Tanvir Anjom Siddique (Alvi)

RUET_CSE19_A

Lab 1st cycle:

Shell: command line interpreter.
Command:
whoami → user info
cwd →current working directory

mkdir→ make directory
mkdir **folder1 folder2**

mkdir folder1/newDirectory

Inside present directory there is 'folder1' inside that directory we will create 'newDirectory' folder.

Clear → shell screen

ls → list of folder and files in current dirctory

cd tilde_sign → goto to home directory
cd ..

touch fileName.txt
touch→ create file

cd / → goto root directory

up/down arrow→ previous or next command will come in shell

mv file.txt **testFolder/**

move file.txt file to testFolder

cp file.txt **testFolder/file.txt**

copy file.txt to testFolder

rm **file.txt**

Remove/Delete file: rm

rmdir → remove directory

ls -l → use long listing format

-h → human readable list
-a → show hidden file

ls -l -h -a
ls -lha → work same

ls -formatName

man commandName → mannual / details of a command

dot before file/folder → hide folder

rm folderName → this folder will be deleted/removed only if it's empty

rm -r folderName → recursively **remove all files/folder** of folderName

files/forlder Removed/deleted by rm command will not move to trash but deleted permanently

cd / → goto root folder
ls → list of files/folder in current directory

cd /*usr/*bin → periferal files are stored
cd /bin → critial commands are stored in this directory → for linux recovery for 1 user files stored here

sudo apt-get update → update repository

apt-get install gcc →can't install gcc compiler → no permission

sudo apt-get install gcc → install c compiler
or,

su→ user=root

sudo → root user will execute this following command which may other user can't execute

cd / → goto root

rm -rf / or, rm -rf . → delete all files including os from root folder → pc will cush
rf→ recursively remove folder and others

ctrl+ c → **quit** something

nano 1.txt → open 1.txt file in shell

more 1.txt → show content of 1.txt

head -n 2 1.txt → 1st 2 line of 1.txt

tail -n 2 1.txt → last 2 line of 1.txt will be shown

ls .. → show contents of current directorys upper directorys all contents
cd .

Mkdir ../test → one directory back create folder test
mkdir ../../os → 1 dirctory up → again 1 directory up → create folder os

touch test.c → create c file
nano test.c →open file and write c code

gcc test.c -o test
./test → run compiled file
Hello linux

echo $? → show return code after execution of c program

tanviranjomsiddique@tanviranjomsiddique-virtual-machine:~/Documents/LabData/oslab-19b$ nano test.c
tanviranjomsiddique@tanviranjomsiddique-virtual-machine:~/Documents/LabData/oslab-19b$ gcc test.c -o o
tanviranjomsiddique@tanviranjomsiddique-virtual-machine:~/Documents/LabData/oslab-19b$ ./o
v= 10
Hello linux
tanviranjomsiddique@tanviranjomsiddique-virtual-machine:~/Documents/LabData/oslab-19b$ echo $?
0

which touch → get location of bin of touch command
which gcc→ get location of bin of gcc compiler

Setup C,C++ environment, Compile& Run:

Although you can install the C++ compiler separately by installation of the gcc package, the recommended way to install the C++ compiler on Ubuntu 22.04 is by installation of the entire development package build-essential.

Get started by opening a command line terminal and typing the following two commands to install C++ compiler by installation of the development package build-essential:

$ sudo apt update
$ **sudo *apt install* build-essential**

DO YOU NEED MULTIPLE C AND/OR C++ COMPILER VERSIONS?
Visit "How to switch between multiple GCC and G++ compiler versions on Ubuntu 22.04" to see how to install multiple compiler versions on the same Ubuntu system.
Check C compiler version to verify installation:

$ gcc --version
gcc (Ubuntu 11.2.0-16ubuntu1) 11.2.0

Create a basic C++ code source. For example let's create hello world C++ program. Save the following code as hello.cc text file:
touch hello.cc

```
#include
using namespace std;

int main()
{
   cout << "Hello, World!";
   return 0;
}
```

Save the above code within hello.cc file, compile and execute it:
Ctrt+X > enter

$ **g++ -o hello hello.cc**
$ **./hello**
Hello, World!

Closing Thoughts

In this tutorial, we saw how to install G++, the very simple C++ compiler on Ubuntu 22.04 Jammy Jellyfish. We also learned how to create a simple Hello World C++ program in order to test the new compiler.

Lab 3rd cycle:

ls *.txt →show all text file in directory.
ls f* → show all directory and files name starting with f.

Find:

dot →present directory
double dot →previous directory of current directory

find . -type f -name "*.cc" → in current directory find all files whose name has 'anyname.cc' substring.

Type = f -> find files , d→ find directory
Find. = find in current directory

man find →mannual of find

find . -name "2*.cc"

WC:

wc → print newline,word and byte counts for each file.

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **wc 2.cc**
 8 14 106 2.cc

There is 8 lines ,14 words & 106 characters in 2.cc file.

Man wc → see all function and manual of wc

Wc -l → count lines in file

Pipelining (command1 | command2):

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ find . -type f -name "2*.cc" | wc -l

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ wc -l 2.cc
8 2.cc
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **find . -type f -name "2.cc"**
./2.cc

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **find . -type f -name "2.cc" | wc**
    1   1   7

cat attendance.txt | sort | uniq -c

cat→ read files
uniq -c → count unique numbers occurance (compare consecutive numbers)→so file should be sorted.

