Introduction to Linux

Every time you are in front of a computer, what let you to communicate with the computer is the operational system or the Kernel. Between the different OS, we are going to discuss the case of Linux. With the versions we have now, this is becoming as friendly as Windows or MacOs. The idea is that the user has access to the Kernel and it can modify the properties. There are many OS in the Linux family, Ubuntu, Fedora, Mint, etc.

Why is this so popular?

It is free and it is open source. You can change anything and it has a whole farm of applications. You do not need an antivirus (very secure!). It can work for years without a reboot!.

It is now more user friendly, maybe not as Windows but not so far.

It is not so difficult to install and there are many manuals on the Web. Here I will start after installation.

Files are ordered in a tree, under the root directory (start of the file system): "/" and it branches out. It is like a warehouse of files and in comparison with windows, files are stored in different directories.

This is a very hierarchical system (no drives like in windows).

In Linux everything is a file (directory, file, printers, usb memory, removables disks, etc), programs are stored in the directory /bin, etc. All files have premissions that allow the use to read, write or execute the corresponding file. Access restrictions can be created to allow users to have different access.

Users in linux can be classified as:

regular user (standard), let say the username of that user is john, then the is a directory called

/home/john

As a user you do not have access to directories of other users.

this is the usual default directory for any user when they login. There are computers (large computers) which have /work directory for each user and they allow users to have more memory and hard drive but they are not usually backed up.

root user: There is in user, also call root, which has access to everything and it has administrative privileges (can install and delete, etc). When you install linux, you always create a user and a root (or superuser) accounts.

service user: it is created to offer services into your machine, this will increase the security of your machine (it is not present in all linux versions).

LINUX FILES NAMES are CASE SENSITIVE (File is not the same than file)

Terminal vs File Manager

The most frequent tasks are deleting, creating and moving files. How do we do that efficiently?

There are two ways.

Command Line interface (CLI) within the Terminal. It is quite flexible and it has more possibilities. For example to create a new use takes one line while in the GUI.

Does not consume RAM as GUI and it is very fast compare to GUI, for example transferring thousand of files from one side to another.

Graphical Unit Interface (GUI). GUI is the most used. But due to the form we are going to interact with the computer, we will use CLI, you can learn the GUI method on your spare time.

CLI

Let us start opening a terminal. Search the terminal command in dash or open terminal from the submenu or use

```
<Contrl>-<Alt>-<T>
```

You will see a line in the terminal as

```
[alromero@srih0001 ~]$
```

Name of the user and name of the host name. The "~" means that you are in the home directory and "$" means that you are working as a regular user. If you see "#", it means that you are as a superuser.

Now the first thing to know is the working directory. This is where you are, if you want to know, type

```
pwd
```

You should get a like as:

```
/users/alromero  or /home/alromero
```

You can move between directories using the command "cd" which stands by "Change Directory". Follow this example.

```
cd /tmp
pwd
cd
cd /
pwd
cd ~
```

Navigating through multiple directories is very easy, for example (when it exist)

```
cd /tmp/aldo/newdirectory
```

if you want to move up one directory

```
cd ..
```

if you want to move two directories

```
cd ../../
```

Now, every file has a path, which indicates where it is located in the tree, for example

```
/scratch/alromero/KTO-KNF/POSCAR
```

In this case, we have the "absolute path". We can also use the the "relative path", which is the one based on the one you are in. Suppose you are in the directory /scratch/alromero/KTO-KNF and you want to go to another directory inside this directory

```
cd cRPA
if you now type pwd, you will see
/scratch/alromero/KTO-KNF/cRPA
```

Before we go on with the tutorial, in you want to get deeper (which will HELP YOU A LOT), you can take a look some tutorials on the web. some are

[https://www.tutorialspoint.com/unix/unix-vi-editor.htm]

[https://ubuntu.com/tutorials/command-line-for-beginners#1-overview]

Linux Commands

Here we will see different handy commands used in linux.

List directories and files

```
ls
```

```
ls -la
```

```
ls -lt
```

```
ls -l *png
```

```
man ls
```

For example, here we will recursively the files, directories and the content of those directories.

```
ls -R
```

If you want to see all information from all files in a given directory

```
ls -la
```

Example of the output:

```
[alromero@srih0001 VW_12_Noncoll_Sp_3]$ ls -la
total 32800
drwxrwxr-x  2 alromero alromero    32768 Aug 17 09:38 .
drwxrwxr-x 45 alromero alromero    32768 Aug 17 09:37 ..
-rw-rw-r--  1 alromero alromero 26020748 Aug 17 09:38 CHG
-rw-rw-r--  1 alromero alromero        0 Aug 17 09:38 CHGCAR
-rw-rw-r--  1 alromero alromero      167 Aug 17 09:38 DOSCAR
-rw-rw-r--  1 alromero alromero      193 Aug 17 09:38 EIGENVAL
```

Let see the details of this info, let me take one of them

```
-rw-rw-r--  1 alromero alromero        0 Aug 17 09:38 CHGCAR
```

-rw-rw-r-- it is the file type and access permissions

1 it is the memory blocks occupied by the fime

alromero the owner of the creator of the file

alromero the group that identifies the user

32768 file size in bytes

Aug 17 09:3 date when it is created

CHG directory or file name. If the file starts with "." they are called hidden files and you do not usually see them, unless you ask for them with the command "ls -a".

