Operating System

Code: PCC-CS592

Laboratory Manual

Department of Computer Science & Engineering (AIML & DS)

Techno Main Salt Lake

GENERAL INSTRUCTIONS FOR STUDENTS

1. Do not enter the Laboratory without prior permission.
2. Switch off your mobile phones during Lab class and maintain silence.
3. Save your files only on the specific destination folders as instructed.
4. Do not play games, watch movies, chat or listen to music during the class.
5. Do not change desktop setting, screen saver or any other system settings.
6. Do not use any external storage device without prior permission.
7. Do not install any software without prior permission.
8. Do not browse any restricted, illegal or spam sites.

GENERAL ADDRESS FOR LABORATORY TEACHERS

1. Submission of documented lab reports related to completed lab assignments should be done during the following lab session.
2. The promptness of submission should be encouraged by way of marking and evaluation patterns as reflected in the lab rubric which eventually will benefit the students.

Program Outcomes (PO)

PO1. **Engineering Knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and engineering specialization to the solution of complex engineering problems.

PO2. **Problem analysis:** Identify, formulate, research literature, and analyse engineering problems to arrive at substantiated conclusions using first principles of mathematics, natural and engineering sciences.

PO3. **Design/Development of solutions:** Design solutions for complex engineering problems and design system components, processes to meet the specifications with consideration for the public health and safety and the cultural societal and environmental considerations.

PO4. **Conduct investigations of complex problems:** Use research based knowledge including design of experiments, analysis and interpretation of data and synthesis of the information to provide valid conclusions.

PO5. **Modern and usage:** 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.

PO6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to access societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of and need for sustainable development.

PO8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9. **Individual and team work:** Function effectively as an individual, and as a member or leader in teams, and in multidisciplinary settings.

PO10. **Communications:** Communicate effectively with the engineering community and with the society at large. Be able to comprehend and write effective repots documentation. Make effective presentations and give and receive clear instructions.

PO11. **Project management and finance:** Demonstrate knowledge and understanding of engineering and management principles and apply these to one's own work, as a member and leader in a team. Manage projects in multidisciplinary environments.

PO12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change.

NAME OF THE PROGRAM:	DEGREE: B. TECH
COURSE NAME: Operating System Lab	SEMESTER: 5th
COURSE CODE: PCC- CS592	COURSE CREDIT: 2
COURSE TYPE: LAB	CONTACT HOURS: 4P

SYLLABUS

1. **Managing Unix/Linux Operating System [8P]:**

Creating a bash shell script, making a script executable, shell syntax (variables, conditions, control structures, functions, commands). Partitions, Swap space, Device files, Raw and Block files, formatting disks, Making file systems, Superblock, I-nodes, File system checker, Mounting file systems, Logical Volumes, Network File systems, Backup schedules and methods Kernel loading, init, and the inittab file, Run-levels, Run level scripts. Password file management, Password security, Shadow file, Groups and the group file, Shells, restricted shells, user management commands, homes and permissions, default files, profiles, locking accounts, setting passwords, Switching users, Switching groups, Removing users &user groups.

2. Process [4P]: starting a new process, replacing a process image, duplicating a process image, waiting for a process, zombie process.

3. Signal [4P]: signal handling, sending signals, signal interface, signal sets.

4. Semaphore [6P]: programming with semaphores (use functions semctl, semget, semop, set_semvalue, del_semvalue, semaphore_p, semaphore_v).

5. POSIX Threads [6P]: programming with pthread functions (viz. pthread_create, pthread_join, pthread_exit, pthread_attr_init, pthread_cancel)

6. Inter-process communication [6P]: pipes(use functions pipe, popen, pclose), named pipes(FIFOs, accessing FIFO), message passing & shared memory).

NAME OF THE PROGRAM:	**DEGREE:**
COURSE NAME:	**SEMESTER:** ***5th***
COURSE CODE:	**COURSE CREDIT:** *2*
COURSE TYPE: *LAB*	**CONTACT HOURS:** *4P*

Exp. No.	List of Experiments	Week
1.	**INTRODUCTION TO OPERATING SYSTEMS**	
2.	**LINUX COMMANDS FILE SYSTEM**	
3.	**SHELL PROGRAMMING BASICS**	
4.	**PROCESS SYSTEM CALLS – FORK, EXIT, WAIT**	

EX.NO- 1: INTRODUCTION TO OPERATING SYSTEMS

AIM: To study the functions of an operating system.

An Operating System (OS) (as shown in Fig 1) is an interface between a computer user and computer hardware. An operating system is software that performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.

Following are some of the important functions of an operating System:

- ✓ Memory Management
- ✓ Processor Management
- ✓ Device Management
- ✓ File Management
- ✓ Security
- ✓ Control over system performance
- ✓ Job Accounting
- ✓ Error-detecting aids
- ✓ Coordination between other software and end users.

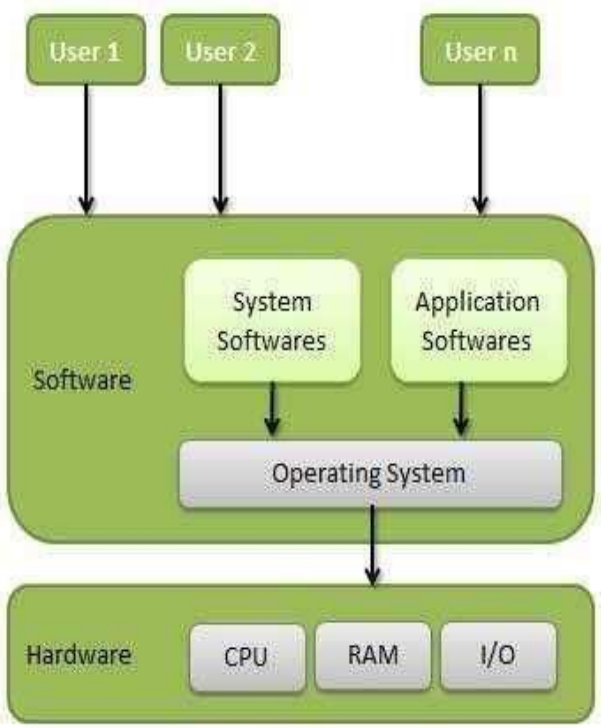


Figure 1. Operating System

LINUX OPERATING SYSTEM

Linux is a free and open source operating system and it is a clone version of UNIX operating system. It is open source as its source code is freely available. It is free to use. Linux was designed considering UNIX compatibility. Its functionality list is quite similar to that of UNIX.

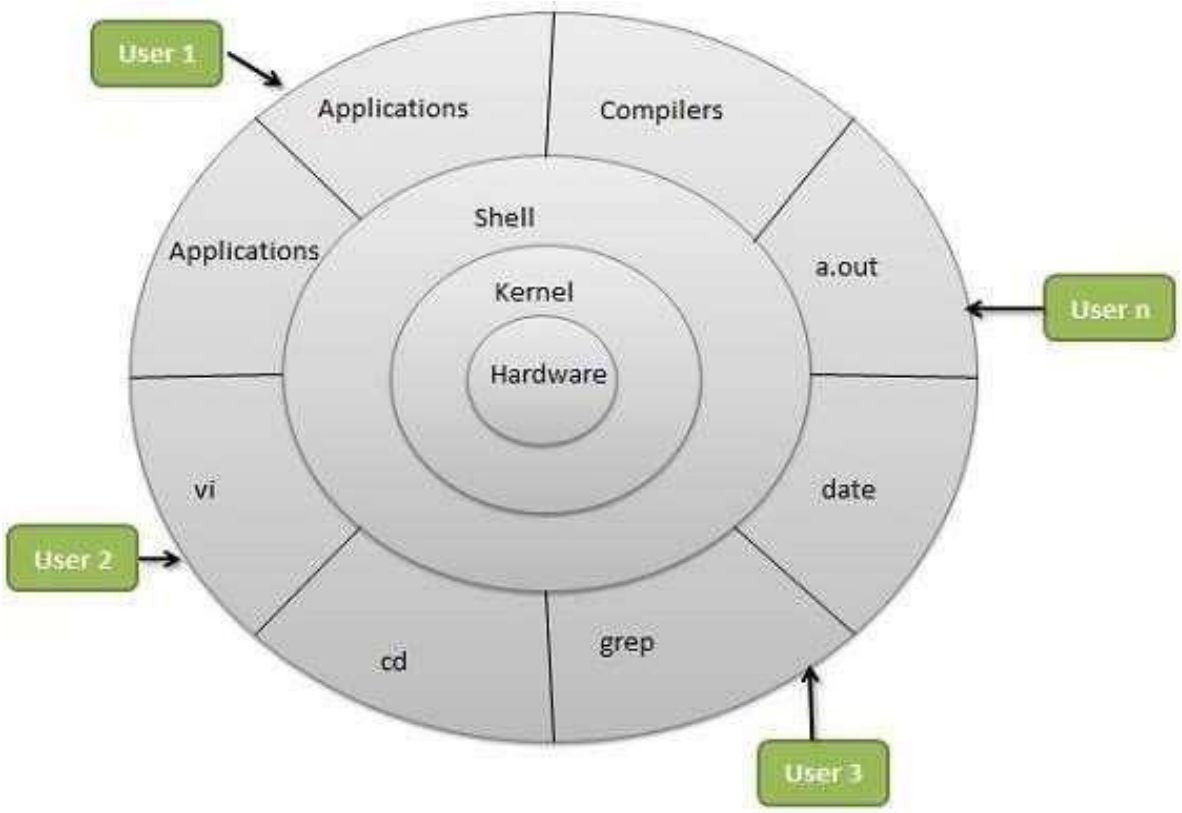


Figure 2. Linux OS Architecture

The architecture of a Linux system consists of the following layers −

Hardware layer: Hardware consists of all peripheral devices (RAM/ HDD/ CPU etc).

Kernel: It is the core component of the Operating System, interacts directly with hardware, and provides low-level services to upper-layer components.

Shell: An interface to kernel, hiding complexity of kernel's functions from users. The shell takes commands from the user and executes kernel's functions.

Utilities: Utility programs that provide the user most of the functionalities of an operating systems.

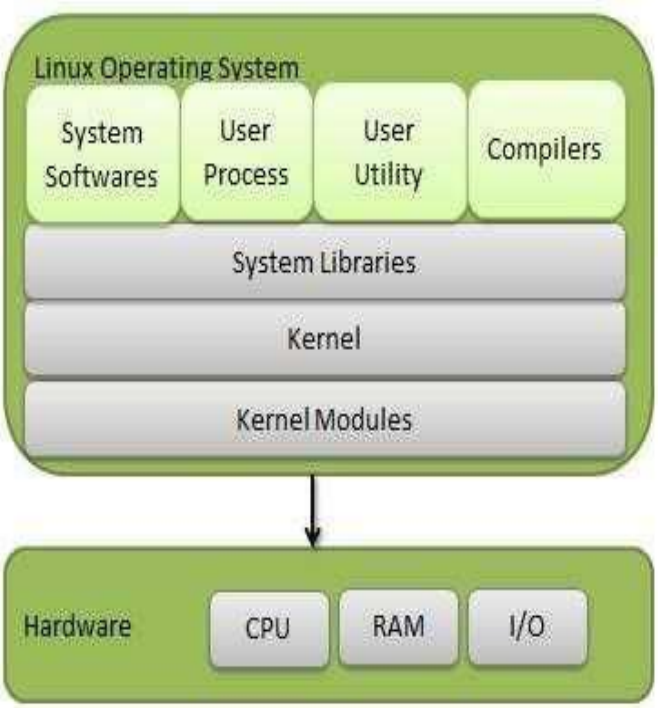
COMPONENTS OF A LINUX OPERATING SYSTEM

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 consists of various modules and it interacts directly with the underlying hardware. Kernel provides the required abstraction to hide low level hardware details to system or application programs.

System Library: System libraries are special functions or programs using which application programs or system utilities accesses Kernel's features. These libraries implement most of the functionalities of the operating system and do not requires kernel module's code access rights.

System Utility: System Utility programs are responsible to do specialized, individual level tasks.



NAME OF THE PROGRAM:	DEGREE:
COURSE NAME:	SEMESTER:
COURSE CODE:	COURSE CREDIT: *2*
COURSE TYPE: *LAB*	CONTACT HOURS: *4P*

EX.NO: 2 LINUX COMMANDS FILE SYSTEM

AIM: To study and implement about the various basic Linux commands.