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **wc -l 2.cc**
8 2.cc
→show no of lines in 2.cc file
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **find . -type f -name "2.cc" | wc**
1 1 7
→show no of line,words,characters →of outut of find command

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **nano 2.cc**
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **find . -type f -name "2.cc"**
./2.cc
→ There is 2.cc file in current directory
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **find . -type f -name "2.cc">outterminal.txt**
→ create outterminal.txt if not created & store output of the command in it
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **cat outterminal.txt**
./2.cc
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **wc outterminal.txt**
1 1 7 outterminal.txt
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$

Find with Regular Expression:

command1: find . -type f -name "*.cc"

= find files with anyname.cc

command2: grep -E "f[0-9]\.txt"

= find files whose name has substing ‘ f+number+.txt ‘

. → in regular expression any character

find . -type f -name "*.cc" | grep -E "f[0-9]\.txt"

→ execute command 1 → get output of command1. Then apply command2 at **output** of command1.

Lab2nd30/**f**older/m1**.txt** → will also be shown.Here os can’t differentiate file and directory name.

Ls | grep -E “lab$” → show all whose name has substring ‘lab’ at the end

$--> last

upperArrow→1st

grep -E “13[0-9]” attendance.txt→ find numbers 130-139 in attendance.txt file

alternative:

cat attendance.txt | grep -E “13[0-9]”

grep -E “1^[2][0-9]” attendance.txt ->show 3 ditgit numbers whose 1st digit 1 and middle number can’t be 2.

Grep -E →grep ‘s extended version.Provide some features that only grep don’t support.

Command line argument C++:

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **cat 2.cc**

```
#include<bits/stdc++.h>
using namespace std;

int main(int argc,char* argv[])
{
//char* -> string
//char* argv[] =array of  strings

//by default argument no=1 if no argument is passed
   cout<<"No of arguments:"<<argc<<endl;
   if(argc>1){
        for(int i=0;i<argc;i++){
                cout<<"arg["<<i<<"]="<<argv[i]<<endl;
        }
```

```
    }
    cout << "Hello, World!\n";
    return 0;
}
```

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **g++ 2.cc -o 2**
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **./2 1 2 hello world hi**

```
No of arguments:6
arg[0]=./2
arg[1]=1
arg[2]=2
arg[3]=hello
arg[4]=world
arg[5]=hi
Hello, World!
```

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **echo $?**

```
0
```

Return 0→ program executed correctly.
Batch script file extention → .sh

Bash script:

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **cat 2.cc**
→see contents of 2.cc file in terminal

```
#include<bits/stdc++.h>
using namespace std;

int main(int argc,char* argv[])
{
//char* -> string
//char* argv[] =array of  strings

//by default argument no=1 if no argument is passed
    cout<<"No of arguments:"<<argc<<endl;
    if(argc>1){
        for(int i=0;i<argc;i++){
            cout<<"arg["<<i<<"]="<<argv[i]<<endl;
        }
    }
    cout << " End of program! \n";
    return 0;
}
```

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **touch bach_file.sh**
→create file

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **nano bach_file.sh**

→edit file

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **which bash**
/usr/bin/bash

→get bash files bin's location in your device which will execute ./bashfile.sh command

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **cat bach_file.sh**

```
#!/usr/bin/bash

echo "C++ is compiled and executed using bash file\n"
g++ 2.cc -o 2
./2 arg1 arg2 arg3
```

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **./bach_file.sh**
bash: ./bach_file.sh: Permission denied

bash script.sh → can execute without shebang **"#!/usr/bin/bash" line in .sh file→ don't need permission to execute script.sh**

./bach_file.sh → this is passed to bash file→ location of bin of bash must be specified.

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$

```
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ ./bach_file.sh
bash: ./bach_file.sh: Permission denied
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ ls -l
total 40
-rwxrwxr-x 1 alviubuntu alviubuntu 17736 অক্টোবর  1 22:32 2
-rwxrwxrwx 1 alviubuntu alviubuntu   373 অক্টোবর  1 22:30 2.cc
-rw-rw-r-- 1 alviubuntu alviubuntu   104 অক্টোবর  1 09:44 bach_file.sh
-rw-rw-r-- 1 alviubuntu alviubuntu    46 অক্টোবর  1 10:30 bash_practice.sh
drwxrwxr-x 2 alviubuntu alviubuntu  4096 সেপ্টেম্বর 18 11:03 oslab_cse19A
-rw-rw-r-- 1 alviubuntu alviubuntu     7 অক্টোবর  1 08:47 outterminal.txt
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ sudo ./bach_file.sh
[sudo] password for alviubuntu:
sudo: ./bach_file.sh: command not found
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ bash bach_file.sh
```

→execute bach_files.sh even if this file has not execution permission(bash program execute this file.) bash command will read bach_file.sh(there is Read permission)& execute it's commands.

→& no need for "#!/usr/bin/bash" line.

C++ is compiled and executed using bash file\n
No of arguments:4
arg[0]=./2
arg[1]=arg1
arg[2]=arg2
arg[3]=arg3
End of program!
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **chmod a+rwx bach_file.sh**
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **./bach_file.sh**
C++ is compiled and executed using bash file\n
No of arguments:4
arg[0]=./2
arg[1]=arg1
arg[2]=arg2
arg[3]=arg3
End of program!

