file is a directory** or already has execute set for user, group, or other.

When using the `-R` option it will recursively set permissions on the files in an entire directory tree.

Examples:

Changing permissions to OTHERS

```
[jmedinar@localhost ~]$ chmod o=rwx sample.txt
```

Changing permissions to the USER

```
[jmedinar@localhost ~]$ chmod u-x sample.txt
```

Changing permissions to the GROUP

```
[jmedinar@localhost ~]$ chmod g+r-x sample.txt
```

Changing permissions to ALL

```
[jmedinar@localhost ~]$ chmod a+x sample.txt
```

Remove read and write permission for group and other

```
[jmedinar@localhost ~]$ chmod go-rw sample.txt
```

Add execute permission for everyone

```
[jmedinar@localhost ~]$ chmod a+x sample.txt
```

Numeric Method

Using the numeric method, permissions are represented by a 3-digit octal number. A single octal digit can be any value from 0-7.

The first number represents the Owner permission; the second represents the Group permissions; and the last number represents the permissions for all other users. The numbers are a binary representation of the rwx string.

- r = 4
- w = 2
- x = 1

You add the numbers to get the integer/number representing the permissions you wish to set. For instance if you want to set Read, Write and Execution Permissions only for the OWNER you add...

Read + Write + Execute this is equal to 4 + 2 + 1 = 7

So you will apply this as follows:

```
[jmedinar@localhost ~]$ chmod 700 sample.txt
```

But if you want to give Read, Write and Execute Permissions to ALL USERS...

```
[jmedinar@localhost ~]$ chmod 777 sample.txt
```

The table below gives numbers for all for permissions types. (But it is easier to just learn the math!)

Number	Permission Type	Symbol
0	No Permission	---
1	Execute	--x
2	Write	-w-
3	Execute + Write	-wx
4	Read	r--
5	Read + Execute	r-x
6	Read +Write	rw-
7	Read + Write +Execute	rwx

Special Permissions

Special permissions constitute a fourth permission type in addition to the basic user, group, and other types. As the name implies, these permissions provide additional access-related features over and above what the basic permission types allow.

SPECIAL PERMISSION	EFFECT ON FILES	EFFECT ON DIRECTORIES
u+s (suid)	File executes as the user that owns the file, not the user that ran the file.	No effect.
g+s (sgid)	File executes as the group that owns the file.	Files newly created in the directory have their group owner set to match the group owner of the directory.
o+t (sticky)	No effect.	Users with write access to the directory can only remove files that they own; they cannot remove or force saves to files owned by other users.

Setuid/Setgid Special Permissions

The setuid/setguid permissions are used to tell the system to run an executable as the owner with the owner's permissions.

Be careful using setuid/setgid bits in permissions. If you incorrectly assign permissions to a file owned by root with the setuid/setgid bit set, then you can open your system to intrusion.

You can only assign the setuid/setgid bit by explicitly defining permissions. The character for the setuid/setguid bit is **s**.

```
[jmedinar@localhost ~]$ chmod g+s sample.txt
```

In a long listing, you can identify the setuid permissions by a lowercase s where you would normally expect the x (owner execute permissions) to be. If the owner does not have execute permissions, this is replaced by an uppercase S.

```
[jmedinar@localhost ~]$ ls -l /usr/bin/passwd
-rwsr-xr-x. 1 root root 35504 Jul 16 2010 /usr/bin/passwd
```

The special permission setgid on a directory means that files created in the directory inherit their group ownership from the directory, rather than inheriting it from the creating user. This is commonly used on group collaborative directories to automatically change a file from the default private group to the shared group, or if files in a directory should be always owned by a specific group. An example of this is the /run/log/journal directory:

```
[user@host ~]$ ls -ld /run/log/journal
drwxr-sr-x. 3 root systemd-journal 60 May 18 09:15 /run/log/journal
```

If setgid is set on an executable file, commands run as the group that owns that file, not as the user that ran the command, in a similar way to setuid works. One example is the locate command:

```
[user@host ~]$ ls -ld /usr/bin/locate
-rwx--s--x. 1 root slocate 47128 Aug 12 17:17 /usr/bin/locate
```

In a long listing, you can identify the setgid permissions by a lowercase s where you would normally expect the x (group execute permissions) to be. If the group does not have execute permissions, this is replaced by an uppercase S.

