==

FUNDAMENTALS OF LINUX TERMINAL

==

Very important part of Linux Environment is Linux Terminal. What we can do in Linux Terminal?
Everything that we do in Graphical Environment, when we use Graphical User Interfaces with help of our mouse, we can also do the same things in terminal, only in much faster and productive way.

In Linux Terminal we can:

- install our software and tools from remote repositories
- create list of our job or create isolated independent service
- copy, add and delete files and directories
- zip and unzip files as packages
- cleaning up files from unnecessary content
- change permissions and ownership of files and directories
- create specific users and groups of users
- mush faster modified, update, create, delete, and read files
- execute scripts with a huge list of commands
- create schedule for our tasks and jobs
- do streaming, redirecting and appending specific source to specific location
- do remote file transfer of our data
- create system variables which help us to better explain and understand environment
- setup firewall and configure our traffic local and remote traffic
- create and remove new partitions of our hard drives

Let's now explore Linux Terminal Commands, and see how they work.
Shortcut for opening terminal is CTRL+ALT+T, or create you own shortcut in Keyboard sector in System Settings.

==

ls – LIST CONTENT COMMAND

==

If we open terminal, we will be in our home directory. If we execute ls command we will see list of files and directories. In my case output will be:

```
a413@mN:~$ ls
a413Kingdom        gitLib        Public       a413library
aos                Music      Desktop      Templates
Documents          Downloads     Pictures     Videos
```

This command can have own attributes, like flags, which will explain to us content with more details.
If you want to see whole list of flags, type **ls --help** in your terminal.

For example if we type **ls -la**, that means that we add **l** and **a** flags into our command.
Now we must see what **l** and **a** flag do.

-a flag do not ignore entries starting with .
- that means that on our list we will have also hidden files if there is any

-l means that we will use long listing format

Some important and very useful flags are:

-d, --directory	list directories themselves, not their contents
-R, --recursive	list sub-directories recursively

```
-s, --size               print the allocated size of each file, in blocks
-t                       sort by modification time, newest first
```

In my case when I execute ls -la command, I get:

```
a413@mN:~$ ls -la
drwxrwxrwx   13 a413 a413        4096 Oct 28 14:16 a413library
-rw-r--r--    1 a413 a413        3890 Oct 26 19:16 .bashrc
```

To explore that all these details means, we will take two examples, two lines from this output. It will be:

```
drwxrwxrwx   13 a413 a413        4096 Oct 28 14:16 a413library
-rw-r--r--    1  a413 a413       3890 Oct 26 19:16 .bashrc
```

==

First content is directory and second is file. How we know that? By looking on first sign.
If first sign is "d", that means that content is directory, and if sign is "-" , it means that content is file.

==

Next 9 signs tells us about permissions that specific content have. What does they do?
We have 9 signs separated is 3 groups. First 3 are first group, second 3 are second, and third 3 are third group.
First group is defined for user permission. Second group is defined for group of users permission. Third group is defined for all users which are not user from first group, and they don't belong to group of users defined in second group, which means they are other.

Every group have the same 3 signs: **rwx**
In case where we have permissions to:

- read, write and execute we will have **rwx** group
- read and write, but we don't have permission to execute, we will have **rw-**
- read, but we don't have permission to write and execute, we will have **r--**

==

Let's look again on our lines:

```
drwxrwxrwx   13 a413 a413        4096 Oct 28 14:16 a413library
-rw-r--r--    1  a413 a413       3890 Oct 26 19:16 .bashrc
```

The second column, in this case numbers 13 and 1,is the number of hard links to the file. For a directory, the number of hard links is the number of immediate sub-directories it has plus its parent directory and itself. In next chapters we will explain what is hard and soft link.

Third column explain us ownership of the content. First a413 is ownership of that specific user and second a413 is ownership of that specific group of users. If it's strange to you how they are the same, don't worry. User can create his own group with his own name, and that's not the problem.

Fourth column tell us size of the content.
Fifth column show us last modified date and time.
Sixth column show us name of the directory or file.

==

cd – CHANGE DIRECTORY COMMAND

==

We use this command to change our working directory or to go on specific location.
For example if I am in home directory, I will list content, and change my current location via **cd** command:

```
a413@mN:~$ ls
a413Kingdom          gitLib              a413library
a413@mN:~$ cd gitLib/
a413@mN:~/gitLib$ ls
demo-checkout-commit  demo-pull-rebase  first_repo
```

If we want to change our current location, **pwd** command will tell us our current working directory.

```
a413@mN:~/gitLib$ pwd
/home/a413/gitLib
```

If we want to go further, then we write another location:

```
a413@mN:~/gitLib$ cd first_repo/
a413@mN:~/gitLib/first_repo$
```

How we can go back, on previous location? With command .. we can go to previous sub-directory, like this:

```
a413@mN:~/gitLib/first_repo$ cd ..
a413@mN:~/gitLib$
```

If we want to go even further in the same command we will use syntax like this:

```
a413@mN:~/gitLib/first_repo$ cd ../..
a413@mN:~$ pwd
/home/a413
```

==

We must mention one important thing. There are **absolute** and **relative** paths. What is the different?

Absolute path or **full path** is defined as specifying the location of a file or directory from the root directory (/), regardless of the current working directory. To do that, it must include the root directory.
Relative path starts from some given working directory, avoiding the need to provide the full **absolute path**.

==

Let's say we are in home directory of our user, at (~/) location. Now we want to create directory.

```
a413@mN:~$ mkdir location1
```

Now we have location1 directory in our home directory.
If we want to enter into location1 directory by typing **absolute path** we will type:

```
a413@mN:~$ cd /home/a413/location1/
a413@mN:~/location1$
```

If we want to enter into location1 directory by typing **relative path** we will type:

```
a413@mN:~$ cd location1/
a413@mN:~/location1$
```

==

mkdir – CREATE NEW DIRECTORY

==

With this command we create new directories. We can create them by typing their absolute or relative path.

```
a413@mN:~$ mkdir location413                   > Create via relative path
a413@mN:~$ mkdir /home/a413/location414        > Create on the same location via absolute path
a413@mN:~$ ls
location413  location414
```

We can also create multiple directory at the same time, in one command:

```
a413@mN:~$ mkdir {location420,location430,location440}
a413@mN:~$ ls
location413  location414 location420 location430 location440
```

We can also create multiple directories like this:

```
a413@mN:~$ mkdir {/home/a413/location450,/home/a413/location460}
a413@mN:~$ ls
location413  location414 location420 location430 location440 location450 location460
```

==

With **mkdir** function we have two important flags that we can add: **p** and **m.**

==

What is flag -p in mkdir command?

