Federal University Dutse

CCS 206 - Introduction to Linux for Security and Forensics

Lecture Note 6 – Permissions
By
Muhammad S. Ali

Permissions

- Linux unlike MS-DOS, is not only multitasking systems but also multi-user systems. This means more than one person can use the computer at the same time. While a typical computer will likely have only one keyboard and monitor, it can still be used by more than one user.
- For instance, if a computer is attached to a network or the internet, remote users log in via ssh (secure shell) and operate the computer. In fact, remote users can execute graphical applications and have the graphical output appear on a remote display . The X Window System supports this as part of its basic design.
- This feature is already deeply embedded into Linux design since its inception.
- In order to make this practical, a method had to be devised 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.

Permission Commands

- In this lesson, we are going to look at Linux system security and introduce the following commands:
- id – Display user identity
- chmod – Change a file’s mode
- umask – Set the default file permissions
- su – Run a shell as another user
- sudo – Execute a command as another user
- chown – Change a file’s owner
- chgrp – Change a file’s group ownership
- passwd – Change a user’s password.

Owners, Group Members, and Everybody Else

- In the Unix security model, a user may own a file 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 the *world*.
- To find out more about a user identity, use the *id* command
- When user accounts are created, users are assigned a number called a *user ID*, or *uid*. This 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 additional groups.
- User accounts are defined in the */etc/passwd* file, and groups are 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.
- For each user account, the */etc/passwd* file defines the user (login) name, the uid, the gid, the account's real name, the home directory, and the login shell.

Reading, Writing, and Executing

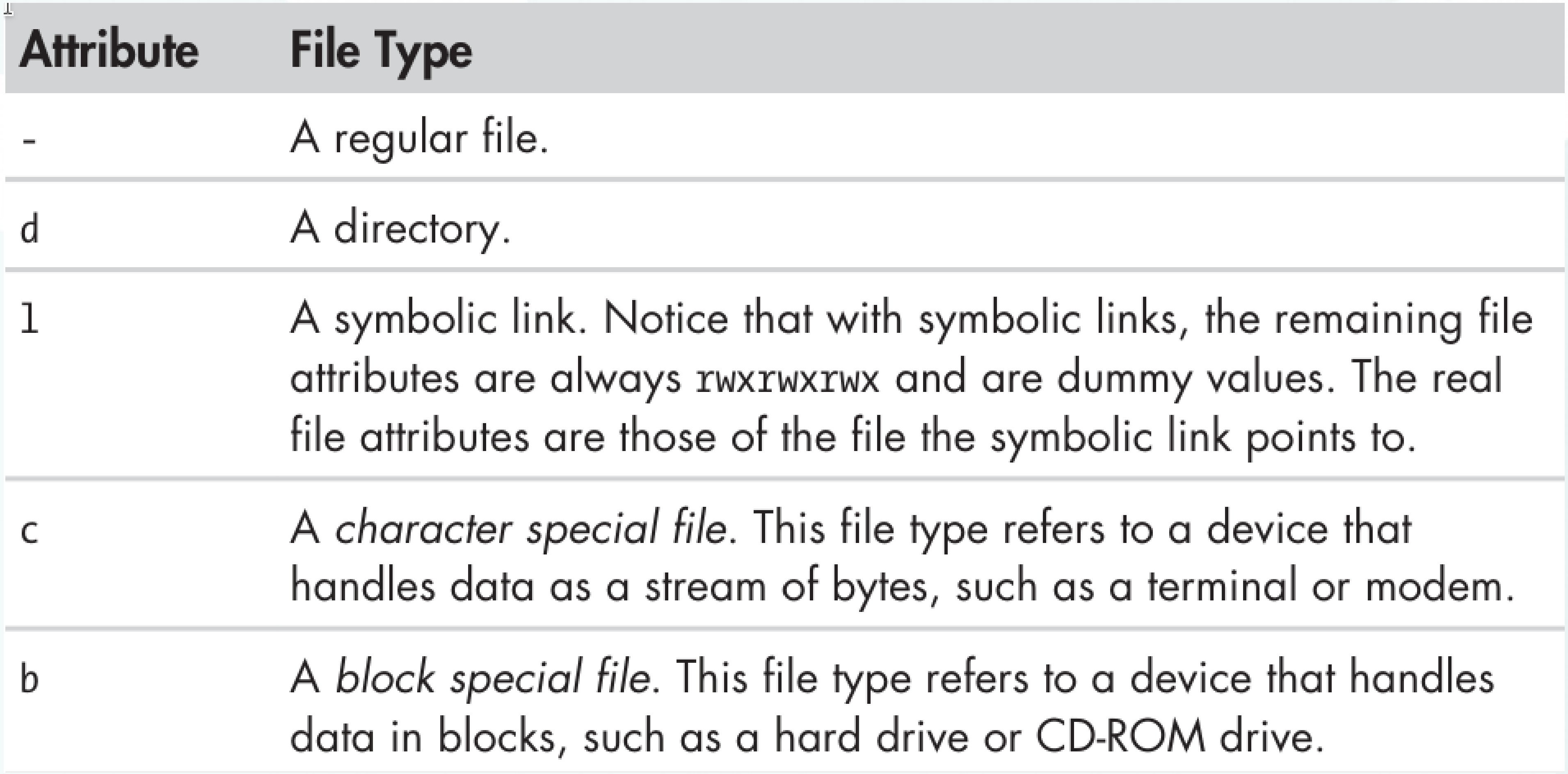
- Access rights to files and directories are defined in terms of read access, write access, and execution access. If look at the output of ls command, we can get some clue as to how this is implemented:

```
salsafh@localhost ~> > foo.txt

  salsafh@localhost ~> ls -l foo.txt
```

- The first 10 characters of the output are the file attributes. The first of these characters is the file type. The remaining nine characters of the file attributes, called the file mode, represent the read (r), write (w) and execute (x) permissions for the file’s owner, the file’s group owner, and everybody else.
- When set, the r, w, and x mode attributes have certain effects of files and directories, as shown in the table below:

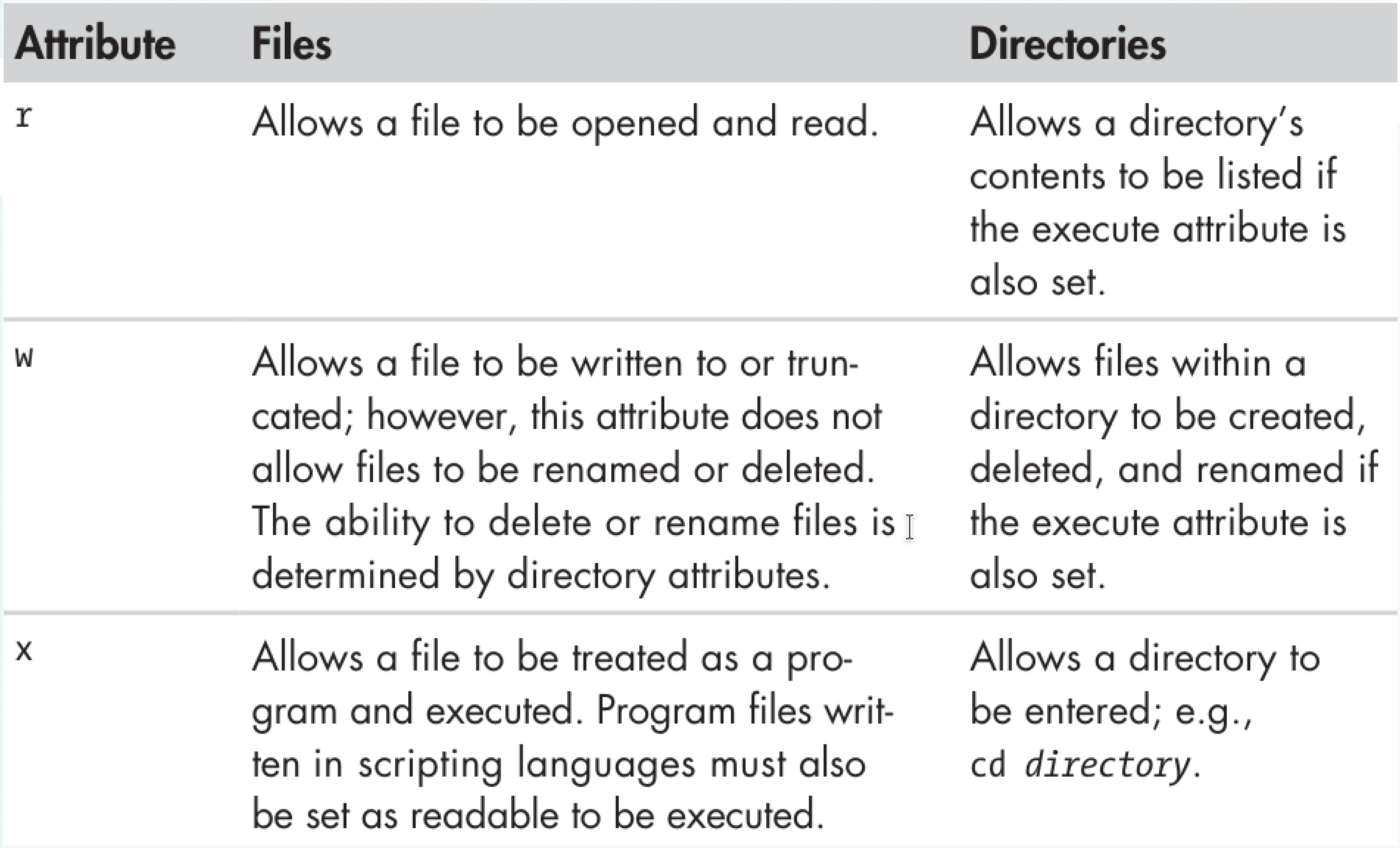
File types



Attribute	File Type
-	A regular file.
d	A directory.
l	A symbolic link. Notice that with symbolic links, the remaining file attributes are always rwxrwxrwx and are dummy values. The real file attributes are those of the file the symbolic link points to.
c	A *character special file*. This file type refers to a device that handles data as a stream of bytes, such as a terminal or modem.
b	A *block special file*. This file type refers to a device that handles data in blocks, such as a hard drive or CD-ROM drive.

