

# Linux Command Line

From Simple Commands  
to Advanced Level

**Become a Linux Expert  
FAST and EASY**

**Matthew Gimson**

Full Edition,  
Revised and Enlarged

# **Linux Command Line From Simple Commands to Advanced Level** Become a Linux Expert FAST and EASY!

***Full Edition Revised and Enlarged By Matthew Gimson***

**Copyright©2017 by Matthew Gimson All Rights Reserved**

Copyright © 2017 by Matthew Gimson All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

# **Table of Contents** [Introduction](#)

[Chapter 1- What is Linux?](#)

[Chapter 2- Linux Command Line Commands](#)

[Chapter 3- Input/output Redirection](#)

[Chapter 4- Wildcards in Linux](#)

[Chapter 5- File security in Linux](#)

[Chapter 6- Jobs and Processes](#)

[Chapter 7- Bash scripting Tricks](#)

[Chapter 8- Linux shell programming](#)

[Chapter 9- Bash One-liners](#)

[Chapter 10- Advanced Shell Programming](#)

[Chapter 11- Compiling UNIX software packages](#)

[Chapter 12- Linux Networking](#)

[Conclusion](#)

**Disclaimer** While all attempts have been made to verify the information provided in this book, the author does assume any responsibility for errors, omissions, or contrary interpretations of the subject matter contained within. The information provided in this book is for educational and entertainment purposes only. The reader is responsible for his or her own actions and the author does not accept any responsibilities for any liabilities or damages, real or perceived, resulting from the use of this information.

**The trademarks that are used are without any consent, and the publication of the trademark is without permission or backing by the trademark owner. All trademarks and brands within this book are for clarifying purposes only and are the owned by the owners themselves, not affiliated with this document. \*\***

# Introduction

Linux is the most widely used operating system in production environments, especially on server machines. This means that there is more complexity involved in the use of a Linux operating system. This calls for the need to learn how to use Linux, from the most basic tasks to the most advanced ones.

Most people use Linux commands to perform purely basic tasks. Little do they know that the same commands can be used to perform more complex tasks as well? A good example of this is the “*cat*” command. Beyond using this to display the file contents of the terminal screen, it can also be used for output/input redirection. This book covers all Linux features through discussions and explanations of the most simple to the most advanced. Enjoy reading!

# Chapter 1- What is Linux?

Linux is an open-source operating system for computers. It was developed to provide a Unix-like free and open source operating system. The kernel is the main component of Linux operating system and Linus Torvalds released its first version in 1991. People rarely use Linux operating system on desktop computers, but it is more widely used on server computers. The reason for this is due to its high level of security. For instance, those who have worked with server computers running Linux would have realized that the operating system does not support graphics.

After booting up the computer, all you will see is a blinking cursor. You then have to provide commands to the terminal. This is of great advantage when it comes to ensuring security of the system. Note that graphics and any other additional feature added to the operating system are an advantage to the hacker, as they can gain access into the system via this feature. If you don't use the commands on the command line, then you'll definitely be stuck. However, the desktop version of Linux supports graphics.

There are various distributions of Linux, commonly known as Linux Distros. These include: Ubuntu, Khali, Red Hat, Fedora, Mint, Centos, SUSE, and others. These Distros exhibit numerous similarities although there are a few differences between them. This involves even the commands used on their terminals. For enterprise distributions of Linux such as Red Hat, a subscription fee must be paid for you to enjoy the services that they offer, even including the ability to update the OS.

The development of Linux began in 1983 when Richard Stallman thought of the idea of developing a completely free Unix-like operating system. This also marked the beginning of the Gnu project. By 1990 the essential components of this OS such as the compilers, libraries, shell and the text editors were created. Other complex components of this operating system such as drivers for devices, kernel and the daemons were developed later.

The development of the Linux Kernel began in 1991. This was after Richard Stallman attended University of Helsinki. The user interface of Linux operating systems supports either graphics or

command line. In desktop versions, both the graphical user interface and the command line are supported. The GUI is made the default, although you can still access the command line.

## Chapter 2- Linux Command Line Commands

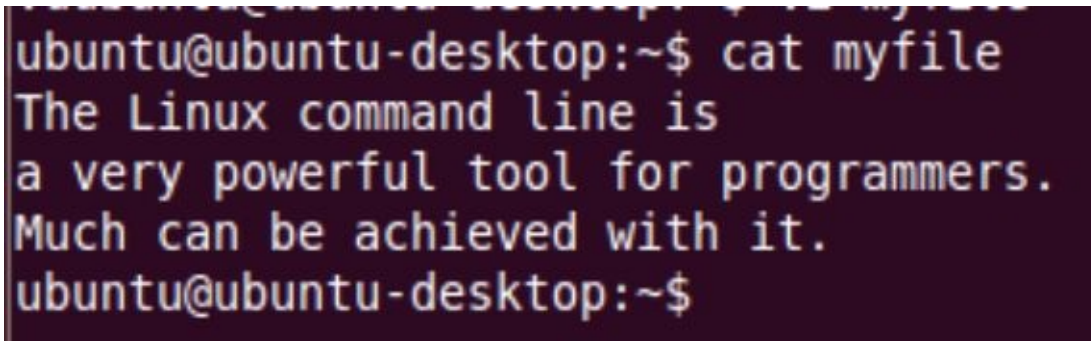
Now let's explore the Linux commands, ranging from the basic ones that are necessary for you to begin, to the complex ones that will bring your programming to a new level. To open the command line, right click on your Desktop and select "*Open Terminal*". If this is not available, find Applications at the top of your desktop. After clicking on this, select *Accessories*. Choose "*Open terminal*". Open a text editor of your choice, such as *Gedit* or *Vim*. To open the *Gedit* text editor, just search for it in the search bar. Type "*text editor*" in this search bar and you will find it. To open *Vim*, commonly known as *vi editor*, type the following command on the command line: **vi filename** Where *filename* is the name of the file. I have called my file *myfile*. On pressing the enter key, the *vi editor* will be opened. Now you can add the text of your choice to the file. Note that to change from command mode in *vi editor*, you have to press the letter *i*, otherwise, you will not be able to add any text to the file.

The text should be as follows: **The Linux command line is a very powerful tool for programmers.**

**Much can be achieved with it.**

You can then save the file to somewhere where you can easily access it, such as on the desktop. In *vi editor*, just press the *Esc* key followed by typing *wq*. This will save the file.

On the command line, type the following command and then press the enter key: **cat myfile** On pressing the enter key, the text you added on your file will be displayed on the standard output. This is illustrated in

A terminal window with a dark background and light-colored text. The prompt is 'ubuntu@ubuntu-desktop:~\$'. The command 'cat myfile' has been entered. The output of the command is displayed on the next three lines: 'The Linux command line is', 'a very powerful tool for programmers.', and 'Much can be achieved with it.'. The prompt 'ubuntu@ubuntu-desktop:~\$' is shown again on the final line.

```
ubuntu@ubuntu-desktop:~$ cat myfile
The Linux command line is
a very powerful tool for programmers.
Much can be achieved with it.
ubuntu@ubuntu-desktop:~$
```

the figure below:

This shows that the *cat* command displays the contents of a file on the standard output. However, you can also use it to display the contents of a short file.



# ***Changing Directories***

Sometimes, you might need to change your working directory. This can be achieved via the terminal using the *cd* (change directory) command as shown below: **cd /home** The command above will change the working directory to home directory and relative to root, due to the use of the forward slash, that is, /. Regardless of the current working directory, the above command will be executed. Type the following command: **cd httpd** After pressing the enter key, the full working directory will become */home/httpd*, meaning that we have changed to the *httpd* directory but relative to the */home* directory. To change back to the user's home directory, use the following command: **cd ~**

The user's home directory is the */home/username* directory, where the username will be the name you have used on the computer. Notice that we have used the symbol *~*, known as the tilde symbol. In Linux, it symbolizes the user's home directory.

## *Copying file*

In Linux, the command *cp* (copy files), is used for copying files. In copying, a duplicate of the original file is made at the location being specified, meaning that the initial file is not deleted. To make a duplicate of the file *myfile*, run the following command: **cp myfile myfile2**

The above command will create a duplicate of the file *myfile* and name it *myfile2*. The files will be similar but with a difference in their naming. However, it is risky to run the above command. This is because if the file *myfile2* already exists in the directory, it will be overwritten without a warning. To take care of this, the *-i* option should be used as shown below: **cp -i myfile myfile2**

If the file *myfile2* already exists in the directory, then we will be warned before it is overwritten, which is a very good idea.

You might need to copy all files contained in a certain directory to another directory. This can be done using the following command: **cp -dpr originaldirectory finaldirectory** The files will then be copied from the *originaldirectory* to the *finaldirectory*. They will not be deleted from the latter. The *-dpr* option is very important as it plays the following role: **-d-** for preservation of links.

**-p-** for preservation of file attributes.

**-r-** for recursive copying.

If you don't specify the above options, the default ones will be applied. This is not recommended, as you might need to preserve the links and file attributes that might not be done with the default options.

Each mounted file system uses some space on the disk. To find out the amount of this space for each file system, use the command: **df**

## ***The less command***

This command is almost similar to the “*more*” command, with the difference being that with this command you will be able to move the page up and down through the file. Consider the command below:

**less myfile** After pressing the enter key, the contents of the file *myfile* will be displayed.

## ***ln command***

To create a symbolic link to your file, use the command *ln*. Type the command shown below: **ln -s myfilemlink** The above command will create a symbolic link called *mlink* to link to the file *myfile*. To show that two files are different and contain different inodes, use the command **ln -i myfilemlink**

## ***locate command***

If you need to search in a database, use the *locate* command. Consider the command shown below:

**slocate -u** The command will create a database and name it *slocate*. The problem is that the process might take a long time, so patience will be required. But the command is important since you can't search for a file without having run it. Ever heard of cron? This is used for scheduling tasks. This will run the above command periodically so you might not need to do so. Type the command shown below and run it:

**locate whois** The command above will search for all the files contained in your system and whose names contain the string "whois".

## ***logout command***

This command just logs out the currently logged in user. Open the terminal and type the following: **logout**

Once you have typed the above command, just press the enter key. You will notice that you will be logged out of the system and then lastly you will be taken to the login screen.

# ls command

Use this command to list the files that are contained in a particular directory, *ls* means list. The command has many options associated with it, which needs to be well understood. Type the command shown below and the press the enter key: **ls** The command will list all the files that are contained in your current directory.

```
ubuntu@ubuntu-desktop:~$ ls
Desktop  Downloads  Music  Pictures  Templates
Documents  examples.desktop  myfile  Public  Videos
ubuntu@ubuntu-desktop:~$
```

The above figure shows the files contained in my directory. However, if this directory contains files whose name starts with a dot (.), they will not be listed with this command. You should also notice that with the above command, only a few details about the file are provided. If you want to know the size of the file, the day it was made and the ownership permissions associated with it, you can achieve this with the following command: **ls -al**

```
ubuntu@ubuntu-desktop:~$ ls -al
total 160
drwxr-xr-x 26 ubuntu ubuntu 4096 2015-03-30 12:39 .
drwxr-xr-x  5 root   root   4096 2015-03-21 08:34 ..
-rw-r----- 1 ubuntu ubuntu 1960 2015-03-29 15:03 .bash_history
-rw-r--r--  1 ubuntu ubuntu  220 2010-04-30 09:17 .bash_logout
-rw-r--r--  1 ubuntu ubuntu 3103 2010-04-30 09:17 .bashrc
drwx----- 4 ubuntu ubuntu 4096 2015-03-30 12:39 .cache
drwxr-xr-x  7 ubuntu ubuntu 4096 2015-03-21 05:57 .config
drwx----- 3 ubuntu ubuntu 4096 2010-04-30 09:24 .dbus
drwxr-xr-x  2 ubuntu ubuntu 4096 2015-03-29 14:07 Desktop
-rw-r--r--  1 ubuntu ubuntu   41 2015-03-30 12:38 .dmrc
drwxr-xr-x  2 ubuntu ubuntu 4096 2010-04-30 09:24 Documents
drwxr-xr-x  2 ubuntu ubuntu 4096 2010-04-30 09:24 Downloads
-rw-r----- 1 ubuntu ubuntu   16 2010-04-30 09:24 .esd_auth
drwxr-xr-x  3 ubuntu ubuntu 4096 2010-04-30 09:25 .evolution
-rw-r--r--  1 ubuntu ubuntu  179 2010-04-30 09:17 examples.desktop
drwx----- 4 ubuntu ubuntu 4096 2015-03-30 12:39 .gconf
drwx----- 2 ubuntu ubuntu 4096 2015-03-30 12:40 .gconfd
drwx----- 6 ubuntu ubuntu 4096 2010-04-30 09:59 .gnome2
drwx----- 2 ubuntu ubuntu 4096 2010-04-30 09:25 .gnome2_private
```

As you can see in the figure above, the output is more detailed. The first part starts with either a “d” or a – (dash), where “d” means that it is a directory, while “-“means that it is a file. The “rwx” stands for

permissions, where  $r$  is for read permission,  $w$  is for write permission and  $x$  means execute permission.

The size of the file and the date of creation are also shown.



## ***more command***

With this command, the contents of a file will be sent to the screen. This will be done one page at a time.

The command is also applicable on piped output.

Type and run the command shown below: **more /etc/profile** The above command will output the contents of the file */etc/profile*. This file contains details about the users of the system. On pressing the enter key, the following will be the output:

```
# /etc/profile: system-wide .profile file for the Bourne shell (sh(1))
# and Bourne compatible shells (bash(1), ksh(1), ash(1), ...).

if [ -d /etc/profile.d ]; then
  for i in /etc/profile.d/*.sh; do
    if [ -r $i ]; then
      . $i
    fi
  done
  unset i
fi

if [ "$PS1" ]; then
  if [ "$BASH" ]; then
    PS1='\u@\h:\w\$ '
    if [ -f /etc/bash.bashrc ]; then
      . /etc/bash.bashrc
    fi
  else
    if [ "`id -u`" -eq 0 ]; then
      PS1='# '
    else
```

Note that this is a built-in file so it comes with the OS itself. The file also contains too much content that cannot fit on a single page. With the above command, these will be displayed one page at a time. Type the command below and run it: **ls -al | more** The output will be the files contained in that directory. This output will then be piped through the more command. Since the output is large, it will be listed one page at a time.

## ***mv command***

The command *mv* is used for moving or renaming files; *mv* stands for move. Type the command shown below: ***mv -i myfile file*** The command will rename the file *myfile* to *file*, meaning that it will have been moved. This is the simplest way to rename files in Linux. Consider the following command: ***mv /directory1/myfile*** The above command will move the file *myfile* from the directory “*directory1*” to the current working directory. This is very simple and it shows the power of the *mv* command.