Let's say we want to create directory **/location500/test1/** in our home directory.
When we execute mkdir **/location500/test1/**, we will get error.

```
a413@mN:~$ mkdir /location500/test1/
mkdir: cannot create directory '/location500/test1/': No such file or directory
```

What is the problem? We have error because we can't create test1 directory. Why?
Because in home directory location500 doesn't exist, so in that case we don't have right way to go to test1 directory.

Now we will execute mkdir with -p flag. Remember that we must use absolute path to create directories.

```
a413@mN:~$ mkdir -p /home/a413/location500/test1
```

Now we didn't get any errors. Why?

Because in this case, if there is any parent directory of test1 directory that does not exist, -p flag will create that directory for us. So in this case, when we check home directory we will see that location500 directory now exist.

```
a413@mN:~$ ls
location413  location414 location420 location430 location440 location450 location460 location500
```

==

What is flag -m in mkdir command?
This flag set directory mode (as in chmod) while we create directory.

What is mode directory? It is permission that we give to specific directory while we create him.
Let's list contect of home directory:

```
a413@mN:~$ ls -la
drwxr-xr-x  2 a413 a413      4096 Oct 30 21:13 location1
```
==

Here we can see that all these contents are directories, and they all have the same mode: **rwxr-xr-x**
This means that:

- first group/U group have permission **rwx**
- second group/G group have permission **r-x**
- third group/O group have permission **r-x**

These permissions we can present in mode of numbers, in this case binary mode.

Every x is presented like 2^0,which means that value for x is 1,if execute is granted. If it's not, value is zero.
Every w is presented like 2^1,which means that value for w is 2,if write is granted. If it's not, value is zero.
Every r is presented like 2^2,which means that value for r is 4,if read is granted. If it's not, value is zero.

So lets convert some permissions into number mode:

```
-rwx    = 4+2+1 = 7         -r-x    = 4+0+1 = 5
-r--    = 4+0+0 = 4         ----    = 0+0+0 = 0
```

Remember, each group have their own number. So when we create new directory, our mode could be like:

777 750 644 600

How we get 777?
That means that U group have value 7, G group have value 7, and O group have value 7. In that case we have:

rwxrwxrwx

Every group have permission to read and write and execute.

How we get 644?
That means that U group have value 6, G group have value 4, and O group have value 4. In that case we have:

rw-r--r--

That means that U group can read and write, while G and O group can only read.
==
So, now if we create directory with specific mode, we will have this:

```
a413@mN:~$ mkdir -m 600 location600
a413@mN:~$ ls -la
drwxr-xr-x  2 a413 a413      4096 Oct 30 21:13 location1
…
…
drw-------  2 a413 a413      4096 Oct 30 22:26 location600
```

Our new directory doesn't have rwxr-xr-x like others.
He have new permission rw------- that we give to him while we creating him.

==

rmdir – REMOVE DIRECTORIES

==

Let's list all directories which have pattern `location` in their name:

```
a413@mN:~$ ls -la | grep location
```

To remove one directory: `a413@mN:~$ rmdir location1`
To remove multiple directories: `a413@mN:~$ rmdir location413 location414`

Now, as we can see we removed location1, location413,location414 directories.

```
a413@mN:~$ ls -la | grep location
```

If we want to remove every directory that have in his beginning pattern `location`, we will do:

```
a413@mN:~$ rmdir location4*
```

Now we can see that 420,430,440,450 and 460 location are gone.

```
a413@mN:~$ ls -la | grep location
drwxr-xr-x  3 a413 a413     4096 Oct 30 21:45 location500
drw-------  2  a413 a413     4096 Oct 30 22:26 location600
```

How patterns works?

loc* > Everything that start with loc
*loc > Everything that end with loc
on6 > Everything that have on6 pattern in middle

If we want to remove location 500, we will execute:

```
a413@mN:~$ rmdir *500
rmdir: failed to remove 'location500': Directory not empty
```

Why we get error? Because location500 is not empty.
How we can remove directory with his files? For that we use **rm** command, which have 3 important flags.

rm -f, --force	ignore nonexistent files and arguments, never prompt
rm -r, -R, --recursive	remove directories and their contents recursively
rm -d, --dir	remove empty directories

Now we execute rm command, with flags and see how location500 is gone.

```
a413@mN:~$ rm -rf *500
a413@mN:~$ ls -la | grep location
drw-------  2 a413 a413     4096 Oct 30 22:26 location600
```

We use rm for removing directories, but rm is also for removing files. Just type rm and name of the file.

```
a413@mN:~$ touch test413        > Creating our new file
a413@mN:~$ rm test413           > Deleting our new file
```

==

LINUX TERMINAL KEYBOARD SHORTCUTS

==

These shortcuts are for working in internal Linux Terminal, which is open by CTRL+ALT+T.