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **./bash_practice.sh**
bash: ./bash_practice.sh: Permission denied
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **bash bash_practice.sh**
This is a bash variable\n
→-rw-rw-r-- 1 alviubuntu alviubuntu 46 অক্টোবর 1 10:30 bash_practice.sh
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **chmod a-rwx bash_practice.sh**

- **No permission to all**
- **- : remove permission**
- **+ : add permission**

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **./bash_practice.sh**
bash: ./bash_practice.sh: Permission denied
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **bash bash_practice.sh**
bash: bash_practice.sh: Permission denied
→---------- 1 alviubuntu alviubuntu 46 অক্টোবর 1 10:30 bash_practice.sh

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **chmod a+rwx bash_practice.sh**

- **chmod= CHange permission Mode**
- **a = for All**
- **+ = add permissions**
- **- = Remove permission**
- **Rwx = Read,Write,eXecute to file bash_practice.sh**

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **./bash_practice.sh**
This is a bash variable\n
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$

See and Provide Permission of a file or directory:

ls -l → get details Permission and others of all files and directories in current directory.
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ ls -l
total 36
-rwxrwxr-x 1 alviubuntu alviubuntu 17736 অক্টোবর 1 09:44 2
-rwxrwxrwx 1 alviubuntu alviubuntu 371 অক্টোবর 1 09:17 2.cc
-rw-rw-r-- 1 alviubuntu alviubuntu 104 অক্টোবর 1 09:44 bach_file.sh
drwxrwxr-x 2 alviubuntu alviubuntu 4096 সেপ্টেম্বর 18 11:03 oslab_cse19A
-rw-rw-r-- 1 alviubuntu alviubuntu 7 অক্টোবর 1 08:47 outterminal.txt

-rwxrwxrwx → all permission for this file is given
- =means file
d = directory

-Owner FileGroup Others
drwxrwxr-x

1. D→ directory
2. Owner permission: rwx
3. File Group permission: rwx
4. Other permission: r-x (read and execute but can't write)

each part:
rwx → read,write,execute
r-x → read , ,execute permission given → but write permission not given

rw- → read , write, permission given → but execution permission not given

4 2 1
rwx:

4→ read permission only
r+w=6 → read and write permission
ls -l:
-rw-rw-r-- 1 alviubuntu alviubuntu 371 অক্টোবর 1 09:17 2.cc

Give Permissions:

- owner group: read & write → r+w→4+2=6
- file group: read and execute →r+x→4+1=5

- other: read→4

chmod 654 2.cc

chmod 777 <filename> →

- owner group:7
- file group:7
- other permission: 7

Provide read,write,execution permission of 2.cc to owner,file group,and others:
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **chmod 777 2.cc**
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **ls -l**
total 36
-rwxrwxr-x 1 alviubuntu alviubuntu 17736 অক্টোবর 1 09:17 2
-rwxrwxrwx 1 alviubuntu alviubuntu 371 অক্টোবর 1 09:17 2.cc

Chmod u+x filename → u=user +x= add permission to execute
Chmod u+x 2.cc

chmod u-x filename → u=user -x= remove permission to execute

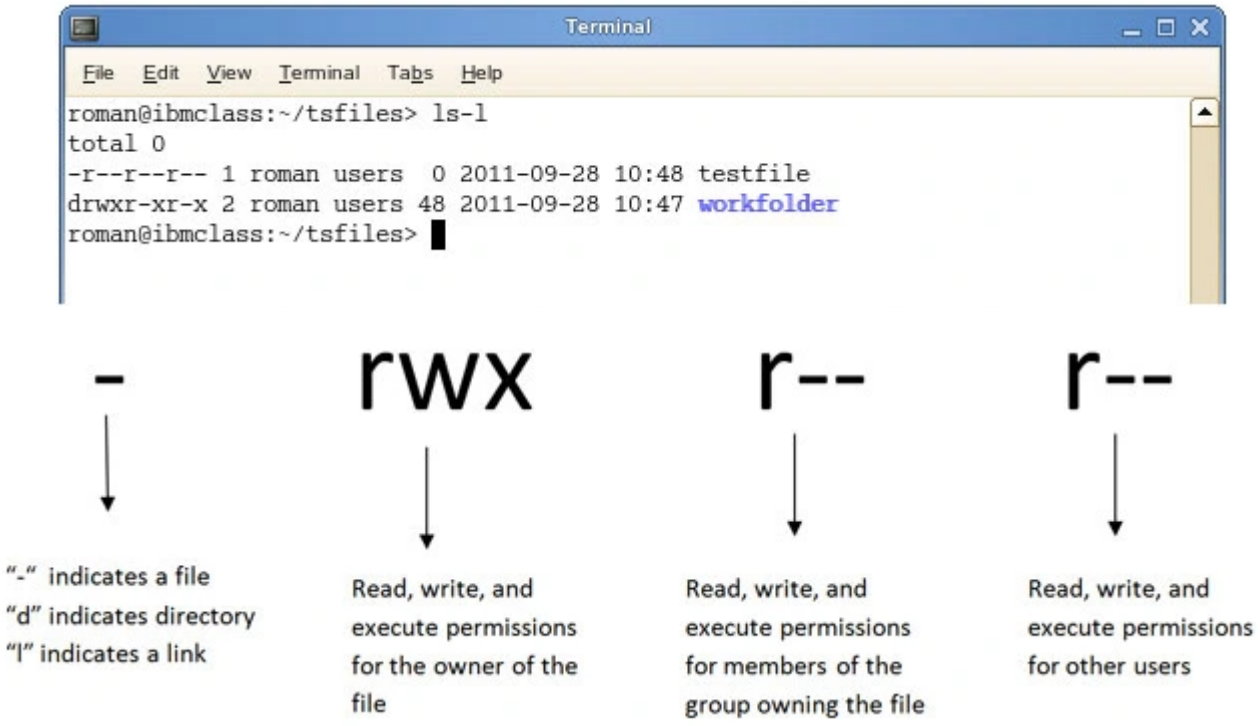
How do I change directory permissions in Linux?chmod(change mode)

To change directory permissions in Linux, use the following:

- **chmod +rwx** filename to add permissions
- **chmod -rwx** directoryname to remove permissions.
- **chmod +x** filename to allow executable permissions.
- **chmod -wx** filename to take out write and executable permissions.

Note that “r” is for read, “w” is for write, and “x” is for execute.

This only changes the permissions for the owner of the file.



What are the three permission groups?

There are three options for permission groups available to you in Linux. These are

- **owners**: these permissions will only apply to owners and will not affect other groups.
- **groups**: you can assign a group of users specific permissions, which will only impact users within the group.
- **all users**: these permissions will apply to all users, and as a result, they present the greatest security risk and should be assigned with caution.

