



MADRAS INSTITUTE OF TECHNOLOGY
ANNA UNIVERSITY



DEPARTMENT OF INFORMATION TECHNOLOGY

IT7411 – OPERATING SYSTEMS LABORATORY
LAB MANUAL
REGULATION - 2015

Vision of the Department

To educate students with conceptual knowledge and technical skills in the field of Information Technology with moral and ethical values to achieve excellence in an academic, industry and research centric environment.

Mission of the Department

1. To inculcate in students a firm foundation in theory and practice of IT skills coupled with the thought process for disruptive innovation and research methodologies, to keep pace with emerging technologies.
2. To provide a conducive environment for all academic, administrative, and interdisciplinary research activities using state-of-the-art technologies.
3. To stimulate the growth of graduates and doctorates, who will enter the workforce as productive IT engineers, researchers, and entrepreneurs with necessary soft skills, and continue higher professional education with competence in the global market.
4. To enable seamless collaboration with the IT industry and Government for consultancy and sponsored research.
5. To cater to cross-cultural, multinational, and demographic diversity of students.
6. To educate the students on the social, ethical, and moral values needed to make significant contributions to society.

Program Educational Objectives (PEO)

After completion of the B.Tech. (IT) course, students will be able to:

PEO1: Demonstrate core competence in basic engineering and mathematics to design, formulate, analyse, and solve hardware/ software engineering problems.

PEO2: Have insight in fundamental areas of Information Technology and related engineering with an inclination towards self-learning to address real-world problems using digital and cognitive technologies.

PEO3: Collaborate with industry, academic and research institutions for product and research related development.

PEO4: Imbibe high professionalism, effective communication skills and team spirit to work on multidisciplinary projects, in diverse professional environments.

PEO5: Practice IT solutions following technical standards with ethical values.

Program Specific Outcomes(PSO)

PSO1: To apply programming principles and practices for the design of software solutions in an internet-enabled world of business and social activities.

PSO2: To identify the resources to build and manage the IT infrastructure using the current technologies in order to solve real world problems with an understanding of the trade-offs involved in the design choices.

PSO3: To plan, design and execute projects for the development of intelligent systems with a focus on the future

OBJECTIVES:

- To learn about the basic commands of operating systems
- To learn various process management schemes in operating systems
- To practice with the important memory management mechanisms in operating system
- To implement the file handling techniques in operating systems

Exercises

1. Basic unix commands such as ls, cd, mkdir, rmdir, cp, rm, mv, more, lpr, man, grep, sed, etc.,
2. Shell script
3. Process control System calls - demonstration of fork, execute and wait
4. Thread management
5. Thread synchronization
6. Deadlock avoidance using semaphores
7. Interprocess communication using pipes
8. Interprocess communication using FIFOs
9. Interprocess communication using signals
10. Implementation of CPU scheduling policy in Linux
11. Implement a memory management policy in Linux
12. Implement a file system in Linux
13. Linux kernel configuration

TOTAL: 60 PERIODS**OUTCOMES:**

On Completion of the course, the students should be able to:

- Learn the concepts to identify, create and maintain the basic command in operating systems
- Express strengths and limitations of various managements schemes in operating systems
- Explain the core issues of operating systems
- Implement algorithms of operating systems.

Mapping of Course Outcomes (COs) with Program Outcomes (POs)

CO	Program Outcomes(POs)												PSO1	PSO2	PSO3
	1	2	3	4	5	6	7	8	9	10	11	12			
Learn the concepts to identify, create and use the basic command in operating systems	1	2	3	1	1	1	0	0	0	0	0	2	2	2	2
Familiarize strengths and limitations of various managements schemes in operating systems	1	2	3	2	1	0	0	0	0	0	0	1	2	2	2
Implement core concepts/ issues of operating systems	1	2	2	3	3	1	0	0	0	0	1	1	2	2	2
Implement algorithms of operating systems	1	2	2	3	3	2	2	0	0	0	1	1	2	2	2
Exploration of memory management methodologies	1	2	2	3	3	2	2	0	0	0	1	1	2	2	2
Exploration of interprocess communication strategies	1	2	3	3	3	3	3	1	0	0	1	1	2	2	2

GRADING RUBRIC FOR LABORATORY COURSES

	Good Marks (81%-100%)	Average Marks (50%-80%)	Satisfactory Marks (< 50%)
<p>Continuous Assessment (Covers Preparedness, Basic implementation, Ability to adapt additional features and coding standards) (Max Marks:25)</p>	<p>Presence of detailed procedure, coding samples with proper implementation.</p> <p>Able to adapt the changes in the code quickly.</p> <p>Proper Coding Style.</p>	<p>Clarity of the procedure and coding samples are average with partial implementation. Able to understand the changes but unable to implement it.</p> <p>Fairly presented code with medium standards.</p>	<p>Lack of detailed procedure as well as coding samples with incorrect implementation.</p> <p>Unable to adapt the changes in coding.</p> <p>Coding standards are not followed. Code is messy.</p>
<p>Laboratory Test (Covers Understanding of problem, Basic Problem Solving and Ability to code, test, run and debug within the stipulated time) (Max Marks:25)</p>	<p>Problem understood clearly and solved.</p> <p>Complete implementation with proper test data within the stipulated time.</p>	<p>Problem understood but problem solving is not full-fledged.</p> <p>Completion of three fourths of the implementation with proper test data.</p>	<p>Lack of understanding and problem-solving ability is poor.</p> <p>Implementation not completed/ Partial implementation within the stipulated time.</p>
<p>Course Oriented Laboratory Project (Covers Problem Selection, Demonstration of the Project, Wide coverage of concepts in the target language) (Max Marks:25)</p>	<p>Selection of good real time problem with Complete implementation with in-depth understanding on the concepts implemented.</p> <p>Wide coverage of concepts in the target language.</p>	<p>Selection of good real time problem with partially complete implementation and good knowledge on the concepts implemented.</p> <p>Moderate coverage of concepts.</p>	<p>Selection of fair problem with incomplete implementation. Lack of proper knowledge and understanding on the concepts implemented.</p> <p>Limited coverage of concepts.</p>

