BHARATI VIDYAPEETH DEEMED UNIVERSITY
COLLEGE OF ENGINEERING, PUNE - 43

DEPARTMENT OF COMPUTER ENGINEERING

Lab Manual

Computer Operating System

B.Tech. Computer Sem IV

DEPARTMENT OF COMPUTER ENGINEERING

VISION OF THE INSTITUTE

“To be World Class Institute for Social Transformation through Dynamic Education”

MISSION OF THE INSTITUTE

- To provide quality technical education with advanced equipment, qualified faculty members, infrastructure to meet needs of profession and society.
- To provide an environment conductive to innovation, creativity, research and entrepreneurial leadership.
- To practice and promote professional ethics, transparency and accountability for social community, economic and environmental conditions.

VISION OF THE DEPARTMENT

“To pursue and excel in the endeavour for creating globally recognized computer engineers through quality education”.

MISSION OF THE DEPARTMENT

1. To impart engineering knowledge and skills confirming to a dynamic curriculum.

2. To develop professional, entrepreneurial & research competencies encompassing continuous intellectual growth.

3. To produce qualified graduates exhibiting societaland ethical responsibilities in working environment.

PROGRAM EDUCATIONAL OBJECTIVES

Graduates upon completion of B. Tech Computer Engineering Programme will able to**:**

1.Demonstrate technical and professional competencies by applying engineering fundamentals, computing principles and technologies.

2.Learn, Practice, and grow as skilled professionals/ entrepreneur/researchers adapting to the evolving computinglandscape.

3.Demonstrate professional attitude, ethics, understanding of social context and interpersonal skills leading to a successful career.

PROGRAM SPECIFIC OUTCOMES:

PSO 1: To design, develop and implement computer programs onhardware towards solving problems.

PSO 2: To employ expertise and ethical practise through continuing intellectual growth and adapting to the working environment.

PROGRAMME OUTCOMES

1. Apply the knowledge of mathematics, science, engineering fundamentals, and computing for the solution of complex engineering problems.
2. Identify, formulate, research literature, and analyse complex engineering problems reaching substantiated conclusions using computer engineering foundations, principles, and technologies.
3. Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.
4. Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
6. Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues, and the consequent responsibilities relevant to the professional engineering practice.
7. Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and the need for sustainable development.
8. Apply ethical principles while committed to professional responsibilities and norms of the engineering practice.
9. Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings
10. Communicate effectively on complex engineering activities with the engineering community and with the society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. Apply the engineering and management principles to one’s work, as a member and leader in a team.
12. Recognise the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

GENRAL INSTUCTIONS:

- Equipment in the lab is meant for the use of students. Students need to maintain a proper decorum in the computer lab. Students must use the equipment with care.
- Students are required to carry their reference materials, files and records with completed assignment while entering the lab.
- Students are supposed to occupy the systems allotted to them and are not supposed to talk or make noise in the lab.
- All the students should perform the given assignment individually.
- Lab can be used in free time/lunch hours by the students who need to use the systems should take prior permission from the lab in-charge.
- All the Students are instructed to carry their identity cards when entering the lab.
- Lab files need to be submitted on or before date of submission.
- Students are not supposed to use pen drives, compact drives or any other storage devices in the lab.
- For Laboratory related updates and assignments students should refer to the notice board in the Lab.

EXAMINATION SCHEME

Practical Exam: 25 Marks

Term Work: 25 Marks

Total: 50 Marks

Minimum Marks required: 10 Marks each head

PROCEDURE OF EVALUATION

Each practical/assignment shall be assessed continuously on the scale of 25 marks. The distribution of marks as follows.

Sr. No	Evaluation Criteria	Marks for each Criteria	Rubrics
1	**Timely Submission**	**07**	➢ Punctuality reflects the work ethics. Students should reflect that work ethics by completing the lab assignments and reports in a timely manner without being reminded or warned.
2	**Presentation**	**06**	➢ Student are expected to write the technical document (lab report) in their own words. The presentation of the contents in the lab report should be complete, unambiguous, clear, understandable. The report should document approach/algorithm/design and code with proper explanation.
3	**Understanding**	**12**	➢ Correctness and Robustness of the code is expected. The Learners should have an in-depth knowledge of the practical assignment performed.The learner should be able to explain methodology used for designing and developing the program/solution. He/she should clearly understand the purpose of the assignment and its outcome.

LABORATORY USAGE

Students use Linux/Unix on computers for executing the lab experiments, document the results and to prepare the technical documents for making the lab reports.

OBJECTIVES:-

To develop ability to use the computational languages and shell script necessary for engineering practice.

PRACTICAL PRE-REQUISITE:-

1. Basic knowledge of Computer architecture and Design, Data structures and algorithms, Programming Skills, Engineering Mathematics.

HARDWARE/ SOFTWARE REQUIREMENTS:-

1. Linux/Unis OS.
2. Software required for shell script

COURSE OUTCOMES:

1. To learn and apply the Concepts of operating system
2. Infer the concept of process, thread and Inter process communication
3. Outline the concept of concurrency and deadlocks.
4. Analyse of Memory Management and Virtual Memory
5. Utilize the concepts of I/O System for communication.
6. Illustrate the Issues in real time operating system.

DEPARTMENT OF COMPUTER ENGINEERING

Academic Year: 2022-23

Program: B.Tech.(Computer Engineering) Semester: - IV
Course: Computer Operating System

List of Assignments

No	Detail of Assignment
1.	Study basics of operating system and installation of Linux operating system.
2.	Study basic linux commands and implement it.
3.	WAP to copy contents of one file into another in linux(in same directory and different directory).
4.	Write a programme (using control statements) a) To find smallest and greatest among all numbers. b) To display factorial of given number. c) To display Fibonacci series for given input.
5.	Write a programme (using control statements) a) To display percentage and grade obtained by student. b) To implement various calculator operations. c) To display whether given numbers are divisible by n.
6.	WAP to implement first come, first served (FCFS) scheduling algorithm.
7.	WAP to implement Shortest Job First(SJF) algorithm.
8.	Understand and implement Round Robin algorithm.
9.	Implement Producer Consumer Problem.
10.	Study and implement Page replacement algorithm.

Course Co-ordinator **Head**

Practical Experiment 1:

AIM: Study basic of operating system and installation of Linux operating system.

UNIX:

It is a multi-user operating system. Developed at AT & T Bell Industries, USA in 1969.

Ken Thomson along with Dennis Ritchie developed it from MULTICS (Multiplexed Information and Computing Service) OS.

By1980, UNIX had been completely rewritten using C language.

LINUX:

It is similar to UNIX, which is created by Linus Torualds. All UNIX commands works in Linux. Linux is a open source software. The main feature of Linux is coexisting with other OS such as windows and UNIX.

STRUCTURE OF A LINUXSYSTEM:

It consists of three parts.

a) UNIX kernel
b) Shells
c) Tools and Applications

UNIX KERNEL:

Kernel is the core of the UNIX OS. It controls all tasks, schedule all Processes and carries out all the functions of OS.

Decides when one programs tops and another starts.

SHELL:

Shell is the command interpreter in the UNIX OS. It accepts command from the user and analyses and interprets them

What is Kernel

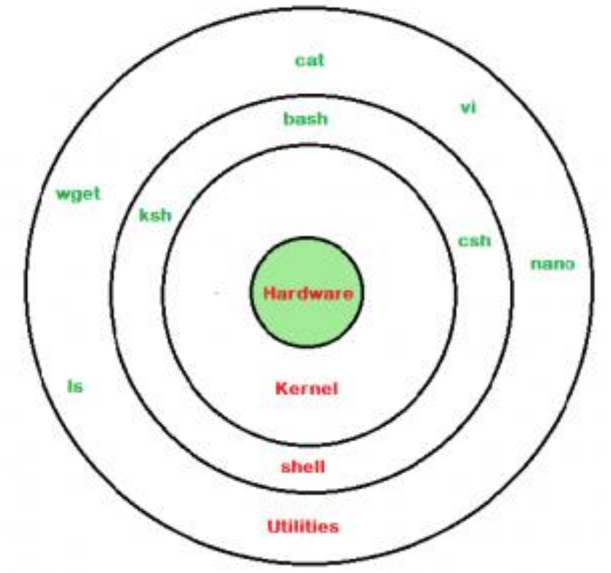
The kernel is a computer program that is the core of a computer’s operating system, with complete control over everything in the system. It manages following resources of the Linux system –

- File management
- Process management
- I/O management
- Memory management
- Device management etc.

It is often mistaken that **Linus Torvalds** has developed Linux OS, but actually he is only responsible for development of Linux kernel. Complete Linux system = Kernel + GNU system utilities and libraries + other management scripts + installation scripts.

What is Shell

A shell is special user program which provide an interface to user to use operating system services. Shell accept human readable commands from user and convert them into something which kernel can understand. It is a command language interpreter that execute commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or start the terminal.



Shell is broadly classified into two categories –

- Command Line Shell
- Graphical shell

Command Line Shell

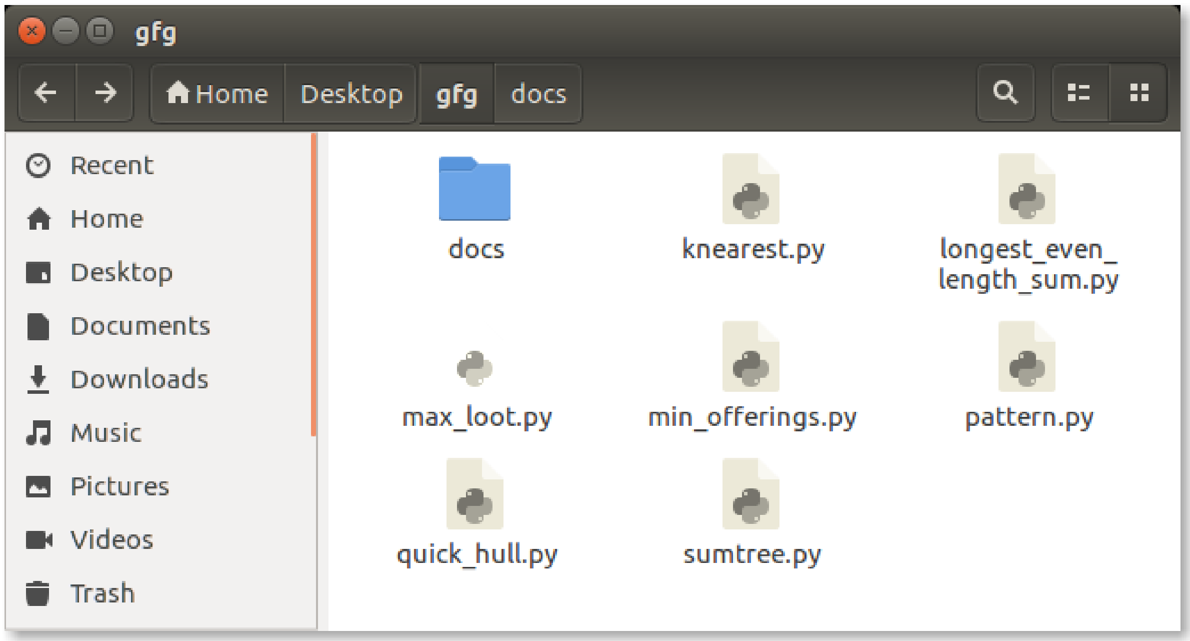
Shell can be accessed by user using a command line interface. A special program called **Terminal** in linux/macOS or **Command Prompt** in Windows OS is provided to type in the human readable commands such as "cat", "ls" etc. and then it is being execute. The result is then displayed on the terminal to the user.

In above screenshot **"ls"** command with **"-l"** option is executed.

It will list all the files in current working directory in long listing format. Working with command line shell is bit difficult for the beginners because it's hard to memorize so many commands. It is very powerful, it allows user to store commands in a file and execute them together. This way any repetitive task can be easily automated. These files are usually called **batch files** in Windows and **Shell** Scripts in Linux/macOS systems.

Graphical Shells

Graphical shells provide means for manipulating programs based on graphical user interface (GUI), by allowing for operations such as opening, closing, moving and resizing windows, as well as switching focus between windows. Window OS or Ubuntu OS can be considered as good example which provide GUI to user for interacting with program. User do not need to type in command for every actions.A typical GUI in Ubuntu system –



GUI shell

There are several shells are available for Linux systems like –

- BASH (Bourne Again SHell) – It is most widely used shell in Linux systems. It is used as default login shell in Linux systems and in macOS. It can also be installed on Windows OS.
- CSH (C SHell) – The C shell's syntax and usage are very similar to the C programming language.
- KSH (Korn SHell) – The Korn Shell also was the base for the POSIX Shell standard specifications etc.

Each shell does the same job but understand different commands and provide different built in functions.

Practical Experiment 2

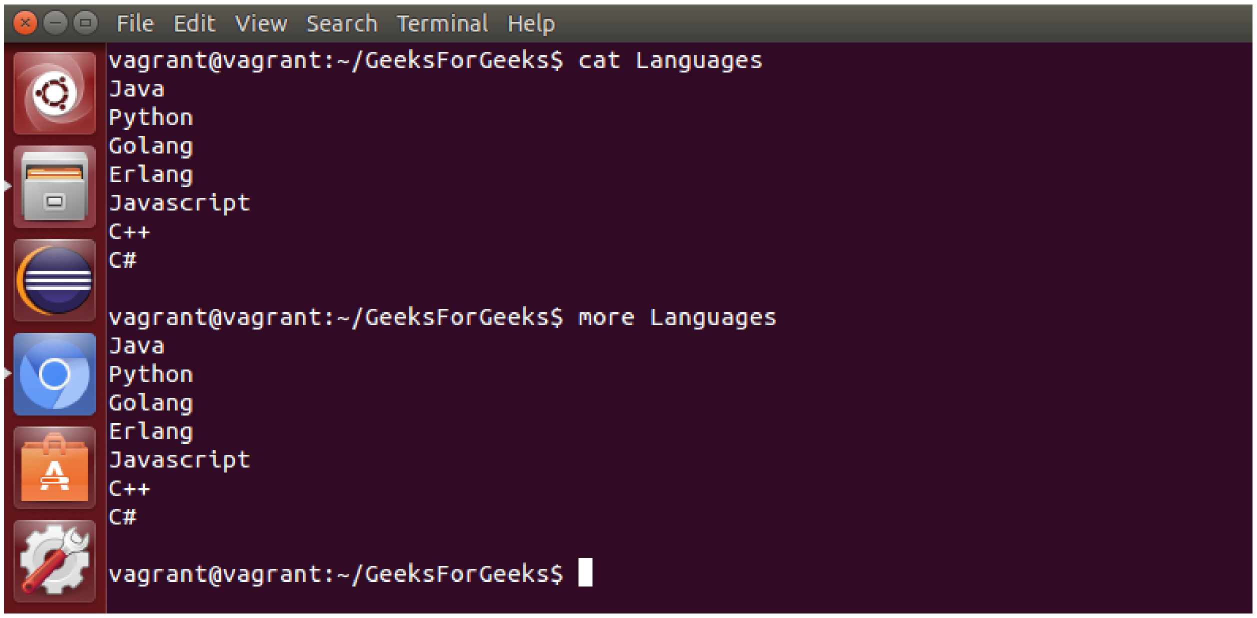
AIM: Study basic linux commands and implement it.

