Linux Commands Part 2

This is a continuation from my first Linux commands tutorial (Part #1), where I introduced you to getting command help, navigating the filesystem and working with files and directories. It can be accessed **here**.

To complete this tutorial, you will need access to a running Linux distribution, or 'distro' for short. There are a number of Linux 'distros'. If you do not already have access to a Linux system, I have a number of VirtualBox tutorials where I demonstrate the installation of CentOS 7 and Ubuntu 22 as virtual machines, accessible **here**.

I also have a few tutorials where I demonstrate the creation of AWS compute instances, RHEL 8 and Ubuntu 20, accessible **here**. Keep in mind that you will need to create an AWS account to be able to create a compute instance. If you do not have an AWS account, my tutorial **Create AWS Free Tier Account** is accessible **here.**

If you prefer, you can use a CentOS 7 or Ubuntu 22 VM instead. The choice is yours.

For this tutorial, I will be using both my RHEL 8, and Ubuntu 20, AWS compute instances, and I will be providing screenshots from my Ubuntu 20 instance. When I encounter a difference in output, I will include the RHEL 8 screenshot.

In this tutorial, I will go through some of the most common Linux commands and features that are used on a daily basis by all Linux users.

Below is a brief listing of the commands and features we will cover in this tutorial, along with brief descriptions.

Command	Description
`man`	• display manual pages for command
`--help`	• some commands have this option that displays command usage and descriptions of the available command options
`which / whereis`	• locate executable programs
`echo`	• display text
`>, >>, <`	• input / output redirection
`| (pipe)`	• combine the actions of several commands into one
`chmod`	• change file permissions
`chown, chgrp`	• change file ownership
`cat`	• display contents of files
`head`	• output the first part of file
`tail`	• output the last part of file
`ln`	• create links between files
`alias`	• create customized commands

Now that you have a running Linux system, are logged in and have access to the command line, we can begin.

Locating Executable Programs (commands)

We will begin by locating the echo command by executing the following:

$ which echo

```
ubuntu@ip-172-31-4-185:~$ which echo
/usr/bin/echo
ubuntu@ip-172-31-4-185:~$
```

You will notice that we are provided with the full path of the echo command.

Now, if we also wanted to know whether a command has an entry in the system's reference manuals (man pages), we issue:

$ whereis echo

```
ubuntu@ip-172-31-4-185:~$ whereis echo
echo: /usr/bin/echo /usr/share/man/man1/echo.1.gz
ubuntu@ip-172-31-4-185:~$
```

Like the **which** & **whereis** commands, the **echo** command is located in a directory that is part of our **PATH**. Therefore, we can use the **echo** command without having to type the full path to the command.

$ echo $PATH

```
ubuntu@ip-172-31-4-185:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin /usr/bin /sbin:/bin:/usr/games:/usr/local/games:/snap/bin
ubuntu@ip-172-31-4-185:~$
```

Next, we will use the echo command to display text to the screen.

$ echo "This is a Linux command tutorial."

```
ubuntu@ip-172-31-4-185:~$ echo "This is a Linux command tutorial."
This is a Linux command tutorial.
ubuntu@ip-172-31-4-185:~$
```

Display Contents of Files

First, we will create three new files for testing. (**NOTE**: this is known as output redirection, which we'll cover later)

$ echo "This is test file #1." > test_file1

$ echo "This is test file #2." > test_file2

$ echo "This is test file #3." > test_file3

Now, we'll verify that the files were created.

$ ls -l test_file1 test_file2 test_file3

```
ubuntu@ip-172-31-4-185:~$ echo "This is test file #1." > test_file1
ubuntu@ip-172-31-4-185:~$ echo "This is test file #2." > test_file2
ubuntu@ip-172-31-4-185:~$ echo "This is test file #3." > test_file3
ubuntu@ip-172-31-4-185:~$ ls -l test_file1 test_file2 test_file3
-rw-rw-r-- 1 ubuntu ubuntu 22 Apr 21 13:30 test_file1
-rw-rw-r-- 1 ubuntu ubuntu 22 Apr 21 13:30 test_file2
-rw-rw-r-- 1 ubuntu ubuntu 22 Apr 21 13:30 test_file3
ubuntu@ip-172-31-4-185:~$
```

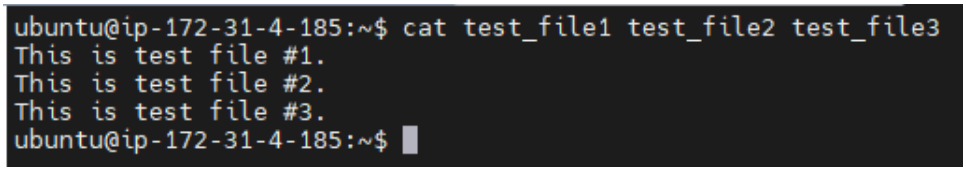
Let's now verify what test_file1 contains.