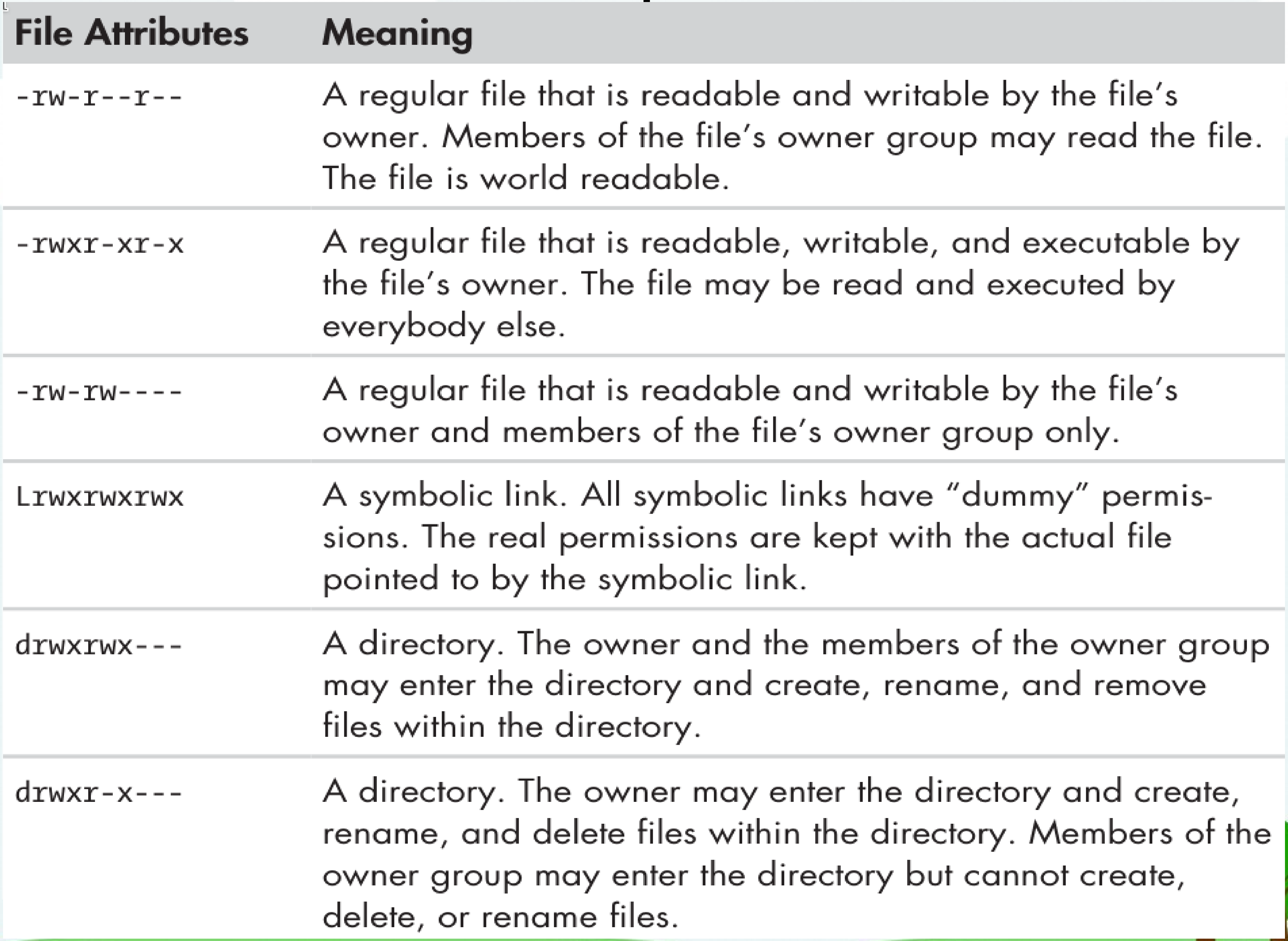
- Permission attributes

Permission Attributes



Attribute	Files	Directories
`r`	Allows a file to be opened and read.	Allows a directory's contents to be listed if the execute attribute is also set.
`w`	Allows a file to be written to or truncated; however, this attribute does not allow files to be renamed or deleted. The ability to delete or rename files is determined by directory attributes.	Allows files within a directory to be created, deleted, and renamed if the execute attribute is also set.
`x`	Allows a file to be treated as a program and executed. Program files written in scripting languages must also be set as readable to be executed.	Allows a directory to be entered; e.g., `cd` *`directory`*.

Permission Attributes Examples



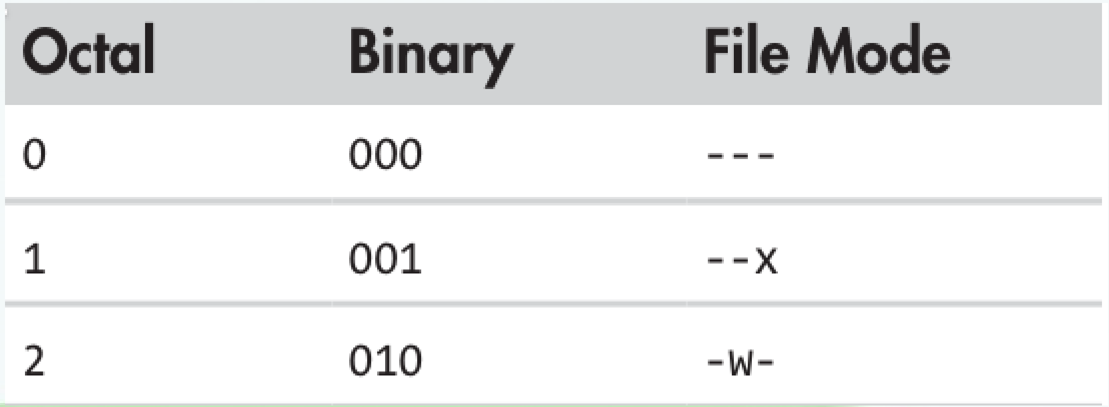
File Attributes	Meaning
`-rw-r--r--`	A regular file that is readable and writable by the file's owner. Members of the file's owner group may read the file. The file is world readable.
`-rwxr-xr-x`	A regular file that is readable, writable, and executable by the file's owner. The file may be read and executed by everybody else.
`-rw-rw----`	A regular file that is readable and writable by the file's owner and members of the file's owner group only.
`Lrwxrwxrwx`	A symbolic link. All symbolic links have "dummy" permissions. The real permissions are kept with the actual file pointed to by the symbolic link.
`drwxrwx---`	A directory. The owner and the members of the owner group may enter the directory and create, rename, and remove files within the directory.
`drwxr-x---`	A directory. The owner may enter the directory and create, rename, and delete files within the directory. Members of the owner group may enter the directory but cannot create, delete, or rename files.

chmod – Change File Mode

- To change the mode (permissions) of a file or directory, the chmod command is used. Note that only the file’s owner or the superuser can change the mode of a file or directory. chmod supports two distinct ways of specifying mode changes: octal number representation and symbolic representation.

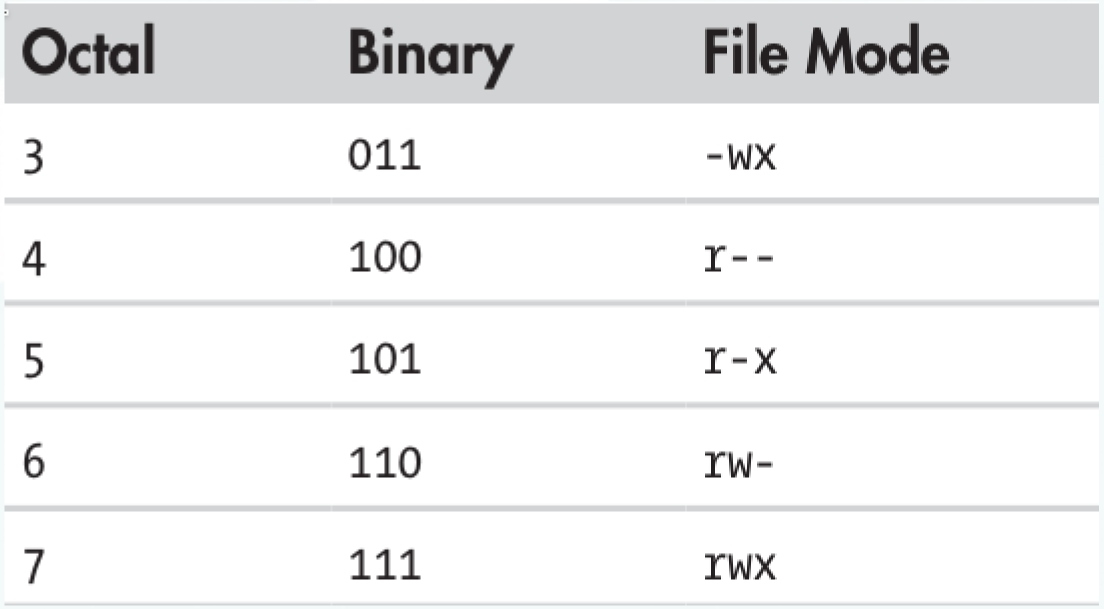
Octal Representation

- With octal notation, we use octal numbers to set the pattern of desired permissions. Since each digit in an octal number represents three binary digits, this maps nicely to the scheme used to store the file mode. See the table below:



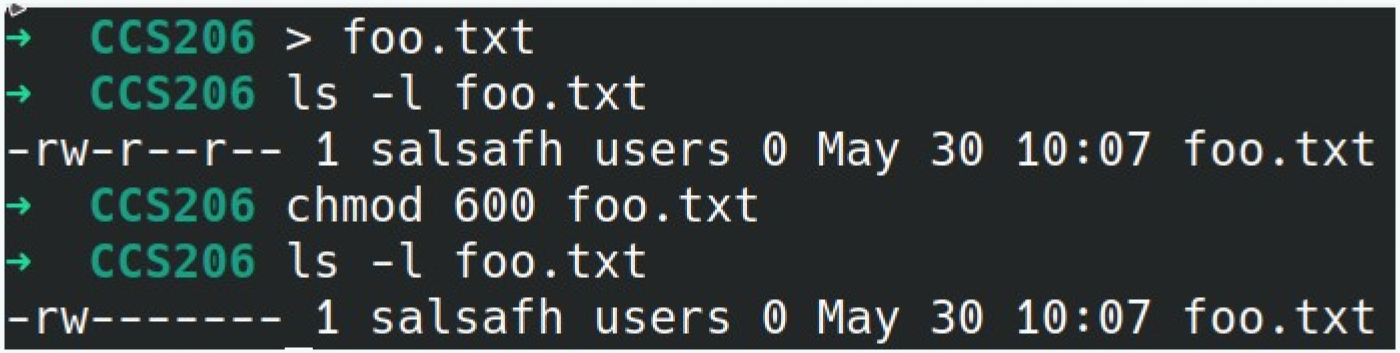
Octal	Binary	File Mode
0	000	`---`
1	001	`--x`
2	010	`-w-`

chmod – Change File Mode



Octal	Binary	File Mode
3	011	-wx
4	100	r--
5	101	r-x
6	110	rw-
7	111	rwx

- By using the three octal digits, we can set the file mode for owner, group owner, and world.



```
➜  CCS206 > foo.txt
➜  CCS206 ls -l foo.txt
-rw-r--r-- 1 salsafh users 0 May 30 10:07 foo.txt
➜  CCS206 chmod 600 foo.txt
➜  CCS206 ls -l foo.txt
-rw------- 1 salsafh users 0 May 30 10:07 foo.txt
```

chmod – Change File Mode

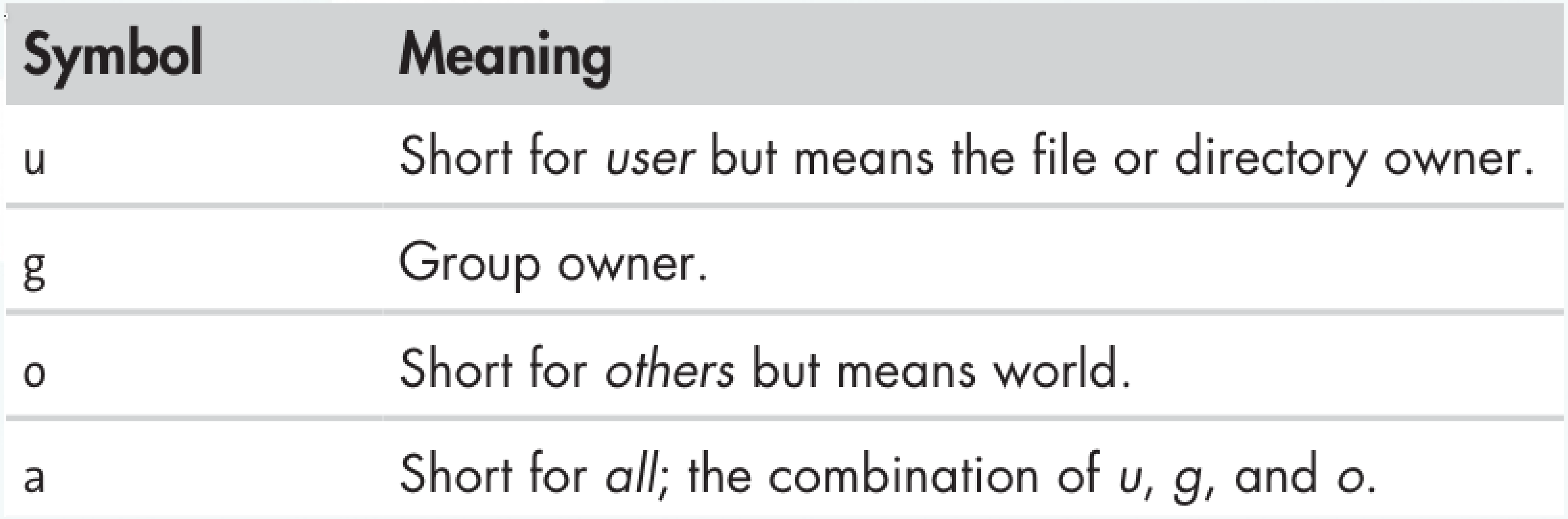
- By passing the argument 600, we were able to set the permissions of the owner to read write while removing all permissions from the group owner and world. Though remembering the octal-to-binary mapping may seem inconvenient, you will usually have to use only a few common ones:

7(rwx), 6(rw-), 5(r-x), 4(r--), and (---).

Symbolic Representation

- chmod also supports a symbolic notation for specifying file modes. Symbolic notation is divided into three parts: whom the change will affect, which operation will be performed, and which permission will be set. To specify who is affected, a combination of the characters *u*, *g*, *o*, and a is used, as shown in the table below:

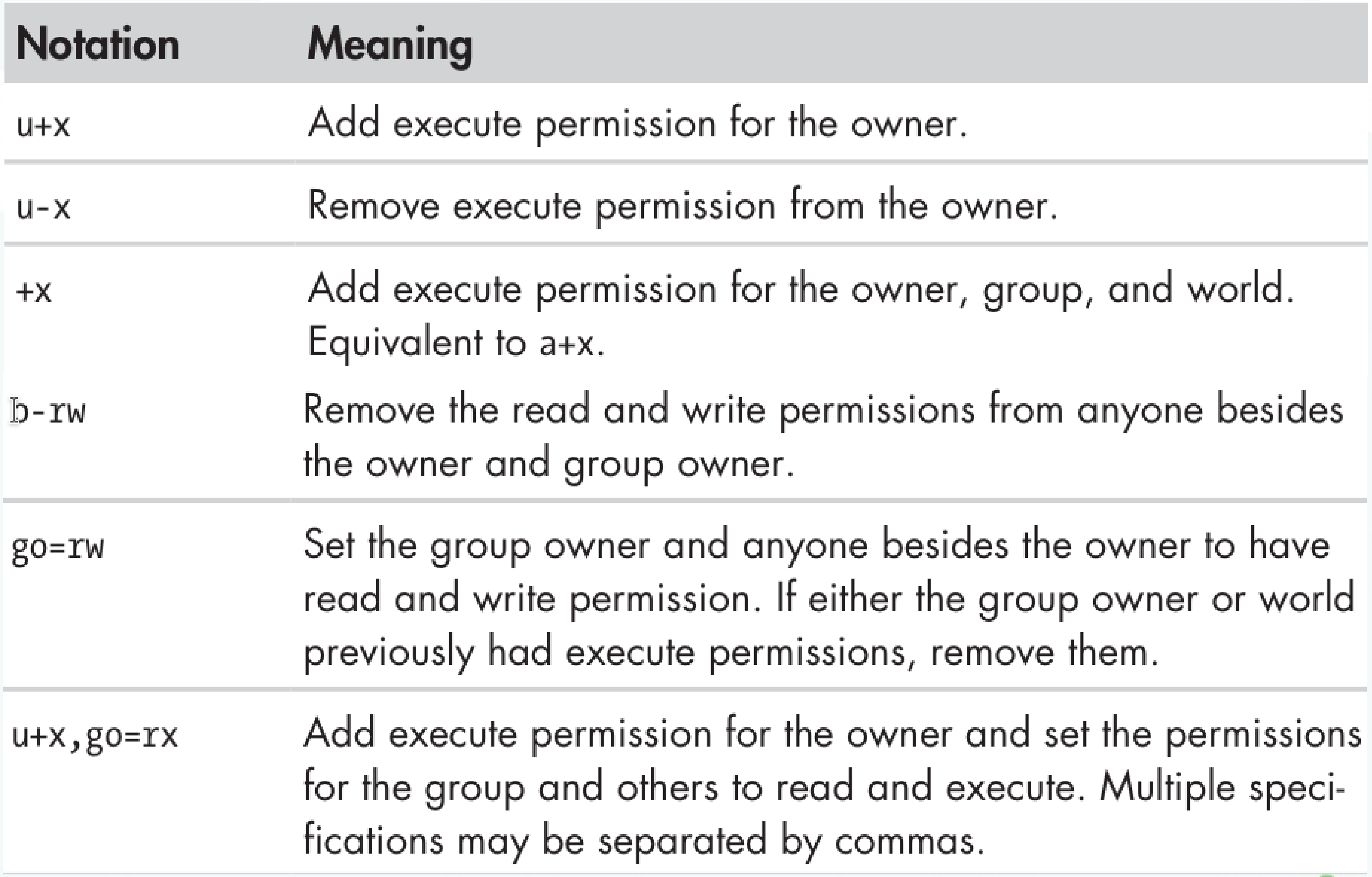
chmod symbolic notation



Symbol	Meaning
u	Short for *user* but means the file or directory owner.
g	Group owner.
o	Short for *others* but means world.
a	Short for *all*; the combination of *u*, *g*, and *o*.

- If no character is specified, *all* will be assumed. The operation may be a + indicating that a permission is to be added, a – indicating that a permission is to be taken away, or a = indicating that only the specified permissions are to be applied and that all others are to be removed. Permissions are specified with the r, w, and x characters. See examples in the table below:
-

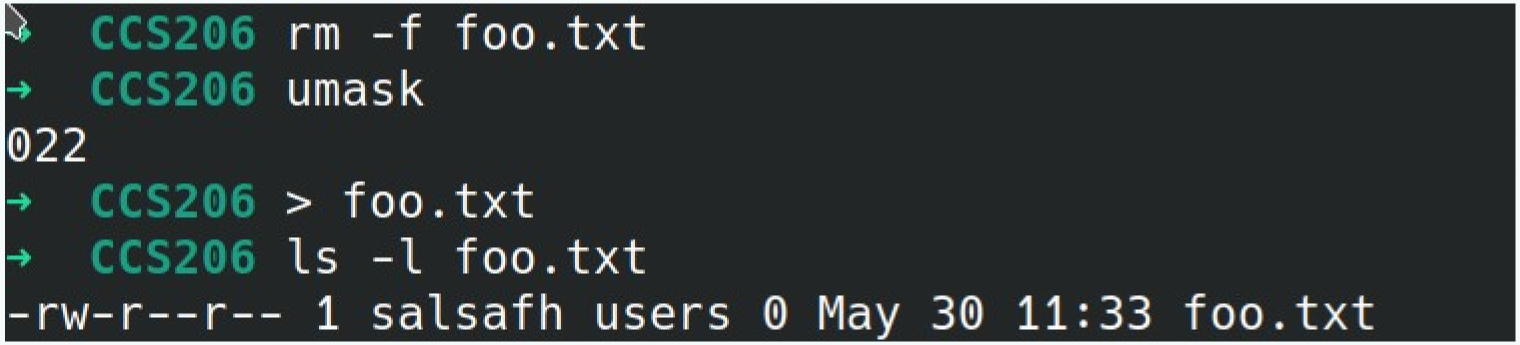
chmod symbolic notation examples



Notation	Meaning
u+x	Add execute permission for the owner.
u-x	Remove execute permission from the owner.
+x	Add execute permission for the owner, group, and world. Equivalent to a+x.
o-rw	Remove the read and write permissions from anyone besides the owner and group owner.
go=rw	Set the group owner and anyone besides the owner to have read and write permission. If either the group owner or world previously had execute permissions, remove them.
u+x,go=rx	Add execute permission for the owner and set the permissions for the group and others to read and execute. Multiple specifications may be separated by commas.

umask – Set Default Permissions

- The *umask* command controls the default permissions given to a file when it is created. It uses the octal notation to express a *mask* of bits to be removed from a file's mode attributes.