1). **Basic Terminal Navigation Commands:**

- **ls** : To get the list of all the files or folders.
- **ls -l:** Optional flags are added to **ls** to modify default behavior, listing contents in extended form **-l** is used for **"long" output**
- **ls -a:** Lists of all files including the hidden files, add **-a flag**
- **cd**: Used to change the directory.
- **du**: Show disk usage.
- **pwd**: Show the present working directory.
- **man**: Used to show the manual of any command present in Linux.
- **rmdir**: It is used to delete a directory if it is empty.
- **ln file1 file2**: Creates a physical link.
- **ln -s file1 file2**: Creates a symbolic link.
- **locate:** It is used to locate a file in Linux System
- **echo:** This command helps us move some data, usually text into a file.
- **df:** It is used to see the available disk space in each of the partitions in your system.
- **tar:** Used to work with tarballs (or files compressed in a tarball archive)

2). **Displaying the file contents on the terminal:**

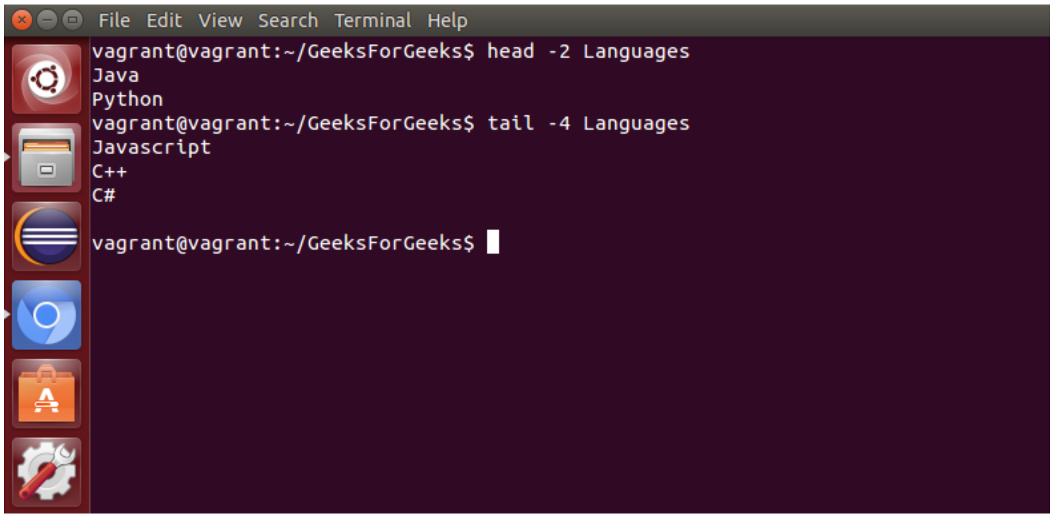
- **cat**: It is generally used to concatenate the files. It gives the output on the standard output.
- **more**: It is a filter for paging through text one screenful at a time.



```
vagrant@vagrant:~/GeeksForGeeks$ cat Languages
Java
Python
Golang
Erlang
Javascript
C++
C#

vagrant@vagrant:~/GeeksForGeeks$ more Languages
Java
Python
Golang
Erlang
Javascript
C++
C#

vagrant@vagrant:~/GeeksForGeeks$
```

- **less**: It is used to viewing the files instead of opening the file.Similar to *more* command but it allows backward as well as forward movement.

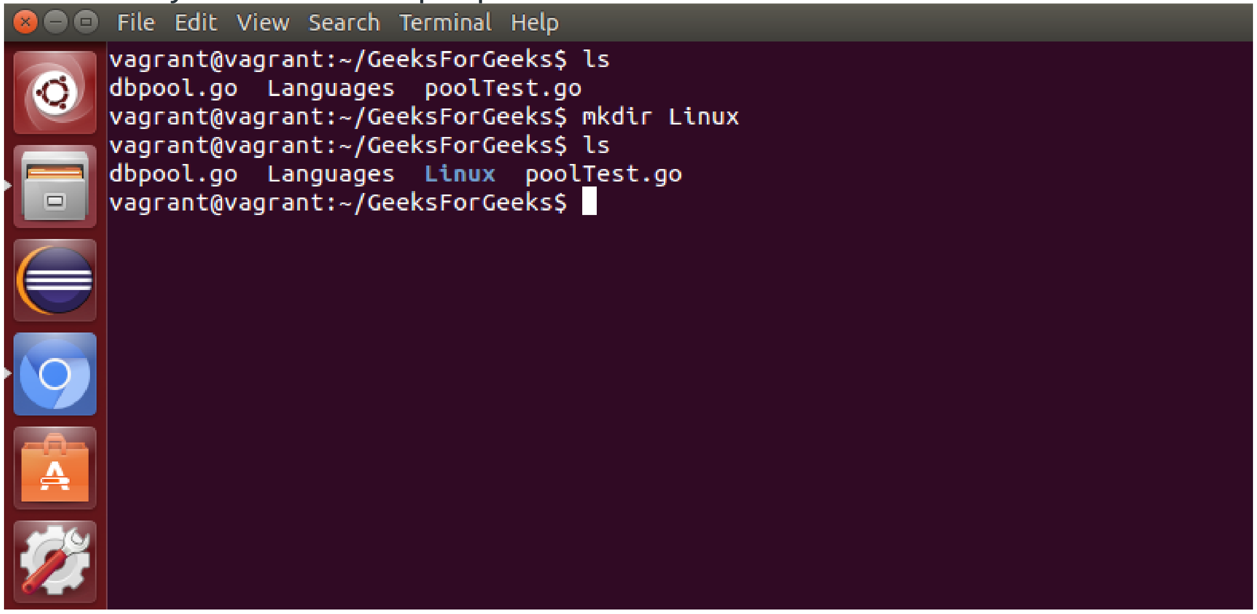
  - **head** : Used to print the first N lines of a file. It accepts N as input and the default value of N is 10.
  - **tail** : Used to print the last N-1 lines of a file. It accepts N as input and the default value of N is 10.



```
vagrant@vagrant:~/GeeksForGeeks$ head -2 Languages
Java
Python
vagrant@vagrant:~/GeeksForGeeks$ tail -4 Languages
Javascript
C++
C#

vagrant@vagrant:~/GeeksForGeeks$ 
```

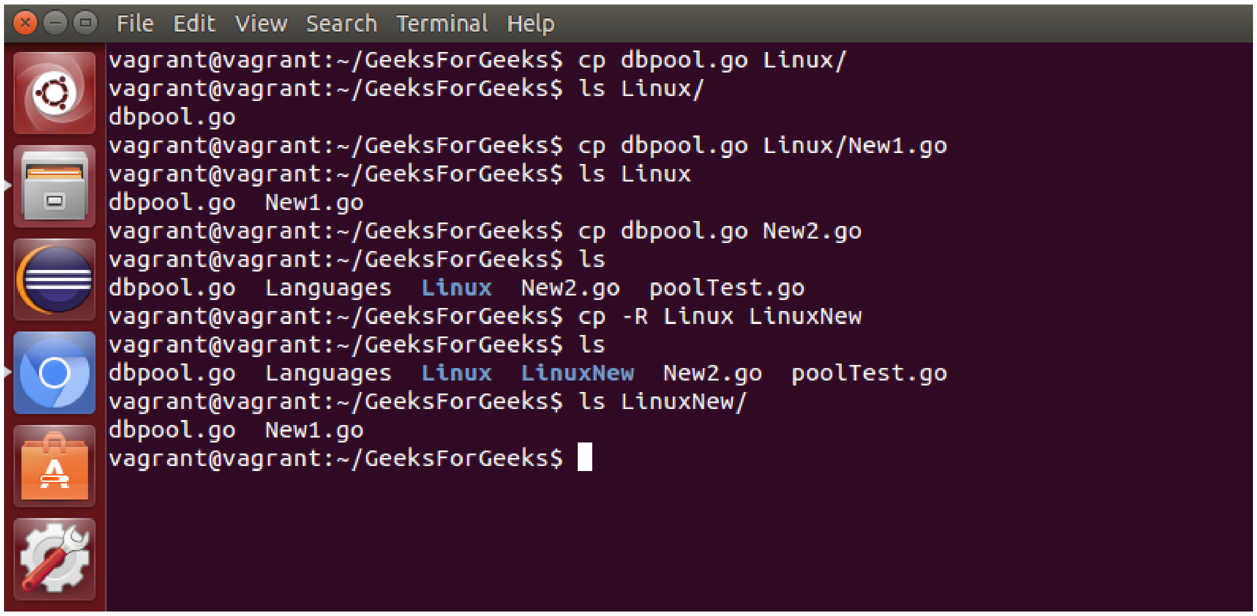
3). File and Directory Manipulation Commands:

- **mkdir** : Used to create a directory if not already exist. It accepts the directory name as an input parameter.



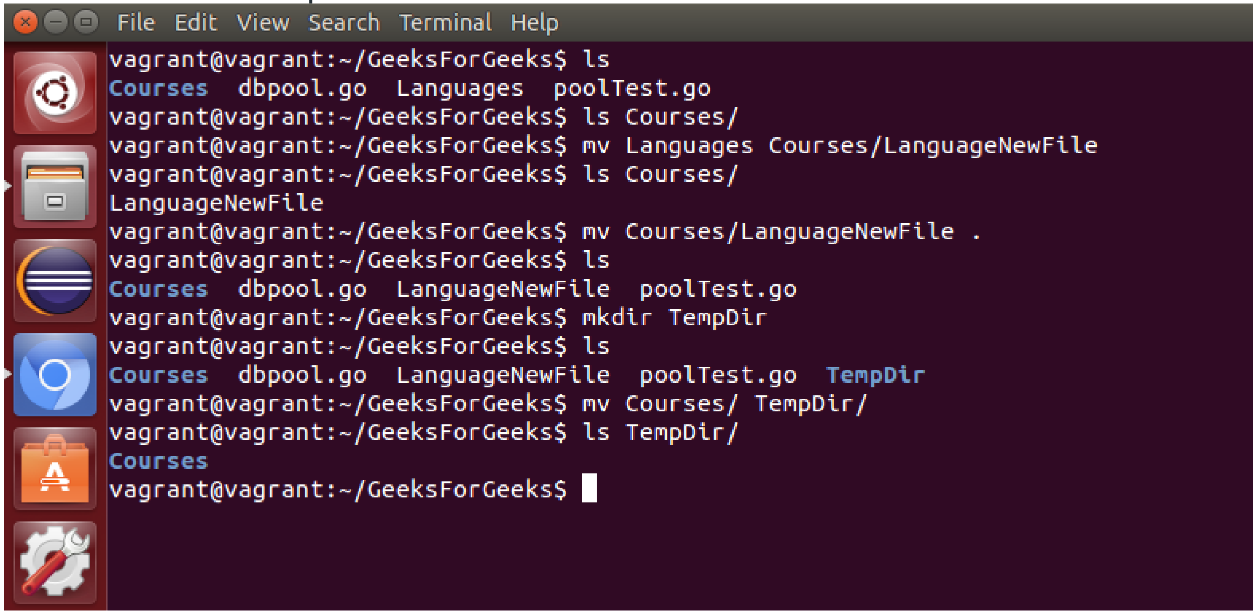
```
vagrant@vagrant:~/GeeksForGeeks$ ls
dbpool.go  Languages  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$ mkdir Linux
vagrant@vagrant:~/GeeksForGeeks$ ls
dbpool.go  Languages  Linux  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$ 
```

-



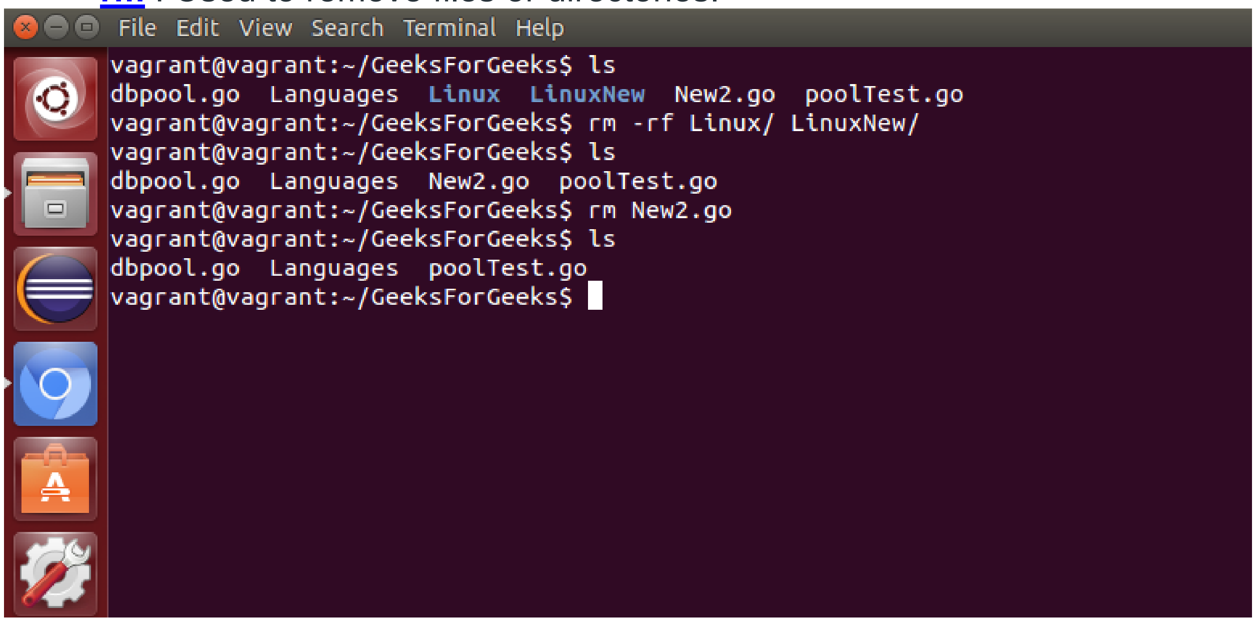
```
vagrant@vagrant:~/GeeksForGeeks$ cp dbpool.go Linux/
vagrant@vagrant:~/GeeksForGeeks$ ls Linux/
dbpool.go
vagrant@vagrant:~/GeeksForGeeks$ cp dbpool.go Linux/New1.go
vagrant@vagrant:~/GeeksForGeeks$ ls Linux
dbpool.go  New1.go
vagrant@vagrant:~/GeeksForGeeks$ cp dbpool.go New2.go
vagrant@vagrant:~/GeeksForGeeks$ ls
dbpool.go  Languages  Linux  New2.go  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$ cp -R Linux LinuxNew
vagrant@vagrant:~/GeeksForGeeks$ ls
dbpool.go  Languages  Linux  LinuxNew  New2.go  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$ ls LinuxNew/
dbpool.go  New1.go
vagrant@vagrant:~/GeeksForGeeks$ 
```

- **mv** : Used to move the files or directories. This command's working is almost similar to *cp* command but it deletes a copy of the file or directory from the source path.