What are the three kinds of file permissions in Linux?

There are three kinds of file permissions in Linux:

- **Read (r)**: Allows a user or group to view a file.
- **Write (w)**: Permits the user to write or modify a file or directory.
- **Execute (x)**: A user or grup with execute permissions can execute a file or view a directory.

More ways to manage permissions

Here's a more comprehensive list of ways you can manage file permissions, groups, and ownership beyond the basic commands listed at the top of this guide.

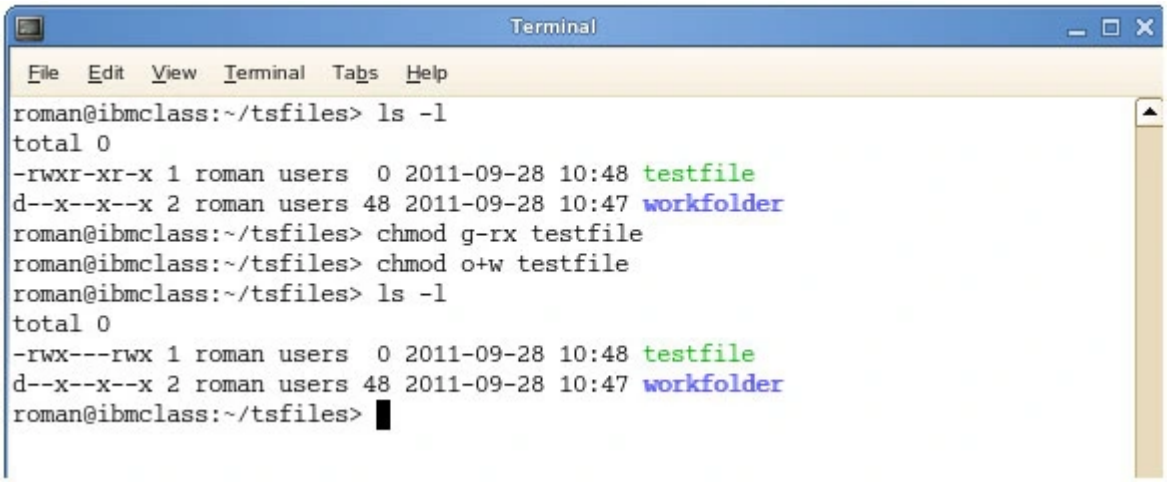
How to Change Directory Permissions in Linux for the Group Owners and Others

The command for changing directory permissions for group owners is similar, but add a “g” for group or “o” for users:

- **chmod g+w** filename
- **chmod g-wx** filename
- **chmod o+w** filename
- **chmod o-rwx** foldername

To change directory permissions for everyone, use **“u” for users**, “**g” for group**, **“o” for others**, and “ugo” or “a” (for all).

- **chmod ugo+rwx foldername** to give read, write, and execute to everyone.
- **chmod a=r foldername** to give only read permission for everyone.



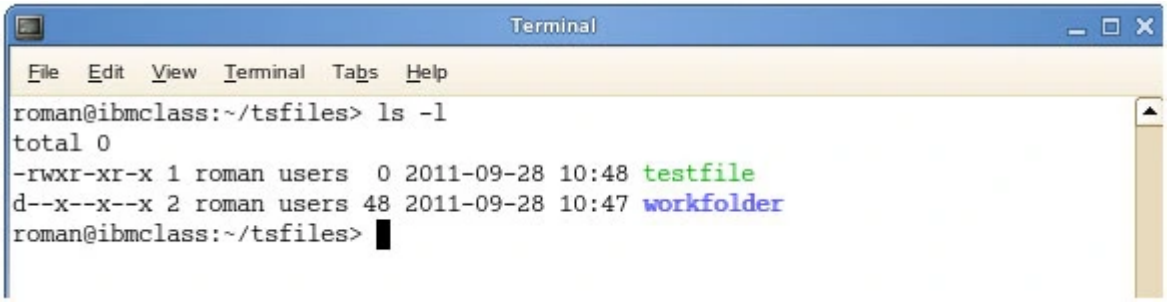
How to Change Groups of Files and Directories in Linux

By issuing these commands, you can change groups of files and directories in Linux.

- **chgrp groupname filename**
- **chgrp groupname foldername**

chgrp= change group

Note that the group must exit before you can assign groups to files and directories.

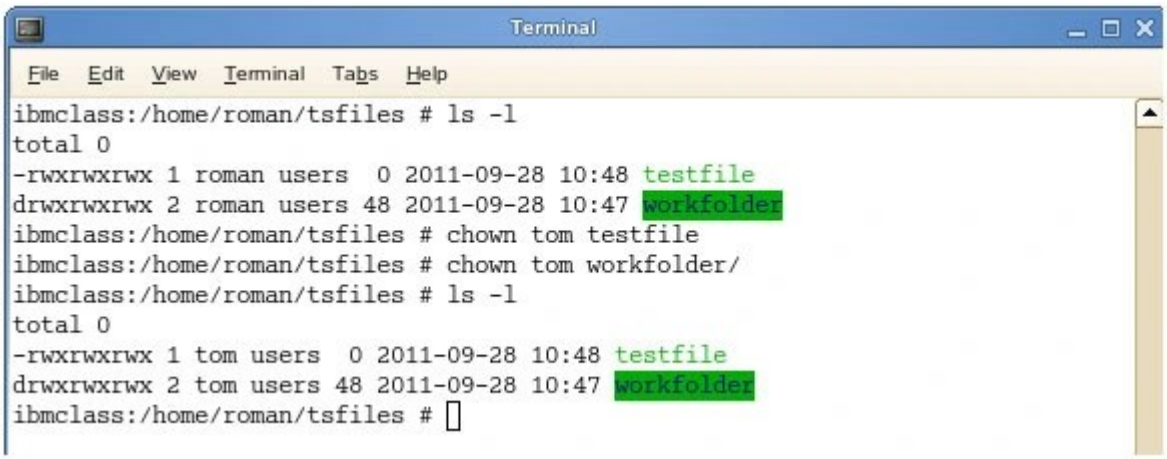


Changing ownership in Linux

Another helpful command is changing ownerships of files and directories in Linux:

- **chown name filename**
- **chown name foldername**

chown= change owner



These commands will give ownership to someone, but all sub files and directories still belong to the original owner.