For example

```
[alromero@srih0001 ~]$ ls
ABINIT              CODES                 important_settings         MHM_over_files_Verb3

[alromero@srih0001 ~]$ ls -a
.                   .bash_history  .config               .gnome2
.matplotlib           .mypython   ABINIT              CODES
important_settings         MHM_over_files_Verb3
```

For creating and view commands, we can use the command "cat" used to display text files or can be used to copy or combine files. Let see how it works

```
ls AB*
 ---- output  --- >   ABIPY
 cat ABIPY

 [alromero@srih0001 ~]$ cat ABIPY
source /shared/software/conda/conda_init.sh
conda activate abienv_py37
```

Now let us create a file and after you are done, type control-d

```
[alromero@srih0001 ~]$ cat > file1
this is my class
I want to learn computational physics
<control>-d
```

Now we can see what is the content of the file by

```
cat file1
```

Now let say we want to combine two files by "piping the result"

```
cat file1 file2 > file3
```

How to delete files?

BECAREFUL!!! remove the file without confirmation

```
rm <filename>
```

if you want to be asked use

```
rm -i <filename>
```

This can be modified once and for all in your .bashrc, later!

We can only delete files that belong to the user!. We can not delete files that do not belong to us.

We can also move files around

```
mv file1 <newlocation>
```

Again, this can only happen if we have the right permissions, to the file and to the directory. If you have installed Linux in your machine, you can use the "su" command or "sudo". In these devices you do not have access to this command.

Let see everything in action

```
[alromero@srih0001 ~]$ cat > file
this is a test.
[alromero@srih0001 ~]$ cat file
this is a test.
[alromero@srih0001 ~]$ ls f*
file
[alromero@srih0001 ~]$ mv file file1
[alromero@srih0001 ~]$ ls f*
file1
```

Manipulating Directories

We can create new directories by

```
mkdir newfiles
```

let see how it works in

```
[alromero@srih0001 ~]$ ls -ld ne*
ls: cannot access ne*: No such file or directory
[alromero@srih0001 ~]$ mkdir newfiles
[alromero@srih0001 ~]$ ls -ld ne*
drwxrwxr-x 2 alromero alromero 512 Aug 17 12:58 newfiles
[alromero@srih0001 ~]$ mkdir newfiles1 newfiles2 newfiles3
[alromero@srih0001 ~]$ ls -ld ne*
drwxrwxr-x 2 alromero alromero 512 Aug 17 12:58 newfiles
drwxrwxr-x 2 alromero alromero 512 Aug 17 12:58 newfiles1
drwxrwxr-x 2 alromero alromero 512 Aug 17 12:58 newfiles2
drwxrwxr-x 2 alromero alromero 512 Aug 17 12:58 newfiles3
```

Now let us remove the directories (two things, they have to be empty!!!)

```
rmdir newf*
```

to rename a directory

```
mv direc direc1
```

To get help

To get help on any command

```
man ls
```

also you can get an idea on what you have type by using the history command

```
history
```

in my case, let me see only the last lines

```
 1103  mkdir newfiles
 1104  ls -lt new*
 1105  ls -la new*
 1106  rmdir newfiles
 1107  ls
 1108  ls new*
 1109  mkdir newfiles
 1110  ls ne*
 1111  ls -ld n*
 1112  rmdir newfile
 1113  rmdir newfiles
 1114  ls -ld ne*
 1115  mkdir newfiles
 1116  ls -ld ne*
 1117  mkdir newfiles1 newfiles2 newfiles3
 1118  ls -ld ne*
 1119  rmdir -i newfi*
 1120  rmdir newfi*
 1121  history
```

You can also use the "upward" arrowkey to move backwards on previous commands.

You can also use tricks to repeat a given command, for example using

```
!-3
```

it will repeat the third line command previously used, in the previous example, it will run

```
rmdir -i newfi*
```

Now you can clean what you have on your window by using

```
clear
```

A handy set of commands is the copy and paste, as you can copy a command and paste it in your command line.

This is done by using "Cntrl-C" and to paste it in the terminal, you need to use "Control-Shift-v" (all together)

File permissions

Linux have two different authorization levels, ownership and permission. Let see what they are:

Ownership. Each file in linux is assigned three types of ownership

User: it is the owner of the file

Group: all users contained in the same group they will have access to a file.