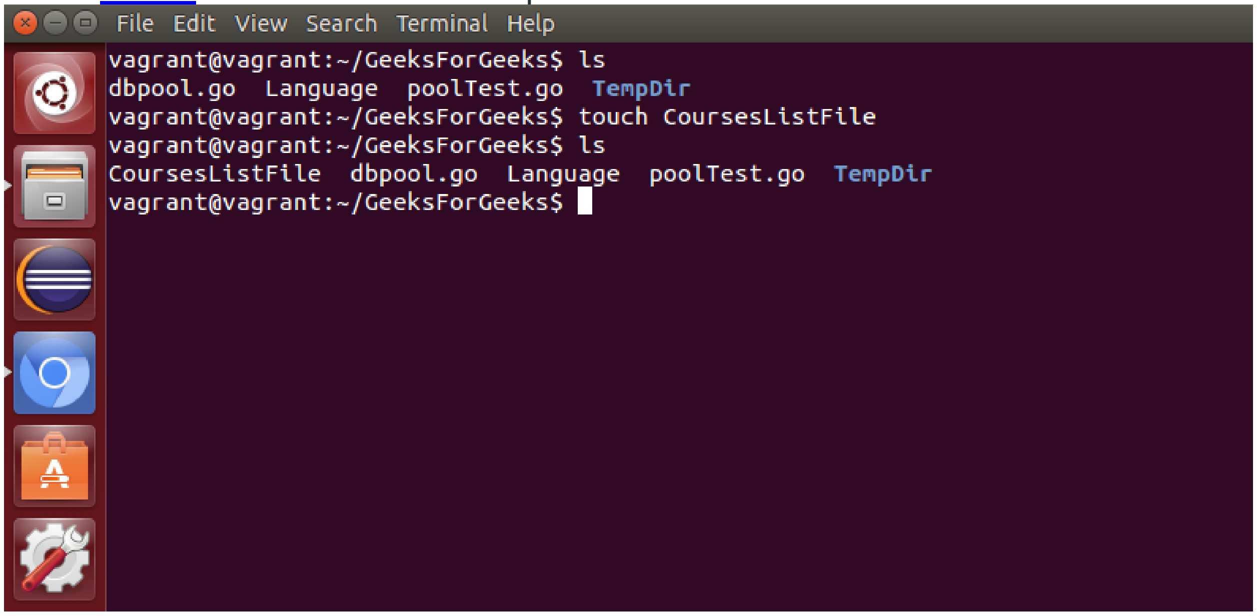
```
vagrant@vagrant:~/GeeksForGeeks$ ls
Courses  dbpool.go  Languages  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$ ls Courses/
vagrant@vagrant:~/GeeksForGeeks$ mv Languages Courses/LanguageNewFile
vagrant@vagrant:~/GeeksForGeeks$ ls Courses/
LanguageNewFile
vagrant@vagrant:~/GeeksForGeeks$ mv Courses/LanguageNewFile .
vagrant@vagrant:~/GeeksForGeeks$ ls
Courses  dbpool.go  LanguageNewFile  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$ mkdir TempDir
vagrant@vagrant:~/GeeksForGeeks$ ls
Courses  dbpool.go  LanguageNewFile  poolTest.go  TempDir
vagrant@vagrant:~/GeeksForGeeks$ mv Courses/ TempDir/
vagrant@vagrant:~/GeeksForGeeks$ ls TempDir/
Courses
vagrant@vagrant:~/GeeksForGeeks$ 
```

- **rm** : Used to remove files or directories.



```
vagrant@vagrant:~/GeeksForGeeks$ ls
dbpool.go  Languages  Linux  LinuxNew  New2.go  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$ rm -rf Linux/ LinuxNew/
vagrant@vagrant:~/GeeksForGeeks$ ls
dbpool.go  Languages  New2.go  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$ rm New2.go
vagrant@vagrant:~/GeeksForGeeks$ ls
dbpool.go  Languages  poolTest.go
vagrant@vagrant:~/GeeksForGeeks$ 
```

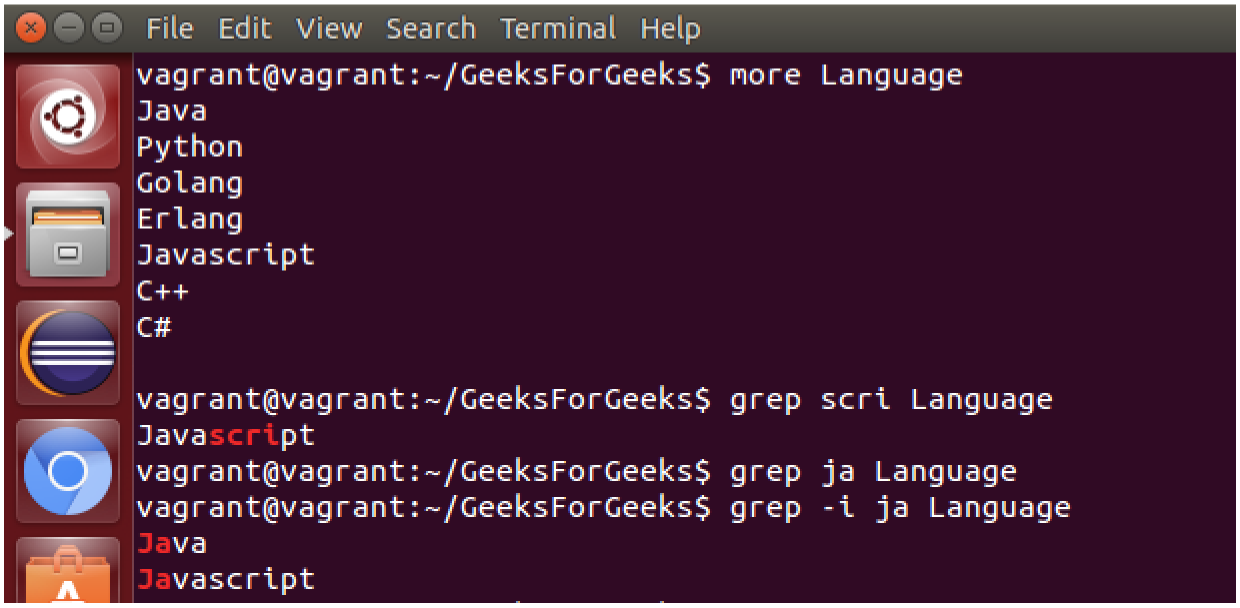
- **touch** : Used to create or update a file.



```
vagrant@vagrant:~/GeeksForGeeks$ ls
dbpool.go  Language  poolTest.go  TempDir
vagrant@vagrant:~/GeeksForGeeks$ touch CoursesListFile
vagrant@vagrant:~/GeeksForGeeks$ ls
CoursesListFile  dbpool.go  Language  poolTest.go  TempDir
vagrant@vagrant:~/GeeksForGeeks$
```

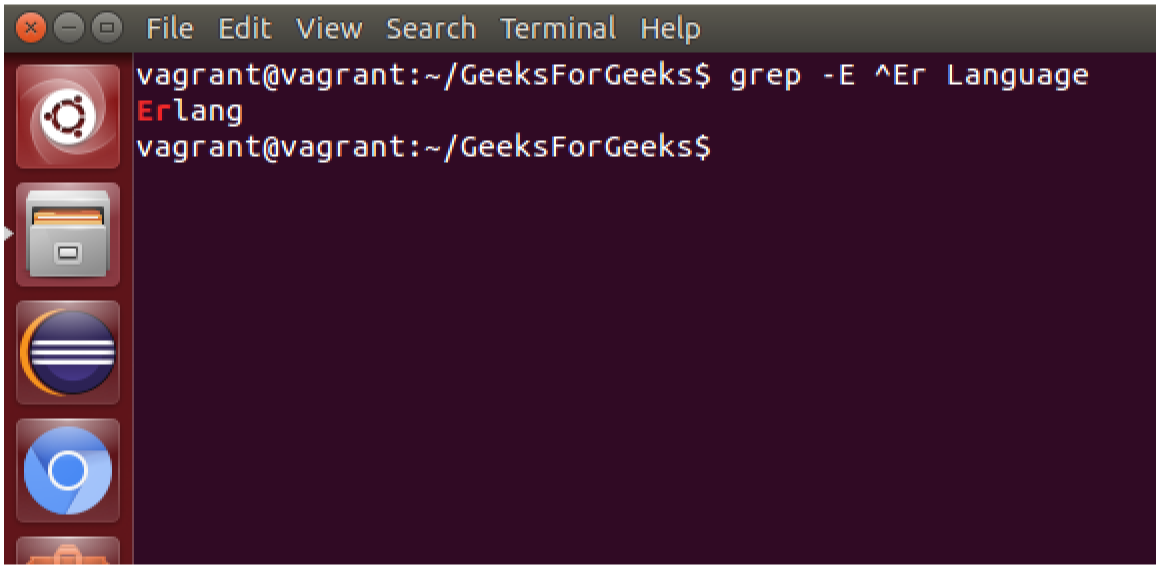
3). Extract, sort, and filter data Commands:

- **grep** : This command is used to search for the specified text in a file.



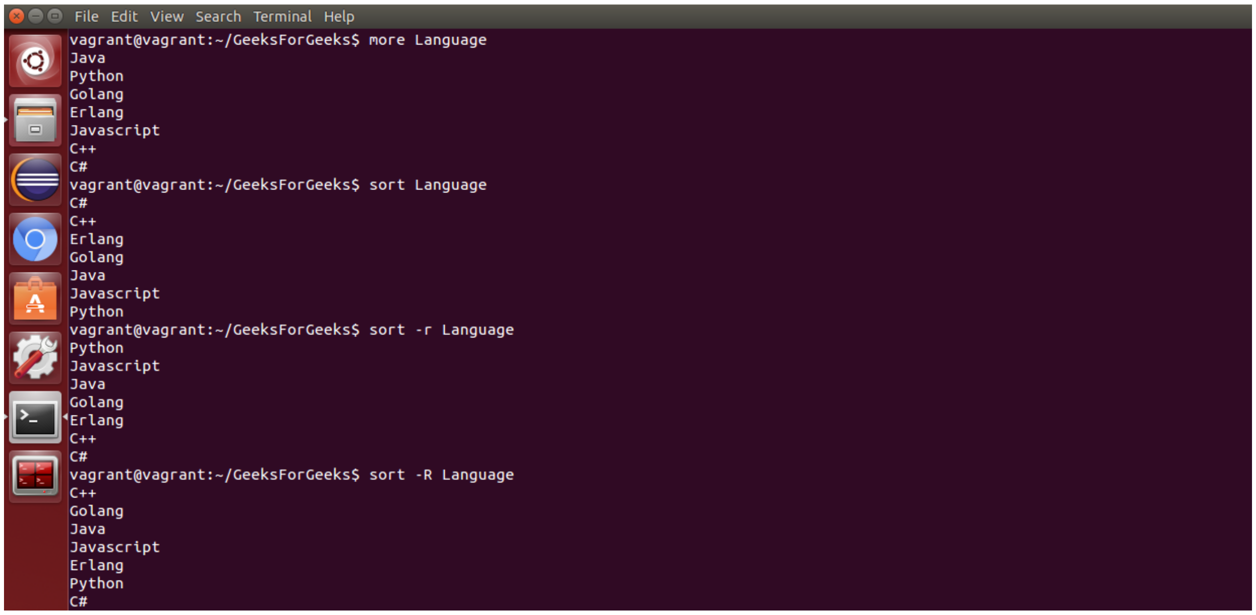
```
vagrant@vagrant:~/GeeksForGeeks$ more Language
Java
Python
Golang
Erlang
Javascript
C++
C#

vagrant@vagrant:~/GeeksForGeeks$ grep scri Language
Javascript
vagrant@vagrant:~/GeeksForGeeks$ grep ja Language
vagrant@vagrant:~/GeeksForGeeks$ grep -i ja Language
Java
Javascript
```

- **grep with Regular Expressions**: Used to search for text using specific regular expressions in file.



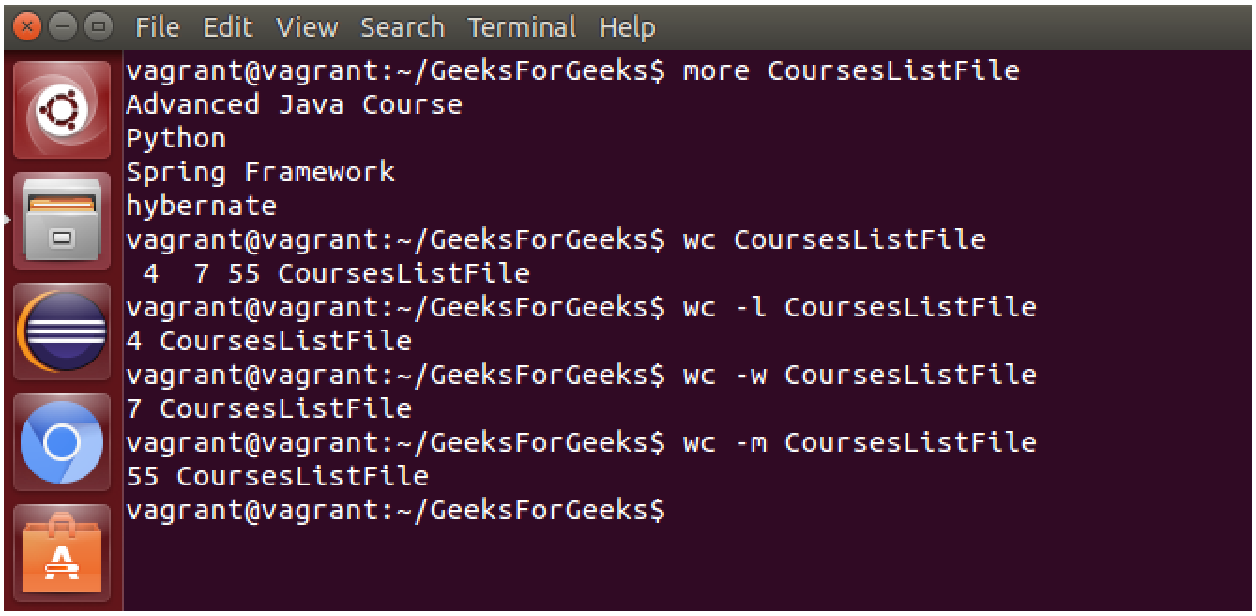
```
vagrant@vagrant:~/GeeksForGeeks$ grep -E ^Er Language
Erlang
vagrant@vagrant:~/GeeksForGeeks$
```

- **sort** : This command is used to sort the contents of files.



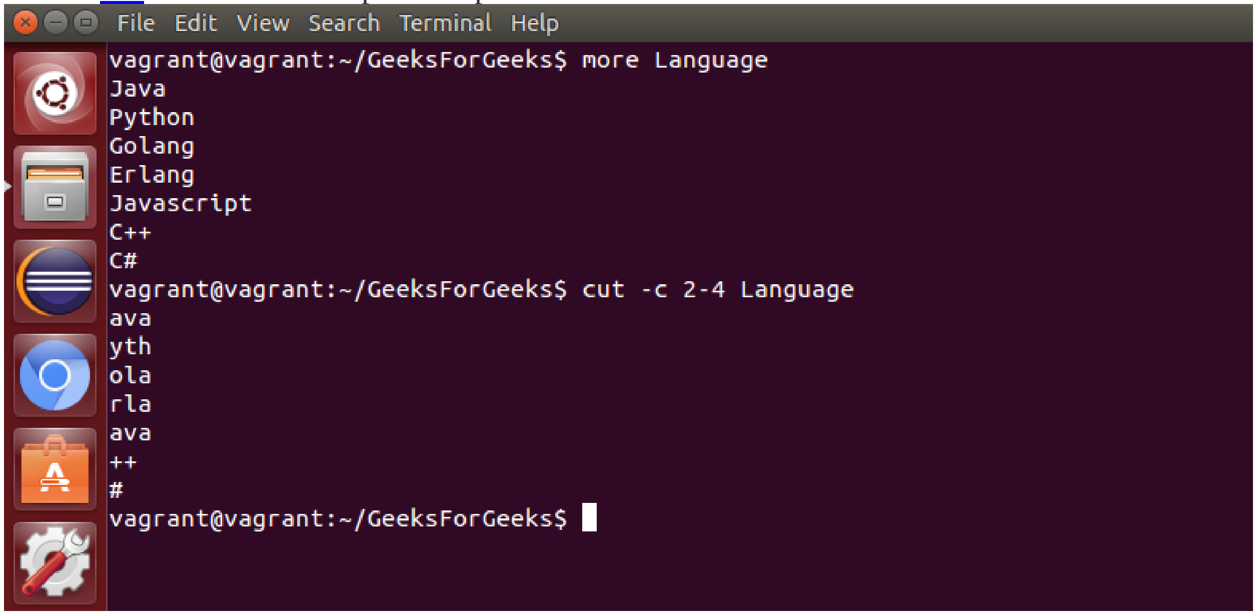
```
vagrant@vagrant:~/GeeksForGeeks$ more Language
Java
Python
Golang
Erlang
Javascript
C++
C#
vagrant@vagrant:~/GeeksForGeeks$ sort Language
C#
C++
Erlang
Golang
Java
Javascript
Python
vagrant@vagrant:~/GeeksForGeeks$ sort -r Language
Python
Javascript
Java
Golang
Erlang
C++
C#
vagrant@vagrant:~/GeeksForGeeks$ sort -R Language
C++
Golang
Java
Javascript
Erlang
Python
C#
```

- **wc** : Used to count the number of characters, words in a file.



```
vagrant@vagrant:~/GeeksForGeeks$ more CoursesListFile
Advanced Java Course
Python
Spring Framework
hybernate
vagrant@vagrant:~/GeeksForGeeks$ wc CoursesListFile
 4  7 55 CoursesListFile
vagrant@vagrant:~/GeeksForGeeks$ wc -l CoursesListFile
4 CoursesListFile
vagrant@vagrant:~/GeeksForGeeks$ wc -w CoursesListFile
7 CoursesListFile
vagrant@vagrant:~/GeeksForGeeks$ wc -m CoursesListFile
55 CoursesListFile
vagrant@vagrant:~/GeeksForGeeks$
```

- **cut** : Used to cut a specified part of a file.



```
vagrant@vagrant:~/GeeksForGeeks$ more Language
Java
Python
Golang
Erlang
Javascript
C++
C#
vagrant@vagrant:~/GeeksForGeeks$ cut -c 2-4 Language
ava
yth
ola
rla
ava
++
#
vagrant@vagrant:~/GeeksForGeeks$
```

5). File Permissions Commands: The *chmod* and *chown* commands are used to control access to files in UNIX and Linux systems.