Ctrl+L	Clear the screen
Tab	Tab key autocomplete the name of command/file/directory etc. For eg. type **"**tou**"** and type tab
UP ARROW Key	Shows you previous command in descending order
DOWN Arrow Key	When you are using **UP ARROW Key** to get previous command. Then try this **DOWN Arrow**,it will show forward command history
CTRL+p	It shows previous history,same like UP arrow key
CTRL+n	It shows forward history,same like DOWN arrow key
Ctrl+A	Move cursor to start of the line
Ctrl+E	Move cursor to end of the line
HOME	Move cursor at the start of the line
END	Move cursor at the end of the line
CTRL + RIGHT ARROW	Move cursor one word right hand side
CTRL + RIGHT ARROW	Move cursor one word left hand side
Ctrl+K	It cut everything from the cursor to end of the line
CTRL+k	It cut the line from the position of the cursor to the end of the line
CTRL+SHIFT+c	To copy selected text
CTRL+SHIFT+v	To paste you last copied by CTRL+SHIFT+c
CTRL+U	Erase the current line.
CTRL+D	Exit Terminal

==

touch – CREATING FILES

==

For creating files just type touch command and name of the files. We can use absolute or relative paths.

```
a413@mN:~$ touch test413
a413@mN:~$ touch /home/a413/test423
a413@mN:~$ ls -la | grep test
-rw-r--r--  1 a413 a413        0 Oct 31 01:26 test413
-rw-r--r--  1 a413 a413        0 Oct 31 01:25 test423
```

If we want to create multiple files on one command, we will type:

```
a413@mN:~$ touch test430 test440
a413@mN:~$ touch /home/a413/test450 /home/a413/test460
a413@mN:~$ ls -la | grep test
-rw-r--r--  1 a413 a413        0 Oct 31 01:26 test413
-rw-r--r--  1 a413 a413        0 Oct 31 01:25 test423
-rw-r--r--  1 a413 a413        0 Oct 31 01:29 test430
-rw-r--r--  1 a413 a413        0 Oct 31 01:29 test440
-rw-r--r--  1 a413 a413        0 Oct 31 01:30 test450
-rw-r--r--  1 a413 a413        0 Oct 31 01:30 test460
```

==

mv – MOVING AND RENAMING

==

How moving in terminal work? We have a pattern:

mv source1 source2 ... destination

If we want to move file to specific directory, we type:

```
a413@mN:~$ mkdir location777
a413@mN:~$ touch test777
a413@mN:~$ mv test777 /home/a413/location777/
a413@mN:~$ cd location777/
a413@mN:~/location777$ ls
test777
```

If we want to move multiple files to specific directory, we type:

```
a413@mN:~/location777$ cd ..
a413@mN:~$ touch test888 test999
a413@mN:~$ mv test888 test999 location777/
a413@mN:~$ cd location777/
a413@mN:~/location777$ ls
test777  test888  test999
```

We can also use pattern to move specific type of files or find files according.

```
a413@mN:~$ touch test1000-1
a413@mN:~$ mv *1000* location777/
a413@mN:~$ cd location777/
a413@mN:~/location777$ ls
test1000-1  test777  test888  test999
```

With mv command we can define some specific additional attributes, like: **i,f,n,u and v.**

mv -i source destination

This command will get for us interactive command line, where we will be asked to apply specific override before override is done:

```
a413@mN:~$ mkdir test1000
a413@mN:~$ mkdir -p test2000/test1000
a413@mN:~$ mv -i test1000/ test2000/
```

In next interactive line, we can type YES or NO, to apply override:

```
mv: overwrite 'test2000/test1000'? Yes
```

==

If we don’t need interactive line, and we want to override location, without any message, then we use **-f** flag.

```
a413@mN:~$ mkdir test1000
a413@mN:~$ mv -f test1000/ test2000/
```

==

If we want to skip any override, then we will use **-n** flag:

```
a413@mN:~$ mkdir test1000 test1001
a413@mN:~$ touch test1000/desc.txt
a413@mN:~$ mv -n test100* test2000/
a413@mN:~$ cd test2000/test1000
a413@mN:~/test2000/test1000$ ls -la
total 8
drwxr-xr-x 2 a413 a413 4096 Jan 16 19:14 .
drwxr-xr-x 4 a413 a413 4096 Jan 16 19:16 ..
```

As we can see, we didn't override test1000 directory, because **-n** flag does not apply any override.

==

If we want to move only when the SOURCE file is newer than the destination file or destination file is missing, we will use **-u** flag, which is state for update:

```
mv -u source destination
```

==

If we need explanation, what has been made with **mv** command, we can use **-v** flag, beside others flags. Example:

```
a413@mN:~$ mv -iv test1000/ test2000/
mv: overwrite 'test2000/test1000'? Yes
renamed 'test1000/' -> 'test2000/test1000'          (Our message!)
```

==

cp – COPY DIRECTORIES AND FILES

==

How we copy directories and files in terminal? To see all options, in terminal, type: **cp --help** . We have a pattern:

cp [options] source dest

cp command main options are:

option	description
cp -a	archive files
cp -f	force copy by removing the destination file if needed
cp -i	interactive - ask before overwrite
cp -l	link files instead of copy
cp -L	follow symbolic links
cp -n	no file overwrite
cp -R	recursive copy (including hidden files)
cp -u	update - copy when source is newer than dest
cp -v	verbose - print informative messages

```
a413@mN:~$ mkdir test413
a413@mN:~$ touch hello1 hello2 hello3

a413@mN:~$ cp hello1 test413/
a413@mN:~$ cp /home/a413/hello2 test413/
a413@mN:~$ cp /home/a413/hello3 /home/a413/test413/
a413@mN:~$ ls -la test413/
-rw-r--r--  1 a413 a413    0 Nov 12 16:53 hello1
-rw-r--r--  1 a413 a413    0 Nov 12 16:53 hello2
-rw-r--r--  1 a413 a413    0 Nov 12 16:53 hello3
a413@mN:~$ rm test413/*
```