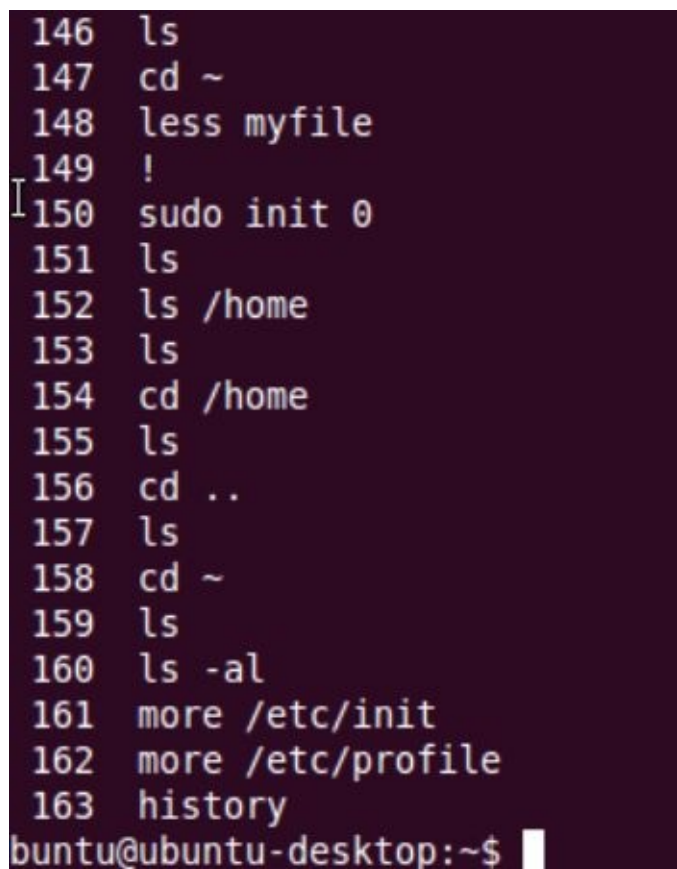
The question is, what is the ‘working directory’? This is the directory you are currently working in. To know the directory you are working in, use the following command: ***pwd*** Just press the enter key as usual. The output will be the directory you are working on. This command is useful for when you don’t know the directory you are in. *pwd* means print working directory.

## ***shutdown command***

Use this command whenever you want to shut down your system. Run the command shown below:

**shutdown -h now** On running the above command, you will notice that the system will halt immediately, that is, it will shut down. Consider the next command: **shutdown -r now** The above command shuts down the system and then boots it up again, meaning that it is used to reboot the system.

You might need to check the commands that you have run previously. The reason might be that you have changed something and now you want to undo the change. To see the list of these commands, run the

A terminal window with a dark purple background and light blue text. It displays a list of 18 commands, each preceded by a line number from 146 to 163. The commands are: 146 ls, 147 cd ~, 148 less myfile, 149 !, 150 sudo init 0, 151 ls, 152 ls /home, 153 ls, 154 cd /home, 155 ls, 156 cd .., 157 ls, 158 cd ~, 159 ls, 160 ls -al, 161 more /etc/init, 162 more /etc/profile, and 163 history. At the bottom, the prompt 'buntu@ubuntu-desktop:~\$' is visible with a white cursor.

```
146 ls
147 cd ~
148 less myfile
149 !
150 sudo init 0
151 ls
152 ls /home
153 ls
154 cd /home
155 ls
156 cd ..
157 ls
158 cd ~
159 ls
160 ls -al
161 more /etc/init
162 more /etc/profile
163 history
buntu@ubuntu-desktop:~$
```

history command as shown in the figure below:

The above figure shows the commands that I have run recently.

# ***sudo command***

*sudo* stands for *super user do*. With this command, any user can execute his or her commands as a super user. The *sudoers* file defines all this. Sensitive commands, especially those that alter the file systems of the Linux OS can only be executed using this command. The *su* command enables you to login as a super user while the *sudo* command borrows the privileges of a super user.

To update the system, you must use the *sudo* command as shown below: **sudo apt-get update** If this is the first time you have run the command, you will be prompted to enter the *sudo* password. To run the above command, you must be connected to the internet; otherwise, you will get an error. In Linux, the updates are fetched from what we call repositories. This is where the Linux development team uploads the latest updates regarding various distros of Linux.

To upgrade the system, run the following command: **sudo apt-get upgrade** The above command will upgrade the version of OS that you are using on your system. Again, you must be connected to the internet. Note that in Red Hat Linux, this command is substituted by the *yum* command, so the *sudo* command is not supported in Red Hat Linux.

***mkdir command*** *mkdir* stands for *make directory*. It is used for creating a new directory. If the folder representing the directory already exists, you will get an error informing you of this. Let me create a directory and call it *myfiles*.

***mkdir myfiles*** The above command shows that I will create a directory named *myfiles*. This is illustrated

```
ubuntu@ubuntu-desktop:~$ mkdir myfiles
ubuntu@ubuntu-desktop:~$ ls
Desktop    Downloads      Music  myfiles  Public  Videos
Documents  examples.desktop  myfile  Pictures  Templates
ubuntu@ubuntu-desktop:~$
```

in the figure shown below:

Note that after creating the directory, I then use the *list* command and the directory will be listed in the available directories. This shows that it has been created. If the above command fails to run, precede it by the *sudo* command and provide the password if prompted to do so.

## *cal command*

This command shows the month of a particular year. The year can be the current one, one from the past or even a future one. Just open the command *prompt* and type the command *cal*. Press the enter key:

```
ubuntu@ubuntu-desktop:~$ cal
      March 2015
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
ubuntu@ubuntu-desktop:~$
```

As seen, it shows the date of the day, and it is correct. Now, let us show the month of May for the year

```
ubuntu@ubuntu-desktop:~$ cal may 1980
      May 1980
Su Mo Tu We Th Fr Sa
      1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
ubuntu@ubuntu-desktop:~$
```

1980, which is a past year:

You can also show the month a future year as shown below:

```
ubuntu@ubuntu-desktop:~$ cal April 2200
```

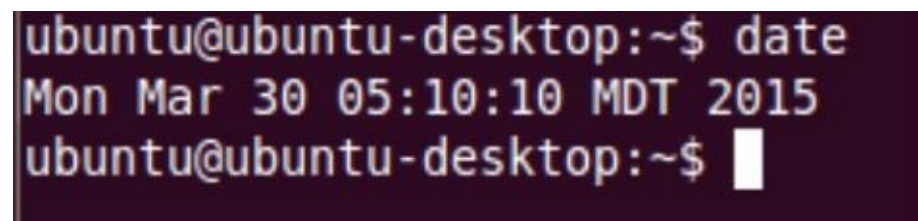
```
April 2200
```

Su	Mo	Tu	We	Th	Fr	Sa
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

```
ubuntu@ubuntu-desktop:~$
```

## *date command*

In case you need to display the current time and date, use this command as shown below:

A terminal window with a dark background. The prompt is 'ubuntu@ubuntu-desktop:~\$'. The command 'date' has been entered, and the output is 'Mon Mar 30 05:10:10 MDT 2015'. The prompt is followed by a cursor.

```
ubuntu@ubuntu-desktop:~$ date
Mon Mar 30 05:10:10 MDT 2015
ubuntu@ubuntu-desktop:~$
```

On executing the *date* command, the current date and the time will form the output, as shown in the figure above. However, you might find that the date on your system is wrong.

It is possible to set it via the command line as shown below: **date --set='1 April 2015 19:30'**

Once you have run the command shown above, the date will be set to *1<sup>st</sup> April 2015* and the time will be set to *19:50 hours*.



## ***grep command***

Sometimes, you might need to search for a pattern or a string from a certain file. In most cases, the various Distros of Linux come installed with this command.

However, if it is not already installed, you can do so using the following command: **sudo apt-get install grep** If you are using Red Hat, install it using the command: **yum install grep**

Previously we created our file and named it *myfile* with the following content:

```
ubuntu@ubuntu-desktop:~$ cat myfile
The Linux command line is
a very powerful tool for programmers.
Much can be achieved with it.
ubuntu@ubuntu-desktop:~$
```

We want to search through it using the *grep* command. To search for the word *powerful* in the file, the following approach should be used: **grep 'powerful' myfile** This is illustrated using the figure shown

below:

```
ubuntu@ubuntu-desktop:~$ grep 'powerful' myfile
a very powerful tool for programmers.
ubuntu@ubuntu-desktop:~$
```

The command outputs the line of the file with the search word and displays the search word in red. This differentiates it from the rest of the words. Note the search is case sensitive.

Let us search for the word '*Powerful*' rather than '*powerful*':

```
ubuntu@ubuntu-desktop:~$ grep 'Powerful' myfile
ubuntu@ubuntu-desktop:~$
```

As shown in the above output, there are no matches with the word '*Powerful*' in the file. This is because we have made the *p* uppercase. However, it is possible to make the search case insensitive. This can be achieved using the *-i* option as shown below:

```
ubuntu@ubuntu-desktop:~$ grep -i 'Powerful' myfile
a very powerful tool for programmers.
ubuntu@ubuntu-desktop:~$
```

The *-i* makes the words '*powerful*' and '*Powerful*' the same, hence we will get our result.

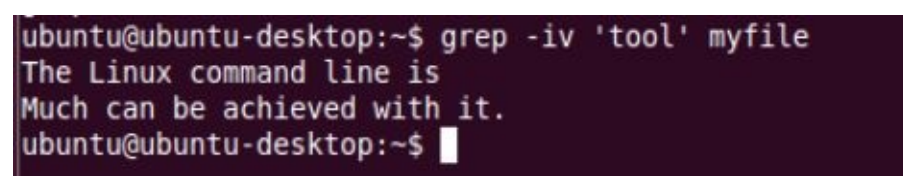
To perform a multiple search, which means that you will search for several words in a file at once, we can still use the *grep* command. This is demonstrated below.

```
ubuntu@ubuntu-desktop:~$ grep -ie 'command' -e 'powerful' myfile
The Linux command line is
a very powerful tool for programmers.
ubuntu@ubuntu-desktop:~$
```

Notice that in the figure above we are searching for two words, that is, *command* and *powerful*. The *-e*

option makes it possible for us to search for more than one word at once. Don't confuse the purpose of the *-i* option, as it only makes the search case insensitive.

The *grep* command can also be used to mean the opposite of your specification. This is illustrated below:

A terminal window with a dark background and light-colored text. The prompt is 'ubuntu@ubuntu-desktop:~\$'. The command 'grep -iv 'tool' myfile' has been entered. The output shows three lines: 'The Linux command line is', 'Much can be achieved with it.', and the prompt 'ubuntu@ubuntu-desktop:~\$' again, indicating the command has finished. The second line of the original file, 'The Linux command line is', is not present in the output.

```
ubuntu@ubuntu-desktop:~$ grep -iv 'tool' myfile
The Linux command line is
Much can be achieved with it.
ubuntu@ubuntu-desktop:~$
```

From the figure shown above, we have searched using the word '*tool*' as our search criteria. The command will only return the lines without the word '*tool*'. This explains why the second line of the file is not part of the output. The result has been achieved using the *-v* option, which returns the opposite of the search criteria. The *-i* is for case insensitive searches, so don't let that confuse you.

To find out the number of the line of the file containing the word '*powerful*', use the command shown

```
ubuntu@ubuntu-desktop:~$ grep -in 'powerful' myfile
2:a very powerful tool for programmers.
ubuntu@ubuntu-desktop:~$
```

below: **grep -in 'powerful' myfile**

Consider the command shown below and its output: **grep -iB1 'can' myfile** The command above outputs

```
ubuntu@ubuntu-desktop:~$ grep -iB1 'can' myfile
a very powerful tool for programmers.
Much can be achieved with it.
ubuntu@ubuntu-desktop:~$
```

the following:

The command outputs the exact line that is above the line with our search criteria '*can*'. This is because

we have used 1. If we used two, the output would be as follows:

```
ubuntu@ubuntu-desktop:~$ grep -iB2 'can' myfile
The Linux command line is
a very powerful tool for programmers.
Much can be achieved with it.
ubuntu@ubuntu-desktop:~$
```

## *tail* command

A very popular command in Linux is the one used to give the last part of a file. It gives only the last 10 lines of the file that you specify. If you need to get the last 10 lines of the file *myfile*, use the following command: **tail myfile** However, we have added only three lines to the file. This means that the result of the above command will be the three lines of the file. If it had more than 10 lines, then it would output only the last 10 lines. You should also specify the extension of the file name. A good example is, if it is a *c* file, add a *.c* extension to the file name.

However, the last 10 lines of the file is the default setting of the *tail* command. You have the choice of specifying the number of lines that you want to form the output.

Consider the command below and its output: **tail myfile -n 2**

```
ubuntu@ubuntu-desktop:~$ tail myfile -n 2
a very powerful tool for programmers.
Much can be achieved with it.
ubuntu@ubuntu-desktop:~$
```

As shown, we have specified that we want to output only the last 2 lines of the file *myfile*. This is what forms our output. The *-n* option, when used with this command, specifies the number of lines that will form the output.

Consider the command shown below: **tail -f myfile | grep 24.13.152.12**

Notice that we are piping the output from the *tail* command into the *grep* command. The command above can be used to monitor updates being made on the file *myfile* in real time, that is, just as they happen. The last 10 lines of the file and any new lines added to it will be piped to the *grep* command. The *grep* command then has the task of outputting these on the standard window. Note that only the lines with the above specified address, that is, *24.13.152.12* will be printed on the standard output.

The command can be used to display the last part of the file in relation to its size rather than the number of lines. The size is usually specified in bytes. To print the last 5 bytes of the file *myfile*, use the following

```
ubuntu@ubuntu-desktop:~$ tail -c5 myfile  
it.  
ubuntu@ubuntu-desktop:~$
```

command: **tail -c5 myfile**

The output implies that only the word “*it*” and the *full stop* (.) form the last 5 bytes of the file. It is also possible to combine the above option with the *plus* (+) symbol to print the bytes from a certain byte.

Consider the command shown below: **tail -c+10 myfile** The above will print the last bytes of the file,

```
ubuntu@ubuntu-desktop:~$ tail -c+10 myfile  
command line is  
a very powerful tool for programmers.  
Much can be achieved with it.  
ubuntu@ubuntu-desktop:~$
```

starting from the 10<sup>th</sup> byte. The output will be as shown below:

**wc command** *wc* stands for *wall count*. This command is used to count the lines contained in a certain file. To find out the number of lines contained in the file *myfile*, run the following command: **wc -l**

**<myfile** The command outputs the following:

```
ubuntu@ubuntu-desktop:~$ wc -l <myfile
3
ubuntu@ubuntu-desktop:~$
```

The command gives the number of lines contained in the file *myfile*, that is, only three lines. If you also need to get the name of the file, use the following command: **wc -l myfile**

```
ubuntu@ubuntu-desktop:~$ wc -l myfile
3 myfile
ubuntu@ubuntu-desktop:~$
```

Run the command shown below: **wc myfile** The following will form the output from the file:

```
ubuntu@ubuntu-desktop:~$ wc myfile
3 17 95 myfile
ubuntu@ubuntu-desktop:~$
```

The command will show the number of lines in the file, the number of words contained in the file, the size of the file in bytes, and the name of the file. These have been displayed in the order mentioned, where 3 is the number of lines in the file, 17 is the total number of words in the file, 95 is the size of the file in bytes and finally we have the file name. However, some people might easily be confused by the above.