Other: any other user that can access a file (the world!)

Now, each one of these could have 3 different type of permissions: read (to open and see the content), write (modify or remove the content of the file or directory) and execute (allow to execute, if the file is executable), each for each one of the ownership

how is this defined?

```
-rwxrw-r--  means that everybody can read, only the user and the group can write but only
the user can execute
r: read permission
w: write permission
-: no permission
The first character can be "-" as it is a file or "d" if this is a directory
```

Now, this is the cool part, you can change the permissions of a file.

The following command can be used in two ways, absolute or symbolic. The absolute user numbers are:

--- 0

--x 1

-w- 2

-wx 3

r-- 4

r-x 5

rw- 6

rwx 7

```
chmode <newpermissions> <filename>
chmod 750 file1
```

In the symbolic mode you can change permissions "locally", with the following notation

"+" add permission

"-" remove permission

"=" set the permissions and overrides the permissions set earlier

and the user notations

u: user

g: group

a: everybody

o: other

to use it:

```
[alromero@srih0001 ~]$ cat > test
Nothing.
[alromero@srih0001 ~]$ ls -lt te*
-rw-rw-r-- 1 alromero alromero 9 Aug 17 15:17 test
[alromero@srih0001 ~]$ chmod a+rwx test
[alromero@srih0001 ~]$ ls -lt te*
-rwxrwxrwx 1 alromero alromero 9 Aug 17 15:17 test
[alromero@srih0001 ~]$ chmod g+x test
[alromero@srih0001 ~]$ ls -lt te*
-rwxrwxrwx 1 alromero alromero 9 Aug 17 15:17 test
```

You can also change the ownership or the group by using

```
chown <user> <filename>
chown <user>:<group> <filename>
chgrp <group> <filename>
```

The file /etc/group has all the groups in the system. You can also use the command "groups" to find all groups where you are a member.

The **newgrp** command is used to change the current group ID during a login session

Redirection in Linux

Most of the things we have discussed are based on input and output. You use a command and the output (for now) reports on screen. Is it possible to pipe the results into something else or read the input from something else?

Easy, use the symbols ">" for output and "<" for input

For example

```
[romeroa@bridges2-login014 Aldo]$ ls > output
[romeroa@bridges2-login014 Aldo]$ cat output
2D_Vasp
Abinit_NbS2_Cr
DISP-0011
InGaS3
LiFeNiO2F2
LiPt2B2
LSMO
MagneticField
Mg_Sobhit
NbSe2
NiW2B2
VDW_VDWFUNC_SCAN_PHONONS_ONEbyONE
[romeroa@bridges2-login014 Aldo]$
```

BUT BECAREFUL, if the output file name already exist in your directory, it will be overwritten!!

For example

```
[romeroa@bridges2-login014 Aldo]$ echo ALDO > output
[romeroa@bridges2-login014 Aldo]$ cat output
ALDO
```

Now, what if you want to add something to the file, you can

```
[romeroa@bridges2-login014 OPTIMIZE_1]$ ls *out> file2
[romeroa@bridges2-login014 OPTIMIZE_1]$ cat file1 file2 > file3
[romeroa@bridges2-login014 OPTIMIZE_1]$ cat file1
```

```
abilog
abinit.11636047.err
abinit.11636047.out
[romeroa@bridges2-login014 OPTIMIZE_1]$ cat file2
abinit.11636047.out
mgb2.out
[romeroa@bridges2-login014 OPTIMIZE_1]$ cat file3
abilog
abinit.11636047.err
abinit.11636047.out
abinit.11636047.out
mgb2.out
[romeroa@bridges2-login014 OPTIMIZE_1]$ cat file1 >> file2
[romeroa@bridges2-login014 OPTIMIZE_1]$ cat file2
abinit.11636047.out
mgb2.out
abilog
abinit.11636047.err
abinit.11636047.out
```

The new content has been added to the file!

File Descriptors

All things in LInux are files, devices (screen, usb, etc), directories and files. Each file has an FD. Therefore, everytime you execute a command at the terminal, 3 files are always open

FD0: Standard Input

FD1: Standard Output

FD2: Standard Error

By default, error is shown at the screen, but sometimes you would like to redirect the error. For example

```
[romeroa@bridges2-login014 OPTIMIZE_1]$ telnet localhost
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying 127.0.0.1...
telnet: connect to address 127.0.0.1: Connection refused
[romeroa@bridges2-login014 OPTIMIZE_1]$ telnet localhost 2> errorfile
Trying ::1...
Trying 127.0.0.1...
[romeroa@bridges2-login014 OPTIMIZE_1]$ cat errorfile
telnet: connect to address ::1: Connection refused
telnet: connect to address 127.0.0.1: Connection refused
```

Another example. Let say you have millions of files in your directory and you are trying to look for one in particular but you do no know where it is. Here a very good command is "find"

For example

```
find . -name '*.py'
```

Now, you can store errors from the command line by

```
find . -name '*.py' 2> error.file
```

Another example, let say you want to store the output and also the standard output into the same file.

```
ls *py ABC > dirlist 2>&1
```

Here ">&" writes the output of one file into the input of another output. Here it means that the error output is redirected to standard output which in turn is redirected to file dirlist.