S.NO	NAME OF THE EXPERIMENT
1.	INTRODUCTION TO OPERATING SYSTEMS
2.	LINUX COMMANDS FILE SYSTEM
3.	SHELL PROGRAMMING BASICS
4.	SHELL SCRIPTING – OPERATORS, FUNCTIONS
5.	SHELL ARRAYS
6.	PROCESS SYSTEM CALLS – FORK, EXIT, WAIT
7.	INTERPROCESS COMMUNICATION USING PIPE
8.	INTERPROCESS COMMUNICATION USING NAMED PIPE
9.	MULTITHREADING USING PYTHON
10.	FILE ALLOCATION STRATEGIES

EX.NO: 1

INTRODUCTION TO OPERATING SYSTEMS

AIM

To the study the functions of an operating system.

1.1 OVERVIEW

An Operating System (OS)(as shown in Fig 1) is an interface between a computer user and computer hardware. An operating system is a software which 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.

Some popular Operating Systems include Linux Operating System, Windows Operating System, VMS, OS/400, AIX, z/OS, etc. Following are some of 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.

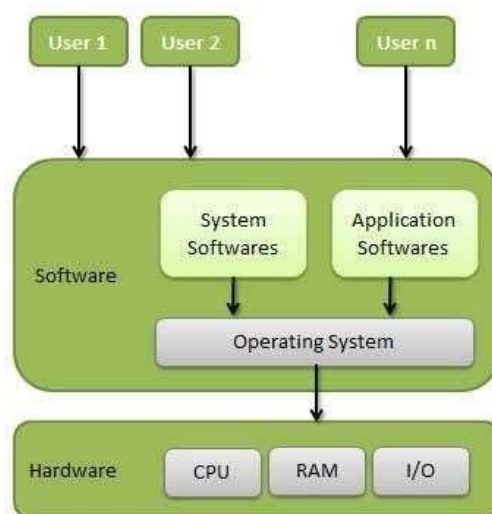


Figure 1. Operating System

1.2 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.

1.2.1 LINUX ARCHITECTURE

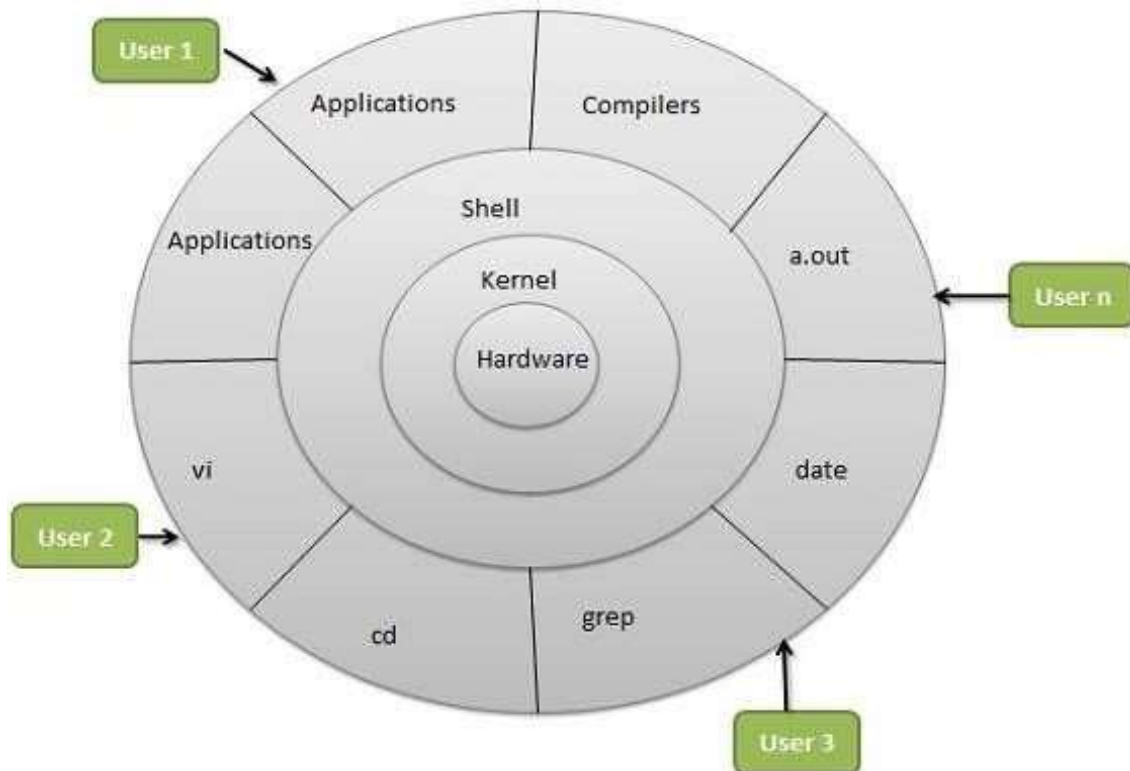


Figure 2. Linux OS Architecture

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 Operating System, interacts directly with hardware, 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.