Sticky Bit Special Permissions

The sticky bit can be very useful in shared environment because when it has been assigned to the permissions on a directory it sets it so only file owner can rename or delete the said file.

You can only assign the sticky bit by explicitly defining permissions. The character for the sticky bit is **t**.

```
[jmedinar@localhost ~]$ chmod +t my_protected_directory
```

In a long listing, you can identify the sticky permissions by a lowercase t where you would normally expect the x (other execute permissions) to be. If other does not have execute permissions, this is replaced by an uppercase T.

```
[jmedinar@localhost ~]$ ls -ld /tmp
drwxrwxrwt. 39 root root 4096 Feb 8 20:52 /tmp
```

Default file permissions

When you create a new file or directory, it is assigned initial permissions. There are two things that affect these initial permissions. The first is whether you are creating a regular file or a directory. The second is the current `umask` value.

If you create a new directory, the operating system starts by assigning it octal permissions `0777 (drwxrwxrwx)`. If you create a new regular file, the operating system assignes it octal permissions `0666 (-rw-rw-rw-)`. You always have to explicitly add execute permission to a regular file. This makes it harder for an attacker to compromise a network service so that it creates a new file and immediately executes it as a program.

However, the shell session will also set a `umask` to further restrict the permissions that are initially set. This is an octal bitmask used to clear the permissions of new files and directories created by a process. If a bit is set in the umask, then the corresponding permission is cleared on new files.

For example, the umask 0002 clears the write bit for other users. The leading zeros indicate the special, user, and group permissions are not cleared. A umask of 0077 clears all the group and other permissions of newly created files.

The umask command without arguments will display the current value of the shell's umask:

```
[jmedinar@localhost ~]$ umask
0002
```

Use the umask command with a single numeric argument to change the umask of the current shell. The numeric argument should be an octal value corresponding to the new umask value. You can omit any leading zeros in the umask.

The system's default umask values for Bash shell users are defined in the `/etc/profile` and `/etc/bashrc` files. Users can override the system defaults in the `.bash_profile` and `.bashrc` files in their home directories.

Working with Files & Directories

At this point, we are ready for some real work!. The following commands are among the most frequently used Linux commands. They are used for manipulating both files and directories.

ACTIVITY	COMMAND SYNTAX
Create a directory	mkdir directory
Copy a file	cp file new-file
Copy a directory and its contents	cp -r directory new-directory
Move or rename a file or directory	mv file new-file
Remove a file	rm file
Remove a directory containing files	rm -r directory
Remove an empty directory	rmdir directory
Create a link	ln origin destination

To manage files, you need to be able to create, remove, copy, and move them. You also need to organize them logically into directories, in which you also need to be able to apply the same operations.

mkdir – Create Directories

The mkdir command creates one or more directories or subdirectories. It takes as arguments a list of paths to the directories you want to create.

```
1 [jmedinar@localhost ~]$ mkdir dir1
```

You can create multiple directories in a list

```
1 [jmedinar@localhost ~]$ mkdir dir1 dir2 dir3
```

The mkdir command will fail with an error if the directory already exists, or if you are trying to create a subdirectory in a directory that does not exist.

The -p (parent) option creates missing parent directories for the requested destination. Use the `mkdir -p` command with caution, because spelling mistakes can create unintended directories without generating error messages.

```
[jmedinar@localhost ~]$ mkdir -p dir1/child1/child2
```

cp – Copy Files And Directories

The cp command copies a file, creating a new file either in the current directory or in a specified directory. It can also copy multiple files to a directory.

It can be used two different ways. First to copy the single file or directory "item1" as "item2".

```
[jmedinar@localhost ~]$ cp item1 item2
```

or to copy multiple items into a different directory.