You can also combine the group and ownership command by using:

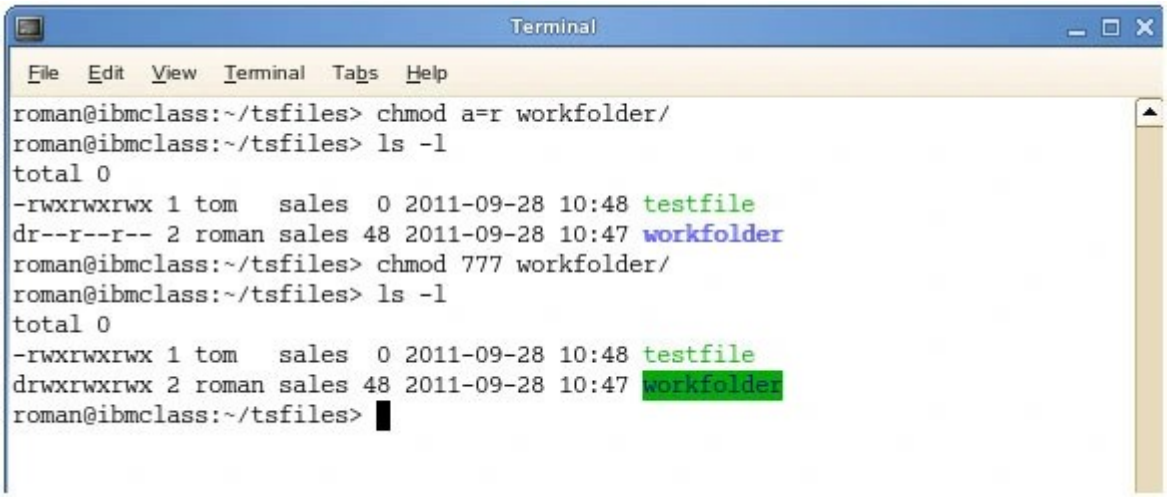
- **chown -R name:filename /home/name/directoryname**

Changing Linux permissions in numeric code

You may need to know how to change permissions in numeric code in Linux, so to do this you use numbers instead of “r”, “w”, or “x”.

- 0 = No Permission
- 1 = Execute
- 2 = Write
- 4 = Read

Basically, you add up the numbers depending on the level of permission you want to give.



Permission numbers are:

- 0 = ---
- 1 = --x
- 2 = -w-

- 3 = -wx
- 4 = r-
- 5 = r-x
- 6 = rw-
- 7 = rwx

For example:

- **chmod 777 foldername** will give read, write, and execute permissions for everyone.
- **chmod 700 foldername** will give read, write, and execute permissions for the user only.
- **chmod 327 foldername** will give write and execute (3) permission for the user, w (2) for the group, and read, write, and execute for the users.

As you can see, there are several options when it comes to permissions. You have the capability to dictate usability among users. While it may be easier to just give all permission to everyone, it may end up biting you in the end. So choose wisely.

Get hep for a command:

```
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ chmod --help
Usage: chmod [OPTION]... MODE[,MODE]... FILE...
  or:  chmod [OPTION]... OCTAL-MODE FILE...
  or:  chmod [OPTION]... --reference=RFILE FILE...
Change the mode of each FILE to MODE.
With --reference, change the mode of each FILE to that of RFILE.

  -c, --changes          like verbose but report only when a change is made
  -f, --silent, --quiet  suppress most error messages
  -v, --verbose          output a diagnostic for every file processed
      --no-preserve-root  do not treat '/' specially (the default)
      --preserve-root    fail to operate recursively on '/'
      --reference=RFILE  use RFILE's mode instead of MODE values
  -R, --recursive        change files and directories recursively
      --help     display this help and exit
      --version  output version information and exit

Each MODE is of the form '[ugoa]*([-+=]([rwxXst]*|[ugo]))+|[-+=][0-7]+'.
```

GNU coreutils online help: <https://www.gnu.org/software/coreutils/>
Full documentation <https://www.gnu.org/software/coreutils/chmod>
or available locally via: info '(coreutils) chmod invocation'
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$

Store or Append output of a command in a file:

Store : command >filename.txt

Append: (contents of this file will remain.At the end of the contends new info will append)