I. FILE RELATED COMMANDS

1. cat

- The cat command is used to create a file.
- The cat command is used to display the contents of a file.
- The cat command is also used merge multiple files into a single file

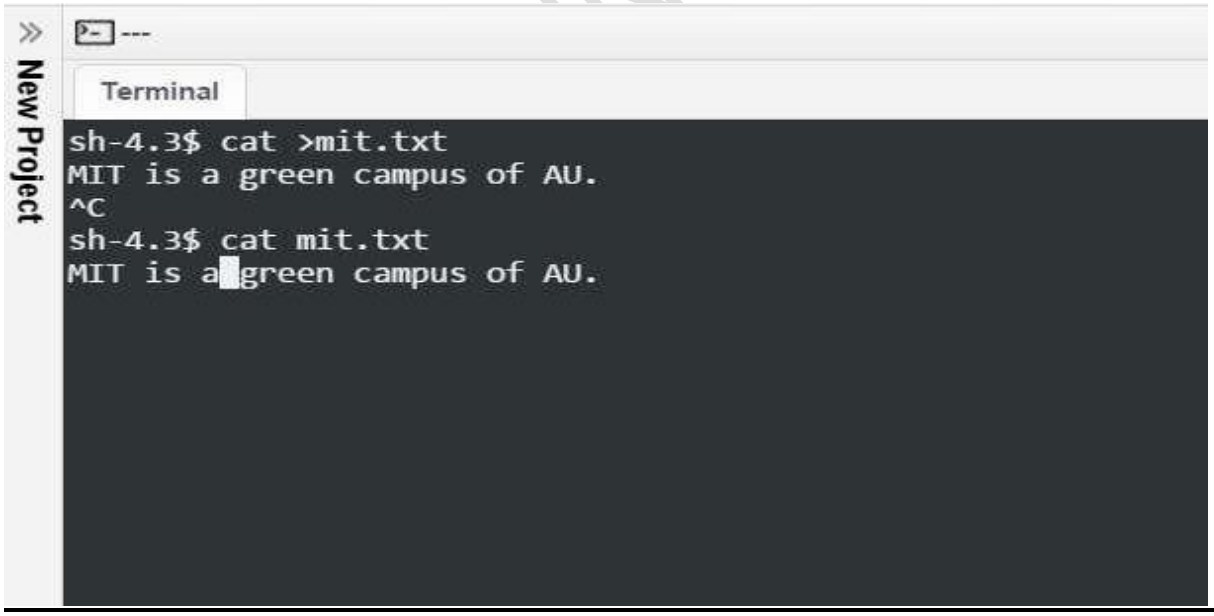
Syntax

$ cat > filename (Create a new file)

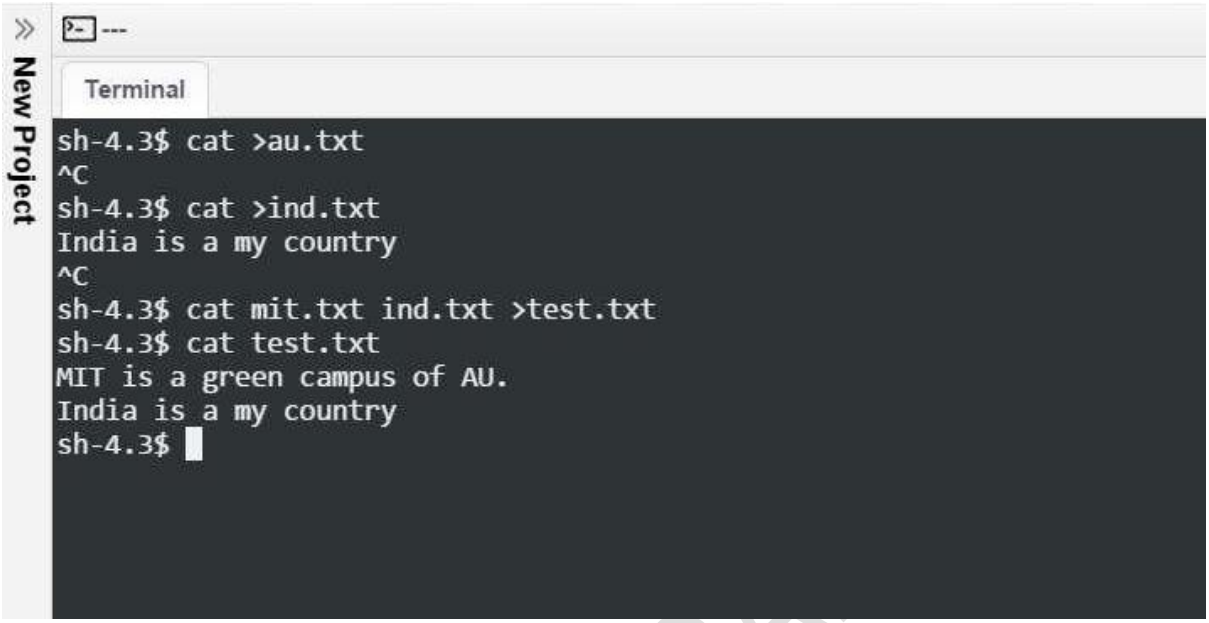
$ cat <filename> (Display the contents of file)

$ cat file1 file2 >file3 (merge the contents of file1, file2 into file 3)

1.1 File Creation & Display its Contents using cat command



1. 2 Files Merging using cat command

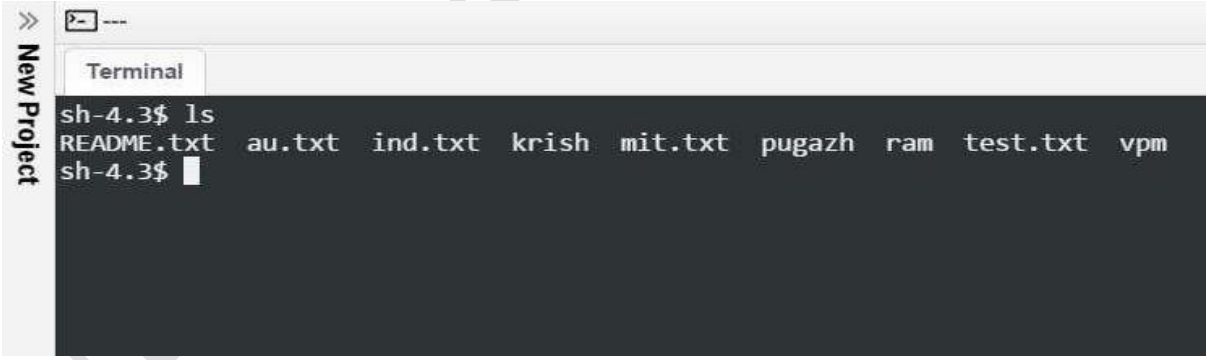


2. ls command

- Listing files and directories
- The ls command is used to display the contents of a directory.

Syntax

$ ls View the contents of directory



3. clear command

- This command is used to clear the terminal screen
- $ clear

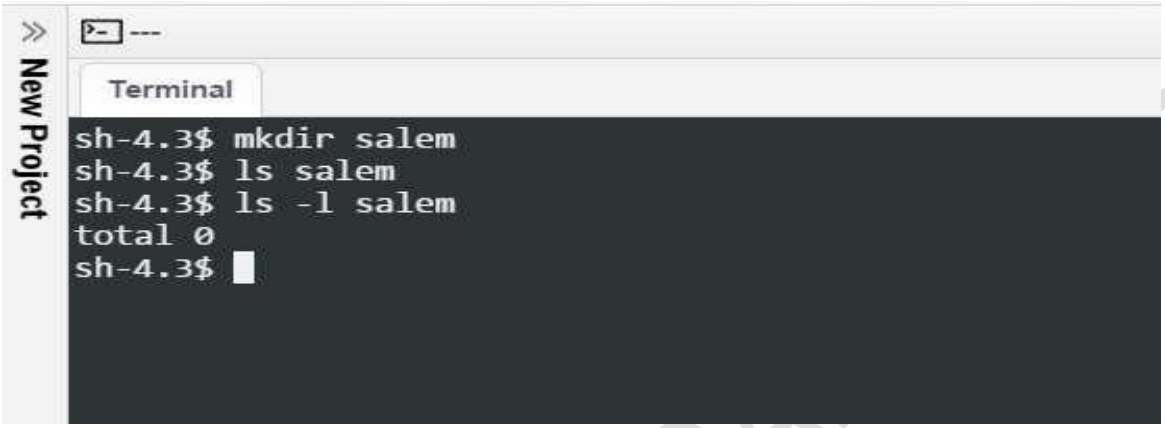
II. DIRECTORY RELATED COMMANDS

1. mkdir

- This command is used create an empty directory in a disk

Syntax

$ mkdir <dname>



1.1 Empty Directory Creation

2. rmdir

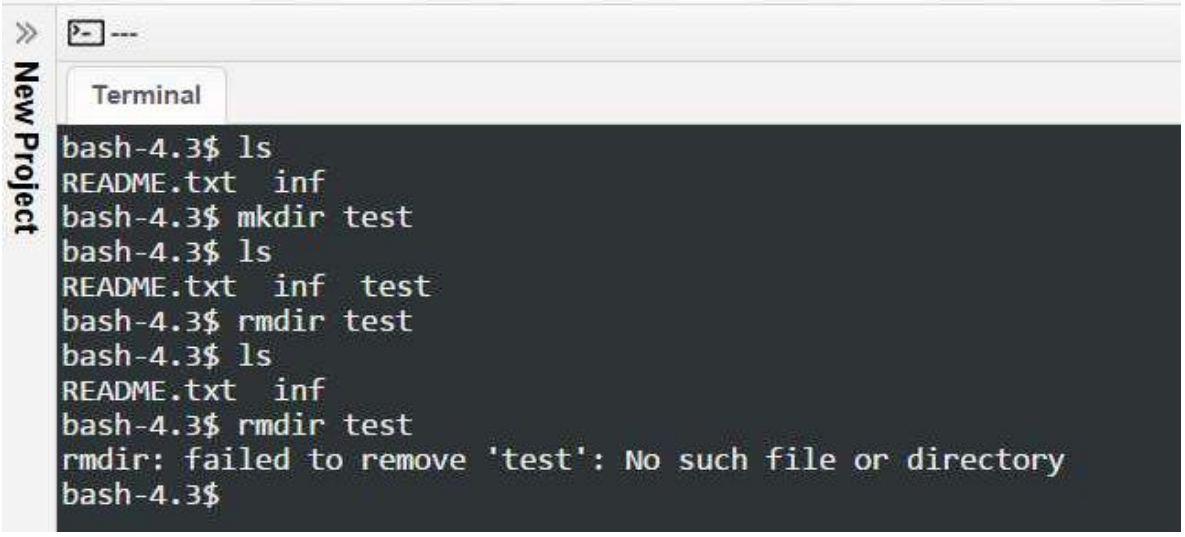
- This command is used to remove a directory from the disk

Rules for Directory Deletion

- Directory must be empty
- Directory can't be a current working directory

Syntax

$ rmdir <dname>

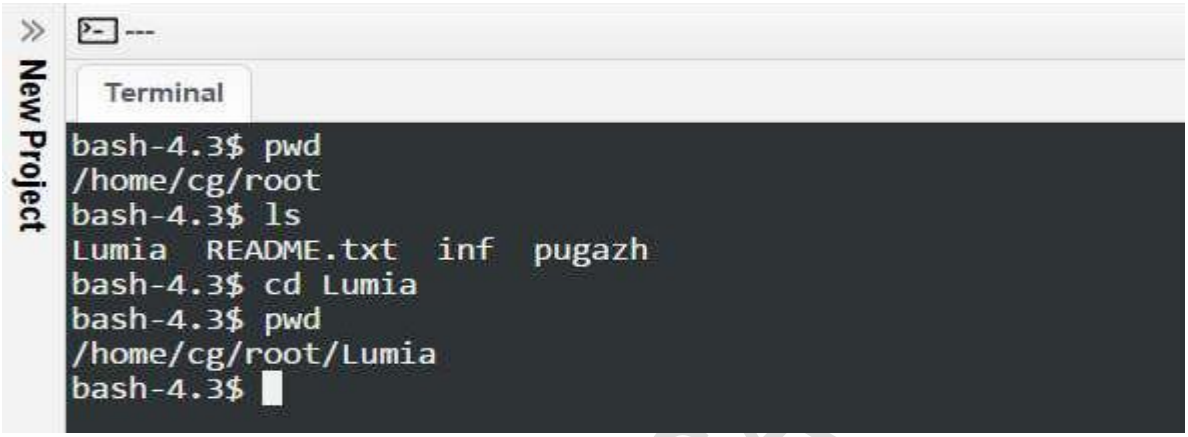


3. cd

- This command is used to move from one directory to another directory.

Syntax

$ cd <dname>



4. pwd

- pwd stands for "print working directory"
- This command is used to print the current working directory

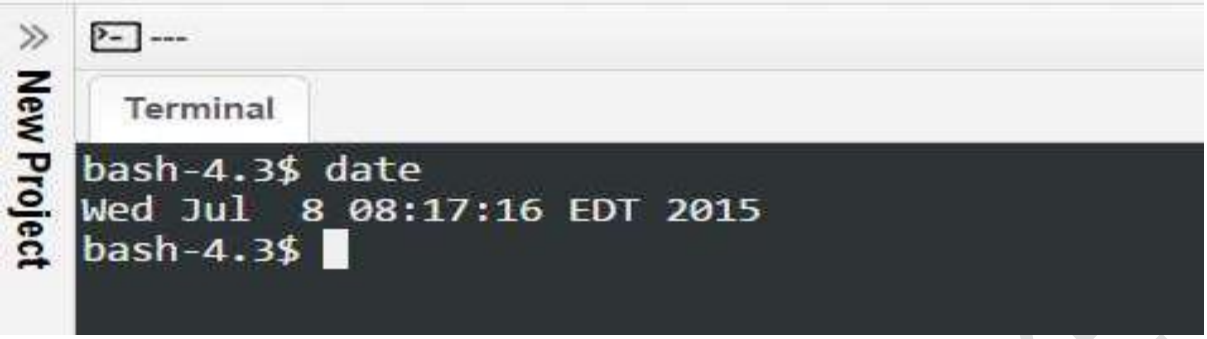
Syntax

$ cd <dname>

III. General Purpose Commands

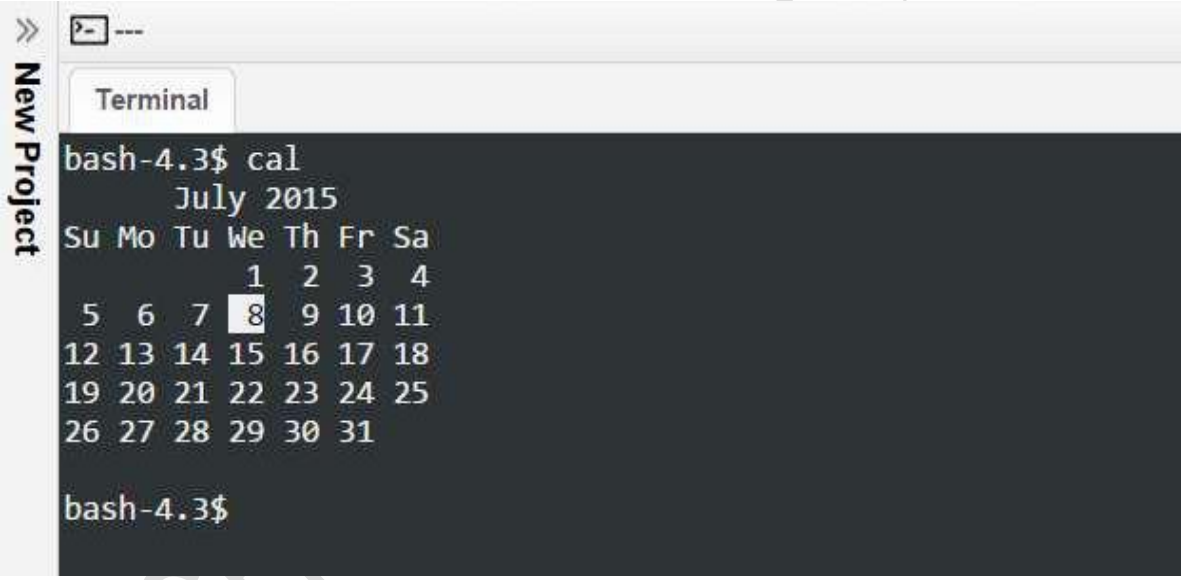
1. date command

- This command is used to display the current date with day, month, date, time (24 Hrs clock) & year
- $ date