$ cat test_file1

```
ubuntu@ip-172-31-4-185:~$ cat test_file1
This is test file #1.
ubuntu@ip-172-31-4-185:~$
```

We can also pass more than one argument to the **cat** command.

```
$ cat test_file1 test_file2 test_file3
```



Standard File Streams

We will now cover Input/Output Streams. When commands are executed, there are three file streams available for use: standard input (usually keyboard), standard output and standard error are displayed on the terminal.

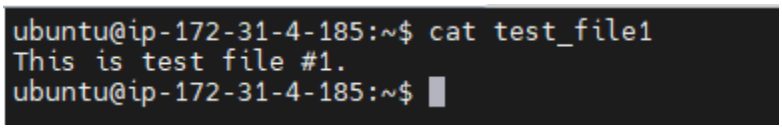
I/O Name	Abbreviation	File Descriptor
Standard Input	stdin	0
Standard Output	stdout	1
Standard Error	stderr	2

Input/Output Redirection

>	Redirects standard output to a file. Overwrites existing contents.
>>	Redirects standard output to a file. Appends to any existing contents.
<	Redirects input from a file to a command.
2>file	Redirect standard error to a file.
&	Used with redirection to signal that a file descriptor is being used.
2>&1	Combine stderr and standard output.
&>	shorthand syntax for the above
> /dev/null	Redirect output to nowhere.

Let's now verify what test_file1 contains.

```
$ cat test_file1
```

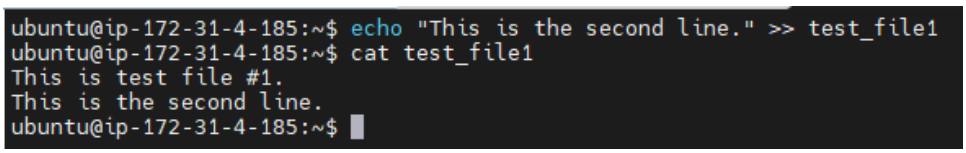


We will now append contents to test_file1:

```
$ echo "This is the second line." >> test_file1
```

Again, we'll verify test_file1's contents.

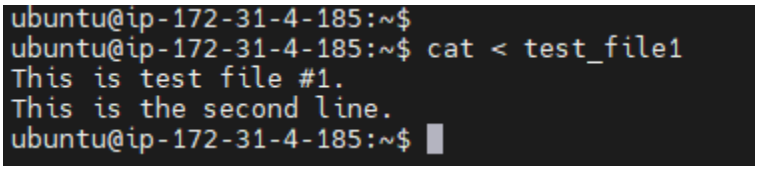
```
$ cat test_file1
```



You will notice that test_file1 now contains 2 lines of text.

We can also redirect input from test_file1 to the **cat** command which will display to standard output (stdout).

```
$ cat < test_file1
```



We know that test_file1 exists, but what happens if we include a non-existent file in our directory listing:

```
$ ls test_file1 test_file4
```

```
ubuntu@ip-172-31-4-185:~$ ls -l test_file1 test_file4
ls: cannot access 'test_file4': No such file or directory
-rw-rw-r-- 1 ubuntu ubuntu 47 Apr 18 09:20 test_file1
ubuntu@ip-172-31-4-185:~$
```

You will notice that both standard output (stdout) and standard error (stderr) are displayed on the terminal.

Let's redirect standard error (stderr) to nowhere:

```
$ ls test_file1 test_file4 2> /dev/null
```

```
ubuntu@ip-172-31-4-185:~$ ls -l test_file1 test_file4 2> /dev/null
-rw-rw-r-- 1 ubuntu ubuntu 47 Apr 18 09:20 test_file1
ubuntu@ip-172-31-4-185:~$
```

Now, test_file1 is displayed onscreen but not the error message.

Our next test will be to redirect both stdout & stderr to a file, then check the contents of that file.

```
$ ls test_file1 test_file4 > logfile 2>&1
```

```
$ cat logfile
```

```
ubuntu@ip-172-31-4-185:~$ ls -l test_file1 test_file4 > logfile 2>&1
ubuntu@ip-172-31-4-185:~$ cat logfile
ls: cannot access 'test_file4': No such file or directory
-rw-rw-r-- 1 ubuntu ubuntu 47 Apr 18 09:20 test_file1
ubuntu@ip-172-31-4-185:~$
```

Next, we will perform the same tasks using a shorthand method.

```
$ ls test_file1 test_file4 &> logfie
```

```
$ cat logfile
```

```
ubuntu@ip-172-31-4-185:~$ ls -l test_file1 test_file4 &> logfile
ubuntu@ip-172-31-4-185:~$ cat logfile
ls: cannot access 'test_file4': No such file or directory
-rw-rw-r-- 1 ubuntu ubuntu 47 Apr 18 09:20 test_file1
ubuntu@ip-172-31-4-185:~$
```

Create Command Pipeline using Pipes

The UNIX/Linux philosophy is to have many simple and short programs (or commands) work together to produce results, rather than have one complex program. In order to accomplish this, we use pipes; you can pipe the output of one command or program into another as its input.