It becomes easy when you split the above output as shown below: **wc -c myfile**— this outputs the total bytes making up the file.

**wc -w myfile**— this outputs the number of words contained in the file.

**wc -m myfile**— this outputs the number of characters making up the file.

If you can't memorize the order used in the first command, then it is advisable you use the simple alternative option above.

# *last command*

This command is useful when you want to know the users who have recently logged into your system. This is very useful in many industries for security purposes. A user won't be able to deny having logged into the system at a certain time. Open the command line and type in the command *last*. Observe the output:

```
ubuntu@ubuntu-desktop:~$ last
ubuntu pts/0 :0.0 Mon Mar 30 12:39 still logged in
ubuntu tty7 :0 Mon Mar 30 12:38 still logged in
reboot system boot 2.6.32-21-generi Mon Mar 30 12:36 - 09:10 (-3:-25)
ubuntu pts/0 :0.0 Sun Mar 29 12:55 - down (02:07)
ubuntu tty7 :0 Sun Mar 29 11:55 - down (03:07)
reboot system boot 2.6.32-21-generi Sun Mar 29 11:54 - 15:02 (03:08)
ubuntu pts/0 :0.0 Sun Mar 22 04:26 - down (1+10:47)
ubuntu pts/0 :0.0 Sun Mar 22 12:16 - 04:26 (-7:-50)
ubuntu tty7 :0 Sun Mar 22 12:15 - down (1+02:58)
reboot system boot 2.6.32-21-generi Sun Mar 22 12:15 - 15:13 (1+02:58)
ubuntu pts/0 :0.0 Sat Mar 21 12:59 - down (02:14)
ubuntu pts/0 :0.0 Sat Mar 21 05:53 - 12:58 (07:05)
ubuntu tty7 :0 Sat Mar 21 05:47 - down (09:25)

wtmp begins Sat Mar 21 05:47:34 2015
ubuntu@ubuntu-desktop:~$
```

The command shows the name of the user and other details including the terminal they were logged on and the time in which they logged in. If any of the users are still logged in, you will be notified of it. The difference between when the user was last logged in and logged out will also be shown. Note that only the last 100 logins will be displayed.

The command is also associated with the following options:

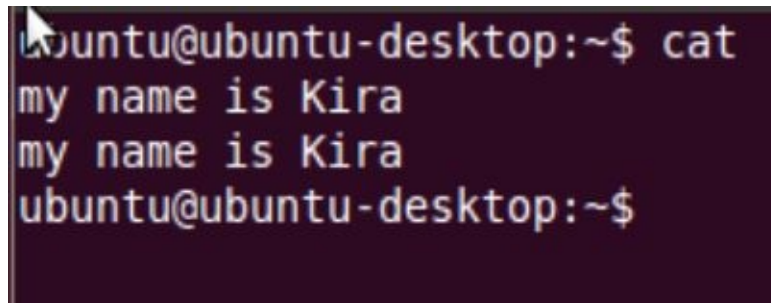
1. **-R** – this will make sure that the host name is not displayed.
2. **-d** – this is used when doing remote login. The *ip* address that identifies the host name is translated into the hostname itself.
3. **-a** – this will display the name of the host in the output's last column.
4. **-x** – this shows the shutdown history and how changes have been made on the run levels.
5. **-i** – displays the *ip* address of the host login in remotely. The *ip* address is usually in dots and number format.
6. **-F** – the times and dates of full logins and logouts into the system ware are displayed.

7. **-w** – the names of the user and the domain name are displayed as the output.

**Chapter 3- Input/output Redirection** Most processes in Linux require standard input, which is using the keyboard and displaying it on a standard output, a terminal screen. Error messages for these processes are written in standard error, which by default is set to the terminal screen.

You already know how to use the “*cat*” command to display the contents of a file on the terminal screen.

To demonstrate another use of this command, try the following: Open the terminal and type “*cat*”. Press the “*Enter*” key. You can then type anything that you want on the terminal and then press the “*Return*” key. Observe what happens.

A terminal window with a dark background and light-colored text. The prompt is 'ubuntu@ubuntu-desktop:~\$'. The user has entered 'cat' and pressed enter. The terminal now shows 'my name is Kira' on two separate lines, indicating that the user typed it and then pressed enter twice. The prompt 'ubuntu@ubuntu-desktop:~\$' is now visible again.

```
ubuntu@ubuntu-desktop:~$ cat
my name is Kira
my name is Kira
ubuntu@ubuntu-desktop:~$
```

As shown in the figure above, the “*cat*” command can also be used to read input from the standard input, or keyboard. To do this, type the command on the terminal without specifying the name of the file. After writing your input, press the “*Return*” key and this will be displayed on the terminal screen. If you need to end, press “*Ctrl + D*”, usually written as “*^D*”.

This shows that it is possible to redirect both the input and the output in Linux.

# ***Output Redirection in Linux***

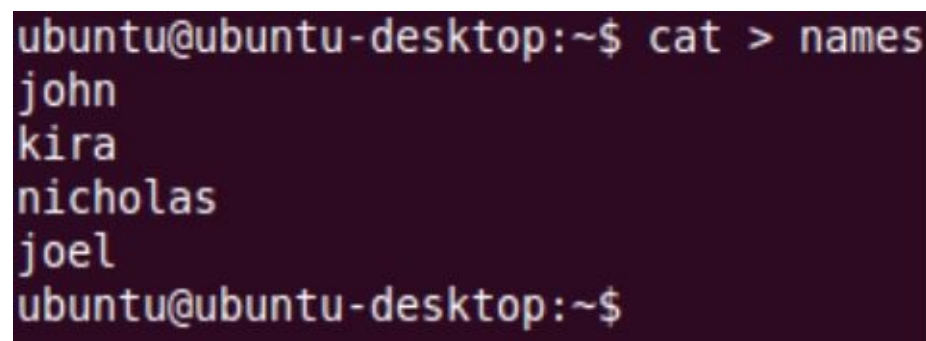
It is possible to redirect output in Linux. We use the ‘greater than’ symbol for this purpose, that is, “>”.

We will then be able to redirect our output.

Let’s create a file named “*names*”, containing a list of names.

Open the terminal and type the following command: **cat > names** You can then type the list of names.

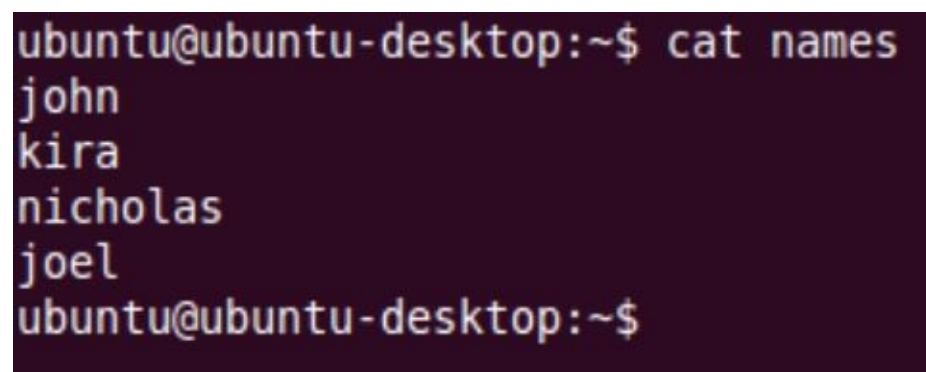
Make sure that you press the “*Return*” key after each name:



```
ubuntu@ubuntu-desktop:~$ cat > names
john
kira
nicholas
joel
ubuntu@ubuntu-desktop:~$
```

The figure above demonstrates another use of the “*cat*” command.

Now run the following command: **cat names** The above command will give the following output:



```
ubuntu@ubuntu-desktop:~$ cat names
john
kira
nicholas
joel
ubuntu@ubuntu-desktop:~$
```

As you can see, the command created a file named “*names*” and then added the names we typed to the file. What happens is that the “*cat*” command reads the input from the standard input and the redirection symbol, that is, “>” redirects the input to the file “*names*”. Again, to end the input, just press “*Ctrl + D*”.

# Appending

To append the standard input to a file in Linux, use the symbol “>>”. To add more names to the file “names” you have just created, do the following: Open the terminal and run the following command: **cat >> names** After typing the above command, press the “Enter” key and then type in the list of names to be added to the file. This is demonstrated in the diagram shown below:

```
ubuntu@ubuntu-desktop:~$ cat >> names
mercy
aladin
clifftone
ubuntu@ubuntu-desktop:~$
```

Open the file to check whether the names have been added to the file. This is demonstrated below:

```
ubuntu@ubuntu-desktop:~$ cat names
john
kira
nicholas
joel
mercy
aladin
clifftone
ubuntu@ubuntu-desktop:~$
```

The above figure clearly shows that the new names have been added to the file. It also depicts how easy it is to add new contents to an already existing file in Linux. Note that to end the input you simply need to press “Ctrl + D”, as we have done before.

It is possible to combine the contents of two files into one file in Linux. Create a new file called “names2” and add some names to it. This is demonstrated below:

```
ubuntu@ubuntu-desktop:~$ cat > names2
caleb
andrew
ubuntu@ubuntu-desktop:~$ cat names2
caleb
andrew
ubuntu@ubuntu-desktop:~$
```

From the above figure, we have created our file and added two names to it. Our aim is to combine the two names in the file with the names in our first file, “*names*” into a single file named “*list*”. Just run the command shown below: **cat names names2 > list** This is demonstrated below:

```
ubuntu@ubuntu-desktop:~$ cat names names2 > list
```

After executing the above command, just check the contents of the file “*list*”. This is demonstrated below:

```
ubuntu@ubuntu-desktop:~$ cat list
john
kira
nicholas
joel
mercy
aladin
clifftone
caleb
andrew
ubuntu@ubuntu-desktop:~$
```

As shown in the figure above we have added the two names in the file “*names2*” to the names in our old file “*names*”. This gives a longer list of names. It is now clear that we can use the redirection symbol “>” to append the contents of one file to another file.

# ***Input Redirection***

If you need to redirect the input of a command, use the less than symbol, that is, “<”. To sort the input, use the “*sort*” command. Let’s demonstrate this.

Open the terminal and type in the following command: **sort** Press the enter key and then type in some words starting with different letters of the alphabet. Press the “*Return*” key after each word. Once done, press “*Ctrl + D*” and observe the output. This is demonstrated in the figure shown below:

```
ubuntu@ubuntu-desktop:~$ sort
art
fellow
dog
zebra
water
```

After typing the above words end the input by pressing the key combination above.

```
art
dog
fellow
water
zebra
ubuntu@ubuntu-desktop:~$
```

The following output should be observed:

Beyond using the keyboard to enter the input, it is possible to specify input from certain files using the same symbol. Type the following command: **sort < list** The above command will give the following



```
ubuntu@ubuntu-desktop:~$ sort < list
aladin
andrew
caleb
clifftone
joel
john
kira
mercy
nicholas
ubuntu@ubuntu-desktop:~$
```

output:

If you want the sorted contents of a file to be outputted in another file, do the following:

```
ubuntu@ubuntu-desktop:~$ sort <list> file
```

In the command shown in the above figure, we are sorting the contents of the file “*list*” and the output will be outputted in the file “*file*”. You can then use the “*cat*” command to view the contents of the file as

```
ubuntu@ubuntu-desktop:~$ cat file
aladin
andrew
caleb
clifftone
joel
john
kira
mercy
nicholas
ubuntu@ubuntu-desktop:~$
```

shown below:

# ***Piping***

Piping means that the output from a certain process is made into the input of another process. In Linux it is represented by the symbol “|”, which is the vertical bar. Consider the command shown below: **who | sort**

In the above command, we are redirecting the output of the “*who*” command to the “*sort*” command. This is a quick method. If you want to know the number of users currently logged into the system, run the following command: **who | wc -l**





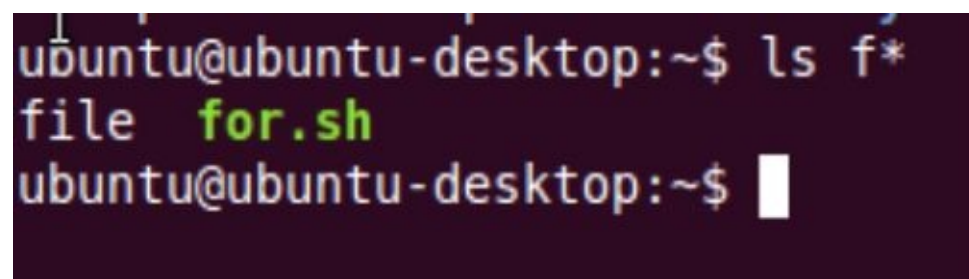
# Chapter 4- Wildcards in Linux

There are numerous wildcards in Linux. Let's examine them.

## *The “\*” wildcard*

This is used to match one or more characters in a specified file. When you want to know the list of files in the current directory whose name start with the letter “f”, run the following command: **ls f\***

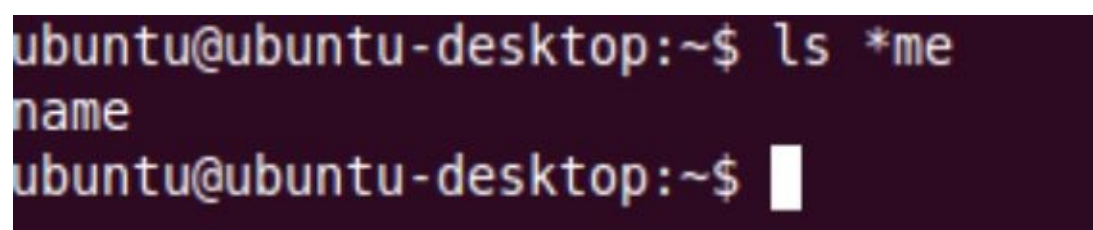
The above command is demonstrated in the figure shown below:



```
ubuntu@ubuntu-desktop:~$ ls f*
file  for.sh
ubuntu@ubuntu-desktop:~$
```

From the figure shown, it is clear that I only have two files whose name start with the letter “f” in the above directory. You can also use the wildcard to find out the files whose names end with a certain pattern. For example: To know all the files whose name end with ...me, run the following command: **ls**

**\*me** On my computer, the above command gives the following output:



```
ubuntu@ubuntu-desktop:~$ ls *me
name
ubuntu@ubuntu-desktop:~$
```

From the above figure it is very clear that only one file in the directory obeys the pattern specified.

## *The “?” wildcard*

In Linux the above wildcard is used to match exactly one character. Run the following command: **ls f?le**

The above command gives the following output in my computer:

```
ubuntu@ubuntu-desktop:~$ ls f?le
file
ubuntu@ubuntu-desktop:~$
```

## ***Getting Help***

It is also possible to get help in Linux. If you want to know more about a certain command, you can consult the manual files. To find out more about the command “*ls*”, run the following command: **man ls**. The above command will tell you more about the “*ls*” command and the options associated with it. If you want a short description of the command, run the following command: **whatis ls**. With the above command, the options associated with the command “*ls*” will not be displayed, so use it only when you are not interested in the options.

