Lab File

**Operating System and Linux Administrator**

PREPARED BY

**Rishi Sharma**

**2021BCA021**

FACULTY GUIDE

**Dr. Gireesh Kaushik**



**Department of Computer Science Engineering**

**Institute of Engineering and Technology (IET)**

**JK Lakshmipat University Jaipur**

|  |  |
| --- | --- |
| **Exp No.** | **NAME OF THE EXPERIMENT** |
| 1. | Introduction to Linux operating system: A Case Study.   * What is Linux? * History of Linux. * Components of Linux System. * Basic features. |
| 2. | Basic LINUX commands and their Use.   * Execution of various file directory handling commands. * Commands related to standard I/O, Redirection, Pipes, and Filters. * Examples Exercise |
| 3. | Process Management Commands in Linux. |
| 4, | **To study the various commands operated in vi editors in Linux.**   * **Shell programming: Introduction** * **Shell programming: Control Structure** * **Shell programming: Working with files** * **Shell programming: Examples/Exercise** |
|  |  |
|  |  |

**Exp No. 1**

***Introduction to Linux operating system: A Case Study***

***What is Linux🡪***

Just like Windows, iOS, and Mac OS, Linux is an operating system. One of the most popular platforms on the planet, Android, is powered by the Linux operating system. An operating system is software that manages all of the hardware resources associated with your desktop or laptop. To put it simply, the operating system manages the communication between your software and your hardware. Without the operating system (OS), the software wouldn’t function.

The Linux operating system comprises several different pieces:

* **Bootloader – The** software that manages the boot process of your computer. For most users, this will simply be a splash screen that pops up and eventually goes away to boot into the operating system.
* **Kernel –** This is the one piece of the whole that is called? Linux? The kernel is the core of the system and manages the CPU, memory, and peripheral devices. The kernel is the lowest level of the OS.
* **Init system –** This is a sub-system that bootstraps the user space and is charged with controlling daemons. One of the most widely used in its systems is the system? which also happens to be one of the most controversial. It is the init system that manages the boot process, once the initial booting is handed over from the bootloader (i.e., GRUB or Grand Unified Bootloader).
* **Daemons –** These are background services (printing, sound, scheduling, etc.) that either start up during boot or after you log into the desktop.
* **Graphical server –** This is the sub-system that displays the graphics on your monitor. It is commonly referred to as the X server or just X.
* **Desktop environment –** This is the piece that the users interact with. There are many desktop environments to choose from (GNOME, Cinnamon, Mate, Pantheon, Enlightenment, KDE, Face, etc.). Each desktop environment includes built-in applications (such as file managers, configuration tools, web browsers, and games).
* **Applications –** Desktop environments do not offer the full array of apps. Just like Windows and macOS, Linux offers thousands upon thousands of high-quality software titles that can be easily found and installed. Most modern Linux distributions (more on this below) include App Store-like tools that centralize and simplify application installation. For example, Ubuntu Linux has the Ubuntu Software Centre (a rebrand of GNOME Software? Figure 1) which allows you to quickly search among the thousands of apps and install them from one centralized location.

***History of Linux***

Unix is one of the most popular operating systems worldwide because of its large support base and distribution. It was originally developed as a multitasking system for minicomputers and mainframes in the mid-1970s. It has since grown to become one of the most widely used operating systems anywhere, despite its sometimes-confusing interface and lack of central standardization.

The real reason for Unix’s popularity? Many hackers feel that Unix is the Right Thing—the One True Operating System. Hence, the development of Linux by an expanding group of Unix hackers who want to get their hands dirty with their system.

Versions of Unix exist for many systems, ranging from personal computers to supercomputers such as the Cray Y-MP. Most versions of Unix for personal computers are quite expensive and cumbersome. At the time of this writing, a one-machine version of AT&T’s System V for the 386 runs at about $US1500.

Linux is a freely distributable version of Unix, originally developed by Linus Torvalds, who began work on Linux in 1991 as a student at the University of Helsinki in Finland. Linus now works for Transmeta Corporation, a start-up in Santa Clara, California, and continues to maintain the Linux *kernel*, that is, the lowest-level core component of the operating system.

Linus released the initial version of Linux for free on the Internet, inadvertently spawning one of the largest software-development phenomena of all time. Today, Linux is authored and maintained by a group of several thousand (if not more) developers loosely collaborating across the Internet. Companies have sprung up to provide Linux support, package it into easy-to-install distributions, and sell workstations pre-installed with the Linux software. In March 1999, the first Linux World Expo trade show was held in San Jose, California, with reportedly well over 12,000 people in attendance. Most estimates place the number of Linux users worldwide somewhere around the 10 million mark (and we expect this number will look small by the time you read this).

Inspired by Andrew Tanenbaum’s Minix operating system (another free Unix for PCs—albeit a very simple one), Linux began as a class project in which Linus wanted to build a simple Unix system that could run on a 386-based PC. The first discussions about Linux were on the Usenet newsgroup *comp. os. Minix*. These discussions were concerned mostly with the development of a small, academic Unix system for Minix users who wanted more.

**Components of Linux Systems –**

Linux operating system has primarily three components -

* **Kernel** – Kernel is the core part of Linux. It is responsible for all major activities of this operating system. It is consisting of various modules and interacts directly with the underlying hardware. The kernel provides the required abstraction to hide low-level hardware details in system or application programs.
* **System Library** – System libraries are special functions or programs using which application programs or system utilities access Kernel’s features. These libraries implement most of the functionalities of the operating system and do not require kernel module’s code access rights.
* **System Utility** – System Utility programs are responsible to do specialized, individual-level tasks.

**Basic features-**

Following are some of the important features of the Linux Operating System.

* **Portable** − Portability means software can work on different types of hardware in the same way. Linux kernel and application programs support their installation on any kind of hardware platform.
* **Open Source** − Linux source code is freely available and it is a community-based development project. Multiple teams work in collaboration to enhance the capability of the Linux operating system and it is continuously evolving.
* **Multi-User** − Linux is a multiuser system means multiple users can access system resources like memory/ ram/ application programs at the same time.
* **Multiprogramming** − Linux is a multiprogramming system that means multiple applications can run at the same time.
* **Hierarchical File System** − Linux provides a standard file structure in which system files/ user files are arranged.
* **Shell** − Linux provides a special interpreter program that can be used to execute commands of the operating system. It can be used to do various types of operations, call application programs. etc.
* **Security** − Linux provides user security using authentication features like password protection/ controlled access to specific files/ encryption of data.

--------------------------------------\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*-------------------------------------------------------------------------\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*-----------------------------------

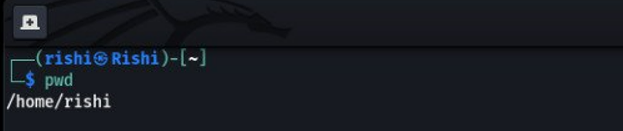
**Exp No. 2**

***Basic LINUX commands and their Use***

**Execution of various file directory handling commands**

**1) pwd COMMAND:**  
PWD – Print Working Directory. PWD command prints the full filename of the current working directory.

SYNTAX:  
The Syntax is  
pwd [options]



**2) cd COMMAND:**  
the cd command is used to change the directory.

SYNTAX:  
The Syntax is  
cd [directory | ~ |. / | ../ | – ]

**3) ls COMMAND:**  
ls command lists the files and directories under the current working directory.

SYNTAX:  
The Syntax is  
ls [OPTIONS]… [FILE]

OPTIONS:

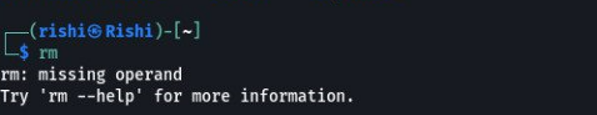
-l Lists all the files, directories and their mode, Number of links, owner of the file, file size, modified date, time, and filename.  
-t Lists in order of last modification time.  
-a Lists all entries including hidden files.  
-d Lists directory files instead of contents.  
-p Puts slash at the end of each directory.  
-u List in order of last access time.  
-I Display inode information.



**4) rm COMMAND:**  
rm Linux command is used to remove/delete the file from the directory.  
  
SYNTAX:  
The Syntax is  
rm [option.] [file | directory]

OPTIONS:

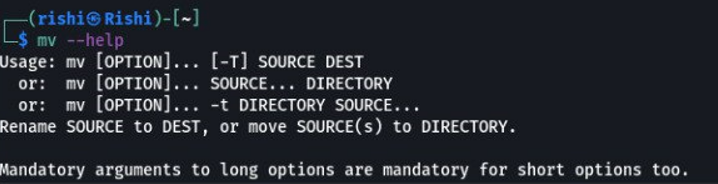
-f Remove all files in a directory without prompting the user.  
-I Interactive. With this option, rm prompts for confirmation before removing any files.



**5) mv COMMAND:**  
mv command which is short for the move. It is used to move/rename files from one directory to another. mv command is different from the cp command as it completely removes the file from the source and moves to the directory specified, where the cp command just copies the content from one file to another.

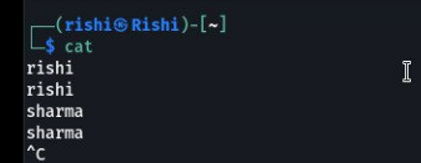
SYNTAX:  
The Syntax is  
mv [-f] [-i] old name newname

OPTIONS:  
-f This will not prompt before overwriting (equivalent to –reply=yes). mv -f will move the file(s) without prompting even if it is writing over an existing target.  
-I Prompts before overwriting another file.



**6) cat COMMAND:**  
cat Linux command concatenates files and prints them on the standard output.  
SYNTAX:  
The Syntax is  
cat [OPTIONS] [FILE]…  
OPTIONS:  
-A Show all.  
-b Omits line numbers for blank space in the output.

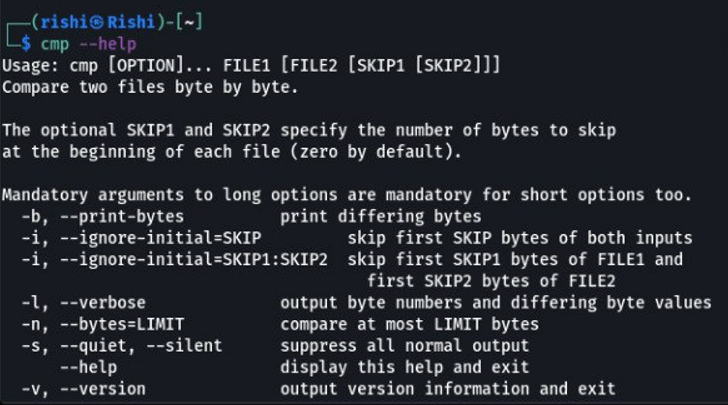
-E Displays a $ (dollar sign) at the end of each line.  
-n Line numbers for all the output lines.



**7) CMP COMMAND:**  
CMP Linux command compares two files and tells you which line numbers are different.

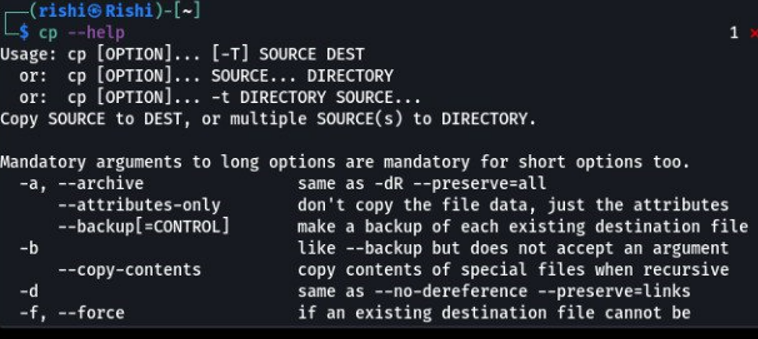
SYNTAX:  
The Syntax is  
cmp [options.] file1 file2  
  
OPTIONS:

– c Output differing bytes as characters.  
– l Print the byte number (decimal) and the differing byte values  
(octal) for each difference.  
– s Prints nothing for differing files, return exit status only.

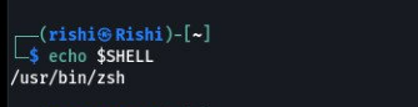


**8) cp COMMAND:**  
cp command copies files from one location to another. If the destination is an existing file, then the file is overwritten; if the destination is an existing directory, the file is copied into the directory (the directory is not overwritten).

SYNTAX:  
The Syntax is  
cp [OPTIONS]… SOURCE DEST



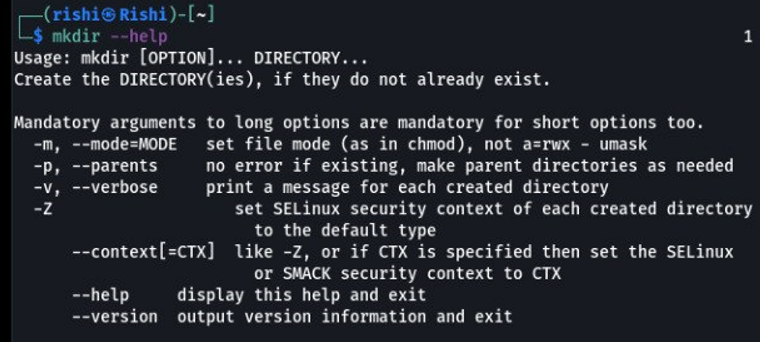
**10) echo COMMAND:**  
echo command prints the given input string to standard output.  
SYNTAX:  
The Syntax is  
echo [options.] [string]



**11)mkdir COMMAND:**  
mkdir command is used to create one or more directories.

SYNTAX:  
The Syntax is  
mkdir [options] directories  
OPTIONS:

-m Set the access mode for the new directories.  
-p Create intervening parent directories if they don’t exist.  
-v Print help message for each directory created.

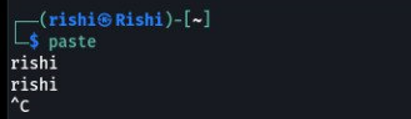


**12) paste COMMAND:**paste command is used to paste the content from one file to another file. It is also used to set the column format for each line.

SYNTAX:  
The Syntax is  
paste [options]

OPTIONS:

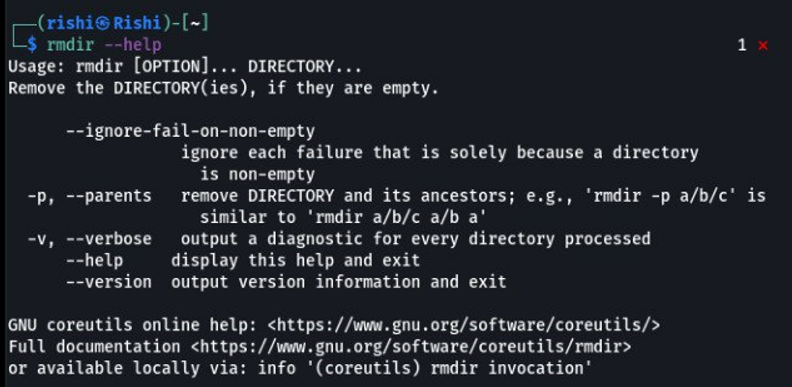
-s Paste one file at a time instead of in parallel.  
-d Reuse characters from LIST instead of TABs.



**13) rmdir COMMAND:**rmdir command is used to delete/remove a directory and its subdirectories.  
SYNTAX:  
The Syntax is  
rmdir [options.] Directory

OPTIONS:

-p Allow users to remove the directory dir name and its parent directories which become empty.



**14) Cl COMMAND:**

The calendar is command is used to display the calendar of any specific month or an entire year. Simple type cal on the prompt and calendar of the current month will be displayed as

**Sub Command:**

1. **$ cal 08 2008**
2. **$ cal 2008**
3. **$ cal 2008 | more**

**date: Displaying the system date**

1. **$date +%d**
2. **$date + “%d %m”**

**Some of the other useful format specifiers are listed below:**

**d-The day of the month**

**y-the last two digits of the year**

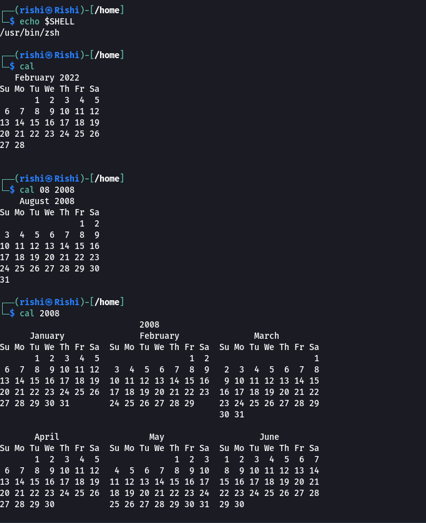
**H- Hour**

**M-Minutes**

**S-Seconds**

**D- the date in mm/dd/yy format.**

**T- the time in the format hh:mm: ss.**

****

**15) bc COMMAND:**

bc: The command line calculator: -

The bc command in Linux is a command-line calculator. In addition to performing simple math functions, it can also perform conversions between different number systems, perform several scientific math functions, and can even run programs that you write and save in a text file. Here's how it works. At the command prompt, execute the following command.

**Sub-command:**

**1. $ bc -l**

((50.1 + 65.12) \* 30.3)

3491.166

l (2000)

7.60090245954208236147

**2. obese = 16**

Now, multiply the number that you want to convert by 2.

14 \* 2

1C

Oh, but wait, you wanted that answer in binary. No problem, just change the obese.

**3. obase=2**

14\*2

11100

To return to normal, set the obese back to 10. Execute the following to find out the square root of a number using the sqrt function available in the math library.

**4. sqrt (40)**

6.32455532033675866399

To return to the command prompt, press Ctrl+d.





**15) echo COMMAND:**

echo: Displaying a message

This command is used to display a message on the terminal. For example, type the following command and press Enter key. The message typed inside double quotes will be displayed on the screen.

$echo “Hi, I am learning Linux commands “

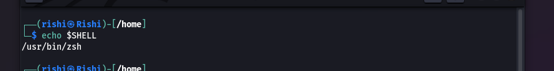
The echo command can be used for displaying variable values. For example, type the following:

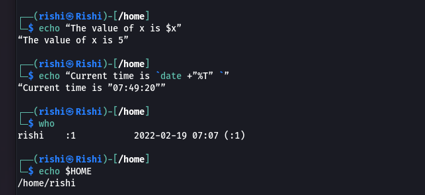
$x=5

$ echo “The value of x is $x”

**Sub-command**

**1. $echo “Current time is `date +”%T” `”**





**16) passwd COMMAND:**

passwd: changing your password

This command is used to change the password of the current login account. Type passwd and press Enter key.

$ passwd

Changing password for user rishi.

Changing password for rishi

(current) UNIX password:

New password:

Retype new password:

passwd: all authentication tokens updated successfully

**17) who COMMAND:**

who: Knowing the list of users currently logged in

This command is used to display the list and information of the users currently logged into the system. Type,

$who

To know about yourself, use the arguments am and I as an argument with who.

**18) pwd COMMAND:**

PWD: Checking your Current directory

Once you are logged in you can move around from one directory to another, but at any point in time, you will be located only in one directory. This directory is known as the current directory. To know your current working directory, type PWD command, and current working directory will be displayed



**19) mkdir COMMAND:**

mkdir: Creating a directory

Directories in Linux can be created using the mkdir command. The command can be followed with the names of the directories to be created. To create a directory named cities, type the following command and press Enter key.

$mkdir cities

You can create multiple directories with one mkdir command.

$mkdir cities talukas villages

This creates three sub-directories named cities, talukas, and villages.

cd: change directory

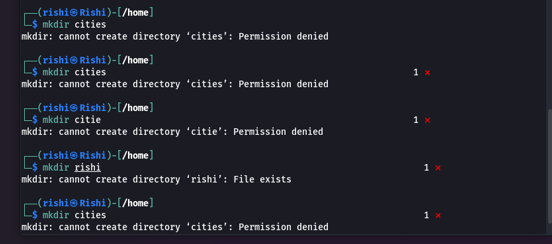
To create a directory within a directory named cities, first, we will have to change the directory. This can be done using the cd command.

$cd cities

$pwd

/home/student/cities

$mkdir Ahmedabad Surat Baroda

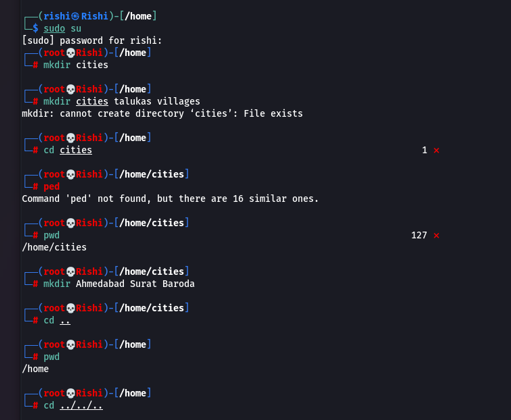


**20) cd COMMAND:**

$cd.

In the above command double dot (..) stands for the parent directory. Note that there should be one space between the cd command and double dot. To change your current working directory to home directory, if you are in /home/student/cities/Ahmedabad, type either

cd .. thrice one by one as shown below or type $cd ../../.



**21) cat COMMAND:**

Cat command is mainly used to display the contents of a small file on the terminal. It can also be used to create a file, concatenate two files and append contents to a file

**Sub-command**

**1. $ cat > about\_cities**

This directory contains information about three cities of Gujarat state, Ahmedabad, Baroda, and Surat.

[Ctrl-d]

File naming conventions

On most Linux systems today, a filename can consist of up to 255 characters. Unlike Windows, files in Linux can practically consist of any ASCII character except the backslash (/) and the NULL character. Any other control characters or unprintable characters are permitted. The following are the valid names in Linux.

. file1 file2. ^file^-++ -{}() test$# ab.cd.ef

However, it is recommended that a file name should contain only alphabetic characters, numerals, period (.), hyphen ( - ), and underscore (\_). Linux is strictly case sensitive and thus file1, File1, FILE1 are three different file names and they can coexist in the same directory. You might have noticed that a file created by us using a cat command is not having any extension still now error was reported. Linux operating system does not impose any rules for framing file extensions. In all cases, its application poses a restriction. Thus, the C compiler expects C program filenames to end with .c Oracle expects .sql extension, and so on. Thus, a file in Linux may or may not have an extension. Although it is recommended to provide an extension as it helps in identification. For example, while creating a shell script if a file name shall be appended with .sh.