In order to do this we use the vertical-bar, |, (pipe symbol) between commands as in:

```
$ command1 | command2 | command3
```

The above represents what we often call a pipeline and allows UNIX/Linux to combine the actions of several commands into one.

Let's check for the existence of the ubuntu user in the /etc/passwd file and get the line count of the result.

```
$ cat /etc/passwd | grep "ubuntu" | wc -l
```

```
ubuntu@ip-172-31-4-185:~$ cat /etc/passwd | grep "ubuntu"
ubuntu:x:1000:1000:Ubuntu:/home/ubuntu:/bin/bash
ubuntu@ip-172-31-4-185:~$ cat /etc/passwd | grep "ubuntu" | wc -l
1
ubuntu@ip-172-31-4-185:~$
```

The **grep** command searches for **ubuntu** in the **/etc/passwd** file and the result is piped to the **wc -l** command which returns the number of lines. In this case, there is only one user named ubuntu, so 1 is returned.

The **grep** and **wc** commands will be covered in a future tutorial (Part #3).

File Ownership and Permissions

In UNIX/Linux based operating systems, every file is associated with a user who is the owner. Every file is also associated with a group (a subset of all users). Finally, other users who are not the owner, or part of the group, can have access to a file. Access to a file is based on permissions: **read**, **write**, and **execute**.

The following commands are used to manage user and group ownership and permission settings.

Command	Usage
chown	Used to change user ownership of a file or directory
chgrp	Used to change group ownership
chmod	Used to change the permissions on the file which can be done separately for owner, group and the rest of the world (often named as other.)

Files have three kinds of permissions: read (r), write (w), execute (x). These are generally represented as **rwx**. These permissions affect three groups of owners: user/owner (u), group (g), and others (o). Using **chmod** in this fashion is known as symbolic notation.

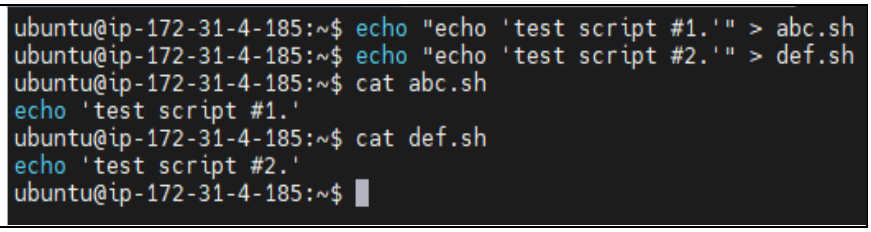
As a result, you have the following three groups of three permissions:

rwx: rwx: rwx

u: g: o:

Before proceeding with a few examples, we will first create two test scripts that will be used during in this section.

```
$ echo "echo 'test script #1.'" > abc.sh
$ echo "echo 'test script #2.'" > def.sh
$ cat abc.sh
$ cat def.sh
```



There are a number of different ways to use **chmod** with symbolic notation.

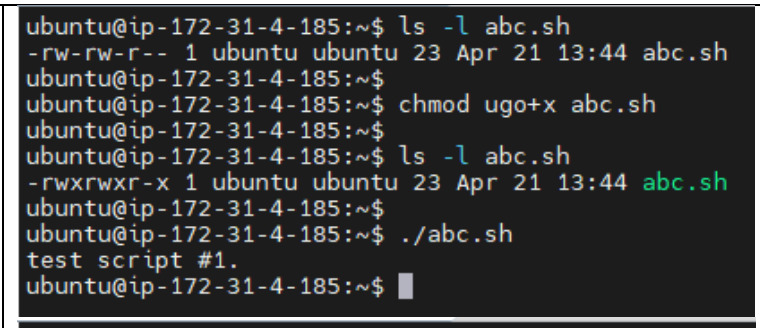
`chmod u=rwx,o= abc.sh`	give user read/write/execute permissions, while removing all permissions from others
`chmod u-x,g=rw,o+r abc.sh`	remove execute permission from user, give read/write access to group & read access to others
`chmod g-rw,o-wx def.sh`	remove group read/write permissions, while removing write/execute permissions from others
`chmod u+x,go-r def.sh`	give user execute permission, while removing read access from group & others
`chmod u-x,g+r,o+w def.sh`	remove execute permission from user, give group read access and give others write access