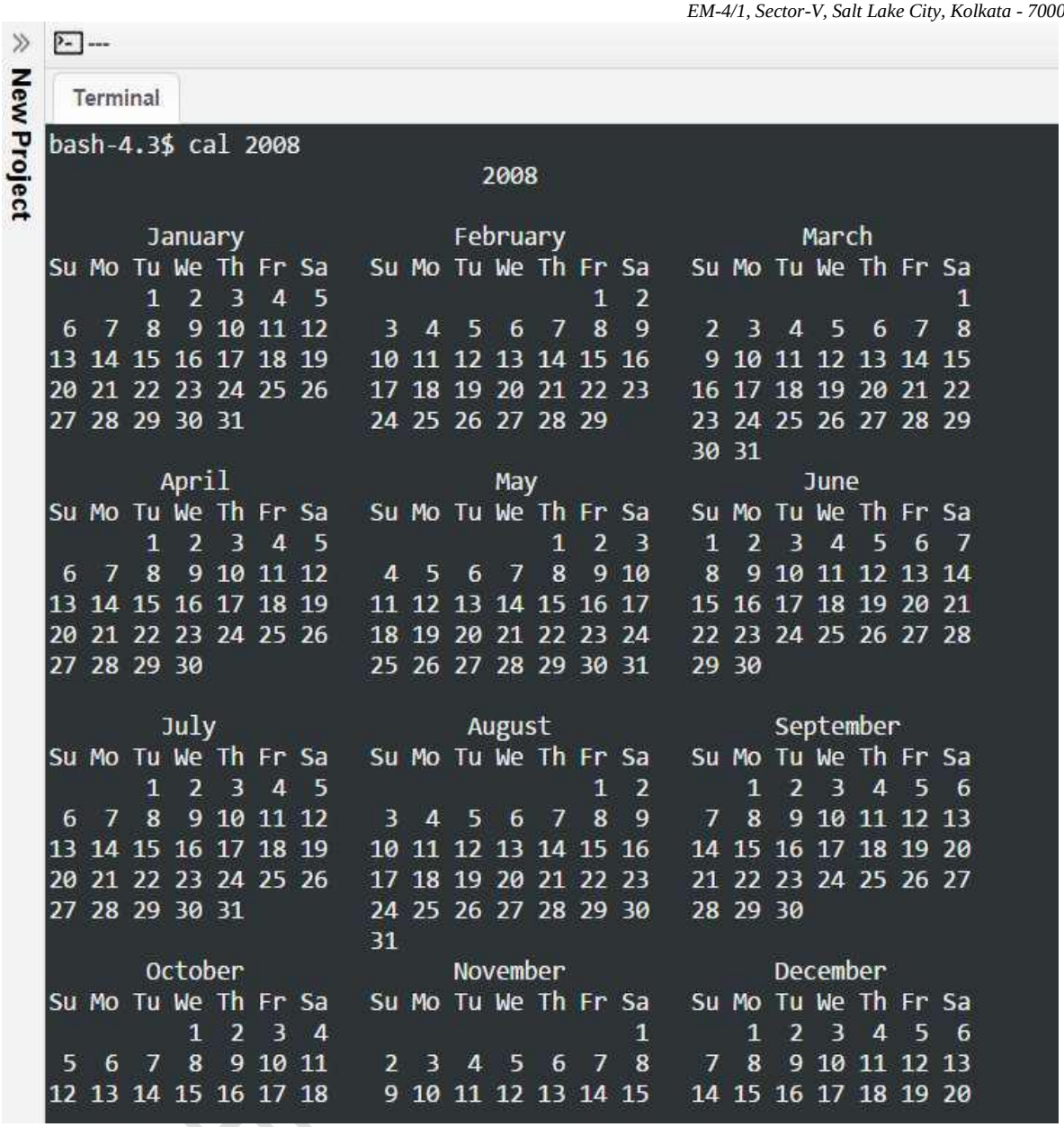


2. cal command

- Linux calendar
- This command is used to display the current month, all months of particular year
- $ cal (Show the current month of current year)
- $ cal 2015 (Show all months in year 2015)

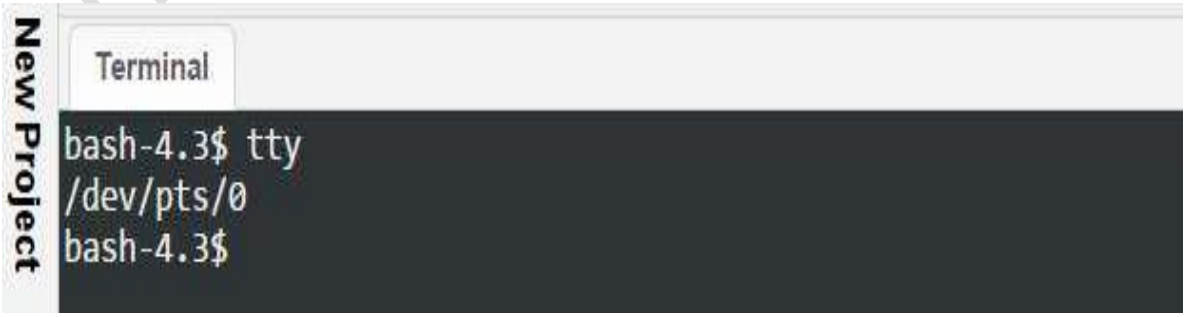


2.1 Displaying Entire Year



3. tty command

- The tty (teletype) command is used to print the current terminal name
- $ tty



3. **cp command in Linux**
   - The **cp command** of Linux is equivalent to copy-paste and cut-paste in Windows.

Command:

```
ls
cp first.txt second.txt
ls
```

Output:

```
first.txt   main.sh
first.txt   main.sh second.txt
```

4. **mv command in Linux**
   - The **mv command** is generally used for renaming the files in Linux.

Command:

```
ls
mv first.txt renamed.txt
ls
```

Output:

```
first.txt   main.sh
main.sh   renamed.txt
```

5. **touch command in Linux**
   - The **touch command** creates an empty file when put in the terminal in this format as touch **<file name>**

Command:

```
ls
touch GeeksforGeeks.txt
ls
```

Output:

```
main.sh
GeeksforGeeks.txt   main.sh
```

6. **ln command in Linux**
   - The **ln command** is used to create a shortcut link to another file. This is among the most important Linux commands to know if you want to operate as a Linux administrator.

Command:

```
1  mkdir Demo
2  mkdir Linked
3  ln -s Demo Linked
```

Output:

```
Linked/Demo
```

7. **ps command in Linux**
   - **ps command** in Linux is used to check the active processes in the terminal.

Command:

```
1  ps
```

Output:

```
PID TTY          TIME CMD
8454 pts/521  00:00:00 bash
11982 pts/521  00:00:00 bash
11983 pts/521  00:00:00 ps
```

8. **man command in Linux**
   - **The** man command **displays a user manual for any commands or utilities available in the Terminal, including their name, description, and options.**
   - **Command to view the full manual: man <command name>**
   - **For example, suppose you want to look up the manual for the ls command:** man ls

Command:

```
1  man -f ls
```

Output:

```
ls (1) - list directory contents
```

9. **echo command in Linux**
   - echo command **in Linux is specially used to print something in the terminal**

Command:

```
1  echo "Hello World"
```

Output:

```
Hello World
```

10. **sort command in Linux**
    - The **sort** command is used generally to sort the output of the file. Let's use the command and see the output.

Command: (We are using the cat command to see the file content)

```
1  cat multiple.txt
```

Output: (The content of multiple.txt file in the terminal)

```
Hello World
GeeksforGeeks
Thank you
```

11. **df command in Linux**
    - **df command** in Linux gets the details of the file system.

Command:

```
1  df -h
```

Output:

```
Filesystem      Size  Used Avail Use% Mounted on
overlay         875G  120G  711G  15% /
tmpfs            63G     0   63G   0% /dev
tmpfs            63G     0   63G   0% /sys/fs/cgroup
/dev/nvme0n1p3  875G  120G  711G  15% /dev/init
shm              64M     0   64M   0% /dev/shm
tmpfs            63G     0   63G   0% /proc/acpi
tmpfs            63G     0   63G   0% /proc/scsi
tmpfs            63G     0   63G   0% /sys/firmware
```

Here we have used **df -h** as simply typing **df** will return the output in bytes which is not readable, so we add **-h** to make the outputs more readable and understandable.

12. wc command in Linux

wc command in Linux indicates the number of words, characters, lines, etc using a set of options.

- **wc -w** shows the number of words
- **wc -l** shows the number of lines
- **wc -m** shows the number of characters present in a file

Let's see one example of these options

Command:

```
1 touch file.txt
2 echo -e "This file has only six words" › file.txt
3 wc -w file.txt
```

Output:

```
6 file.txt
```

Here we used the **touch** command to create a text file and then used the **echo** command to input a sentence that contains six words and we used the **wc -w** command to calculate the number of words in it.

13. ls Command in Linux

ls is a Linux shell command that lists directory contents of files and directories. It provides valuable information about files, directories, and their attributes.

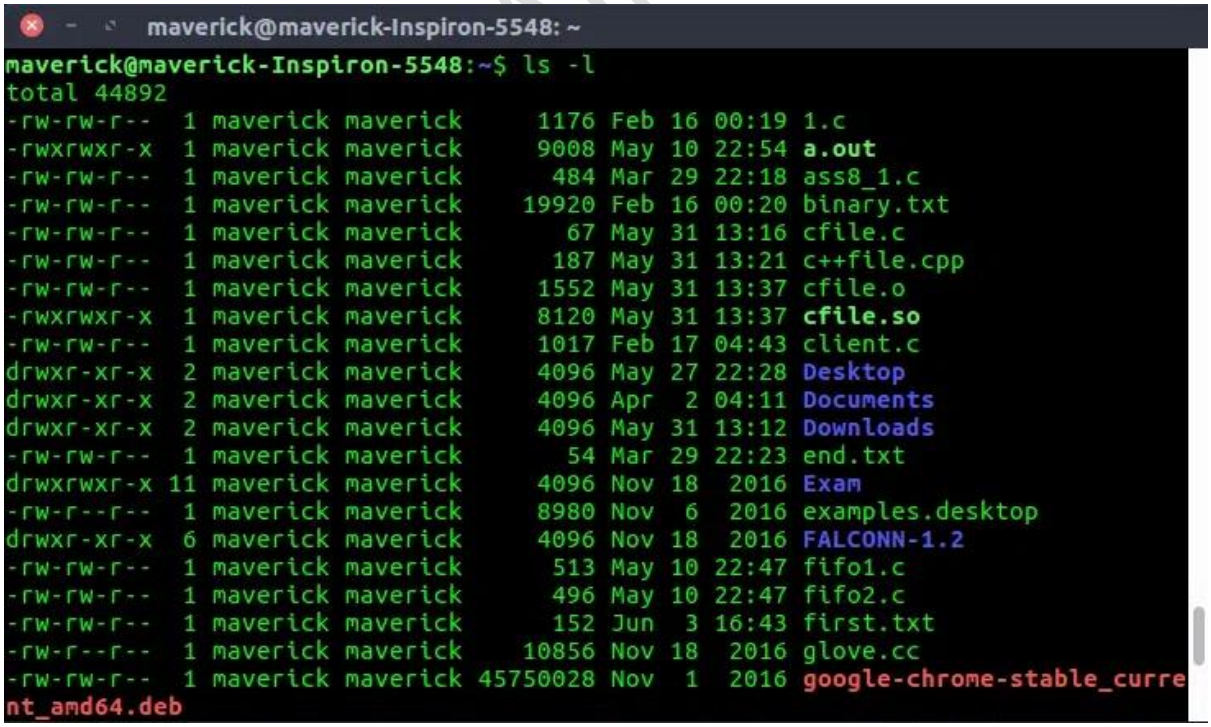
- **ls [option] [file/directory]**

Commonly Used Options in `ls` command in Linux

Options	Description
-l	known as a long format that displays detailed information about files and directories.
-a	Represent all files Include hidden files and directories in the listing.
-t	Sort files and directories by their last modification time, displaying the most recently modified ones first.
-r	known as reverse order which is used to reverse the default order of listing.
-S	Sort files and directories by their sizes, listing the largest ones first.
-R	List files and directories recursively, including subdirectories.
-i	known as inode which displays the index number (inode) of each file and directory.
-g	known as group which displays the group ownership of files and directories instead of the owner.
-h	Print file sizes in human-readable format (e.g., 1K, 234M, 2G).
-d	List directories themselves, rather than their contents.

13.1 Display All Information About Files/Directories Using `ls -l`

To show long listing information about the file/directory.\



ls -l

-rw-rw-r– 1 maverick maverick 1176 Feb 16 00:19 1.c 1st Character – File Type: First character specifies the type of the file. In the example above the hyphen (-) in the 1st character indicates that this is a normal file. Following are the possible file type options in the 1st character of the ls -l output.

Field Explanation

- – normal file
- d : directory
- s : socket file
- l : link file

Field 1 – File Permissions: The next characters specify the permission of the file. Every 3 characters specify read, write, and execute permissions for the user(root), group, and others respectively in order. Taking the above example, -rw-rw-r– indicates read-write permission for the user(root), read permission for the group, and no permission for others respectively. If all three permissions are given to the user(root), group and others, the format looks like -rwxrwxrwx

Field 2 – Number of links: The second field specifies the number of links for that file. In this example, 1 indicates only one link to this file.

Field 3 – Owner: The third field specifies the owner of the file. In this example, this file is owned by the username ‘maverick’.

Field 4 – Group: The fourth field specifies the group of the file. In this example, this file belongs to” maverick’ group.

Field 5 – Size: The fifth field specifies the size of the file in bytes. In this example, ‘1176’ indicates the file size in bytes.

Field 6 – Last modified date and time: The sixth field specifies the date and time of the last modification of the file. In this example, ‘Feb 16 00:19’ specifies the last modification time of the file.

Field 7 – File name: The last field is the name of the file. In this example, the file name is 1.c.