1.2.2 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.

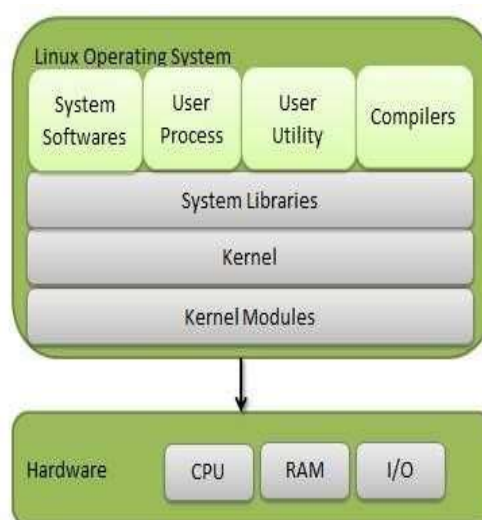


Figure 3. Linux Operating System

1.2.3 KERNEL MODE VS USER MODE

Kernel component code executes in a special privileged mode called kernel mode with full access to all resources of the computer. This code represents a single process, executes in single address space and do not require any context switch and hence is very efficient and fast. Kernel runs each processes and provides system services to processes, provides protected access to hardware to processes.

Support code which is not required to run in kernel mode is in System Library. User programs and other system programs works in User Mode which has no access to system hardware and kernel code. User programs/ utilities use System libraries to access Kernel functions to get system's low level tasks.

1.2.4 BASIC FEATURES

Following are some of the important features of linux operating system

- **Portable**

Portability means software can work on different types of hardware in same way. Linux kernel and application programs supports their installation on any kind of hardware platform.

- **Open Source**

Linux source code is freely available and it is community based development project. Multiple teams work in collaboration to enhance the capability of Linux operating system and it is continuously evolving.

- **Multi User**

Linux is a multiuser system means multiple users can access system resources like memory/ ram/ application programs at same time.

- **Multiprogramming**

Linux is a multiprogramming system means multiple applications can run at same time.

- **Hierarchical File System**

Linux provides a standard file structure in which system files/ user files are arranged.

- **Shell**

Linux provides a special interpreter program which can be used to execute commands of the operating system. It can be used to do various types of operations, call application programs. etc.

- **Security**

Linux provides user security using authentication features like password protection/ controlled access to specific files/ encryption of data.

1.3 COMPARISONS BETWEEN LINUX OS WITH DIFFERENT OS

1.3.1 LINUX VS WINDOWS

Linux OS

Linux could be a free and open supply OS supported operating system standards. It provides programming interface still as programme compatible with operating system primarily based systems and provides giant selection applications. A UNIX operating system additionally contains several severally developed parts, leading to UNIX operating system that is totally compatible and free from proprietary.

Windows OS

Windows may be a commissioned OS within which ASCII text file is inaccessible. it's designed for the people with the angle of getting no programming information and for business and alternative industrial users. it's terribly straightforward and simple to use.

The distinction between Linux and Windows package is that Linux is completely freed from price whereas windows is marketable package and is expensive. Associate operating system could be a program meant to regulate the pc or computer hardware.

Linux is an open supply package wherever users will access the ASCII text file and might improve the code victimisation the system. On the opposite hand, in windows, users can't access ASCII text file, and it's an authorized OS.

Let's sees that the difference between Linux and windows:

S.N	LINUX	WINDOWS
1.	It is an open source operating system	It is a commercial operating system (closed source)
2.	In linux, monolithic kernel is used	In windows, micro kernel is used.
3.	In the comparison of file system, linux runs faster even with older hardware	Windows are slower compared to Linux
4.	Linux files are ordered in a tree structure starting with the root directory.	In windows, files are stored in folders on different data drives like C: D: E:
5.	It is customizable	It is not possible to customize the windows OS
6.	It supports multiple desktop environments	It supports only preinstalled desktop environment
7.	It is more secure than windows	Vulnerable to viruses and malware attacks.
8.	Booting takes either primary or logical partition in linux	In windows, booting supports only primary partition

1.3.2 LINUX VS MAC

Linux

Linux is a group of open source Unix-like operating systems which was developed by Linus Torvalds. It is a packaged of Linux distribution. Some of the mostly used Linux distribution are Debian, Fedora and Ubuntu. It was basically written in C language and assembly language. Kernel used in Linux is Monolithic kernel. The target systems of Linux distributions are cloud computing, embedded systems, mobile devices, personal computers, servers, mainframe computers and supercomputers. The first version of Linux was launched in 1991.

MAC

Mac OS is a series of proprietary graphical operating systems which is provided by Apple Incorporation. It was earlier known as Mac OS X and later OS X. It is specifically designed for Apple mac computers. It is based on Unix operating system. It was developed using C, C++, Objective-C, assembly language and Swift. It is the second most used operating system in personal computers after Windows. The first version of macOS was launched by Apple in 2001.