==

```
a413@mN:~$ cp hello* test413/
a413@mN:~$ ls -la test413/
-rw-r--r--  1 a413 a413     0 Nov 12 17:00 hello1
-rw-r--r--  1 a413 a413     0 Nov 12 17:00 hello2
-rw-r--r--  1 a413 a413     0 Nov 12 17:00 hello3
```
==
```
a413@mN:~$ mkdir test513
a413@mN:~$ mkdir test513/location1
a413@mN:~$ touch test513/location1/matrix

a413@mN:~$ cp -R test513/ test413/
a413@mN:~$ sudo apt-get update && sudo apt-get install tree
```
==

chmod – CHANGE DIRECTORIES AND FILES PERMISSIONS

==

When we created new files, we explained what is file mode. This can be used only by super user.
Now we want to know how to change permissions of directories and files that already exist.

```
a413@mN:~$ ls -la | grep test2000
-rw-r--r--  1 a413 a413          0 Oct 31 03:49 test2000-1
a413@mN:~$ chmod 750 test2000-1
a413@mN:~$ ls -la | grep test2000
-rwxr-x---  1 a413 a413          0 Oct 31 03:49 test2000-1
```

If we want to change permissions for directory and all content inside, then we add -R flag:

```
a413@mN:~$ cd location777/
a413@mN:~/location777$ ls
test1000-1  test777  test888  test999
a413@mN:~/location777$ ls -la
-rw-r--r--  1 a413 a413     0 Oct 31 03:38 test1000-1
-rw-r--r--  1 a413 a413     0 Oct 31 03:12 test777
-rw-r--r--  1 a413 a413     0 Oct 31 03:17 test888
-rw-r--r--  1 a413 a413     0 Oct 31 03:17 test999
```

We can change permissions with absolute path, like this:

```
a413@mN:~/location777$ chmod 750 -R /home/a413/location777/
```

Or with relative path, like this:

```
a413@mN:~/location777$ chmod 750 -R location777/
a413@mN:~/location777$ ls -la
-rwxr-x---  1 a413 a413     0 Oct 31 03:38 test1000-1
-rwxr-x---  1 a413 a413     0 Oct 31 03:12 test777
-rwxr-x---  1 a413 a413     0 Oct 31 03:17 test888
-rwxr-x---  1 a413 a413     0 Oct 31 03:17 test999
```

Note: If we change directory permissions, make sure that specific directory has **x** permissions, because without **x** permissions we can’t access that location.

==

chown – CHANGE DIRECTORIES AND FILES OWNERSHIP

==

This can be executed only by super user.

```
a413@mN:~/location777$ ls -la
-rwxr-x---   1      a413 a413     0   Oct 31 03:38 test1000-1
-rwxr-x---   1      a413 a413     0   Oct 31 03:12 test777
-rwxr-x---   1      a413 a413     0   Oct 31 03:17 test888
-rwxr-x---   1      a413 a413     0   Oct 31 03:17 test999
```

As we can see, user that owns files is a413 and group that have ownership is a413. If we want to change that, type:

```
a413@mN:~/location777$ chown root:root test1000-1
chown: changing ownership of 'test1000-1': Operation not permitted
a413@mN:~/location777$ sudo chown root:root test1000-1
a413@mN:~/location777$ ls -la | grep test1000-1
-rwxr-x---   1 root root     0 Oct 31 03:38 test1000-
```

==

nano – TEXT EDITOR

==

Nano editor is place where we can create, edit and explore our files. Run editor with: **sudo nano name_of_your_file**

==

CTRL+X	(F2)	Close the current file buffer / Exit from nano
CTRL+O	(F3)	Write the current file to disk
F11		Full Screen
CTRL+W	(F6)	Search for a string or a regular expression
CTRL+\	(M-R)	Replace a string or a regular expression
CTRL+K	(F9)	Cut the current line and store it in the cut buffer
CTRL+U	(F10)	Uncut from the cut buffer into the current line – Only with this we can duplicate line
CTRL+C		**Display the position of the cursor**
ALT+G		**Go to line and column number**
ALT+\	(M-|)	Go to the first line of the file
ALT+/	(M-?)	Go to the last line of the file
ALT+-]		Go to the matching bracket
ALT+A		Mark text starting from the cursor position
ALT+U		Undo the last operation
ALT+E		Redo the last undone operation
CTRL+LEFT		Go back one word
CTRL+RIGHT		Go forward one word
CTRL+A	(Home)	Go to beginning of current line
CTRL+E	(End)	Go to end of current line
CTRL+P	(Up)	Go to previous line
CTRL+N	(Down)	Go to next line
∧Z		Suspend the editor (if suspension is enabled)

===

REDIRECTS (> AND >>)

===

Redirects are used to send already existing form to another place or file.

Sign **(>)** is used to **override** redirected value on location where we are sending.
Sign **(>>)** is used to **append** redirected value on location where we are sending.

Note: Override means that one process will take place of another process, because of higher priority.
Append means that we add some value to already existing location or object.

There is one command that can help us with that.

Echo command is string data type, with which we can print message on our terminal, in case we are part of some sort of script, or we can redirect that string to specific location, for specific purpose.

```
a413@mN:~$ echo "Hello Golang World!"
Hello Golang World!
```

We can define string in our echo command and redirect him to specific file. If file exist, redirect will affect on that file. If file doesn't exist, redirect will create new file, and execute following redirect.

```
a413@mN:~$ touch test5000
a413@mN:~$ echo "Hello Golang World!" > test5000
a413@mN:~$ echo "Golang modules are great!" >> test5000
a413@mN:~$ cat test5000
Hello Golang World!
Golang modules are great!
a413@mN:~$ ls -la | grep test 6000
grep: 6000: No such file or directory
a413@mN:~$ echo "Golang is compiled language!" > test6000
a413@mN:~$ cat test6000
Golang is compiled language!
```

===

TEXT COMMANDS

===

Now, we know what is redirect. But what is **cat** command? It prints output of our files in directly in terminal.

```
a413@mN:~$ cat test5000 test6000       > we combine two outputs in one
Hello Golang World!
Golang modules are great!
Golang is compiled language!

a413@mN:~$ cat test*                   > combine all outputs with prefix file
cat: test3001: Is a directory
cat: test4001: Is a directory
Hello Golang World!
Golang modules are great!
Golang is compiled language!
```

```
a413@mN:~$ cat test5* test6*
Hello Golang World!
Golang modules are great!
Golang is compiled language!
a413@mN:~$ cat test5* test6* > test7000             > output is copied to new text file
a413@mN:~$ cat test7000
Hello Golang World!
Golang modules are great!
Golang is compiled language!
```

If the file exist it will override, else it will create new file, and add output!

```
awdawdawd 2>> test3.txt
```

This will append error in our text file because we don't have awdawdawd command!