If you are not sure how to write a certain command in Linux, use the command “*apropos*” to get help. For example: To find out the exact name of the “*copy*” command, type the following: **apropos copy**

# Chapter 5- File security in Linux

Once you have created a long list of your files using the “ls” command, the permissions associated with each file will also be shown in the output. This is demonstrated below.

After running the command “ls -l” on my system, I get the following output:

```
drwxr-xr-x 2 ubuntu ubuntu 4096 2015-03-29 14:07 Desktop
drwxr-xr-x 2 ubuntu ubuntu 4096 2010-04-30 09:24 Documents
drwxr-xr-x 2 ubuntu ubuntu 4096 2010-04-30 09:24 Downloads
-rw-r--r-- 1 ubuntu ubuntu 179 2010-04-30 09:17 examples.desktop
-rw-r--r-- 1 ubuntu ubuntu 60 2015-04-29 06:17 file
-rwxr--r-- 1 ubuntu ubuntu 58 2015-03-31 07:37 for.sh
-rwxr--r-- 1 ubuntu ubuntu 31 2015-04-02 12:32 hello.sh
-rw-r--r-- 1 ubuntu ubuntu 60 2015-04-29 03:56 list
-rwxr--r-- 1 ubuntu ubuntu 143 2015-03-31 02:36 list.sh
drwxr-xr-x 2 ubuntu ubuntu 4096 2010-04-30 09:24 Music
-rw-r--r-- 1 ubuntu ubuntu 95 2015-03-29 12:56 myfile
drwxr-xr-x 2 ubuntu ubuntu 4096 2015-03-30 14:16 myfiles
-rw-r--r-- 1 ubuntu ubuntu 309 2015-03-31 02:51 name
```

The output shows all files in the directory and the permissions associated with each file.

## Manipulating access rights on a file

### chmod (change mode)

Only the owner of a file is allowed to change its mode. The following options are associated with the “chmod” command:

u	-user
g	-group
o	- other
a	-all
r	-read
w	-write (and delete)

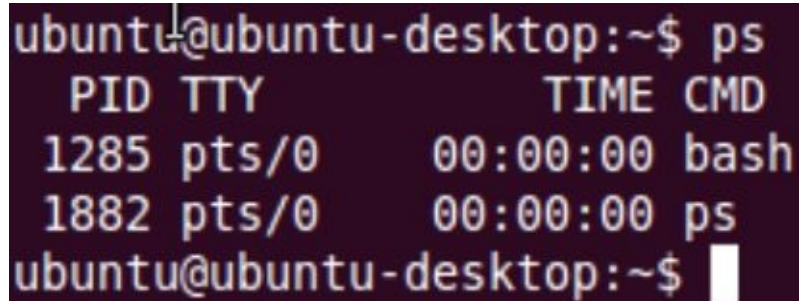


x	-execute (and access directory)
+	-add permission
-	-delete permission

If you need the read, write and execute permissions for a group or others from the file “*names*”, then run the following command: **chmod go-rwx names** After running the above command, the group and others will have no read, write and execute permissions on the file “*names*”. Other permissions on the file will not be affected. To assign a read and write permission to all on the file “*names*”, run the following command: **chmod a+rw names**

# Chapter 6- Jobs and Processes

A process is a program in execution and it is uniquely identified by a process identifier, or PID. If you want to know the processes that are currently running on your system, together with their status and associated process IDs, run the following command: **ps** On my system, the above command outputs the

A terminal window with a dark background and light-colored text. The prompt is 'ubuntu@ubuntu-desktop:~\$'. The command 'ps' has been entered. The output shows a table with four columns: PID, TTY, TIME, and CMD. There are two rows of data: one for PID 1285, TTY pts/0, TIME 00:00:00, CMD bash; and another for PID 1882, TTY pts/0, TIME 00:00:00, CMD ps. The prompt 'ubuntu@ubuntu-desktop:~\$' is shown again with a cursor at the end.

```
ubuntu@ubuntu-desktop:~$ ps
  PID TTY          TIME CMD
 1285 pts/0    00:00:00 bash
 1882 pts/0    00:00:00 ps
ubuntu@ubuntu-desktop:~$
```

following:

Linux processes can run either in the foreground or background. They can also be suspended. Once a process starts and is running you will only be able to interact with the command prompt once the process has finished executing. If the process takes a long time to run you will be unable to interact with the command prompt.

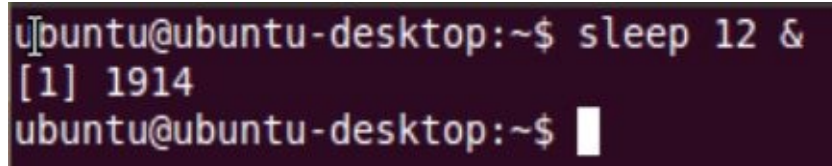
It is therefore advisable to run these processes in the background so that you can continue to interact with the command prompt.

***Running Processes in the Background*** In Linux, the use of the “and” symbol, or “&” is used to run processes in the background. If the process takes a long time to run the command prompt will be immediately returned to you. To demonstrate this, use the “*sleep*” command, which causes your computer to wait for a number of seconds before continuing with processing.

Type the following command: **sleep 12**

After typing, press the “*Enter*” key. You will notice the inability to interact with the command prompt until 12 seconds are over.

To run the above process in the background, run it as follows: **sleep 12 &** You will notice that after running the command the prompt will be immediately returned. This is because the long running process is being run in the background. When the background process finishes, the user will be notified. The job number and the PID of the process are also returned to the user by the machine.



```
ubuntu@ubuntu-desktop:~$ sleep 12 &
[1] 1914
ubuntu@ubuntu-desktop:~$
```

This is shown below:

The first number, enclosed in square brackets, is the job number followed by the process ID, or the PID.

You might forget to run a long-running process in the background. This will make it impossible for you to interact with the command prompt. Instead of cancelling the process, it is still possible to remove it from foreground and then run it in the background. This is demonstrated below.

Open the command prompt and run the following command: **sleep 500**

The above command will cause the computer to stop execution for a period of 500 seconds. Now that you cannot interact with the prompt, you need to place the process into background. Press “*Ctrl + Z*”. This will suspend the process, meaning that it will stop running.

On the terminal, type the following command: **bg** The above command will then take the process to the

```
ubuntu@ubuntu-desktop:~$ sleep 500
^Z[1]  Done                  sleep 12

[2]+  Stopped                  sleep 500
ubuntu@ubuntu-desktop:~$ bg
[2]+  sleep 500 &
ubuntu@ubuntu-desktop:~$
```

background. This is demonstrated below:

After suspending or taking a process into the background, they are entered into a list along with their job id.

To see a process that is currently suspended or running in the background, run the following command:

**jobs** On my system the above command gives the following output:

```
ubuntu@ubuntu-desktop:~$ jobs
[2]+  Running                  sleep 500 &
ubuntu@ubuntu-desktop:~$
```

This shows that there is one process running in the background. Notice that the *job id* in the background is also listed in square brackets.

To bring a suspended process, or one running in the background, to the foreground, run the following command: **fg %2**

The *job id* of the process running in the background on the system is 2. That explains the 2 used in the above command.

In Linux it is possible to kill certain jobs. A good example of this is an infinite loop. If a process is running in the foreground and you want to kill it, press “*Ctrl + C*”. This will kill it as demonstrated below: Open the terminal and type in the following command: **sleep 200**

Press the enter key. You will notice that you can’t interact with the command prompt, since the process is a long-running one. Press “*Ctrl + C*”. Observe what happens. You will have the command prompt back.

This shows that the process has been killed as demonstrated in the figure below:

```
ubuntu@ubuntu-desktop:~$ sleep 200
^C
ubuntu@ubuntu-desktop:~$
```

If you have already suspended the process, or if you are running it in the background, kill it using the following command: **kill %*jobnumber*** The “*jobnumber*” is the job number of the process suspended, or running in the background.

This is demonstrated in the following figure:

```
ubuntu@ubuntu-desktop:~$ sleep 200
^Z[1]  Done                  sleep 500

[2]+  Stopped                  sleep 200
ubuntu@ubuntu-desktop:~$ bg
[2]+ sleep 200 &
ubuntu@ubuntu-desktop:~$ jobs
[2]+  Running                  sleep 200 &
ubuntu@ubuntu-desktop:~$ kill %2
```

The figure shows that after running the process you can take it to the background and kill it. You can then run the “*jobs*” command to check whether it has happened.

Another way to kill a process is by looking for its PID and then using it to complete the task. For the PID of a process, use the command “*ps*”. Make use of this to kill the process. This is demonstrated below:

```
ubuntu@ubuntu-desktop:~$ sleep 200 &
[1] 2049
ubuntu@ubuntu-desktop:~$ ps
  PID TTY          TIME CMD
 1898 pts/0    00:00:00 bash
 2049 pts/0    00:00:00 sleep
 2050 pts/0    00:00:00 ps
ubuntu@ubuntu-desktop:~$ kill 2049
[1]+  Terminated          sleep 200
ubuntu@ubuntu-desktop:~$
```

As shown in the above figure, run a long-running process in the background. Look for its PID and kill it using its PID. After the procedure above, it is a good idea to check whether the process has been killed using the “*ps*” command.

If it has not been killed, force this using the -9 option as shown below: **kill -9 2049**

# Useful Linux commands

Each user on a Linux system is given a certain amount of space to store files. If this is exceeded, seven days are given to the user to get rid of the excess files. The space is usually located on the hard disk and is always about 100MB.

To find out the amount of quota used and the amount that is remaining, run the following command: **quota**

–v If you need to know the amount of space being left on the file system, use the “df” command.

To find out the amount of space that is remaining on the file server, run the following command: **df .**

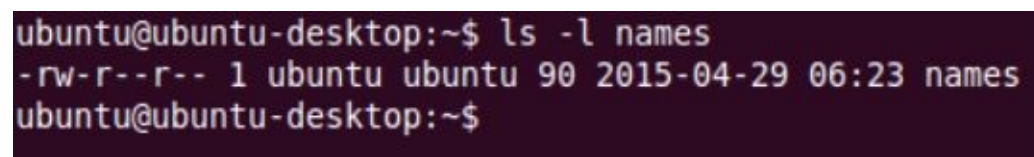
To find out the number of kilobytes that have been used by each subdirectory, use the “du” command. If you have overused your quarter, you might need to know the subdirectory with the most files.

Navigate to your home directory and run the following command: **du -s \***

With the “-s” option, the output will be summarized and with the “\*”. All the files and directories will be listed.

You might need to use less disk space. To minimize the size of a file to free up some disk space, run the following command: **ls -l names** After running the above command, keenly note the size of the file names.

Mine are shown in the following figure:



```
ubuntu@ubuntu-desktop:~$ ls -l names
-rw-r--r-- 1 ubuntu ubuntu 90 2015-04-29 06:23 names
ubuntu@ubuntu-desktop:~$
```

The size of the file “names” is 90 kilobytes, as shown in the figure above. To minimize size, run the following command: **gzip names** You can run the “ls” command. You will notice that the file will be renamed to “namez.gz”. Run the following command to check on its current size.

This is demonstrated in the figure shown below:



```
ubuntu@ubuntu-desktop:~$ ls -l names.gz
-rw-r--r-- 1 ubuntu ubuntu 79 2015-04-29 06:23 names.gz
ubuntu@ubuntu-desktop:~$
```

As shown in the figure above, the size of the file has been reduced to 79 kilobytes. To expand the file to its original size, use the “*gunzip*” command as shown below:

```
ubuntu@ubuntu-desktop:~$ gunzip names.gz
ubuntu@ubuntu-desktop:~$ ls -l names
-rw-r--r-- 1 ubuntu ubuntu 90 2015-04-29 06:23 names
ubuntu@ubuntu-desktop:~$
```

Zip the file using the command above. It is tricky when it comes to reading the file without having to undo the compression. It is possible using the “*zcat*” command.

To read the contents of the file “*names*”, run the following command: **zcat names.gz** In case the text scrolls on the screen too fast, pipe it through the “*less*” command as shown below: **zcat science.txt.gz | less** You can also use the “*file*” command to classify the named files according to the type of data that they contain. The data can be pictures, ascii (text) or compressed data.

To classify all the files contained in the current directory, run the following command: **file \***

You might also need to compare two files and get the differences between them.

To do this, use the “*diff*” command as shown below: **diff names names2**

The above command compares the contents of the files “*names*” and “*names2*” and then outputs the differences between the two files.

The following output will be observed after running the above command:

```
ubuntu@ubuntu-desktop:~$ diff names names2
1,2c1,2
< ubuntu      tty7          2015-04-29 02:51 (:0)
< ubuntu      pts/0         2015-04-29 02:59 (:0.0)
---
> caleb
> andrew
ubuntu@ubuntu-desktop:~$
```

For the lines that begin with the symbol “<”, this denotes the first file which is “*names*”, while for the

lines beginning with “>”, this denotes the second file which is “*names2*”.

With the “*find*” command you can search for files with specification of the desired attributes such as name, size and date. The command also has many options, so you can learn about them from the main pages.

If you want to find the files whose name ends with a “.*sh*” extension in the current directory through all the sub-directories and then print on the screen, run the following command: **find . -name "\*.sh" -print** You might also need to search for files using the size as the search criteria.

This can be achieved as follows: **find . -size +2M -ls** The above command will search for all files in the current directory whose size is more than 2MB. These files will form the output.



## Chapter 7- Bash scripting Tricks

*Bash* stands for *Bourne Again shell*, and in most Linux distros, it forms the default shell. Many Unix and Linux users really like it due to its user-friendliness. In this chapter, we are going to learn about the Bash shell.

## Brace Expansion

A list of strings is used as arguments in this case. These strings are separated using commas. There should be no space after the comma. Example: **echo {John,Julius,Jacob}**

```
ubuntu@ubuntu-desktop:~$ echo {John,Julius,Jacob}
John Julius Jacob
ubuntu@ubuntu-desktop:~$
```

From the figure, you can see the output, showing just the arguments we provided in the braces.

Now, type the following on the command line: **echo John,Julius,Jacob** The output will just be the same as in the first example above. You might not be able to see how the braces can be used. Consider the example shown below: **echo {John,Julius,Jacob}David** Maybe the three are David's sons. Observe the

output:

```
ubuntu@ubuntu-desktop:~$ echo {John,Julius,Jacob}David
JohnDavid JuliusDavid JacobDavid
ubuntu@ubuntu-desktop:~$
```

You can now see how the braces are used in bash scripting. They are very useful when the list placed in the braces should occur after, inside or before another string. In the case above, these should occur before the string "David".

The above can be implemented in another way, as follows: **echo David{John,Julius,Jacob}**

The following will form the output of the above script:

```
ubuntu@ubuntu-desktop:~$ echo David{John,Julius,Jacob}
DavidJohn DavidJulius DavidJacob
ubuntu@ubuntu-desktop:~$
```

As you can see, the list specified inside the braces occurs after the string "David".