14. mv command in Linux

The mv, aka the **move** command is the most common way to move files and directories in Linux. Here’s the syntax:

$ mv [SOURCE] [DESTINATION]

15.1. Moving Multiple Files or Directories

To move multiple files or directories in Linux, you need to specify all the SOURCE files or directories and the DESTINATION directory where they will be moved to. It's important to note that when moving multiple files or directories, the DESTINATION must be a directory, not a file.

$ mv SOURCE1 SOURCE2 SOURCE3 ... DESTINATION

```
sam@sam-VirtualBox:~/Public/directory1$ mv file1 file2 file3 new_dir
sam@sam-VirtualBox:~/Public/directory1$
```

With the **mv** command in Linux, you can also employ **pattern matching** to move files. To illustrate, we can move all **.pdf** files from the current directory to the *~/Documents* directory using the following command:

```
sam@sam-VirtualBox:~/Public/directory1$ mv *.pdf ~/Documents
sam@sam-VirtualBox:~/Public/directory1$
```

-i option

This option stands for "interactive". If you use the **-i** option, the **mv** command will prompt you before overwriting any existing files with the same name as the file we are moving.

```
sam@sam-VirtualBox:~/Public/directory1$ mv -i example.txt new_folder
mv: overwrite 'new_folder/example.txt'? y
sam@sam-VirtualBox:~/Public/directory1$
```

-f option

This option stands for "force". The **-f** option allows you to overwrite any existing files with the same name as the file you are moving without being prompted. Use this option with caution, as it can result in data loss if used improperly.

```
sam@sam-VirtualBox:~/Public/directory1$ mv -f example.txt new_folder
sam@sam-VirtualBox:~/Public/directory1$
```

-v option

This option stands for "verbose". If you use the -v option, the mv command will display a message for each moved file or directory, providing more detailed information about the process.

```
sam@sam-VirtualBox:~/Public/directory1$ mv -v example.txt new_folder
renamed 'example.txt' -> 'new_folder/example.txt'
sam@sam-VirtualBox:~/Public/directory1$
```

The **-v** option can be especially useful when moving multiple files or directories, as it allows we to see a clear output of the process and confirm that everything has been moved correctly.

-n option

This option stands for "no clobber". The -n option prevents the mv command from overwriting any existing files with the same name as the file we are moving. If a file with the same name already exists in the destination directory, the mv command will simply skip that file.

```
sam@sam-VirtualBox:~/Public/directory1$ mv -n example.txt new_folder
sam@sam-VirtualBox:~/Public/directory1$
```

-u option

This option stands for "update". If you use the -u option, the mv command will only move files newer than those in the destination directory. This can be useful when you want to update files in one directory with newer versions from another directory.

```
sam@sam-VirtualBox:~/Public/directory1$ mv -u example.txt new_folder
sam@sam-VirtualBox:~/Public/directory1$
```

EX.NO- 1: SHELL PROGRAMMING BASICS

AIM: To execute the fundamentals of shell programming such as control flow statements.

SHELL SCRIPTS (MULTITASKING)

- In order to solve the problems of shell command, the shell programming is introduced here
- Doing more than one job at a time (multitasking)
- It is also called as shell programming

VARIABLES SECTION

- Names given to the memory location
- Shell program supports dynamic data typed system which means that no need to use specific data type for variables declaration

Syntax: Variable-Name=Initial-Value

Example:

```
a=10                    # integer type
str=”Sachin”            # string type
c=’G’                   # character type
f=true                  # boolean value
readonly id=14          # integer constant
```

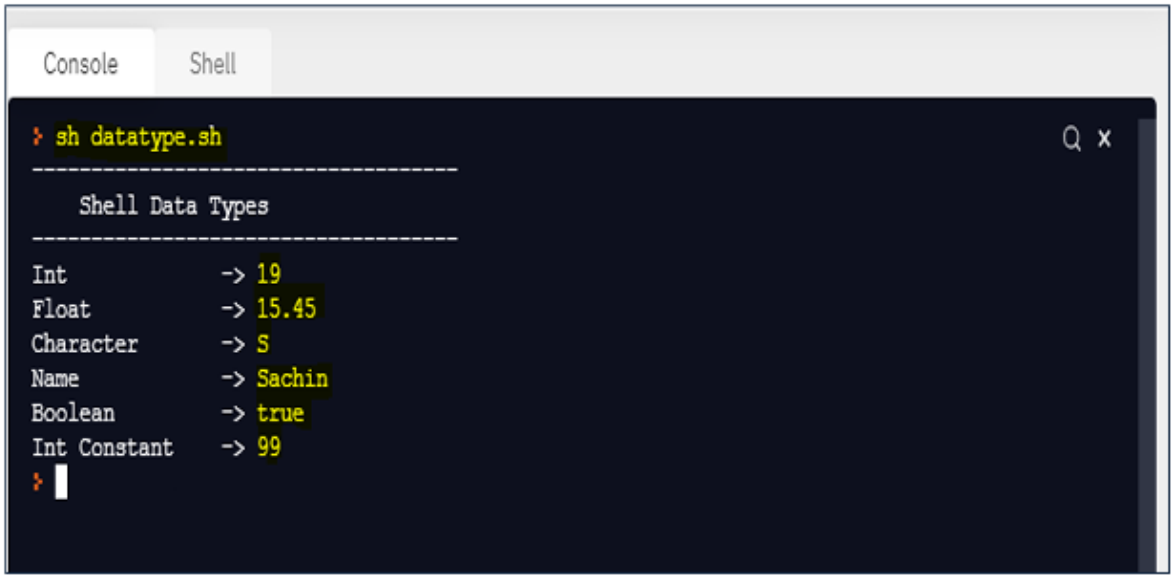
EXAMPLE OF DATA TYPES IN SHELL CODE

SOURCE CODE

```
echo "-----------------------------------"
echo -e "\tShell Data Types"
echo "-----------------------------------"
# variables definition
a=19
b=15.45
c='S'
str="Sachin"
flag=true
```

```
# constant variable definition

readonly id=99

echo  -e "Int\t\t\t\t-> $a"

echo -e "Float\t\t\t-> $b"

echo -e "Character\t\t-> $c"

echo -e "Name\t\t\t-> $str"

echo -e "Boolean\t\t\t-> $flag"

echo -e "Int Constant\t-> $id"
```

2. OUTPUT



Single line statement

- The single-line statement is indicated by the '#' symbol in the shell program Example.

Example:

```
# This is single line statement
```

Multi-line statements

- It is used to ignore more than one statements
- This is indicated by :' ' symbol in the shell program

Example:

```
:' Variable Declarations

a=20
```

k=25 ‘

SELECTION STATEMENTS

1. Simple If statement

2. If else Statement

3. If else if Statement

4. Case Statement

1. Simple If Statement Syntax

```
if [ condition ]
then
    true statement
 fi
```

It is important to note that, the space should be given before and after the operator symbol [

2. If else Statement Syntax

```
if [ condition ]
then
        true statement
else
        false statement
fi
```

3. If…elif…else Statement

```
if [ condition ]
then
        true statement
 elif [ condition ]
then
        true statement
else
        false statement
fi
```

It is important to note that, the simple if, if else and if-elif-else should be closed by fi keyword.

1. **Case Statement**
   - It is equivalent to switch case statements in c language
   - It is used to execute several statements based on the value of expression
   - This is done by using the reserved word case
   - It is an alternative option for if..elif..else statements

```
case < variable>in
Pattern 1)
        Commands / statements
        ;;
 Pattern 2)
        Commands / statements
        ;;
        …
easc
```

Where, ";;" represents the break part in case statements

LOOPING STATEMENTS

1. While loop (while)
2. Until loop (until)
3. For loop (for)

A. While loop

```
while [ condition ]
do
        true statement
done
```

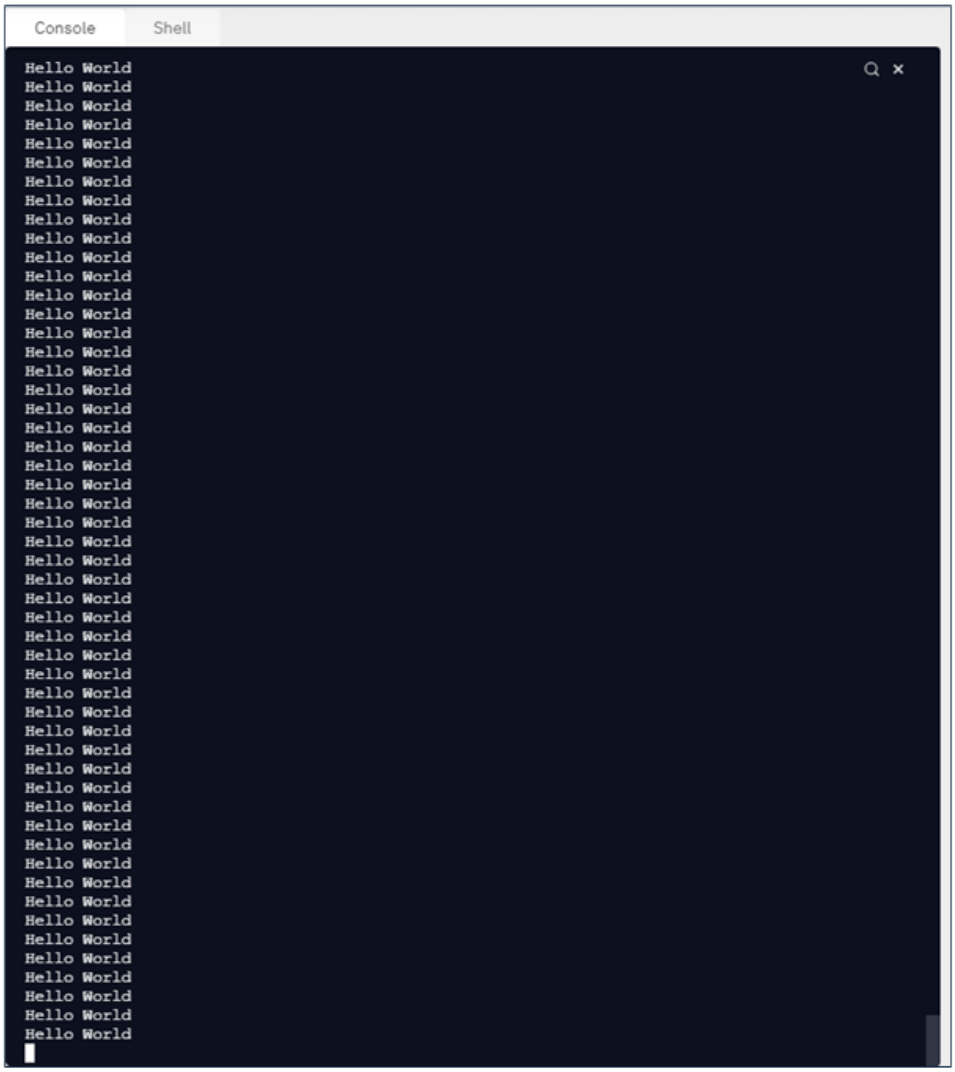
Infinite While loop

- It is important to note that, the colon (:) operator or true keyword is used for creating an infinite loop
- The colon (:) operator is used instead of the operator symbols [ ]

EXAMPLE OF INFINITE LOOP USING WHILE LOOP

SOURCE CODE

```
while :
do
        echo "Hello World"
done
                (OR)
while true
do
        echo "Hello World"
done
```



2. Until loop

- Here loop is executing until the condition is false
- If the condition becomes true it will exit from the loop

until [condition]

do

true statement

done

Example:

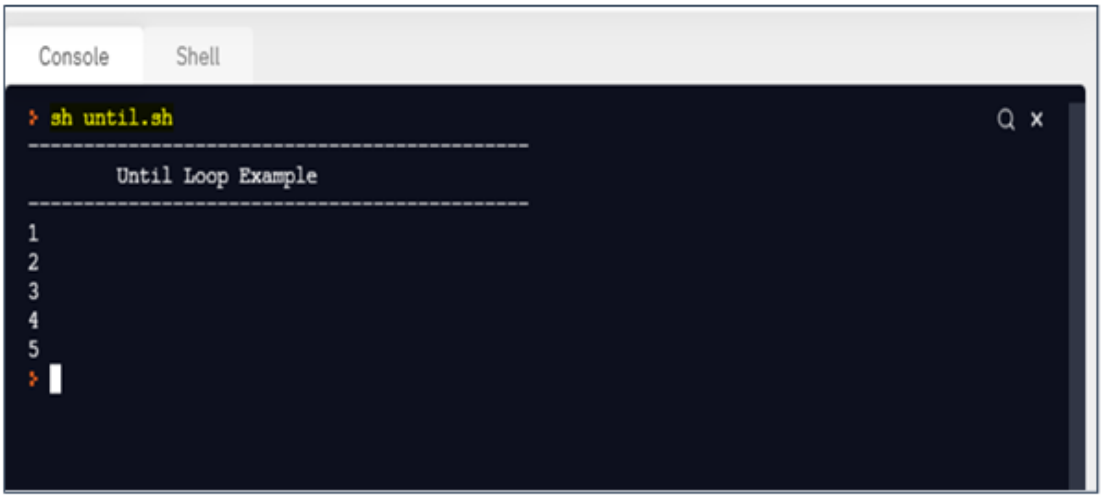
SOURCE CODE

```
echo "-------------------------------------------"
echo "\t\tUntil Loop Example"
echo "-------------------------------------------"
i=1
until [ $i -gt 5 ]
do
echo $i
        i=`expr $i + 1`
done
```

Here the looping statements are executed until the condition becomes fail like 1>5, 2>5, 3>5, 4>5, 5>5