When copying multiple files with one command, the last argument must be a directory. Copied files retain their original names in the new directory. If a file with the same name exists in the target directory, the existing file is overwritten. By default, the cp does not copy directories; it ignores them.

```
[jmedinar@localhost ~]$ cp items* directory/
```

If you want to copy a file to the current working directory, you can use the special `.` directory:

```
[jmedinar@localhost ~]$ cp /etc/hostname .
```

Use the copy command with the `-r` (recursive) option, to copy a directory and its contents to another directory.

```
[jmedinar@localhost ~]$ cp -r dir1 dir2
```

mv – Move And Rename Files

The mv command performs both file **moving** and file **renaming**, depending on how it is used. In either case, the original filename no longer exists after the operation. File contents remain unchanged. mv is used in much the same way as cp:

```
[jmedinar@localhost ~]$ mv item1 item2
```

rm – Remove Files And Directories

The rm command is used to remove (delete) files and directories, rm will not remove directories that contain files, unless you add the `-r` or `--recursive` option.

It is a good idea to verify your current working directory before removing a file or directory.

```
[jmedinar@localhost ~]$ pwd
/home/student/Documents
```

Use the rm command to remove a single file from your working directory.

```
[jmedinar@localhost ~]$ rm item
```

If you attempt to use the rm command to remove a directory without using the -r option, the command will fail.

```
[jmedinar@localhost ~]$ rm Thesis/Chapter1
rm: cannot remove `Thesis/Chapter1': Is a directory
```

Use the rm -r command to remove a subdirectory and its contents.

```
[jmedinar@localhost ~]$ rm -r Thesis/Chapter1
```

Be Careful With rm!. Linux do not have an undelete command. Once you delete something with rm, it's gone.

Linux assumes you're smart and you know what you're doing. Be particularly careful with wild-cards.

ln – Create Links

It is possible to create multiple names that point to the same file. There are two ways to do this: by creating a hard link to the file, or by creating a soft link (sometimes called a symbolic link) to the file. Each has its advantages and disadvantages.

The ln command is used to create either hard or symbolic links. It is used in one of two ways:

1. to create a hard link

```
$ ln file link
```

2. to create a symbolic link where "item" is either a file or a directory.

```
$ ln -s item link
```

Hard Links

When we create a hard link, we create an additional directory entry for a file. Hard links have two important limitations:

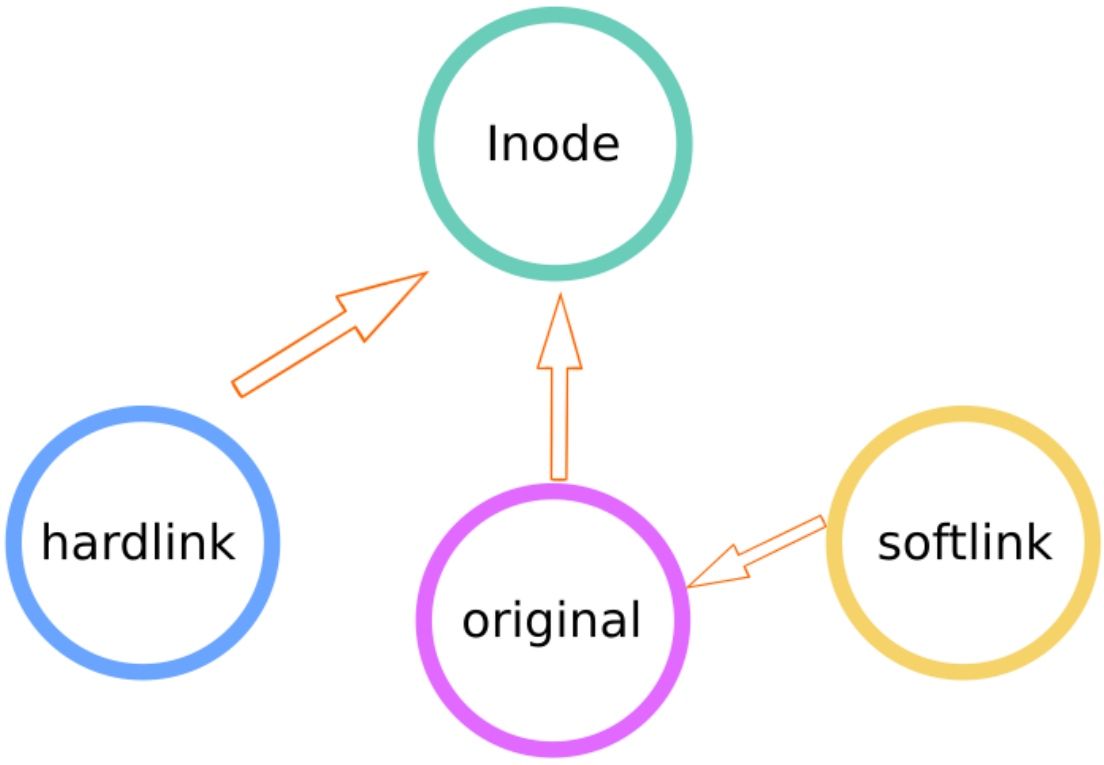
1. A hard link cannot reference a file outside its own file system. This means a link cannot reference a file that is not on the same disk partition as the link itself.
2. A hard link may not reference a directory.