Example.

```
[romeroa@bridges2-login014 romeroa]$ ls
aldo  Aldo  Camilo  sobhit
[romeroa@bridges2-login014 romeroa]$ ls Aldo Jim
ls: cannot access 'Jim': No such file or directory
Aldo:
2D_Vasp         InGaS3      LSMO           NbSe2     OPTIMIZE_1
[romeroa@bridges2-login014 romeroa]$ ls Aldo Jim > dirlist
ls: cannot access 'Jim': No such file or directory
[romeroa@bridges2-login014 romeroa]$ ls Aldo Jim > dirlist 2>&1
[romeroa@bridges2-login014 romeroa]$ cat dirlist
ls: cannot access 'Jim': No such file or directory
Aldo:
2D_Vasp
InGaS3
LSMO
NbSe2
OPTIMIZE_1
```

Now you see that in the file dirlist, the standard output and the error output are written.

Pipes or filters in Linux

Here, we can now connect several commands one after the other. Here we can perform very complex tasks

```
cat <bigfile> | less
```

This allows to move one page at the time and scroll with the arrow keys and use q to exist.

Searching many files for information

Here we introduce the command ``grep''. It is used as grep <search_string> file

For example

```
[romeroa@bridges2-login014 DISP-010]$ cat INCAR | grep ENCUT
ENCUT = 500
```

grep is very powerful and it has some options that you can use to focus the search

-v show all lines that do not match the searched string

-c display only the count of matching lines

-n shows the matching line and its number

-i match upper and lower case

-l shows the name of the file with the string

```
[romeroa@bridges2-login014 DISP-010]$ cat INCAR | grep -v ENCUT | sort
BRION = -1
EDIFF = 1.0e-08
IALGO = 38
ISMEAR = 0; SIGMA = 0.01
ISPIN = 2
IVDW = 12
LCHARG = .FALSE.
LORBIT = 11
LREAL = .FALSE.
LWAVE = .FALSE.
MAGMOM = 54*0  108*0   9*3  9*-3
NELMIN    =  8
PREC = Accurate
```

Here we have used sort, this takes each line and sort the output alphabetically. sort also have some options

-r reverse sorting

-n sorts numerically

-f case insentitive sorting

Regular Expressions

Characters that allows to match very complicated expressions

. replaces any character

^ matches start of a string

$ matches end of a string

"*" matches up zero or or more times the preceding character

\ represent special characters

() group regular expressions

? matches up to exactly one character

Example:

```
[romeroa@bridges2-login014 DISP-010]$ cat INCAR | grep L
IALGO = 38
LREAL = .FALSE.
LWAVE = .FALSE.
LCHARG = .FALSE.
NELMIN    =  8
LORBIT = 11
[romeroa@bridges2-login014 DISP-010]$ cat INCAR | grep ^L
LREAL = .FALSE.
LWAVE = .FALSE.
LCHARG = .FALSE.
LORBIT = 11
```

Only lines that start with "L" are reported

Now that we can find strings, we can also use with more properties such as the number of timesa character appears in a string. The expressions used are

{n} Matches the preceding character appearing "n" times exactly

{n,m} Matches the preceding character appearing "n" times exactly but no more than m

{n,} Matches the preceding character only when it appears 'n' or more

Example

```
[romeroa@bridges2-login014 DISP-010]$ cat NEW | grep L
IALGO = 38
LREAL = .FALSE.
LWAVE = .FALSE.
LLWAVE = .FALSE.
LCHARG = .FALSE.
LCHARLG = .FALSE.
NELMIN    =  8
LORBIT = 11
[romeroa@bridges2-login014 DISP-010]$ cat NEW | grep -E L{2}
LLWAVE = .FALSE.
```

Here we are looking for a string which has 2 L character, one after the other.

Now we can introduce, extended regular expressions, which are a combination of different regular expressions. Some are

```
\+  Matches one of more occurrence of the previous expression
\?  Matches zero or one occurrence of the previous character
```

In the previous example, we can to look for lines where character L precedes character R

```
[romeroa@bridges2-login014 DISP-010]$ cat NEW | grep "L\+R"
LREAL = .FALSE.
```

Now, let see a more complicated command by using the brace expansion.

```
[romeroa@bridges2-login014 DISP-010]$ echo {aaa,bbb,ccc,ddd}
aaa bbb ccc ddd
[romeroa@bridges2-login014 DISP-010]$ echo {a..h}
a b c d e f g h
[romeroa@bridges2-login014 DISP-010]$ echo {1..9}
1 2 3 4 5 6 7 8 9
[romeroa@bridges2-login014 DISP-010]$ echo a{1..9}b
a1b a2b a3b a4b a5b a6b a7b a8b a9b
```

Environment Variables

A variable is to store a value and it has a name. When we call the name, we have access to its value.