==

Some other commands for file manipulation:

==

```
a413@mN:~$ touch golang.txt
a413@mN:~$ echo "Code - Line 1" >> golang.txt
a413@mN:~$ echo "Code - Line 2" >> golang.txt
a413@mN:~$ echo "Code - Line 3" >> golang.txt
```

If we need to get number of lines in our file, we can use **wc** command:

```
a413@mN:~$ cat golang.txt | wc -l
3
```

We can also use, for multiple files:

```
a413@mN:~$ echo "Code - Line 4" >> golang_2.txt
a413@mN:~$ cat golan* | wc -l
4
```

If we need to split one file into two, we can use **split** command, with attribute **-l** which define number of specific line after which we will split file into two parts, while original file stay untouched:

```
a413@mN:~$ split -l 2 golang.txt
```

Two new files are defined by name **xaa** and **xab**:

```
a413@mN:~$ ls -la | grep xa
-rw-r--r--    1 a413 a413        28 Jan 16 21:38 xaa
-rw-r--r--    1 a413 a413        14 Jan 16 21:38 xab
```

Let's see context of two new files:

```
a413@mN:~$ cat xaa
Code - Line 1
Code - Line 2
a413@mN:~$ cat xab
Code - Line 3
```

If we want to see the difference between those two files, we can use **diff** command:

```
a413@mN:~$ diff xaa xab
1,2c1
< Code - Line 1
< Code - Line 2
---
> Code - Line 3
```

==

We can also use **head** and **tail** commands, to get lines from start or from end of file.

```
head file_name        >> prints first 10 lines
tail  file_name       >> prints last 10 lines
```

If there is no more then 10 lines inside of file, it will print whole file.
If we want specific number of lines from file, we can set **-n** flag like this:

```
a413@mN:~$ head -n 2 golang.txt
Code - Line 1
Code - Line 2
a413@mN:~$ tail -n 2 golang.txt
Code - Line 2
Code - Line 3
```

We can also redirect specific lines into another file:

```
a413@mN:~$ tail -n 2 golang.txt > golang_3.txt
a413@mN:~$ cat golang_3.txt
Code - Line 2
Code - Line 3
```

==

We can also use streams to better understand our redirects.
We can use **2>** and **&1** tags to redirect our errors, in case that we can't execute specific command:

```
a413@mN:~$ cat file1 file2 > combineFiles 2> myErrorFile
```

This command redirect our error in myErrorFile file. If we see content of that file, we will get:

```
a413@mN:~$ cat myErrorFile

a413@mN:~$ cat file1 file2 > combineFiles 2>&1
```

This command redirect our error in combineFiles. If we see content of that file, we will get:

```
a413@mN:~$ cat combineFiles
```

We can also use **stdout** to add data to file directly from terminal, like this:

```
a413@mN:~$ cat 1> matrix
We love gRPC!                (After Enter, press CTRL+D to finish your stream!)
a413@mN:~$ cat 1>> matrix
We also love Postgresql!     (After Enter, press CTRL+D to finish your stream!)
a413@mN:~$ cat < matrix
We love gRPC!
We also love Postgresql!
```

==

SH EXTENSION AND EXECUTION OF FILE

==

We can execute text file with .txt extension or we can execute .sh extension with sh command.
In new txt file, for example, test5.txt, then we will add next lines:

```
awdawd
ls /etc/
```

Next, we will add execute permissions to that file: **chmod u+x test5.txt**
Then, we execute that file: **./test5.txt**
If we have .sh extension, we will execute file like this: **sh script.sh**

We will get both outputs, error and the list of etc directory!

./test5.txt > output.out	>>	this will not work because we can't execute this!
./test5.txt 2> output.out	>>	this tell us that errors from file, if exist will be written in output.out file!
./test5.txt &> output.out	>>	this will put whole command log into file, as override!
./test5.txt &>> output.out	>>	this will append whole command log into file, as override!

"|" - pipe means that you take output from previous command as input for next command

awdawdawd 2>&1	>> &1 (we define error as standard output, so we can use it in pipe!
awdawdawd 2>&1 | grep "command"	>> awdawdawd: command not found ("command" have red background)

Example with SH:

```
a413@mN:~$ touch script413.sh
a413@mN:~$ echo "ls -la | grep D" > script413.sh
a413@mN:~$ sh script413.sh
drwxr-xr-x  2 a413 a413      4096 Oct 26 13:38 Desktop
drwxr-xr-x  3 a413 a413      4096 Oct 27 09:34 Documents
drwxr-xr-x  4 a413 a413      4096 Oct 31 13:00 Downloads
```

==

Archive, Compress, Unpack and Uncompress using TAR and GZIP

==

If we want to compress directory:

1. First we need to create archive (man tar):

```
a413@mN:~$ mkdir test9000
a413@mN:~$ touch hello1 hello2
a413@mN:~$ tar -cvf archive1.tar test9000/ hello1 hello2
```

>> c stand for create, v is for us to seen what tar did, and f is to using specific archive name

2. Command **tar -tf archive1.tar** will show us content of archive:

```
a413@mN:~$ tar -tf archive1.tar
test9000/
hello1
hello2
```

Now, when we have tar archive, we create tar.gz:

```
a413@mN:~$ gzip archive1.tar
a413@mN:~$ ls -la | grep archive
-rw-r--r--  1 a413 a413       168 Nov 12 15:22 archive1.tar.gz
```

3. If we want to compress directly into tar.gz file, we will:

```
a413@mN:~$ tar -cvzf archive2.tar.gz test9000/ hello1 hello2
a413@mN:~$ ls -la | grep archive
-rw-r--r--  1 a413 a413       168 Nov 12 15:22 archive1.tar.gz
-rw-r--r--  1 a413 a413       155 Nov 12 15:27 archive2.tar.gz
```

4. If we want to uncompress, will use:

```
a413@mN:~$ gzip -d archive1.tar.gz
a413@mN:~$ ls -la | grep archive
-rw-r--r--  1 a413 a413     10240 Nov 12 15:22 archive1.tar
-rw-r--r--  1 a413 a413       155 Nov 12 15:27 archive2.tar.gz
```

5. Now we want to extract that tar archive. We use: **tar -xvf archive1.tar**
6. If we want to extract tar.gz archive directly into failes, we use: **tar -xzvf archive1.tar.gz**

If we have already files in active dir under that names, like we have in compressed archive, files will be override!