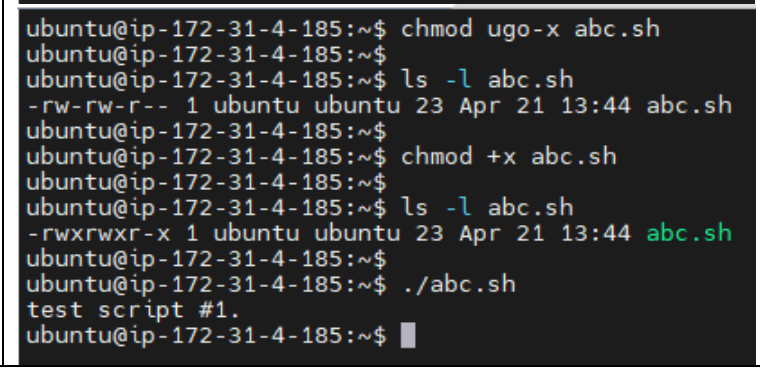
For instance, we will give the owner, group and others execute permission on **abc.sh**. (**u** stands for user (owner), **g** stands for group and **o** stands for other (everyone else))

```
$ ls -l abc.sh

$ chmod ugo+x abc.sh
$ ls -l abc.sh
# as the owner, I execute abc.sh
$ ./abc.sh

# remove execute permission
$ chmod ugo-x abc.sh
$ ls -l abc.sh
# we can omit ugo, if all 3
# require same permissions
$ chmod +x abc.sh
$ ls -l abc.sh
# as the owner, I execute abc.sh
$ ./abc.sh
```





This kind of syntax can be difficult to type and remember, so we can use a shorthand which lets you set all the permissions in one step and is referred to as octal notation. This is done by specifying all three permission bits for each entity (user, group or others). This digit is the sum of:

4 if read permission is desired.

2 if write permission is desired.

1 if execute permission is desired.

7 means read/write/execute, 6 means read/write, and 5 means read/execute.

When you apply this to the **chmod** command you have to give three digits, each digit represents the permissions assigned to each entity (user, group or others). For example, 765 can be translated into:

7 user has read/write/execute permissions

6 group has read/write permissions

5 others have read/execute permissions

For the next demonstration we will use **chmod** to set permissions using octal notation.

```
$ cat def.sh

$ ls -l def.sh # currently 664

$ chmod 755 def.sh

$ ls -l def.sh # now 755
# as the owner, I execute def.sh
$ ./def
```

```
ubuntu@ip-172-31-4-185:~$ cat def.sh
echo 'test script #2.'
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ls -l def.sh
-rw-rw-r-- 1 ubuntu ubuntu 23 Apr 21 13:44 def.sh
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ chmod 755 def.sh
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ls -l def.sh
-rwxr-xr-x 1 ubuntu ubuntu 23 Apr 21 13:44 def.sh
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ./def.sh
test script #2.
ubuntu@ip-172-31-4-185:~$
```

To change the ownership of a file, use the **chown** command. To change the group ownership use the **chgrp** command. Both operations require **sudo** privileges, which enable a user to run programs with the security privileges of the root user (superuser).

We will change the ownership of a file we created earlier, **test_file1**:

```
$ cat test_file1

$ ls -l test_file1

$ sudo chown root test_file1

$ ls -l test_file1
```

```
ubuntu@ip-172-31-4-185:~$ cat test_file1
This is test file #1.
This is the second line.
ubuntu@ip-172-31-4-185:~$ ls -l test_file1
-rw-rw-r-- 1 ubuntu ubuntu 47 Apr 21 13:38 test_file1
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ sudo chown root test_file1
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ls -l test_file1
-rw-rw-r-- 1 root ubuntu 47 Apr 21 13:38 test_file1
ubuntu@ip-172-31-4-185:~$
```

We will now change the group ownership of **test_file1**:

```
$ sudo chgrp root test_file1

$ ls -l test_file1
```

```
ubuntu@ip-172-31-4-185:~$ sudo chgrp root test_file1
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ls -l test_file1
-rw-rw-r-- 1 root root 47 Apr 21 13:38 test_file1
ubuntu@ip-172-31-4-185:~$
```

To perform both operations at once, issue the following:

```
$ sudo chown root:root test_file1
```

We will now see if we can view the contents of test_file1.

```
$ ls -l test_file1

$ cat test_file1
```

```
ubuntu@ip-172-31-4-185:~$ ls -l test_file1
-rw-rw-r-- 1 root root 47 Apr 21 13:38 test_file1
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat test_file1
This is test file #1.
This is the second line.
ubuntu@ip-172-31-4-185:~$
```

You will notice that we can still view the contents of **test_file1**, even though the **cat** command was executed by the **ubuntu** user. This is because '**others**' (world) still have read access to the file. Now, in order to change the permissions of a file we do not own, we will need to use **sudo** privileges.

```
$ ls -l test_file1

$ sudo chmod o-r test_file1

$ ls -l test_file1

$ cat test_file1


$ sudo cat test_file1
```

```
ubuntu@ip-172-31-4-185:~$ ls -l test_file1
-rw-rw-r-- 1 root root 47 Apr 21 13:38 test_file1
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ sudo chmod o-r test_file1
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ls -l test_file1
-rw-rw---- 1 root root 47 Apr 21 13:38 test_file1
ubuntu@ip-172-31-4-185:~$ cat test_file1
cat: test_file1: Permission denied
ubuntu@ip-172-31-4-185:~$ sudo cat test_file1
This is test file #1.
This is the second line.
ubuntu@ip-172-31-4-185:~$
```

As demonstrated above, to be able to view a file that we do not own (**test_file1**), or have permission to view, we will need to use **sudo** privileges.