**2. $cat >> about\_cities**

**3. $ cat test1 test2 > newfile**

**22) ls COMMAND:**

ls: Listing files and Directories

Till now we have seen creating directories and files. Now how to list the directories and files already created by us? ls command in Linux gives us the list of the contents in the current or specified directory. ls command also has a lot of options. Let us begin with the plain and simple ls without any options.

$ ls

about\_cities Ahmedabad Baroda Surat

Note that files and directories that are presented in the current directory named cities are listed in alphabetical order.

Let us now create a file and then check whether we can list it or not?

**Sub-command**

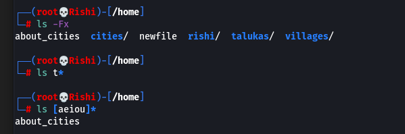
**1. $ ls –a**

**2. $ ls –R**

**3. $ ls -Fx**

**4. $ ls t\***

**5. $ ls [aeiou]\***



**23) cp COMMAND:**

cp: Copying a file

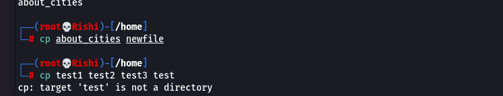
The cp command copies a file or group of files specified as an argument to it. It creates an exact image of a file on the disk with a different file name. cp command needs at least two arguments. The first argument will be considered as a source file and the second will be considered as a destination file.

For example, to create a copy of file about\_cities execute the following command

**Sub-command**

**1. $ cp about\_cities new file**

**2. $ cp test1 test2 test3 test**



**24) mv COMMAND:**

mv: Renaming files and moving files

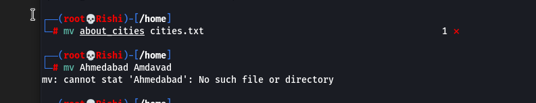
The move command can be used for renaming a file or directory.

For example, to rename the file about\_cities to cities.txt, execute the following command.

**Sub-command**

**1. $ mv about\_cities cities.txt**

**2. $ mv Ahmedabad Amdavad**



**25) wc COMMAND:**

WC: Counting lines, words, and characters in a file

who is a simple but useful command? It counts the number of lines, words, and characters in the specified file or files. WC command comes with three options -l, -w, and –c for counting lines, words, and characters respectively. For instance, to count the number of lines in file cities.txt, execute the following command.

**Sub-command**

**1. $wc –l cities.txt**

**2. $wc –l –w -c cities.txt**



**26) chmod COMMAND:**

**chmod: Changing permissions**

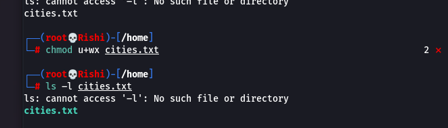
**As an owner of a file, you can change the permissions of a file using the chmod command. For instance, we have seen that owner has read and written permissions on the file. To make cities.txt file read-only file, execute the following command**

**Sub-command**

**1. $chmod ugo-w cities.txt**

**2. $chmod u+wx cities.txt**

**3. $ls –l cities.txt**



**28) gzip COMMAND:**

gzip: Compressing files

Many times, to send a large file as an attachment or to save disk space, we need to compress the file. Every Linux system comes with compression and decompression utilities. Select a file to be compressed and before compressing it check the size of the file using the wc command with the –c option.

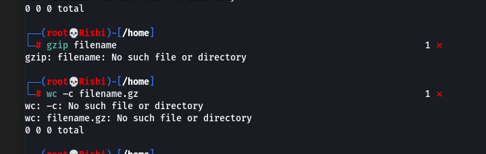
**Sub-command**

**1. $gzip filename**

**2. $gunzip: Decompressing files**

**3. $gunzip filename.gz**

**4. $gzip –d filename.gz**



**29) mount COMMAND:**

mount: attaching file system

In the earlier chapter, we have discussed that the Linux file system follows a tree structure with forward-slash (/) designated as a root. The mount command is used to mount a new file system to an existing file system. For instance, you would like to copy data from a USB pen drive to a hard disk using the Linux Operating system. Attach a pen drive, Open My computer and if you find that your machine has not detected the USB pen drive, you need to attach the file system of a new device to the existing file system of the Operating system. We can attach the new do it using the mount command. Note that the mount command can only be used by a root user. So, get logged in as a root user and try the following commands**.**

**Sub-command**

**1. #mkdir /mnt/pendrive**

**2. #fdisk –l**

**3. #mount /dev/sda1 /mnt/pendrive**

**4. $umount:unmounting file system**

**5. $umount /pendrive**

**30) head COMMAND:**

head: Displaying top lines of the file

The head command as the name implies displays the beginning lines of the file. When used without an option it displays the first ten lines of the file. For instance, execute the following command and you will find that the first ten lines of the files will be displayed.

**Sub Command**

**1. $ head -5 users.txt**

**2. $ tail cities.txt**



**31) cut COMMAND:**

cut: Cutting a file vertically

To understand how the cut command works let us create a file containing a list of your friends, their phone numbers, and email addresses as shown below.

$gra student.lst

**Sub-command**

**1. Cutting fields (-f)**

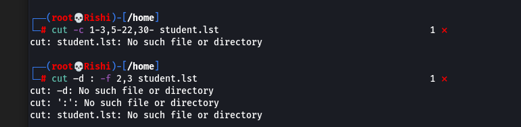
**2. $ cut -c 1-3,5-22,30- student.lst**

**3. $ cut –d : -f 2,3 student.lst**

**4. Cut –d “ “ –f 6,7 student.lst**

**5. $ cut –d : -f 1,4- student.lst > contacts**

**6. $cat contacts**



**32) paste COMMAND:**

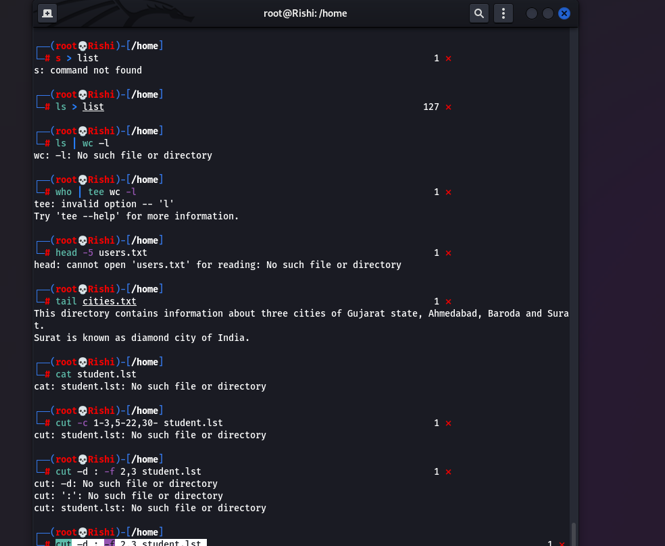
paste: Pasting Files

The paste command is a way of pasting two files together. Ensure that you have the same number of lines in every file - if not, the paste will paste from the top of the file. Ensure that both the files exist in your working directory

In the previous section, we have used the cut command to create two files stream and contacts. Using the paste command, we can fix them laterally.

**Sub-command**

**1. $paste stream contacts**



**33) sort COMMAND:**

sort: Ordering a file

The sort command is used for the ordering of data in ascending or descending sequence. Like the cut command, it also identifies fields and can sort on specified fields.

Let us sort the contents of the file student. lst. Execute the following command.

**Sub-command**

**1. $ sort student.lst**

**2. $ sort -r student.lst**

**3. $ cut -d : -f 2 student.lst | sort**

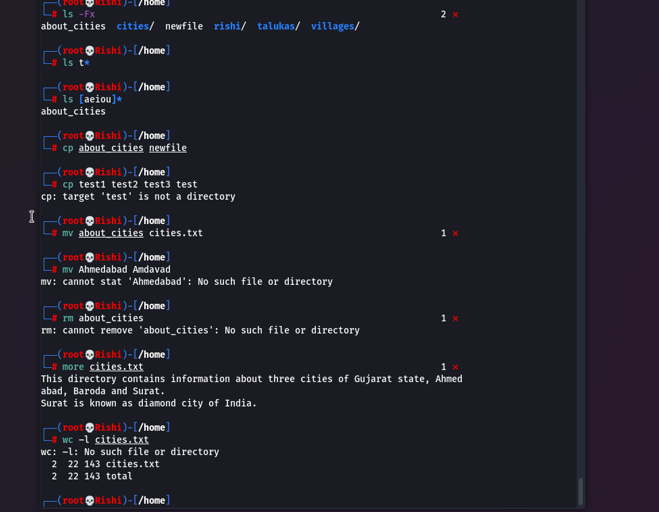
**34) uniq COMMAND:**

uniq: Locate unique lines from the file

The uniq command in Linux extracts unique lines from the input provided to it. For example, let us find out the month on which you have created files in your working directory. Execute the ls –l command to display the files in the current directory. Notice that the 6th column contains the month name. So, if we cut the 6th column from the output of the ls –l command we can get the list of Months. But you might have created multiple files in the same month. So, the month name will get repeated in the output. To display unique month names, the output with repeated months can be transferred to the uniq command and we will get the result as shown in the following command

**Sub-command**

**1. $ ls -l | tr -s ' ' | cut -d " " -f 6 | uniq |sort**



**35) tr COMMAND:**

tr: Translating characters

So, the filter commands we have been discussing have been handling either lines or columns. The tr (translate) command manipulates individual characters in a line. The translate command is used to translate strings or patterns from one set of characters to another. For example, suppose we have a file with lowercase letters, and we want to translate that all to uppercase letters, the simplest way to do that is to use the translate command. Cat-ting our file (columns.txt) and then piping the output of the cat command to the input of the translate command causing all lowercase names to be translated to uppercase names.

**Sub-command**

**1. $ cat student.lst | tr '[a-z]' '[A-Z]'**

**2. Cat student.lst | tr [ab][c] [s][t]**

**3. Tr [“d ab”] [“sonal”]**

**4. $ cat student.lst | tr '[a-z]' '[A-Z]' > UpCase\_student.lst**

**5. $ tr -s ' ' < student.lst**

**36) grep COMMAND:**

grep: Searching for a pattern

The name of this command comes from a command in the Unix text editor -ed- that takes the form g/re/p meaning search globally for a regular expression and display lines where instances are found. grep can be used to do a whole host of tricks and magic. We can use it as either a filter or to look inside files. It also uses regular expressions.

Let's start with using grep to look inside files. To display the details of a student named “Harshit” from the student.lst file, following command, can be used.

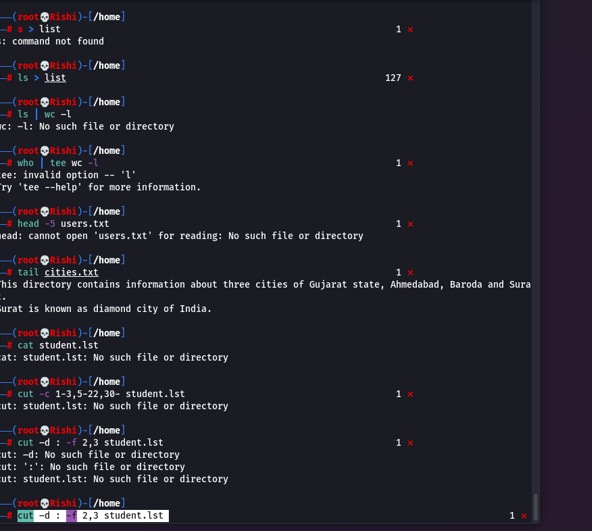
***Sub-command***

**1. $ ls –l | grep “stream”**

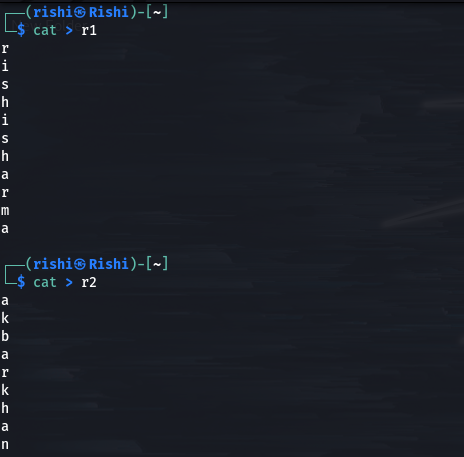
**2. $ ls –l | grep –i “stream”**

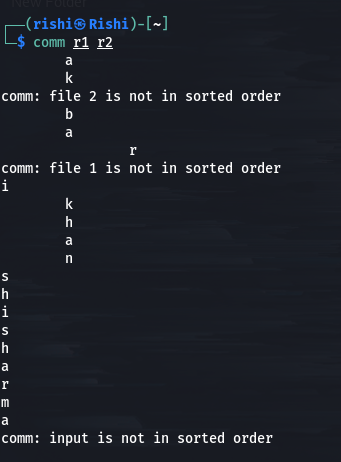
**3. $ grep –v "HARSHIT" student.lst**

**4. $ grep -ic "science" student.lst**



**37) comm COMMAND:**





***Examples Exercise***

***Exercise -1***

Simply type

$ pwd

and press enter key and read on :) title: pwd

Can you see the output similar to */home/yourname* ? cool,you have found your current working directory.Congrats,You have joined exclusive club of linux commandline users :)

As you realized typing

pwd

will display your current working directory.Yeah,your home is a directory. Now lets try to create a new directory.type the following on the prompt

mkdir -v dir1

and press enter key. title: mkdir

Did it say?

*mkdir: created directory dir1*

Wow,now you created a new directory. Lets say you want to create more than one directory instead of invoking mkdir multiple(three) times-like.

mkdir -v dir2

mkdir -v dir2/dir3

mkdir -v dir2/dir3/dir4

you can simply use

mkdir -vp dir2/dir3/dir4

"-p" option will create parent directories for "dir4" as needed. In this case,it creates dir2,dir3 automatically.Now we have created 4 directories.How to view them?

To view type 'ls' and press enter

ls

title: ls

listed dir1 dir2 as directory content right? Thats exactly what we wanted

`dumb tutor: yes,the guy with blue-t-shirt,

Yeah, you ,why you look so confused?`

`blue-t-shirt:I created 4 directories,

where is the missing dir3,dir4?`

Good question.They are created inside dir2 they won't be listed with simple command like ls.you need to use "complex" command to view them. Try this:

ls -R

really "complex" isn't it :P ,btw -R stands for recursive.

Okay,we have created a new directories and listed them.Now lets move into a new directory.

cd dir2

title: cd

cool,you have changed to dir2 Now confirm this location by using previously learned pwd command.To move into next directory dir3

cd dir3

will place you under "dir3" directory.

Tips and tricks: Typing

cd ..

will move to parent directory.i.e dir2. Now type,

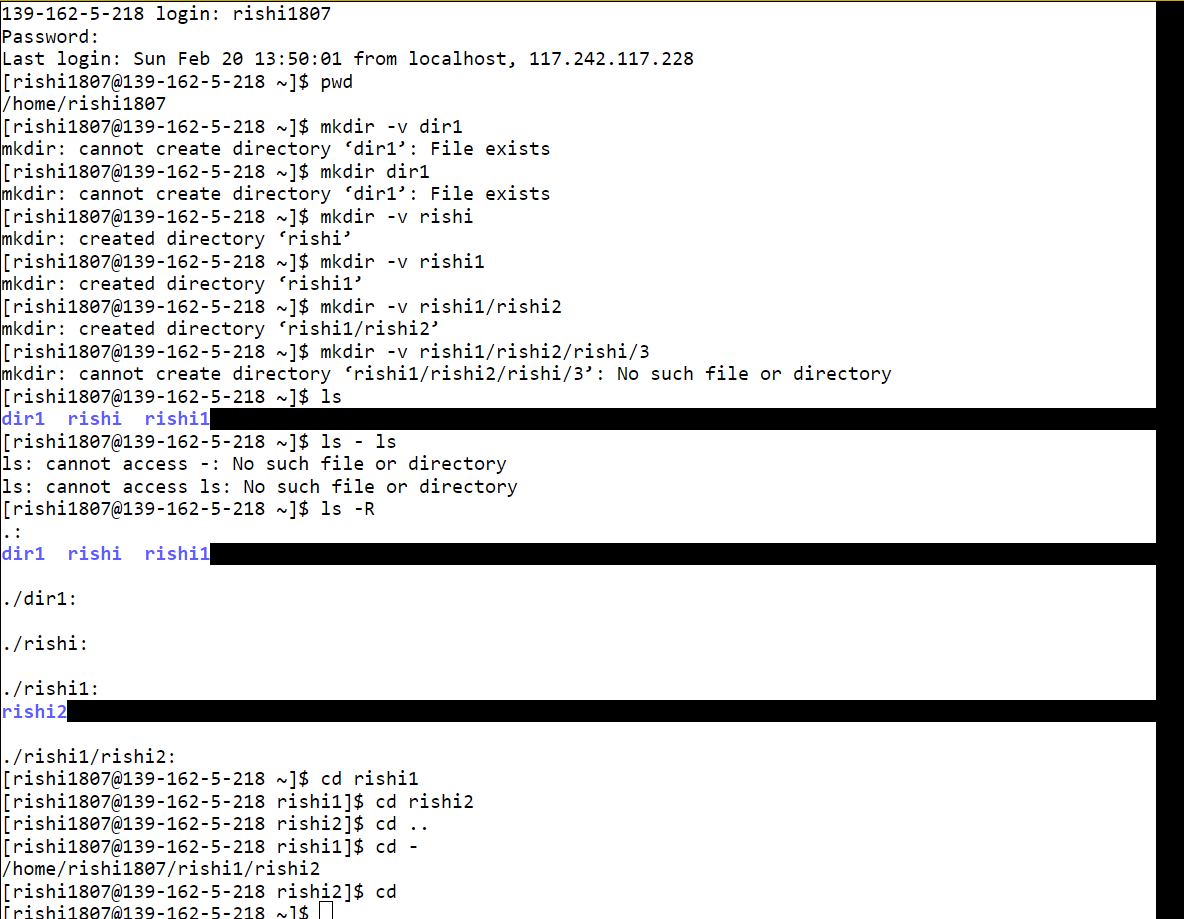
cd -

will move you to previous working directory i.e dir3 Cool ,isn't it? and a simple

cd

will move to the your home directory.

***Output***



***Exercise -2***

Lets learn to create a new file,

touch file1.txt

and press enter key and read on :)

title: touch

touch command will create a new file or change time stamp of an existing file. Now try again,

touch file1.txt

this time it will change file1.txt created/last access and modified time to current time.

touch file2.txt

will create an empty new file ,if the file is not already exists. to view directory contents ,you can also use

dir

title: dir

dir is used to list directory contents.Yeah,as you guessed it correctly , dir is equivalent to ls -C -b (I know you didn't guess that :P)

that is, by default files are listed in columns, sorted vertically, and special characters are represented by backslash escape sequences. To clear a screen,the command is

clear

title: clear

Viola! terminal screen is cleared!!! Lets print some message on the terminal,

echo "hello"

title: echo

Cool! the message is displayed on the screen. Lets redirect the message to a new file instead of screen.

echo "hello" > hello.txt

To append data you must use >> not just >

echo "linux" >> hello.txt

echo "world" >> hello.txt

Done.To view the file content ,do

cat hello.txt

title: cat

so now you have viewed the file content.cat is used to display the entire file content.

To view only first two lines from the file

head -2 hello.txt

title: head

see,it showed us first two lines from files. By default,head will display the first 10 lines when you run,

head hello.txt

Now how to view last two lines?.Its simple,use tail

tail -2 hello.txt

title: tail

cool. Thus head will be used to display lines from begining and tail will be used to display last few lines. As with head

tail hello.txt

by default will display last 10 lines from the line.

Lets check some stats of the files and directories we have create so far.

stat hello.txt

title: stat

carefully examine few important fields the output. The first line shows the filename.second line says its a regular file with size as 18.Third line shows Inode number and no.of links to that inode.

Fourth one,says owner(Uid),group(Gid) who has read-write permission but other have read permission.Final three lines show access,modified and change time.They mean:

access - when the file was last accessed/read.

modified - when the contents was last

modified written.

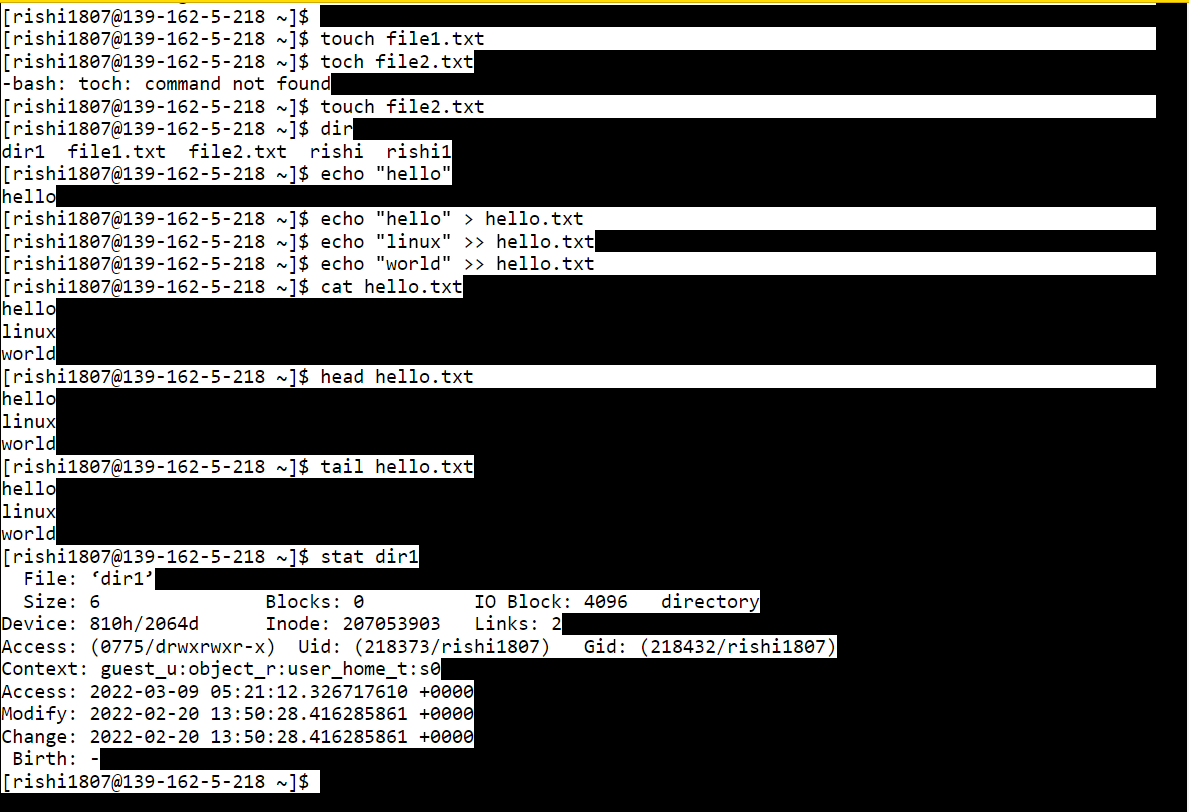
change - denotes changes to files metadata

like changing user permission.