This loop will terminate whenever the condition becomes true like 6>5

OUTPUT



DISPLAYING FILES AND DIRECTORIES USING FOR LOOP

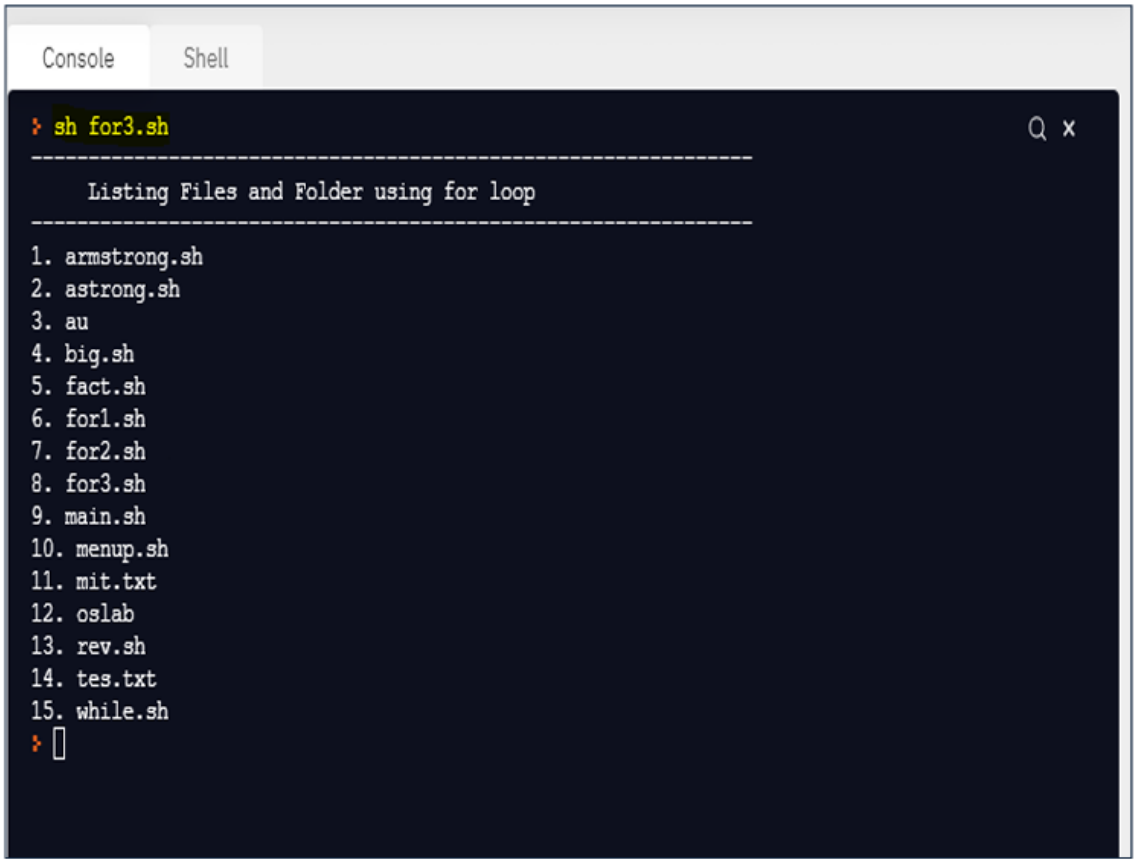
```
echo "------------------------------------------------------------"
echo -e "\t Listing Files and Folder using for loop"
echo "------------------------------------------------------------"
# get all the files and store them to variable
fset=`ls`
k=1
# loop the variable fset
for i in $fset
do
        echo "$k. $i"
        k=`expr $k + 1`
done
```

Store list of files to the variable fset using ls command.

Command Substitution: Storing the output of the command to a user defined variable. This is done by using the operator `command` or $(command)

k=k+1 or k++

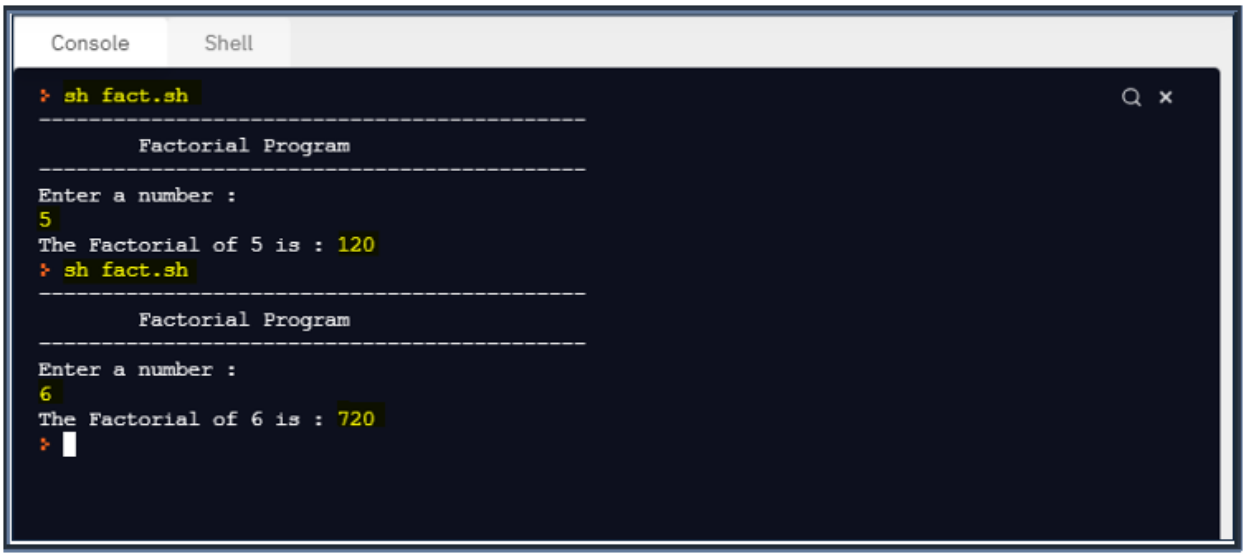
OUTPUT



FACTORIAL OF A NUMBER

```
echo "-----------------------------------------"
echo "\t\tFactorial Program"
echo "-----------------------------------------"
echo "Enter a number: "
read n
i=0
f=1
while [ $i -lt $n ]
        do
                # increment i by 1
                i=`expr $i + 1`
                f=`expr $f \* $i`
        done
echo "The Factorial of $n is : $f"
```

OUTPUT



REVERSE OF A NUMBER

```
echo "-----------------------------------------"
echo "\t\tReverse of Number"
echo "-----------------------------------------"
echo "Enter a number: "
read n
duplicate=$n
res=0
while [ $n -ne 0 ]
        do
                # find the reminder
                rem=`expr $n % 10`
                # multiply reverse number with 10
                res=`expr $res \* 10`
                # add the resultant number with a remainder number
                res=`expr $res + $rem`
                # divide n by 10
                n=`expr $n / 10`
        done
echo "The Reverse Number of $duplicate is $res"
```

Read a number

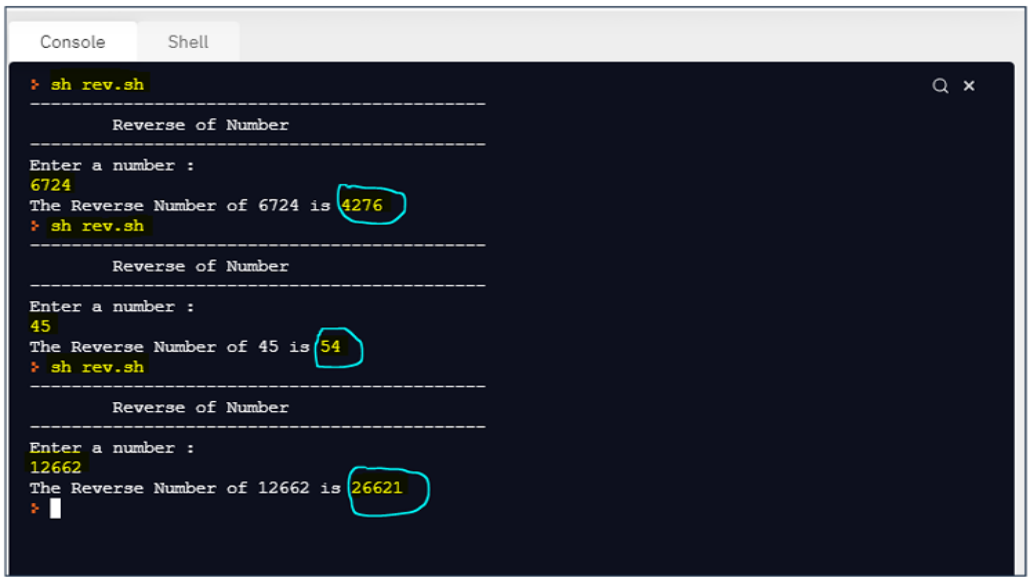
while (n!=0)

rem=n%10

res=res*10

n=n/10

OUTPUT



MENU DRIVEN PROGRAM

```
while [ true ]
do
        echo "-------------------------------------"
        echo "\t\t Menu Program"
        echo "-------------------------------------"
        echo  -e  "1. View Files \t 2.Date"
        echo "3. Users List \t4.Calendar"
        echo "5. Exit"
        echo "\tEnter ur choice : "
read ch
#switch case
        case $ch in
        1) ls;;
        2) date;;
        3) w;;
        4) cal;;
        5) exit;;
        esac
# read choice from user for the continuation of program execution
echo "Do you want to continue: Press Yes/No"
read ch
        if [ $ch = "yes" ] || [ $ch = "Yes" ] || [ $ch = "YES" ]
        then
        continue
        else
        exit
        fi
done
```

N.B.: It is important to note that, the single equal operator (=) is used for string comparison and the symbol –eq or == is used for number comparison in the shell program.

OUTPUT


Console
Shell
$ sh menup.sh
Menu Program
1. View Files 2.Date
3. Users List 4.Calendar
5. Exit
Enter ur choice :
1
armstrong.sh au fact.sh menup.sh oslab tes.txt
astrong.sh big.sh main.sh mit.txt rev.sh
Do you want to continue : Press Yes/No
Yes
Menu Program
1. View Files 2.Date
3. Users List 4.Calendar
5. Exit
Enter ur choice :
3
05:10:41 up 1:47, 0 users, load average: 8.98, 7.29, 6.29
USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT
Do you want to continue : Press Yes/No
yes
Menu Program
1. View Files 2.Date
3. Users List 4.Calendar
5. Exit
Enter ur choice :
4
April 2021
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
Do you want to continue : Press Yes/No
YES
Menu Program
1. View Files 2.Date
3. Users List 4.Calendar
5. Exit
Enter ur choice :
2
Thu Apr 29 05:10:57 UTC 2021
Do you want to continue : Press Yes/No
Do you want to continue : Press Yes/No
yes
Menu Program
1. View Files 2.Date
3. Users List 4.Calendar
5. Exit
Enter ur choice :
5
$

Basic Shell Scripts with "for" Loop:

Print Numbers from 5 to 1

```
#!/bin/bash
for ((i=5; i>=1; i--))
do
echo $i
done
```

OUTPUT

```
5
4
3
2
1
```

Print Even Numbers From 1 to 10

```
#!/bin/bash
for (( i=1; i<=10; i++ ))
do
if [ $((i%2)) == 0 ]
then
echo $i
fi
done
```

OUTPUT

```
2
4
6
8
10
```

Print the Multiplication Table of a Number.

```
#!/bin/bash
read -p "Enter a number: " num
for (( i=1; i<=10; i++ ))
do
echo "$num x $i = $((num*i))"
done
```

OUTPUT

```
Enter a number: 12
12 x 1 = 12
12 x 2 = 24
12 x 3 = 36
12 x 4 = 48
12 x 5 = 60
12 x 6 = 72
12 x 7 = 84
12 x 8 = 96
12 x 9 = 108
12 x 10 = 120
```

Loop Through a String Character-by-Character

```
#!/bin/bash
read -p "Enter a string: " str
for ((i=0; i<${#str}; i++)); do
echo ${str:i:1}
done
```

OUTPUT

```
Enter a string: Linux
L
i
n
u
x
```

Loop Through Array Elements

For accessing each array element you can use the **for loop** in the following manner. Indicate the desired array using "**${ARRAY_NAME[@]}**" and access each item stored in the array.

```
#!/bin/bash
arr=("mango" "grape" "apple" "cherry" "orange")
for item in "${arr[@]}"; do
echo $item
done
```

OUTPUT

```
mango
grape
apple
cherry
orange
```

Calculate the Factorial of a Number

```
#!/bin/bash
read -p "Enter a number: " num
temp=1
for (( i=1; i<=$num; i++ ))
do
temp=$((temp*i))
done
echo "The factorial of $num is: $temp"
```

OUTPUT

```
Enter a number: 6
The factorial of 6 is: 720
```

Calculate the Sum of the First "n" Numbers

To calculate the sum of the first n numbers run a for loop and addition operation till n.

```
#!/bin/bash
read -p "Enter a number: " num
sum=0
for (( i=1; i<=$num; i++ ))
do
sum=$((sum + i))
done
echo "Sum of first $num numbers: $sum"
```

OUTPUT

```
Enter a number: 100
Sum of first 100 numbers: 5050
```

Find the Smallest and Largest Elements in an Array

First, initialize a small number and a large number. Then compare the array elements with these numbers inside any loop.

```
#!/bin/bash
arr=(24 27 84 11 99)
echo "Given array: ${arr[*]}"
s=100000
l=0
for num in "${arr[@]}"
        do
        if [ $num -lt $s ]
        then
        s=$num
        fi
if [ $num -gt $l ]
        then
        l=$num
fi
```

```
done
echo "The smallest element: $s"
echo "The largest: $l"
```

OUTPUT

```
Given array: 24 27 84 11 99
The smallest element: 11
The largest: 99
```

Calculate the Average of an Array of Numbers

Find the sum of array elements using a "**for**" loop and divide it by the number of elements i.e. **${#arr[@]}**.

```
#!/bin/bash
echo "Enter an array of numbers (separated by space):"
read -a arr
sum=0
for i in "${arr[@]}"
do
sum=$((sum+i))
done
avg=$((sum/${#arr[@]}))
echo "Average of the array elements: $avg"
```

OUTPUT

```
Enter an array of numbers (separated by space):
23 45 11 99 100
Average of the array elements: 55
```

Task-Specific Shell Scripts with "for" Loop

In addition to the conceptual bash scripts, in this section, you will find some task-specific script examples. These scripts are mostly related to the regular process that you run on your system. Hence, follow the examples below to improve your experience with **Shell Scripting:**

Take Multiple Filenames and Prints Their Contents

The below script is for reading the contents of multiple files. It will take the file names as user input and display their contents on the screen. If any filename does not exist, it will show a separate error message for that file.

```
#!/bin/bash
read -p "Enter the file names: " files
IFS=' ' read -ra array <<< "$files"
for file in "${array[@]}"
do
        if [ -e "$file" ]; then
        echo "Contents of $file:"
        cat "$file"
        else
        echo "Error: $file does not exist"
        fi
done
```

OUTPUT

```
Enter the file names: message.txt passage.txt
Contents of message.txt:
“Merry Christmas! May your happiness be large and your bills be small.”
Contents of passage.txt:
The students told the headmaster that they wanted to celebrate the victory of the
National Debate Competition.
```

Read Lines from a File

The following script can be used to read and display each line from a file. Here, a filename is taken as user input and the IFS(Internal Field Separator) is set to New Line (\n) which enables the for loop to recognize each line individually inside the file.

```
#!/bin/bash
read -p "Enter a filename: " file
echo Lines:
IFS=$'\n'
        for line in $(cat "$file"); do
        echo "$line"
        done
```

OUTPUT

```
Enter a filename: textfile.txt
Lines:
I wandered lonely as a cloud
That floats on high o'er vales and hills,
When all at once I saw a crowd,
A host, of golden daffodils;
```

Loop Through Files with a Specific Extension

The given script takes a file extension as user input and looks for the files with that extension within the current directory using for loop. Inside the loop, it prints each file name.

```
#!/bin/bash

read -p "Enter a file extension (i.e. txt, jpg, ..): " ext

for file in *.$ext; do
        echo $file

done
```

OUTPUT

```
Enter a file extension (i.e. txt, jpg, ..): txt
file1.txt
file2.txt
textfile.txt
urls.txt
```

Loop Through Files in Multiple Directories

```
#!/bin/bash

read -p "Enter a list of directories: " directories

for dir in $directories; do
        for file in $dir/*;
        do
        echo $file
        done

done
```

OUTPUT

```
Enter a list of directories: Documents Pictures
Documents/list1.txt
Documents/list2.txt
Documents/message.txt
Documents/packets.pcap
Documents/ping.txt
Documents/poem.txt
Pictures/Screenshots
Pictures/ss1.png
Pictures/ss2.png
```

Organizes Files in a Directory Based on Their File Types

The script given below organizes files in a directory depending on their type. The user needs to give a destination directory path to organize the files along with the source directory path.