In particular, there are environment variables. These exist in all operational systems and provides information about the system behavior. In Linux there are several environmental variables, to access one

```
[romeroa@bridges2-login014 DISP-010]$ echo $LANG
en_US.UTF-8
```

Several important variables Ithey are case sensitive!!)

```
[romeroa@bridges2-login014 DISP-010]$ echo $PATH
/jet/home/romeroa/.local/bin:/jet/home/romeroa/bin:/opt/packages/allocations/bin:/usr/loca
l/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/bin:/opt/packages/interact/bin:/opt/puppetlabs/b
in
```

Here are the "directions" where the system look for executables

```
[romeroa@bridges2-login014 DISP-010]$ echo $USER
romeroa
[romeroa@bridges2-login014 DISP-010]$ echo $HOME
/jet/home/romeroa
```

The command "env" gives ALL environmental variables. You can also define your own variable.

```
[romeroa@bridges2-login014 DISP-010]$ myname=aldo
[romeroa@bridges2-login014 DISP-010]$ echo $myname
aldo
```

You can also delete variables

```
[romeroa@bridges2-login014 DISP-010]$ unset myname
[romeroa@bridges2-login014 DISP-010]$ echo $myname
```

Communicating with other devices or Networks

ping : check if connection is heavy or not and checking the network in general

```
(base) MacBook-Pro-103:~ aldoromero$ ping bridges2.psc.edu
PING bridges2-login.ddns.psc.edu (128.182.81.30): 56 data bytes
64 bytes from 128.182.81.30: icmp_seq=0 ttl=55 time=15.339 ms
64 bytes from 128.182.81.30: icmp_seq=1 ttl=55 time=15.815 ms
64 bytes from 128.182.81.30: icmp_seq=2 ttl=55 time=17.122 ms
64 bytes from 128.182.81.30: icmp_seq=3 ttl=55 time=28.299 ms
^C
--- bridges2-login.ddns.psc.edu ping statistics ---
4 packets transmitted, 4 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 15.339/19.144/28.299/5.326 ms
```

There are some others non discussed here ftp and telnet

I focus on ssh, which is the one you use most of the time and it ensures a secure connection to the remote computer.

```
ssh <username>@<ipaddress or hostname>

here you will be asked by the password and after, you will be "physically" connected to
the remote computer.

Use exit to leave the remote computer.
```

Managing Processes

An instance of a program is called a process. Any call then create a process. In Linux, you can have different processes, even from the same program. We can classify the processes as "foreground" and "background". Foreground are those that are interfacing with you, for example Word... but a background one is a program that runs and it does not interact with you (no user input, for example the antivirus).

Starting a program in the foreground, it is easy, you just launch the program you are interested. To run one in the background you can

1. Start the program
2. Control-Z
3. bg

Now the program runs in the background and the terminal is free to be used to something else (important for "long programs"). You can bring it back to the terminal using

```
fg <jobname>
```

You can always check all processes in your system

```
top
```

For example

```
top - 09:02:11 up 152 days, 20:06, 19 users,  load average: 0.10, 0.12, 0.09
Tasks: 627 total,   1 running, 626 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.1%us,  0.2%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  132118188k total, 92999912k used, 39118276k free,  1043004k buffers
Swap: 16777212k total,    28032k used, 16749180k free, 46625628k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 5902 root      20   0 9893m 351m 3372 S  1.3  0.3   1538:40 pbs_server

 6435 root      20   0 5064m 699m 6684 S  1.3  0.5   6427:41 moab
 ...
```

PID is the process number

PR is the priority, 20 is the highest and -20 is the lowest

NI user nice cpu time (or) % CPU time spent on low priority processes

VIRT Virtual Memory

RES Physical memory

SHR Shared memory

S Status R(running), S(Sleeping), D (Uninterruptible sleep), T (traced or stopped), Z(Zombie)

You can also check the system by using

```
ps
```

for example to check all processes running from a user check the status of a single program

```
(materials_discovery) [alromero@srih0001 MHM]$ ps ux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
alromero  2307  0.0  0.0 122132  2116 ?        S    Aug17   0:00 sshd: alromero@pts/7
alromero  2308  0.0  0.0 112716  2088 pts/7    Ss+  Aug17   0:00 -bash
alromero 27807  0.0  0.0 122132  2116 ?        S    Aug16   0:00 sshd: alromero@pts/3
alromero 27808  0.0  0.0 112816  2176 pts/3    Ss   Aug16   0:00 -bash
alromero 50379  0.0  0.0 122132  2116 ?        S    09:47   0:00 sshd: alromero@pts/5
alromero 50380  0.0  0.0 112716  2040 pts/5    Ss+  09:47   0:00 -bash
alromero 50976  0.0  0.0 114488  1300 pts/3    R+   09:55   0:00 ps ux


or

ps 2307
```

```
Note: if we do not knot he PI of a process, you can use
pidof <prgram name>
```

To terminate processes that are running in your Linux system, you should use

```
kill <PID>
```

This command could be useful, as when you are trying to run something into your Linux and it becomes really slow, you can see which of those are taking large chunk of the processor and kill the ones you do not want.