Now lets do a stat on directory.

stat dir1

***Output***



***Exercise -3***

Now check this command

du

title: du

it displays the disk usage of current directory.(Please note the current total of du output).Use the h switch to output in a human readable format and the x switch to exclude other file systems and ~ denotes your home.

du -xh ~

Tips and tricks:

du can take a long time so you can specify the max.directory depth using "--max-depth" option.

du --max-depth 3 ~

Now lets copy hello.txt to dir2 directory.

cp -v hello.txt dir2

title: cp

now file is copied to new location.Now compute the usage again using, du now you should see usage has been increased by file size.

Tips and tricks:

cp -v hello.txt dir2/file2.txt

This will copy hello.txt into dir2 at the same time, rename it as "file2.txt".

cp -vr dir2/\*.txt dir2/dir3

This will copy all files ending with ".txt" from dir2 into dir2/dir3.

cp -vr dir2/dir3 .

This will copy the directory named "dir3" to current directory.

Use ls,it should show you dir3.

now we have copied few files,how do we verify its file integrity?simple cat should be enough.But If its large file or binary file,we can't use cat.We have to use,

md5sum hello.txt

title: md5sum

b8d5079c5d6a9dbb3294b31d318d74c0 is the calculated checksum for a file.This helps with detecting accidental or deliberate file corruption.

When transfering a file from machine to another or downloading files from internet,to verify the file integrity compare md5sum on source and destination machines,

md5sum dir2/hello.txt

should be same as

md5sum hello.txt

now lets move to another command,

mv hello.txt dir2/dir3/dir4/hi.txt

title: mv

will move a file into directory dir4 and names it as hi.txt. so how mv is different from cp?.Try ls it will not show hello.txt.

When you use cp there exists two copies of a file (similar to copy-paste "ctrl-c" and "ctrl-v") with mv there is one copy (its cut-paste ctrl-x and ctrl-v). unlike (cp,rm) other commands mv don't need "-r" for directories.

create a new directory dir5

mkdir dir5

now

mv dir2/\*.txt dir5

mv dir5 dir50

will move all "\*.txt" files under dir2 into dir5. then rename the directory "dir5" as "dir50".

with mv command we moved hello.txt under dir4,instead of accessing them as dir2/dir3/dir4/hi.txt everytime,we can create a link and after that,you can access or edit dir2/dir3/dir4/hi.txt file as simply hello

ln dir2/dir3/dir4/hi.txt hello

title: ln

Great! you have created a link. There are two types of links, hardlinks. where a same inode pointed by two different names and softlinks which work more like shortcuts.

Hard links are created by default.

stat hello

and perform

stat dir2/dir3/dir4/hi.txt

see both uses same inode and link count shown as 2. Soft links are created using the s switch.

ln -s dir2/dir3/dir4/hi.txt softlink

again do

stat softlink

and examine its output.New inode is created for this new symbolic link "softlink" but link count remains as 1. To remove individual file use

rm -i file2.txt

title: rm

will prompt you with a message.rm: remove regular empty file 'file2.txt'? type y to delete the file.To remove directory, first remove it's contents using option "r",

rm -ri dir50/\*

Tips and tricks:

If you want to remove files content without begin prompted for confirmation use -f option. It's extremely dangerous to use "rm -rf",because you may delete very important files by mistake-so make sure you delete correct files before running rm -rf"

rm -rf junk/\*

rmdir dir50

***Output***

******

***Exercise -4***

Now check this widely used

ps

title: ps

output is nothing but a snapshot of the currently running processes. lets create a new process.

sleep 60 &

title: ps

can you see process id on screen?Now again do

ps

you can see the sleeping process,now-right? lets see how to stop/kill this process replace 12345 will your sleeping process id,you got above

kill 12345

title: kill

Check again the running process list with

ps

sleeping process is Gone! right?

kill 12345

Tips and tricks:

Sometimes process won't die with simple kill command,in such cases scream die!die!die! while running kill command.(hehe..just kidding) you have to use "-9" option.

kill -9 12345

start two process like

sleep 30 &

sleep 30 &

checking with "ps",we can see we have two process named sleep,now type

killall sleep

title: killall

did it gave an output like

Terminated sleep 30

right?thus killall terminates processes by process name.

Tips and tricks:

killall -u webminal

This kills only processes owned by user "webminal"

killall -w find

Wait for all find process to die. killall checks once per second if any of the killed processes still exist and only returns if none are left. Note that killall may wait forever if the signal was ignored, had no effect. To find a process id (pid) of a process you can use,

pidof bash

title: pidof

provides the process ID of a running program bash

Tips and tricks:

pidof -s bash

returns only one process id , instead of all process running as bash You can adjust the pripority of your process by starting a process like,

nice -n 19 sleep 30 &

title: nice

runs a program with modified scheduling priority. Nice runs a command with an adjusted niceness, which affects process scheduling.Nicenesses range from -20 (most favorable scheduling) to 19 (least favorable-the affected processes will run only when nothing else in the system wants to).Only root can increase the priority ,for example setting process nice to -20 others can lower the priority of processes they own.

how to adjust priority of currently running process with pid 12345?

renice -n 19 12345

title: renice

changes priority of running processes.

renice +1 3176

3176: old priority 0, new priority 1

renice +4 3176

3176: old priority 1, new priority 4

Only root can increase the priority ,for example setting process nice to -20.others can lower the priority of processes they own.

note with renice command,Non super-users can not increase scheduling priorities of their own processes,even if they were the ones that decreased the priorities in the first place.

To adjust priority for all process owned by a user "webminal",

renice +1 -u webminal

to display running process ,you can also use

top

title: top

see it provides a dynamic real-time view of a running system. spend sometime ,examining the output.To quit from the top command,press q. To display commands in a tree like structure,type

pstree

title: pstree

display a tree of processes,to display pid , use -p option with pstree.

pstree -p

below command will let us know how long it took to complete a command.

time ls -l

title: time

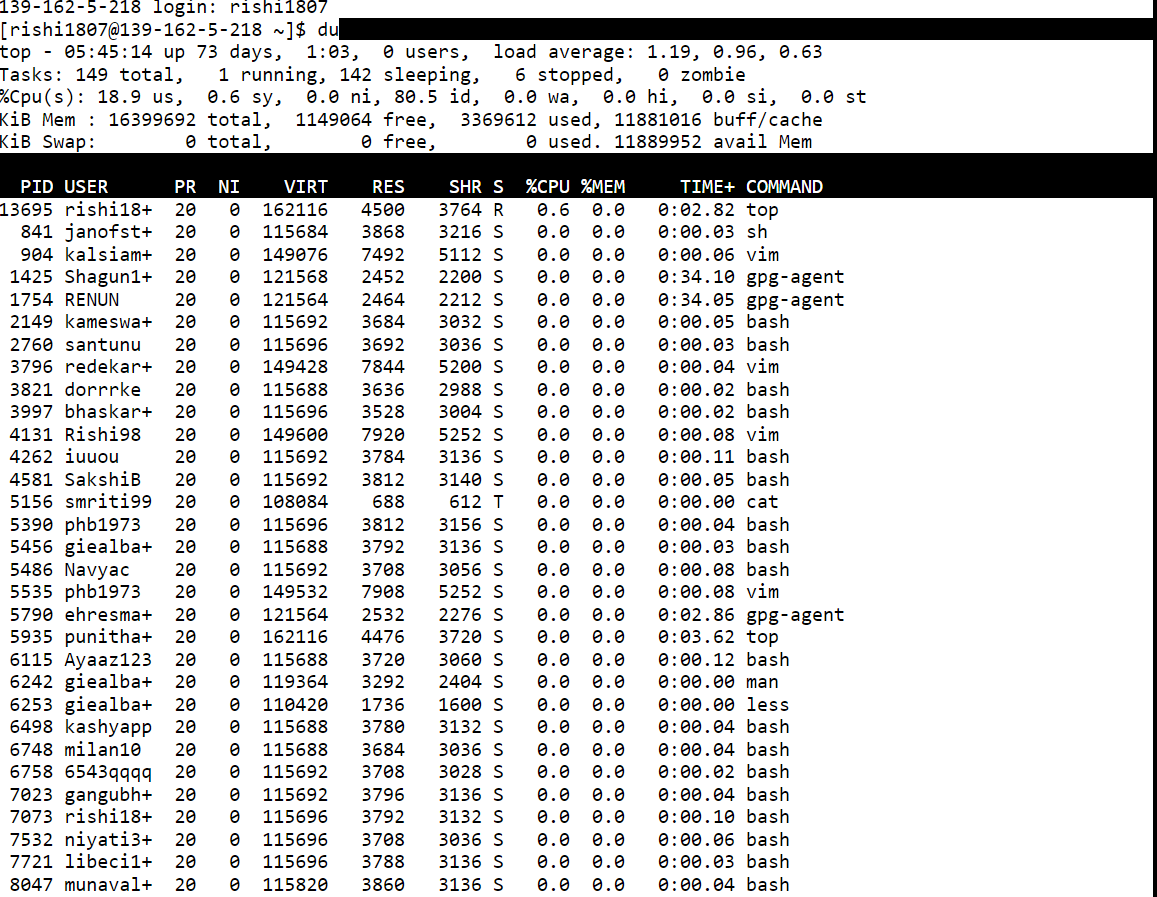
time gives statistics about the program it ran.

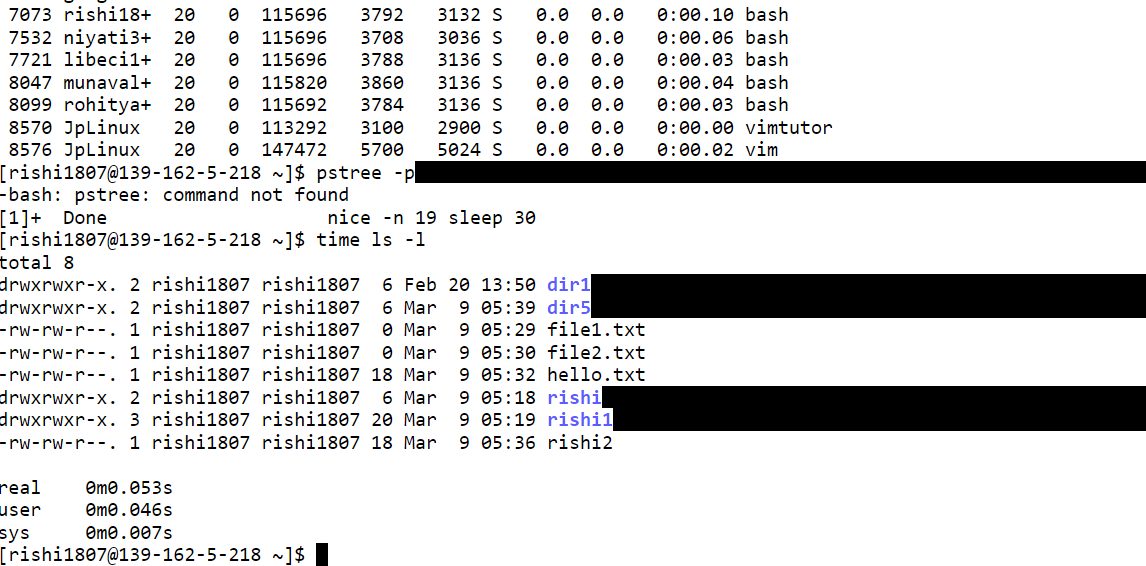
real - the elapsed real time between invocation and termination.

user - the user CPU time .

sys - the system CPU time .

***output***

******

******

***Exercise -5***

Lets try this widely used

grep "linux" hello

title: grep

grep searches for matching words or line on the file To search entire directory of files, supply the directory name

grep -r 'Hello' .

By default grep is case sensitive (a is not the same as A) but you can ignore case by using the i switch

grep -i 'lINUX' hello

Tips and tricks:

To display line numbers:

grep -n 'linux' hello

To display lines that don't match the pattern:

grep -v 'world' hello

To count no.of words,lines and character on a file use wc hello title: wc

thus wc counts lines/words/bytes in a file. first field is no.of lines , second column is no.of words and third column denotes no.of bytes.

Tips and tricks:

wc -L hello

to find the length of longest line in the file.Lets create a file with some contents with echo.

echo -e "col1 col2 r1\ncol5 col6 r2\ncol3 col4 r3 " >> new.txt

echo -e "Hello\nlinux\nProgrammers paradise" >> linux.txt

Okay,you have two files new.txt,linux.txt now,lets cut it ! :D

cut -f1 -d' ' new.txt

title: cut

So it extracted the first column from the file and to extract the third column

cut -f3 -d' ' new.txt

As you have noticed -f can be used to mention the column number and -d is used to specify the delimiter.now we have seen how to cut a file lets check out the another one ,

paste hello new.txt

title: paste

paste merges the lines of files

Tips and tricks:

to paste one file at time,

paste -s hello new.txt

In order to sort a file content, we could use

sort new.txt

title: sort

File contents are sorted.Remember,we have two files new.txt and linux.txt.lets compare them

diff hello linux.txt

title: diff

File contents are sorted.Remember,we have two files new.txt and linux.txt.lets compare them

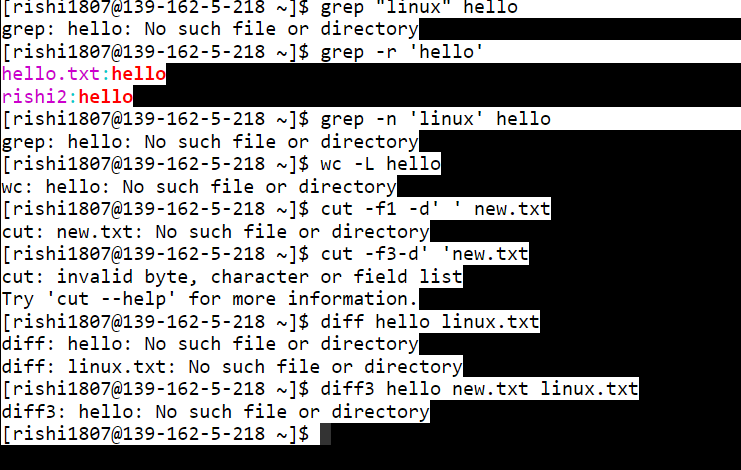
diff hello linux.txt

Compare files line by line. < denotes first file(hello) and > denotes second file(linux.txt). you can compare three files with

diff3 hello new.txt linux.txt

title: diff3

***output***

******

***Exercise -6***

Lets begin with a command that manipulates pathname,

dirname dir2/dir3/dir4/hi.txt

title: dirname

strip non-directory suffix from path name ,gave you the output

dir2/dir3/dir4

lets use the same path with different command this time

basename dir2/dir3/dir4/hi.txt

title: basename

this strips directory and suffix from pathname and gives the last entry.

hi.txt

Pretty useful commands :D lets change file access permission

chmod -v 666 file1.txt

title: chmod

You should have seen a output like mode of file1.txt changed to 0666 (rw-rw-rw-)

That will set the file "file1.txt" to be "world writeable".This means the owner, group and others can read and write into file. The same effect can be achived (remember you can verify it by using stat file1.txt) by

chmod a+rw file1.txt

where as below makes it so that no one can read or write into this file, not even it's owner!

chmod a-rw file1.txt

with next command only owner can read or write into this file. chmod u+rw file1.txt. Tips and tricks: To change permission for more than one file use the -R switch

chmod -R 644 ~/chmod\_dir

now to change file owner , chown root file1.txt title: chown

chown: changing ownership of file1.txt: Operation not permitted

oh,thats expected error message,you can use chown only as root user, but anyway thats the syntax/usage of chown command.Now we can change file owner and group,by chown root:staff file1.txt

Note: Rest of commands on lesson6 are expected to fail with below message, they are listed here for sake of completeness.

chown: changing ownership of file1.txt: Operation not permitted

This does the same, but additionally changes the group to "staff"

Tips and tricks:

To change permission on all files and sub-directories, use the -R switch.

chown root:staff -R ~/dir2

Use option "--from" to change files that belongs to specific user group.

chown --from=webminal:webminal root:staff -R ~/dir2

will change the files the belong to webminal user and webminal group to root and other user files left as it is.Lets change the group alone-

chgrp root file1.txt

title: chgrp

chgrp: changing group of `file1.txt': Operation not permitted

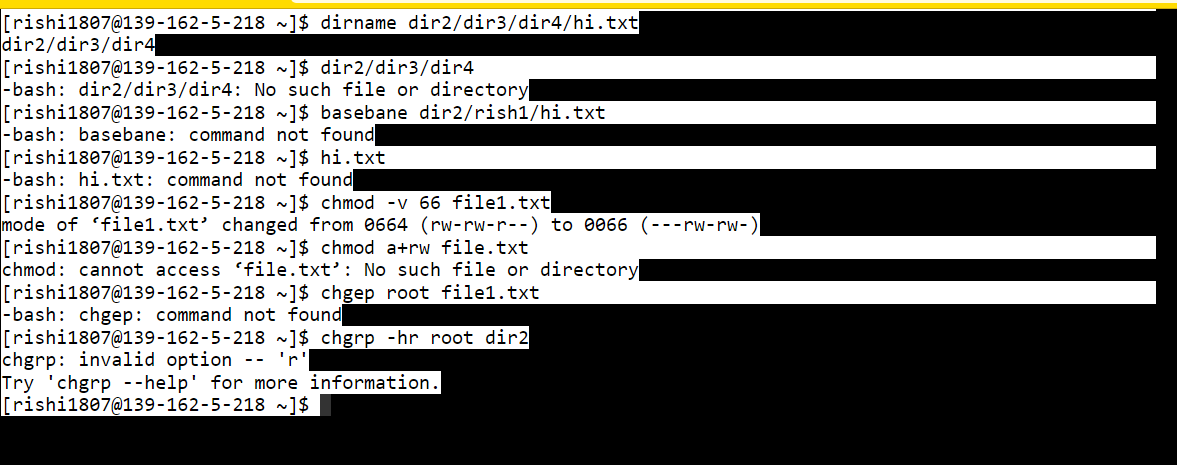
hehe..again thats expected error message :) ,you can use chgrp only as root user, but anyway thats the syntax/usage of chgrp command.

Tips and tricks:

To change the group of dir2 and subfiles to "root".

chgrp -hR root dir2

***output***

******

***Exercise -7***

Often we need to figure out a file type,for such task, we can use

file linux.txt

title: file

determines the type of a file as ASCII text

file /dev/null

/dev/null: characater special says,its a character device.

Tips and tricks:

You can also find about file system details of special devices. (below command listed here for sake of completeness, you will get permission denied error message)

file -s /dev/sda2

says /dev/sda2: x86 boot sector, code offset 0x52, OEM-ID "NTFS ", sectors/cluster 8, reserved sectors 0, Media descriptor 0xf8, heads 255, hidden sectors 161792, dos < 4.0 BootSector (0x80)

often we need to find the location of a certain file

whereis ls

title: whereis

you should see an output

ls: /bin/ls /usr/share/man/man1p/ls.1p.gz /usr/share/man/man1/ls.1.gz

whereis command will locate source files and binaries,lets see another example,finding source file

whereis stdio.h

will give you

stdio: /usr/include/stdio.h /usr/share/man/man3/stdio.3.gz

Assume,you have installed two version a php (php4 and php5),when you simply type

php

which version will get executed?we don't know. In order to find it out,we use

which php

title: which

To locate a binary file or if you have two version of a binary file installed ,you can find "which" one is currently used with this command.

Can we use which command to search for a file on a given directory? No,we can't. "which" searches only pre-defined directories shown by echo $PATH.

so in order to search a file on any directory,

find ~ -name "linux.txt"

title: find

Searches for files in a directory hierarchy.