A hard link is indistinguishable from the file itself. Unlike a symbolic link, when you list a directory containing a hard link you will see no special indication of the link. When a hard link is deleted, the link is removed but the contents of the file itself continue to exist until all links to the file are deleted.

Symbolic Links

Symbolic links were created to overcome the limitations of hard links. Symbolic links work by creating a special type of file that contains a text pointer to the referenced file or directory. In this regard, they operate in much the same way as a **Windows shortcut.**

A file pointed to by a symbolic link, and the symbolic link itself is largely indistinguishable from one another. For example, if you write something to the symbolic link, the referenced file is written too. However when you delete a symbolic link, only the link is deleted, not the file itself. If the file is deleted before the symbolic link, the link will continue to exist but will point to `nothing`. In this case, the link is said to be broken. In many implementations, the ls command will display broken links in a distinguishing color, such as red, to reveal their presence



Archiving & Compressing

One of the primary tasks of a `sysadmin` is keeping the system's data secure but it's also a very convenient way of transferring and moving a lot of files over the network. It is often useful to make copies of things and to move large collections of files from place to place and from device to device.

Archiving Files

Archiving is the process of gathering up many files and bundling them together into a single large file. Archiving is often done as a part of system backups. It is also used when old data is moved from a system to some type of long-term storage.

The tar command

It is the classic tool for archiving files. Its name, short for **tape archive**, reveals its roots as a tool for making backup tapes. While it is still used for that traditional task, it is equally adept at other storage devices as well.

The tar command can list the contents of archives or extract their files to the current system.

We often see filenames that end with the extension .tar or `.tgz`, which indicate a "plain" tar archive and a `gzipped` archive, respectively. A tar archive can consist of a group of separate files, one or more directory hierarchies, or a mixture of both. The command syntax works like this:

```
1 [jmedinar@localhost ~]$ tar mode[options] pathname...
```

Where mode is one of the following operating modes:

- c Create an archive from a list of files and/or directories.
- x Extract an archive.
- r Append specified pathnames to the end of an archive.
- t List the contents of an archive.

tar uses a slightly odd way of expressing options, so we'll need some examples to show how it works.

```
1 [jmedinar@localhost ~]$ tar cf myarchive.tar mydirectory
```

To list the contents of the archive, we can do this:

```
1 [jmedinar@localhost ~]$ tar tf myarchive.tar
```

For a more detailed listing, we can add the v (verbose) option:

```
1 [jmedinar@localhost ~]$ tar tvf myarchive.tar
```

Now, let's extract in a new location

```
[jmedinar@localhost ~]$ tar xf myarchive.tar newlocation
```

The are some other options that can be useful when running the above operations

- v Verbose. Shows which files get archived or extracted.
- f File. File name. This option must be followed by the file name of the archive to use or create.
- p Preserve. Preserve the permissions of files and directories when extracting an archive.
- z gZip. Use gzip compression (.tar.gz).
- j bZip2. Use bzip2 compression (.tar.bz2). bzip2 typically achieves a better compression ratio than gzip.
- J xZip. Use xz compression (.tar.xz). The xz compression typically achieves a better compression ratio than bzip2.