This script will create five directories: 1) Documents, 2) Images, 3) Music, 4) Videos, and 5) Others only if they do not already exist on the destination path. Then, it will check all the files and their extension and move them to the corresponding directory. If there is any unknown file extension, then the script will move the file to the Others Directory.

```
#!/bin/bash
# Specify the source and destination directories
read -p "Enter the path to the source directory: " source_dir
read -p "Enter the path to the destination directory: " dest_dir

# Create the destination directories if they don't exist
        mkdir -p "${dest_dir}/Documents"
        mkdir -p "${dest_dir}/Images"
        mkdir -p "${dest_dir}/Music"
        mkdir -p "${dest_dir}/Videos"
        mkdir -p "${dest_dir}/Others"

# Move files to the appropriate directories based on their extensions
for file in "${source_dir}"/*; do
        if [ -f "${file}" ]; then
        extension="${file##*.}"
        case "${extension}" in
        txt|pdf|doc|docx|odt|rtf)
        mv "${file}" "${dest_dir}/Documents"
        ;;
```

```
        jpg|jpeg|png|gif|bmp)
        mv "${file}" "${dest_dir}/Images"
        ;;
        mp3|wav|ogg|flac)
        mv "${file}" "${dest_dir}/Music"
        ;;
        mp4|avi|wmv|mkv|mov)
        mv "${file}" "${dest_dir}/Videos"
        ;;
        *)
        mv "${file}" "${dest_dir}/Others"
        ;;
        esac
fi
done
echo "Files organized successfully!"
```

OUTPUT

```
Enter path to the source directory: /home/anonnya/Downloads
Enter path to the destination directory: /home/anonnya/Downloads_Organized
Files organized successfully!
```

Loop Through Command Output

```
#!/bin/bash
read -p "Enter a command: " comm
for result in $($comm); do
echo $result
done
```

OUTPUT

```
Enter a command: ls Documents
list1.txt
list2.txt
message.txt
packets.pcap
ping.txt
poem.txt
```

Command substitution-file copying between directories

```
echo "--------------------------------------------------"
echo "\t\tCopying Files B/W Directories"
echo "--------------------------------------------------"
# path of source directory
path="/home/runner/OS-Lab/d1"
# k=$(ls $path)
src=`ls $path`
# path of target directory
tar="/home/runner/OS-Lab/d5"
# for loop
for i in $src
do
        cp $i $tar
        echo "$i is successfully copied to $tar/$i"
done
echo "--------------------------------------------------"
```

Command Substitution using the symbol ` `

Path of Source Directory (d1). Parent Path can be get using pwd command.

Path of Target Directory (d5).

OUTPUT

```
Console  Shell
$ mkdir d5
$ ls d5
$ sh test.sh
--------------------------------------------------
        Copying Files B/W Directories
--------------------------------------------------
fact.sh is successfully copied to /home/runner/OS-Lab/d5/fact.sh
hh.txt is successfully copied to /home/runner/OS-Lab/d5/hh.txt
--------------------------------------------------
$ ls d5
fact.sh  hh.txt
$ ls d1
fact.sh  hh.txt
$
```

File type operators

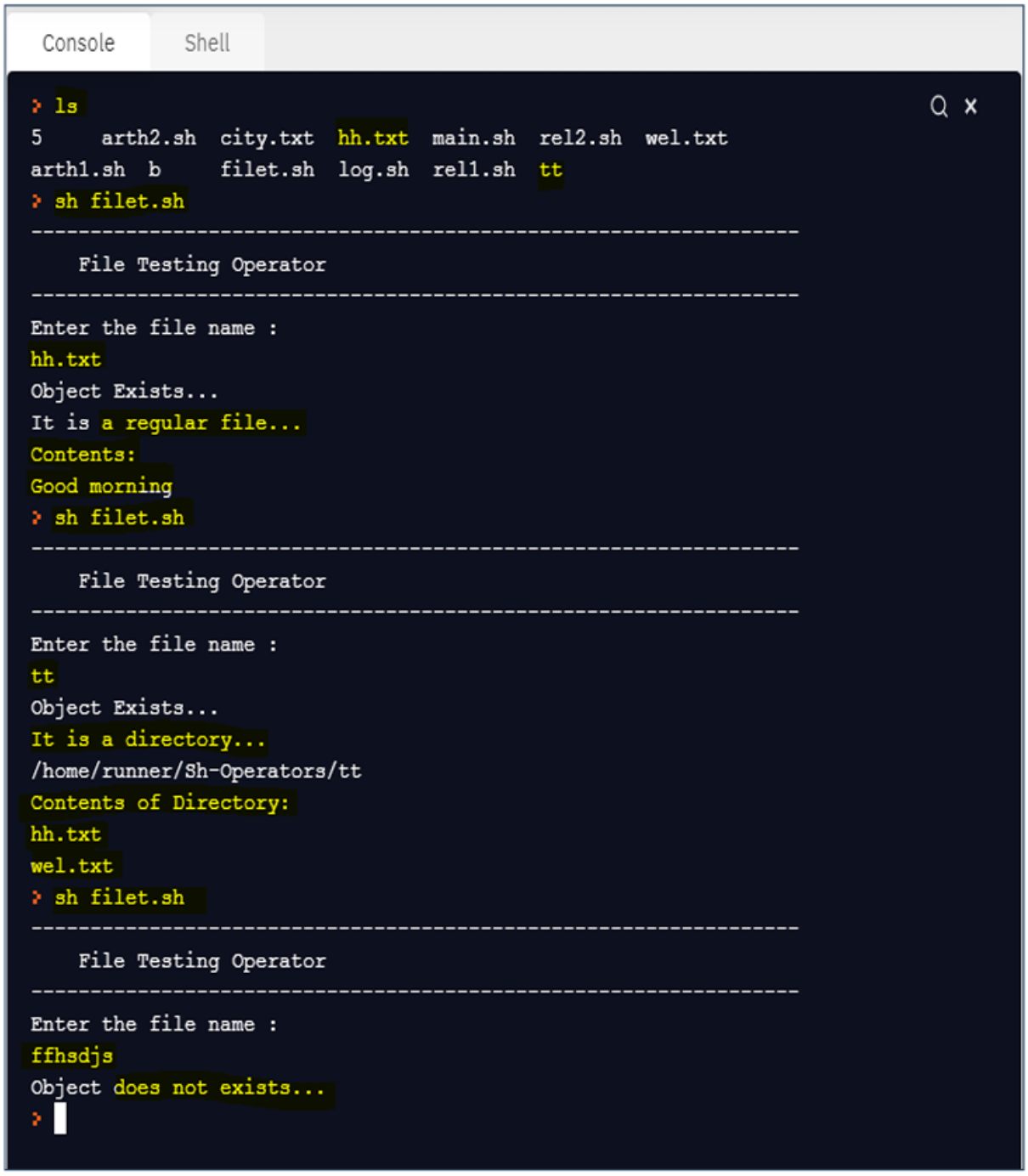
S.N	OPERATOR	DESCRIPTION
1.	-f	Returns true if exists and if it is a regular file (.txt, .c. sh, etc,…)
2.	-d	Returns true if exists and if it is a directory
3.	-e	Returns true if exists
4.	-z	Returns true if file is empty (file has zero length)
5.	-r	Returns true if exists and is readable mode
6.	-w	Returns true if exists and is writable mode
7.	-x	Returns true if exists and is executable mode

Example of file test operators

```
echo "------------------------------------------------------------------"
echo "\tFile Testing Operator"
echo "------------------------------------------------------------------"
echo "Enter the file name : "
read fp
if [ -e $fp ]
then
   echo "Object Exists..."
   if [ -f $fp ]
   then
      echo "It is a regular file..."
      echo "Contents:"
      echo $(cat $fp)
   elif [ -d $fp ]
   then
      echo "It is a directory..."
      dpath=`pwd`/$fp
      echo $dpath
      echo "Contents of Directory: "
      for i in $(ls $dpath)
```

```
        do
            echo $i
        done
    else
        echo "It is a special file..."
    fi
else
    echo "Object does not exists..."
fi
```

OUTPUT



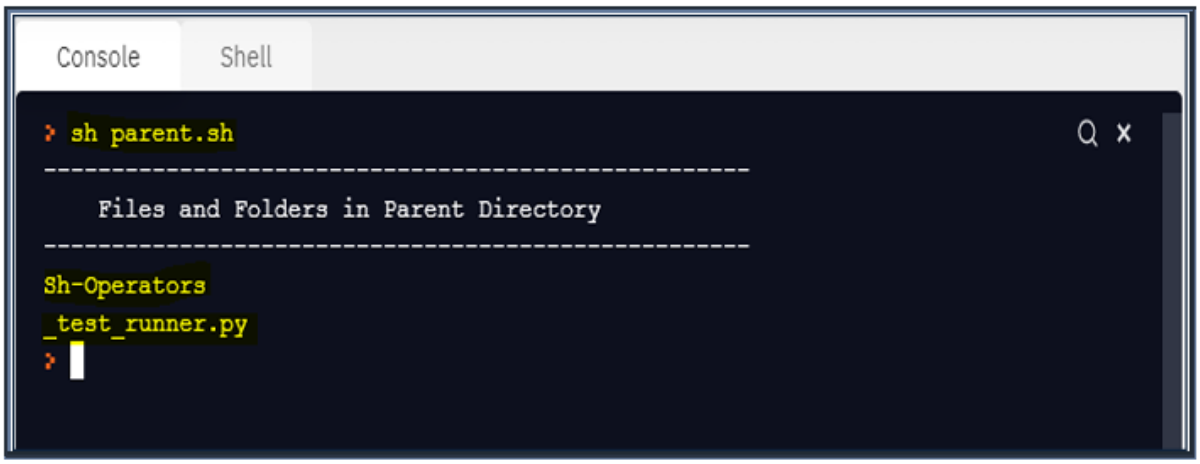
Directories

S.N	OPERATOR	DESCRIPTION
1.	ls (OR) ls .	Shows the list of files and folders in current directory
2.	ls ..	Shows the list of files and folders in parent directory
3.	ls /	Shows the list of files and folders in root working directory
4.	ls -l	Shows the files and folders in long listing format
5.	ls -s	Shows the size of files and folders in current directory

Listing files and folders in parent directory

```
echo "--------------------------------------------------"
echo "\tFiles and Folders in Parent Directory"
echo "--------------------------------------------------"
for i in $(ls ..)
do
   echo $i
done
```

OUTPUT



Relational operators

Relational operators for numbers [using $()]

S.N	OPERATOR	DESCRIPTION
1.	==	Equal
2.	!=	Not Equal
3.	<	Lesser than
4.	<=	Lesser than or Equal to
5.	>	Greater than
6.	>=	Greater than or Equal to

Numeric comparison operators for numbers

S.N	OPERATOR	DESCRIPTION
1.	-eq	Equal
2.	-ne	Not Equal
3.	-gt	Greater than
4.	-ge	Greater than or Equal to
5.	-lt	Lesser than
6.	-le	Lesser than or Equal to

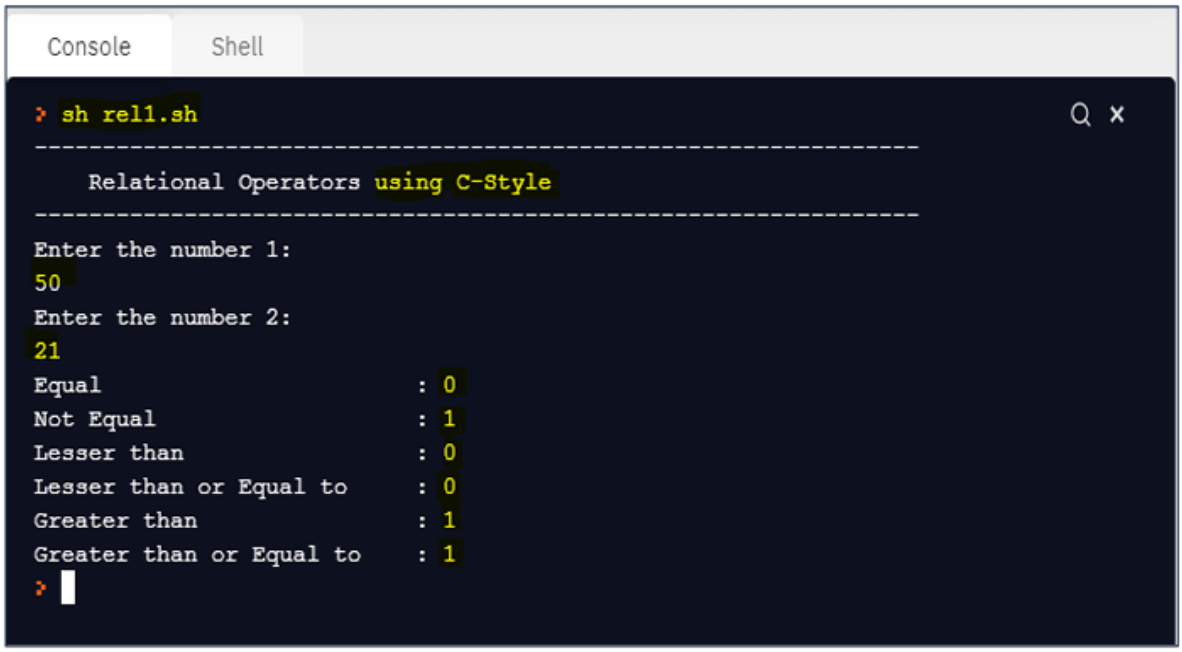
Relational operators for strings [using if]