Tips and tricks:

To find regular files and invoke the file command on the results, run

find . -type f -exec file '{}' \;

To find regular files and display their attributes using the ls command, run

find . -type f -exec ls -l '{}' \;

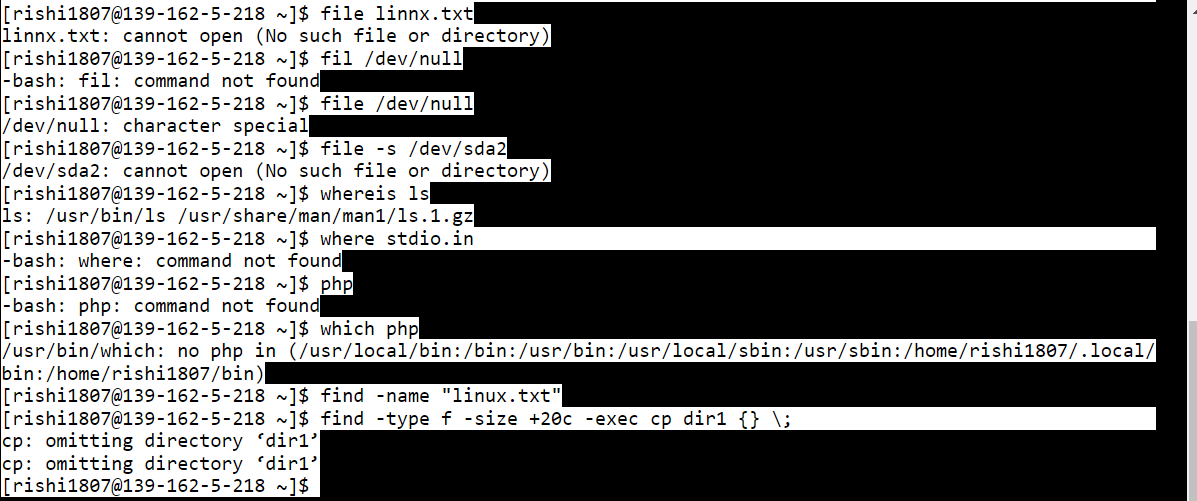
To find files over 20 bytes in size and list them out, run

find ~ -type f -size +20c -exec ls -hl {} \;

What this last command does is left as an exercise for you.

find ~ -type f -size +20c -exec cp dir1 {} \;

***output***



***Exercise -8***

uptime

title: uptime

uptime gives ,the current time, how long the system has been running, how many users are currently logged on, and the system load averages for the past 1, 5,and 15 minutes.

To know current date and time simply use

date

title: date

Okay that display the current time of server running webminal.org website.

To display details about currently logged users

who

title: who

can you see other linux users ? :)

who -a

print information about users who are currently logged into the system. You can also use a single letter command,

w

title: w

see it gives more detailed informatio than who. w can display information about the users currently on the machine, and their processes.The header shows, in this order,the current time,how long the system has been running, how many users are currently logged on, and the system load averages for the past 1, 5, and 15 minutes.

Displays list of mounted file system

mount

title: mount

provides list of mounted file systems.

Tips and tricks:

to view only ext4 file system,

mount -t ext4

to display free disk space on mounted devices.

df -h

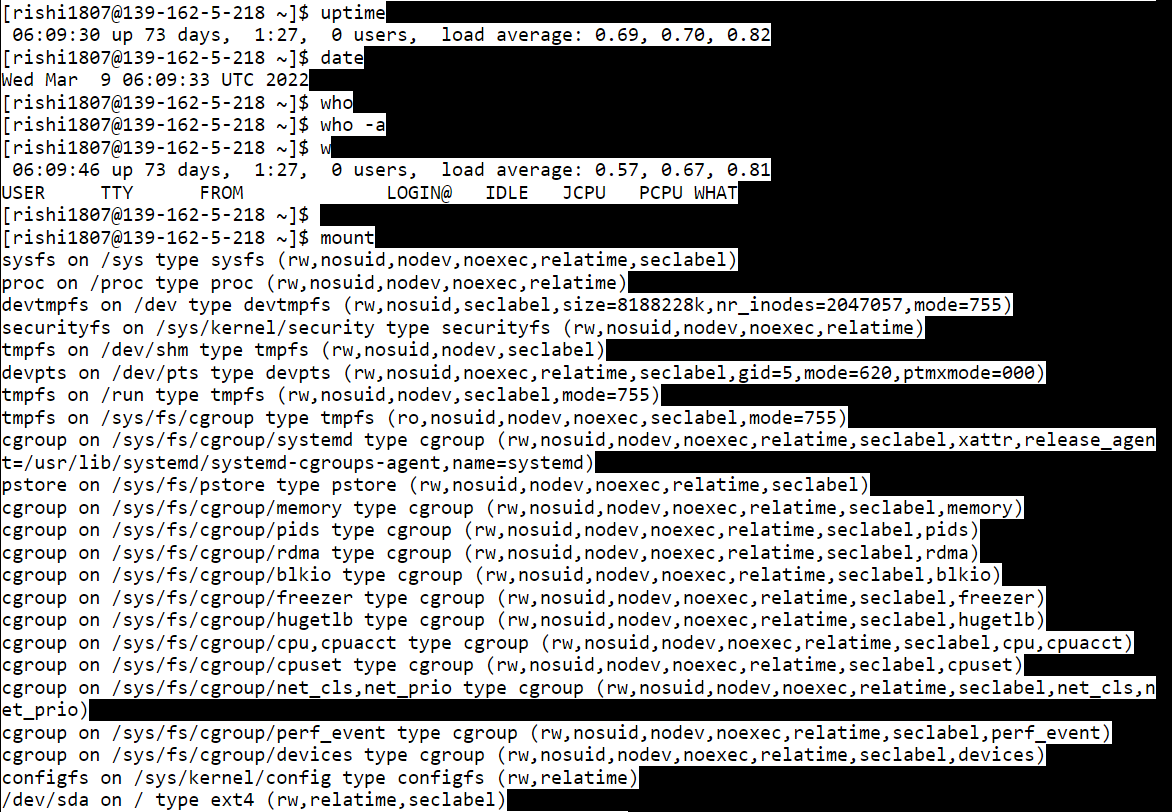
title: df

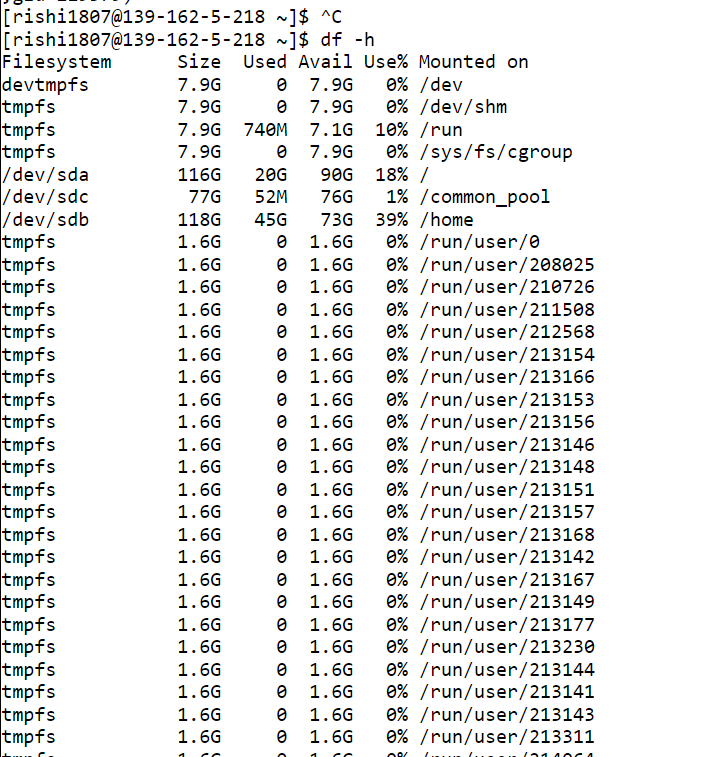
-h switch makes the output more headable for humans. so we found df finds disk usage,but to find memory usage,we need to use

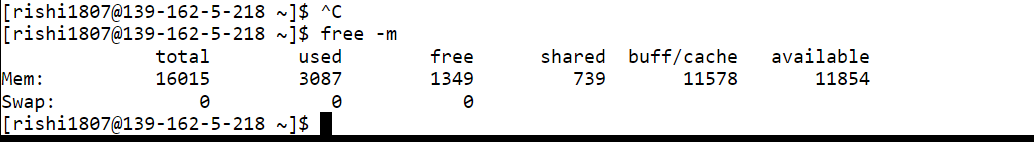
free -m

title: free

displays the total amount of free and used physical and swap memory in the system, as well as the buffers used by the kernel.







***Exercise -9***

hostname

Go ahead and type it. Did it display

\*fedori\*

again?

Great!. You will be thinking "Wait a second, we wanted to discuss about process not about hostname of system."

Yes, I agree with you. But again, how did you got the hostname? Angry response will be hostnameto be more precise /bin/hostname file.

Let me ask one more question, /bin/hostname is a file or process? Of course its a file, always remember In Linux everything is file.

Its a special file, go ahead use the 'file' command you learned in previous lessons.

file /bin/hostname

It tell you something like

\*/bin/hostname: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.18, stripped\*

Its a ELF 32-bit executable file. We ignore rest of the output. So this /bin/hostname is not a process? Yes, its just an executable file.

When we say its executable file. /bin/hostname contains instructions stored in a binary format. These instruction will be Kernel to do some task or action.

Lets go ahead view those instructions /bin/hostname. Go ahead (never be afraid of Linux commandline)

cat /bin/hostname

We got combination readable and unreadable characters. Somewhere within those output, we are telling Kernel to read a file name '/etc/hosts' and search for entry '127.0.0.1'.(which typically points to host machine name) and display that content.

When we type an executable '/bin/hostname' on bash prompt, Its content whatever you saw few lines above is loaded into RAM. Kernel follows those instruction and takes action.

Finally, here's our definition of a Process.

"Process is nothing but a file-content which is residing in RAM"

Now lets revisit the Wiki :

"In computing, a process is an instance of a computer program

that is being executed."

Here, an instance of a computer program means its residing in RAM and being executed means Kernel follows its instruction.

I hope it makes some sense now.

Our 'fedori' has lot of Process executed by Users or by Kernel itself. Each process has different states. Let's display some process currently, residing at RAM.

To view them we can use command called 'ps'. Its similar to 'ls' except that ls will show files on disk. 'ps' will show files on RAM

ps

Gave us below output. Lets spend sometime on these.

\*`

PID TTY TIME CMD

27447 pts/9 00:00:00 bash

29731 pts/9 00:00:00 ps

`\*

### **Understanding more about process:**

It has 4 field PID,TTY,TIME,CMD

Lets start with easy one,

CMD as name indicates, its tells what's the executable file name currently residing in-memory. Its has unique id,which is called 'pid'.

PID - Each process has unique number assigned to it. Since Kernel needs to know 'who' is responsiable for these instruction.

Why can't Kernel get this information from CMD itself? think about it. We will discuss answer little later :)

TTY - Its terminal allocated to the process. If Kernel needs to ask something or print something it will use this terminal for this process.

Finally, TIME - It refers to how time CPU took to execute this process's instruction. Its denoted in [dd-]hh:mm:ss format

We have two process with id 27447 and 29731.

### **Parent process:**

We saw 'hostname' was loaded into RAM and executed by Kernel. But who instructed Kernel to find and load '/bin/hostname' into RAM in the first place?

There must be some process sitting already in RAM, waiting for you to type characters and press 'Enter' key. right? So only job of this guy is to watch for any input (on TTY) and analyze and tell Kernel whether, Kernel needs to take any action on the input request.

In real world, this looks like a 'Manager' job. Who wants other to do the task.Its like, BOSS telling, "My work is wait for some work to arrive, when it arrives I'll assign to my employee/programmer to complete that task and report it to me" :P

In this case, 'Manager' is Bash shell. He assigns tasks to poor-hardworking Kernel. But wait, Kernel is kind of a 'Super Programmer'. HE decides whether to complete the task or refuse the task depending on this own evaluation. ;)

Okay, so this 'Bash shell' is called parent process which instructed Kernel to load hostname and execute it. Parent process has its own unique pid. Can we list the parent process of a process? Yes, we can,that the whole point of above two paragraphs :P Lets do that:

type: bash

Yes, its kind of 'secondary manager'. Now type 'ps' again

\*`

PID TTY TIME CMD

27447 pts/9 00:00:00 bash

31400 pts/9 00:00:00 bash

31414 pts/9 00:00:00 ps

`\*

Can you note the similarity? The process id '27447' remains the same. He's your primary bash shell. He instructed Kernel to create another Bash shell. It has id 31400 finally 31414 is the id of 'ps' the command used to print above output.

with ps we can pass an option '-o ppid' to get the parent id.Lets get the parent of 31400 process.

ps -o ppid 31400

\*`

PPID

27447

`\*

Yeah,we got the expected output. If you are not convinced, you can also print name like

ps -o ppid,cmd 31400

\*`

PPID CMD

27447 bash

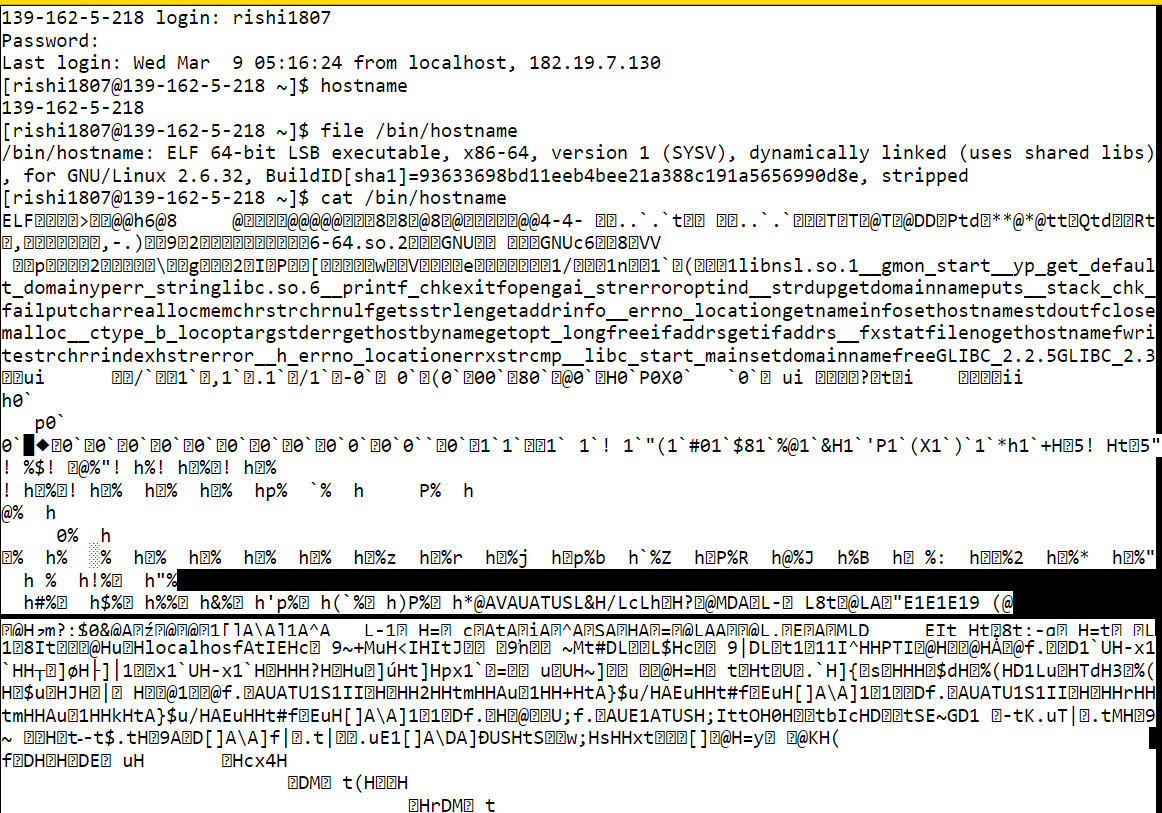
`\*

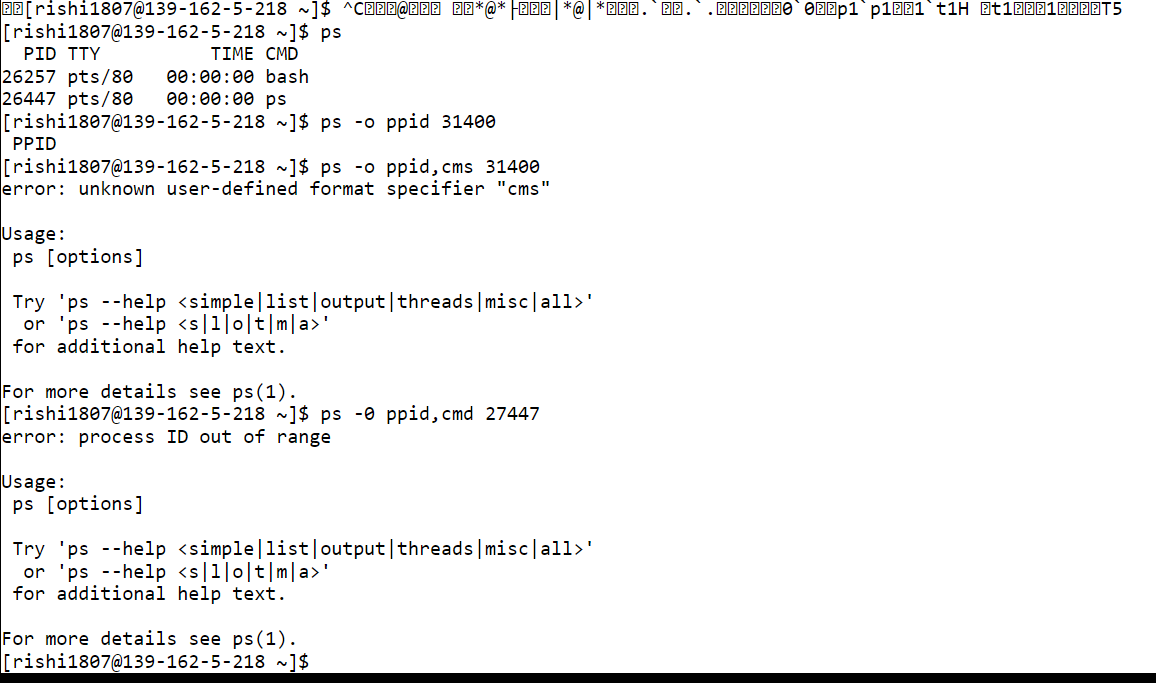
We call 27447 as parent process and 31400 as child process. Also note child pid is higher than parent pid. Most often, this will remain true. Unless Kernel runs out of pid and starts re-using it.

Can we find the Parent of our primary Bash prompt (ie. grandparent of 31400)?

ps -o ppid,cmd 27447

***output***

******

******

***Exercise -10***

So we know there is a first process named 'init' with pid. This is parent of all process in the system. And a process named 'bash' interacts with Kernel on behalf of user requests or commands.

Now when I log in and type ps - I get below output:

$ ps

PID TTY TIME CMD

5254 pts/1 00:00:00 bash

5336 pts/1 00:00:00 ps

See this is different from our previous output. Everytime you login,new parent bash is created.In case our Manager Bash job id is 5254. You know each command is a process, right? Lets create few process for this session.

Type

sleep 5

Did it hang for 5 seconds and then provide you the bash prompt again?

The way, our bash creates child process is blocking call. It means, run the child process and wait for it to complete and then return to me again.

### **Background process**

So when we ran our child process (sleep), 'Bash' shell waited for it to finish. User request below commands:

sleep 5

sleep 2

Shell will run sleep 5 first and wait for it to end. Then it runs sleep 2. But what if job-2 in this case 'sleep 2' is more critical than first job. There is unnecessary delay of 5 seconds right? Shell has an option of running child process on background - that means it wont wait for child to finish before accepting inputs from user.

We can put any child process in background by just appending & character to it!

Lets try that:

$ sleep 5 &

[1] 5781

$ sleep 2

Look, we can ran sleep 2 without hanging for 5 seconds! and the number you see 5781 is the background child process id. Its for our reference. How to verify its indeed the pid of child? Simple just start child process in background and execute 'ps' command.

$ sleep 5 &

[1] 6095

$ ps

PID TTY TIME CMD

5254 pts/1 00:00:00 bash

6095 pts/1 00:00:00 sleep

6099 pts/1 00:00:00 ps

Our sleep 5 pid matches with ps output. Do we have better way to visualize this? yes do. We have a command named 'pstree' which will tell you the mapping between child and parent process!!

Lets try again, note down your Bash shell pid (here its 5254)

$ pstree 5254

bash───pstree

It tells, we have Bash and one child process named 'pstree'.run our child,

$ sleep 5 &

[1] 6208

$ pstree 5254

bash─┬─pstree

└─sleep

Now it says, 'Bash' has two child named, pstree and out background process 'sleep' pstree has an option to mention -p which displays pid next to process name.

$sleep 5 &

[1] 6272

$ pstree -p 5254

bash(5254)─┬─pstree(6276)

└─sleep(6272)

the pid we got while starting background process is same as what we got from pstree output.

Lets say we have started 4 long running background jobs each runs for 145 seconds and 1 very long process for 3000 seconds.

$sleep 45 &

[1] 6393

$sleep 45 &

[2] 6397

$sleep 45 &

[3] 6401

$sleep 45 &

[4] 6406

$ sleep 3000 &

[5] 6557

$pstree -p 5254

bash(5254)─┬─pstree(6410)

├─sleep(6393)

├─sleep(6397)

├─sleep(6401)

└─sleep(6406)

├─sleep(6557)

### **List background jobs**

pstree gives information about all jobs. We do not need pstree(6410), because are intereste in only background jobs. How to view only those jobs. For this purpose, we have 'jobs' command will give output like:

jobs

[1] Running sleep 145 &

[2] Running sleep 145 &

[3]- Running sleep 145 &

[4]+ Running sleep 145 &

[5]+ Running sleep 3000 &

It lists only our background processes and its status. So far good right?

### **Foreground process**

Our process-[5] runs for 3000 seconds, it takes long time to complete. Background will take less CPU time compared to non-background process .ie foreground process. So lets bring to foreground process. type, fg in our case, we need to bring background job-5.

fg 5

sleep 3000

Can you shell hanging now? wait..hanging is a wrong word to use. shell executing sleep command now? :) Are you going to wait for 3000 seconds? aka 50 minutes?

### **Switch between foreground to background**

Lets admit it, by mistake without thinking, you (yes, its you :p )brought this background process to foreground, now you desperately want to put in on background again!!

Dont worry, Linux is so flexible we can do that too. Just press 'ctrl+z'.You get output which says job is stopped.

^Z

[5]+ Stopped sleep 3000

verify the status by checking output of jobs command.