Let's see that the difference between Linux and mac:

S.N	LINUX	MAC
1.	It is an open source operating system	It is a commercial operating system (closed source)
2.	In linux, monolithic kernel is used	MacOS is based the Xnu hybrid micro kernel
3.	It is used as OS, as server provide platform to run other application	Mac is an operating system provides platform to run other application
4.	It is customizable	It is not possible to customize the Mac OS
5.	It supports many flavours like RedHat, Ubuntu, Fedora, Suse, etc,	It does have any flavours.
6.	It supports multiple desktop environments like GNOME, KDE, Mate, Budgie, Cinnamon, Deepin etc,...	It supports only preinstalled desktop environment

1.3.3 LINUX VS UNIX

Linux

Linux is an open source multi-tasking, multi-user operating system. It was initially developed by Linus Torvalds in 1991. Linux OS is widely used in desktops, mobiles, mainframes etc.

Unix

Unix is multi-tasking, multi-user operating system but is not free to use and is not open source. It was developed in 1969 by Ken Thompson team at AT&T Bell Labs. It is widely used on servers, workstations etc. Following are the important differences between Linux and Unix.

Let's see that the difference between Linux and windows:

S.N	LINUX	UNIX
1.	It is an open source operating system	It is a licensed OS (closed source)
2.	It is developed by linux community of developers	It was developed by AT and T Bell labs
3.	Linux uses KDE and Gnome. Other GUI supported are LXDE, Xfce, Unity, Mate.	Unix was initially a command based OS. Most of the unix distributions now have Gnome.
4.	Bash (B ourne A gain S hell) is default shell for Linux.	Bourne Shell is default shell for Unix.
5.	Its flavours are RedHat, Ubuntu, Suse, Kali Linux, etc,...	Its flavours are SunOS, Solaris, HP-UX, AIX, Sco Unix, etc,...
6.	Linux is used in wide varieties from desktop, servers, smartphones to mainframes.	It is mostly used on servers, workstations or PCs.

1.4 SCHEDULING OF JOBS IN OPERATING SYSTEM

Job scheduling is the process of allocating system resources to many different tasks by an operating system (OS). The system handles prioritized job queues that are awaiting CPU time and it should determine which job to be taken from which queue and the amount of time to be allocated for the job.

This type of scheduling makes sure that all jobs are carried out fairly and on time. Most OSs like Unix, Windows, etc., include standard job-scheduling abilities. A number of programs including database management systems (DBMS), backup, enterprise resource planning (ERP) and business process management (BPM) feature specific job scheduling capabilities as well.

1.5 PROCESS MANAGEMENT

Process management involves various tasks like creation, scheduling, termination of processes, and a dead lock. Process is a program that is under execution, which is an important part of modern-day operating systems. The OS must allocate resources that enable processes to share and exchange information.

It also protects the resources of each process from other methods and allows synchronization among processes. It is the job of OS to manage all the running processes of the system. It handles operations by performing tasks like process scheduling and such as resource allocation.

1.6 PROCESS ARCHITECTURE

Stack

The stack stores temporary data like function parameters, returns addresses, and local variables.

Heap

It allocates memory, which may be processed during its run time.

Data

It contains the variable.

Text

It includes the current activity, which is represented by the value of the Program Counter.

1.7 MEMORY MANAGEMENT IN OPERATING SYSTEMS

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

The operating system takes care of mapping the logical addresses to physical addresses at the time of memory allocation to the program. There are three types of addresses used in a program before and after memory is allocated –

1. Symbolic addresses

The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space.

2. Relative addresses

At the time of compilation, a compiler converts symbolic addresses into relative addresses.

3. Physical addresses

The loader generates these addresses at the time when a program is loaded into main memory.

1.8 SYSTEM CALLS IN OPERATING SYSTEMS

System call is a mechanism that provides the interface between a process and the operating system. It is a programmatic method in which a computer program requests a service from the kernel of the OS. System call offers the services of the operating system to the user programs via API (Application Programming Interface). System calls are the only entry points for the kernel system. For example, if we need to write a program code to read data from one file, copy that data into another file.

The first information that the program requires is the name of the two files, the input and output files. In an interactive system, this type of program execution requires some system calls by OS.

- First call is to write a prompting message on the screen.
- Second, to read from the keyboard, the characters which define the two files.

1.9 DAY TO DAY USAGE OF OPERATING SYSTEMS

The operating system is used everywhere nowadays especially such as banks, schools, colleges, universities, govt. organizations, IT companies, mobile, etc, ...

No device can operate without an operating system because it controls all the user's commands. Linux /unix operating systems is used in bank because it's very secure operating systems.

Symbian OS, windows mobile, IOS and Android OS are used in mobile phones operating systems as these operating systems are a lightweight operating systems.

RESULT

The architecture and features of an operating system has been studied successfully.

EX.NO: 2 LINUX COMMANDS FILE SYSTEM

AIM

To study and implement about the various basic linux commands.