It is also possible to place the list in between two strings. This is illustrated below: **echo David{John,Julius,Jacob}Mercy** The output will be as follows:

```
ubuntu@ubuntu-desktop:~$ echo David{John,Julius,Jacob}Mercy
DavidJohnMercy DavidJuliusMercy DavidJacobMercy
ubuntu@ubuntu-desktop:~$
```

The words in the list have been sandwiched between the two strings that we have specified. Again, make sure that you don't include any spaces. There are no spaces between the arguments, nor between the braces and the strings. Including this might lead to an error or an undesired output.

However, if you need to use a space, make use that you also use double quotes as shown below: **echo "{John,Julius,Jacob} David"**

The following will be the output of the above:

```
ubuntu@ubuntu-desktop:~$ echo "{John,Julius,Jacob} David"
{John,Julius,Jacob} David
ubuntu@ubuntu-desktop:~$
```

You might need an output like that, so that's how to get it. Consider the example given below: **echo {"John ", "Julius ", "Jacob "} David** The above will give the following as the output:

```
ubuntu@ubuntu-desktop:~$ echo {"John ", "Julius ", "Jacob "}David
John David Julius David Jacob David
ubuntu@ubuntu-desktop:~$
```

Braces can also be nested, whereby braces are placed within other braces as shown below: **echo {{a,b,c},a,b,c}**

The following will be the output of the above example:

```
ubuntu@ubuntu-desktop:~$ echo {{a,b,c},a,b,c}
a b c a b c
ubuntu@ubuntu-desktop:~$
```

Consider the example shown below: **echo {{6,7,8}1,2,3}** The output will be as follows:

```
ubuntu@ubuntu-desktop:~$ echo {{6,7,8}1,2,3}
61 71 81 2 3
ubuntu@ubuntu-desktop:~$
```

Where this trick of brace expansion should be applied? Of course, when making a backup of a file, you will need to use this trick, hence it is very important. An example of this is given below: **cp /etc/httpd/conf/httpd.conf{,.backup}**

In the above example, we are backing up the *httpd.conf* file.

# Command Substitution

This is also another useful bash scripting trick. It includes the use of the \$ sign and parenthesis on the standard output to enclose any command that is aimed at generating output. This trick is applied when assigning some value to a variable.

You can assign the date to a variable as follows: **date +%d-%b-%Y**

The above will give you the current date as shown below:

```
ubuntu@ubuntu-desktop:~$ date +%d-%b-%Y
30-Mar-2015
ubuntu@ubuntu-desktop:~$
```

This can also be achieved using the following: **echo \$today today=\$(date +%d-%b-%Y)** The output will just be the same. To get information about multiple *RPM* packages simultaneously, you can also use command substitution.

Command substitution can also be achieved after surrounding a command using back quotes. This can be illustrated using the example shown below: **day=`date +%d-%b-%Y`**

**echo \$day** The above will give the following as the final result:

```
ubuntu@ubuntu-desktop:~$ day=`date +%d-%b-%Y`
ubuntu@ubuntu-desktop:~$ echo $day
30-Mar-2015
ubuntu@ubuntu-desktop:~$
```

The current date will be displayed. With the above style, commands can easily be nested and they will become easy to read.

The output from a certain command can also be directed to a certain file. This can be done as shown below: **find / -name file > result.txt** In the above example, we are redirecting the output from the command *find* to the file *result.txt*.

However, you will notice that error messages will be produced. If you are not interested in these, you can also redirect them to a file. A 2 in front of the redirection symbol can assist in this.

They can also be redirected to */dev/null* if we are not interested in them at all. This is illustrated below:

**find / -name file 2> /dev/null** The location of the file “*file*” will be shown without the error messages; 2 stands for the standard error output stream. Most Linux commands redirect their errors here, whereas the correct output that is sent to the standard output is represented using 1.

Consider the commands shown below: **find / -name file > result.txt** **find / -name file 1> result.txt** You might think that the above commands are different. However, they are the same. The 1 represents the normal standard output.

You may need to save both, that is, the error message and the standard output to a file. Cron jobs are usually used for this purpose. This can be achieved as shown below: **find / -name file > result.txt 2> result.txt** Note that both output streams have been directed to the same file. However, doing it this way can be tiresome. It can be achieved with greater simplicity, as shown below: **find / -name file > result.txt 2>&1**

In the example above, we have used an ampersand to tie the standard error stream and the standard output stream. After this, the standard error will go wherever the standard output goes. Again pay attention to the syntax used in the example above, as it is very important.

You might also be interested in piping the output to another command. This is shown below: **find -name file.sh 2>&1 | d /tmp/result2.txt** The above line will work just as is expected. Now consider the line shown below: **find -name file.sh | d /tmp/result2.txt 2>&1**

On running the above line, you will notice that it won’t work as is expected. This is in contradiction to the other line.

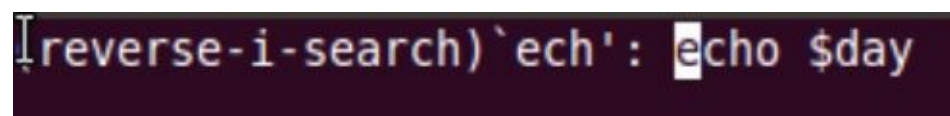
# Command History

With Bash, you can search through the commands that have been run recently. You can then use the up and down arrow keys to navigate through these commands. If the command you are looking for was executed between the last 10-20 executed commands, then this will be easy for you.

If you want to search interactively, meaning that suggestions for what you are searching for will be provided, just press *Ctrl-R*.

The prompt will then be changed to the following: **(reverse-i-search)`**: You can then start typing the letters of your command. You will notice that you will be provided with suggestions. However, the suggestions will only include the recently typed and related commands.

In my case, after pressing the *Ctrl-R*, and then typing *ech*, the following is the suggestion:

A terminal window with a dark background. The prompt is (reverse-i-search)`ech': and the suggestion is echo \$day. The text is white with a light blue highlight on the word echo.

```
(reverse-i-search)`ech': echo $day
```

The above shows that the command I ran recently containing the words *ech* is the “*echo day*” command.

To execute the command, just press the enter key and it will be executed. If you need to do some editing to the command, just press the left or right arrow keys. The command will be placed on the normal prompt and then you will be able to edit it.

# ***Loops***

With *Bash*, it is possible to create loops on the command line. However, this is not suitable if the code is large or too complex. On the command line, loops can be written in two different ways. The first way involves separating the different lines of code using a semicolon. To create a loop that will back up all the files contained in a certain directory, use this method: **for doc in \*; do cp \$doc \$doc.bak; done** The above line of code will back up all the files in the directory. Rather than separating the lines using a semicolon, a secondary prompt can be used. This can be done by pressing the enter key once you are through with each single line. Once you have pressed the enter key, Bash will understand that you want to create a loop and it will provide you with a secondary prompt.

You can then enter your second line. This is illustrated below: **for doc in \***

**> do cp \$doc \$doc.bak > done** Sometimes, your Linux system can run out of memory. This means that you will not be able to execute your commands including the simple ones such as *ls* and the ones for changing directories. If *ls* fails and you want to view the files contained in a particular directory, then this can be achieved as follows: **for doc in \*; do echo \$doc; done** This will solve the *ls* problem. This shows the power of bash scripting.

# Chapter 8- Linux shell programming

In Linux, it is very easy to create shell scripts. With shell programming, we can group multiple commands in a chain and then execute them to obtain the desired output. Let's begin by writing out first script. Open your editor of choice; I am using the *vi editor*. Add the following shell program to the file: **#!/bin/bash**

## Echo "Hello World"

The above is an example of a shell program. Save the file and give it the name `hello.sh`. Note that we have used the `.sh` extension to imply that it is a shell program. Now, the next step is to make the script executable. This can be achieved as follows: Open the command prompt and type the following command: **chmod 744 hello.sh** After running the above command, an executable of the file `hello.sh` will be created; *chmod* stands for change mode.

`744` refers to the permissions and we have used this to add an execute permission to the file. Since we have made the program executable, we now need to execute it.

This can be achieved by running the following command on the terminal: **./hello.sh** After running the above command, the following output will be observed on the standard output:

```
ubuntu@ubuntu-desktop:~$ chmod 744 hello.sh
ubuntu@ubuntu-desktop:~$ ./hello.sh
Hello World
ubuntu@ubuntu-desktop:~$ █
```

The command outputs a text saying "*Hello World*". Remember that this is the text that we command the `echo` command to output.

**#!/bin/bash** is called the *shebang* and is used to precede shell programs.

To add comments to your program, precede them with a pound sign (`#`).

Consider the shell program shown below: **#!/bin/bash echo "Hey \$USER"**

**echo "Hello, i am" \$USER "The following are the processes running currently"**

**echo "List of running processes:"**

**ps** You can write and run the program. The following output will be observed after running the program:



```
Hey ubuntu
Hello, i am ubuntu The following are the processes running currently
List of running processes:
  PID TTY          TIME CMD
 1400 pts/0    00:00:00 bash
 1928 pts/0    00:00:00 list.sh
 1929 pts/0    00:00:00 ps
```

As shown in the above figure, the program outputs your name. It also shows the list of processes that are currently running on your system.

Now, let's write a more interactive program. Write the code shown below and run it: **#!/bin/bash echo "Enter your name"**

```
read name; echo "welcome Mr. /Mrs. $name "; echo "Thanks Mr./Mrs. $name. You have provided
us with your names"; echo "-----"
echo "Mr. /Mrs. $name, good bye friend"
```

After running the above program, the following output will be observed:

```
Enter your name
Nicholas
Welcome Mr./Mrs. Nicholas
Thanks Mr./Mrs. Nicholas. You have provided us with your name.
-----
Mr./Mrs. Nicholas, good bye friend
ubuntu@ubuntu-desktop:~$
```

As you can see, you are prompted to provide your name. This has been achieved by use of the read command, which prompts users to provide their input. This input has then been displayed alongside some other text.

***Variables in Shell*** The two types of shell variables are:

1. System variables- these are created and maintained by the OS. They are defined in upper case letters.
2. User Defined Variables- the user creates and maintains these. They are defined in lower case letters.

The following are some of the system variables in shell: BASH- the shell name.

HOME- our home directory LOGNAME- our login name.

PATH- our setting for the path.

USERNAME- the currently logged in user.

COLUMNS- number of columns making up our screen.

OSTYPE- the type of the OS we are running.

PWD- the current working directory.

The above shows only a sample of the system variables. If you want to print any of the above variables, use the echo and the \$ sign command as shown below: **echo \$HOME**

**echo \$PWD**

The next section explains how to work with user defined variables.

Use the following syntax to define a user defined variable: **variable name=value** The 'name' refers to the name to be used to refer to the variable. The value is what it is. This is usually a number or a string.

Example: **num=10**

**name=Mary**

## *Tips when naming variable*

1. There shouldn't be any spaces on either side of the equal sign. Example:

**num=12- this is right num= 12- this is wrong num =12- this is wrong num = 12- this is wrong**

2. The name of the variable should begin with an underscore or any of the alphanumeric characters. This should then be followed by any of the alphanumeric characters. The following are examples of valid variables:

**PWD**

**SYSTEM\_VERSION**

**num**

3. The names of variables are case sensitive. Example:

**Num=2**

**num=6**

**nUm=9**

**nuM=10**

**NUM=90**

The above shows that the above variable names are not equal. To print any of the above, also make sure that you specify these correctly. For example, to print 10, use the following code: **echo**

**\$nuM**

4. To define null variables, do the following:

**num=**

**num=""**

To print it, use the following code: **E=echo \$num** However, note that nothing will be displayed since the variable is null.

5. Some characters should not be used to name variables. These include the asterisk (\*) and the question mark (?).

# ***Loops in Shell***

A loop represents a group of instructions executed repeatedly.

## ***for Loop***

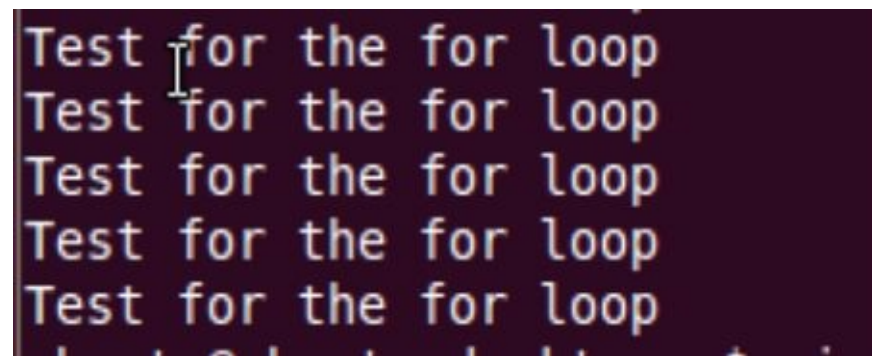
This loop follows the following syntax: **for { name of the variable} in {list }**

**do execute the items in the list one after another until finished.**

**done** Consider the program shown below: **for j in 1 2 3 4 5**

**do echo “Test for the for loop”**

**done** After running the above program, you will observe the following as the output:

A terminal window with a dark background and light-colored text. It displays the output of a shell script. The output consists of five identical lines, each reading "Test for the for loop". A cursor is visible at the end of the first line.

The program begins by defining the variable *j* and sets its value from 1 to 5. The line *echo “Test for the for loop”* of the program will then be executed 5 times since the value of *j* will increment up to a maximum of 5. This explains the source of the output.

## ***while Loop***

The while loop in shell takes the following syntax: **while [ condition]**

**do** 1<sup>st</sup> command 2<sup>nd</sup> command 3<sup>rd</sup> command ..

....

**done** As long as the condition that you have specified is true, the loop will execute.

# Case statement

This is also common in Linux. One can use it rather than the *if-else* statement.

It takes the following syntax: **case \$name\_of\_the\_variable in 1<sup>st</sup> pattern) command ...**

**..**

**command;; 2<sup>nd</sup> pattern) command ...**

**..**

**command;; n<sup>th</sup> pattern) command ...**

**..**

**command;; \*) command ...**

**..**

**command;; esac** Note that we start with *case* but we end with an *esac* statement. Consider the

example shown below: **if [ -y \$1 ]**

**then**

**r="the vehicle is unknown"**

**elif [ -x \$1 ]**

**then**

**r=\$1**

**fi case \$r in**

**"lorry") echo "For \$r 10 per k/m";;**

**"merc") echo "For \$r 80 per k/m";;**

**"corolla") echo "For \$r 15 per k/m";;**

**"motorbyke") echo "For \$r 5 per k/m";;**

**\*) echo "we can't obtain a \$r for you";;**

**esac**

## ***If-else-fi for decision making***

The command *if-else-fi* can be used in shell programming for the purpose of decision making. The programmer specifies a condition and once met, a command is executed. If not met, the other command is executed. It takes the following syntax: **if condition then the condition is zero execute commands until you meet the else statement else if the condition is false execute commands until you find the fi statement fi**

Consider the example shown below: **#!/bin/bash**

**if [ \$# -eq 0 ]**

**then**

**echo "\$0 : an integer must be supplied"**

**exit 1**

**fi if t \$2 -gt 0**

**then**

**echo "\$2 is a positive number"**

**else**

**echo "\$2 is a negative number"**

**fi** If the command line is not given, then an error will be printed, that is, an integer must be supplied. The next part involves checking the number of arguments that have been passed. If we passed any argument, then the “*if*” will turn out to be false and true otherwise. The *exit 1* statement will terminate the program after a successful execution.