7. If we want to see the difference in override files, we will use:

```
tar -dvf archive.tar.gz
```

8. If we need information about compression, we use:

```
a413@mN:~$ gzip -l archive2.tar.gz
```

==

CREATE, DELETE, MODIFY LOCAL USER ACCOUNTS

==

To create a new user account a413 the adduser command you would run:

```
sudo adduser a413
```

You will be asked a series of questions. The password is required and all other fields are optional.
Finally, confirm that the information is correct by entering Y.

The command will create the new user's home directory, and copy files from /etc/skel directory to the user's home directory. Within the home directory, the user can write, edit, and delete files and directories.
By default on Ubuntu, members of the group sudo are granted with sudo access.
If you want the newly created user to have administrative rights, add the user to the sudo group:

```
sudo usermod -aG sudo a413
```

To delete the user, without deleting the user files, run:

```
sudo deluser a413
```

If you want to delete and the user's home directory and mail spool use the --remove-home flag:

```
sudo deluser --remove-home a413
```

The same commands apply for any Ubuntu-based distribution, including Debian, Kubuntu, and Linux Mint.

==

When we type **id** in terminal, we can see details about current login user:

```
a413@mN:~$ id
uid=1000(a413) gid=1000(a413) groups=1000(a413),4(adm),24(cdrom),27(sudo),30(dip)
```

There are 3 groups od ID for users:

```
> 0         >> reserved for root user
> 1-200     >> system users for specific processes
> 201-999   >> use system processes but don't own files on the system
```

Every user can have one primary group. When we create new user, system create new group under same name.
Now explore following files:

```
> nano /etc/shadow
> nano /etc/group
> nano /etc/gshadow
> nano /etc/default/useradd
```

To explore useradd command type: `useradd --help`
To change user password type: `sudo passwd username`

Now, we will create new user b413, with new ID 1413, with home directory /home/b413/:

```
a413@mN:~$ sudo useradd -u 1413 -d /home/b413 b413
a413@mN:~$ cat /etc/passwd
```

When we execute second command, we can see that last add user into our list is user b413.
Now we want to add b413 to new group.

```
a413@mN:~$ sudo usermod -G golang b413
usermod: group 'golang' does not exist
```

We get an error and message that golang group doesn't exist. How we can create new group?
Let's first list all groups:

```
a413@mN:~$ groups
a413 adm cdrom sudo dip plugdev lpadmin sambashare
```

First we will create new gruup, then we add our new user to new group:

```
a413@mN:~$ sudo groupadd golang
a413@mN:~$ sudo usermod -aG golang b413
```

The -a option tells usermod we are appending and the -G option tells usermod we are appending to the group name that follows the option. Now we check existence of golang groups, and who is part of that group:

```
a413@mN:~$ getent group | grep golang
golang:x:1414:b413
```

==

Here we can see that b413 user belong to golang group, and that id of golang group is 1414.

```
a413@mN:~$ id b413
uid=1413(b413) gid=1413(b413) groups=1413(b413),1414(golang)
```

If you want to delete user home directory, while you deleting user as well, you type:

```
sudo deluser --remove-home username
```

To remove the user and all files owned by this user on the whole system, type:

```
sudo deluser --remove-all-files username
```

==

To modify the username of a user: **usermod -l new_username old_username**
To change the password for a user: **sudo passwd username**
To change the shell for a user: **sudo chsh username**

And, of course, see also: man adduser, man useradd, man userdel... and so on.

We have primary and secondary groups:

```
$ usermod -g class1 student          > change primary group
$ groups student                     > check primary group
```

New file is created under primary group. Primary group is group in which user belong after login.
If we have,let's say chmod 070, only members of the specific group will have access to rwx.
If the student belong to some group, but that group isn't his primary, he also have permissions!

```
$ newgrp class1        > defines our new primary group of our active user
```

We can have case where 3 groups can belong to 1 group. That is suplementary group. Here, 10 students belong to MATH group, 10 students belong to BIOLOGY group, but MATH and BIOLOGY belong to SCHOOL1 group!

==

SOFT AND HARD LINKS

==

A **symbolic** or **soft link** is an actual link to the original file, whereas a **hard link** is a mirror copy of the original file. If you delete the original file, the soft link has no value, because it points to a non-existent file. But in the case of hard link, it is entirely opposite. Even if you delete the original file, the hard link will still has the data of the original file. Because hard link acts as a mirror copy of the original file.

In a nutshell, a soft link:

- can cross the file system,
- allows you to link between directories,
- has different id node number and file permissions than original file,
- permissions will not be updated,
- has only the path of the original file, not the contents.

A hard Link:

- can't cross the file system boundaries (i.e. A hardlink can only work on the same filesystem),
- can't link directories,
- has the same id node number and permissions of original file,
- permissions will be updated if we change the permissions of source file,
- has the actual contents of original file, so that you still can view the contents, even if the original file moved or removed.