I. FILE RELATED COMMANDS

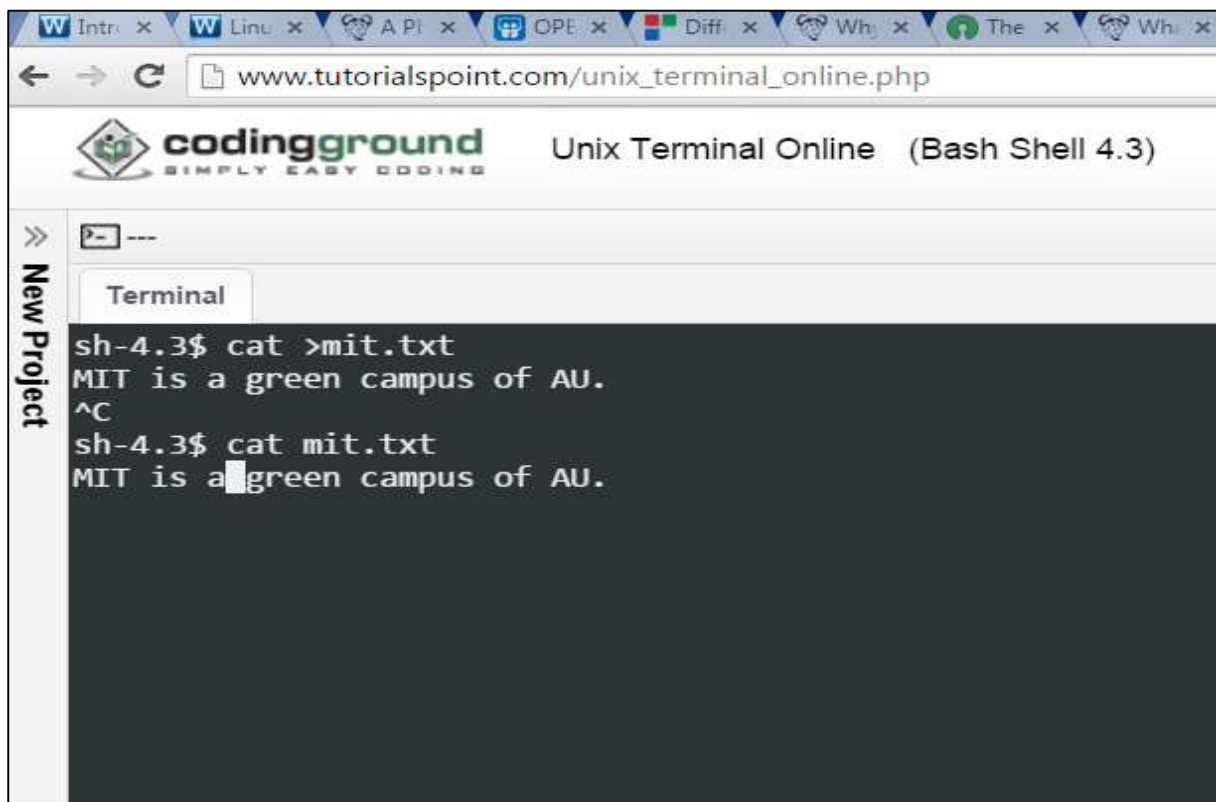
1. cat

- 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

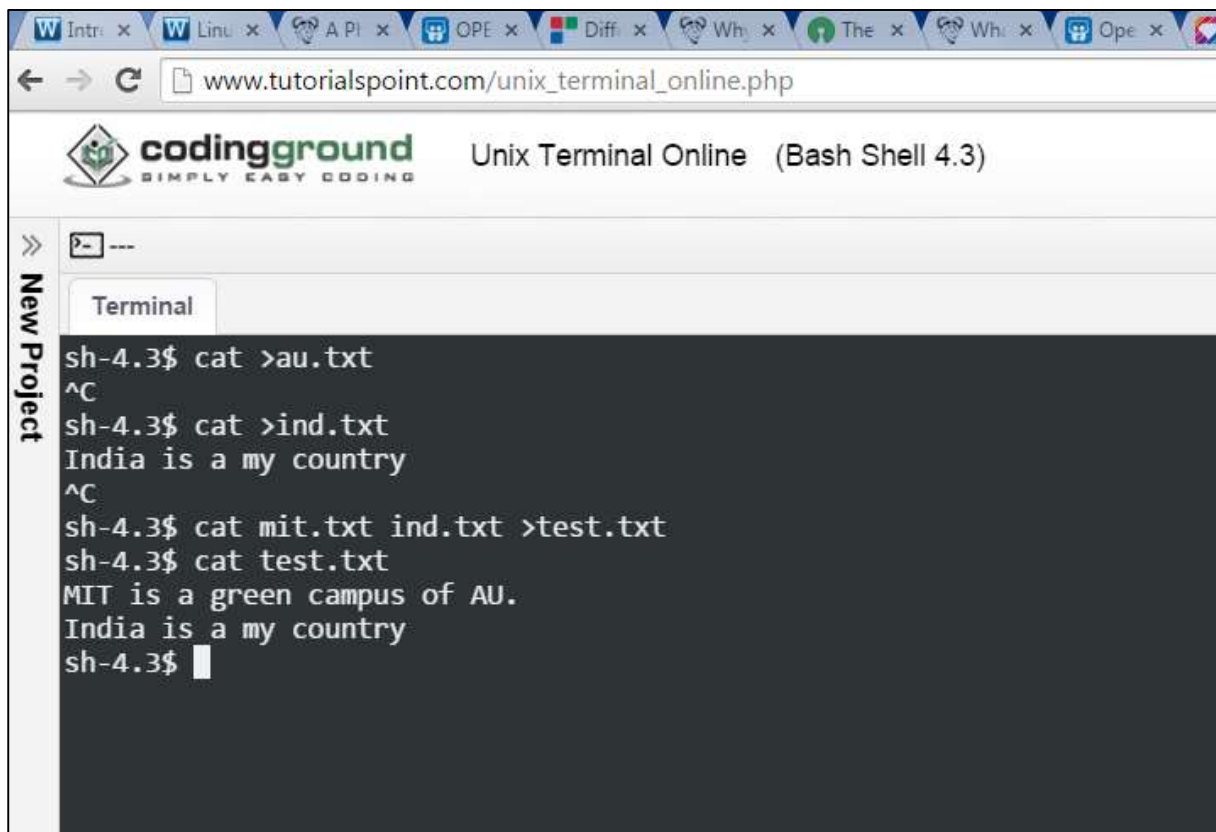
```
$ cat > filename            (Create a new file)
$ cat <filename>            (Display the contents of file)
$ cat file1 file2 >file3    (merge the contents of file1, file2 into file 3)
```