$jobs

[5]+ Stopped sleep 3000

Its stopped, we can restart the process again in background with 'bg '

$bg 5

[5]+ sleep 3000 &

$jobs

[5]+ Running sleep 3000 &

***Output***

******

***Exercise -11***

### **Linux Process states**

Lets discuss about process states. man ps shows process can be any one of the following states

D Uninterruptible sleep (usually IO)

R Running or runnable (on run queue)

S Interruptible sleep (waiting for an event to complete)

T Stopped, either by a job control signal.

X dead (should never be seen)

Z Defunct ("zombie") process, terminated but not reaped by its parent.

We try to reproduce some of the above states to understand it better. We will begin by listing existing process and its states via command :

ps -S

PID TTY STAT TIME COMMAND

16454 pts/4 Ss 0:03 bash

28682 pts/4 R+ 0:00 ps -S

As you can see STAT S stands for Interruptible sleep. Here, Our bash/shell is waiting for its child 28682 to complete. 28682 is in running state. s - stands for session leader and + tells it runs in foreground.

Lets move a process to Stopped state.

ps S

PID TTY STAT TIME COMMAND

16454 pts/4 Ss 0:05 bash

29789 pts/4 R+ 0:00 ps S

Now type sleep 100 then and press Ctrl+z

sleep 100

^Z

Now running ps shows the below output. Can you find out the meaning for T ?

ps S

PID TTY STAT TIME COMMAND

16454 pts/4 Ss 0:05 bash

29796 pts/4 T 0:00 sleep 10

29846 pts/4 R+ 0:00 ps S

Note this process is stopped its not terminated. That means, we can resume this stopped process. To prints upto 50000 on the screen we can perform

seq 1 500000

After running it - I stopped it via ctrl+z

19957

19958

19959

19960

^Z

As you can its stopped at 19960. Go ahead verify the process status via ps S command. Now its resume this stopped process via fg command. You can see its continues to print from 19961 to 50000. What? too fast and can't see its started from 19961? thats pretty bad, you need to go for eye-checkup :-)

### **Zombie process**

Zombie is a terminated process but not reaped by its parent. When we say not reaped by its parent, we means "parent is yet to collect the exit status from child". Child is completed its execution ready with the exit status and waiting for parent to ask for it. This is tricky case to reproduce lets try:We have our session leader 2249

ps

PID TTY TIME CMD

2249 pts/1 00:00:00 bash

2294 pts/1 00:00:00 ps

Lets create subshell:

bash

Our subshell has pid 2498

ps

PID TTY TIME CMD

2249 pts/1 00:00:00 bash

2498 pts/1 00:00:00 bash

2540 pts/1 00:00:00 ps

What we going to do is create two more subshells and from there stop this subshell. with command like :

( ( kill -STOP 2498 ) )

[1]+ Stopped bash

Our subshell (2498) is stopped from its grandchild shell. So 2498 yet to collect the exit state of its child:

ps S

PID TTY STAT TIME COMMAND

2249 pts/1 Ss 0:00 bash

2498 pts/1 T 0:00 bash

2547 pts/1 Z 0:00 [bash] <defunct>

2551 pts/1 R+ 0:00 ps S

As you can see T shows our process subshell is stopped without collecting exit status from its child. Apparently its child 2547 is in Zombie state Z

Note we only stopped the subshell, we can see them with command:

$ jobs

[1]+ Stopped bash

If you start this bash shell, it will collect the exit status from 2547 and reap its child. Thus the zombie process will disappear. Lets start our stopped shell:

fg

bash

Now lets check the status:

ps S

PID TTY STAT TIME COMMAND

2249 pts/1 Ss 0:00 bash

2498 pts/1 S 0:00 bash

2561 pts/1 R+ 0:00 ps S

See Zombie process is disappeared!

### **Orphaned process**

Orphaned is running process which with no parents. Parent is died without reaping the child (getting the exit status from child). These child process without parents are named orphaned process. These process are adopted by init which has pid 1. Lets try to create orphaned process similar to above steps:

ps S

PID TTY STAT TIME COMMAND

2249 pts/1 Ss 0:01 bash

3325 pts/1 R+ 0:00 ps S

Create our subshell

bash

ps

PID TTY TIME CMD

2249 pts/1 00:00:00 bash

3329 pts/1 00:00:00 bash

3371 pts/1 00:00:00 ps

From grandchild skill our sub-shell 3329 where as his child is still running.

( sleep 100 & ( kill -9 3329 ))

Killed

Now verify that subshell with 3329 is indeed dead.

ps S

PID TTY STAT TIME COMMAND

2249 pts/1 Ss 0:01 bash

3376 pts/1 S 0:00 sleep 100

3381 pts/1 R+ 0:00 ps S

Go ahead verify the parent of our orphaned child.

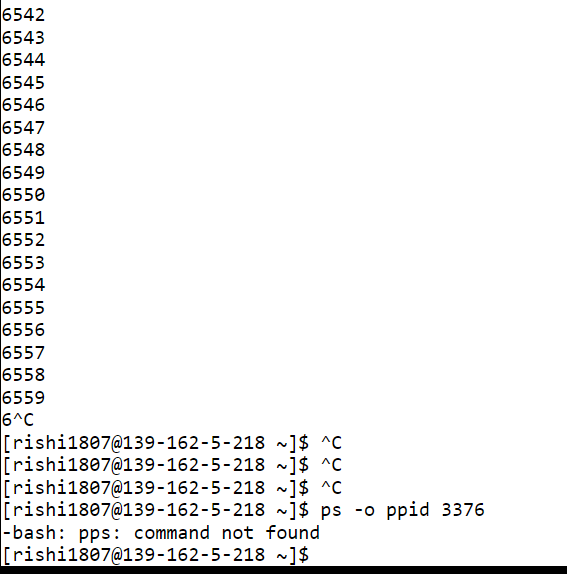
ps -o ppid 3376

PPID

1

Its our init process. It adopted it!

***Output***

******

**Exp No. 3**

***Process Management Commands in Linux.***

Starting a Process

When you start a process (run a command), there are two ways you can run it −

* + Foreground Processes
  + Background Processes

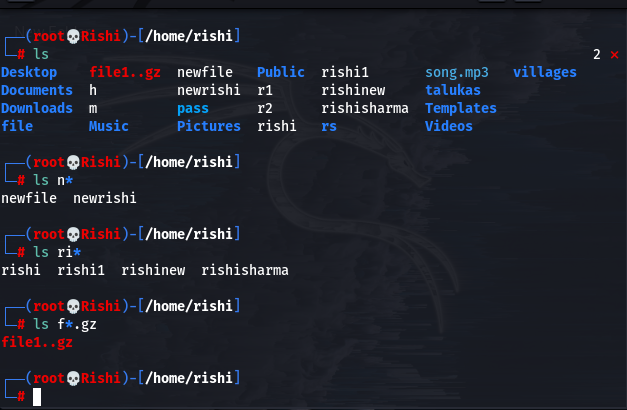
**Foreground Processes**

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

You can see this happen with the ls command. If you wish to list all the files in your current directory, you can use the following command −

This would display all the files, the names of which start with rishi and end with .gz –

**1. ls and file name and \* command**

******

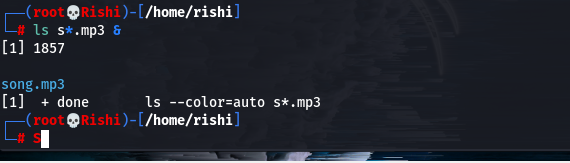
**Background Processes**

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

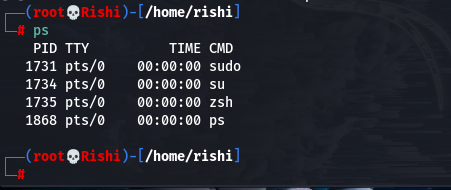
The simplest way to start a background process is to add an ampersand (&) at the end of the command.

**2. ls and file name and \* .file type & command**



***Listing Running Processes 🡪***

It is easy to see your own processes by running the ps (process status) command as follows –

******

One of the most commonly used flags for ps is the -f ( f for full) option, which provides more information as shown in the following example –

Here is the description of all the fields displayed by **ps -f** command −

|  |  |
| --- | --- |
| **S.No.** | **Column & Description** |
| 1 | **UID**  User ID that this process belongs to (the person running it) |
| 2 | **PID**  Process ID |
| 3 | **PPID**  Parent process ID (the ID of the process that started it) |
| 4 | **C**  CPU utilization of process |
| 5 | **STIME**  Process start time |
| 6 | **TTY**  Terminal type associated with the process |
| 7 | **TIME**  CPU time taken by the process |
| 8 | **CMD**  The command that started this process |

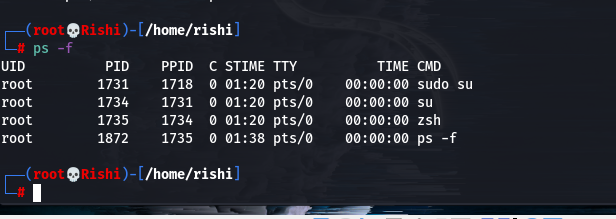
There are other options that can be used along with **ps** command −

|  |  |
| --- | --- |
| **S.No.** | **Option & Description** |
| 1 | **-a**  Shows information about all users |
| 2 | **-x**  Shows information about processes without terminals |
| 3 | **-u**  Shows additional information like -f option |
| 4 | **-e**  Displays extended information |

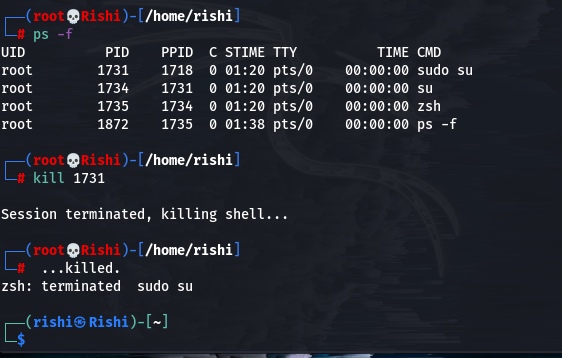
**Stopping Processes**

Ending a process can be done in several different ways. Often, from a console-based command, sending a CTRL + C keystroke (the default interrupt character) will exit the command. This works when the process is running in the foreground mode.

If a process is running in the background, you should get its Job ID using the **ps** command. After that, you can use the **kill** command to kill the process as follows −

******

Here, the kill command terminates the first\_one process. If a process ignores a regular kill command, you can use kill -9 followed by the process ID as follows

******

**Exp No. 4**

***To study the various commands operated in vi editors in Linux***

**1. Shell programming: Introduction --:**

A **shell program** sometimes referred to as a shell script[**1**](https://homepages.uc.edu/~thomam/Intro_Unix_Text/Shell_Prog.html#shell_foot1), is simply a program constructed of shell commands. Shell programs are interpreted each time they are run. This means each command is processed (i.e. executed) by the shell a single line at a time. This is different from languages such as C or C++, which are translated in their entirety by a compiler program into a binary image. A shell program may be simple and consist of just a few shell commands, or it may be very complex and consist of thousands[**2**](https://homepages.uc.edu/~thomam/Intro_Unix_Text/Shell_Prog.html#shell_foot2) of shell commands. The complexity of the shell program is in the hand of the programmer. In general, a shell program can be characterized by:

* shell programs consist of one or more primitive shell commands
* shell programs are created using your text editor of choice, e.g. vi or emacs
* shell programs are executed just as shell commands are, by typing the name of the program followed by the *[Enter]* key
* shell programs have permission modes as do any other file, and must have the correct permissions set to execute the program
* as with other programming languages, the shell language has the functionality to allow input & output, iteration, logical decision making, file creation and deletion, and system call capability
* shell programs are free format, as long as the syntax of each shell command is correct. This means that blank lines, indentation and abundant whitespace can be used freely.

**2. Shell programming: Control Structure --:**

You can control the execution of Linux commands in a shell script with control structures. Control structures allow you to repeat commands and to select certain commands over others. A control structure consists of two major components: a test and commands. If the test is successful, then the commands are executed. In this way, you can use control structures to make decisions as to whether commands should be executed.

There are two different kinds of control structures: *loops* and *conditions.* A loop repeats commands, whereas a condition executes a command when certain conditions are met. The BASH shell has three loop control structures: **while**, **for**, and **for-in**. There are two condition structures: **if** and **case**. The control structures have as their test the execution of a Linux command. All Linux commands return an exit status after they have finished executing. If a command is successful, its exit status will be 0. If the command fails for any reason, its exit status will be a positive value referencing the type of failure that occurred. The control structures check to see if the exit status of a Linux command is 0 or some other value. In the case of the **if** and **while** structures, if the exit status is a zero value, then the command was successful and the structure continues.

### Test Operations

With the **test** command, you can compare integers, compare strings, and even perform logical operations. The command consists of the keyword **test** followed by the values being compared, separated by an option that specifies what kind of comparison is taking place. The option can be thought of as the operator, but it is written, like other options, with a minus sign and letter codes. For example, **-eq** is the option that represents the equality comparison. However, there are two string operations that actually use an operator instead of an option. When you compare two strings for equality you use the equal sign, **=**. For inequality you use **!=**. [Table 8-6](http://litux.nl/Reference/books/7213/ddu0071.html#wbp12chapter8t6p6) lists some of the commonly used options and operators used by **test**. The syntax for the **test** command is shown here:

test *value* -*option value*

test *string* = *string*

| **BASH Shell Test Operators** | |
| --- | --- |
| **Integer Comparisons** | **Function** |
| **-gt** | Greater-than |
| **-lt** | Less-than |
| **-ge** | Greater-than-or-equal-to |
| **-le** | Less-than-or-equal-to |
| **-eq** | Equal |
| **-ne** | Not-equal |
| **String Comparisons** |  |
| **-z** | Tests for empty string |
| **=** | Equal strings |
| **!=** | Not-equal strings |
| **Logical Operations** |  |
| **-a** | Logical AND |
| **-o** | Logical OR |
| **!** | Logical NOT |
| **File Tests** |  |
| **-f** | File exists and is a regular file |
| **-s** | File is not empty |
| **-r** | File is readable |
| **-w** | File can be written to, modified |
| **-x** | File is executable |
| **-d** | Filename is a directory name |

In the next example, the user compares two integer values to see if they are equal. In this case, you need to use the equality option, **-eq**. The exit status of the **test** command is examined to find out the result of the test operation. The shell special variable **$?** holds the exit status of the most recently executed Linux command.

$ **num=5**

$ **test $num -eq 10**

$ **echo $?**

1

Instead of using the keyword **test** for the **test** command, you can use enclosing brackets. The command **test** **$greeting** = "hi" can be written as

$ **[ $greeting = "hi" ]**

Similarly, the test command **test** **$num** -eq 10 can be written as

$ **[ $num -eq 10** **]**

The brackets themselves must be surrounded by white space: a space, TAB, or ENTER. Without the spaces, it would be invalid.

**Conditional Control Structures**

The BASH shell has a set of conditional control structures that allow you to choose what Linux commands to execute. Many of these are similar to conditional control structures found in programming languages, but there are some differences. The **if** condition tests the success of a Linux command, not an expression. Furthermore, the end of an **if-then** command must be indicated with the keyword **fi**, and the end of a **case** command is indicated with the keyword **esac**. The condition control structures are listed in [Table 8-7](http://litux.nl/Reference/books/7213/ddu0071.html#wbp12chapter8t7p7).

The **if** structure places a condition on commands. That condition is the exit status of a specific Linux command. If a command is successful, returning an exit status of 0, then the commands within the **if** structure are executed. If the exit status is anything other than 0, then the command has failed and the commands within the **if** structure are not executed. The **if** command begins with the keyword **if** and is followed by a Linux command whose exit condition will be evaluated. The keyword **fi** ends the command. The **elsels**script in the next example executes the **ls** command to list files with two different possible options, either by size or with all file information. If the user enters an **s**, files are listed by size; otherwise, all file information is listed.

| **Table 8-7: BASH Shell Control Structures** | |
| --- | --- |
| **Condition Control Structures: if, else, elif, case** | **Function** |
| **if** *command* **then**     *command* **fi** | **if** executes an action if its test command is true. |
| **if** *command* **then**     *command* **else**     *command* **fi** | **if-else** executes an action if the exit status of its test command is true; if false, then the **else** action is executed. |
| **if** *command* **then**     *command* **elif** *command* **then**     *command* **else**     *command* **fi** | **elif** allows you to nest **if** structures, enabling selection among several alternatives; at the first true **if** structure, its commands are executed and control leaves the entire **elif** structure. |
| **case** *string* **in** *pattern****)***     *command****;;*** **esac** | **case** matches the string value to any of several patterns; if a pattern is matched, its associated commands are executed. |
| *command* **&&** *command* | The logical AND condition returns a true 0 value if both commands return a true 0 value; if one returns a non-zero value, then the AND condition is false and also returns a non-zero value. |
| *command* **||** *command* | The logical OR condition returns a true 0 value if one or the other command returns a true 0 value; if both commands return a non-zero value, then the OR condition is false and also returns a non-zero value. |
| **!** *command* | The logical NOT condition inverts the return value of the command. |
| **Loop Control Structures:** **while, until, for, for-in, select** |  |
| **while** *command* **do**     *command* **done** | **while** executes an action as long as its test command is true. |
| **until** *command* **do**     *command* **done** | **until** executes an action as long as its test command is false. |
| **for** *variable* **in** *list-values* **do**     *command* **done** | **for-in** is designed for use with lists of values; the variable operand is consecutively assigned the values in the list. |
| **for** *variable* **do**     *command* **done** | **for** is designed for reference script arguments; the variable operand is consecutively assigned each argument value. |
| **select** *string* **in** *item-list* **do**     *command* **done** | **select** creates a menu based on the items in the *item-list;* then it executes the command; the command is usually a **case**. |

**elsels**

**echo Enter s to list file sizes,**

**echo otherwise all file information is listed.**

**echo -n "Please enter option: "**

**read choice**

**if [ "$choice" = s ]**

**then**

**ls -s**

**else**

**ls -l**

**fi**

**echo Good-bye**

|  |
| --- |
|  |

A run of the program follows:

$ **elsels**

Enter s to list file sizes,

otherwise all file information is listed.

Please enter option: s

total 2

1 monday 2 today

$

### Loop Control Structures

The **while** loop repeats commands. A **while** loop begins with the keyword **while** and is followed by a Linux command. The keyword **do** follows on the next line. The end of the loop is specified by the keyword **done**. The Linux command used in **while** structures is often a test command indicated by enclosing brackets.

The **for-in** structure is designed to reference a list of values sequentially. It takes two operands—a variable and a list of values. The values in the list are assigned one by one to the variable in the **for-in** structure. Like the **while** command, the **for-in** structure is a loop. Each time through the loop, the next value in the list is assigned to the variable. When the end of the list is reached, the loop stops. Like the **while** loop, the body of a **for-in** loop begins with the keyword **do** and ends with the keyword **done**. The **cbackup**script makes a backup of each file and places it in a directory called **sourcebak**. Notice the use of the **\*** special character to generate a list of all filenames with a **.c** extension.

**cbackup**

**for backfile in \*.c**

**do**

**cp $backfile sourcebak/$backfile**

**echo $backfile**

**done**

|  |
| --- |
|  |

A run of the program follows:

$ **cbackup**

io.c

lib.c

main.c

$

The **for** structure without a specified list of values takes as its list of values the command line arguments. The arguments specified on the command line when the shell file is invoked become a list of values referenced by the **for** command. The variable used in the **for** command is set automatically to each argument value in sequence. The first time through the loop, the variable is set to the value of the first argument. The second time, it is set to the value of the second argument.

**3. Shell programming: Working with files --:**

Most of the time, we use shell scripting to interact with the files. Shell scripting offers some operators as well as some commands to check and perform different properties and functionalities associated with the file. For our convenience, we create a file named ‘geeks.txt’ and another .sh file (or simply run on the command line) to execute different functions or operations on that file. Operations may be reading the contents of the file or testing the file type. These are being discussed below with proper examples:

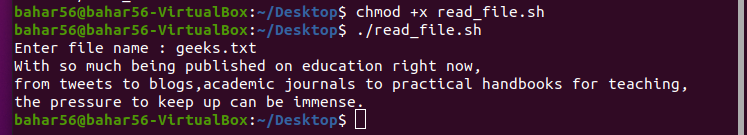
File Reading Functionalities

File reading is an interesting task in a programmer’s life. Shell scripting offers some functionalities for reading the file, reversing the contents, counting words, lines, etc.

1. Reading line by line: First, we take input using the read command then run the while loop which runs line after line.

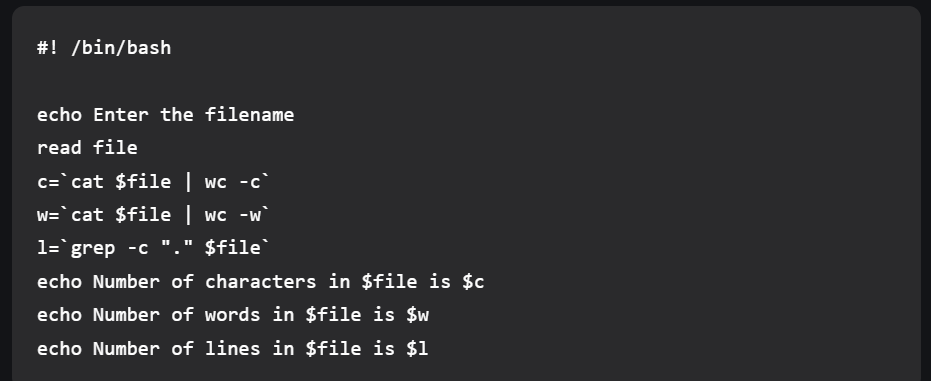
Script:

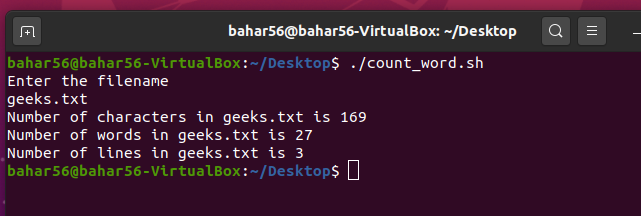




2. Counting characters, words & lines in the file: We take three variables, one for counting characters, words, and lines respectively. We use ‘wc’ command, which stands for word count and counts the number of characters and words as well. For counting lines, we pass ‘grep ‘ which keeps count of the lines that match a pattern. Then we print out each variable.