Creating Soft Link or Symbolic Link

Let us create an empty directory called **"test9000"** and change to that directory:

```
a413@mN:~$ mkdir test9000 && cd test9000
```

Now, create a new file called **source.file** with some data as shown below.

```
a413@mN:~/test9000$ echo "Welcome to Linux OS" > source.file
```

Let us view the data of the source.file.

```
a413@mN:~/test9000$ cat source.file
Welcome to Linux OS
```

Well, the source.file has been created. Now, create the a symbolic or soft link to the source.file. To do so, run:

```
a413@mN:~/test9000$ ln -s source.file softlink.file
```

Let us compare the data of both source.file and softlink.file.

```
a413@mN:~/test9000$ cat source.file
Welcome to Linux OS
a413@mN:~/test9000$ cat softlink.file
Welcome to Linux OS
```

As you see in the above output, softlink.file displays the same data as source.file.
Let us check the inodes and permissions of softlink.file and source.file.

```
a413@mN:~/test9000$ ls -lia
2622988 lrwxrwxrwx   1 a413 a413   11 Oct 31 23:46 softlink.file -> source.file
2622987 -rw-r--r--   1 a413 a413   20 Oct 31 23:44 source.file
```

The **inode number (2622988** vs **2622987)** and **file permissions (lrwxrwxrwx** vs **-rw-r–r–)** are **different**, even though the softlink.file has same contents as source.file, . Hence, it is proved that soft link don't share the same inode number and permissions of original file. Now, remove the original file (i.e source.file) and see what happens.

```
a413@mN:~/test9000$ rm source.file
```

Check contents of the softlink.file using command:

```
a413@mN:~/test9000$ cat softlink.file
cat: softlink.file: No such file or directory
```

As you see above, there is no such file or directory called softlink.file after we removed the original file (i.e source.file). So, now we understand that soft link is just a link that points to the original file. The softlink is like a shortcut to a file. If you remove the file, the shortcut is useless.
==

Creating Hard Link

Create a file called **source.file** with some contents as shown below.

```
a413@mN:~/test9000$ rm softlink.file
a413@mN:~/test9000$ rm source.file
a413@mN:~/test9000$ echo "Welcome to Linux OS" > source.file
```

Let us verify the contents of the file.

```
$ cat source.file
 Welcome to Linux OS
```

source.file has been created now. Now, let us create the hard link to the source.file as shown below.

```
a413@mN:~/test9000$ ln source.file hardlink.file
```

Check the contents of hardlink.file.

```
$ cat hardlink.file
Welcome to Linux OS
```

You see the hardlink.file displays the same data as source.file.
Let us check the inode and permissions of softlink.file and source.file.

```
a413@mN:~/test9000$ ls -lia
2622987 -rw-r--r--  2 a413 a413   20 Nov  1 00:38 hardlink.file
2622987 -rw-r--r--  2 a413 a413   20 Nov  1 00:38 source.file
```
==
Both hardlink.file and source.file have the same the **inodes number (2622987)** and **file permissions (-rw-r--r--).**
Hence, it is proved that hard link file shares the same inodes number and permissions of original file.
Note: If we change the permissions on source.file, the same permission will be applied to the hardlink.file as well.

Now, remove the original file (i.e source.file) and see what happens.

```
a413@mN:~/test9000$ rm source.file
```

Check contents of hardlink.file using command:

```
a413@mN:~/test9000$ cat hardlink.file
Welcome to Linux OS
```

==

As you see above, even if I deleted the source file, I can view contents of the hardlink.file. Hence, it is proved that Hard link shares the same inode number, the permissions and data of the original file.

So, what is the difference between Hard link and the normal copied file?

You might be wondering why would we create a hard link while we can easily copy/paste the original file? Creating a hard link to a file is different than copying it. If you copy a file, it will just duplicate the content. So if you modify the content of a one file (either original or hard link), it has no effect on the other one. However if you create a hard link to a file and change the content of either of the files, the change will be be seen on both.

==

The source file has a single line that says – Welcome to Linux OS.

Append a new line, for example "Welcome to Linux" in source.file or hardlink.file. Let's start again:

```
a413@mN:~/test9000$ rm source.file
a413@mN:~/test9000$ rm hardlink.file
a413@mN:~/test9000$ echo "Welcome to Linux OS" > source.file
a413@mN:~/test9000$ ln source.file hardlink.file
a413@mN:~/test9000$ echo "Welcome to Golang Programming" >> hardlink.file
a413@mN:~/test9000$ cat source.file
Welcome to Linux OS
Welcome to Golang Programming
```

Now check contents of other file.

See? The changes we just made on source.file are updated in both files. Meaning – both files (source and hard link) synchronizes. Whatever changes you do in any file will be reflected on other one. If you normally copy/paste the file, you will not see any new changes in other file. For more details, check the man pages.

==

WGET

==

wget is a free utility for non-interactive download of files from the web. It supports HTTP, HTTPS, and FTP protocols, as well as retrieval through HTTP proxies.

wget is non-interactive, meaning that it can work in the background, while the user is not logged on, which allows you to start a retrieval and disconnect from the system, letting wget finish the work. By contrast, most web browsers require constant user interaction, which make transferring a lot of data difficult.

Address would be something like: **https://go.dev/dl/go1.17.3.linux-amd64.tar.gz**

Now go to Linux Terminal, and type:

```
a413@mN:~$ wget https://go.dev/dl/go1.17.3.linux-amd64.tar.gz
```

You will see how golang package is downloading into your working directory.

==

RUNNING MULTI COMMANDS AS ONE COMMAND

==

We can run multiple commands as one command. We can use two separators: ";" and "&&".

```
a413@mN:~$ mkdir go100;cd go100;
a413@mN:~/go100$

a413@mN:~/go100$ mkdir go200 && cd go200
a413@mN:~/go100/go200$
```

What is the difference between these two separators?

Commands separated by a ; are executed sequentially. The shell waits for each command to terminate in turn.
Command after && is executed if, and only if, command before && returns an exit status of zero, which means that there is no error in execution of that command. You can think of it as AND operator.

How we can run command in multiple lines, if we need more space?
We use separator "\" on the end of the line, and then press Enter. Terminal will give us new line for use.

```
a413@mN:~$ mkdir go300 \
> && cd go300 \
> && touch go_main
a413@mN:~/go300$ ls
go_main
```

==

ALIASES

==

Linux users often need to use one command over and over again. Typing or copying the same command again and again reduces your productivity and distracts you from what you are actually doing.