Command >>filename.txt

```
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ ls -l>l.txt
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ cat l.txt
total 40
-rwxrwxr-x 1 alviubuntu alviubuntu 17736 অক্টোবর   1 22:37 2
-rwxrwxrwx 1 alviubuntu alviubuntu   373 অক্টোবর   1 22:30 2.cc
-rwxrwxrwx 1 alviubuntu alviubuntu   104 অক্টোবর   1 09:44 bach_file.sh
-rwxrwxrwx 1 alviubuntu alviubuntu    46 অক্টোবর   1 10:30 bash_practice.sh
-rw-rw-r-- 1 alviubuntu alviubuntu     0 অক্টোবর   1 23:34 l.txt
drwxrwxr-x 2 alviubuntu alviubuntu  4096 সেপ্টেম্বর 18 11:03 oslab_cse19A
-rw-rw-r-- 1 alviubuntu alviubuntu     7 অক্টোবর   1 08:47 outterminal.txt
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ pwd>>l.txt
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ cat l.txt
total 40
-rwxrwxr-x 1 alviubuntu alviubuntu 17736 অক্টোবর   1 22:37 2
-rwxrwxrwx 1 alviubuntu alviubuntu   373 অক্টোবর   1 22:30 2.cc
-rwxrwxrwx 1 alviubuntu alviubuntu   104 অক্টোবর   1 09:44 bach_file.sh
-rwxrwxrwx 1 alviubuntu alviubuntu    46 অক্টোবর   1 10:30 bash_practice.sh
-rw-rw-r-- 1 alviubuntu alviubuntu     0 অক্টোবর   1 23:34 l.txt
drwxrwxr-x 2 alviubuntu alviubuntu  4096 সেপ্টেম্বর 18 11:03 oslab_cse19A
-rw-rw-r-- 1 alviubuntu alviubuntu     7 অক্টোবর   1 08:47 outterminal.txt
/home/alviubuntu/Documents/Lab3200
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ rm l.txt
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$
```