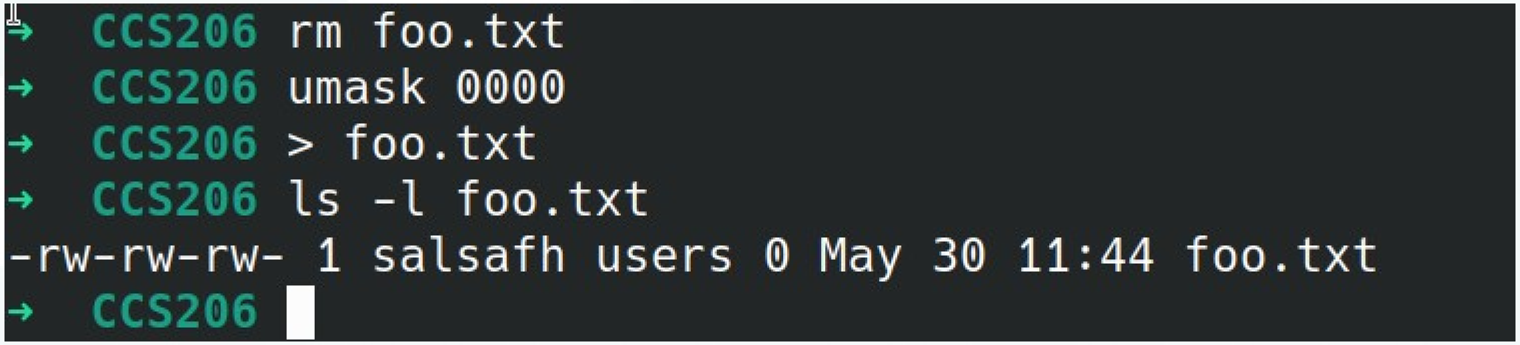


```
→  CCS206 rm -f foo.txt
→  CCS206 umask
022
→  CCS206 > foo.txt
→  CCS206 ls -l foo.txt
-rw-r--r-- 1 salsafh users 0 May 30 11:33 foo.txt
```

- We first removed any existing copy of foo.txt to make sue we were starting fresh. Next, we ran the umask command without argument to see the current value. It responded with the value 022 (the value 002 is another common default value), which is the octal representation of our mask. We then created a new instance of the file foo.txt and observed its permissions.

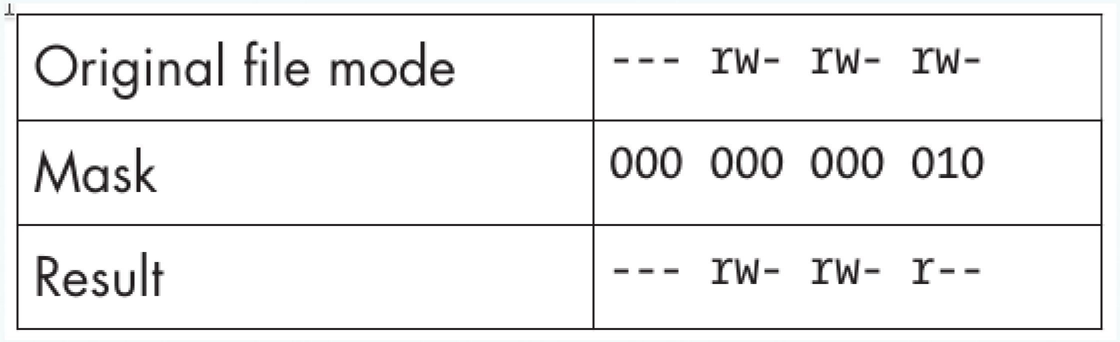
- We can see that only the owner get read and write permissions, while everyone else gets only read permission. Everyone else has only read permission because of the value of the mask. Let us repeat our example, this time setting the mask manually:

umask – Set Default Permissions



```
→  CCS206 rm foo.txt
→  CCS206 umask 0000
→  CCS206 > foo.txt
→  CCS206 ls -l foo.txt
-rw-rw-rw- 1 salsafh users 0 May 30 11:44 foo.txt
→  CCS206 
```

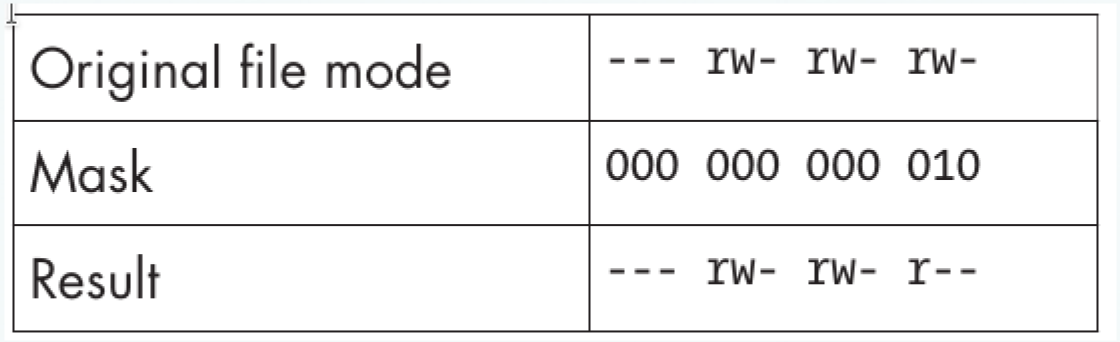
- When we set the mask to 0000 (effectively turning it off), we see that the file is now world writable. To understand how this works, we have to look at octal numbers again. If we expand the mask into binary and then compare it to the attributes, we can see what happens:



Original file mode	`--- rw- rw- rw-`
Mask	000 000 000 010
Result	`--- rw- rw- r--`

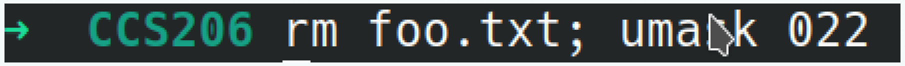
umask – Set Default Permissions

- Ignore the for the moment the leading 0s and observe that where the 1 appears in our mask, an attribute was removed – in this case, the world write permission. That is what the mask does. Everywhere a 1 appears in the binary value of the mask, an attribute is unset. If we look at a mask value of 0022, we can see what it does:



Original file mode	`--- rw- rw- rw-`
Mask	000 000 000 010
Result	`--- rw- rw- r--`

- Again, where a 1 appears in the binary value, the corresponding attribute is unset. Play with some values (try some 7s) to get used to how this works. When you are done, remember to clean up:



```
➜  CCS206 rm foo.txt; umask 022
```

Changing Identities

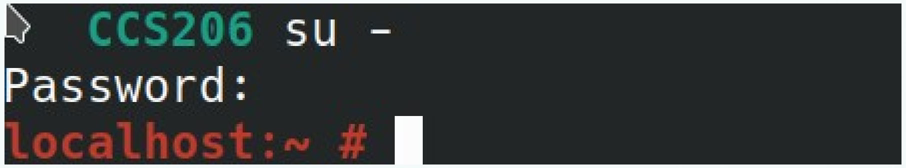
- At various times, we may find it necessary to take on the identity of another user. Often we want to gain superuser privileges to carry out some administrative task, but it is also possible to “become” another regular user to perform such such tasks as testing account. There are three ways to take on an alternate identity:
- > Log out and log back in as the alternate user.
- > Use the su command.
- > Use the sudo command
- From within your own shell session, the *su* command allows you to assume the identity of another user and either start a new shell session with the user’s ID or issue a single command as the user.
- The *sudo* command allows and administrator to set up a configuration file called /*etc/sudoers* and define specific commands that particular users are permitted to use largely determined by which Linux distribution you use.