1.1 File Creation & Display its Contents using cat command



```
W Intr. x W Linu x A Pi x OPE x Diff. x Wh. x The x Wh. x
www.tutorialspoint.com/unix_terminal_online.php
codingground SIMPLY EASY CODING Unix Terminal Online (Bash Shell 4.3)
New Project
Terminal
sh-4.3$ cat >mit.txt
MIT is a green campus of AU.
^C
sh-4.3$ cat mit.txt
MIT is a green campus of AU.
```

1. 2 Files Merging using cat command



The screenshot shows a web browser window with the URL www.tutorialspoint.com/unix_terminal_online.php. The page title is "Unix Terminal Online (Bash Shell 4.3)". The terminal interface shows the following commands and output:

```
sh-4.3$ cat >au.txt
^C
sh-4.3$ cat >ind.txt
India is a my country
^C
sh-4.3$ cat mit.txt ind.txt >test.txt
sh-4.3$ cat test.txt
MIT is a green campus of AU.
India is a my country
sh-4.3$
```

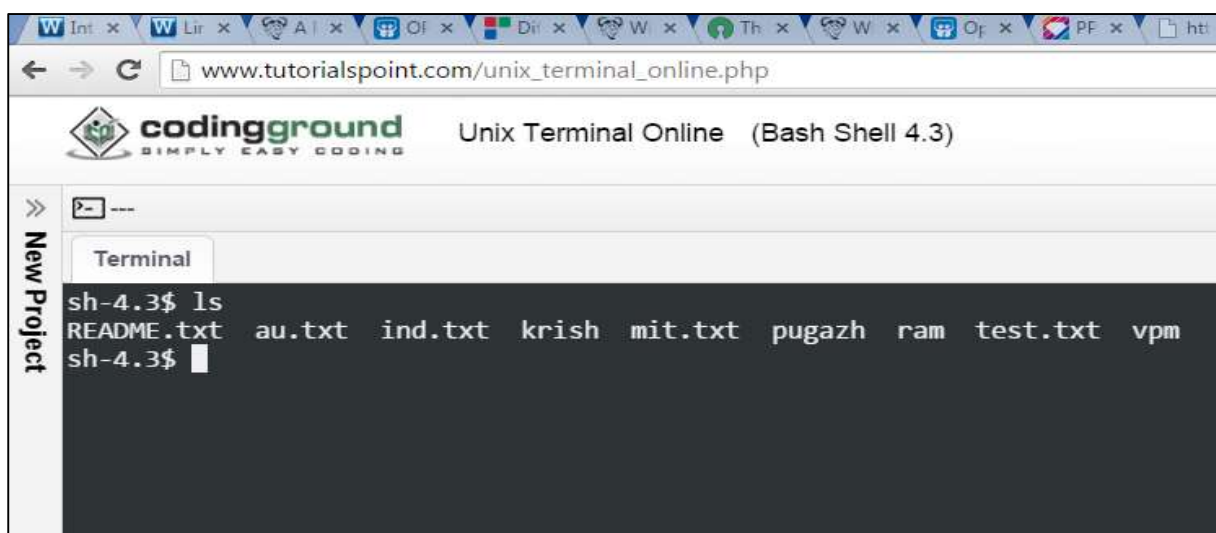
2. Is command

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

Syntax

\$ ls → View the contents of directory

2. Displaying Files and Folders



The screenshot shows the same web browser window as above. The terminal interface shows the following command and output:

```
sh-4.3$ ls
README.txt  au.txt  ind.txt  krish  mit.txt  pugazh  ram  test.txt  vpm
sh-4.3$
```