Display Contents of System & Log Files

We now want to display the contents of the /etc/passwd file to check which users exist on the system:

```
$ cat /etc/passwd
```

```
ubuntu@ip-172-31-4-185:~$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-network:x:100:102:systemd Network Management,,,:/run/systemd:/usr/sbin/nologin
systemd-resolve:x:101:103:systemd Resolver,,,:/run/systemd:/usr/sbin/nologin
systemd-timesync:x:102:104:systemd Time Synchronization,,,:/run/systemd:/usr/sbin/nologin
messagebus:x:103:106::/nonexistent:/usr/sbin/nologin
syslog:x:104:110::/home/syslog:/usr/sbin/nologin
_apt:x:105:65534::/nonexistent:/usr/sbin/nologin
tss:x:106:111:TPM software stack,,,:/var/lib/tpm:/bin/false
uuidd:x:107:112::/run/uuidd:/usr/sbin/nologin
tcpdump:x:108:113::/nonexistent:/usr/sbin/nologin
sshd:x:109:65534::/run/sshd:/usr/sbin/nologin
landscape:x:110:115::/var/lib/landscape:/usr/sbin/nologin
pollinate:x:111:1::/var/cache/pollinate:/bin/false
ec2-instance-connect:x:112:65534::/nonexistent:/usr/sbin/nologin
systemd-coredump:x:999:999:systemd Core Dumper:/:/usr/sbin/nologin
ubuntu:x:1000:1000:Ubuntu:/home/ubuntu:/bin/bash
lxd:x:998:100::/var/snap/lxd/common/lxd:/bin/false
ubuntu@ip-172-31-4-185:~$
```

You will notice that the **ubuntu** user is present in the file on my Ubuntu 20 compute instance.

Now, on my RHEL 8 compute instance.

```
[ec2-user@ip-172-31-5-221 ~]$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:65534:65534:Kernel Overflow User:/:/sbin/nologin
dbus:x:81:81:System message bus:/:/sbin/nologin
systemd-coredump:x:999:997:systemd Core Dumper:/:/sbin/nologin
systemd-resolve:x:193:193:systemd Resolver:/:/sbin/nologin
tss:x:59:59:Account used for TPM access:/dev/null:/sbin/nologin
polkitd:x:998:996:User for polkitd:/:/sbin/nologin
unbound:x:997:994:Unbound DNS resolver:/etc/unbound:/sbin/nologin
sssd:x:996:993:User for sssd:/:/sbin/nologin
chrony:x:995:992::/var/lib/chrony:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
ec2-user:x:1000:1000:Cloud User:/home/ec2-user:/bin/bash
[ec2-user@ip-172-31-5-221 ~]$
```

You will notice that the **ec2-user** is present in the file on my RHEL 8 compute instance.

Having connected to both of my AWS instances (RHEL 8 & Ubuntu 20), I know that the SSHD service is running.

Two other commands, **head** and **tail**, can be used to view the top or bottom parts of files. They are useful for viewing parts of log files. By default, 10 lines are returned. By using the **-n** option, I can specify the number of lines returned by both the **head** and **tail** commands. Let's check the **SSHD** log entries for both.

For Ubuntu 20, the SSHD log entries are stored in /var/log/auth.log

NOTE: auth.log group is **adm**, **ubuntu** user is part of the group **adm**, so sudo is **not** required to execute command.

```
ubuntu@ip-172-31-4-185:~$ ls -l /var/log/auth.log
-rw-r----- 1 syslog adm 190996 Apr 18 08:49 /var/log/auth.log
ubuntu@ip-172-31-4-185:~$ id
uid=1000(ubuntu) gid=1000(ubuntu) groups=1000(ubuntu),4(adm),20(dialout),24(cdrom),
dev),117(netdev),118(lxd)
ubuntu@ip-172-31-4-185:~$
```