> *NOTE*
>
> *When archiving files by absolute path names, the leading / of the path is removed from the file name by default. Removing the leading / of the path help users to avoid overwriting important files when extracting the archive. The tar command extracts files relative to the current working directory.*

Compressing Files

Data compression is the process of removing redundancy from data. Imagine you have a black picture file with the dimensions of 100 pixels by 100 pixels that will occupy 30,000 bytes of storage. If we encode the data in such a way that we simply represent a number as "100x100=black pixels" instead of storing a block of data containing 30,000 zeros, we could compress the data much smaller.

Such a data compression scheme is called run-length encoding and is one of the most rudimentary compression techniques. Today's techniques are much more advanced and complex but the basic goal remains the same—**get rid of redundant data.**

The gzip command

The `gzip` program is used to compress one or more files. When executed, it replaces the original file with a compressed version of the original. The corresponding `gunzip` program is used to restore compressed files to their original, uncompressed form. Here is an example:

```
[jmedinar@localhost ~]$ ls -l /etc/ > myetc.list
[jmedinar@localhost ~]$ du -sh myetc.list
20K myetc.list
[jmedinar@localhost ~]$ gzip myetc.list
[jmedinar@localhost ~]$ du -sh myetc.list.gz
4.0K myetc.list.gz
```

We created a file with the listed content of /etc that was 20k in size and after compression it was 4k. Notice that the name of the file changed adding the 'gz' extension to indicate it was `gzipped`!

The bzip2 command

The `bzip2` program is similar to `gzip` but uses a different compression algorithm that achieves higher levels of compression at the cost of compression speed. In most regards, it works in the same fashion as `gzip`. A file compressed with `bzip2` is denoted with the extension `.bz2`. Comes with `bunzip2` and `bzcat` for decompressing files

Don't try to compress the already compressed file.

If you attempt to compress a file, that has already been compressed with an effective compression algorithm, by doing something like this you're probably just wasting time and space! you will actually end up a larger file.

```
[jmedinar@localhost ~]$ gzip picture.jpg
```

Compressing with the tar command

The tar command supports the two compression methods reviewed above and one more the XZ.

- The gzip compression is the fastest and oldest one and is most widely available across distributions.
- bzip2 compression creates smaller archive files compared to gzip but is less widely available than gzip
- xz compression method is relatively new, but usually offers the best compression ratio of the methods available.

Use one of the following options to create a compressed tar archive:

- -z or --gzip for gzip compression (filename.tar.gz or filename.tgz)
- -j or --bzip2 for bzip2 compression (filename.tar.bz2)
- -J or -xz for xz compression (filename.tar.xz)

To create a gzip compressed archive named `/root/etcbackup.tar.gz`, with the contents from the `/etc` directory on host:

```
[root@localhost ~]# tar -czf /root/etcbackup.tar.gz /etc
tar: Removing leading `/' from member names
```

To create a bzip2 compressed archive named `/root/logbackup.tar.bz2`, with the contents from the `/var/log` directory on host:

```
[root@localhost ~]$ tar -cjf /root/logbackup.tar.bz2 /var/log
tar: Removing leading `/' from member names
```

To create a xz compressed archive named, `/root/sshconfig.tar.xz`, with the contents from the `/etc/ssh` directory on host:

```
[root@localhost ~]$ tar -cJf /root/sshconfig.tar.xz /etc/ssh
tar: Removing leading `/' from member names
```

Decompressing Files

The gunzip command

The `gunzip` program, which uncompresses `gzip` files, assumes that filenames end in the extension `.gz`, so it's not necessary to specify it, as long as the specified name is not in conflict with an existing uncompressed file:

```
[jmedinar@localhost ~]$ gunzip myetc.list
[jmedinar@localhost ~]$ du -sh myetc.list
20K myetc.list
```

Notice the '`gz`' extension was removed and the file is back to it's original size.

Decompressing with the tar command

The first step when extracting a compressed tar archive is to determine where the archived files should be extracted to, then create and change to the target directory. The tar command determines which compression was used and it is usually not necessary to use the same compression option used when creating the archive. It is valid to add the decompression method to the tar command. If one chooses to do so, the correct decompression type option must be used; otherwise tar yields an error about the decompression type specified in the options not matching the file's decompression type.