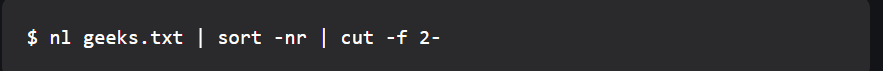
Script:

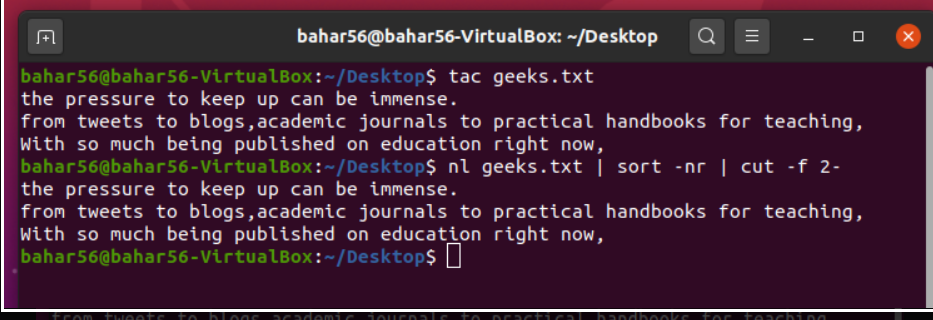
echo Number of lines in $file is $l



3. Display file contents in reverse: To print the contents of any file in reverse, we use tac or nl, sort, cut commands. Tac is simply the reverse of a cat and simply prints the file in reverse order. Whereas nl commands numbers, the file contents sort the numbered file in the reverse order, and the cut command removes the number and prints the file contents.

Script:

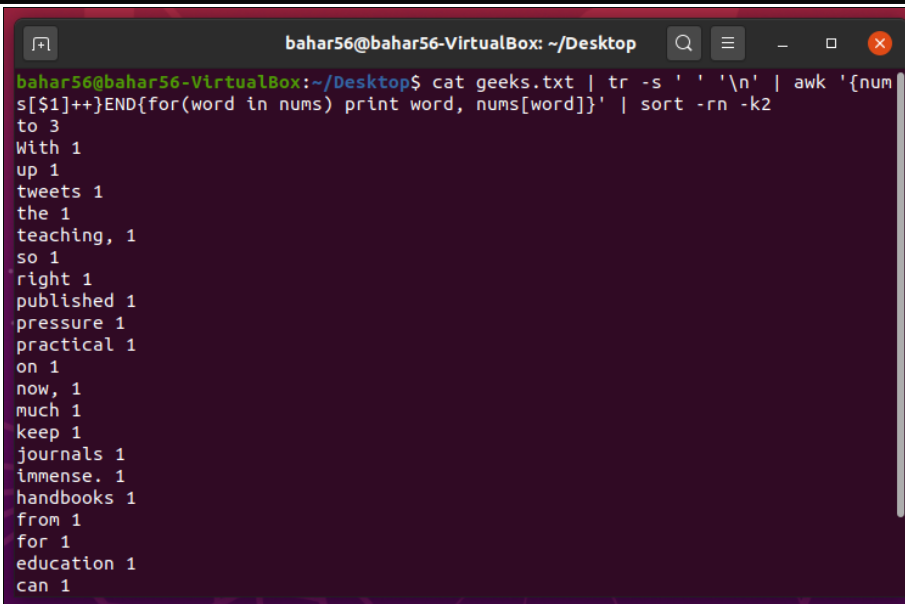




4. Frequency of a particular word in the file: To count the frequency of each word in a file, we use certain commands. These are xargs which applies print to every line in the output, the sort which sorts, current buffer piped to it, uniq -c displays the counts of each line in the buffer and, lastly awk, prints the 2nd column and then the 1st column based on the problem requirement.

Script:

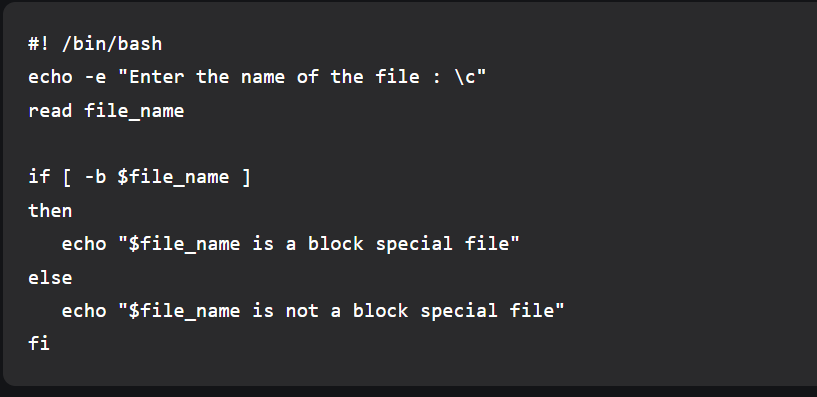




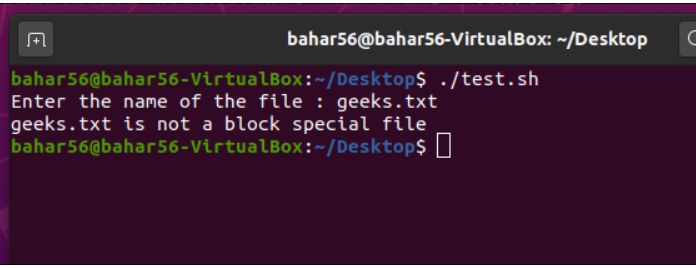
**File Test Operators**

1. -b file: This operator checks if the file is a block special file or not and returns true or false subsequently. It is [-b $file] syntactically.

Script:

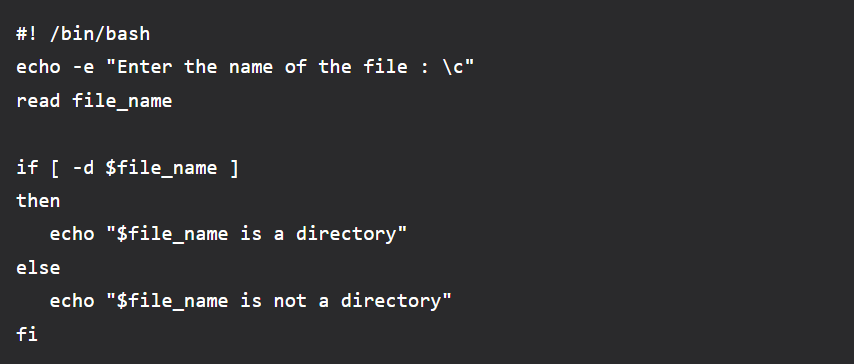


Output:

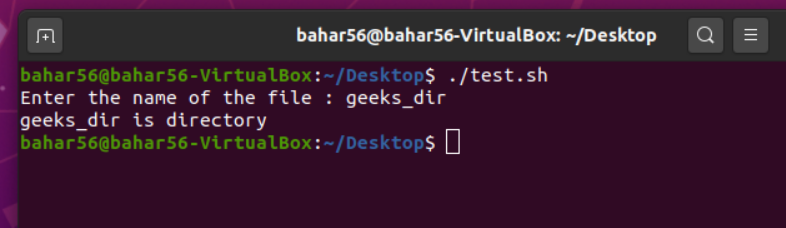


2. -d file: The operator looks over if the file is present as a directory. If Yes, it returns true else false. It is [-d $file] syntactically.

Script:

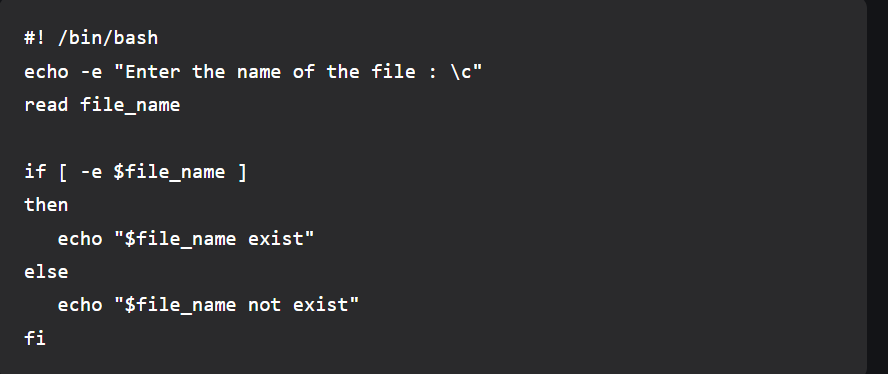


Output:

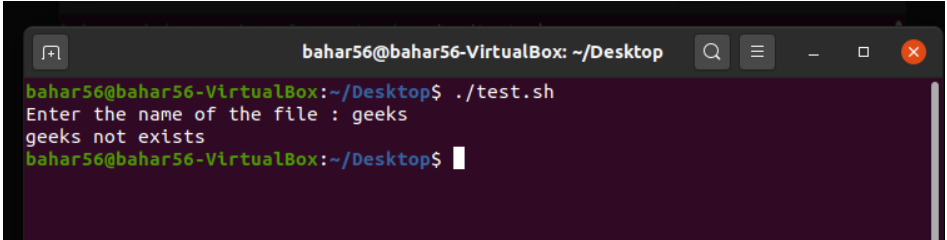


3. -e file: The operator inspects if the file exists or not. Even if a directory is passed, it returns true if the directory exists. It is [-e $file] syntactically.

Script:

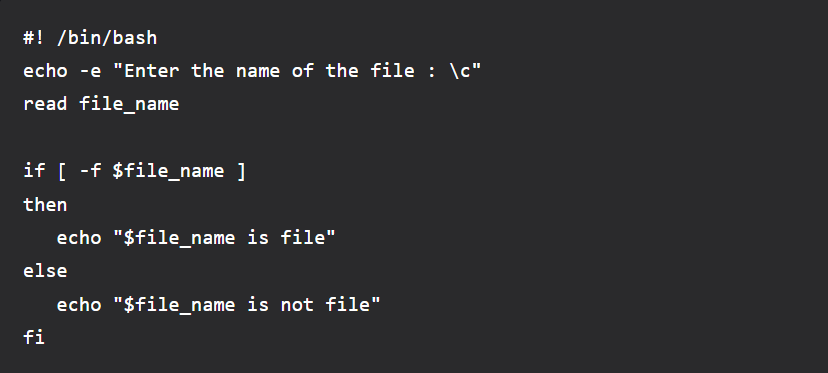


Output:

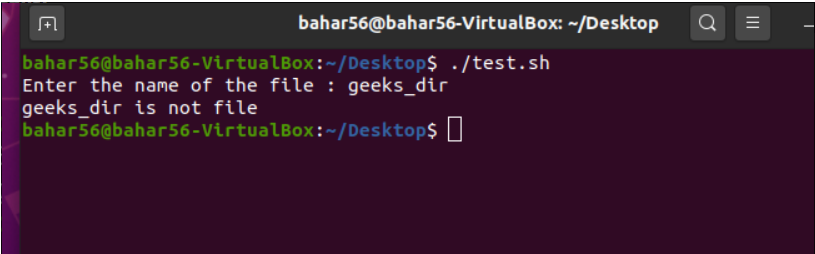


4. -f file: If the file is an ordinary file or special file, then it returns true else false. It is [-f $file] syntactically.

Script:

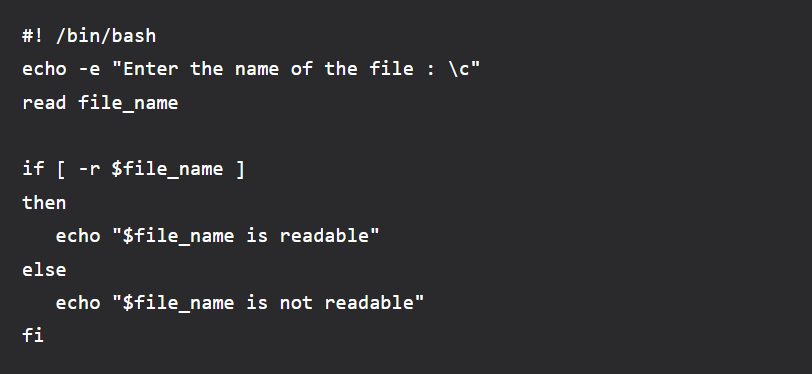


Output:

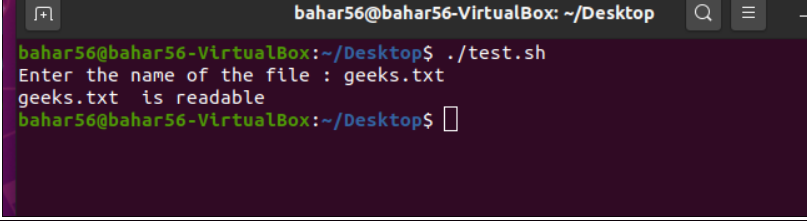


5. -r file: This checks if the file is readable. If found yes, then return true else false. It is [-r $file] syntactically.

Script:

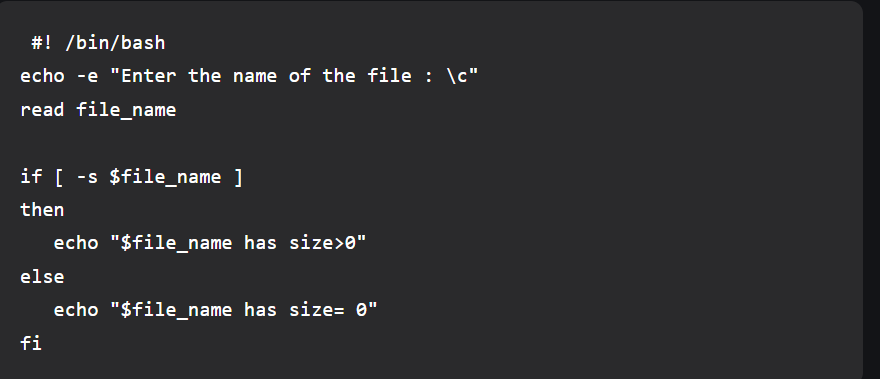


Output:

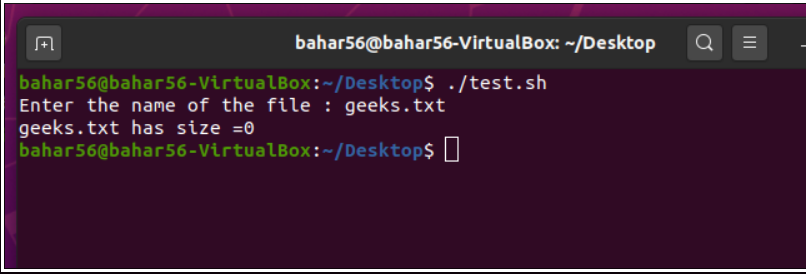


6. -s file: This operator checks if the file has a size greater than zero or not, which returns true or false subsequently. It is [-s $file] syntactically.

Script:

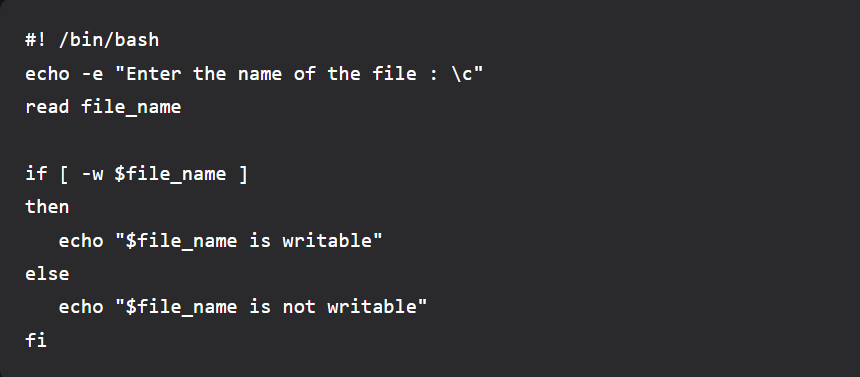


Output:

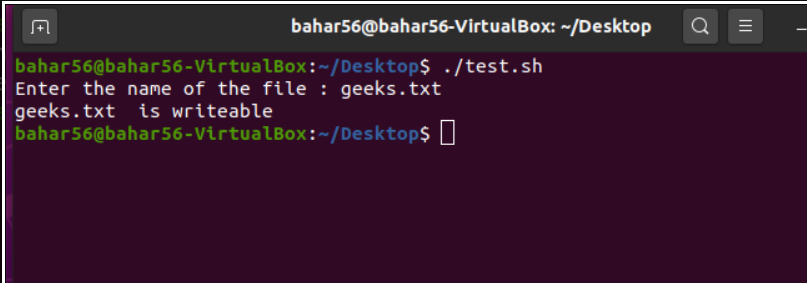


7. -w file: If wringing is allowed over the file, then the operator returns true, and if not then false. It is [-w $file] syntactically.

Script:

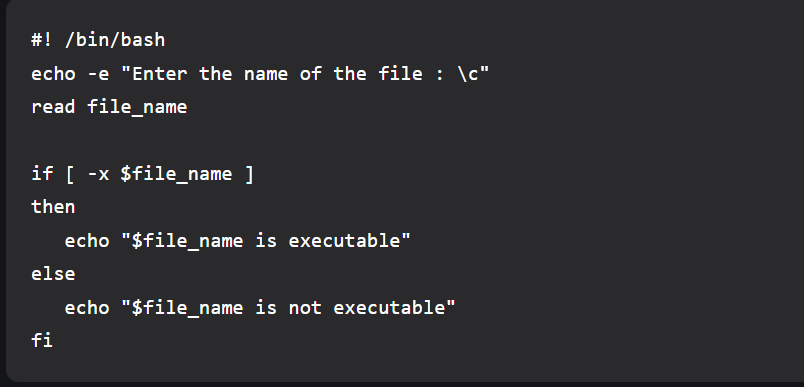


Output:

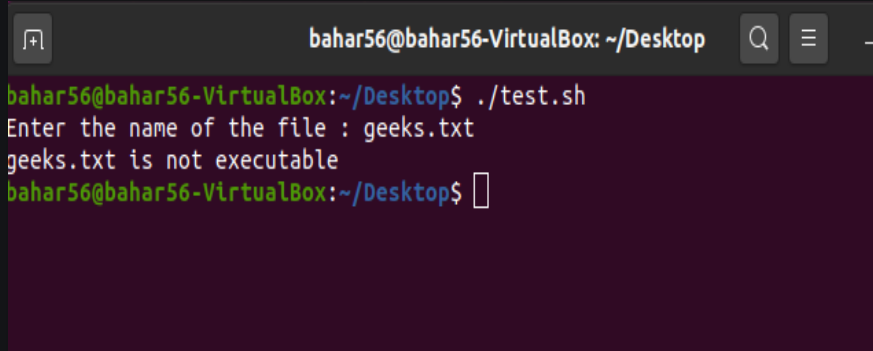


8. -x file: The operator looks over if the file is executable or not, and returns true and false subsequently. It is [-x $file] syntactically.

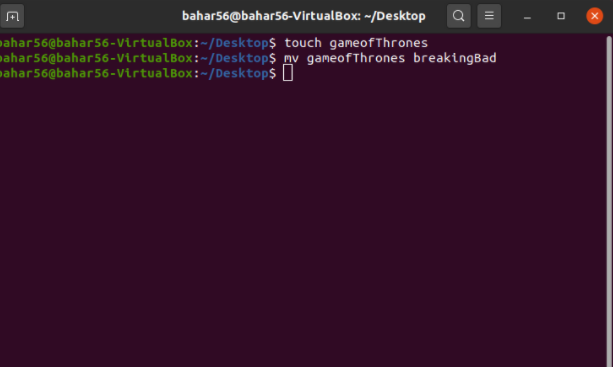
Script:



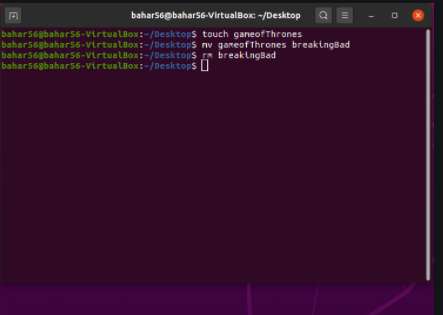
Output:



9. Rename & delete file: To rename the file, we use the ‘mv’ command which changes the name of the file and ‘rm’ to delete the file.



10. As we can see below the command line, breakingBad file (after renaming ) and deleting it with ‘rm’ command, it is no longer there**.**

****

**4. Shell programming: Examples/Exercise --;**

In the previous chapter, we have discussed commands used to work with Linux operating system. To execute a command, we typed it on command prompt and pressed Enter key. The output of the command gets displayed on the screen. Now what if we want to execute multiple commands in a sequence? Shall we type them one by one? But we may require executing same set of commands in sequence multiple times. Then typing same sequence multiple times is a tedious task. Don‟t worry; we have a way out of it. We can type sequence of commands in a text file. We can then give command to shell to execute this text file. Shell will execute all the commands available in the text file in a sequence. This text file is known as shell script. Shell script can be defined as series of commands written in a plain text file. In this chapter, we also discuss shell programming features like declaring and using variables, decision making and looping constructs. Shell scripts are commonly used by system administrators to automate daily administration tasks.

# Writing a simple shell script

Let us start with writing a simple shell script which prints hello world on the screen. We require a text editor to write a shell script. Various text editors like vi, vim, emacs and pico are available to serve the purpose of writing shell scripts. Gedit is a graphical editor available with GNOME desktop environment. Kwrite is a graphical editor available with KDE desktop environment. We will use vi editor to write the shell script as this editor is available with almost all Unix and Linux flavors. vi stands for „Visual display editor‟. The vi editor lets a user create new files or edit existing files. In this chapter we will discuss some of the vi‟s facilities, enough to allow us to create and edit shell scripts discussed in this chapter. So let us discuss how to use vi editor to write shell scripts.