$ head -n 5 /var/log/auth.log

$ tail -n 5 /var/log/auth.log

```
ubuntu@ip-172-31-4-185:~$ head -n 5 /var/log/auth.log
Apr 17 00:00:38 ip-172-31-4-185 sshd[67084]: Invalid user adm from 92.255.85.135 port 63708
Apr 17 00:00:38 ip-172-31-4-185 sshd[67084]: Received disconnect from 92.255.85.135 port 63708:11: Bye Bye [preauth]
Apr 17 00:00:38 ip-172-31-4-185 sshd[67084]: Disconnected from invalid user adm 92.255.85.135 port 63708 [preauth]
Apr 17 00:14:47 ip-172-31-4-185 sshd[67497]: Connection closed by authenticating user bin 176.111.173.44 port 54220 [preauth]
Apr 17 00:17:01 ip-172-31-4-185 CRON[67502]: pam_unix(cron:session): session opened for user root by (uid=0)
ubuntu@ip-172-31-4-185:~$ tail -n 5 /var/log/auth.log
Apr 18 09:00:13 ip-172-31-4-185 sshd[72874]: Received disconnect from 142.93.134.242 port 39928:11: Bye Bye [preauth]
Apr 18 09:00:13 ip-172-31-4-185 sshd[72874]: Disconnected from invalid user yazminvl 142.93.134.242 port 39928 [preauth]
Apr 18 09:00:48 ip-172-31-4-185 sshd[72879]: Invalid user operator from 92.255.85.135 port 58600
Apr 18 09:00:48 ip-172-31-4-185 sshd[72879]: Received disconnect from 92.255.85.135 port 58600:11: Bye Bye [preauth]
Apr 18 09:00:48 ip-172-31-4-185 sshd[72879]: Disconnected from invalid user operator 92.255.85.135 port 58600 [preauth]
ubuntu@ip-172-31-4-185:~$
```

For RHEL 8, the SSHD log entries are stored in /var/log/secure

```
[ec2-user@ip-172-31-5-221 ~]$ ls -l /var/log/secure
-rw-------. 1 root root 1015313 Apr 21 14:11 /var/log/secure
[ec2-user@ip-172-31-5-221 ~]$
```

You will notice from the image above that the **ec2-user** does **not** have permission to view /var/log/secure, **sudo** privileges required.

```
$ sudo head -n 5 /var/log/secure
```

```
$ sudo tail -n 5 /var/log/secure
```

```
[ec2-user@ip-172-31-5-221 ~]$ sudo head -n 5 /var/log/secure
Apr 17 03:37:44 ip-172-31-5-221 sshd[18009]: error: kex_exchange_identification: Connection closed by remote host
Apr 17 03:37:46 ip-172-31-5-221 sshd[18010]: error: kex_exchange_identification: Connection closed by remote host
Apr 17 03:41:06 ip-172-31-5-221 sshd[18019]: error: kex_exchange_identification: read: Connection reset by peer
Apr 17 03:41:13 ip-172-31-5-221 sshd[18020]: Connection reset by 45.67.34.100 port 14282 [preauth]
Apr 17 03:41:13 ip-172-31-5-221 sshd[18021]: Connection reset by 45.67.34.100 port 14330 [preauth]
[ec2-user@ip-172-31-5-221 ~]$ sudo tail -n 5 /var/log/secure
Apr 18 09:09:36 ip-172-31-5-221 sshd[24140]: Disconnected from authenticating user root 180.76.149.53 port 38254 [
Apr 18 09:09:44 ip-172-31-5-221 sudo[24143]: ec2-user : TTY=pts/0 ; PWD=/home/ec2-user ; USER=root ; COMMAND=/bin/
e
Apr 18 09:09:44 ip-172-31-5-221 sudo[24143]: pam_systemd(sudo:session): Cannot create session: Already running in
Apr 18 09:09:44 ip-172-31-5-221 sudo[24143]: pam_unix(sudo:session): session opened for user root by ec2-user(uid=
Apr 18 09:09:44 ip-172-31-5-221 sudo[24143]: pam_unix(sudo:session): session closed for user root
[ec2-user@ip-172-31-5-221 ~]$
```

Hard and Soft (Symbolic) Links

The **ln** command can be used to create hard links and soft links (with the **-s** option).

We will create a hard link named test_file4

```
$ ln test_file3 test_file4
```

Note that two files now appear to exist. We will perform a closer inspection of the file listing

```
$ ls -li test_file3 test_file4
```

```
ubuntu@ip-172-31-4-185:~$ ln test_file3 test_file4
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ls -l test_file3 test_file4
-rw-rw-r-- 2 ubuntu ubuntu 21 Apr 21 14:28 test_file3
-rw-rw-r-- 2 ubuntu ubuntu 21 Apr 21 14:28 test_file4
ubuntu@ip-172-31-4-185:~$
```

The **-i** option to **ls** prints out in the first column the **inode** number, which is unique for each file object. This field is the same for both of these files. It is only one file but it has more than one name associated with it, note the 2 that appears in the **ls** output.

If we delete either of the two files (**test_file3** or the linked **test_file4**), a file will remain.

```
ubuntu@ip-172-31-4-185:~$ rm test_file3
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ls -li test_file3 test_file4 2> /dev/null
256327 -rw-rw-r-- 1 ubuntu ubuntu 21 Apr 21 14:28 test_file4
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat test_file4
This is test file #3
ubuntu@ip-172-31-4-185:~$
```

Notice that **test_file4** has the same contents that **test_file3** had.