To extract the contents of a gzip compressed archive named `/root/etcbackup.tar.gz` in the `/tmp/etcbackup` directory:

```
[root@localhost ~]# mkdir /tmp/etcbackup
[root@localhost ~]# cd /tmp/etcbackup
[root@localhost etcbackup]# tar -tf /root/etcbackup.tar.gz
etc/
etc/fstab
etc/crypttab
etc/mtab
...output omitted...
[root@localhost etcbackup]# tar -xzf /root/etcbackup.tar.gz
```

To extract the contents of a bzip2 compressed archive named `/root/logbackup.tar.bz2` in the `/tmp/logbackup` directory:

```
[root@localhost ~]# mkdir /tmp/logbackup
[root@localhost ~]# cd /tmp/logbackup
[root@localhost logbackup]# tar -tf /root/logbackup.tar.bz2
var/log/
var/log/lastlog
var/log/wtmp
var/log/btmp
...output omitted...
[root@localhost logbackup]# tar -xjf /root/logbackup.tar.bz2
```

To extract the contents of a xz compressed archive named `/root/sshbackup.tar.xz` in the `/tmp/sshbackup` directory:

```
[root@localhost ~]$ mkdir /tmp/sshbackup
[root@localhost ~]# cd /tmp/sshbackup
[root@localhost logbackup]# tar -tf /root/sshbackup.tar.xz
etc/ssh/
etc/ssh/moduli
etc/ssh/ssh_config.d/05-redhat.conf
etc/ssh/sshd_config
...output omitted...
[root@localhost sshbackup]# tar -xJf /root/sshbackup.tar.xz
```

Listing a compressed tar archive works in the same way as listing an uncompressed tar archive.

Synchronizing Files And Directories

The rsync command

The rsync command is a way to securely copy files from one system to another. The tool uses an algorithm that minimizes the amount of data copied by synchronizing only the portions of files that have changed. It differs from scp in that if two files or directories are similar between two servers, rsync copies the differences between the file systems on the two servers, while scp would need to copy everything.

One of the advantages of rsync is that it can copy files between a local system and a remote system securely and efficiently. While the initial synchronization of a directory takes about the same time as copying it, any subsequent synchronization only requires the differences to be copied over the network, speeding updates, possibly substantially.

The most important option to remember from this command is:

- The `-n` option to perform a dry run. A dry run is a simulation of what happens when the command gets executed. The dry run shows the changes rsync would perform when the command is run without making any change. You should perform a dry run before performing an rsync operation to ensure no important files get overwritten or deleted.

The two most common options when synchronizing files and directories with rsync are:

- The `-v` or `--verbose` option provides more detailed output as the synchronization runs. This is useful for troubleshooting and to help see progress.
- The `-a` or `--archive` option enables "archive mode". This is a quick way to enable recursive copying and turn on a large number of useful options to preserve most characteristics of the files. Archive mode is the same as specifying the following options:

OPTION	DESCRIPTION
-r, --recursive	synchronize recursively the whole directory tree
-l, --links	synchronize symbolic links
-p, --perms	preserve permissions
-t, --times	preserve time stamps
-g, --group	preserve group ownership
-o, --owner	preserve the owner of the files
-D, --devices	synchronize device file

Let's try `rsync` out on some local files by synchronizing the /tmp directory with a corresponding copy in /home/tmp:

```
[root@localhost ~]# rsync -av /tmp /home/tmp
sending incremental file list
created directory /home/tmp
tmp/
tmp/.X0-lock
tmp/.X1-lock
tmp/.X1024-lock
tmp/.X1025-lock
...output omitted...
```

While the command runs, we will see a list of the files and directories being copied. If we run the command again, we will see a different result

```
[root@localhost ~]# rsync -av /tmp /home/tmp
sending incremental file list
tmp/

sent 2,990 bytes  received 61 bytes  6,102.00 bytes/sec
total size is 1,351,955  speedup is 443.12
```

Notice that there was no listing of files. This is because rsync detected that there were no differences between the original and the copy, and therefore it didn't need to copy anything.