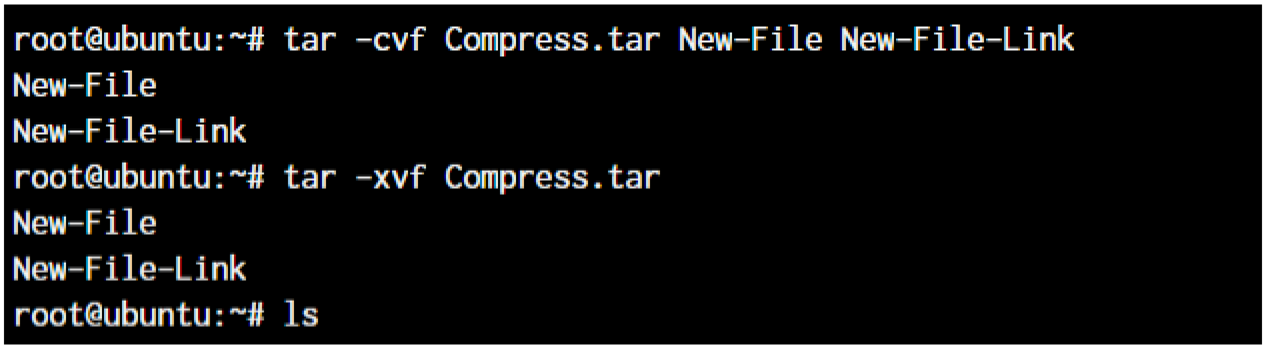
- **chown** : Used to change the owner of the file.
- **chgrp** : Used to change the group owner of the file.
- **chmod** : Used to modify the access/permission of a user.

The tar, zip, and unzip commands

The tar command in Linux is used to create and extract archived files in Linux. We can extract multiple different archive files using the tar command.

To create an archive, we use the -c parameter and to extract an archive, we use the -x parameter. Let's see it working.

```
#Compress
root@ubuntu:~# tar -cvf <archive name> <files seperated by space>
#Extract
root@ubuntu:~# tar -xvf <archive name>
```
Copy

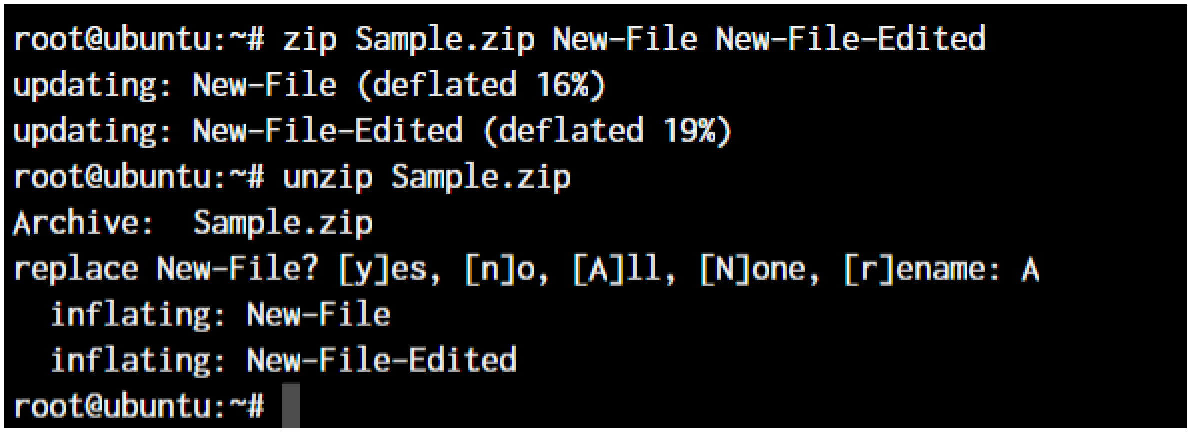


```
root@ubuntu:~# tar -cvf Compress.tar New-File New-File-Link
New-File
New-File-Link
root@ubuntu:~# tar -xvf Compress.tar
New-File
New-File-Link
root@ubuntu:~# ls
```

In the first line, we created an archive named **Compress.tar** with the New-File and New-File-Link. In the next command, we have extracted those files from the archive.

Now coming to the zip and unzip commands. Both these commands are very straight forward. You can use them without any parameters and they'll work as intended. Let's see an example below.

```
root@ubuntu:~# zip <archive name> <file names separated by space>
root@ubuntu:~# unzip <archive name>
```
Copy



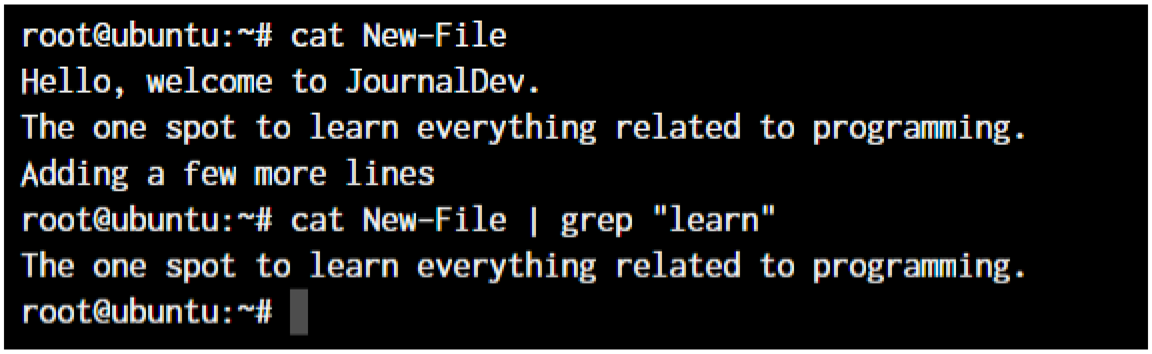
```
root@ubuntu:~# zip Sample.zip New-File New-File-Edited
updating: New-File (deflated 16%)
updating: New-File-Edited (deflated 19%)
root@ubuntu:~# unzip Sample.zip
Archive:  Sample.zip
replace New-File? [y]es, [n]o, [A]ll, [N]one, [r]ename: A
  inflating: New-File
  inflating: New-File-Edited
root@ubuntu:~#
```

Since we already have those files in the same directory, the unzip command prompts us before overwriting those files.

The grep command in Linux

If you wish to search for a specific string within an output, the grep command comes into the picture. We can pipe (|) the output to the grep command and extract the required string.

```
root@ubuntu:~# <Any command with output> | grep "<string to find>"
```
Copy



```
root@ubuntu:~# cat New-File
Hello, welcome to JournalDev.
The one spot to learn everything related to programming.
Adding a few more lines
root@ubuntu:~# cat New-File | grep "learn"
The one spot to learn everything related to programming.
root@ubuntu:~#
```

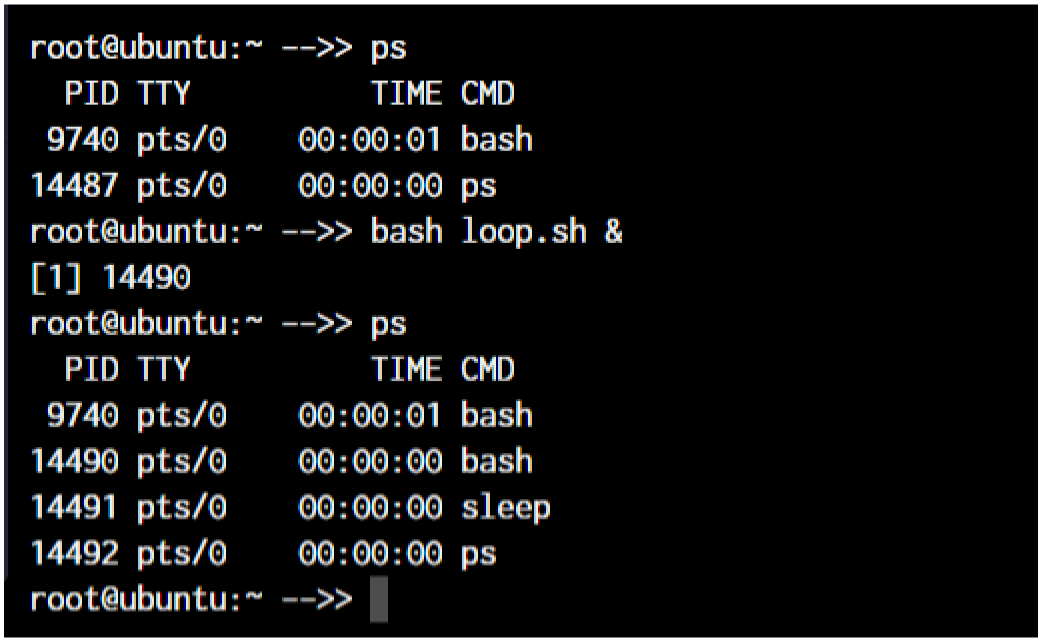
The ps, kill, and killall commands

To find the running processes, we can simply type **ps** in the terminal prompt and get the list of running processes.

```
root@ubuntu:~ -->> ps
root@ubuntu:~ -->> kill <process ID>
root@ubuntu:~ -->> killall <process name>
```
Copy

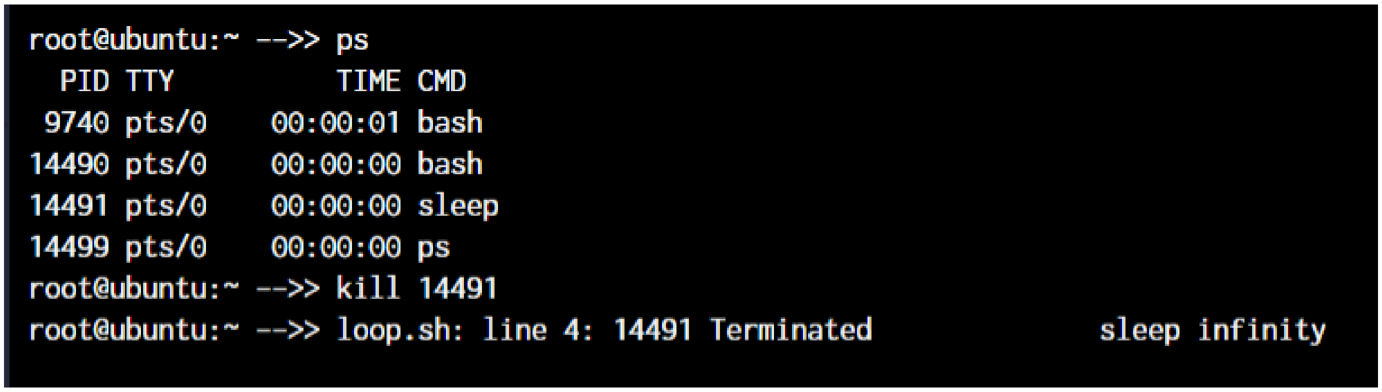
For demonstration purposes, I’m creating a shell script with an infinite loop and will run it in the background.

With the use of the **&** symbol, I can pass a process into the background. As you can see, a new bash process with PID 14490 is created.



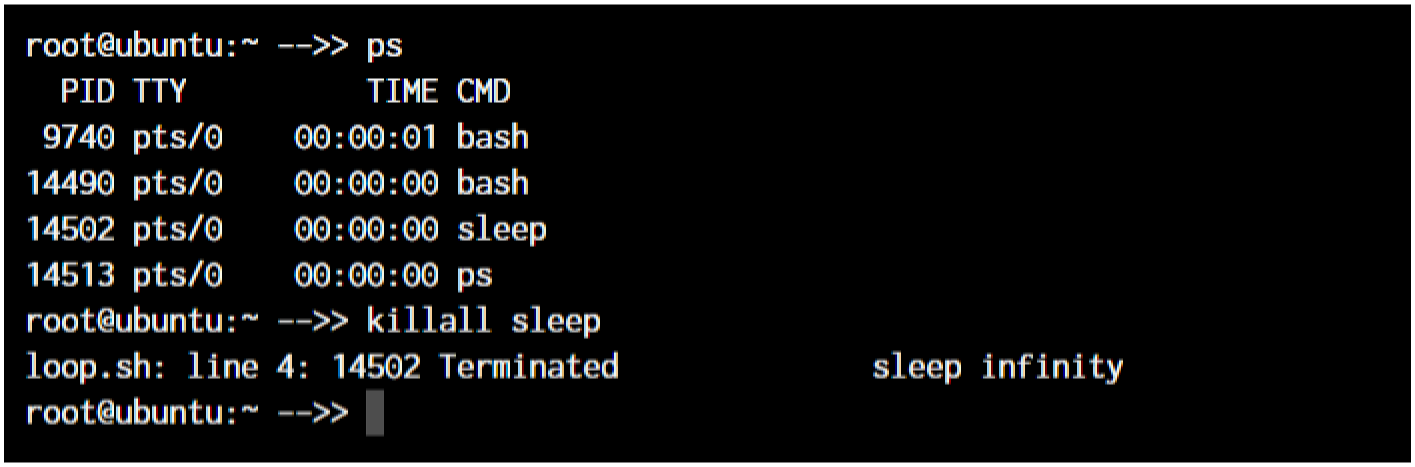
```
root@ubuntu:~ -->> ps
  PID TTY          TIME CMD
 9740 pts/0    00:00:01 bash
14487 pts/0    00:00:00 ps
root@ubuntu:~ -->> bash loop.sh &
[1] 14490
root@ubuntu:~ -->> ps
  PID TTY          TIME CMD
 9740 pts/0    00:00:01 bash
14490 pts/0    00:00:00 bash
14491 pts/0    00:00:00 sleep
14492 pts/0    00:00:00 ps
root@ubuntu:~ -->>
```

Now, to kill a process with the **kill** command, you can type **kill** followed b the PID of the process.



```
root@ubuntu:~ -->> ps
  PID TTY          TIME CMD
 9740 pts/0    00:00:01 bash
14490 pts/0    00:00:00 bash
14491 pts/0    00:00:00 sleep
14499 pts/0    00:00:00 ps
root@ubuntu:~ -->> kill 14491
root@ubuntu:~ -->> loop.sh: line 4: 14491 Terminated              sleep infinity
```

But if you do not know the process ID and just want to kill the process with the name, you can make use of the killall command.



```
root@ubuntu:~ -->> ps
  PID TTY          TIME CMD
 9740 pts/0    00:00:01 bash
14490 pts/0    00:00:00 bash
14502 pts/0    00:00:00 sleep
14513 pts/0    00:00:00 ps
root@ubuntu:~ -->> killall sleep
loop.sh: line 4: 14502 Terminated              sleep infinity
root@ubuntu:~ -->>
```

You will notice that PID 14490 stayed active. That is because both the times, I killed the sleep process.