su – Run a Shell with Substitute User and Group IDs

- The su command is used to start a shell as another user. The command syntax is as follows:

su [-[l]] [user]

- If the -l option is included, the resulting shell session is a *login shell* for the specified user. This means that the user's environment is loaded and the working directory is changed to the user's home directory. This is usually what we want. If the user is not specified, the superuser is assumed. Notice that (strangely) the -l may be abbreviated as -, which is how it is most often used. To start a shell for the superuser, we would do this:

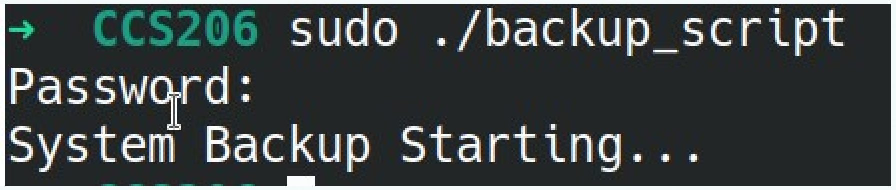


```
  CCS206 su -
Password:
localhost:~ #
```

- After entering the command, we are prompted for the superuser's password. If it is successfully entered, a new shell prompt appears indicating that this shell has superuser privileges. Type exit to go back to the normal user.

sudo – Execute a Command as Another User

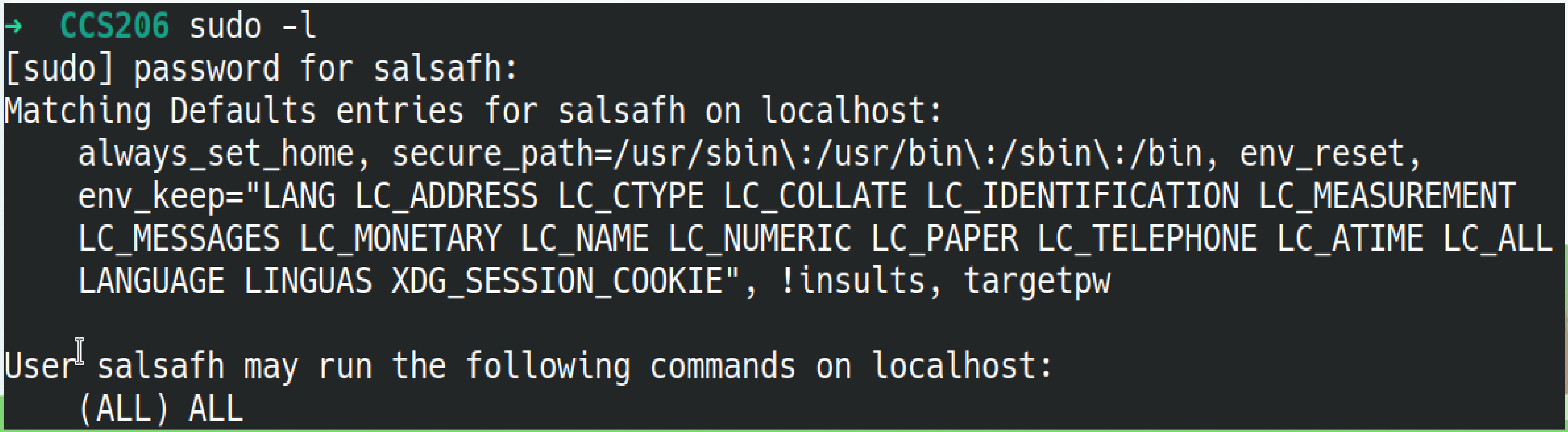
- The *sudo* command is like *su* in may ways but has some important additional capabilities. The administrator can configure *sudo* to allow an ordinary user to execute commands as a different user (usually the superuser) in a very controlled way. In particular, a user may be restricted to one or more specific commands and no others. Another important difference is that the use of *sudo* does not require access to the superuser’s password. To authenticated using *sudo*, the user enters his password. Let’s say, for example, that *sudo* has been configured to allow us to run a fictitious backup program called backup_script, which requires superuser privileges. With *sudo* it would be done like this:



```
→  CCS206 sudo ./backup_script
Password:
System Backup Starting...
```

sudo – Execute a Command as Another User

- After entering the command, we are prompted for our password (not the superuser's), and once the authentication is complete, the specified command is carried out. One important difference between *su* and *sudo* is that sudo does not start a new shell, nor does it load another user's environment. This means that commands do not need to be quoted any differently than they would be without using *sudo*. Note that this behavior can be overridden by specifying various options. See the *sudo* man page for details. To see what privileges are granted by *sudo*, use the *-l* option to list them:



```
→  CCS206 sudo -l
[sudo] password for salsafh:
Matching Defaults entries for salsafh on localhost:
    always_set_home, secure_path=/usr/sbin\:/usr/bin\:/sbin\:/bin, env_reset,
    env_keep="LANG LC_ADDRESS LC_CTYPE LC_COLLATE LC_IDENTIFICATION LC_MEASUREMENT
    LC_MESSAGES LC_MONETARY LC_NAME LC_NUMERIC LC_PAPER LC_TELEPHONE LC_ATIME LC_ALL
    LANGUAGE LINGUAS XDG_SESSION_COOKIE", !insults, targetpw

User salsafh may run the following commands on localhost:
    (ALL) ALL
```

