On Linux and other Unix-like operating systems, there is a set of rules for each file and directory that define who can access it, and what they can do with it. These rules are called permissions or modes.

Permissions specify what a particular person may or may not do with a file or directory. As such, permissions are important in creating a secure environment. For instance, you don't want other people to be changing your files, and you also want system files to be safe from damage (either accidental or deliberate). Luckily, permissions in a Linux system are quite easy to work with.

Linux permissions dictate 3 things you may do with a file, read, write and execute. They are referred to in Linux by a single letter each.

- r read - you may view the contents of the file.
- w write - you may change the contents of the file.
- x execute - you may execute or run the file if it is a program or script.

With directories:

- r - you have the ability to read the contents of the directory (ie do an ls)
- w - you have the ability to write into the directory (ie create files and directories)
- x - you have the ability to enter that directory (ie cd)

For every file (or directory) we define 3 sets of people for whom we may specify permissions.

- owner - a single person who owns the file. (typically the person who created the file but ownership may be granted to someone else by certain users)
- group - every file belongs to a single group of people.
- others - everyone else who is not in the group or the owner.

---

ls -l Part of the output for every entity when we do ls -l looks something like this:

-rwxr-xr--

The first character identifies the file type. If it is a dash (-) then it is a normal file. If it is a ‘d’ then it is a directory.

The next 3 characters represent the permissions for the owner. A letter represents the presence of a permission and a dash (-) represents the absence of a permission. In this example the owner has all permissions (read, write, and execute).

The following 3 characters represent the permissions for the group. In this example the group has the ability to read and execute but not write. Note that the order of permissions is always read, then write then execute.

Finally, the last 3 characters represent the permissions for others (or everyone else). In this example they have the read permission and nothing else.

You can use the -d option to ls to see this for directories: ls -ld

chmod is used to change the permissions of files or directories. It stands for stands for "change mode". General usage:

```
chmod options permissions filename
```

Important options:

-R change files and directories recursively

--reference=RFILE Set permissions to match those of file RFILE, ignoring any other specified permissions

If no options are specified, chmod modifies the permissions of the file specified to the permissions specified.

You can represent permissions with their alphanumeric characters, or with octal numbers (the digits 0 through 7).

```
chmod u=rwx,g=rx,o=r myfile
```

```
chmod 754 myfile
```

The octal numbers correspond to r,w,x as follows. Use one octal digit for each of the owner (user), group, and other.

Octal	Binary
0	0 0 0
1	0 0 1
2	0 1 0
3	0 1 1
4	1 0 0
5	1 0 1
6	1 1 0
7	1 1 1

chown changes the owner and group.

-R and --reference work the same.

Change the owner of file.txt to Blake:
```
chown Blake file.txt
```

Recursively grant ownership of the directory /files/work and all files and subdirectories to user Blake:
```
chown –R Blake /files/work
```

Change the group of the file to the group “friends”
```
chown :friends file.txt
```

---

id prints user and group information for the specified USERNAME, or, when USERNAME omitted, for the current user.

-g prints only primary group id number
-G prints all group id numbers
-n uses group names instead of id numbers
-u prints the user id number
-un prints the username (not helpful?)

---

users prints the names of all users currently logged in to the host.

You can count the current users with `users | wc –w`

---

who is very similar. It prints information about all users who are currently logged in.

Some salient options:
-a print all possible information.
-b Display the time of the last system boot.
-d Display dead processes.
-l Print system login processes.
-m Only print information about the user and host associated with standard input (the terminal where the command was issued).
-q Displays all login names, and a count of all logged-on users.

---

whoami prints the username associated with the current effective user ID.
same as running id -un

---

newgrp allows a user to log in to a new group. (change the current group ID)

If a dash (-) is included as an argument, then the user's environment is initialized as though he or she had just logged in; otherwise, the current working environment remains unchanged.

newgrp changes the current real group ID to the specified group, or, if no group is specified, to the default group listed in the file /etc/passwd.

newgrp also tries to add the group to the user groupset.

- If the user is root, he or she will not be prompted for a password.
- If the user is not root, he or she will be prompted for a group password if:
    - the user does not have a password, but the group does, or if
    - the user is not listed as a group member, and the group has a password.
- If there is no group password set, and the user is not listed as a member of the group, the user will be denied access.

```
newgrp thehomies
newgrp – thehomies
```

---

chgrp changes group ownership of a file or files.
Has -R and --reference

```
chgrp thehomies file.txt
chgrp –R staff /office/files
```

---

groupadd creates a new group.
For this command to work you must have superuser rights or be logged in as root.

```
groupadd newgroup
```

The above example would create a new group called "newgroup". This new group could then have users added to it using `useradd –g` (to set initial login group) or `useradd –G` (to add as a secondary group).

groupdel is used by a superuser or root to remove a group.

gpasswd administers groups. We won't cover this in detail.
-a user adds a user to the group
-d user removes a user from the group
-r removes the password so that only existing members can log in.

---

sudo (superuser do) allows a user with proper permissions to execute a command as another user, such as the superuser.

Run a bash script as root (with root permissions and whatnot):
```
sudo bash myscript.sh
```

Run the command ls as the user Blake.
```
sudo -u Blake ls
```

---

su (substitute user) changes the current user ID to that of the superuser or another user.

This makes it possible to change a login session's owner (i.e., the user who originally created that session by logging on to the system) without the owner having to first log out of that session.

Although su can be used to change the ownership of a session to any user, it is most commonly employed to change the ownership from an ordinary user to the root (i.e., administrative) user, thereby providing access to all parts of and all commands on the computer or system. For this reason, it is often referred to (although somewhat inaccurately) as the superuser command. It is also sometimes called the switch user command.

Superuser (root) is default
```
su
```

Change the owner of the current login session to a user named bob:
```
su bob
```

Switch the current user ID to that of bob, and set the environment to bob's login environment:
```
su – bob
```

---

sg executes commands as a different group ID.

The sg command works similar to newgrp but accepts a command. With most shells you may run sg from, you need to enclose multi-word commands in quotes. Another difference between newgrp and sg is that some shells treat newgrp specially, replacing themselves with a new instance of a shell that newgrp creates. This doesn't happen with sg, so upon exit from a sg command you are returned to your previous group ID.

When new files are created, they have certain permissions determined by the umask. The **umask** command shows or sets this mask.

Show the current umask value:
```
umask
```

Show a symbolic representation of the current umask value:
```
umask –S
```

When a new file is created, each digit of the umask is "subtracted" from the OS's default value to arrive at the default value that you define. It's not really subtraction; technically, the mask is negated (its bitwise compliment is taken) and this value is then applied to the default permissions using a logical AND operation. The result is that the umask tells the operating system which permission bits to "turn off" when it creates a file.

In Linux, the default permissions value is 666 for a regular file, and 777 for a directory. When creating a new file or directory, the kernel takes this default value, "subtracts" the umask value, and gives the new files the resulting permissions.

If we do:
```
umask 022
```
Then new files have the permissions 644

Set the umask to give read permissions to the group:
```
umask g+r
```

Set the mask so that new files will be readable and writable by the owner, but may not be executed; group members and others will have no permissions to the file:
```
umask u=rw,go=
```