The df and mount commands

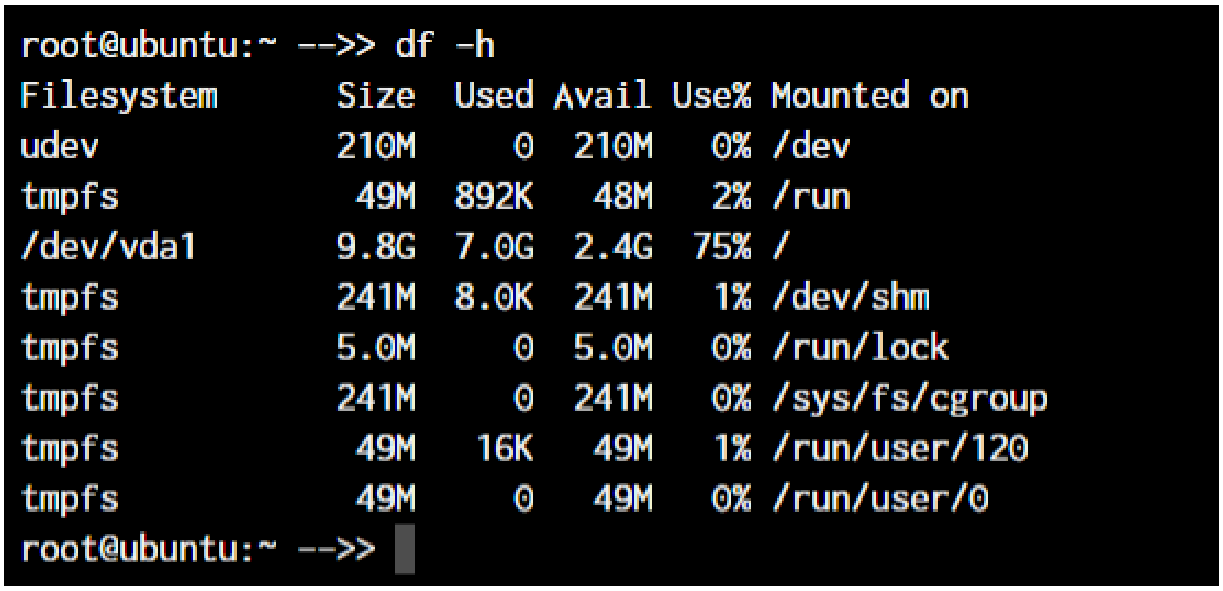
When working with Linux, the df and mount commands are very efficient utilities to mount filesystems and get details of the file system.

When I say mount, it means that we'll connect the device to a folder so we can access the files from our filesystem. The default syntax to mount a filesystem is below:

```
root@ubuntu:~ -->> mount /dev/cdrom /mnt
root@ubuntu:~ -->> df -h
```
Copy

In the above case, **/dev/cdrom** is the device that needs to be mounted. Usually, a mountable device is found inside the /dev folder. **/mnt** is the destination folder to mount the device to. You can change it to any folder you want but I've used /mnt as it's pretty much a system default folder for mounting devices.

To see the mounted devices and get more information about them, we make use of the df command. Just typing **df** will give us the data in bytes which is not readable. So we'll use the **-h** parameter to make the data human-readable.



```
root@ubuntu:~ -->> df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            210M     0  210M   0% /dev
tmpfs            49M  892K   48M   2% /run
/dev/vda1       9.8G  7.0G  2.4G  75% /
tmpfs           241M  8.0K  241M   1% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           241M     0  241M   0% /sys/fs/cgroup
tmpfs            49M   16K   49M   1% /run/user/120
tmpfs            49M     0   49M   0% /run/user/0
root@ubuntu:~ -->>
```

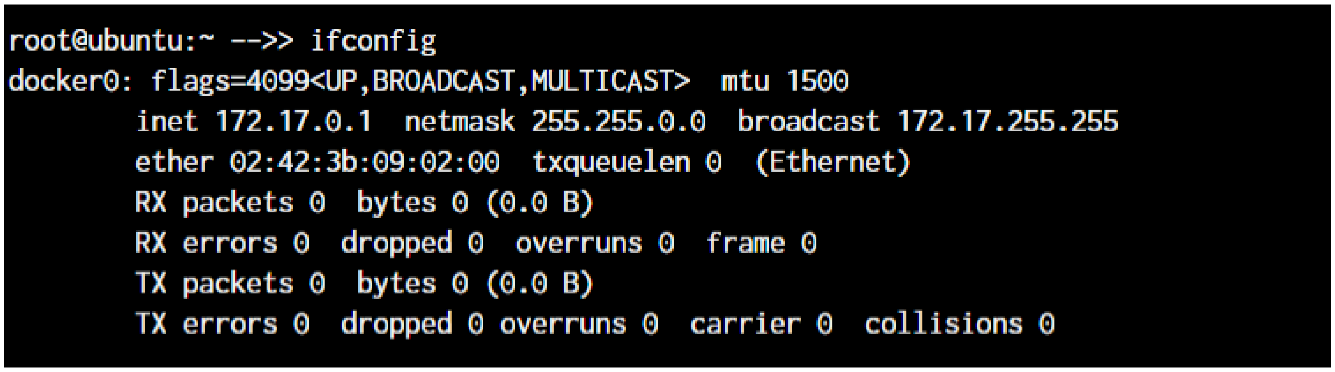
The ifconfig and traceroute commands

Moving on to the networking section in Linux, we come across the ifconfig and traceroute commands which will be frequently used if you manage a network.

The ifconfig command will give you the list of all the network interfaces along with the IP addresses, MAC addresses and other information about the interface.

```
root@ubuntu:~ -->> ifconfig
```
Copy

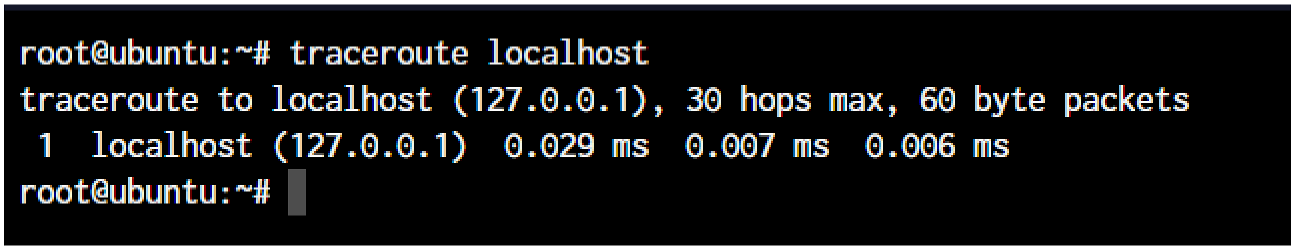
There are multiple parameters that can be used but we'll work with the basic command here.



```
root@ubuntu:~ -->> ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:3b:09:02:00  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

When working with traceroute, you can simply specify the IP address, the hostname or the domain name of the endpoint.

```
root@ubuntu:~ -->> traceroute <destination address>
```



```
root@ubuntu:~# traceroute localhost
traceroute to localhost (127.0.0.1), 30 hops max, 60 byte packets
 1  localhost (127.0.0.1)  0.029 ms  0.007 ms  0.006 ms
root@ubuntu:~#
```

Now obviously, localhost is just one hop (which is the network interface itself). You can try this same command with any other domain name or IP address to see all the routers that your data packets pass through to reach the destination.

The wget command in Linux

If you want to download a file from within the terminal, the wget command is one of the handiest command-line utilities available. This will be one of the important Linux commands you should know when working with source files.

When you specify the link for download, it has to directly be a link to the file. If the file cannot be accessed by the wget command, it will simply download the webpage in HTML format instead of the actual file that you wanted.

Let's try an example. The basic syntax of the wget command is :

```
root@ubuntu:~ -->> wget <link to file>
OR
root@ubuntu:~ -->> wget -c <link to file>
```

Copy

The **-c** argument allows us to resume an interrupted download.

The ufw and iptables commands

UFW and IPTables are firewall interfaces for the Linux Kernel's netfilter firewall. IPTables directly passes firewall rules to netfilter while UFW configures the rules in IPTables which then sends those rules to netfilter.

Why do we need UFW when we have IPTables? Because IPTables is pretty difficult for a newbie. UFW makes things extremely easy. See the below example where we are trying to allow the port 80 for our webserver.

```
root@ubuntu:~# iptables -A INPUT -p tcp -m tcp --dport 80 -j ACCEPT
root@ubuntu:~# ufw allow 80
```

Copy

I'm sure you now know why UFW was created! Look at how easy the syntax becomes. Both these firewalls are very comprehensive and can allow you to create any kind of configuration required for your network. Learn at least the basics of UFW or IPTables firewall as these are the Linux commands you must know.

Package Managers in Linux

Different distros of Linux make use of different package managers. Since we're working on a Ubuntu server, we have the **apt package manager**. But for someone working on a Fedora, Red Hat, Arch, or Centos machine, the package manager will be different.

- **Debian and Debian-based distros** - apt install <package name>
- **Arch and Arch-based distros** - pacman -S <package name>
- **Red Hat and Red Hat-based distros** - yum install <package name>
- **Fedora and CentOS** - yum install <package>

Getting yourself well versed with the package manager of your distribution will make things much easier for you in the long run. So even if you have a GUI based package management tool installed, try an make use of the CLI based tool before you move on to the GUI utility. Add these to your list of Linux commands you must know.

The sudo command in Linux

"With great power, comes great responsibility"

This is the quote that's displayed when a sudo enabled user(sudoer) first makes use of the sudo command to escalate privileges. This command is equivalent to having logged in as root (based on what permissions you have as a sudoer).

```
non-root-user@ubuntu:~# sudo <command you want to run>
Password:
```

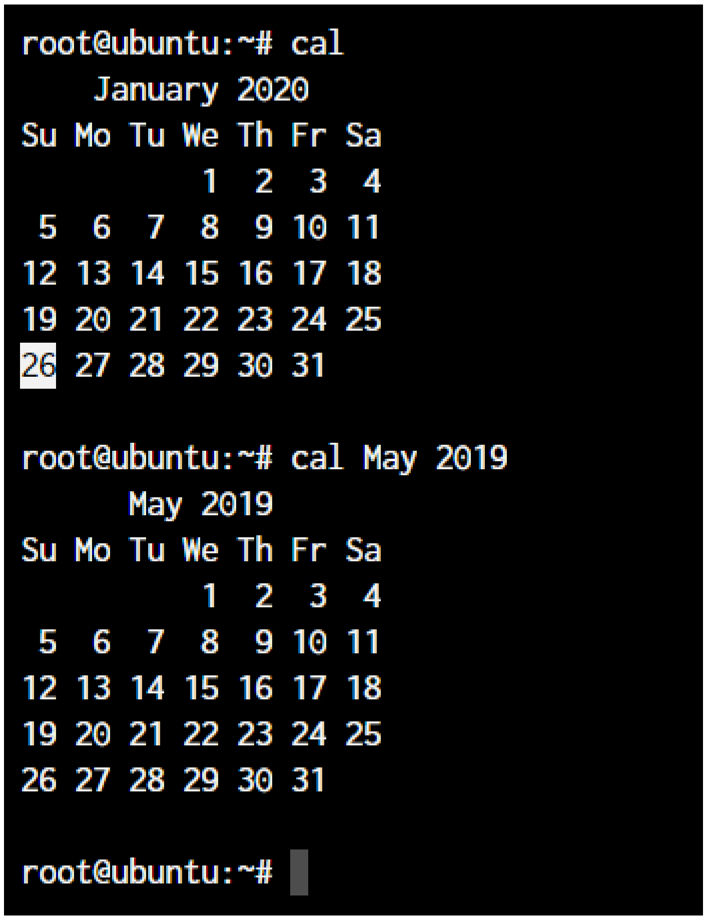
Just add the word **sudo** before any command that you need to run with escalated privileges and that's it. It's very simple to use, but can also be an added security risk if a malicious user gains access to a sudoer.

The cal command in Linux

Ever wanted to view the calendar in the terminal? Me neither! But there apparently are people who wanted it to happen and well here it is.

The **cal** command displays a well-presented calendar on the terminal. Just enter the word cal on your terminal prompt.

```
root@ubuntu:~# cal
root@ubuntu:~# cal May 2019
```



```
root@ubuntu:~# cal
    January 2020
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

root@ubuntu:~# cal May 2019
      May 2019
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

root@ubuntu:~#
```

Even though I don't need it, it's a really cool addition! I'm sure there are people who are terminal fans and this is a really amazing option for them.

The alias command

Do you have some commands that you run very frequently while using the terminal? It could be **rm -r** or **ls -l**, or it could be something longer like **tar -xvzf**. This is one of the productivity-boosting Linux commands you must know.

If you know a command that you run very often, it's time to create an alias. What's an alias? In simple terms, it's another name for a command that you've defined.

```
root@ubuntu:~# alias lsl="ls -l"
OR
root@ubuntu:~# alias rmd="rm -r"
```

Now every time you enter **lsl** or **rmd** in the terminal, you'll receive the output that you'd have received if you had used the full commands.

The examples here are for really small commands that you can still type by hand every time. But in some situations where a command has too many arguments that you need to type, it's best to create a shorthand version of the same.

The dd command in Linux

This command was created to convert and copy files from multiple file system formats. In the current day, the command is simply used to create bootable USB for Linux but there still are some things important you can do with the command.

For example, if I wanted to back up the entire hard drive as is to another drive, I'll make use of the dd command.

```
root@ubuntu:~# dd if = /dev/sdb of = /dev/sda
```

The **if** and **of** arguments stand for **input file** and **output file**.

The whereis and whatis commands

The names of the commands make it very clear as to their functionality. But let's demonstrate their functionality to make things more clear.

The **whereis** command will output the exact location of any command that you type in after the **whereis** command.