# Chapter 9- Bash One-liners

## *Working with files*

1. To truncate the size of a file to 0 (emptying) - Use the redirection command in this case. If the file exists, then its size is truncated to zero whereas if it does not exist, it is created. Note the operator opens the file for writing.

**\$ > file** To create a file containing content of your choice or to replace the file's contents, use the following command: **\$ echo "string to replace" > file** The string that you specify in between the quotes will act as the replacement string.

2. To add a string to your file - the on-liner uses the output redirection operator **>>** to append some content to your file. Example:

**\$ echo "content to be appended" >> file** If the operator fails to find the file, a new one is created. After the string has been appended to the file, a new line will follow it.

To avoid this, use the **-n** option as shown below: **\$ echo -n "content to be appended" >> file**

3. To assign the first line of a file to a variable- in this case, you need to read the first line of the file and then assign this line to a variable. The input redirection operator **<** is used.

Consider the line of code shown below: **\$ read -r myline < file** *myline* is a variable. The command **read** will read only the first line of the file and place it in the variable *myline*. The **-r** option will ensure that this line is read raw.

4. To read a file - a *read* command can be combined with the *while* loop to read a file line-by-line. This is illustrated below:

**while read -r myline; do # perform something on the variable myline done < file** A code of failure will be returned when the *read* command meets end-of-failure. The *while* loop will then halt.

Note that we have placed **< file** at the end. To avoid this, pipe the file's content to a while loop as shown below: **cat file | while IFS= read -r myline; do # perform something on the variable**

**myline done**

5. To read any line of a file and assign it to a variable - external programs assist the bash to read the lines of a program. Consider the code shown below:

**\$ read -r rline < <(shuf file)** *Shuf* is available on modern Linux systems and helps with this. The above code will read a random line of a file and assign it to the variable *rline*.

6. To extract a tar archive, use the following command:

**tar xvf b.tar**

7. To find the number of lines common between two files:

**sort doc1 doc2 | uniq -d**

8. To randomize the lines of a file:

**shuf filename.txt**

9. To sum a particular column:

**awk '{ sum+=\$1} END {print sum}' document.txt**

# Chapter 10- Advanced Shell Programming

In the first part of this book you were introduced to shell programming and most of its basics were explored. Now we are going to explore the advanced features of shell programming.

## *Functions in Shell*

Shell functions have the following syntax: **function\_name ( ) command** They are usually laid out as follows: **function\_name() {**

**Commands }**

There are different exit statuses for functions in a shell. By default they will return an exit status of zero (0). The programmer should specify the exit status needed. It is also possible to define variables locally within a shell function.

Consider the shell program shown below: **#!/bin/sh function\_increment() { # we start by defining the increment so as to use it echo \$(( \$1 + \$2 )) # this will echo the result after addition of the first and the second parameters }**

**# checking for the availability of all the command line arguments if [ "\$1" "" ] || [ "\$2" = "" ] || [ "\$3" = "" ]**

**then echo USAGE: echo " counter initialvalue incrementvalue finalvalue "**

**else c=\$1 # renaming variables having clearer names value=\$2**

**final=\$3**

**while [ \$c -lt \$final ] # if the c is less than final, then loop do echo \$c c=\$(function\_increment \$c \$value) 2# Calling for increment with c and value as the parameters done # the c will be incremented by value fi** Note how we have begun by defining our function. We have then added together the first and second parameters being passed into the function. The use of the “echo” command will print

the result to the standard output. For referencing purposes we use command substitution. This is the line “c=\$(function\_increment \$c \$value)”.

The parameters “c” and “value” will be passed into the line where we specified the first and second argument being passed into the function, which is the line “echo \$(((\$1 + \$2))”. These will then be added together and the result will be printed on the standard output.

## *Scope of variables*

Consider the shell program shown below: `#!/bin/sh function_increment() {`

```
    local v=5
```

```
    echo "The value is $v within the function\\n"
```

```
    echo "\\b$1 is $1 within the function"
```

```
}
```

```
v=6
```

```
echo "The value is $v before the function"
```

```
echo "\$1 is $1 before the function"
```

```
echo echo -e $(function_increment $v) echo "The value is $v after the function"
```

```
echo "\$1 is $1 after the function"
```

We have begun by assigning a value of 5 to our local variable “v” and then specified the desired output.

We have then called our function using the following line of code: `echo -e $(function_increment $v)`

That is what is called a ‘function’ in shell programming. The use of the “-e” option allows the ability to process the slashes in the correct way. Note the use of “[\\n](#)” as a new line character.

The following output will be observed from the program: **The value is 6 before the function \$1 is 2 before the function The value is 5 within the function \$1 is 5 within the function The value is 6 after the function \$1 is 2 after the function**

# Creating Aliases

In Linux, aliases are used to represent commands. If you need to know all the aliases defined on your machine, run the following command on your terminal: **alias** On the example machine, the above command

```
ubuntu@ubuntu-desktop:~$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -aLF'
alias ls='ls --color=auto'
ubuntu@ubuntu-desktop:~$
```

gave the following result:

The above figure shows the aliases defined on this system. All the above are default aliases. You can create aliases as well.

To create aliases, use the following syntax: **alias name='command'**

**alias name='command argument1 argument2'**

In Linux, the command “*clear*” is used to clear the terminal screen. To use the letter “s” to represent the command, that is, create an alias for the command, do the following: **alias s= ‘clear’**

Run the above command on your terminal. This is demonstrated in the figure shown below:

```
ubuntu@ubuntu-desktop:~$ alias s='clear'
ubuntu@ubuntu-desktop:~$ ls
Desktop      file      list.sh   name      name.sh\  Templates
Documents    for.sh    Music     names     nm.sh     Videos
Downloads    hello.sh  myfile.gz names2     Pictures
examples.desktop list      myfiles   name.sh   Public
ubuntu@ubuntu-desktop:~$
```

Since I have used the “*ls*” command to see the files in the directory, I need to clear the terminal. Type the letter “s” on the terminal and press the enter key. It will clear the terminal.

The command “*date*” in Linux is used to display the current date. Create an alias for it. Use letter “*d*” as an alias for the same command.

This is demonstrated in the figure shown below:

```
ubuntu@ubuntu-desktop:~$ alias d='date'
ubuntu@ubuntu-desktop:~$ d
Wed Apr 29 11:43:46 MDT 2015
ubuntu@ubuntu-desktop:~$
```

As shown in the above figure, the letter “*d*” is used as an alias for the “*date*” command. After typing the letter “*d*” on the terminal, press the “*Enter*” key and the current date is displayed.

Aliases are very important in Linux. If you want to save on typing time then create them. The name for an alias should be easy to remember, including the commands that they represent.

You might find the need to delete aliases you have created. This can be achieved as follows: On this

```
ubuntu@ubuntu-desktop:~$ alias
alias d='date'
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -aLF'
alias ls='ls --color=auto'
alias s='clear'
ubuntu@ubuntu-desktop:~$
```

system, I have the following aliases:

Note the presence of the two aliases that were created previously. To delete the alias “*d*” representing the “*date*” command, run the following command: **unalias d** This is demonstrated in the figure shown below:

```
ubuntu@ubuntu-desktop:~$ unalias d
ubuntu@ubuntu-desktop:~$ d
d: command not found
ubuntu@ubuntu-desktop:~$
```

As shown in the above figure, after deleting the alias and then trying to use it, the computer states that it is not found.

It is possible to remove more than one alias at once. If you need to remove the aliases “*d*” for “*date*” and “*s*” for “*clear*” the following command should be used: **unalias d s** The above command will delete the two aliases. After running the command you can then run the “*alias*” command to check whether they are still available.



# Tilde Expansion in Linux

In most cases Linux users use the tilde (~) symbol to refer to their home directory, while others use the home directory.

To see your home directory file listing run the following command: **ls ~**

To view the “.bashrc” file located in the home directory, run the following command: **cat ~/.bashrc ls**

**~/.bashrc** The first command will open the file on the terminal window. This is demonstrated below:

```
ubuntu@ubuntu-desktop:~$ cat ~/.bashrc
# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples

# If not running interactively, don't do anything
[ -z "$PS1" ] && return

# don't put duplicate lines in the history. See bash(1) for more options
# ... or force ignoredups and ignorespace
HISTCONTROL=ignoredups:ignorespace

# append to the history file, don't overwrite it
shopt -s histappend
```

The second command will show the directory where the file is located. This is shown in the following

```
ubuntu@ubuntu-desktop:~$ ls ~/.bashrc
/home/ubuntu/.bashrc
ubuntu@ubuntu-desktop:~$
```

figure:

If the prefix for the tilde symbol is a plus (+), this will substitute the “PWD” command. If it is preceded by a negative sign, or (-), then the variable “OLDPWD” is substituted if it had been set.

It is worth noting that Linux commands can either be built-in or in an external binary file. To know where a command belongs, we use the “type” command.

To find out whether the command “ls” is built-in, run the following command: **type -a ls** The above

```
ubuntu@ubuntu-desktop:~$ type -a ls
ls is aliased to `ls --color=auto'
ls is /bin/ls
```

command will give the following output:

To find out whether the “*history*” command is built-in or an external command, run the following

```
ubuntu@ubuntu-desktop:~$ type -a history
history is a shell builtin
ubuntu@ubuntu-desktop:~$
```

command: **type -a history**

It is worth noting that some commands in Linux can be both built-in and external.

To demonstrate this, run the following commands: **type -a echo** The above command will give the

```
ubuntu@ubuntu-desktop:~$ type -a echo
echo is a shell builtin
echo is /bin/echo
ubuntu@ubuntu-desktop:~$
```

following output:

## ***Nested “ifs” in Shell***

Nested “*if*” means that it is an “*if*” statement inside another “*if*” statement.

It follows the following structure: **if condition then if condition then . . .**

**Statement to be executed else ...**

**Stamen to be executed fi else ...**

**....**

**Statement to be executed fi**

# *Exit status for commands*

After a command executes and terminates either normally or abnormally, it must return an exit status. The exit status is usually an integer value. An exit status of zero (0) means that the command executed successfully. Any other exit status, which can range from 1-255, means that the command failed to execute.

To know the exit status of a previously executed command, use the variable “?”, which is a special character in shell.

To determine the exit status, run the following command: **echo \$?**

This will give the exit status of the previously executed command. In the example system it offers the

```
ubuntu@ubuntu-desktop:~$ echo $?  
0  
ubuntu@ubuntu-desktop:~$
```

following result:

The result is a zero (0), meaning that the previously executed command on the system executed successfully.

To demonstrate this practically, begin by running the “ls” command on your system. Make sure that it runs successfully. On the example system it gives the following result:

```
ubuntu@ubuntu-desktop:~$ ls  
Desktop      file      list.sh    name      name.sh\  Templates  
Documents    for.sh    Music      names     nm.sh     Videos  
Downloads    hello.sh  myfile.gz  names2    Pictures  
examples.desktop list      myfiles    name.sh   Public  
ubuntu@ubuntu-desktop:~$
```

Since it has run successfully, try to check its status using the special shell character.

Run the following command: **echo \$?**

The above command should return an exit status of zero (0), since the previous command executed successfully.

```
ubuntu@ubuntu-desktop:~$ echo $?  
0  
ubuntu@ubuntu-desktop:~$
```

This is demonstrated in the figure shown below:

This time try to run a command that is not recognized and then check the exit status. Run the following command: **ls1**

You should be aware that the above command does not exist in Linux. Running it will result in an error.

In the example system it gives the following result:

```
ubuntu@ubuntu-desktop:~$ ls1  
No command 'ls1' found, did you mean:  
Command 'lsh' from package 'lsh-client' (universe)  
Command 'lsw' from package 'dwm-tools' (universe)  
Command 'ls' from package 'coreutils' (main)  
ls1: command not found
```

The output shows that the command has not been found. This means that it does not exist in Linux.

To check its exit status, run the usual command, which is shown below: **echo \$?**

The above command gives the following output on the example system:

```
ubuntu@ubuntu-desktop:~$ echo $?  
127  
ubuntu@ubuntu-desktop:~$
```

The exit status is not zero (0), meaning that the command did not execute successfully. The returned integer is between 0 and 255, as we initially said. To conclude this, any command whose exit status is not zero did not execute successfully.

# Conditional execution in shell

In Linux shell programming it is possible to join two commands, where the execution of the second command will be based on the first.

## Logical AND

This takes the following syntax: **1<sup>st</sup>command && 2<sup>nd</sup>command** With this conditional execution, the “2<sup>nd</sup>command” will be executed if and only if the “1<sup>st</sup>command” returned an exit status of zero (0). This means that you should first run the “1<sup>st</sup>command” if the “2<sup>nd</sup>command” runs successfully.

Let’s demonstrate this using an example.

In the current directory I have the following file:

```
ubuntu@ubuntu-desktop:~$ ls
Desktop      file      list.sh   name      name.sh\  Templates
Documents    for.sh    Music     names     nm.sh     Videos
Downloads    hello.sh  myfile.gz names2     Pictures
examples.desktop list      myfiles   name.sh   Public
```

Let me try to delete the file named “*file*” and then echo a message afterwards. I run the following command: **rm file && echo “file has been deleted”**

The above command will return the following result:

```
ubuntu@ubuntu-desktop:~$ rm file && echo "file has been deleted"
file has been deleted
```

The output clearly shows that the first command ran successfully, followed by the second command. Try to run an incorrect command first to see what happens.

From the list of files in the directory there is no file named “fruits”. Try to delete a file with that name and then echo a message afterwards.

Run the following command: **rm fruits && echo “the file has been deleted”**



The above command will give the following result:

```
ubuntu@ubuntu-desktop:~$ rm fruits && echo "file has been deleted"
rm: cannot remove `fruits': No such file or directory
ubuntu@ubuntu-desktop:~$
```

The command returns an error message. This means that the first command had no exit status of zero since it did not run successfully. If it had run successfully we would have the correct result.

Suppose you want to use the “*grep*” command to search for a particular word or name in a file. You can make use of the logic AND to echo the result. Give the file “*names2*” two names. Search for the name “*caleb*” in the file.

Run the following command: **grep “caleb” names2 && echo “The name was found”**

The above command gives the following result:

```
ubuntu@ubuntu-desktop:~$ grep "caleb" names2 && echo "The name was found"
caleb
The name was found
ubuntu@ubuntu-desktop:~$
```

The above figure shows that the first command exited with an exit status of 0, so the second command was run successfully.

# Logic OR

This is a Boolean operator. Programmers can make use of this operator to execute a certain command based on another command.