S.N	OPERATOR	DESCRIPTION
1.	=	Equal
2.	!=	Not Equal
3.	\<	Lesser than
4.	\>	Greater than

Example of relational operators using c style

```
echo "------------------------------------------------------------------"
echo "\tRelational Operators using C-Style"
echo "------------------------------------------------------------------"
        echo "Enter the number 1: "
        read a
        echo "Enter the number 2: "
        read b
# performing relational operators using C style $()
        r1=$((a==b))
        r2=$((a!=b))
        r3=$((a<b))
        r4=$((a<=b))
        r5=$((a>b))
        r6=$((a>=b))
# print the relational results
        echo "Equal \t\t\t\t\t: $r1"
        echo "Not Equal \t\t\t\t: $r2"
        echo "Lesser than \t\t\t\t: $r3"
        echo "Lesser than or Equal to \t: $r4"
        echo "Greater than  \t\t\t\t: $r5"
        echo "Greater than or Equal to \t: $r6"
```

OUTPUT



Example of relational operators for strings

```
echo "------------------------------------------------------------------"
echo -e  "\tRelational Operators for Strings"
echo "------------------------------------------------------------------"
        echo "Enter the name 1: "
        read a
        echo "Enter the name 2: "
        read b
# performing relational operators for strings
if [ $a = $b ]
then
   echo "----------Equal Results-----------------"
   echo "Both Strings are Equal"
else
   echo "----------Equal Results-----------------"
   echo "Both Strings are NOT Equal"
fi
if [ $a != $b ]
then
   echo "----------NOT Equal Results-----------------"
   echo "Both Strings are NOT Equal"
else
   echo "----------NOT Equal Results-----------------"
   echo "Both Strings are Equal"
fi
if [ $a \> $b ]
then
   echo "----------Greater Results-----------------"
   echo "$a is Greater than $b"
else
   echo "----------Greater Results-----------------"
   echo "$b is Greater than $a"
fi
if [ $a \< $b ]
then
   echo "----------Lesser Results-----------------"
   echo "$a is Lesser than $b"
else
   echo "----------Lesser Results-----------------"
   echo "$b is Lesser than $a"
fi
```

OUTPUT


Console
Shell
> sh rel2.sh
Relational Operators for Strings
Enter the name 1:
Sachin
Enter the name 2:
Sachin
----------Equal Results----------
Both Strings are Equal
----------NOT Equal Results----------
Both Strings are Equal
----------Greater Results----------
Sachin is Greater than Sachin
----------Lesser Results----------
Sachin is Lesser than Sachin
> sh rel2.sh
Relational Operators for Strings
Enter the name 1:
Rohit
Enter the name 2:
Sachin
----------Equal Results----------
Both Strings are NOT Equal
----------NOT Equal Results----------
Both Strings are NOT Equal
----------Greater Results----------
Sachin is Greater than Rohit
----------Lesser Results----------
Rohit is Lesser than Sachin
>

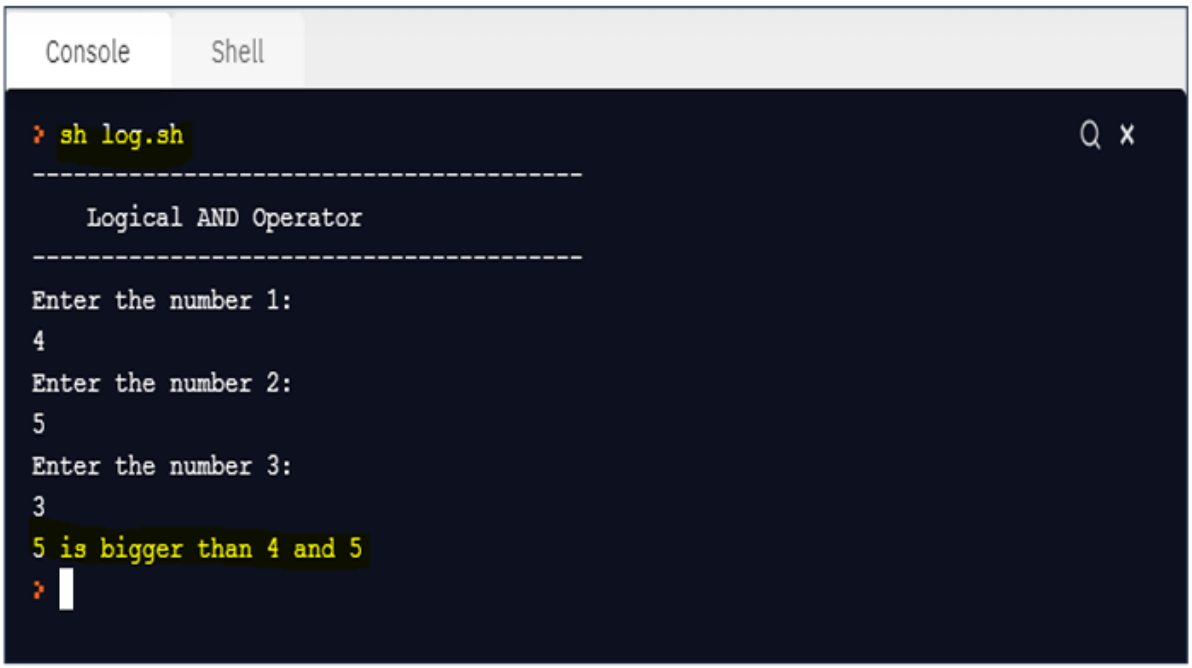
LOGICAL OPERATORS (BOOLEAN OPERATORS)

S.N	OPERATOR	OPERATORS IN SHELL	DESCRIPTION
1.	Logical AND	&&	Binary operator which returns true if both the operands are true otherwise returns false value
2.	Logical OR	||	Binary operator which returns true if one of the operand or both is true otherwise returns false value
3.	Not Equal to	!	Unary operator returns true if the operand is false and returns true if the operand is true

Example of logical and operator

```
echo "---------------------------------------"
echo "\tLogical AND Operator"
echo "---------------------------------------"
echo "Enter the number 1: "
read a
echo "Enter the number 2: "
read b
echo "Enter the number 3: "
read c
if [ $a -gt $b ] && [ $a -gt $c ]
then
   echo "$a is bigger than $b and $c"
elif [ $b -gt $c ]
then
    echo "$b is bigger than $a and $b"
else
fi
```

OUTPUT



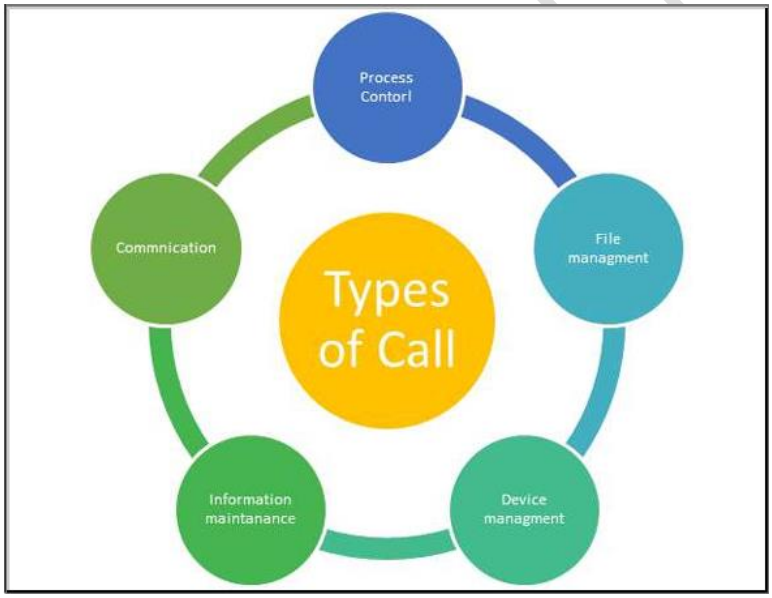
EX.NO- 4: PROCESS SYSTEM CALLS – FORK, EXIT, WAIT

AIM: To practice the system calls such as fork, wait, exit using linux c programming.

SYSTEM CALLS

- ✓ It is an interface between process and kernel
- ✓ It is way for programs to interact with OS
- ✓ Five different types of system calls are available. They are

1. Process Control
2. File Management
3. Device Management
4. Information Management
5. Communication



PROCESS CONTROL SYSTEM CALLS

- It deals with process creation, process termination, etc, …

Examples

S.N	Linux	Windows	Description
1.	fork()	CreateProcess()	Create a child process
2.	exit()	ExitProcess()	Terminate the process
3.	wait()	WaitForSignalObject()	Wait for the child process termination

fork()

- It is an important system calls which is used to create a new process in the OS
- The newly created process is called as child process and caller of the child process is called as parent process
- It takes no arguments and returns the process ID
- It is called once but returns twice (once in parent and once in the child)
- It is an important to note that, Unix / Linux will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.

It returns the following values:

- Negative ➡ new process creation was unsuccessful
- Zero ➡ returned to child process
- Positive ➡ returned to parent (caller) process

Child Process

- The newly created process is called the child process
- The return code identifies it is 0

Parent Process

- The caller of the newly created process is called the parent process
- The return code identifies it is a positive value

Formula

Total number of processes (T): 2n

Total number of Child processes (C): 2n-1

Total number of Child processes (P): T-C

Compiler: gcc compiler (gcc <filename>.c)

Execution: ./a.out (assembly output)

NOTE

- ✓ **After the fork(), both parent and child processes are running simultaneously**
- ✓ The newly created process is called the child process which is identified by the return code is 0 and the caller of the child process is called the parent process which is identified by a positive value (>0)
- ✓ **Program statements before fork() is common for both child and parent processes but after the fork() call, the rest of the program instructions will be allocated separately for the child and parent process**

WHICH PROCESS RUNS FIRST (b/w parent and child process)

- **It an important to note that, there is no rule about which process runs first between parent and child processes.**
- **As soon as a process is ready for execution (i.e. the fork system call returns), it may run according to the scheduling configuration (priority, scheduler chosen, etc.).**

EXAMPLE OF SINGLE FORK

SOURCE CODE

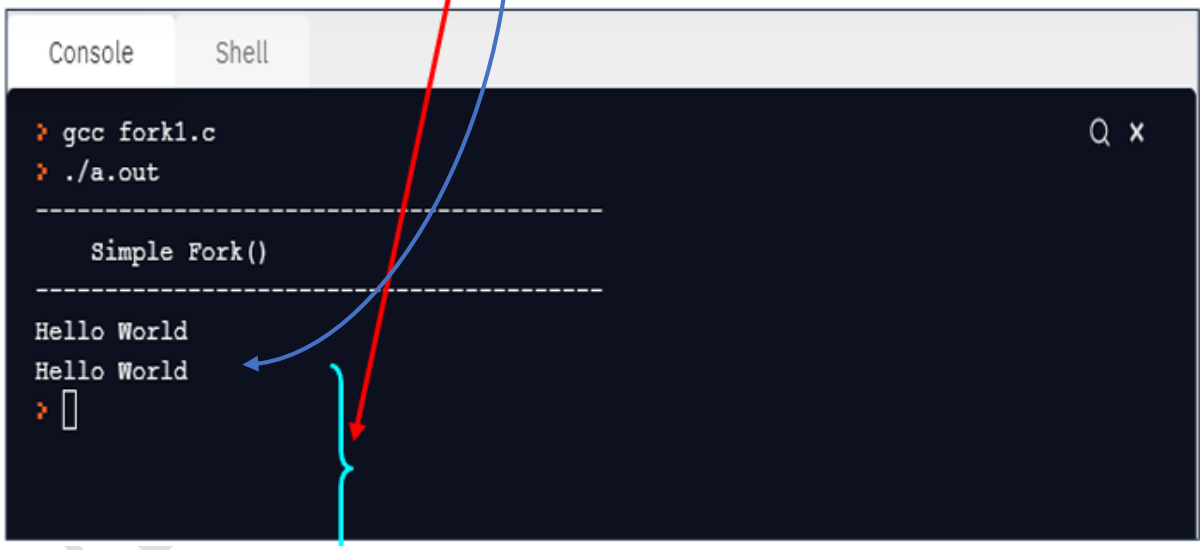
```
#include<unistd.h>
#include<stdio.h>
int main()
{
    printf("----------------------------------------\n");
    printf("\tSimple Fork()\n");
    printf("----------------------------------------\n");
// calling fork()
    fork();
    printf("Hello World\n");
    return 0;
}
```