Now we will create the symbolic link named **test_file5** and perform a file listing for closer inspection.

```
$ ln -s test_file2 test_file5
```

```
$ ls -li test_file2 test_file5
```

```
ubuntu@ip-172-31-4-185:~$ ln -s test_file2 test_file5
ubuntu@ip-172-31-4-185:~$ ls -li test_file2 test_file5
256207 -rw-rw-r-- 1 ubuntu ubuntu 22 Apr 18 09:16 test_file2
256376 lrwxrwxrwx 1 ubuntu ubuntu 10 Apr 18 09:58 test_file5 -> test_file2
ubuntu@ip-172-31-4-185:~$
```

You will notice that the inode numbers are different and that test_file5 is linked to test_file2.

We can delete test_file5 but **not** test_file2. After test_file2 is deleted, test_file5 is no longer accessible.

```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ rm test_file2
ubuntu@ip-172-31-4-185:~$ ls -li test_file2 test_file5 2> /dev/null
307445 lrwxrwxrwx 1 ubuntu ubuntu 10 Apr 18 10:12 test_file5 -> test_file2
ubuntu@ip-172-31-4-185:~$ cat test_file5
cat: test_file5: No such file or directory
ubuntu@ip-172-31-4-185:~$ rm test_file5
ubuntu@ip-172-31-4-185:~$
```

Create customized commands with Aliases

You can create customized commands by creating aliases. Most often these aliases can then be placed in your ~/**.bashrc** file so they are available to any command shells you create. (**NOTE**: the tilde **~** represents your home directory)

Typing alias with no arguments will list currently defined aliases.

```
$ alias
```

```
ubuntu@ip-172-31-4-185:~$ alias
alias alert='notify-send --urgency=low -i "$([ $? = 0 ] && echo terminal
;s/[;&|]\s*alert$//'\'')"'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
ubuntu@ip-172-31-4-185:~$
```

Let's create an alias for easy access to a project directory. First, we will create the directory. Then, the alias.

On my Ubuntu 20 instance:

```
$ mkdir -p /home/ubuntu/projects/my_linux_stuff
```

```
$ alias stuff='cd /home/ubuntu/projects/my_linux_stuff'
```

```
ubuntu@ip-172-31-4-185:~$ mkdir -p /home/ubuntu/projects/my_linux_stuff
ubuntu@ip-172-31-4-185:~$ alias stuff='cd /home/ubuntu/projects/my_linux_stuff'
ubuntu@ip-172-31-4-185:~$ alias
alias alert='notify-send --urgency=low -i "$([ $? = 0 ] && echo terminal || echo error)"
//;s/[;&|]\s*alert$//'\'')"'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
alias stuff='cd /home/ubuntu/projects/my_linux_stuff'
ubuntu@ip-172-31-4-185:~$
```

Now, we will verify that using the alias **stuff** will automatically change to the directory specified.

```
$ stuff
```

```
ubuntu@ip-172-31-4-185:~$ stuff
ubuntu@ip-172-31-4-185:~/projects/my_linux_stuff$ pwd
/home/ubuntu/projects/my_linux_stuff
ubuntu@ip-172-31-4-185:~/projects/my_linux_stuff$
```

It worked, but for it to always be available, as stated in **.bashrc**, I should add any new aliases to a file called **.bash_aliases** in my home directory (/home/ubuntu).

```
$ cd
```

```
$ cat .bashrc | grep aliases
```

```
ubuntu@ip-172-31-4-185:~$ cat .bashrc | grep aliases
# enable color support of ls and also add handy aliases
# some more ls aliases
# ~/.bash_aliases, instead of adding them here directly.
if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
ubuntu@ip-172-31-4-185:~$
```

Although the file does not already exist, I will use the append redirection operator to add the **stuff** alias to the file. I will do this with the following command:

```
$ echo "alias stuff='cd /home/ubuntu/projects/my_linux_stuff'" >> .bash_aliases
```

```
$ cat .bash_aliases
```

```
ubuntu@ip-172-31-4-185:~$ echo "alias stuff='cd /home/ubuntu/projects/my_linux_stuff'" >> .bash_aliases
ubuntu@ip-172-31-4-185:~$ cat .bash_aliases
alias stuff='cd /home/ubuntu/projects/my_linux_stuff'
ubuntu@ip-172-31-4-185:~$
```

I will now test this by opening another SSH connection to my Ubuntu 20 instance, and verify.

```
ubuntu@ip-172-31-4-185:~$ alias
alias alert='notify-send --urgency=low -i "$([ $? = 0 ] && echo terminal || echo error)"
;s/[;&|]\s*alert$//'\'')"'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
alias stuff='cd /home/ubuntu/projects/my_linux_stuff'
ubuntu@ip-172-31-4-185:~$ stuff
ubuntu@ip-172-31-4-185:~/projects/my_linux_stuff$ pwd
/home/ubuntu/projects/my_linux_stuff
ubuntu@ip-172-31-4-185:~/projects/my_linux_stuff$
```

You will notice that in the newly opened session I was able to successfully execute the stuff alias.