Of course, you can also change the priority of one job, which is call niceness in Linux. The default value of all processes is 0, if yo want to start a process with a given priority, then use

```
nice -n <new_nice_value> process name
or
nice -n <new_nice_value> -p PID

nice_value can go from 19(lower) to -20 (higher)

if you want to change the priority

renice -n <value> -p PID
```

To know the disk space, we can use

```
df

an example

(materials_discovery) [alromero@srih0001 MHM]$ df
Filesystem              1K-blocks         Used    Available Use% Mounted on
/dev/mapper/VolGroup00-lv_root
                         66956936     18610340     44939492  30% /
tmpfs                    66059092            0     66059092   0% /dev/shm
/dev/sda1                 1032088       128028       851632  14% /boot
/dev/mapper/VolGroup00-lv_home
                         10190136        23184      9642664   1% /home
/dev/mapper/VolGroup00-lv_opt
                        113402528     89288788     18347724  83% /opt
/dev/mapper/VolGroup00-lv_usr
                         20511356      8625584     10837196  45% /usr
/dev/mapper/VolGroup00-lv_var
                         72117576     54230952     14217376  80% /var
/dev/mapper/VolGroup00-lv_varlog
```

```
                         56635708     10188436     43564328  19% /var/log
/dev/mapper/VolGroup00-lv_local
                        103081248      6430332     91408036   7% /local
192.168.104.22:/gpfs10/depot
                      936168652800 488213496832 447955155968  53% /depot
192.168.104.22:/gpfs10/depot/rcadmin/admin2
                       10737418240    494853120  10242565120   5% /opt/admin2
gpfs01                797169418240 656635020288 140534397952  83% /gpfs
```

or even better

```
(materials_discovery) [alromero@srih0001 MHM]$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/VolGroup00-lv_root
                       64G   18G   43G  30% /
tmpfs                  63G     0   63G   0% /dev/shm
/dev/sda1            1008M  126M  832M  14% /boot
/dev/mapper/VolGroup00-lv_home
                      9.8G   23M  9.2G   1% /home
/dev/mapper/VolGroup00-lv_opt
                      109G   86G   18G  83% /opt
/dev/mapper/VolGroup00-lv_usr
                       20G  8.3G   11G  45% /usr
/dev/mapper/VolGroup00-lv_var
                       69G   52G   14G  80% /var
/dev/mapper/VolGroup00-lv_varlog
                       55G  9.8G   42G  19% /var/log
/dev/mapper/VolGroup00-lv_local
                       99G  6.2G   88G   7% /local
192.168.104.22:/gpfs10/depot
                      872T  455T  418T  53% /depot
192.168.104.22:/gpfs10/depot/rcadmin/admin2
                       10T  472G  9.6T   5% /opt/admin2
gpfs01                743T  612T  131T  83% /gpfs
```

with respect to the RAM memory

```
(materials_discovery) [alromero@srih0001 MHM]$ free -g
             total       used       free     shared    buffers     cached
Mem:           125         88         37          0          0         44
-/+ buffers/cache:         43         82
Swap:           15          0         15
```

Finally, Editors in Linux. Vi

Now we are going to learn how to work with editors in Linux. While there are many different editors in Linux (edit, pico, nano, vile, etc), vi is one of the most used ones as it exist by default in all linux installations and it works equally in all systems. Recently, many systems are coming with an improved version called "vim".

Vi has two operation modes

Command mode. It uses the cursor to copy, paste text, it also saves the changes, etc

Insert mode. It is used to insert text in the file, while you are in command mode, you can type "i" to go to insert mode. To go back to the command mode, you can use "esc"

To start

```
vi <newfile or existing_file>
```

When you open, you will see a lot ot "tilde" signs, they are unused lines. To insert content to the file, type "i"

Now let me give you some keystrokes used to edit a file when you are in the COMMAND MODE

i goes into insert mode

a goes into insert mode AFTER the cursor (append)

A write at the end of the line

ESC Terminate insert mode

u Undo last change

U Undo all changes in the entire line

o open a new line and goes into insert mode

dd delete lines

3dd delete the next three lines

D delete content after the cursor

c delete contents of line after the cursor and insert new text

dw delete a word

4dw delete the next 4 words

cw change word

x delete character at cursor

r replace character

R overwrites characters from cursor onward

s substitute one character under cursor continue to insert

S substitute entire line and begin to insert at beginning of line

~ change case of individual character

Now to move within the file, you need to be in the command mode.

k cursor up

j cursor down

h cursor left

l cursor right

or arrow keys in your keyboard.