chown – Change File Owner and Group

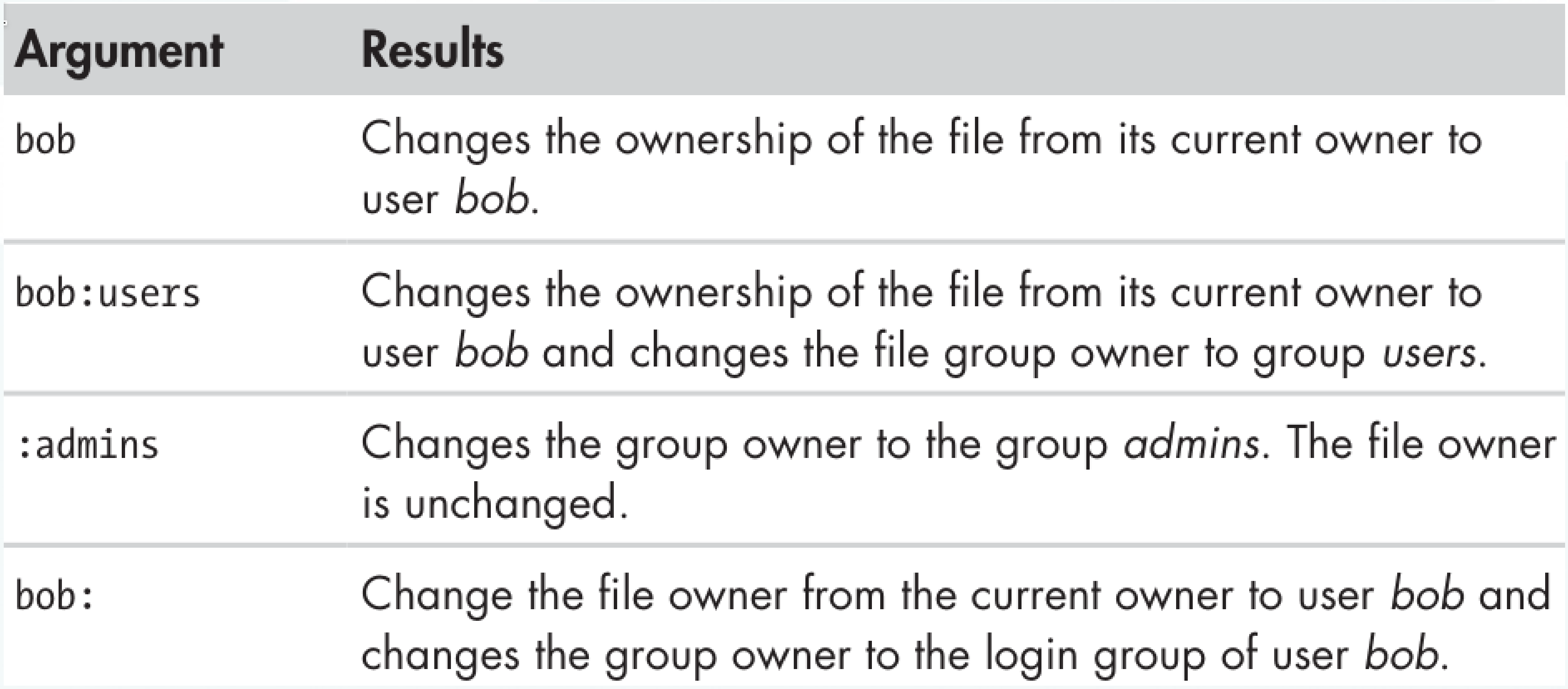
- The *chown* command is used to change the owner and group owner of a file or directory. Superuser privileges are required to use this command. The syntax of *chown* looks like this:

  chown [owner][:[gpriroup]] file…

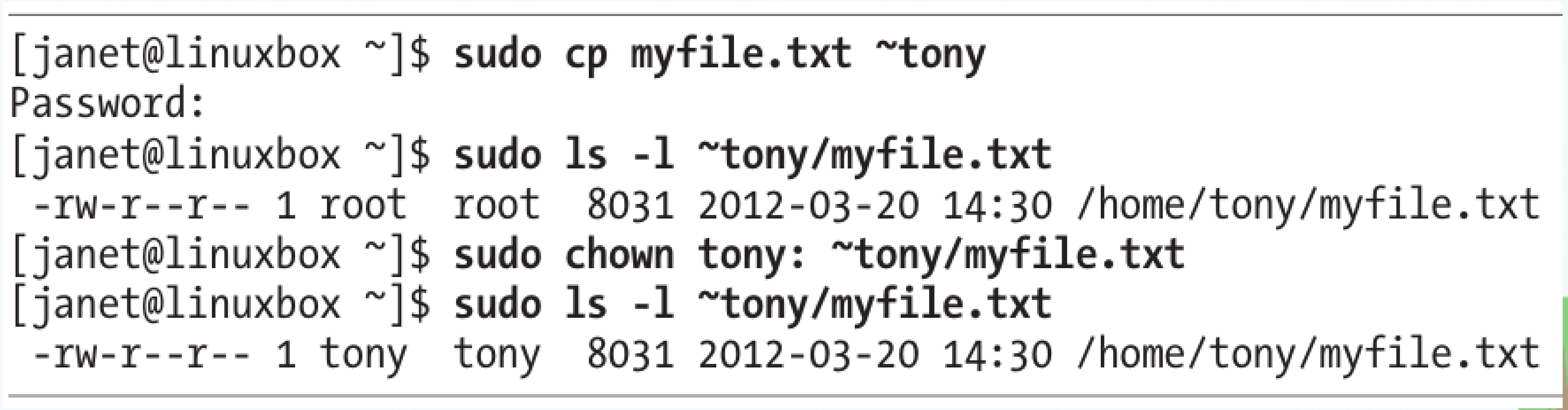
- chown can change the file owner and/or the file group owner depending on the first argument of the command. The table in the following slide lists some examples.

- Let’s say that we have two users: janet, who has access to superuser privileges, and tony, who does not. User janet wants to copy a file from her home directory to the home directory of user tony. Since user janet wants tony to be able to edit the file, jane changes the ownership of the copied file from janet to tony:

chown – Change File Owner and Group



Argument	Results
`bob`	Changes the ownership of the file from its current owner to user *bob*.
`bob:users`	Changes the ownership of the file from its current owner to user *bob* and changes the file group owner to group *users*.
`:admins`	Changes the group owner to the group *admins*. The file owner is unchanged.
`bob:`	Change the file owner from the current owner to user *bob* and changes the group owner to the login group of user *bob*.



```
[janet@linuxbox ~]$ sudo cp myfile.txt ~tony
Password:
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
 -rw-r--r-- 1 root  root  8031 2012-03-20 14:30 /home/tony/myfile.txt
[janet@linuxbox ~]$ sudo chown tony: ~tony/myfile.txt
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
 -rw-r--r-- 1 tony  tony  8031 2012-03-20 14:30 /home/tony/myfile.txt
```

chown – Change File Owner and Group

- Here we see user janet copy the file from her directory to the home directory of user tony. Next, janet changes the ownership of the file from root (a result of using sudo) to tony. Using the trailing colon in the first argument, janet also changed the group ownership of the file to the login group of tony, which happens to be group tony.
- Notice that after the first used of sudo, janet was not prompted for her password? This is because sudo, in most configurations, “trusts” you for several minutes (until its timer runs out).

chgrp – Change Group Ownership

- In older versions of Unix, the chown command changed only file ownership, not group ownership. For that purpose a separate command, chgrp, was used. It works much the same ways chown, except for being more limited.