On my RHEL 8 instance:

```
$ alias
```

```
[ec2-user@ip-172-31-5-221 ~]$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias which='(alias; declare -f) | /usr/bin/which --tty-only --read-alias
alias xzegrep='xzegrep --color=auto'
alias xzfgrep='xzfgrep --color=auto'
alias xzgrep='xzgrep --color=auto'
alias zegrep='zegrep --color=auto'
alias zfgrep='zfgrep --color=auto'
alias zgrep='zgrep --color=auto'
[ec2-user@ip-172-31-5-221 ~]$
```

```
$ mkdir -p /home/ec2-user/projects/my_linux_stuff
$ alias stuff='cd /home/ec2-user/projects/my_linux_stuff'
```

```
[ec2-user@ip-172-31-5-221 ~]$ mkdir -p /home/ec2-user/projects/my_linux_stuff
[ec2-user@ip-172-31-5-221 ~]$ alias stuff='cd /home/ec2-user/projects/my_linux_stuff'
[ec2-user@ip-172-31-5-221 ~]$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias stuff='cd /home/ec2-user/projects/my_linux_stuff'
alias which='(alias; declare -f) | /usr/bin/which --tty-only --read-alias --read-functions
alias xzegrep='xzegrep --color=auto'
alias xzfgrep='xzfgrep --color=auto'
alias xzgrep='xzgrep --color=auto'
alias zegrep='zegrep --color=auto'
alias zfgrep='zfgrep --color=auto'
alias zgrep='zgrep --color=auto'
[ec2-user@ip-172-31-5-221 ~]$
```

Now, we will verify that the alias **stuff** will automatically change to the directory specified.

```
$ stuff
```

```
[ec2-user@ip-172-31-5-221 ~]$ stuff
[ec2-user@ip-172-31-5-221 my_linux_stuff]$ pwd
/home/ec2-user/projects/my_linux_stuff
[ec2-user@ip-172-31-5-221 my_linux_stuff]$
```

It worked, but for it to always be available, I will need to add it to **.bashrc** in my home directory (/home/ec2-user). I will use the append redirection operator to append the **stuff** alias to the file. I will do this with the following command:

```
$ cd
$ echo "alias stuff='cd /home/ec2-user/projects/my_linux_stuff'" >> .bashrc
$ cat .bashrc | grep alias
```

```
[ec2-user@ip-172-31-5-221 ~]$ echo "alias stuff='cd /home/ec2-user/projects/my_linux_stuff'" >> .bashrc
[ec2-user@ip-172-31-5-221 ~]$ cat .bashrc | grep alias
# User specific aliases and functions
alias stuff='cd /home/ec2-user/projects/my_linux_stuff'
[ec2-user@ip-172-31-5-221 ~]$
```

I will now test this by opening another SSH connection to my RHEL 8 instance, and verify.

```
[ec2-user@ip-172-31-5-221 ~]$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias stuff='cd /home/ec2-user/projects/my_linux_stuff'
alias which='(alias; declare -f) | /usr/bin/which --tty-only --read-alias
alias xzegrep='xzegrep --color=auto'
alias xzfgrep='xzfgrep --color=auto'
alias xzgrep='xzgrep --color=auto'
alias zegrep='zegrep --color=auto'
alias zfgrep='zfgrep --color=auto'
alias zgrep='zgrep --color=auto'
[ec2-user@ip-172-31-5-221 ~]$ stuff
[ec2-user@ip-172-31-5-221 my_linux_stuff]$ pwd
/home/ec2-user/projects/my_linux_stuff
[ec2-user@ip-172-31-5-221 my_linux_stuff]$
```

You will notice that in the newly opened session I was able to successfully execute the **stuff** alias.

I hope you have enjoyed completing this tutorial and found it helpful.

We discussed how to locate commands and display the contents of files. We also covered the standard file streams, followed by working with input/output redirection, as well as, making use of the command pipeline. There was also an introduction to file ownership and permissions. Finally, we touched on file links and aliases.

In my next Linux commands tutorial (Part #3), we will discuss wildcard operators, regular expressions and search patterns. These can be used to search for files on the system (using find), as well as, search the contents of files (using grep). I will also introduce you to a number of utilities that allow us to change the contents of files and reformat standard output. It can be accessed **here**.

If you are interested in continuing your Linux learning journey, I have a number of other Linux tutorials that can be accessed **here**, while my main tutorials page can be accessed **here**.

Back to Top