Total Number of Processes are: 2n

2^1

2

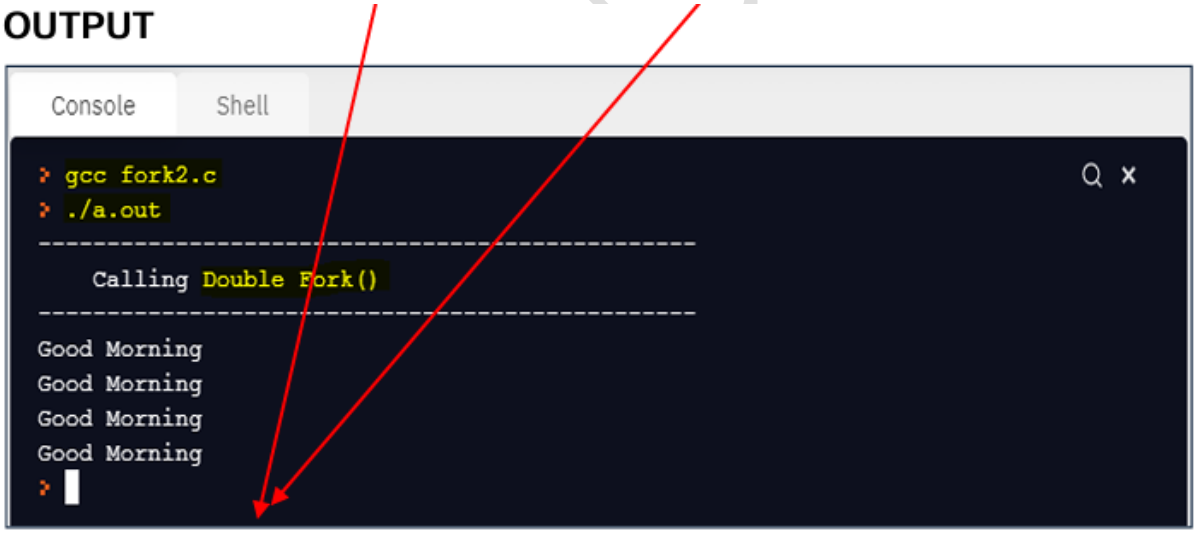
OUTPUT



CALLING DOUBLE FORK

```
#include<unistd.h>
#include<stdio.h>
int main()
{
printf("-----------------------------------------------\n");
printf("\tCalling Double Fork()\n");
printf("-----------------------------------------------\n");
// calling double fork()
fork();
fork();
printf("Good Morning\n");
return 0;
```

OUTPUT



PRINT THE RETURNED VALUE OF CHILD AND PARENT PROCESSES USING FORK

SOURCE CODE

```
#include<unistd.h>
#include<stdio.h>
#include<sys/types.h>
int main()
{
```

This header supports the pid_t

```
    pid_t id;
    printf("----------------------------------------\n");
    printf("\tReturn Code of Parent & Child\n");
    printf("----------------------------------------\n");
// calling fork()
    id=fork();
    if(id==0)
    {
       printf("Child Process is calling ...\n");
       printf("Returned Value of Child Process : %d\n",id);
    }
          else
             {
                printf("Parent Process is calling ...\n");
                printf("Returned Value of Parent Process : %d\n",id);
             }
             return 0;
          }
```

Child Process: It is identified by 0

Parent Process: It is identified by >0

OUTPUT

```
Console  Shell

> gcc rc_fork.c
> ./a.out
----------------------------------------
    Return Code of Parent & Child
----------------------------------------
Parent Process is calling ...
Returned Value of Parent Process : 1111
> Child Process is calling ...
Returned Value of Child Process : 0
```

DISPLAY THE PROCESS ID (PID) OF PARENT AND CHILD PROCESSES USING GETPID() AND GETPPID()

SOURCE CODE

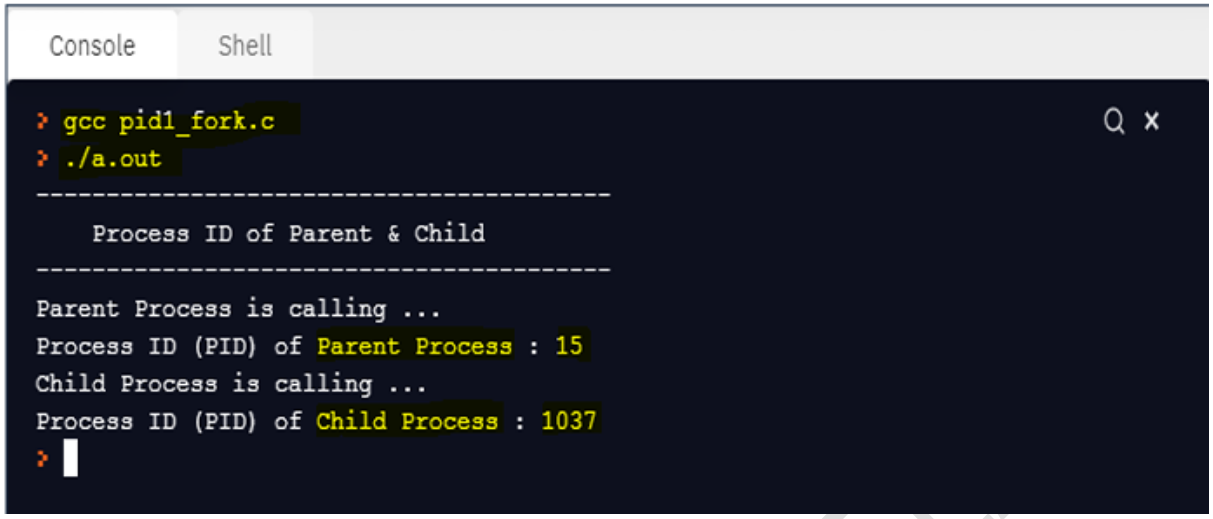
```
#include<unistd.h>
#include<stdio.h>
#include  <sys/types.h>
int main()
{
   int id;
   printf("----------------------------------------\n");
   printf("\tProcess ID of Parent & Child\n");
   printf("----------------------------------------\n");
// calling fork()
   id=fork();
   if(id==0)
   {
      printf("Child Process is calling ...\n");
      printf("Process ID (PID) of Child Process : %d\n",getpid());
   }
   else
   {
      printf("Parent Process is calling ...\n");
      printf("Process ID (PID) of Parent Process : %d\n",getppid());
   }
   return 0;
}
```

Getpid() and getppid() are built-in function and available in #include header file.

getpid() used to return the process ID of child process (newly created process).

Getppid() used to return the process ID of parent process (caller of the newly created process).

OUTPUT



Wait()

- ✓ It is a system call and available in #include<sys/wait.h> file
- ✓ It blocks the current process (calling process), until one of its child processes terminates or a signal is received
- ✓ It takes one argument which is the address of an integer variable (stores the information of the process) and returns the process ID (PID) of the completed child process.
- ✓ Return type: pid_t
- ✓ If only one child process is terminated (finished its execution), then it returns the process ID of the terminated child process
- ✓ If more than one child processes are terminated, then it returns the process ID of any terminated arbitrary child process.

The execution of wait() could have two possible situations.

1. If there is at least one child process running when the call to wait() is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.

2. If there is no child process running when the call to wait() is made, then this wait() has no effect at all. It returns -1 immediately.

NOTABLE POINTS

- wait(NULL) will block the parent process until any of its children has finished their execution (parent process will be blocked until child process returns an exit status to the operating system which is then returned to parent process)
- If child finishes before parent reaches wait(NULL) then it will read the exit status, release the process entry in the process table and continue execution until it finishes as well.
- Wait can be used to make the parent process wait for the child to terminate (finish) but not the other way around

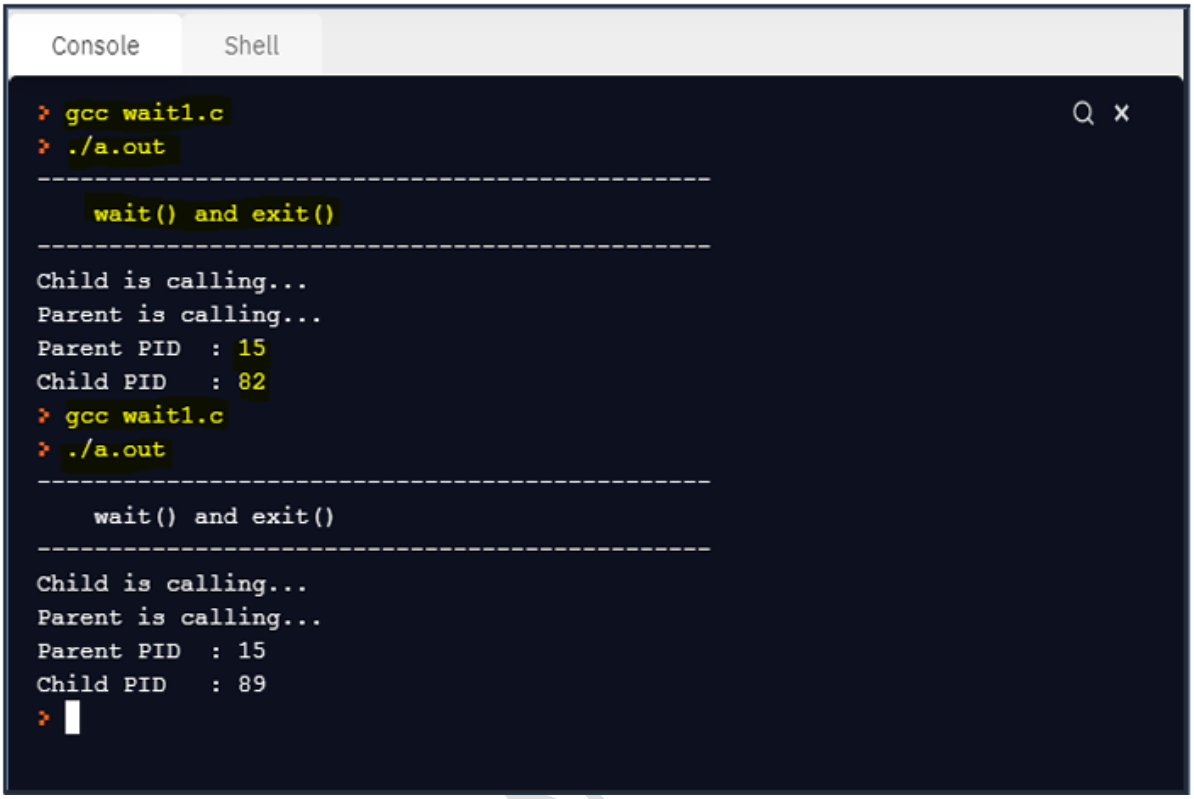
exit()

- It is system call and available in #include<stdlib> file
- It takes only one parameter which is exit status as a parameter
- It is used to close all files, sockets, frees all memory and then terminates the process.
- The parameter 0 indicates that the termination is normal.

EXAMPLE OF WAIT AND EXIT SYSTEM CALLS

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/wait.h>
int main()
{
   pid_t pd;
   printf("-----------------------------------------------\n");
   printf("\twait() and exit()\n");
   printf("-----------------------------------------------\n");
// execution of fork() call
   if (fork()== 0)
   {
      printf("Child is calling...\n");
// normal termination
      exit(0);
   }
   else
   {
// get the PID of terminated child process
      pd = wait(NULL);
      printf("Parent is calling...\n");
// print the process IDs of parent and child processes
      printf("Parent PID\t: %d\n", getppid());
      printf("Child PID\t: %d\n", pd);
   }
  return 0;
}
```

OUTPUT



SUM OF NUMBERS IN ARRAY USING CHILD AND PARENT PROCESS

Question: Write a Linux c program to find the sum of the numbers in the array in the child process and execute the parent process after the execution of the child process using system calls.

Used System calls: fork(), wait()

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/wait.h>
int main()
{
   int i,a[]={1,5,7,8,9};
int s=0;
   printf("--------------------------------------------------------------------------\n");
   printf("\tSum of Numbers in Child and Execution of Parent after the child\n");
```

```
    printf("--------------------------------------------------------------------------\n");
// create new process by fork and return the value in p
    int p=fork();
// if the returned value is negative value (unsuccessful status)
    if(p<0)
    {
       printf("Failed to create a new Process ...\n");
       exit(0);
    }
// if the returned value is equal to 0 (checking child process)
    else if(p==0)
    {
       printf("Child Process is calling...\n");
       for(i=0;i<5;i++)
       {
        s=s+a[i];
       }
       printf("The result is: %d\n",s);
       printf("Child Process is completed...\n");
    }
// if the returned value is positive (checking parent process)
    else
    {
        wait(NULL);
        printf("Parent Process is calling after the Child Process ...\n");
    }
  return 0;
}
```

Wait() System call: wait parent until the child has to terminate.

OUTPUT


Console
Shell
> gcc sumfork.c
> ./a.out
Sum of Numbers in Child and Execution of Parent after the child
Child Process is calling...
The result is: 30
Child Process is completed...
Parent Process is calling after the Child Process ...
> gcc sumfork.c
> ./a.out
Sum of Numbers in Child and Execution of Parent after the child
Child Process is calling...
The result is: 30
Child Process is completed...
Parent Process is calling after the Child Process ...
> ./a.out
Sum of Numbers in Child and Execution of Parent after the child
Child Process is calling...
The result is: 30
Child Process is completed...
Parent Process is calling after the Child Process ...
>

Score Criteria	Excellent (100%)	Good (80%)	Average (60%)	Poor (40%)	Absent (0%)	CO Mapping	PO PS Mapp
1. Lab Participation	Students are able to identify the problem/ analyze the problem/ Design the solutions and solve the problem applying various algorithms with appropriate test cases	Students are able to identify the problem/ analyze the problem/ Design the solutions and solve the problem applying various algorithms with appropriate test cases; students are able to include boundary conditions in test cases.	Students are able to identify the problem/ analyze the problem/ Design the solutions and solve the problem applying various algorithms with appropriate test cases	Student is not able to understand/ analyze / design the problem or interpret the problem into specified language		CO1, CO2	PO1, PO2, PSO1, PSO2
2. Effective utilization of the modern tools and their properties, compilers	Students are able to exploit the full potential of the tool/ property/ topic under consideration for the specified language	Students are able to exploit the important features of the tool/ property/ topic under consideration for the specified language	Students are able to use specified tool/property/ topic as per the problem requirement only under consideration for the specified language	Students are not able to use tool/ property/ topic under consideration for the specified language		CO3	PO5

3. Individual or team work	Students are able to work effectively, sincerely and ethically as an individual or in a member of a team	Students are able to work ethically as an individual or in a member of a team	Students are able to work as an individual or in a member of a team	Students are not able to work effectively, sincerely and ethically as an individual or in a member of a team		CO4	PO9
4. Documentation	Students will prepare effective documentation of lab classes mentioning problem statement	Students will prepare effective documentation of lab classes mentioning problem statement, input-output, test cases	Students will prepare effective documentation of lab classes mentioning problem statement, input-output	Students will nor prepare effective documentation of lab classes mentioning objective, input-output, test cases, boundary conditions		CO5	PO10