It takes the following syntax: **1<sup>st</sup>command || 2<sup>nd</sup>command** The command “2<sup>nd</sup>command” will be executed if and only if the command “1<sup>st</sup>command” executes unsuccessfully, meaning that it returns a non-zero exit status. In other words, you can run only one of the commands. This means that if the first command runs successfully, the second command will be unsuccessful and vice versa.

Consider the following Linux command: **grep “caleb” names2 || echo “The name was not found”**

The above command will output the following:

```
ubuntu@ubuntu-desktop:~$ grep "caleb" names2 || echo "The name was found"
caleb
ubuntu@ubuntu-desktop:~$
```

The output shows that the name “caleb” was successfully found in the specified file. Note that the second command was not executed. This is because we can only execute one of the commands.

Consider the example command shown below: **grep “john” names2 || echo “The name was not found”**

The above command gives the following output:

```
ubuntu@ubuntu-desktop:~$ grep "john" names2 || echo "The name was found"
The name was found
ubuntu@ubuntu-desktop:~$
```

From the output shown above it is very clear that the second was executed. The first command executed unsuccessfully and this led to the execution of the second command. This is because there is no name “john” in the specified file.

This can be demonstrated using another example.

Try to delete a certain file from the current directory. The following command should be used: **rm for.sh ||**



## echo “file not deleted”

In the directory there is a file named “*for.sh*”. I then try to delete it. Since the first command succeeds, meaning that the file will be deleted, the second command will not be executed. After running the command I am taken back to the terminal since the operation has been completed.

This is demonstrated in the figure shown below:

```
ubuntu@ubuntu-desktop:~$ rm for.sh || echo "file not deleted"
ubuntu@ubuntu-desktop:~$
```

What will happen when trying to delete a file that is not present? This is demonstrated using the following example.

Try to delete a file that does not exist in the directory. Run the following command: **rm myfile || echo |file not found”**

The above command tries to delete a file that does not exist in the current directory. If the deletion fails, the second part of the command should be executed. In the example system the above command gives the

```
ubuntu@ubuntu-desktop:~$ rm myfile || echo "file not found"
rm: cannot remove `myfile': No such file or directory
file not found
ubuntu@ubuntu-desktop:~$
```

following result:

The first part of the command ran unsuccessfully, meaning that it had a non-zero exit status. This led to the execution of the second part of the command.

It is also possible to combine the two logical operators into one. Consider the command shown below: **grep “caleb” names2 && echo “name found” || echo “not found”**

On this system the above command gives the following output:

```
ubuntu@ubuntu-desktop:~$ grep "caleb" names2 && echo "name found" || echo "not found"
caleb
name found
ubuntu@ubuntu-desktop:~$
```

Note that only the first two commands have been executed. The last command has not been executed. Try

to search for a name that is not available in the specified file.

The command below can be used: **grep “john” names2 && echo “name found” || echo “not found”**

The above command gives the following output:

```
ubuntu@ubuntu-desktop:~$ grep "john" names2 && echo "name found" || echo "not found"
not found
ubuntu@ubuntu-desktop:~$
```

It is very clear that the first two parts of the above command have not been executed. The second part of the command will only be executed if the first part of the command returns an exit status of zero (0). This is due to the use of the logical “*AND*” operator.

The last part of the command is executed on its own if and only if the first two parts of the command run unsuccessfully.

## ***Logical Not***

This is also a logical operator and it is used for testing whether an expression is true or not.

It takes the following syntax: **! expression** It can also take the following syntax: **[ ! expression ]**

You can combine it with the “*if*” statement as shown below: **if test ! condition then 1<sup>st</sup>command  
2<sup>nd</sup>command fi** Or **if [ ! condition ]**

**then 1<sup>st</sup>command 2<sup>nd</sup>command fi** If the expression is false it will return true. Consider the example shown below: **! -f name && exit** After running the above the command prompt will close if the file “*name*” is not found. In this case it doesn’t exit since there is a file with the name.

## ***“Continue” statement in shell***

This statement is used in shell to resume an iteration of a FOR, WHILE or UNTIL loop enclosing. It takes a very simple syntax: ...

```
while true do [ 1stcondition ] && continue Command 1
```

```
Command 2
```

```
[ 2ndcondition ] && break done ...
```

The statement can also take the following syntax: ..

```
for j in thing do [ condition ] && continue Command 1
```

```
Command 2
```

```
done ..
```

```
...
```

The following is an example of a MYSQL backup script that makes use of the “continue” statement:

```
#!/bin/bash # A script to backup mysql # run it while logged in as a root user # -----  
-----  
  
# Login information MUSER="admin" # MySQL user name MHOST="192.168.160.1" # MySQL  
server ip address MPASS="password" # MySQL password # date format NOW=$(date +"%d-%m-  
%Y") # path for the Backupfile BACKUPPATH=/backup/mysql/$NOW  
  
# create the backup path if it doesn't exist [ ! -d $BACKUPPATH ] && mkdir -p $BACKUPPATH  
  
# obtain name lists from the database DBS="$(/usr/bin/mysql -u $MUSER -h $MHOST -p$MPASS  
-Bse 'show databases')"  
  
for database in $DBS  
  
do # Backup the name of the file FILE="${BPATH}/${database}.gz "  
  
# if the name of the database is server or mint, then skip the backup [ "$database" == "server" ]  
&& continue [ "$database" == "mint" ] && continue # if okay, then we dump the database backup  
/usr/bin/mysqldump -u $MUSER -h $MHOST -p$MPASS $database | /bin/gzip -9 > $FILE  
  
done
```

# ***Exit command***

This command has the following syntax: **exit N**

Programmers use it to exit a shell script whose status is “N”. You can use it to signal either a successful or an unsuccessful termination of a program. If “N” is omitted, then the exit status will be that of the last command to be executed. You can use this command to end your scripts if an error occurs. Setting “N” to zero (0) means a successful completion of the script.

To demonstrate this, write the following shell script: **#!/bin/bash echo "A test shell script."**

**# terminate our script with a success message exit 0**

I have named this script “*shell.sh*”. Note that shell scripts must end with a “.sh” extension, otherwise they will not be executed as shell files. Now you can run the script.

Open the terminal and run the following commands: **chmod a+x shell.sh ./shell.sh** The first command will create an executable of the file, which means that it is in the compilation stage. The second command will execute the program. It does so from the executable created by the first command.

After running the above commands you will get the following output:

```
ubuntu@ubuntu-desktop:~$ chmod a+x shell.sh
ubuntu@ubuntu-desktop:~$ ./shell.sh
A test shell script.
```

Now let's check the exit status of the above script.

This can be done by running the following command: **echo \$?**

The above command will give the following output:

```
ubuntu@ubuntu-desktop:~$ echo $?
0
ubuntu@ubuntu-desktop:~$
```

The exit status of the script is zero (0). This shows that it ran successfully.

# Shell yes/no dialog

It is possible to create a dialog that prompts a user to choose either “yes” or “no” in shell. This is useful when presenting questions to users whose answer is either a “yes” or a “no”. The user is also able to navigate between the “yes” and the “no” buttons using the “Tab” key. Let’s demonstrate this using an example program.

```
Open your editor and write the following shell script: #!/bin/bash # Yes/No dialog box example dialog -  
-title "Exit"  
  
--backtitle "A yes/no dialog box example"  
  
--yesno "Are you sure you want to exit the system \"/tmp/foo.txt\"?" 7 60  
  
# obtain the exit status # 0 represents the user hits the [yes] button.  
  
# 1 represents the user hits the [no] button.  
  
# 255 represents the user hits the [Esc] key.  
  
action=$?
```

```
case $action in 0) echo "You have exited.";; 1) echo "You have not exited.";; 255) echo "You  
pressed the [ESC] key.";; esac Save the file and then run it. I have given my script the name “exit.sh”.
```

To run it, run the following command: **chmod a+x exit.sh ./exit.sh** After the above commands, a dialog will appear with “yes/no” options.

This is shown below:



A text asking on whether to exit system will also appear.

This is the text that we initially specified:



**Password Box** The password box is a box where you can provide a password.

The password that you type is not visible, but appears in a form of dots or asterisks. In other password boxes nothing appears as the user types in the password. This is for the purpose of ensuring the security of your system.

To demonstrate this, write the following shell script: **#!/bin/bash # passwordbox.sh – a shell script that creates a password box # storing the password d=\$(tempfile 2>/dev/null) # trap the password trap "rm -f \$d" 0 1 2 5 15**

```
# receive the entered password dialog --title "Password Box" \
```

```
--clear \
```

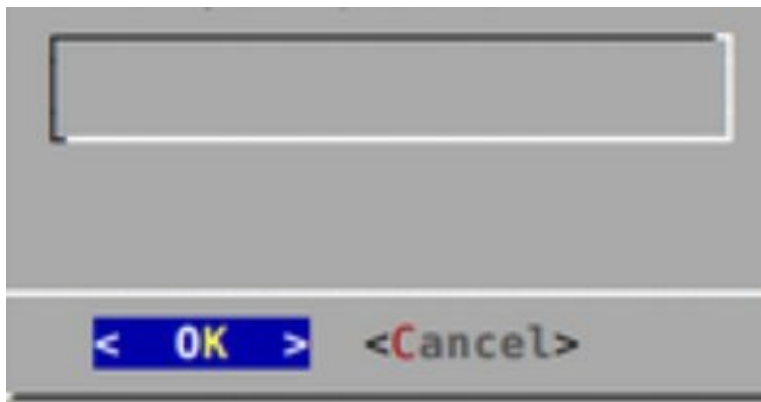
```
--passwordbox "Key in your password" 10 30 2> $data r=$?
```

```
# make your decision case $r in 0) echo "Password is $(cat $d)";; 1) echo "You pressed  
Cancel.";; 255) [ -s $d ] && cat $d || echo "You pressed the Esc key.";; esac
```

Save the script and give it the name “*passwordbox.sh*”. You can then run the script.

Open the terminal and run the following commands: **chmod a+x passwordbox.sh ./passwordbox.sh**

After running the above commands, a dialog with a box will appear. This is shown below:



Try to type in your password. You will notice that nothing appears in the password box.

Most people are used to password boxes in which an asterisk appears for each character typed. It is

possible to achieve this in Linux. It is called the in-secure option.

Write the following shell script: **#!/bin/bash # passwordbox2.sh – a shell script example to read user password # storing the password d=\$(tempfile 2>/dev/null) # trapping the password trap "rm -f \$d" 0 1 2 5 15**

**# use the in-secure option to get the password dialog --title "In-secure Password" \**

**--clear \**

**--insecure \**

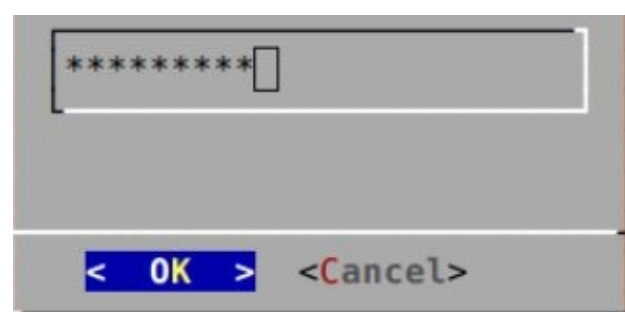
**--passwordbox "Key in your password" 10 30 2> \$d r=\$?**

**# making a decision case \$r in 0) echo "The Password is \$(cat \$d)";; 1) echo "You pressed the Cancel button.";; 255) [ -s \$d ] && cat \$d || echo "You pressed the Esc key.";; Esac** You can

then save the script and give the file a name. I have given my file the name “*passwordbox2.sh*”.

Don’t forget the “.sh” extension.

Now run the script by opening the terminal and executing the following commands: **Chmod a+x passwordbox2.sh ./passwordbox2.sh** After executing the above commands, a dialog with a box will appear. Try to key in the password. You will observe the following:



Asterisks will appear for each character that you type.

# ***Progress bar***

It is possible to create your own progress bar in shell. You can use it to show the progress of tasks that are being performed, such as deleting files, moving files or opening songs. It consists of a meter and a percentage indicating the progress of the task. The percentage keeps on increasing as the task progresses until it gets to 100%.

To demonstrate this write the following shell script: **#!/bin/bash # pbar.sh – a shell script example to create a progress bar # initialize the counter to 0**

```
counter=0
```

```
(
```

```
# create an infinite “while” loop while : do cat <<EOF
```

```
XXX
```

```
$counter Disk copy /dev/dvd to /home/data ( $counter%): XXX
```

```
EOF
```

```
# the counter will be incremented in increments of 5 each (( counter+=5 )) [ $counter -eq 100 ]
```

```
&& break # delay it for a short time of 2 seconds sleep 2
```

```
done ) |
```

```
dialog --title "Copying file" --gauge "percentage completion" 7 70 0
```

You can now save the script. I have named my script “*pbar.sh*”. To run it, run the following commands:

**chmod a+x pbar.sh ./pbar.sh** After running the above commands, the following output will be observed:



Above is the progress bar. As shown in the figure, it indicates the percentage of completion of the task.

Mine is at 15% completion. Don't worry if yours shows a different percentage as the speeds of computers will always be different. It might be less or more than the 15%. That is how you can simply create your own a progress bar.

# Chapter 11- Compiling UNIX software packages

Our systems have numerous commercial as well as public domain software installed, and these are available to all users. It is advisable that you should download and install software packages to your home directory, but you should also ensure that you only download the software packages that are useful to you.

Installation of any software in Linux takes the following steps:

- Locate and then download its source code, in a compressed format
- Unpack source code
- Compile the code
- Install the resulting executable
- Set paths to installation directory

***Source Code Compilation*** Any code that is written in a high-level language has to be converted into a form that the computer can understand. A good example is the C programming language, which must first be converted into the assembly language. This assembly code is then converted into machine code, which the computer is capable of understanding. The object code is then linked so as to form code libraries, and these are made up of a number of built-in functions. This stage, which is the final one, gives us the executable program.

The ordinary user is unable to do this as it is beyond their means. The process is too complicated such that an ordinary user cannot do it. There are a number of utilities and tools that have been developed for end users as well as programmers so that they can do this more easily.

***make and makefile*** The “make” commands becomes useful to programmers when they need to manage a number of programs that are in groups. The command helps us to make large programs, since it allows us to know or detect the parts of the program that have been changed. Only the parts that have been affected by the change since the last time the compilation was done will be compiled.

The compile rules for the “make”command are contained in a file named“ Makefile and this file is kept in the same directory as the source files. The information on how the software should be compiled is contained in this file. Examples of such information include whether the debugging information should be added to the executable and optimization level. The file also has information stating where the finished compiled binaries or the executables should be installed, the data files, the manual pages, the configuration files, the dependent library files and others.

Some packages will expect you to do some manual editing on the Makefile so as to set the final installation directory as well as the other parameters. However, the GNU configure utility is now being used for distribution of many packages.

***Configure*** With an increase in the number of UNIX variants, it has become hard to write programs capable of running all the variants. The developers are not granted access to all the systems, while the characteristics of some of the systems keep on changing from version to version. The configure and build system for GNU makes it simple for us to build programs that are distributed as source code. The building of programs allows a standard process with two steps only. There is no need for the program builder to install any specialized tools in order to be in a position to build the program.