Resource Link: https://learnxinyminutes.com/docs/bash/

Bash Commands:

```
#!/usr/bin/env bash

# First line of the script is the shebang which tells the
system how to execute
```

```
# the script:
https://en.wikipedia.org/wiki/Shebang_(Unix)

# As you already figured, comments start with #. Shebang
is also a comment.


# Simple hello world example:

echo "Hello world!" # => Hello world!


# Each command starts on a new line, or after a
semicolon:

echo "This is the first command"; echo "This is the
second command"

# => This is the first command

# => This is the second command


# Declaring a variable looks like this:

variable="Some string"


# But not like this:

variable = "Some string" # => returns error "variable:
command not found"

# Bash will decide that `variable` is a command it must
execute and give an error

# because it can't be found.


# Nor like this:

variable= "Some string" # => returns error: "Some string:
command not found"

# Bash will decide that "Some string" is a command it
must execute and give an
```

```
# error because it can't be found. In this case the
"variable=" part is seen

# as a variable assignment valid only for the scope of
the "Some string"

# command.


# Using the variable:

echo "$variable" # => Some string

echo '$variable' # => $variable

# When you use a variable itself — assign it, export it,
or else — you write

# its name without $. If you want to use the variable's
value, you should use $.

# Note that ' (single quote) won't expand the variables!

# You can write variable without surrounding quotes but
it's not recommended.


# Parameter expansion ${...}:

echo "${variable}" # => Some string

# This is a simple usage of parameter expansion such as
two examples above.

# Parameter expansion gets a value from a variable.

# It "expands" or prints the value.

# During the expansion time the value or parameter can be
modified.

# Below are other modifications that add onto this
expansion.


# String substitution in variables:

echo "${variable/Some/A}" # => A string
```

```
# This will substitute the first occurrence of "Some"
with "A".


# Substring from a variable:

length=7

echo "${variable:0:length}" # => Some st

# This will return only the first 7 characters of the
value

echo "${variable: -5}" # => tring

# This will return the last 5 characters (note the space
before -5).

# The space before minus is mandatory here.


# String length:

echo "${#variable}" # => 11


# Indirect expansion:

other_variable="variable"

echo ${!other_variable} # => Some string

# This will expand the value of `other_variable`.


# The default value for variable:

echo "${foo:-"DefaultValueIfFooIsMissingOrEmpty"}"

# => DefaultValueIfFooIsMissingOrEmpty

# This works for null (foo=) and empty string (foo="");
zero (foo=0) returns 0.

# Note that it only returns default value and doesn't
change variable value.
```

```
# Declare an array with 6 elements:
array=(one two three four five six)
# Print the first element:
echo "${array[0]}" # => "one"
# Print all elements:
echo "${array[@]}" # => "one two three four five six"
# Print the number of elements:
echo "${#array[@]}" # => "6"
# Print the number of characters in third element
echo "${#array[2]}" # => "5"
# Print 2 elements starting from fourth:
echo "${array[@]:3:2}" # => "four five"
# Print all elements each of them on new line.
for item in "${array[@]}"; do
    echo "$item"
done

# Built-in variables:
# There are some useful built-in variables, like:
echo "Last program's return value: $?"
echo "Script's PID: $$"
echo "Number of arguments passed to script: $#"
echo "All arguments passed to script: $@"
echo "Script's arguments separated into different
variables: $1 $2..."
```

```
# Brace Expansion {...}

# used to generate arbitrary strings:

echo {1..10} # => 1 2 3 4 5 6 7 8 9 10

echo {a..z} # => a b c d e f g h i j k l m n o p q r s t
u v w x y z

# This will output the range from the start value to the
end value.

# Note that you can't use variables here:

from=1

to=10

echo {$from..$to} # => {$from..$to}


# Now that we know how to echo and use variables,

# let's learn some of the other basics of Bash!


# Our current directory is available through the command
`pwd`.

# `pwd` stands for "print working directory".

# We can also use the built-in variable `$PWD`.

# Observe that the following are equivalent:

echo "I'm in $(pwd)" # execs `pwd` and interpolates
output

echo "I'm in $PWD" # interpolates the variable


# If you get too much output in your terminal, or from a
script, the command

# `clear` clears your screen:

clear
```

```
# Ctrl-L also works for clearing output.


# Reading a value from input:
echo "What's your name?"
read name
# Note that we didn't need to declare a new variable.
echo "Hello, $name!"


# We have the usual if structure.
# Condition is true if the value of $name is not equal to
the current user's login username:
if [[ "$name" != "$USER" ]]; then
    echo "Your name isn't your username"
else
    echo "Your name is your username"
fi


# To use && and || with if statements, you need multiple
pairs of square brackets:
read age
if [[ "$name" == "Steve" ]] && [[ "$age" -eq 15 ]]; then
    echo "This will run if $name is Steve AND $age is
15."
fi


if [[ "$name" == "Daniya" ]] || [[ "$name" == "Zach" ]];
then
    echo "This will run if $name is Daniya OR Zach."
```

```
fi

# There are other comparison operators for numbers listed below:

# -ne - not equal

# -lt - less than

# -gt - greater than

# -le - less than or equal to

# -ge - greater than or equal to


# There is also the `=~` operator, which tests a string against the Regex pattern:

email=me@example.com

if [[ "$email" =~ [a-z]+@[a-z]{2,}\.(com|net|org) ]]

then

    echo "Valid email!"

fi


# There is also conditional execution

echo "Always executed" || echo "Only executed if first command fails"

# => Always executed

echo "Always executed" && echo "Only executed if first command does NOT fail"

# => Always executed

# => Only executed if first command does NOT fail


# A single ampersand & after a command runs it in the background. A background command's
```

```
# output is printed to the terminal, but it cannot read
from the input.

sleep 30 &

# List background jobs

jobs # => [1]+  Running                 sleep 30 &

# Bring the background job to the foreground

fg

# Ctrl-C to kill the process, or Ctrl-Z to pause it

# Resume a background process after it has been paused
with Ctrl-Z

bg

# Kill job number 2

kill %2

# %1, %2, etc. can be used for fg and bg as well


# Redefine command `ping` as alias to send only 5 packets

alias ping='ping -c 5'

# Escape the alias and use command with this name instead

\ping 192.168.1.1

# Print all aliases

alias -p


# Expressions are denoted with the following format:

echo $(( 10 + 5 )) # => 15


# Unlike other programming languages, bash is a shell so
it works in the context
```

```
# of a current directory. You can list files and
directories in the current

# directory with the ls command:

ls # Lists the files and subdirectories contained in the
current directory


# This command has options that control its execution:

ls -l # Lists every file and directory on a separate line

ls -t # Sorts the directory contents by last-modified
date (descending)

ls -R # Recursively `ls` this directory and all of its
subdirectories


# Results (stdout) of the previous command can be passed
as input (stdin) to the next command

# using a pipe |. Commands chained in this way are called
a "pipeline", and are run concurrently.

# The `grep` command filters the input with provided
patterns.

# That's how we can list .txt files in the current
directory:

ls -l | grep "\.txt"


# Use `cat` to print files to stdout:

cat file.txt


# We can also read the file using `cat`:

Contents=$(cat file.txt)

# "\n" prints a new line character
```

```
# "-e" to interpret the newline escape characters as
escape characters
echo -e "START OF FILE\n$Contents\nEND OF FILE"
# => START OF FILE
# => [contents of file.txt]
# => END OF FILE


# Use `cp` to copy files or directories from one place to
another.
# `cp` creates NEW versions of the sources,
# so editing the copy won't affect the original (and vice
versa).
# Note that it will overwrite the destination if it
already exists.
cp srcFile.txt clone.txt
cp -r srcDirectory/ dst/ # recursively copy


# Look into `scp` or `sftp` if you plan on exchanging
files between computers.
# `scp` behaves very similarly to `cp`.
# `sftp` is more interactive.


# Use `mv` to move files or directories from one place to
another.