3. clear command

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

II. DIRECTORY RELATED COMMANDS

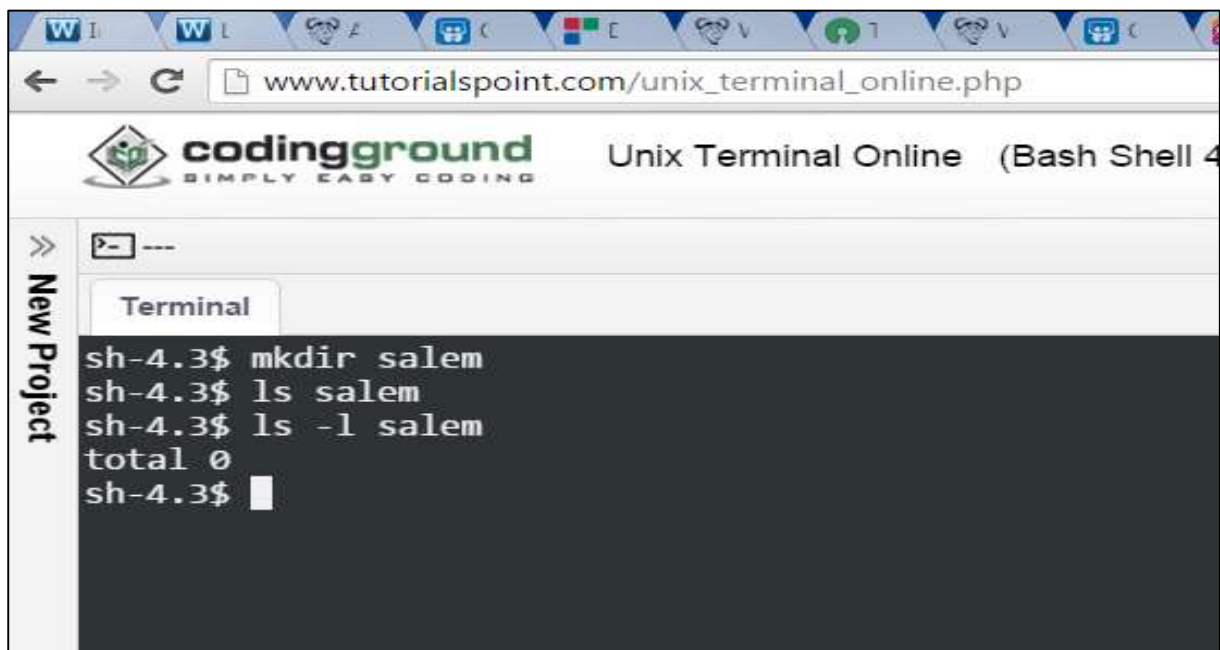
1. mkdir

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

Syntax

```
$ mkdir <dname>
```

1.1 Empty Directory Creation



```
www.tutorialspoint.com/unix_terminal_online.php  
codingground SIMPLY EASY CODING Unix Terminal Online (Bash Shell 4  
New Project  
Terminal  
sh-4.3$ mkdir salem  
sh-4.3$ ls salem  
sh-4.3$ ls -l salem  
total 0  
sh-4.3$
```

2. rmdir

- This command is used remove a directory from the disk

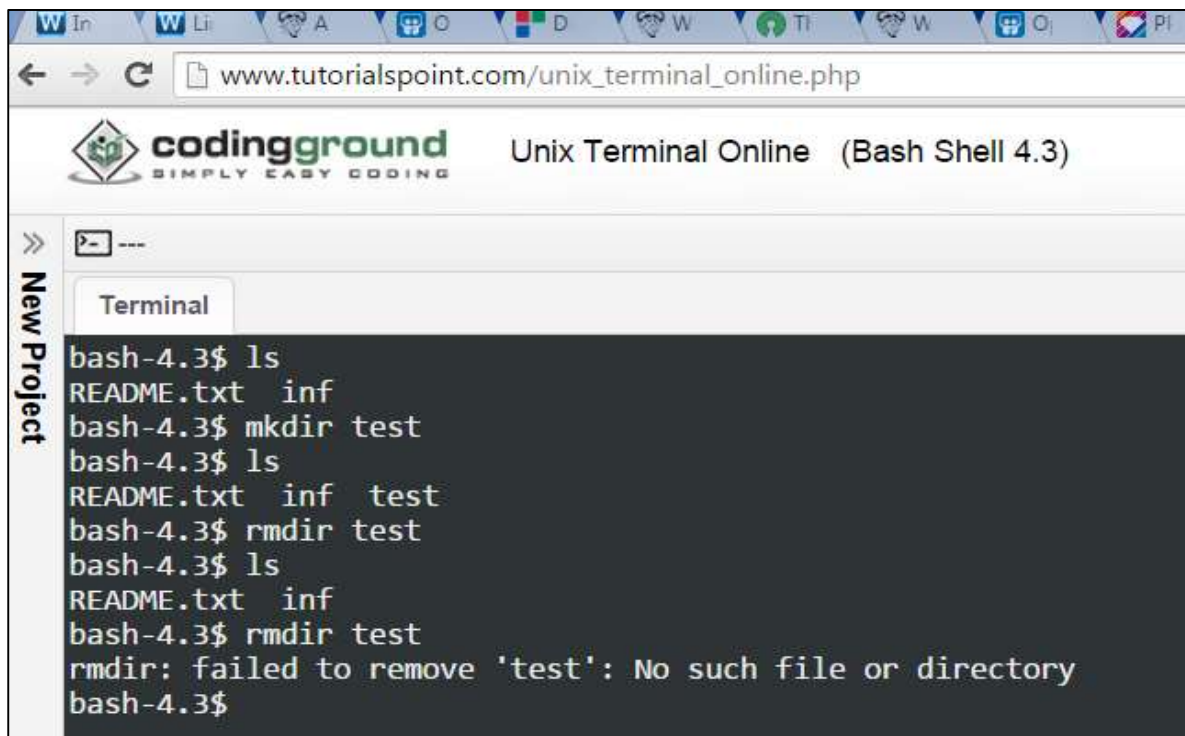
Rules for Directory Deletion

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

Syntax

```
$ rmdir <dname>
```

2.1 Empty Directory Deletion



The screenshot shows a web browser window with the URL www.tutorialspoint.com/unix_terminal_online.php. The page title is "codingground" with the tagline "SIMPLY EASY CODING". The main content is a terminal window titled "Terminal" with the following output:

```
bash-4.3$ ls
README.txt  inf
bash-4.3$ mkdir test
bash-4.3$ ls
README.txt  inf  test
bash-4.3$ rmdir test
bash-4.3$ ls
README.txt  inf
bash-4.3$ rmdir test
rmdir: failed to remove 'test': No such file or directory
bash-4.3$
```