The configure shell script tries to guess the correct values for the system-dependent variables that are



used during the compilation process. The values are used for creating the Makefile for each directory contained in the package. The following are the easiest steps for how to compile a package:

- `cd` to directory with the source code for the package.
- Type `./configure` for configuring the package for the system.
- Type `make` for compiling the package.
- Optionally, type `make check` to run any self-tests that come with the package.
- Type `make install` to install programs and any other data files and the documentation.
- Optionally, type `make clean` to remove program binaries and the object files from source code directory.

The `configure` utility is capable of supporting a wide range of options. If you need to know more about a `configure` script, use the `--help` command and all of its options will be presented to you. The generic options that you are capable of using include the `--exec-prefix` and `--prefix`. These are the options that are used for specification of the installation directories.

The name that is given a name by use of the option `--prefix` will be responsible for holding the machine independent files such as data, documentation and the configuration files. The directory that is named by use of the option `--exec-prefix` is responsible for holding the files that are machine dependent. Examples of such files are the executables.

***Downloading the Source Code*** We need to demonstrate how we can download source code from the internet.

Let's begin by holding a directory into which we will save the source code that we download. We will give it the name "dir1". We will use the make directory command to create it as shown below: **\$ mkdir dir1**

Now that you have created the directory, go ahead and download the source code you need and then transfer it to that new directory. Suppose you have downloaded a source code named "myfile.tar.gz".

You can then navigate to the "dir1" directory and then see its contents by listing them. This is shown below: **\$ cd dir1**

**\$ ls -l** You will notice that the file that you have just downloaded ends with a .tar.gz. When the tar command is used, several files are turned into a single tar file. The compression is then done by use of the gzip command so as to create a tar.gz file.

Let's first use the gunzip command to unzip our file. The following command will help us achieve this: **\$ gunzip myfile.tar.gz** This will leave you with a tar file, with a .tar extension. Your next step should be to extract the contents of this tar file. This can be achieved by use of the following command: **\$ tar -xvf myfile.tar** You can go ahead to list the contents of the directory named "dir1". Once there, move ahead to the "myfile" sub-directory. You can achieve this by use of the change directory (cd) command as shown below: **\$ cd myfile**

***Configuration and Creation of the Makefile*** The first thing that you should do is to read the README and then install the text files using the “less” command. These have very crucial information regarding the installation and running of the software.

The units package makes use of the GNU configure system for compiling the source code. It is our duty to specify the directory for installation, as the default directory will be the main system area that you have no write permission for. It will be useful for us to create an install directory in the home directory.

**\$ mkdir ~/myfile** You can then set the installation path to this configure utility. The following command can help do this: **\$ ./configure --prefix=\$HOME/myfile** The \$HOME variable represents an environment variable. Its value is the path that leads to the home directory. To get its value, type the following command: **\$ echo \$HOME**

The “configure” was executed successfully, so a “Makefile” will have been created together with all the necessary options. You may need to view the contents of this file but make sure that you don’t edit it.

***Build the Package*** The `make` command will help you to build the package. You can run the command as follows: `$ make` The executable will be created after a minute or two. This will be determined by the speed of the computer. To see and be sure that everyone is working correctly, you can run the following command: `$ make check` If you find that all is okay, you can go ahead to install the package. This can be done as follows: `$ make install` The files will then be installed in the “~/myfile” directory, which had been created earlier.

***Running the Software*** At this point, you will be ready to run your software. You can change your directory to it as shown below: `$ cd ~/myfile` Once you list the contents of this directory, you will see that it has a number of sub-directories. These include the following:

1. `bin`- this directory has the binary executables.
2. `info`- this has the GNU info in a formatted manner.
3. `man`- this has the man pages.
4. `share`- for shared data files.

To run this program, change your directory to the “bin” sub-directory and then run the following command: `$ ./myfile` If you need to view the whole documentation, just change to the “info” directory and then run the following command: `$ info --file=units.info`

***Removing Unnecessary Code*** Once a particular piece of software has been developed, it is usually a good idea for the developer to include the debugging information to the executable that has been created. This is so, if you encounter problems when you are running the executable, the programmer will be in a position to load the executable into a debugging software package and he or she will be in a position to detect any bugs in the software.

This could become very useful to the developer or the programmer, but it is of no use to the user. It can be assumed that by the time the software is available for download, it has been tested and found to have no bugs. However, during the compilation of our program, even the information regarding debugging was made part of the executable. Since there is no possibility of us having to use this debugging information, we can choose to remove it from the executable. This brings about the advantage that the executable will have a smaller size, and its execution will be faster.

We are going to consider the file size before and after. Begin by changing to the “bin” directory of the installation directory. You can then list its contents. These steps are given below: **\$ cd ~/myfile/bin**  
**\$ ls -l** Look at the section that tells you the size of the file. In my case, the file is over 100KB. You can use the “file” command followed by the name of the file while still in this directory and you will see its exact size.

It is now time for us to strip unnecessary information from the file. We will strip the numbering and the debug information from this file. Use the following command to do this: **\$ strip myfile**  
**\$ ls -l** The last command shown above is for listing the contents of the directory. Note the new size of the file. You will notice that it has reduced significantly. A big part of the code will have gone. That's it!

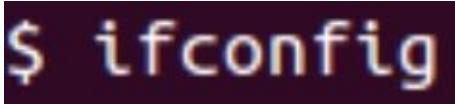
# Chapter 12- Linux Networking

Computers are connected to one another over a network for the purpose of exchanging information. The network of these computers can either be small, medium or large.

A network administrator is responsible for maintaining the network, and this administration involves configuration and troubleshooting. Let's discuss some of the Linux networking commands.

**Linux ifconfig** This command stands for “interface configuration”. It helps us to initialize an interface, assign the IP address and enable or disable some interface. It is also responsible for showing the route and the network interface.

The ifconfig command can help you view the MAC address, the IP address, and the MTU (Maximum Transfer Unit). Just open the terminal of your Linux OS and then type the “ifconfig” command. Hit the

Enter key: 

The command will show you all the network interfaces that are on your computer. The IP addresses for each of these interfaces will be shown.

It is also possible for you to get the specific details of a particular interface by use of the ifconfig command. Suppose you have an Ethernet interface in your system named “eth0”. You can view the

specific details of this interface as follows: 

The command will give you more details regarding the interface. The parameter “HWaddr” represents the MAC address of the interface, which is always unique for the interface.

For the case of the “wlan0”, which is a wireless LAN interface, I get its specific details using of the

following command: 

This command will also give the MAC address of the interface in the form of HWaddr, as well as its IP address. For the case of the loopback interface, which is denoted using the name “lo”, its ip address will be 127.0.0.1.

***IP Address Assignment*** It is possible to assign an IP address as well as a gateway to an interface, but you should know that these would be changed once the machine reboots. The `ifconfig` command allows us to do this using the following syntax: **`ifconfig eth0 <address> netmask <address>`**

It is also possible to enable or disable a specific interface, such as `eth0`. The following command can help you to enable the interface: **`ifup eth0`**

To disable this interface, you can use the following command: **`ifdown eth0`**

***The MTU Size*** The default size for the MTU (Maximum Transfer Unit) is 1500. However, it is possible to change it to another value. The following command can help you change this size: **`Ifconfig eth0 mtu xxxx`** In this case, the `xxxx` will be the new size that you want.

***Linux ip*** This forms the newer version for the `ifconfig` command. It takes the following syntax: **`ip a or ip addr`** The following commands can help you see the details of a specific interface by use of the “ip” command: **`ip a show eth0`**

**`ip a show lo ip a show wlan0`**



***Linux traceroute*** This is a tool for network troubleshooting. It helps us to know the number of hops that a packet will traverse before it can reach its destination. For those who have not installed the traceroute utility into their system, you can install it by running the following command: **sudo apt-get install inetutils-traceroute** The command takes the following syntax: **traceroute <destination>** A good example is shown below: **traceroute facebook.com**

***Linux tracepath*** This command can be seen to be similar to the traceroute command. The difference is that it does not need any administrative privileges. By default, Ubuntu comes with this tool installed. It works by tracing the path taken by a packet and it then reports more about the hops that are found on the network. If your network is weak, then this tool will help you to know where there is a weakness.

This command takes the following syntax: **tracepath <destination>** Try to type this command followed by a certain domain for a website as shown below: **tracepath facebook.com**

***Linux Ping*** This command stands for Packet Internet Groper. It is used to check for the connectivity between two nodes, that is, whether you are able to reach a server from a particular node. This command will execute and continue to send packets until you interrupt it. To stop it, you have to press “ctrl + c”.

## ***Ping with DNS***

We can use the ping command together with the domain, or the domain name server (DNS). This will take the following syntax: **ping <destination>** The following example demonstrates this: **ping facebook.com**

***Ping with IP Address*** The IP address can be used together with the ping command. Consider the following example: **ping 192.168.160.1**

When using the ping command, you can use the **-c** option so as to impose a limit on the number of packets to receive. In this case, you will not have to use the **ctrl + c** to stop the command from running. This is because you will have specified the number of packets to receive. If all the packets specified are received, the command will stop execution. Consider the example given below: **ping -c 5 facebook.com**  
In the above example, only 5 packets will be sent and you will not have to press any keys.

***Linux netstat*** This command stands for Network statistics. It will display information regarding the different network interface statistics such as the open sockets, the connection information and the routing tables. To use it, just open the terminal and type as shown below: **\$ netstat** The command will then display the open sockets in your system.

With the use of the **-p** option, you will be in a position to see the programs that are associated with the open sockets. To see this, you just have to open the terminal and then type the following: **\$ netstat -p** You will see all programs that are using the open sockets in your system. To get detailed information regarding the ports that are being used, you should use the **-s** option. This is shown below: **\$ netstat -s** To see the information regarding the routing table, you should use this command together with the **-r** option. This is shown below: **\$ netstat -r** This will display the routing table for you.

**Linux ss** This command replaces the netstat command. However, when it is used, you will get more information than what you can get with the netstat command. All the information is obtained from the kernel userspace, and the command runs faster compared to the netstat. To use it, you just have to open the terminal and type as shown below: **\$ ss** The command will then show all the TCP, UDP and the socket connections. The command can also be used for displaying the listening as well as the connected ports for UDP, TCP and UNIX. To achieve this, we use the **-u**, **-t** and the **-x** options combined with the **-a** command. These are shown below: **ss -ta ss -ua ss -xa** You can run them on your terminal and you will see the results from each command.

The listening for UDP, TCP and UNIX can be listed by use of the **-u**, **-t** and **-x** options respectively and then combined with the **l** command. These commands can be written as shown below: **ss -lu ss -lt ss -lx**

***Linux dig*** This stands for “Domain Information Groper”. It is used for tasks that are related to DNS lookup and querying the DNS name servers. It is usually used for troubleshooting any issues related to DNS.

If you need to find a record about a particular domain, use the dig command followed by the name of the domain. This command takes the following syntax: **dig <domainName>** A good example of the use of this command is shown below: **\$ dig facebook.com** You will then get more information regarding the domain you have typed.

***Linux nslookup*** This command can also be used when we need to find any DNS related query. It takes the syntax given below: **nslookup <domainName>** An example on how this command can be used is shown below: **\$ nslookup facebook.com** The command will then return the record information about facebook.com.

***Linux route*** This command is used for displaying the route table and for manipulation of the IP routing table for the system. The router is the device that is commonly used for determining the best route a packet can follow to reach its destination. To use the command, you just have to open the terminal and then type it: **\$ route** The command will then give you the information that has been put into your routing

table. This is shown below:

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
default	10.0.0.1	0.0.0.0	UG	0	0	0	wlan0
10.0.0.0	*	255.255.255.0	U	2	0	0	wlan0
link-local	*	255.255.0.0	U	1000	0	0	wlan0

In this case, the routing table indicates that the destination for the packet lies between 10.0.0.1 and 10.0.0.255. The gateway in this case is \*, and this represents 0.0.0.0. This address is special and it is used for packets that have no destination.

***Showing IP Addresses*** The route command can be combined with the **-n** option so as to get the IP address in a numerical form. This takes the following syntax: **\$ route -n** Just type the above command on your terminal and see the result it gives. The command will then show the IP addresses of the system in their numerical form.

***Addition of a Default Gateway*** The packets that are not found to belong to the network range will be sent through the default gateway. The gateway can be specified using the following command:

**route add default gw <IP address>** ***Routing the Cache Information*** The kernel usually

keeps a routing cache table so that you can route the packets faster. The following command can help you

to list such information: **\$ route -Cn**







# Chapter 13- Introducing the Vi Editor

The Vi represents the visual editor. It comes installed in each UNIX system. It is a very powerful tool that is mostly used by Linux geeks. However, as a beginner, it is good for you to learn how to use it rather than rely on basic text editors such as the gedit.

The vi editor has two main modes that include the following:

1. Command mode- in this mode, actions are taken against the file. The editor starts in this mode, and anything typed will be seen as a command. If you need to pass a command, then you must be in the command mode.
2. Insert mode- in this mode, the text you type is entered into the file. To get back to the command mode, you have to hit the Esc key.

If you are in the command mode and you need to get to the insert mode, then you should press the letter “i”.

You should be aware that the vi editor is case sensitive. A good example is the letter P that will allow you to paste after current line, while p will allow you to paste before the current line.

To create a new file in the vi editor, you use the following syntax: **vi <fileName>** If the file you specify exists, it will be opened, but if it does not exist, then it will be created and opened.

Let us discuss some of the commands that are supported in vi:

1. :wq- save and quit
2. :w- for saving
3. :q- to quit
4. ZZ- save and quit
5. q!- quit without saving
6. w!- save
7. j- to move down

8. k- to move up
9. h- to move left
10. l- to move right
11. i- to begin type before current character.
12. a- start typing after current character.
13. A- start typing at the end of the current line.
14. x- deletes the current character
15. X- will delete the character that is before the current one.
16. r- will replace the current character.
17. xp- will switch two characters.

# Conclusion

In conclusion, Linux is an open source operating system that has various distributions. The idea behind creating Linux was to develop a free Unix-like operating system for computers. Each of the Linux distributions comes in two versions, that is, the server and the desktop versions. The desktop versions support both the GUI and the command line.

By default, you can access the GUI after logging into the system. The command line can then be launched if needed. The server versions of these distributions support only the command line. No graphics are used in Linux servers. The purpose of this is to ensure the security of these systems, since hackers usually take advantage of any feature added to this system. This explains why you should be familiar with the Linux commands.

The Linux command Line can be used for nearly everything needed in the system. You will earn a lot of respect in the tech world when you become an expert in using the Linux command line. With shell scripting, programmers can put together multiple commands and run them simultaneously so as to achieve the desired output. This also saves on time and space available on the system. Shell scripts are preceded by the shebang.

This alerts the system that a shell script is just about to be executed. These must be made executable and then execution will take place after this. Bash stands for Bourne Again Shell. It is the default shell in most Linux distribution. Bash scripts are run on the command line, and you can achieve a lot with them, plus learn a lot about the system. Bash on-liners are also important in Linux. You can use them to play around with your files.