You can save yourself some time by creating aliases for your most used commands. Aliases are like custom shortcuts used to represent a command (or set of commands) executed with or without custom options. Chances are you are already using aliases on your Linux system.

You can see a list of defined aliases on your profile by simply executing alias command. Creating aliases is relatively easy and quick process. You can create two types of aliases – temporary ones and permanent.

==

To create **temporary alias**, type following syntax in your terminal: **alias alias_name="command"**

```
a413@mN:~$ alias l1="ls -la"
a413@mN:~$ l1
```

Why is he temporary? Because if we close that terminal, and open new one, new session won't have that alias.

==

To keep aliases between sessions, as **permanent alias**, you can save them in your user's shell configuration profile file.

File that we need is **~/.bashrc**. Here we will add list of our aliases, and after that they will be permanent.
We can edit **~/.bashrc** file with nano editor, or we can add aliases as redirect. If you add with nano, go to end of file, and there put list of your alias. After that save the file, and execute **source ~/.bashrc** to update your system.

```
a413@mN:~$ echo "alias l1=\"ls -la\"" >> ~/.bashrc
a413@mN:~$ tail -n1 ~/.bashrc
alias l1="ls -la"
a413@mN:~$ l1 | grep D
drwxr-xr-x  2 a413 a413      4096 Oct 26 13:38 Desktop
drwxr-xr-x  3 a413 a413      4096 Oct 27 09:34 Documents
drwxr-xr-x  4 a413 a413      4096 Nov  1 04:21 Downloads
```

Our alias is now permanent, and we can use him in any session.
To define a permanent alias we must add it in **~/.bashrc** file.
Useful list of aliases that can be productive, which you can add into your ~/.bashrc file:

==

```
alias r1="rm -rf -R"
alias sur="sudo su - root"
alias sn="sudo nano"
alias l1="ls -la"
alias l2="ls -la | grep"
alias l3="ls -la | grep -E"
```

===

$PATH VARIABLE

===

When you type a command into the command prompt in Linux, or in other Linux-like operating systems, all you're doing is telling it to run a program. Even simple commands, like **ls**, **cd**, **mkdir**, **rm**, and others are just small programs that usually live inside a directory on your computer called **/usr/bin.**

There are other places on your system that commonly hold executable programs as well; some common ones include **/usr/local/bin**, **/usr/local/sbin**, and **/usr/sbin**. Executable program can live practically anywhere on your computer: it doesn't have to be limited to one of these directories.

When you type a command into your Linux shell, it doesn't look in every directory to see if there's a program by that name, just the ones you specify. How does it know to look in the directories mentioned above? It's simple: They are a part of an environment variable, called **$PATH**, which your shell checks in order to know where to look.

Sometimes, you may wish to install programs into other locations on your computer, but be able to execute them easily without specifying their exact location. You can do this easily by adding a directory to your **$PATH**. To see what's in your **$PATH** right now, type in:

```
echo $PATH
```

==

Let's setup real example. We will install Golang Programming Language.

```
wget https://dl.google.com/go/go1.13.linux-amd64.tar.gz
sudo tar -C /usr/local -xzf go1.13.linux-amd64.tar.gz
sudo echo "export PATH=$PATH:/usr/local/go/bin" >> ~/.bashrc
source ~/.bashrc
go version

a413@mN:~$ which go
/usr/local/go/bin/go
```

This means that location of our go execute file is /usr/local/go/bin/go.

==

Our **go** is execute file which we need to run, build and test our Golang files.
When we execute go command, we will see this:

```
a413@mN:~$ go
Go is a tool for managing Go source code.
```

Because we add **/usr/local/go/bin** into our $PATH variable in our **~/.bashrc** file, we don't need to use whole absolute path to execute our go command. Now only we need to do is to type go, and command will be functional.

==

APT-GET

==

Use **apt** to work with packages in the Linux command line. Use **apt-get** to install packages. This requires root privileges, so use the **sudo** command with it. For example, if you want to install the text editor **jed** (as I mentioned earlier), we can type in the command **“sudo apt-get install jed”**. Similarly, any packages can be installed like this.

It is good to update your repository each time you try to install a new package. You can do that by typing **“sudo apt-get update”**. You can upgrade the system by typing **“sudo apt-get upgrade”**. We can also upgrade the distro by typing **“sudo apt-get dist-upgrade”**. The command **“apt-cache search”** is used to search for a package. If you want to search for one, you can type in **“apt-cache search jed”** (this doesn't require root).

While apt isn’t a command in itself, there are three commands that you must know to make full use of APT: add-apt-repository (**for locating third-party packages**), apt-get (for actually installing packages), and apt-cache (for searching your repositories). If your distro doesn’t use APT, it may use YUM, RPM, or some other alternative. Look into their equivalent commands.

==

SUDO – SUDOERS LIST

==

Before we execute commands, we need to know one thing.

What is SUDO?

SUDO is command that allows a permitted user to execute a command as the superuser or admin of the system. By default, SUDO requires that users authenticate themselves with a password.

That is fine for security reasons, but there will be a lots of commands executed with SUDO on start.
Then, in that case every time when we want to execute command, system will ask us for password.

Can we do something so system wouldn’t ask us every time for password? **YES WE CAN!**

There is a list of all administrators and superusers of the system.
That file is **/etc/sudoers**. How we can modify that file?

Execute in terminal following command:

```
sudo nano /etc/sudoers
```

Here **sudo** is sub-command, **nano** is text editor with whom we will edit our files and **/etc/sudoers** is our file which we will modify. Scroll to the last line of the file and add following line:

```
a413 ALL=(ALL) NOPASSWD:ALL
```

My user in this case is **a413**. So in your case, if for example, your user is **alex413**, your line is:

```
alex413 ALL=(ALL) NOPASSWD:ALL
```

Now when we execute command with sudo, inside of new terminal,
system won’t ask again our user for password. GREAT!

Created By Djurdjevic Nikola

Contact:

Discord:	Architect413#9400
Email:	architect@413.world