```
root@ubuntu:~# whereis sudo
sudo: /usr/bin/sudo /usr/lib/sudo /usr/share/man/man8/sudo.8.gz
```

The **whatis** command gives us an explanation of what a command actually is. Similar to the whereis command, you'll receive the information for any command that you type after the **whatis** command.

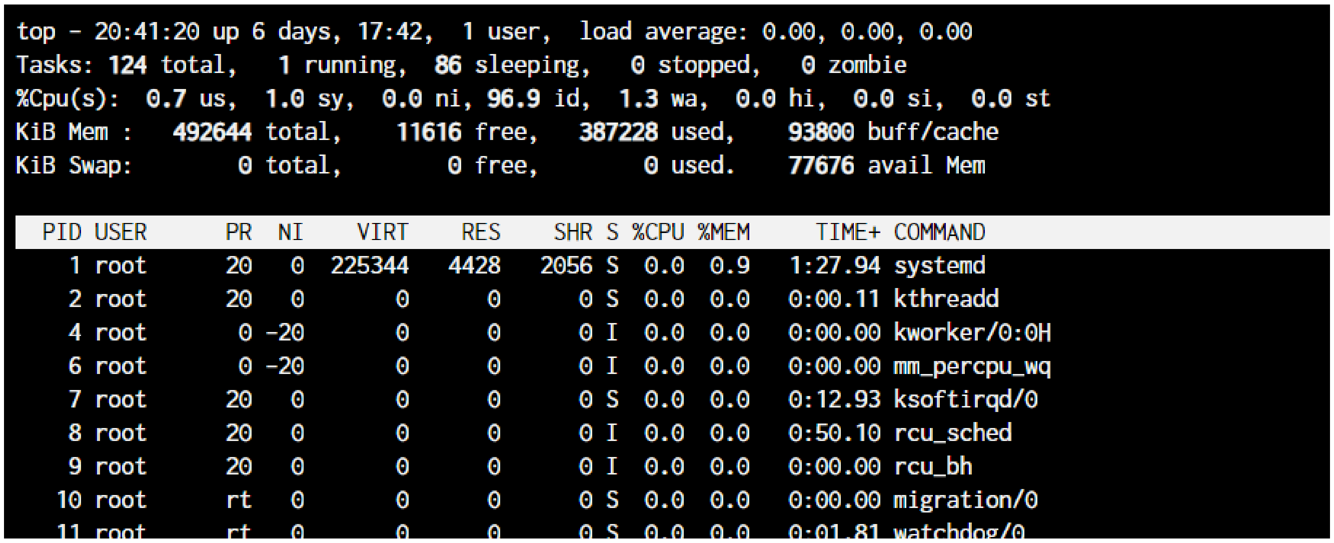
```
root@ubuntu:~# whatis sudo
sudo (8) - execute a command as another user
```

The top command in Linux

A few sections earlier, we talked about the ps command. You observed that the ps command will output the active processes and end itself.

The top command is like a CLI version of the task manager in Windows. You get a live view of the processes and all the information accompanying those processes like memory usage, CPU usage, etc.

To get the top command, all you need to do is type the word **top** in your terminal.



```
top - 20:41:20 up 6 days, 17:42,  1 user,  load average: 0.00, 0.00, 0.00
Tasks: 124 total,   1 running,  86 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.7 us,  1.0 sy,  0.0 ni, 96.9 id,  1.3 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :   492644 total,    11616 free,   387228 used,    93800 buff/cache
KiB Swap:        0 total,        0 free,        0 used.    77676 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
    1 root      20   0  225344   4428   2056 S  0.0  0.9   1:27.94 systemd
    2 root      20   0       0      0      0 S  0.0  0.0   0:00.11 kthreadd
    4 root       0 -20       0      0      0 I  0.0  0.0   0:00.00 kworker/0:0H
    6 root       0 -20       0      0      0 I  0.0  0.0   0:00.00 mm_percpu_wq
    7 root      20   0       0      0      0 S  0.0  0.0   0:12.93 ksoftirqd/0
    8 root      20   0       0      0      0 I  0.0  0.0   0:50.10 rcu_sched
    9 root      20   0       0      0      0 I  0.0  0.0   0:00.00 rcu_bh
   10 root      rt   0       0      0      0 S  0.0  0.0   0:00.00 migration/0
   11 root      rt   0       0      0      0 S  0.0  0.0   0:01.81 watchdog/0
```

The useradd and usermod commands

The **useradd** or **adduser** commands are the exact same commands where adduser is just a symbolic link to the useradd command. This command allows us to create a new user in Linux.

```
root@ubuntu:~# useradd JournalDev -d /home/JD
```

Copy

The above command will create a new user named JournalDev with the home directory as **/home/JD**.

The **usermod** command, on the other hand, is used to modify existing users. You can modify any value of the user including the groups, the permissions, etc.

For example, if you want to add more groups to the user, you can type in:

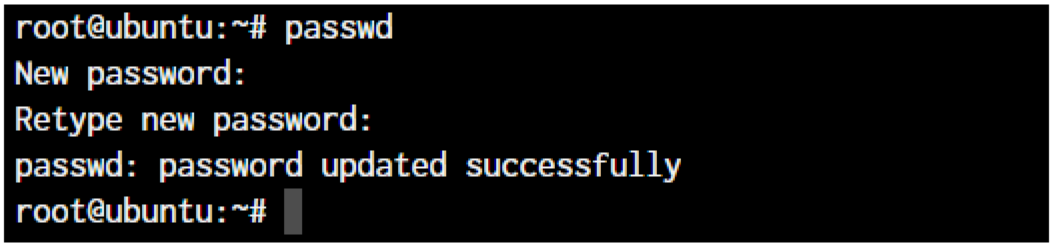
```
root@ubuntu:~# usermod JournalDev -a -G sudo, audio, mysql
```

The passwd command in Linux

Now that you know how to create new users, let's also set the password for them. The **passwd** command lets you set the password for your own account, or if you have the permissions, set the password for other accounts.

The command usage is pretty simple:

```
root@ubuntu:~# passwd
New password:
```



```
root@ubuntu:~# passwd
New password:
Retype new password:
passwd: password updated successfully
root@ubuntu:~#
```

If you add the username after **passwd**, you can set passwords for other users. Enter the new password twice and you’re done. That’s it! You will have a new password set for the user!

Assignment 3:

WAP to copy contents of one file into another in linux(in same directory and different directory).

cp command-

You use the cp command for copying files from one location to another. This command can also copy directories (folders).
The syntax of this command is:

```
cp [...file/directory-sources] [destination]
```

[file/directory-sources] specifies the sources of the files or directories you want to copy. And the [destination] argument specifies the location you want to copy the file to.
To understand the rest of this article, I will use this folder structure example. Let's say a directory called **DirectoryA** has two directories in it: **DirectoryA_1** and **DirectoryA_2**. These subdirectories have many files and sub directories in them.
I'll also assume you're currently in the **DirectoryA** location in the terminal, so if you aren't, make sure you are:

```
cd DirectoryA
```

How to copy files with the cp command?

If you want to copy a file, say **README.txt** from **DirectoryA_1** to **DirectoryA_2**, you will use the cp command like this:

```
cp ./DirectoryA_1/README.txt ./DirectoryA_2
# ./DirectoryA_1/README.txt is the source file
# ./DirectoryA_2 is the destination
```

If you want to copy more than a file from **DirectoryA_1** to **DirectoryA_2**, you will use the cp command like this:

```
cp ./DirectoryA_1/README.txt ./DirectoryA_1/ANOTHER_FILE.txt ./DirectoryA_2
```

As you can see, you will put all the source files first, and the last argument will be the destination.

Assignment 4:

Write a programme (using control statements)
a) To find smallest and greatest among all numbers.
b) To display factorial of given number.
c) To display Fibonacci series for given input.

Basic Operators in Shell Scripting

There are **5** basic operators in bash/shell scripting:

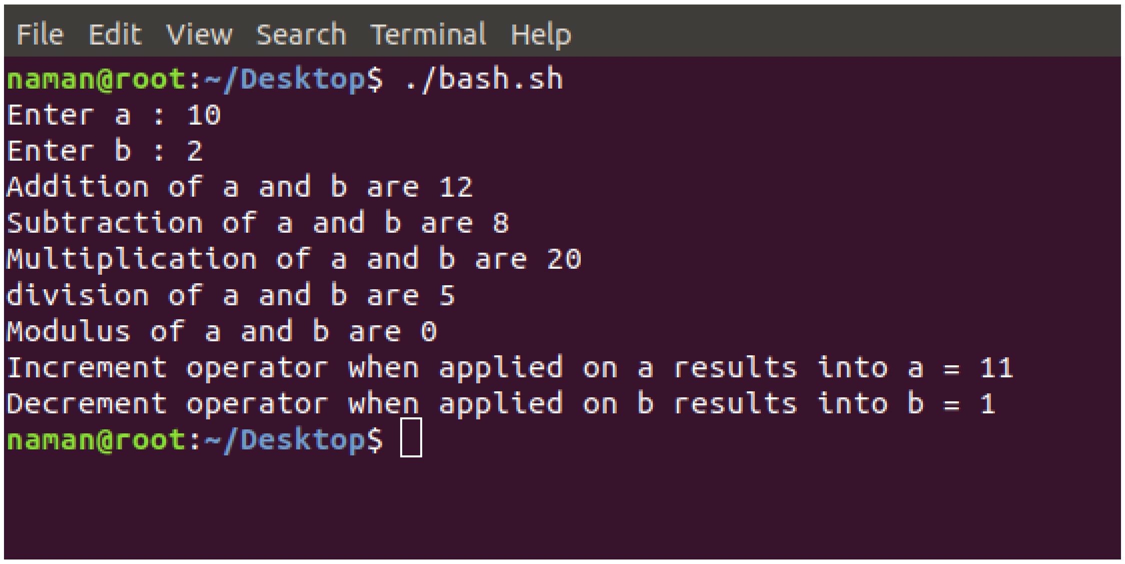
- Arithmetic Operators
- Relational Operators
- Boolean Operators
- Bitwise Operators
- File Test Operators

1. Arithmetic Operators: These operators are used to perform normal arithmetics/mathematical operations. There are 7 arithmetic operators:

- **Addition (+)**: Binary operation used to add two operands.
- **Subtraction (-)**: Binary operation used to subtract two operands.
- **Multiplication (*)**: Binary operation used to multiply two operands.
- **Division (/)**: Binary operation used to divide two operands.
- **Modulus (%)**: Binary operation used to find remainder of two operands.
- **Increment Operator (++)**: Unary operator used to increase the value of operand by one.
- **Decrement Operator (- -)**: Unary operator used to decrease the value of a operand by one

```
#reading data from the user
read - p 'Enter a : ' a
     read
  - p 'Enter b : ' b

      add = $((a + b))
    echo Addition of a and b are $add
       sub  = $((a - b))
    echo Subtraction of a and b are $sub
       mul = $((a * b))
    echo Multiplication of a and b are $mul
       div = $((a / b))
    echo division of a and b are $div
       mod = $((a % b))
      echo Modulus of a  and b are $mod
    ((++a))
      echo Increment
      operator when applied on "a" results into a = $a
    ((--b))
      echo Decrement
      operator when applied on "b" results into b = $b
```

Output:



```
naman@root:~/Desktop$ ./bash.sh
Enter a : 10
Enter b : 2
Addition of a and b are 12
Subtraction of a and b are 8
Multiplication of a and b are 20
division of a and b are 5
Modulus of a and b are 0
Increment operator when applied on a results into a = 11
Decrement operator when applied on b results into b = 1
naman@root:~/Desktop$
```

2. Relational Operators: Relational operators are those operators which define the relation between two operands. They give either true or false depending upon the relation. They are of 6 types:

- **'==' Operator**: Double equal to operator compares the two operands. Its returns true is they are equal otherwise returns false.
- **'!=' Operator**: Not Equal to operator return true if the two operands are not equal otherwise it returns false.
- **'<' Operator**: Less than operator returns true if first operand is less than second operand otherwise returns false.
- **'<=' Operator**: Less than or equal to operator returns true if first operand is less than or equal to second operand otherwise returns false
- **'>' Operator**: Greater than operator return true if the first operand is greater than the second operand otherwise return false.
- **'>=' Operator**: Greater than or equal to operator returns true if first operand is greater than or equal to second operand otherwise returns false.

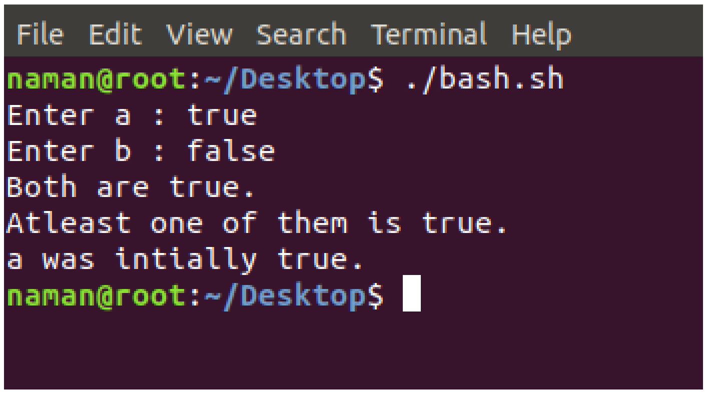
WAP to compare two numbers

```
read -p 'Enter a : ' a
read -p 'Enter b : ' b

if(( $a==$b ))
then
    echo a is equal to b.
else
    echo a is not equal to b.
fi

if(( $a!=$b ))
then
    echo a is not equal to b.
else
    echo a is equal to b.
fi

if(( $a<$b ))
then
    echo a is less than b.
else
    echo a is not less than b.
fi

if(( $a<=$b ))
then
    echo a is less than or equal to b.
else
    echo a is not less than or equal to b.
fi

if(( $a>$b ))
then
    echo a is greater than b.
else
    echo a is not greater than b.
fi

if(( $a>=$b ))
then
    echo a is greater than or equal to b.
else
    echo a is not greater than or equal to b.
fi
```

3. Logical Operators : They are also known as boolean operators. These are used to perform logical operations. They are of 3 types:

- **Logical AND (&&)**: This is a binary operator, which returns true if both the operands are true otherwise returns false.
- **Logical OR (||)**: This is a binary operator, which returns true is either of the operand is true or both the operands are true and return false if none of then is false.
- **Not Equal to (!)**: This is a unary operator which returns true if the operand is false and returns false if the operand is true.

```
read -p 'Enter a : ' a
read -p 'Enter b : ' b

if(($a == "true" & $b == "true" ))
then
    echo Both are true.
else
    echo Both are not true.
fi

if(($a == "true" || $b == "true" ))
then
    echo Atleast one of them is true.
else
    echo None of them is true.
fi

if(( ! $a == "true"  ))
then
    echo "a" was initially false.
else
     echo "a" was initially true.
 fi
```



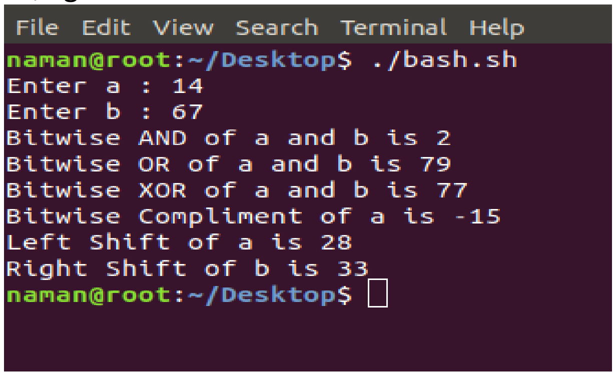
```
naman@root:~/Desktop$ ./bash.sh
Enter a : true
Enter b : false
Both are true.
Atleast one of them is true.
a was intially true.
naman@root:~/Desktop$ 
```

4. Bitwise Operators: A bitwise operator is an operator used to perform bitwise operations on bit patterns. They are of 6 types:

- **Bitwise And (&)**: Bitwise & operator performs binary AND operation bit by bit on the operands.
- **Bitwise OR (|)**: Bitwise | operator performs binary OR operation bit by bit on the operands.
- **Bitwise XOR (^)**: Bitwise ^ operator performs binary XOR operation bit by bit on the operands.
- **Bitwise complement (~)**: Bitwise ~ operator performs binary NOT operation bit by bit on the operand.
- **Left Shift (<<)**: This operator shifts the bits of the left operand to left by number of times specified by right operand.
- **Right Shift (>>)**: This operator shifts the bits of the left operand to right by number of times specified by right operand.