Now to save or close a file, first you have to be in the command line and then

shift+zz save the file and quit

:w save the file but keep it open

:q quit without saving

:wq save the file and quit

Shell Scripting

Operating systems have two primary components, kernel (the core) and the shell (it is the last step between the user and the computer).

This is what the terminal is doing, interfacing with shell to communicate with the computer. Shell take care of the system, such that the user do not cross boundaries.

There are two main shells in linux.

The bourne shell, which uses "$" and it has different versions as POSIX shell (sh), Korn Shell(sh) and Bourne Again Shell (bash). The last one is the most used one.

The C-shell, which uses the prompt "%" and the versions are C-shell (csh)) and Tops C-shell (tcsh)

Now, let us now talk about shell scripting and why this is good to learn.

A script is to write a series of commands to the kernel, such that they are executed, usually repetitive commands that help the user to be quite effective.

How do we do it?

Create a file and terminate the file .sh.. something like my script.sh

Start the script with the line #!/bin/sh <- this line send the code to the interpretator location

Write some commands

Save the file

For example, create the following script

```
#!/bin/sh
ls
```

and after you save it, for example to myscript.sh, you can run it two ways

```
bash myscrip.sh
```

Or you can change the user properties and make it executable and run it as

```
./myscript.sh
```

The syntax in the script we have created follow the notation

```
#   adding this symbol to the first line, says it is a comment
```

You can also define variables, for example

```
#!/bin/sh
variable="My variable"
echo $variable


Produces

My variable
```

Another example

```
#!/bin/sh
echo "What is your name?"
read my_name
echo "whats up $my_name, how are you doing?"

THis will give

What is your name?
aldo romero
whats up aldo romero, how are you doing?
```

Of course, this has been very simple, we can add conditional statements, loops and even we are able to define functions.

More examples:

Arguments passed from the **command line** can be accessed by using `$0`, `$1`, `$2`, ... notation. `$0` is the name of the script itself, `$1` is the first argument, `$2` is the second argument and so on.

```
#!/bin/sh

echo $1
echo $1+$2
# Number of arguments supplied to the script
echo $#
#
echo "Quoted Values: $@"
#$ is the PID
echo $$

# Now let us use FOR loop to go over the input
for TOKEN in $*
do
   echo $TOKEN
done


# Now let us support you would like to store several strings as
#NAME01="a"
#NAME02="b"
#NAME03="c"
#NAME04="d"
#NAME05="e"

# we can store them in a single array
NAME[0]="a"
NAME[1]="b"
NAME[2]="c"
NAME[3]="d"
NAME[4]="e"

echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[1]}"
echo "First Method: ${NAME[*]}"
echo "Second Method: ${NAME[@]}"
```

Running the previous script, we get

```
1
1+2
3
Quoted Values: 1 2 3
59161
1
2
3
First Index: a
Second Index: b
First Method: a b c d e
Second Method: a b c d e
```

Now let see how we can do some math

+ (Addition)	**Adds values on either side of the operator**	**`expr $a + $b` will give 30**
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
* (Multiplication)	Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/ (Division)	Divides left hand operand by right hand operand	`expr $b / $a` will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
= (Assignment)	Assigns right operand in left operand	a = $b would assign value of b into a
== (Equality)	Compares two numbers, if both are same then returns true.	[$a ==$b] would return false.
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[$a!$ =b] would return true.

Examples

```
#!/bin/sh

a=10
b=20
```

```
val=`expr $a + $b`
echo "a + b : $val"

val=`expr $a - $b`
echo "a - b : $val"

val=`expr $a \* $b`
echo "a * b : $val"

val=`expr $b / $a`
echo "b / a : $val"

val=`expr $b % $a`
echo "b % a : $val"

if [ $a == $b ]
then
   echo "a is equal to b"
fi

if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

Also we can use relation between variables

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a - eq$b] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a - ne$b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a - gt$b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a - lt$b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a - ge$b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a - le$b] is true.

Boolean operators

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical **OR**. If one of the operands is true, then the condition becomes true.	[$a - lt20 - o$b -gt 100] is true.
-a	This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false.	[$a - lt20 - a$b -gt 100] is false.

Examples:

```
#!/bin/sh

a=10
b=20

if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi
```

```
if [ $a -lt 100 -a $b -gt 15 ]
then
   echo "$a -lt 100 -a $b -gt 15 : returns true"
else
   echo "$a -lt 100 -a $b -gt 15 : returns false"
fi

if [ $a -lt 100 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi

if [ $a -lt 5 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi
```

Now with strings

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a =$b] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[$a! =$b] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a] is not false.
str	Checks if **str** is not the empty string; if it is empty, then it returns false.	[$a] is not false.

Example:

```
#!/bin/sh

a="abc"
```

```
b="efg"

if [ $a = $b ]
then
   echo "$a = $b : a is equal to b"
else
   echo "$a = $b: a is not equal to b"
fi

if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi

if [ -z $a ]
then
   echo "-z $a : string length is zero"
else
   echo "-z $a : string length is not zero"
fi

if [ -n $a ]
then
   echo "-n $a : string length is not zero"
else
   echo "-n $a : string length is zero"
fi

if [ $a ]
then
   echo "$a : string is not empty"
else
   echo "$a : string is empty"
fi
```

Now we can check things on files,

Operator	Description	Example
-b file	Checks if file is a block special file; if yes, then the condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file; if yes, then the condition becomes true.	[-c $file] is false.
-d file	Checks if file is a directory; if yes, then the condition becomes true.	[-d $file] is not true.
-f file	Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set; if yes, then the condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe; if yes, then the condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true.	[-t $file] is false.
-u file	Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable; if yes, then the condition becomes true.	[-r $file] is true.
-w file	Checks if file is writable; if yes, then the condition becomes true.	[-w $file] is true.
-x file	Checks if file is executable; if yes, then the condition becomes true.	[-x $file] is true.
-s file	Checks if file has size greater than 0; if yes, then condition becomes true.	[-s $file] is true.
-e file	Checks if file exists; is true even if file is a directory but exists.	[-e $file] is true.

Examples:

```
#!/bin/sh
```

```
file="~/script1.sh"

if [ -r $file ]
then
   echo "File has read access"
else
   echo "File does not have read access"
fi

if [ -w $file ]
then
   echo "File has write permission"
else
   echo "File does not have write permission"
fi

if [ -x $file ]
then
   echo "File has execute permission"
else
   echo "File does not have execute permission"
fi

if [ -f $file ]
then
   echo "File is an ordinary file"
else
   echo "This is sepcial file"
fi

if [ -d $file ]
then
   echo "File is a directory"
else
   echo "This is not a directory"
fi

if [ -s $file ]
then
   echo "File size is not zero"
else
   echo "File size is zero"
fi

if [ -e $file ]
then
```

```
   echo "File exists"
else
   echo "File does not exist"
fi
```

Decision Making in Shell

While writing a shell script, there may be a situation when you need to adopt one path out of the given two paths. So you need to make use of conditional statements that allow your program to make correct decisions and perform the right actions.

Linux Shell supports conditional statements which are used to perform different actions based on different conditions. We will now understand two decision-making statements here −

- The **if...else** statement
- The **case...esac** statement

The if...else statements

If else statements are useful decision-making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of **if...else** statement −

- if...fi statement

  ```
  #!/bin/sh

  a=10
  b=20

  if [ $a == $b ]
  then
     echo "a is equal to b"
  fi

  if [ $a != $b ]
  then
     echo "a is not equal to b"
  fi
  ```

- if...else...fi statement

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
else
   echo "a is not equal to b"
fi
```

- if...elif...else...fi statement

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
elif [ $a -gt $b ]
then
   echo "a is greater than b"
elif [ $a -lt $b ]
then
   echo "a is less than b"
else
   echo "None of the condition met"
fi
```

Most of the if statements check relations using relational operators discussed in the previous chapter.

The case...esac Statement

You can use multiple **if...elif** statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated **if...elif** statements.

There is only one form of **case...esac** statement which has been described in detail here −

- case...esac statement

```
#!/bin/sh

option="${1}"
case ${option} in
   -f) FILE="${2}"
      echo "File name is $FILE"
      ;;
   -d) DIR="${2}"
      echo "Dir name is $DIR"
      ;;
   *)
      echo "`basename ${0}`:usage: [-f file] | [-d directory]"
      exit 1 # Command to come out of the program with status 1
      ;;
esac
```

Nesting While Loops

```
#!/bin/sh

a=0
while [ "$a" -lt 10 ]    # this is loop1
do
   b="$a"
   while [ "$b" -ge 0 ]  # this is loop2
   do
      echo -n "$b "
      b=`expr $b - 1`
   done
   echo
   a=`expr $a + 1`
done
```

If you run it, you will get

```
0
1 0
2 1 0
3 2 1 0
4 3 2 1 0
5 4 3 2 1 0
6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
8 7 6 5 4 3 2 1 0
9 8 7 6 5 4 3 2 1 0
```

Now, you can also run infinite loops but you can stop the run under a conditions, for example

```
#!/bin/sh

a=0

while [ $a -lt 10 ]
do
   echo $a
   if [ $a -eq 5 ]
   then
      break
   fi
   a=`expr $a + 1`
done
```

Now, you can dive in!!!