# `mv` is similar to `cp`, but it deletes the source.
# `mv` is also useful for renaming files!
mv s0urc3.txt dst.txt # sorry, l33t hackers...
```

```
# Since bash works in the context of a current directory,
you might want to

# run your command in some other directory. We have cd
for changing location:

cd ~    # change to home directory

cd      # also goes to home directory

cd ..   # go up one directory

        # (^^say, from /home/username/Downloads to
/home/username)

cd /home/username/Documents   # change to specified
directory

cd ~/Documents/..    # now in home directory (if
~/Documents exists)

cd -    # change to last directory

# => /home/username/Documents


# Use subshells to work across directories

(echo "First, I'm here: $PWD") && (cd someDir; echo
"Then, I'm here: $PWD")

pwd # still in first directory


# Use `mkdir` to create new directories.

mkdir myNewDir

# The `-p` flag causes new intermediate directories to be
created as necessary.

mkdir -p myNewDir/with/intermediate/directories

# if the intermediate directories didn't already exist,
running the above

# command without the `-p` flag would return an error
```

```
# You can redirect command input and output (stdin,
stdout, and stderr)

# using "redirection operators". Unlike a pipe, which
passes output to a command,

# a redirection operator has a command's input come from
a file or stream, or

# sends its output to a file or stream.


# Read from stdin until ^EOF$ and overwrite hello.py with
the lines

# between "EOF" (which are called a "here document"):

cat > hello.py << EOF

#!/usr/bin/env python

from __future__ import print_function

import sys

print("#stdout", file=sys.stdout)

print("#stderr", file=sys.stderr)

for line in sys.stdin:

    print(line, file=sys.stdout)

EOF

# Variables will be expanded if the first "EOF" is not
quoted


# Run the hello.py Python script with various stdin,
stdout, and

# stderr redirections:

python hello.py < "input.in" # pass input.in as input to
the script
```

```
python hello.py > "output.out" # redirect output from the
script to output.out


python hello.py 2> "error.err" # redirect error output to
error.err


python hello.py > "output-and-error.log" 2>&1

# redirect both output and errors to output-and-error.log

# &1 means file descriptor 1 (stdout), so 2>&1 redirects
stderr (2) to the current

# destination of stdout (1), which has been redirected to
output-and-error.log.


python hello.py > /dev/null 2>&1

# redirect all output and errors to the black hole,
/dev/null, i.e., no output


# The output error will overwrite the file if it exists,

# if you want to append instead, use ">>":

python hello.py >> "output.out" 2>> "error.err"


# Overwrite output.out, append to error.err, and count
lines:

info bash 'Basic Shell Features' 'Redirections' >
output.out 2>> error.err

wc -l output.out error.err


# Run a command and print its file descriptor (e.g.
/dev/fd/123)

# see: man fd
```

```
echo <(echo "#helloworld")


# Overwrite output.out with "#helloworld":
cat > output.out <(echo "#helloworld")
echo "#helloworld" > output.out
echo "#helloworld" | cat > output.out
echo "#helloworld" | tee output.out >/dev/null


# Cleanup temporary files verbosely (add '-i' for
interactive)
# WARNING: `rm` commands cannot be undone
rm -v output.out error.err output-and-error.log
rm -r tempDir/ # recursively delete
# You can install the `trash-cli` Python package to have
`trash`
# which puts files in the system trash and doesn't delete
them directly
# see https://pypi.org/project/trash-cli/ if you want to
be careful


# Commands can be substituted within other commands using
$( ):
# The following command displays the number of files and
directories in the
# current directory.
echo "There are $(ls | wc -l) items here."


# The same can be done using backticks `` but they can't
be nested -
```

```
# the preferred way is to use $( ).
echo "There are `ls | wc -l` items here."

# Bash uses a `case` statement that works similarly to
switch in Java and C++:
case "$Variable" in
    # List patterns for the conditions you want to meet
    0) echo "There is a zero.";;
    1) echo "There is a one.";;
    *) echo "It is not null.";;  # match everything
esac

# `for` loops iterate for as many arguments given:
# The contents of $Variable is printed three times.
for Variable in {1..3}
do
    echo "$Variable"
done
# => 1
# => 2
# => 3


# Or write it the "traditional for loop" way:
for ((a=1; a <= 3; a++))
do
    echo $a
```

```
done
# => 1
# => 2
# => 3

# They can also be used to act on files..
# This will run the command `cat` on file1 and file2
for Variable in file1 file2
do
    cat "$Variable"
done

# ..or the output from a command
# This will `cat` the output from `ls`.
for Output in $(ls)
do
    cat "$Output"
done

# Bash can also accept patterns, like this to `cat`
# all the Markdown files in current directory
for Output in ./*.markdown
do
    cat "$Output"
done
```

```
# while loop:
while [ true ]
do
    echo "loop body here..."
    break
done
# => loop body here...

# You can also define functions
# Definition:
function foo ()
{
    echo "Arguments work just like script arguments: $@"
    echo "And: $1 $2..."
    echo "This is a function"
    returnValue=0    # Variable values can be returned
    return $returnValue
}
# Call the function `foo` with two arguments, arg1 and
arg2:
foo arg1 arg2
# => Arguments work just like script arguments: arg1 arg2
# => And: arg1 arg2...
# => This is a function
# Return values can be obtained with $?
resultValue=$?
```

```
# More than 9 arguments are also possible by using
braces, e.g. ${10}, ${11}, ...


# or simply
bar ()
{
    echo "Another way to declare functions!"
    return 0
}
# Call the function `bar` with no arguments:
bar # => Another way to declare functions!

# Calling your function
foo "My name is" $Name

# There are a lot of useful commands you should learn:
# prints last 10 lines of file.txt
tail -n 10 file.txt

# prints first 10 lines of file.txt
head -n 10 file.txt

# print file.txt's lines in sorted order
sort file.txt

# report or omit repeated lines, with -d it reports them
uniq -d file.txt
```

```
# prints only the first column before the ',' character
cut -d ',' -f 1 file.txt


# replaces every occurrence of 'okay' with 'great' in file.txt
# (regex compatible)
sed -i 's/okay/great/g' file.txt
# be aware that this -i flag means that file.txt will be changed
# -i or --in-place erase the input file (use --in-place=.backup to keep a back-up)


# print to stdout all lines of file.txt which match some regex
# The example prints lines which begin with "foo" and end in "bar"
grep "^foo.*bar$" file.txt


# pass the option "-c" to instead print the number of lines matching the regex
grep -c "^foo.*bar$" file.txt


# Other useful options are:
grep -r "^foo.*bar$" someDir/ # recursively `grep`
grep -n "^foo.*bar$" file.txt # give line numbers
grep -rI "^foo.*bar$" someDir/ # recursively `grep`, but ignore binary files
```

```
# perform the same initial search, but filter out the
lines containing "baz"

grep "^foo.*bar$" file.txt | grep -v "baz"


# if you literally want to search for the string,

# and not the regex, use `fgrep` (or `grep -F`)

fgrep "foobar" file.txt


# The `trap` command allows you to execute a command
whenever your script

# receives a signal. Here, `trap` will execute `rm` if it
receives any of the

# three listed signals.

trap "rm $TEMP_FILE; exit" SIGHUP SIGINT SIGTERM


# `sudo` is used to perform commands as the superuser

# usually it will ask interactively the password of
superuser

NAME1=$(whoami)

NAME2=$(sudo whoami)

echo "Was $NAME1, then became more powerful $NAME2"


# Read Bash shell built-ins documentation with the bash
`help` built-in:

help

help help

help for

help return
```

```
help source
help .

# Read Bash manpage documentation with `man`
apropos bash
man 1 bash
man bash

# Read info documentation with `info` (`?` for help)
apropos info | grep '^info.*('
man info
info info
info 5 info

# Read bash info documentation:
info bash
info bash 'Bash Features'
info bash 6
info --apropos bash
```

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ **touch bash_practice.sh**

alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ nano bash_practice.sh

```
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ bash bash_practice.sh
This is a bash variable\n
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$ cat bash_practice.sh
var1="This is a bash variable"
echo "$var1\n"
alviubuntu@alviubuntu-virtual-machine:~/Documents/Lab3200$
```