Let's make a minor change in the original location and then run the command again

```
[root@localhost ~]# touch /tmp/file{1..10}.tmp
[root@localhost ~]# rsync -av /tmp /home/tmp
sending incremental file list
tmp/
tmp/file1.tmp
tmp/file10.tmp
tmp/file2.tmp
tmp/file3.tmp
tmp/file4.tmp
tmp/file5.tmp
tmp/file6.tmp
tmp/file7.tmp
tmp/file8.tmp
tmp/file9.tmp

sent 3,559 bytes  received 251 bytes  7,620.00 bytes/sec
total size is 1,351,955  speedup is 354.84
```

We can see how only the recently created files where copied.

Using rsync Over A Network

One of the real beauties of rsync is that it can be used to copy files over a network. After all, the "r" in rsync stands for "remote." Remote copying can be done with another system that has rsync installed, along with a remote shell program such as ssh, we could do this:

```
[root@localhost ~]# sudo rsync -av --delete --rsh=ssh /etc /home
/usr/local remotehost:/backup
```

We made two changes to our command to facilitate the network copy. First, we added the `--rsh=ssh` option, which instructs rsync to use the ssh program as its remote shell. Second, we specified the remote host by prefixing its name `remotehost` to the destination pathname.

Linux/Mac Terminal Tutorial: How To Use The rsync Command

```
### The SCP command
```

OpenSSH is useful for securely running shell commands on remote systems. The Secure Copy command, scp, which is part of the OpenSSH suite, copies files from a remote system to the local system or from the local system to a remote system. The command uses the SSH server for authentication and encrypts data when it is being transferred.

You can specify a remote location for the source or destination of the files you are copying. The format of the remote location should be in the form `user@host:/path`. The user@ portion of the argument is optional. If it is missing, your current local username will be used. When you run the command, your scp client will authenticate to the remote SSH server just like ssh, using key-based authentication or prompting you for your password.

The following example demonstrates how to copy the local `/etc/yum.conf` and `/etc/hosts` files on host, to the remoteuser's home directory on the remotehost remote system:

```
[jmedinar@localhost ~]$ scp /etc/yum.conf /etc/hosts
remoteuser@remotehost:/home/remoteuser
remoteuser@remotehost's password: password
yum.conf          100%  813    0.8KB/s   00:00
hosts             100%  227    0.2KB/s   00:00
```

You can also copy a file in the other direction, from a remote system to the local file system. In this example, the file /etc/hostname on remotehost is copyed to the local directory /home/user.

The scp command authenticates to remotehost as the user remoteuser.

```
[jmedinar@localhost ~]$ scp remoteuser@remotehost:/etc/hostname
/home/jmedinar
remoteuser@remotehost's password: password
hostname              100%  22 0.0KB/s   00:00
```

To copy a whole directory tree recursively, use the -r option.

```
[jmedinar@localhost ~]$ scp -r root@remoteuser:/var/log /tmp
root@remotehost's password: password
...output omitted...
```

The sFTP command

To interactively upload or download files from a SSH server, use the Secure File Transfer Program, sftp. A session with the sftp command uses the secure authentication mechanism and encrypted data transfer to and from the SSH server.

Just like the scp command, the sftp command uses `user@host` to identify the target system and user name. If you do not specify a user, the command will attempt to log in using your local user name as the remote user name. You will then be presented with an sftp> prompt.

```
[jmedinar@localhost ~]$ sftp remoteuser@remotehost
remoteuser@remotehost's password: password
Connected to remotehost.
sftp>
```

The interactive sftp session accepts various commands that work the same way on the remote file system as they do in the local file system, such as `ls, cd, mkdir, rmdir, and pwd`. The put command uploads a file to the remote system. The get command downloads a file from the remote system. The exit command exits the sftp session.

To upload the `/etc/hosts` file on the local system to the newly created directory `/home/remoteuser/hostbackup` on remotehost:

```
sftp> mkdir hostbackup
sftp> cd hostbackup
sftp> put /etc/hosts
Uploading /etc/hosts to /home/remoteuser/hostbackup/hosts
/etc/hosts                    100% 227 0.2KB/s  00:00
sftp>
```

To download `/etc/yum.conf` from the remote host to the current directory on the local system, execute the command get `/etc/yum.conf`and exit the sftp session with the exit command.

```
sftp> get /etc/yum.conf
Fetching /etc/yum.conf to yum.conf
/etc/yum.conf                  100%  813   0.8KB/s  00:00
sftp> exit
[user@host ~]$
```