```
read -p 'Enter a : '  a
read -p 'Enter b : '  b

bitwiseAND=$(( a&b ))
echo Bitwise AND of a and b is $bitwiseAND

bitwiseOR=$(( a|b ))
echo Bitwise OR of a and b is $bitwiseOR

bitwiseXOR=$(( a^b ))
echo Bitwise XOR of a and b is $bitwiseXOR

bitiwiseComplement=$(( ~a ))
echo Bitwise Compliment of a is $bitiwiseComplement

leftshift=$(( a<<1 ))
echo Left Shift of a is $leftshift

rightshift=$(( b>>1 ))
echo Right Shift of b is $rightshift
```

- **Output:**



```
naman@root:~/Desktop$ ./bash.sh
Enter a : 14
Enter b : 67
Bitwise AND of a and b is 2
Bitwise OR of a and b is 79
Bitwise XOR of a and b is 77
Bitwise Compliment of a is -15
Left Shift of a is 28
Right Shift of b is 33
naman@root:~/Desktop$
```

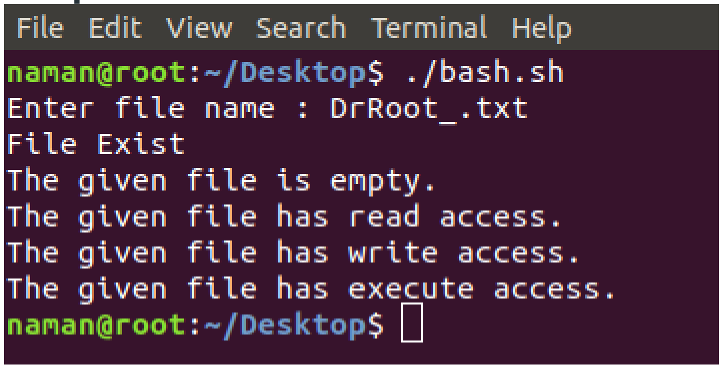
5. File Test Operator: These operators are used to test a particular property of a file.

- **-b operator**: This operator check whether a file is a block special file or not. It returns true if the file is a block special file otherwise false.
- **-c operator**: This operator checks whether a file is a character special file or not. It returns true if it is a character special file otherwise false.
- **-d operator**: This operator checks if the given directory exists or not. If it exists then operators returns true otherwise false.
- **-e operator**: This operator checks whether the given file exists or not. If it exits this operator returns true otherwise false.
- **-r operator**: This operator checks whether the given file has read access or not. If it has read access then it returns true otherwise false.
- **-w operator**: This operator check whether the given file has write access or not. If it has write then it returns true otherwise false.
- **-x operator**: This operator check whether the given file has execute access or not. If it has execute access then it returns true otherwise false.
- **-s operator**: This operator checks the size of the given file. If the size of given file is greater than 0 then it returns true otherwise it is false.

```
read -p 'Enter file name : '   FileName

if  [ -e $FileName ]
then
   echo File Exist
else
   echo File doesnot exist
fi

if  [ -s $FileName ]
then
   echo The given file is not empty.
else
   echo The given file is empty.
fi

if  [ -r $FileName ]
then
   echo The given file has read access.
else
   echo The given file does not has read access.
fi

if  [ -w $FileName ]
then
   echo The given file has write access.
```

```
else
   echo The given file does not has write access.
fi

if  [ -x $FileName ]
then
   echo The given file has execute access.
else
   echo The given file does not has execute access.
fi
```

Output:



```
naman@root:~/Desktop$ ./bash.sh
Enter file name : DrRoot_.txt
File Exist
The given file is empty.
The given file has read access.
The given file has write access.
The given file has execute access.
naman@root:~/Desktop$
```

Looping Statements in Shell Scripting: There are total 3 looping statements which can be used in bash programming

1. while statement
2. for statement
3. until statement

To alter the flow of loop statements, two commands are used they are,
break

1. continue

Their descriptions and syntax are as follows:

while statement

Here command is evaluated and based on the result loop will executed, if command raise to false then loop will be terminated

for statement

The for loop operate on lists of items. It repeats a set of commands for every item in a list. Here var is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

until statement

The until loop is executed as many as times the condition/command evaluates to false. The loop terminates when the condition/command becomes true.

Do loop

```
do
   echo  "infinite loops [ hit CTRL+C to stop]"
done
```

Display numbers from 1 to 10:

```
i=1
echo "Counting from 1 to 10: "
while (( $i <= 10 ))
do
   echo "$i"
  (( i=$i+1 ))
Done
```

#Start of for loop

```
for a in 1 2 3 4 5 6 7 8 9 10
do
    # if a is equal to 5 break the loop
    if [ $a == 5 ]
    then
        break
    fi
    # Print the value
    echo "Iteration no $a"
done
```

forloop ((; ;))

Display numbers from 1 to 10:

```
for((i=1;i<=10;i++));
do
echo "Number = $i"
done
```

For with break

```
for I in 1 2 3 4 5
do
  statements1
  statements2
  if (condition)
  then
    break
  fi
  statements3
done
```

For with continue

```
for I in 1 2 3 4 5
do
  statements1      #Executed for all values of "I", up to a disaster-condition if any.
  statements2
  if (condition)
  then
    continue   #Go to next iteration of I in the loop and skip statements3
  fi
  statements3
done
```

To display factorial of given number.

```
echo "Enter a number"
read num

fact=1

while [ $num -gt 1 ]
do
  fact=$((fact * num))  #fact = fact * num
  num=$((num - 1))      #num = num - 1
done

echo $fact
```

To display 1 to 100 numbers

```
i=1
while [ $i -le 100 ]
do
    echo $i
    i=$(($i+1))
done
```

for loop with start and end values

```
START=5
END=10
for i in $(seq $START $END)
do
   echo "Doing something with $i ..."
done
```

WAP to select subjects of your choice:

```
echo "SELECT YOUR FAVORITE SUBJECT";
echo "1. English"
echo "2. Maths"
echo "3. Computer"
echo "4. Exit from menu "
echo -n "Enter your menu choice [1-4]: "
while :
do
read choice
case $choice in
  # Pattern 1
  1)  echo "You have selected the option 1"
      echo "Selected SUBJECT is English. ";;
  # Pattern 2
  2)  echo "You have selected the option 2"
      echo "Selected SUBJECT is Maths. ";;
  # Pattern 3
  3)  echo "You have selected the option 3"
      echo "Selected SUBJECT is Computer. ";;
  # Pattern 4
  4)  echo "Quitting ..."
      exit;;
  # Default Pattern
  *) echo "invalid option";;
  esac
  echo -n "Enter your menu choice [1-4]: "
done
```

Fibonacci series

```
echo "How many number of terms to be generated ?"

  read n

function fib

{

  x=0

  y=1

  i=2

  echo "Fibonacci Series up to $n terms :"

  echo "$x"

  echo "$y"

  while [ $i -lt $n ]

  do

      i=`expr $i + 1 `

      z=`expr $x + $y `

      echo "$z"

      x=$y

      y=$z

  done

}

r=`fib $n`

echo "$r"
```

Assignment No- 5

Aim: Write a programme (using control statements)
a) To display percentage and grade obtained by student.
b) To implement various calculator operations.
c) To display whether given numbers are divisible by n.

How to define function and use it

```
# Define your function here

sample () {

  echo "Hello World"

}

# Invoke your function

Sample
```

```
#!/bin/sh

# Calling one function from another
number_one () {
   echo "This is the first function speaking..."
   number_two
}

number_two () {
   echo "This is now the second function speaking..."
}

# Calling function one.
number_one
```

a) To display percentage and grade obtained by student.

```
echo "Student Marksheet"

echo "Enter Operating System Marks:"

read os

echo "Enater C++ Marks:"

read cpp

echo "Enater Java Marks:"

read java

echo "*****************"

total=`expr $os + $cpp + $java`

echo "Total Marks:"$total

percentage=`expr $total / 3`

echo "Percentage:" $percentage %

if [ $percentage -ge 60 ]

then

echo "Class: First Class Distinction"

elif [ $percentage -ge 50 ]

then

echo "Class: First class"

elif [ $percentage -ge 40 ]

then

echo "Class: Second class"

else

echo "Class: Fail"

fi
```

b) To implement various calculator operations.

```
clear
sum=0
i="y"
echo " Enter first  no."
read n1
echo "Enter second no."
read n2
while [ $i = "y" ]
do
echo "1.Addition"
echo "2.Subtraction"
echo "3.Multiplication"
echo "4.Division"
echo "Enter your choice"
read ch
case $ch in
   1)sum=`expr $n1 + $n2`
    echo "Sum ="$sum;;
   2)sum=`expr $n1 - $n2`
    echo "Sub = "$sum;;
   3)sum=`expr $n1 \* $n2`
    echo "Mul = "$sum;;
   4)sum=`echo "scale=2;$n1/$n2"|bc`
      echo "div=" $sum;;
   *)echo "Invalid choice";;
esac
echo "Do u want to continue ?[y/n]"
read i
if [ $i != "y" ]
then
   exit
fi
done
```

c) To display whether given numbers are divisible by n.

```
clear

echo "Enter the number "
read n
c=`expr $n % 7`
if test $c -eq 0
 then
        echo "$n is divisible by 7"
else
     echo "$n is not divisible by 7"
     n1=`expr $n - $c`
     n2=`expr $n - $c + 7`
     echo "Previous number which is divisible is " $n1
     echo "next number which is divisible is     " $n2
fi
```

Assignment No- 6

Aim: WAP to implement first come, first served (FCFS) scheduling algorithm.

Theory :

Given n processes with their burst times, the task is to find average waiting time and average turnaround time using FCFS scheduling algorithm. First in, first out (FIFO), also known as first come, first served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes in the order that they arrive in the ready queue. In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed. Here we are considering that arrival time for all processes is 0.

How to compute below times in Round Robin using a program?

1. Completion Time: Time at which process completes its execution.
2. Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time
3. Waiting Time(W.T): Time Difference between turn around time and burst time. Waiting Time = Turn Around Time – Burst Time

First Come First Serve is a scheduling algorithm used by the CPU to schedule jobs. It is a Non-Preemptive Algorithm.

First Come First Serve

- Completion time: Time when the execution is completed.
- Turn Around Time: The sum of the burst time and waiting time gives the turn-around time
- Waiting time: The time the processes need to wait for it to get the CPU and start execution is called waiting time.

Important Points

- It is non-preemptive
- Average waiting time is not optimal
- Cannot utilize resources in parallel

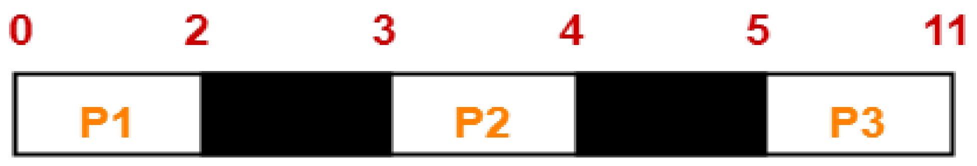
Consider the set of 3 processes whose arrival time and burst time are given below-

Process Id	Arrival time	Burst time
P1	0	2
P2	3	1
P3	5	6

If the CPU scheduling policy is FCFS, calculate the average waiting time and average turn around time.

Solution-

Gantt Chart-



Gantt Chart

Here, black box represents the idle time of CPU.

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	Exit time	Turn Around time	Waiting time
P1	2	2 – 0 = 2	2 – 2 = 0
P2	4	4 – 3 = 1	1 – 1 = 0
P3	11	11- 5 = 6	6 – 6 = 0

Now,

- Average Turn Around time = (2 + 1 + 6) / 3 = 9 / 3 = 3 unit
- Average waiting time = (0 + 0 + 0) / 3 = 0 / 3 = 0 unit

Code:

```
 #include<stdio.h>
main()
{
int n,a[10],b[10],t[10],w[10],g[10],i,m;
float att=0,awt=0;
for(i=0;i<0;i++)
{
a[i]=0;b[i]=0;w[i]=0;g[i]=0;
}
printf("Enter the number of processes:");
scanf("%d",&n);
printf("Enter the burst time:");
for(i=0;i<n;i++)
scanf("%d",&b[i]);
printf("Enter the arrival time:");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
g[0]=0;
for(i=0;i<10;i++)
{
w[i]=g[i]-a[i];
t[i]=g[i+1]-a[i];
awt=awt+w[i]
att=att+t[i];
}
awt=awt/n;
att=att/n;
printf("\n\t Process\t Waiting time \t Turnaround time\n");
for(i=0;i<n;i++)
{
printf("\t %d\t\t  %d\t\\t  %d\n"I,w[i],t[i]);
}
printf("\n The average waiting time is %t \n",awt);
printf("The average turnaround time is %f\n",att);
```

OUTPUT:

```
[examuser35@localhost Jebastin]$ cc fcfs.c
[examuser35@localhost Jebastin]$ ./a.out
Enter the number of processes:3
Enter the burst time:1
2
```

```
3
Enter the arrival time:0
1
2

Process      Waiting time        Turnaround time
    1
    1                3                    2
    2                6                    4

The average waiting time is 3.3333
The average turnaround time is 2.333
```

Assignment No- 7

Aim: WAP to implement Shortest Job First(SJF) algorithm.

Shortest job first(SJF) is a scheduling algorithm, that is used to schedule processes in an operating system. It is a very important topic in Scheduling when compared to round-robin and FCFS Scheduling.

There are two types of SJF

- Pre-emptive SJF
- Non-Preemptive SJF

These algorithms schedule processes in the order in which the shortest job is done first. It has a minimum average waiting time.

There are 3 factors to consider while solving SJF, they are

1. BURST Time
2. Average waiting time
3. Average turnaround time

Non-Preemptive Shortest Job First

Here is an example

Processes Id	Burst Time	Waiting Time	Turn Around Time
4	3	0	3
1	6	3	9
3	7	9	16
2	8	16	25

Average waiting time = **7**

Average turnaround time = **13**

T.A.T= waiting time + burst time

Code:

```
#include<stdio.h>
main()
{
int n,j,temp,temp1,temp2,pr[10],b[10],t[10],w[10],p[10],i;
float att=0;awt=0;
for(i=0;i<10;i++)
{
  b[i]=0;w[i]=0;
}
printf(“Enter the number of process:”);
scanf(“%d”,&n);
printf(“\n Enter the burst time”);
for(i=0;i<n;i++)
{
scanf(“%d”,&b[i]);
p[i]=i;
}
for(i=0;i<n;i++)
{
for(j=1;j<n;j++)
{
if(b[i]>b[j]
{
temp=b[i];
temp1=p[i];
b[i]=b[j];
p[i]=p[j];
b[j]=temp;
p[j]=temp1;
}
}
w[0]=0;
for(i=0;i<n;i++)
w[i+1]=w[i]+b[i];
for(i=0;i<n;i++)
{
t[i]=w[i]+b[i];
awt=aw+w[i]:
att=att+t[i];
}
awt=awt/n;
att=att/n;
printf("\n\t Process \tWaitimgtime \t Turnaroundtime\n);
for(i=0;i<n;i++)
printf("\t  p[%d] \t %d\t\t %d\n",p[i].w[i],t[i]);
printf("\n The average waiting time is %t \n",awt);
printf("The average turnaround time is %f\n",att);
return 1;
}
```

OUTPUT:

```
[examuser35@localhost Jebastin]$ cc sjf.c
[examuser35@localhost Jebastin]$ ./a.out
Enter the number of process:5
Enter the burst time:5
4
3
2
1

Process       Waiting time        Turnaround time
 P[4]              0                    1

 P[0]              1                    6

 P[1]              6                    10
 P[2]              10                   13
 P[3]              13                   15