# Using vi editor

We will start with opening a vi editor and let us see how it looks like.

* Right-click on the desktop and choose Open Terminal option from the quick menu displayed. You will get command prompt signifying that Linux Shell is ready to interact with you.

Now you have two options to open editor. Type vi and press Enter key or type vi

*filename* and press Enter key.

* Type vi myfile and press Enter key as shown in figure 12.1.

# in vi editor

Unlike other text editors, vi editor works in three modes, viz., insert mode, command mode and colon mode or ex command mode. We are aware that basic text processing features required to prepare any text file is to enter the text, commands to manipulate the text and commands to save the file, search the text within a file etc. The insert mode of vi puts the text typed using keyboard into the current file. The command mode allows the entry of commands to manipulate text. These commands are usually one or two characters long, and can be entered with few keystrokes. The ex mode permits us to give commands at the command line. The vi editor uses the command line to display messages and commands.

# Creating shell scripts: Getting started

Now let us learn how to create a simple shell script that prints “Hello World” on the screen. To type the text in vi editor it has to be in insert mode. By default vi starts in the command mode. So the first step is to change the vi into insert mode. There are several commands that put the vi editor into insert mode. The most commonly used commands to get into insert mode are a and i.

* Press Escape key and press i and vi editor will be now in Insert mode. You might have noticed that the current mode (Insert) is reflected on the screen as shown in figure 12

**Vi editor in insert mode**

* Now type first line as echo “Hello world… ” and Press Enter key.
* Type echo second line as echo “This is my first shell script”.

**Saving the file**

To save a file using vi editor, we need to switch to colon mode from insert mode. Press Esc key and you will notice that the word INSERT displayed at the bottom gets disappear signifying that vi editor is now in command mode. Type **:** (colon) and you will notice that colon is displayed in the bottom of the screen. Now vi editor is in colon mode. Type **wq** as shown in figure 12.4 and press Enter key. The vi editor will be closed and shell prompt will be displayed.

# Saving a file and Quiting vi editor

There are several other commands available to save a file depending on the current status and usage. Table 12.2 shows commands along with their use.

|  |  |
| --- | --- |
| **Command** | **Usage** |
| :w | To save file and remain in editing mode |
| :wq | To save file and quit editing mode |
| :x | To save file and quit editing mode (same as above) |
| :q | To quit editing mode when no changes are made |
| :q! | To quite editing mode without saving changes made in  the file |

Note that if you have opened vi editor without typing file name initially, then type file name along with the command. For example, type :wq myfile and then press enter key. Thus text typed in a buffer will be saved using myfile as name of the file.

**Executing Shell script**

Now you can execute the scripts by typing following commands:

1. bash your-script-name
2. sh your-script-name
3. ./your-script-name

* To execute our first shell script named myfile, type sh myfile and press Enter key. The output will be displayed as shown in figure 12.5.

# Figure 12.5: Executing a shell script

You may get error like “Permission denied” displayed on the screen. Do not worry; to execute shell script, Linux systems may expect execute permission on the file. Linux being a security favoring operating system does not provide execute permission by itself when a file is created. Type the following command to assign execute permission to the file named “myfile” created by us.

$chmod u+x myfile

Now to execute our first shell script, type sh myfile and press Enter key. You will find that two lines passed as arguments to echo statements are displayed in the screen.

# Editing and Navigating text

Now, let us make a small change in the myfile. Instead of “This is my first shell script” the out put of script should display your name instead of word my. Thus puja‟s script should display “This is puja‟s first shell script”. To edit the shell script,

* Type vi myfile to open the file. This time you may notice that filename, total number of lines and columns are displayed on the command line.

To type *puja* instead of *my,* we require cursor movements. The vi editor presents the keys h, j, k and l as a command to move the cursor in the four directions. These keys are placed adjacent to one another in the middle row of keyboard. Table 12.3 shows the keys and its usage.

|  |  |
| --- | --- |
| **Command** | **Usage** |
| k | To move cursor in up direction |
| j | To move cursor in down direction |

|  |  |
| --- | --- |
| h | To move cursor in left direction |
| l | To move cursor in right direction |

* + Using the commands explained in the table 12.3, position the cursor on letter *m* in second line. These commands can be used in command mode. Thus if you are working in Insert mode, press Esc key to change vi to command mode.
  + Now let us delete the letters m and y. The simplest text deletion in vi editor can be done using x command. Press x on the keyboard and the character on the cursor position will be deleted.
  + Press x once again to delete letter *y*.
  + Press i to change the vi mode to Insert mode.
  + Type *puja’s.* Alternatively, you can type **$LOGNAME** in place of word **my** to display login name.
  + Press Esc, colon(:) , w and q to save and quit the editor.
  + Type sh myfile, and press Enter key and you will find that the output of the script is now displayed as shown in figure 12.6.

# Figure 12.6: Output of shell script

You can use dd to delete entire line on which the cursor is positioned.

Let us now understand that how the shell script got executed. So far we have been executing commands at the dollar prompt. When we execute a shell script instead of commands from the dollar prompt the shell accepts them from our shell script. It executes all the statements written in the script in an sequential manner.

# Why to use Shell Scripts

Note that any task that can be performed by writing a shell script can be accomplished by executing those commands at the dollar prompt. Then why to write a shell script? Shell scripts are written when we want to carry out repetitive tasks. Thus instead of spending time in typing the commands on the prompt every time a task is to be done, prepare a shell script and executing it by typing a single line command.

Following is an example of a very simple shell script, which gives a brief indication of the types of things you might do. Suppose occasionally you want the system to tell you your user id, what directory you are in, list the files in that directory, and display the date. You want each of these pieces of information to be preceded by an appropriate statement. The individual Linux commands you need are `echo,' `who am i,' `pwd,' `ls,' and `date.' So we would create text file with name *script1.sh* with the following lines in it.

**echo you are**

**who am i**

**echo The directory you are in is pwd**

**echo The files in this directory are ls**

**echo The date is date**

You can execute the file called *script1.sh*, when you are in the same directory it is in, simply by entering sh script1.sh. When you do this, the following information might be displayed on your screen. Note that .sh in above file name does not signify extension of a file as in Windows operating system. We have ended file name with .sh so that by looking name of the file itself we can identify that it is a shell script.

# Figure 12.7: Executing Script1.sh

Shell scripts can also be useful in modifying the working and display output of commands. For example, we have discussed date command in the last chapter which displays date and time both. We can write a shell script named time such that it displays only time and not the date. Type the script as shown in script3. Save the file and execute it to display current time.

**#Script 2: Shell script to display current time**

**echo "Current time is `date "+ %H: %M: %S"`"**

Once you learn to create shell scripts and use them, you should see that shell scripts are useful Linux tools that might save you a lot of time in your day-to-day computing.

# Shell variables

Shell variables like any other programming language are integral part of shell programming. They allow us to store and manipulate the values. Let us write a shell script which accepts a file name and displays total number of lines and words in it.

**#Script3 : Shell script to display name and age clear**

**name="Harshit Jain" age=25**

**echo "My name is $name and age is $age"**

Type the script 3 and execute it. Output as shown in Figure 12.8 will be displayed.

# Figure 12.8: Execution of script 2

Now let us understand above script. The first line begins with a #.This is the symbol which marks the beginning of the comment. Comments are messages that help in understanding the script. Thus this statement will not be executed. Second line is a command which clears the screen. Third and Fourth statement is assigning a value to a variable using a simple assignment operator. The strings should be enclosed in double quotes as shown in the script if it contains white spaces.

Note that while assigning values to the variables using the assignment operator „=‟, there should be no spaces on either side of =. If you leave a space the shell will try to interpret the value being assigned as a command to be executed. While carrying out the assignment if the variable does not exist, it will be created and the value would be assigned to the variable. On the other hand, if the variable already exists, old value will be overwritten. The fifth statement is extracting the value of a variable. The $ symbol preceding the variable name means instructing shell to extract the value of the variable. Thus the echo statement written in the above script displays the value of name and age on the screen.

The rules for building shell variable are as follows:

1. A variable name can be a combination of alphabets, digits and an underscore („\_‟).
2. No commas or blanks allowed within a variable name.
3. The first character of a variable name must either be an alphabet or an underscore.

# Interacting with user

Interacting with a shell script is possible only if we have shell commands ready to accept the value from the user. Read command reads in data from the standard input device. This command takes whatever you type and will place it in the variable name supplied to

it as an argument. Let us write a shell script which takes input (a file name) from a user, reads it and displays back the total number of lines and words in that file.

**#Script 4: Shell script to display total number of lines and words in that file**

**echo -n "Enter file name: " read fname**

**echo "Total number of lines in a file is `cat $fname | wc -l `" echo "Total number of words in a file is `cat $fname | wc -c`"**

Type the script 4 and name it script4.sh. Now execute the script as shown in the figure 12.9.

# Figure 12.9: Execution of script 4

Note that –n is a command-line parameter to echo command to tell echo not to print a new line after the text. You might have noticed earlier that by default, echo outputs a new line after displaying its argument. Thus in script 3, user will be able to type the name in the same line where the statement “Enter file name :” is displayed.

# Shell arithmetic

So far we have seen how to assign values to variables. Let us now see how we can operate upon these values. Let us write a shell script that accepts birth year from the user and displays current age in years. Type the script as shown in script 5.

**# Script 5: This script calculates age in years**

**clear**

**echo -n "Enter your birth year " read byear**

**cyear=`date | tr -s ' ' | cut -d " " -f 6` age=`expr $cyear - $byear`**

**echo "Your current age in years is $age"**

Look at the fifth statement which performs addition on values stored in two variables and stores the result in the third variable. expr is a command in Linux which is capable of evaluating an arithmetic expression. - is an operator used for subtraction. Note that thre should be one space between operator (-) and operands ($cyear, $byear). Additionally, there should be no space before and after = operator. Save the script with name script5.sh and execute it. Input 1982 and the correct age according to the current year will be displayed as a result.

To perform addition, multiplication, division and modular division, +, \*, / and % can be used respectively. Note that while performing multiplication prefix the multiplication symbol with backslash (\) character. Otherwise the shell will treat it as a wildcard character for all files in the current directory. While evaluating an expression expr performs the various arithmetic operations according to the following priorities.

/ , \* , % - First

+ , - Second

In case of a tie between operations of same priority, preference is given to the operator which occurs first. To force one operation to be performed earlier than the other, enclose it in brackets. For ex, in the expression $a \\* \($b + $c \) /$d, since the operation $b + $c is parenthesized it would be evaluated earlier than multiplication or division. The parentheses have been preceded by a „\‟ to take away its special meaning.

# Taking Decisions

Till now we have seen shell scripts in which the various steps are executed in the same order in which they appear in the program. This is the default nature of any programming language. At times, we require executing statement(s) depending on some situation. We may also have some statements to be executed in one situation and others in different situation. Linux bash shell offers following four decision making instructions:

* 1. if-then-fi
  2. if-then-else-fi
  3. if-then-elif-else-fi
  4. case-esac

Unlike if-constructs of other languages, if statement of Linux is also concerned with the exit status of a command. The exit status indicates whether the command was successfully executed or not. The exit status of command is 0 if it has been executed successfully; otherwise it is set to 1. Let is write a script to compare two files and display a message if they are same else it will display a message that they are different. For writing this shell script we will have to use **cmp** command which compares two files passed to it as an argument.

**#Script 6: To compare two files**

**clear**

**echo "Enter First file" read file1**

**echo "Enter Second file" read file2**

**if cmp $file1 $file2 then**

**echo "Files are same………" else**

**echo "Files are different…."**

**fi**

Script 6 accepts two files from a user, compares them and displays the appropriate message. Output of command can also be used in if condition statement. For instance, script 7 displays Good morning, Good afternoon or Good evening depending on the output of date command.

**#Script 7: This script greets a user with a message.**

**clear**

**hour=` date +"%H"`**

**if [ $hour –ge 0 –a $hour –lt 12 ] then**

**echo “Good morning”**

**elif [ $hour –ge 12 –a $hour –lt 18 ] then**

**echo “Good afternoon” else**

**echo “Good evening” fi**

We can also carry out numerical tests in Linux. Let us write a shell script which accepts a month from the user and displays whether more than 5 files are created in that month.

**#Script 8 : Script to check whether more than 5 files are created in a month c=`ls -l | grep -c "$1"`**

**if [ $c -gt 5 ] then**

**echo "Yes you have created more than 5 files in $1 month"**

**fi**

To execute script 8, execute it as **check.sh Aug** as shown in Figure 12.10, where check.sh is a file name and Aug is the name of the month. This Aug will be stored in $1. Linux stores the values provided through command line in dollar variables, named $1, $2, $3 and so on. First argument will be stored in $1, second in $2, third in $3 and so on till $9. These are known as command line arguments. First line of the script searches and counts the files created in the month supplied as an argument, and it stores the result in the variable c. The third line checks whether more than 5 files are created.

# Figure 12.10: Execution of script 7

Note that the condition is enclosed in a square bracket and there should be one space after opening square bracket and one before closing square bracket. If the condition is evaluated to true then statements typed inside **then** block will be executed else not. Thus if a more than 5 files are found then only a message will be displayed else nothing will be displayed while executing this script. **fi** is the end of the if statement. Note that **then** should be typed below if statement.

# The test command

Linux also provides test command which can be used in place of square brackets used in script 8. Let us write script 8 again using test statement.

**#Script 9 : Script to check whether more than 5 files are created in a month c=`ls -l | grep -c "$1"`**

**if test $c -gt 5 then**

**echo "Yes you have created more than 5 files in $1 month"**

**fi**

The test command can carry out several types of tests like numerical tests, string tests and file tests. As name suggests, numerical tests are used when comparisons between values of two numbers is to be done. We have used –gt operator in script 8 and script 9. The other operators that can be used for numerical test are as shown in table 12.4.

|  |  |
| --- | --- |
| **Operator** | **Usage** |
| -gt | **greater than** |
| -lt | **less than** |

|  |  |
| --- | --- |
| -ge | **greater than or equal to** |
| -le | **less than or equal to** |
| -ne | **not equal to** |
| -eq | **equal to** |

# Table 12.4: Numeric comparison operators

**File tests**

The test command in Linux has several options for checking the status of a file as shown in table 12.5. Using it we can find out whether the specified file is an ordinary file or a directory, or whether it has read, write or execute permissions etc.

|  |  |
| --- | --- |
| **Condition** | **Usage** |
| -s file | True if the file exists and is of size greater than 0 |
| -f file | True if file exists and is not a directory |
| -d file | True if file exists and is a directory file |
| -r file | True if file exists and you have read permission on it |
| -w file | True if file exists and you have write permission on it |
| -x file | True if file exists and you have execute permission on it |

# Table 12.5: File tests

Let us write a shell script which works like a cp command to copy one file to other.

**#Script 10: Shell script to create a copy of a file if [ -r $1 ]**

**then**

**cp $1 $2**

**echo "File copied successfully. "**

**else**

**echo "You should have read permission on source file to create its copy.." fi**

To execute script 10, type **copy.sh file1 file2**, where copy.sh is name of the script, file1 is source file and file2 is target file. Script 10 uses two command line arguments. First argument is used as a source file and second as a target file. It then checks whether source file exists in the current directory and has read permission or not. If condition is evaluated to true then file is copied else the message saying "You should have read permission on source file to create its copy.” will be displayed.

# Logical Operators

Shell allows usage of three logical operators while testing a condition. These are:

1. -a (AND)
2. -o (OR)
3. ! (NOT)

The first two operators –a and –o allows us to combine two or more conditions to be combined in a test. Not operator is used to negate a condition. Let us see how they can be used in a shell script by writing a shell script to check whether a file has read, write and execute permission or not.

**#Script 11: Shell script to check whether the file has all the permissions or not**

**if test -r $1 -a -w $1 -a -x $1 then**

**echo "You have all the permissions on the file." fi**

To execute above script type name of the file and pass file name as command line argument with it as shown in figure 12.10. Similarly –o operator can be used. The usage of NOT (!) operator is shown in script 12.

**#Script 12: Shell script to check whether the file has all the permissions or not**

**if ! test -r $1 -a -w $1 -a -x $1 then**

**echo "You do not have all the permissions on the file." fi**

# Hierarchy of Logical Operators

Shell has its own priorities of operators while executing a statement with multiple operators. Table 12.5 summarizes the operators we have seen so far. The higher an operator is in the table, the higher is the priority.

|  |  |
| --- | --- |
| **Operators** | **Type** |
| ! | Logical Not |
| -lt,-gt,-le,-ge,-eq,-ne | Relational |
| -a | Logical And |
| -o | Logical Or |

# Table 12.5: Operators priority

As discussed earlier, test command can be used for string comparisons also. Table 12.6 shows string comparison conditions and its result.

|  |  |
| --- | --- |
| **Condition** | **Result** |
| string1 = string2 | True if the strings are same |

|  |  |
| --- | --- |
| string 1 != string2 | True if the strings are different |
| -n string | True of the length of the string is greater than 0 |
| -z string | True if the length of the string is zero |
| string | True if the string is not a null string |

Script 13 is a script to test whether two strings entered by user are equal or not.

**#Script 13: To check whether two strings are equal or not**

**clear**

**echo -n "Enter first string :" read str1**

**echo -n "Enter second string: " read str2**

**if [ $str1 = $str2 ] then**

**echo "Strings are equal"**

**else**

**echo "Strings are unequal"**

**fi**

Till now we have seen examples of if –then-fi and if-then-else-fi statements. The third form of if statement is if-then-elif-then-else-fi. This form is used when we require grouping of several alternatives one after the other.

So let us see the usage of it by writing a script that accepts three files from user and display the file with the smallest size.

**#Script 14: Script to find the file with the smallest size.**

**clear**

**echo $1 $2 $3**

**fsize1=`wc -c $1| cut -d " " -f 1` fsize2=`wc -c $2| cut -d " " -f 1` fsize3=`wc -c $3| cut -d " " -f 1`**

**if [ $fsize1 -lt $fsize2 ];then**

**if [ $fsize1 –lt $fsize3];then echo “$1 is the smallest”**

**fi**

**elif [ $fsize2 -lt $fsize1 ];then**

**if [ $fsize2 -lt $fsize3 ];then echo “$2 is the smallest”**

**fi**

**else**

**echo “$3 is the smallest”**

**fi**

Type script 14 and execute it with three files as arguments. The three arguments will be stored in shell positional parameters $1, $2 and $3. Second, Third and Fourth statements calculate size of the files. wc command counts the number of characters. It displays number of character followed by name of the file as an output. Thus the output is passed to the cut command and first field is cut from the output using space (“ “) as a delimiter. Using if condition, the script finds out the file with the smallest size.

We have discussed how if-statement can be used to execute statements based on certain conditions. But if-statements looks complex as number of possible actions increases. Whenever we have conditions with a number of choices and a script that requires specific actions for each choice, we can use case statement.

The general syntax of case statement is :

Case variable in Value1)

Command1 Command 2

….

;;

Value 2)

**# Script 15: This script displays list of current files and directories in the current directory according to user’s choice.**

**clear**

**echo “1. Only files”**

**echo “2.Only directories” echo “ 3.Hidden files” echo “ 4. Executable files” echo “ 5. Quit”**

**echo –n “Enter your choice [1-5] : ” read ch**

**case $ch in 1)**

**ls -l | grep ^-**

**;; 2)**

**ls -l | grep ^d**

**;;**

**3)**

**ls -a**

**;;**

**4)**

**ls -x**

**;;**

Command 1

Command 2

….

;;

\*)

esac

Command 1

Command 2

….

;;

Let us write a menu driven script which displays various options on the screen, accepts user‟s choice and performs the action according to his entered choice.

**5)**

**exit 1**

**;;**

**\*)**

**echo “Sorry ! Incorrect choice, Try again!!!!!”**

**;;**

**esac**

Type the above script and a menu will be displayed and then will be prompted to provide your choice. Enter choice 1 and you will find that list of files in the current directory is displayed. This is because in script 15 we have written separate sections for each possible choice values in the case statement. Case statement checks the value of variable (ch in script 15), and enters into the section with a matching variable value specified before the closing round brackets. It executes all the statements written in that section till two semicolons (;;) are found. Once these semicolons are found, shell starts executing statements written after end of case statement which is specified by esac keyword. If no statements are found after the end of case statement, the shell program is ended.

If user enters unexpected value, like any value except 1 to 5 in above example, then case statement includes a facility of specifying asterisk(\*). Note that it is not compulsory to specify a section with the asterisk (\*) as a value. It is used to handle unexpected input.

# Looping

The shell scripts we have seen so far were either sequential in a nature or were consisting of decision control structure. While writing scripts for certain tasks we may require performing an action multiple times. The process of repeating the same statement a given number of times is known as looping. Linux shell facilitates following three types of loop statements:

1. For statement
2. While statement
3. Until statement

Let us discuss each of them one by one.

The For loop allows us to specify a list of values in its statement. The loop is then executed for each value mentioned in the list. The general syntax for statement is as under:

for control-variable in value1, value2, value3….. do

done

command 1

command 2

command 3

Let us write a shell script to convert all the sub directories in the current directory to read only directories.

**#Script 16: To move all the empty files from the working directory to a directory #named empty.**

**clear**

**if ! [ -d empty] mkdir empty else**

**for i in `ls` do**

**if [ –s $i ] then**

**mv $i ./empty**

**fi done fi**

As discussed earlier, statements written in For loop between do and done will be executed once for each value specified in the for statement. In script 16, through the command **ls** names of the files and directories will be listed. For loop will be executed once for each name. If statement written within for whether the value generated is a file or a directory. If it is a directory then it will be shifted to directory named empty using mv command discussed in the previous chapter.

# Looping While a condition is True

Like for loop while loop repeats its block of commands a number of times. However, unlike for loop, the while loop iterates until the condition evaluates to false. The basic syntax is:

while [test\_condition] do

commands….

done

Script 17 is an example of while loop that demonstrates how a shell script can repeatedly execute the statements till the test condition in the loop is evaluated to false.