The average waiting time is 6.000000
The average turnaround time is 9.000000
```

Assignment No- 8

Aim: Understand and implement Round Robin algorithm.

Characteristics of Round-Robin Scheduling

Here are the important characteristics of Round-Robin Scheduling:

- Round robin is a pre-emptive algorithm
- The CPU is shifted to the next process after fixed interval time, which is called time quantum/time slice.
- The process that is preempted is added to the end of the queue.
- Round robin is a hybrid model which is clock-driven
- Time slice should be minimum, which is assigned for a specific task that needs to be processed. However, it may differ OS to OS.
- It is a real time algorithm which responds to the event within a specific time limit.
- Round robin is one of the oldest, fairest, and easiest algorithm.
- Widely used scheduling method in traditional OS.

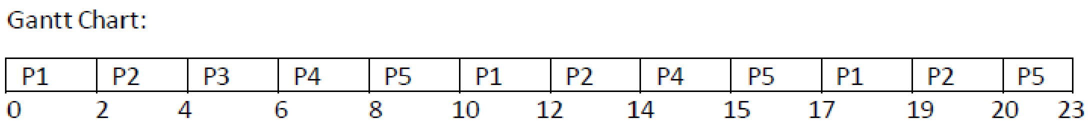
– Once a process is executed for a given time period, the process is preempted and the next process execution starts for the given time period.

– Round robin scheduling uses context switching to save states of preempted process.

- **Let us take an example to understand the scheduling –**
- Now, we will take different examples to demonstrate how does round robin cpu scheduling works.
- **Example 1**
- Assume there are 5 processes with process ID and burst time given below

– Time quantum: 2
– Assume that all process arrives at 0.

- Now, we will calculate average waiting time for these processes to complete.
- **Solution –**
- We can represent execution of above processes using GANTT chart as shown below –



- Round Robin Example Gantt Chart – 1

– First p1 process is picked from the ready queue and executes for 2 per unit time (time slice = 2).
If arrival time is not available, it behaves like FCFS with time slice.
– After P2 is executed for 2 per unit time, P3 is picked up from the ready queue. Since P3 burst
time is 2 so it will finish the process execution at once.
– Like P1 & P2 process execution, P4 and p5 will execute 2 time slices and then again it will start
from P1 same as above.

Waiting time = Turn Around Time – Burst Time
P1 = 19 – 6 = 13
P2 = 20 – 5 = 15
P3 = 6 – 2 = 4

P4 = 15 – 3 = 12
P5 = 23 – 7 = 16

Average waiting time = (13+15+4+12+16) / 5 = 12

Source Code-

```
round_robin()
{
   #Variables and arrays
   declare -a processesRR=("${!1}")
   declare -a arrivalsRR=("${!2}")
   declare -a burstsRR=("${!3}")
   time_quantum="${!4}" #Redirection again needed so $4 won't just print out tquantum
   numberOfProcesses="${#processesRR[@]}" #declare how many processes are there
   numberOfProcessesRunning=$numberOfProcesses #for keeping track on which processes have finished.

   #Time remaining assignment
   for ((i=0; i<$numberOfProcesses; i++))
   do
      remaining_time[i]=${burstsRR[i]}
   done

   printf "Program Name\t Arrival Time\t Burst Time\t Completion Time\t Turnaround Time\t Waiting
Time\t\n\n"
   #Actual algorithm. This also generates the table.
   complete=0 #For checking the completion of a process
   time=0 #For checking time
   i=0 #Index
   resetPrinciple=$(($numberOfProcesses - 1)) #Fix to test operation not allowing arithmetic
   while [[ $numberOfProcessesRunning != 0 ]]
   do
      if (( ${remaining_time[i]} <= $time_quantum && ${remaining_time[i]} > 0 )) #indicates process
completes here.
      then
         ((time += ${remaining_time[i]}))
         remaining_time[i]=0
         complete=1

         elif (( ${remaining_time[i]} > 0 )) #process was not completed
         then
            ((${remaining_time[i]} - $time_quantum))
            ((time += ${remaining_time[i]}))
      fi

      if (( ${remaining_time[i]} == 0 && $complete == 1 ))
      then
         numberOfProcessesRunning=$(($numberOfProcessesRunning - 1))
         completion_timeRR[i]=$time
         turnaround_timeRR[i]=$(($time - ${arrivalsRR[i]}))
         waiting_timeRR[i]=$((${turnaround_timeRR[i]} - ${burstsRR[i]}))
```

```
          total_turnaroundRR=$(($total_turnaroundRR + ${turnaround_timeRR[i]}))
          total_waitingRR=$(($total_waitingRR + ${waiting_timeRR[i]}))
          complete=0 #start again anew
          printf " %s\t %s\t %s\t %s\t %s\t %s\t\n\n" "${processesRR[i]}" "${arrivalsRR[i]}"
"${burstsRR[i]}" "${completion_timeRR[i]}" "${turnaround_timeRR[i]}" "${waiting_timeRR[i]}"
        fi

        if (( $i == $resetPrinciple )) #to avoid non-existent entries and let it reset
        then
          i=0
          elif (( ${arrivalsRR[i]} <= $time )) #check time if we can increment
          then
            ((i++))
        else # ¯\_(ツ)_/¯
          i=0
        fi
    done
    avrWaitTime=$(($total_waitingRR / $numberOfProcesses))
    avrTurnTime=$(($total_turnaroundRR / $numberOfProcesses))
    printf "Average waiting time: %s\n" "$avrWaitTime"
    printf "Average turnaround time: %s\n" "$avrTurnTime"

}
# based on http://blockofcodes.blogspot.com/2015/08/first-come-first-serve-scheduling.html
    # print Gantt chart
    printf "\n" # Empty line
    printf "          GANTT CHART          \n"
    printf "          ***********          \n"


    # print top bar
    printf "\n"
    for((i=0; i<$numberOfProcesses; i++))
      do
        for((j=0; j<"${burstsFCFS[i]}"; j++))
          do
            printf -- "--" #this fixes the argument error - Gabs
            printf ""
          done
      done
    printf "\n|"

    # printing process id in the middle
    for((i=0; i<$numberOfProcesses; i++))

      do
        for((j=0; j<${burstsFCFS[i]} - 1; j++))
        do
          #printf "\n"
          printf ${processesFCFS[i]}
        done
        for((j=0; j<${burstsFCFS[i]} - 1; j++))
        do
            printf " "
            printf "|"
        done
    done

printf "\n "
# printing bottom bar
for((i=0; i<$numberOfProcesses; i++))

    do
        for((j=0; j<${burstsFCFS[i]} - 1; j++))
            do
                printf -- "--"
                printf ""
            done
    done

printf "\n"

# printing the time line
printf "0"
for((i=0; i<$numberOfProcesses; i++))

    do
    turnaround_time[i]=$((${completion_time[i]} - ${arrivalsFCFS[i]}))
    for((j=0; j<${burstsFCFS[i]}; j++))

        do
            printf " "
        if [[ ${turnaround_time[i]} > ${turnaround_time[$((j + 1))]} ]]

            then
            printf "\b" # backspace : remove 1 space
            printf "%d " "${turnaround_time[i]}"
        fi
        done
    done


printf "\n"
```

Assignment No- 9

Aim: Implement Producer Consumer Problem using semaphore.

What is Producer Consumer Problem?

Before knowing what is Producer-Consumer Problem we have to know what are Producer and Consumer.

- In operating System **Producer is a process which is able to produce data/item**.
- Consumer is a **Process that is able to consume the data/item produced by the Producer**.
- Both Producer and Consumer share a common memory buffer. This buffer is a space of a certain size in the memory of the system which is used for storage. The producer produces the data into the buffer and the consumer consumes the data from the buffer.

- The producer produces the data into the buffer and the consumer consumes the data from the buffer.

So, what are the Producer-Consumer Problems?

1. Producer Process should not produce any data when the shared buffer is full.

2. Consumer Process should not consume any data when the shared buffer is empty.
3. The access to the shared buffer should be mutually exclusive i.e at a time only one process should be able to access the shared buffer and make changes to it.

For consistent data synchronization between Producer and Consumer, the above problem should be resolved.

Solution For Producer Consumer Problem

To solve the Producer-Consumer problem three **semaphores variable** are used :

Semaphores are variables used to indicate the number of resources available in the system at a particular time. semaphore variables are used to achieve `Process Synchronization.

Full

The full variable is used to track the space filled in the buffer by the Producer process. It is initialized to 0 initially as initially no space is filled by the Producer process.

Empty

The Empty variable is used to track the empty space in the buffer. The Empty variable is initially initialized to the **BUFFER-SIZE** as initially, the whole buffer is empty.

Mutex

Mutex is used to achieve mutual exclusion. mutex ensures that at any particular time only the producer or the consumer is accessing the buffer.

Mutex - mutex is a binary semaphore variable that has a value of 0 or 1.

We will use the Signal() and wait() operation in the above-mentioned semaphores to arrive at a solution to the Producer-Consumer problem.

Signal() - The signal function increases the semaphore value by 1. **Wait()** - The wait operation decreases the semaphore value by 1.

Let's look at the code of Producer-Consumer Process

The code for Producer Process is as follows :

```
void Producer(){

    while(true){

        // producer produces an item/data
        wait(Empty);

        wait(mutex);

        add();

        signal(mutex);

        signal(Full);

    }

}
```

Let's understand the above Producer process code :

- **wait(Empty)** - Before producing items, the producer process checks for the empty space in the buffer. If the buffer is full producer process waits for the consumer process to consume items from the buffer. so, the producer process executes wait(Empty) before producing any item.
- **wait(mutex)** - Only one process can access the buffer at a time. So, once the producer process enters into the critical section of the code it decreases the value of mutex by executing wait(mutex) so that no other process can access the buffer at the same time.
- **add()** - This method adds the item to the buffer produced by the Producer process. once the Producer process reaches add function in the code, it is guaranteed that no other process will be able to access the shared buffer concurrently which helps in data consistency.
- **signal(mutex)** - Now, once the Producer process added the item into the buffer it increases the mutex value by 1 so that other processes which were in a busy-waiting state can access the critical section.
- **signal(Full)** - when the producer process adds an item into the buffer spaces is filled by one item so it increases the Full semaphore so that it indicates the filled spaces in the buffer correctly.

The code for the Consumer Process is as follows :

```
void Consumer() {

    while(true){

    // consumer consumes an item

        wait(Full);
        wait(mutex);

        consume();

        signal(mutex);

        signal(Empty);

    }

}
```

Let's understand the above Consumer process code :

- **wait(Full)** - Before the consumer process starts consuming any item from the buffer it checks if the buffer is empty or has some item in it. So, the consumer process creates one more empty space in the buffer and this is indicated by the full variable. The value of the full variable decreases by one when the wait(Full) is executed. If the Full variable is already zero i.e the buffer is empty then the consumer process cannot consume any item from the buffer and it goes in the busy-waiting state.
- **wait(mutex)** - It does the same as explained in the producer process. It decreases the mutex by 1 and restricts another process to enter the critical section until the consumer process increases the value of mutex by 1.
- **consume()** - This function consumes an item from the buffer. when code reaches the consuming () function it will not allow any other process to access the critical section which maintains the data consistency.
- **signal(mutex)** - After consuming the item it increases the mutex value by 1 so that other processes which are in a busy-waiting state can access the critical section now.
- **signal(Empty)** - when a consumer process consumes an item it increases the value of the Empty variable indicating that the empty space in the buffer is increased by 1.

Why can mutex solve the producer consumer Problem ?

Mutex is used to solve the producer-consumer problem as mutex helps in mutual exclusion. It prevents more than one process to enter the critical section. As mutexes have binary values i.e 0 and 1. So whenever any process tries to enter the critical section code it first checks for the mutex value by using the wait operation.

wait(mutex);

wait(mutex) decreases the value of mutex by 1. so, suppose a process P1 tries to enter the critical section when **mutex value is 1**. P1 executes wait(mutex) and decreases the value of mutex. Now, the **value of mutex becomes 0** when P1 enters the critical section of the code.

Now, suppose Process P2 tries to enter the critical section then it will again try to decrease the value of mutex. But the mutex value is already 0. So, wait(mutex) will not execute, and P2 will now keep waiting for P1 to come out of the critical section.

Now, suppose if P2 comes out of the critical section by executing signal(mutex).

signal(mutex)

signal(mutex) increases the value of mutex by 1.**mutex value again becomes 1**. Now, the process P2 which was in a busy-waiting state will be able to enter the critical section by executing wait(mutex).

So, mutex helps in the mutual exclusion of the processes.

In the above section in both the Producer process code and consumer process code, we have the wait and signal operation on mutex which helps in mutual exclusion and solves the problem of the Producer consumer process.

Conclusion

- Producer Process produces data item and consumer process consumes data item.
- Both producer and consumer processes share a common memory buffer.
- Producer should not produce any item if the buffer is full.
- Consumer should not consume any item if the buffer is empty.
- Not more than one process should access the buffer at a time i.e mutual exclusion should be there.
- Full, Empty and mutex semaphore help to solve Producer-consumer problem.
- Full semaphore checks for the number of filled space in the buffer by the producer process
- Empty semaphore checks for the number of empty spaces in the buffer.
- mutex checks for the mutual exclusion.

Assignment No- 10

Aim: Study and implement Page replacement algorithm.

Page Replacement Algorithms In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

1.First In First Out (FIFO) page replacement algorithm –

This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

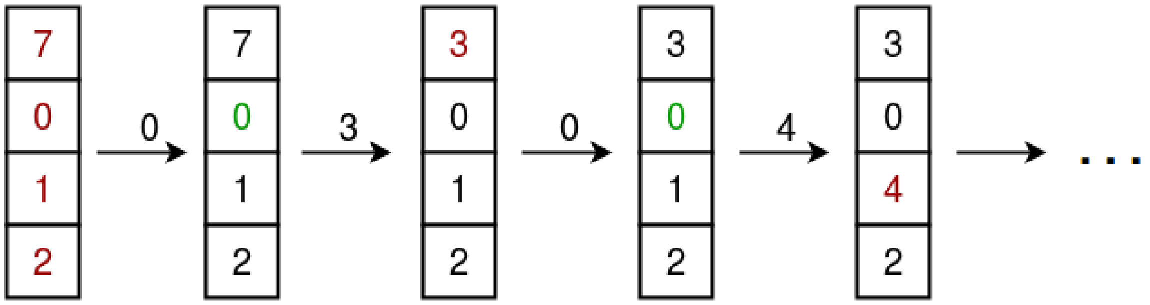
Example -1. Consider page reference string 1, 3, 0, 3, 5, 6 and 3 page slots. Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> 3 **Page Faults.**
when 3 comes, it is already in memory so —> 0 Page Faults. Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>1**Page Fault.**
Finally 6 comes, it is also not available in memory so it replaces the oldest page slot i.e. 3 —>6 **Page Fault.**
So total page faults = **5**.

Example -2. Consider the following reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1. Using FIFO page replacement algorithm –

0	2	1	6	4	0	1	0	3	1	2	1
0	0	0	0	4	4			4	4	2	
	2	2	2	2	0		hit	0	0	0	
		1	1	1	1	hit		3	3	3	
			6	6	6			6	1	1	hit

So, total number of page faults = 9. Given memory capacity (as number of pages it can hold) and a string representing pages to be referred, write a function to find number of page faults.

2.In Least Recently Used (LRU) algorithm is a Greedy algorithm where the pageto be replaced is least recently used. The idea is based on locality of reference, the least recently used page is not likely.
Let say the page reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 . Initially we have 4 page slots empty.
Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**
0 is already there so —> **0 Page fault.**
when 3 came it will take the place of 7 because it is least recently used —>**1 Page fault**
0 is already in memory so —> **0 Page fault**.
4 will takes place of 1 —> **1 Page Fault**
Now for the further page reference string —> **0 Page fault** because they are already available in the memory.



Total Page faults = 6