**#Script 17: Creating multiple directories**

**clear**

**echo -n "enter the number of directories you want to create:" read n**

**i=1**

**while [ $i -le $n ] do**

**echo –n “Enter the name of the directory: “ read dirname**

**mkdir $dirname**

**if [ $? -eq 1 ] ; then**

**echo “The creation of directory failed “**

**fi**

**i=`expr $i + 1` done**

Above script 17 uses i as initialization variable. The statements inside while loop will be executed until the condition is true. This means in script 17, statements will be executed till value of i is less than n. The value of i is incremented inside the while loop. If the condition is evaluated to true, then user is prompted to enter the name of the directory. Read statement accepts the name provided by the user and mkdir command creates the directory. $? contains exit status of the command. It contains 1 if the command is not executed successfully and 0 if the command is executed successfully. Thus in case of mkdir command fails ( for example a directory with the same name exists in the current directory), $? will contain 1 and the a message **“The creation of directory failed “** will be echoed on the screen.

# Until loop

The until loop is very similar to the while loop. However, unlike while which executes till the condition is true, until loop executes till the condition is false. The basic syntax of the until loop is:

until [test\_condition] do

commands….

done

For example, let us write a shell script that displays sum of first n natural numbers. This n will be provided by the user.

**#Script 18: Script to do the sum of first n natural numbers clear**

**echo –n “Enter the number “ read n**

**i=1 sum=0**

**until [ $i –gt $n ] do**

**sum=`expr $sum + $i` i=`expr $i + 1`**

**done**

**echo “The sum of $n numbers is $sum”**

So far, we have seen how we can use decision-making and looping constructs to write shell scripts. Script 19 is an example of shell script which uses all the constructs discussed above. It is a menu driven script demonstrating until-loop, to display list of files in a current directory, changing password, displaying current date and time and searching a word from a file.

**#Script 19: Menu driven program demonstrating until loop**

**choice=y echo $choice**

**until [ $choice = n ] do**

**clear**

**echo " "**

**echo " "**

**echo "a: List of files and directories in a current directory." echo "b: Display current working directory"**

**echo "c: Display current date and time" echo "d: Searching a word from file" echo "e: Exit"**

**echo " "**

**echo " "**

**echo -n "Enter your choice: [a-e] " read ch**

**case $ch in a)**

**ls -l**

**;;**

**b)**

**echo "You are working in `pwd`"**

**;;**

**c)**

**echo "Current date and time is is `date`"**

**;;**

**d)**

**echo -n "Enter the word to be searched : " read word**

**echo -n "Enter the file in which the word is to be searched" read file**

**grep $word $file**

**;;**

**e) exit**

**;;**

**\*)**

**echo "Oops!!! incorrect choice, try again.."**

**;;**

**esac**

**echo -n "Do you want to continue? : " read choice**

**done**

Type and execute above script 19. Enter different choice each time. The program will keep on executing till user enters e as a choice and exit statement is executed. After execution of any other option, user is prompted to show his wish to continue the program. If he enters y then the entire script will be executed again else the program will be terminated.

At the end of this chapter you might be aware that how we can give shell a list of task instead of a single task one by one. The shell script offers new horizons to combine the power of various operating system commands and features of programming language.

FCFS

*package* com.company;  
*import* java.util.\*;  
*public class* fcfs {  
 *public static void* main(String[] args) {  
 Scanner sc = *new* Scanner(System.***in***);  
 System.***out***.println("Enter the total number of process ");  
 *int* n = sc.nextInt();  
 *int* arival[] = *new int*[n];  
 *int* burst[] = *new int*[n];  
 *for* (*int* i = 0; i<n; i++){  
 System.***out***.println("Enter arrival time ");  
 arival[i] = sc.nextInt();  
 System.***out***.println("Enter burst time ");  
 burst[i]= sc.nextInt();  
 }  
 *int* start = 0;  
 *int* completion = 0;  
 *int* waiting[] = *new int*[n];  
 *int* tat [] = *new int*[n];  
 *for* (*int* i = 0; i<n; i++){  
 waiting[i] = start - arival[i];  
 start = start + burst[i];  
 tat[i] = completion + burst[i];  
 completion += burst[i];  
 }  
  
 *float* sum = 0.0f;  
 *float* summ = 0.0f;  
 *for*(*int* i = 0; i<n; i++){  
 sum += waiting[i];  
 summ += tat[i];  
 }  
 System.***out***.println("Average waiting time = "+sum/n);  
 System.***out***.println("Turn around time = "+summ/n);  
  
 }  
 }

SJF

*import* java.util.\*;  
*public class* sjf {  
 *static void* arrangeArrival(*int* num, *int*[][] mat) {  
 *for* (*int* i = 0; i < num; i++) {  
 *for* (*int* j = 0; j < num - i - 1; j++) {  
 *if* (mat[j][1] > mat[j + 1][1]) {  
 *for* (*int* k = 0; k < 5; k++) {  
 *int* temp = mat[j][k];  
 mat[j][k] = mat[j + 1][k];  
 mat[j + 1][k] = temp;  
 }  
 }  
 }  
 }  
 }  
 *static void* completionTime(*int* num, *int*[][] mat) {  
 *int* temp, val = -1;  
 mat[0][3] = mat[0][1] + mat[0][2];  
 mat[0][5] = mat[0][3] - mat[0][1];  
 mat[0][4] = mat[0][5] - mat[0][2];  
 *for* (*int* i = 1; i < num; i++) {  
 temp = mat[i - 1][3];  
 *int* low = mat[i][2];  
 *for* (*int* j = i; j < num; j++) {  
 *if* (temp >= mat[j][1] && low >= mat[j][2]) {  
 low = mat[j][2];  
 val = j;  
 }  
 }  
 mat[val][3] = temp + mat[val][2];  
 mat[val][5] = mat[val][3] - mat[val][1];  
 mat[val][4] = mat[val][5] - mat[val][2];  
 *for* (*int* k = 0; k < 6; k++) {  
 *int* tem = mat[val][k];  
 mat[val][k] = mat[i][k];  
 mat[i][k] = tem;  
 }  
 }  
 }  
 *static void* findAvgTime(*int* num, *int*[][] mat) {  
 *int* total\_waitingTime = 0, total\_turnAroundTime = 0;  
 *for* (*int* i = 0; i < num; i++) {  
 total\_waitingTime = total\_waitingTime + mat[i][4];  
 total\_turnAroundTime = total\_turnAroundTime + mat[i][5];  
 }  
 *float* s = (*float*) total\_waitingTime / (*float*) num;  
 *float* t = (*float*) total\_turnAroundTime / (*float*) num;  
 System.***out***.printf("\nAverage waiting time = %f", s);  
 System.***out***.printf("\n");  
 System.***out***.printf("Average turn around time = %f ", t);  
 }  
 *public static void* main(String[] args) {  
 Scanner sc = *new* Scanner(System.***in***);  
 System.***out***.print("Enter number of Process: ");  
 *int* num = sc.nextInt();  
 *int*[][] mat = *new int*[num][6];  
 *for* (*int* i = 0; i < num; i++) {  
 System.***out***.println("...Process " + (i + 1) + "...");  
 System.***out***.print("\tProcess Id: ");  
 mat[i][0] = sc.nextInt();  
 System.***out***.print("\tArrival Time: ");  
 mat[i][1] = sc.nextInt();  
 System.***out***.print("\tBurst Time: ");  
 mat[i][2] = sc.nextInt();  
 }  
 System.***out***.println("Before Arrange...");  
 System.***out***.println("Process ID\tArrival Time\tBurst Time");  
 *for* (*int* i = 0; i < num; i++) {  
 System.***out***.printf("%d\t\t%d\t\t%d\n", mat[i][0], mat[i][1], mat[i]  
 [2]);  
 }  
 *arrangeArrival*(num, mat);  
 *completionTime*(num, mat);  
 System.***out***.println("Final Result...");  
 System.***out***.println(  
 "Process ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time"  
 );  
 *for* (*int* i = 0; i < num; i++) {  
 System.***out***.printf(  
 "%d\t\t%d\t\t%d\t\t%d\t\t%d\n",  
 mat[i][0],  
 mat[i][1],  
 mat[i][2],  
 mat[i][4],  
 mat[i][5]  
 );  
 }  
 *findAvgTime*(num, mat);  
 sc.close();  
 }  
}

Round Robin

*package* com.company;*// Java program for implementation of RR scheduling  
import* java.util.\*;  
*public class* round\_robin  
{  
 *static void* findWaitingTime(  
 *int* processes[],  
 *int* n,  
 *int* burstTime[],  
 *int* waitingTime[],  
 *int* quantum  
 ) {  
 *int* rem\_burstTime[] = *new int*[n];  
 *for* (*int* i = 0; i < n; i++) {  
 rem\_burstTime[i] = burstTime[i];  
 }  
 *int* t = 0;  
 *while* (*true*) {  
 *boolean* done = *true*;  
 *for* (*int* i = 0; i < n; i++) {  
 *if* (rem\_burstTime[i] > 0) {  
 done = *false*;  
 *if* (rem\_burstTime[i] > quantum) {  
 t += quantum;  
 rem\_burstTime[i] -= quantum;  
 } *else* {  
 t = t + rem\_burstTime[i];  
 waitingTime[i] = t - burstTime[i];  
 rem\_burstTime[i] = 0;  
 }  
 }  
 }  
 *if* (done == *true*) *break*;  
 }  
 }  
 *static void* findTurnAroundTime(  
 *int* processes[],  
 *int* n,  
 *int* burstTime[],  
 *int* waitingTime[],  
 *int* turnAroundTime[]  
 ) {  
 *for* (*int* i = 0; i < n; i++) turnAroundTime[i] =  
 burstTime[i] + waitingTime[i];  
 }  
 *static void* findAvgTime(  
 *int* processes[],  
 *int* n,  
 *int* burstTime[],  
 *int* quantum  
 ) {  
 *int* waitingTime[] = *new int*[n], turnAroundTime[] = *new int*[n];  
 *int* total\_waitingTime = 0, total\_turnAroundTime = 0;  
 *findWaitingTime*(processes, n, burstTime, waitingTime, quantum);  
 *findTurnAroundTime*(processes, n, burstTime, waitingTime,  
 turnAroundTime);  
 System.***out***.println(  
 "\nProcesses " + " Burst time " + " Waiting time " + " Turn aroundtime");  
 *for* (*int* i = 0; i < n; i++) {  
 total\_waitingTime = total\_waitingTime + waitingTime[i];  
 total\_turnAroundTime = total\_turnAroundTime + turnAroundTime[i];  
 System.***out***.println(  
 " " +  
 (i + 1) +  
 "\t\t" +  
 burstTime[i] +  
 "\t " +  
 waitingTime[i] +  
 "\t\t " +  
 turnAroundTime[i]  
 );  
 }  
 System.***out***.println(  
 "\nAverage waiting time = " + (*float*) total\_waitingTime / (*float*) n  
 );  
 System.***out***.println(  
 "Average turn around time = " + (*float*) total\_turnAroundTime /  
 (*float*) n  
 );  
 }  
 *public static void* main(String[] args) {  
 Scanner sc = *new* Scanner(System.***in***);  
 System.***out***.print("Enter number of Process: ");  
 *int* num = sc.nextInt();  
 System.***out***.print("Enter value of Quantum: ");  
 *int* quantum = sc.nextInt();  
 *int* processes[] = *new int*[num];  
 *int* burstTime[] = *new int*[num];  
 *for* (*int* i = 0; i < num; i++) {  
 System.***out***.print("...Enter the BurstTime for process " + (i + 1) + ": ");  
 processes[i] = i + 1;  
 burstTime[i] = sc.nextInt();  
 }  
 *findAvgTime*(processes, num, burstTime, quantum);  
 sc.close();  
 }  
}

Priority

*package* com.company;  
*import* java.util.\*;  
  
*public class* Priority {  
 *public static void* main(String[] args) {  
 Scanner sc = *new* Scanner(System.***in***);  
 System.***out***.print("Enter Number of Process: ");  
 *int* numberOfProcess = sc.nextInt();  
 *int*[] process = *new int*[numberOfProcess];  
 *int* burstTime[] = *new int*[numberOfProcess];  
 *int* priority[] = *new int*[numberOfProcess];  
 *for* (*int* i = 0; i < numberOfProcess; i++) {  
 System.***out***.println("...Process " + (i + 1) + "...");  
 System.***out***.print("\tProcess Id : ");  
 process[i] = sc.nextInt();  
 System.***out***.print("\tBurst Time Time : ");  
 burstTime[i] = sc.nextInt();  
 System.***out***.print("\tPriority : ");  
 priority[i] = sc.nextInt();  
 }  
 *int* temp;  
 *int* temp2;  
 *for* (*int* i = 0; i < numberOfProcess - 1; i++) {  
 *for* (*int* j = 0; j < numberOfProcess - 1; j++) {  
 *if* (priority[j] > priority[j + 1]) {  
 temp = priority[j];  
 priority[j] = priority[j + 1];  
 priority[j + 1] = temp;  
 temp = burstTime[j];  
 burstTime[j] = burstTime[j + 1];  
 burstTime[j + 1] = temp;  
 temp2 = process[j];  
 process[j] = process[j + 1];  
 process[j + 1] = temp2;  
 }  
 }  
 }  
 *int* TAT[] = *new int*[numberOfProcess + 1];  
 *int* waitingTime[] = *new int*[numberOfProcess + 1];  
 *for* (*int* i = 0; i < numberOfProcess; i++) {  
 TAT[i] = burstTime[i] + waitingTime[i];  
 waitingTime[i + 1] = TAT[i];  
 }  
 *int* totalWT = 0;  
 *int* totalTAT = 0;  
 *double* avgWT;  
 *double* avgTAT;  
 System.***out***.println(  
 "Process ID\tBurst Time\tWaiting Time\tTurn Around Time"  
 );  
 *for* (*int* i = 0; i < numberOfProcess; i++) {  
 System.***out***.println(  
 process[i] +  
 "\t\t" +  
 burstTime[i] +  
 "\t\t" +  
 waitingTime[i] +  
 "\t\t" +  
 (TAT[i])  
 );  
 totalTAT += (waitingTime[i] + burstTime[i]);  
 totalWT += waitingTime[i];  
 }  
 avgWT = totalWT / (*double*) numberOfProcess;  
 avgTAT = totalTAT / (*double*) numberOfProcess;  
 System.***out***.println("\n Average Waiting Time: " + avgWT);  
 System.***out***.println(" Average Turn Around Time: " + avgTAT);  
 sc.close();  
 }  
}

Samaphore

*package* com.company;  
*import* java.util.concurrent.Semaphore;  
*class* Q {  
 *// an item  
 int* item;  
 *// semCon initialized with 0 permits  
// to ensure put() executes first  
 static* Semaphore *semCon* = *new* Semaphore(0);  
 *static* Semaphore *semProd* = *new* Semaphore(1);  
 *// to get an item from buffer  
 void* get() {  
 *try* {  
*// Before consumer can consume an item,  
// it must acquire a permit from semCon  
 semCon*.acquire();  
 } *catch* (InterruptedException e) {  
 System.***out***.println("InterruptedException caught");  
 }  
*// consumer consuming an item* System.***out***.println("Consumer consumed item : " + item);  
*// After consumer consumes the item,  
// it releases semProd to notify producer  
 semProd*.release();  
 }  
 *// to put an item in buffer  
 void* put(*int* item) {  
 *try* {*// Before producer can produce an item,  
// it must acquire a permit from semProd  
 semProd*.acquire();  
 } *catch* (InterruptedException e) {  
 System.***out***.println("InterruptedException caught");  
 }  
*// producer producing an item  
 this*.item = item;  
 System.***out***.println("Producer produced item : " + item);  
*// After producer produces the item,  
// it releases semCon to notify consumer  
 semCon*.release();  
 }  
}  
*// Producer class  
class* Producer *implements Runnable* {  
 Q q;  
 Producer(Q q) {  
 *this*.q = q;  
 *new* Thread(*this*, "Producer").start();  
 }  
 *public void* run() {  
 *for* (  
 *int* i = 0;  
 i < 5;  
 i++  
 ) q.put(i); *// producer put items* }  
}  
*// Consumer class  
class* Consumer *implements Runnable* {  
 Q q;  
 Consumer(Q q) {  
 *this*.q = q;  
 *new* Thread(*this*, "Consumer").start();  
 }  
 *public void* run() {  
 *for* (  
 *int* i = 0;  
 i < 5;  
 i++  
 ) q.get(); *// consumer get items* }  
}  
*// Driver class  
class* PC {  
 *public static void* main(String args[]) {  
*// creating buffer queue* Q q = *new* Q();  
*// starting consumer thread  
 new* Consumer(q);  
*// starting producer thread  
 new* Producer(q);  
 }  
}

Banker’s Algorithms

*package* com.company;  
  
*public class* bankers\_algorithm {  
 *static int P* = 5;  
 *static int R* = 3;  
 *//Function to find the need of each process  
 static void* calculateNeed(*int* need[][], *int* maxm[][], *int* allot[][]) {  
 *for* (*int* i = 0; i < *P*; i++) {  
 *for* (*int* j = 0; j < *R*; j++) {  
 need[i][j] = maxm[i][j] - allot[i][j];  
 }  
 }  
 }  
 *//Function to find the system is in safe state or not  
 static boolean* isSafe(  
 *int* processes[],  
 *int* avail[],  
 *int* maxm[][],  
 *int* allot[][]  
 ) {  
 *int*[][] need = *new int*[*P*][*R*];  
 *calculateNeed*(need, maxm, allot);  
 *boolean*[] finish = *new boolean*[*P*];  
 *int*[] safeSeq = *new int*[*P*];  
 *int*[] work = *new int*[*R*];  
 *for* (*int* i = 0; i < *R*; i++) {  
 work[i] = avail[i];  
 }  
 *int* count = 0;  
 *while* (count < *P*) {  
 *boolean* found = *false*;  
 *for* (*int* p = 0; p < *P*; p++) {  
 *if* (finish[p] == *false*) {  
 *int* j;  
 *for* (j = 0; j < *R*; j++) {  
 *if* (need[p][j] > work[j]) {  
 *break*;  
 }  
 }  
 *if* (j == *R*) {  
 *for* (*int* k = 0; k < *R*; k++) {  
 work[k] += allot[p][k];  
 }  
 safeSeq[count++] = p;  
 finish[p] = *true*;  
 found = *true*;  
 }  
 }  
 }  
 *if* (found == *false*) {  
 System.***out***.print("System is not in safe state");  
 *return false*;  
 }  
 }  
 System.***out***.print("System is in safe state.\nSafe" + " sequence is: ");  
 *for* (*int* i = 0; i < *P*; i++) {  
 System.***out***.print(safeSeq[i] + " ");  
 }  
 *return true*;  
 }  
 *public static void* main(String[] args) {  
 *int* processes[] = { 0, 1, 2, 3, 4 };  
 *int* avail[] = { 3, 3, 2 };  
 *int* maxm[][] = {  
 { 7, 5, 3 },  
 { 3, 2, 2 },  
 { 9, 0, 2 },  
 { 2, 2, 2 },  
 { 4, 3, 3 },  
 };  
 *int* allot[][] = {  
 { 0, 1, 0 },  
 { 2, 0, 0 },  
 { 3, 0, 2 },  
 { 2, 1, 1 },  
 { 0, 0, 2 },  
 };  
 *isSafe*(processes, avail, maxm, allot);  
 }  
}

FIFO

*package* com.company;  
*import* java.util.HashSet;  
*import* java.util.LinkedList;  
*import* java.util.*Queue*;  
*class* FIFO {  
 *// Method to find page faults using FIFO  
 static int* pageFaults(*int* pages[], *int* n, *int* capacity) {  
*// To represent set of current pages. We use  
// an unordered\_set so that we quickly check  
// if a page is present in set or not* HashSet<Integer> s = *new* HashSet<>(capacity);  
*// To store the pages in FIFO manner  
 Queue*<Integer> indexes = *new* LinkedList<>();  
 *// Start from initial page  
 int* page\_faults = 0;  
 *for* (*int* i = 0; i < n; i++) {  
*// Check if the set can hold more pages  
 if* (s.size() < capacity) {  
*// Insert it into set if not present  
// already which represents page fault  
 if* (!s.contains(pages[i])) {  
 s.add(pages[i]);  
*// increment page fault* page\_faults++;  
*// Push the current page into the queue* indexes.add(pages[i]);  
 }  
 }  
*// If the set is full then need to perform FIFO  
// i.e. remove the first page of the queue from  
// set and queue both and insert the current page  
 else* {  
*// Check if current page is not already  
// present in the set  
 if* (!s.contains(pages[i])) {  
*//Pop the first page from the queue  
 int* val = indexes.peek();  
 indexes.poll();  
*// Remove the indexes page* s.remove(val);  
*// insert the current page* s.add(pages[i]);  
*// push the current page into  
// the queue* indexes.add(pages[i]);  
*// Increment page faults* page\_faults++;  
 }  
 }  
 }  
 *return* page\_faults;  
 }  
 *public static void* main(String args[]) {  
 *int* pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};  
 *int* capacity = 4;  
 System.***out***.println(*pageFaults*(pages, pages.length, capacity));  
 }  
  
}

LRU

*package* com.company;  
*import* java.util.ArrayList;  
*public class* LRU {  
 *public static void* main(String[] args) {  
 *int* capacity = 4;  
 *int* arr[] = { 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 };  
*// To represent set of current pages.We use  
// an Arraylist* ArrayList<Integer> s = *new* ArrayList<>(capacity);  
 *int* count = 0;  
 *int* page\_faults = 0;  
 *for* (*int* i : arr) {  
*// Insert it into set if not present  
// already which represents page fault  
 if* (!s.contains(i)) {  
*// Check if the set can hold equal pages  
 if* (s.size() == capacity) {  
 s.remove(0);  
 s.add(capacity - 1, i);  
 } *else* s.add(count, i);  
*// Increment page faults* page\_faults++;  
 ++count;  
 } *else* {  
*// Remove the indexes page* s.remove((Object) i);  
*// insert the current page* s.add(s.size(), i);  
 }  
 }  
 System.***out***.println(page\_faults);  
 }  
}