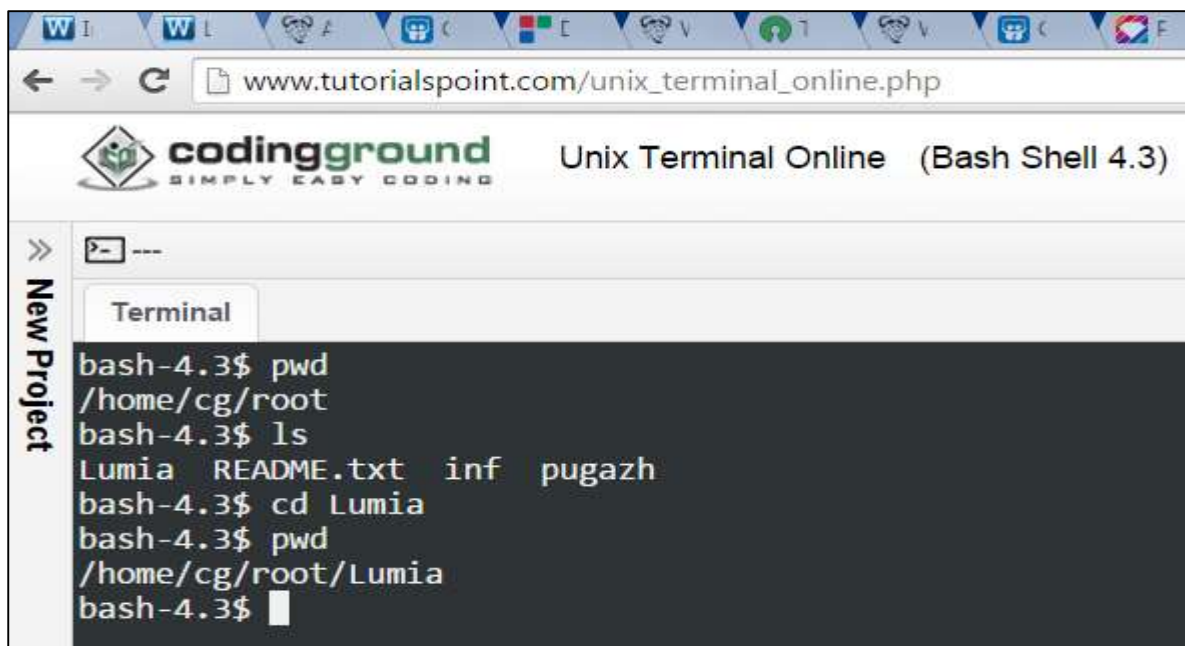
3. cd

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

Syntax

\$ cd <dirname>

3.1 Changing the Working Directory



The screenshot shows a web browser window with the URL www.tutorialspoint.com/unix_terminal_online.php. The page title is "codingground" with the tagline "SIMPLY EASY CODING". The main content is a terminal window titled "Terminal" with the following output:

```
bash-4.3$ pwd
/home/cg/root
bash-4.3$ ls
Lumia README.txt  inf  pugazh
bash-4.3$ cd Lumia
bash-4.3$ pwd
/home/cg/root/Lumia
bash-4.3$
```

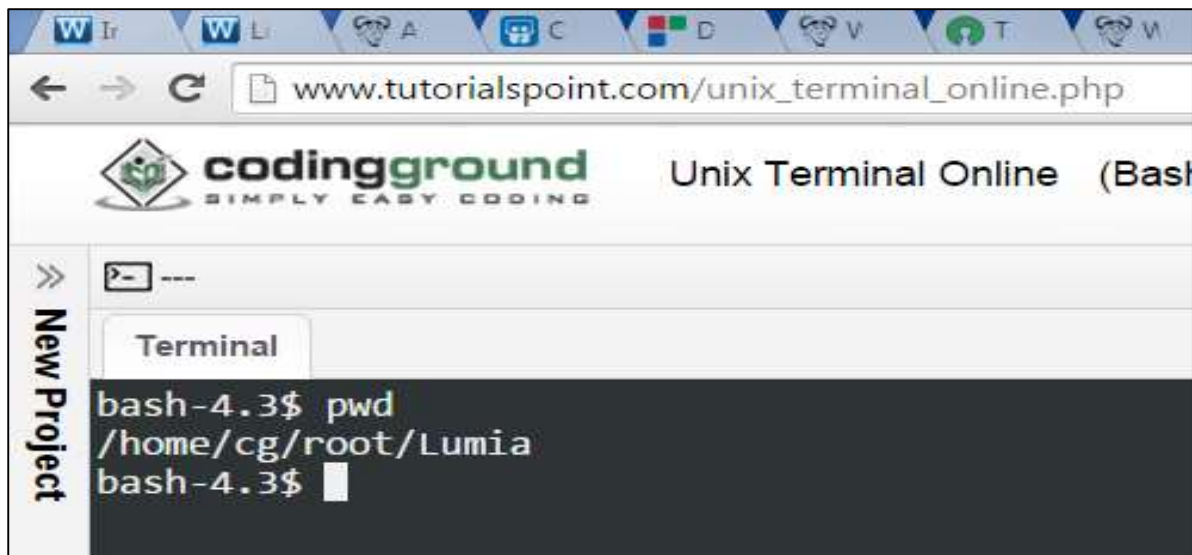
4. pwd

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

Syntax

```
$ cd <dtype>
```

4.1 Displaying Current Working Directory



The screenshot shows a web browser window with the URL www.tutorialspoint.com/unix_terminal_online.php. The page features the 'codingground' logo and the text 'Unix Terminal Online (Bash)'. A terminal window titled 'New Project' is open, displaying the following text:

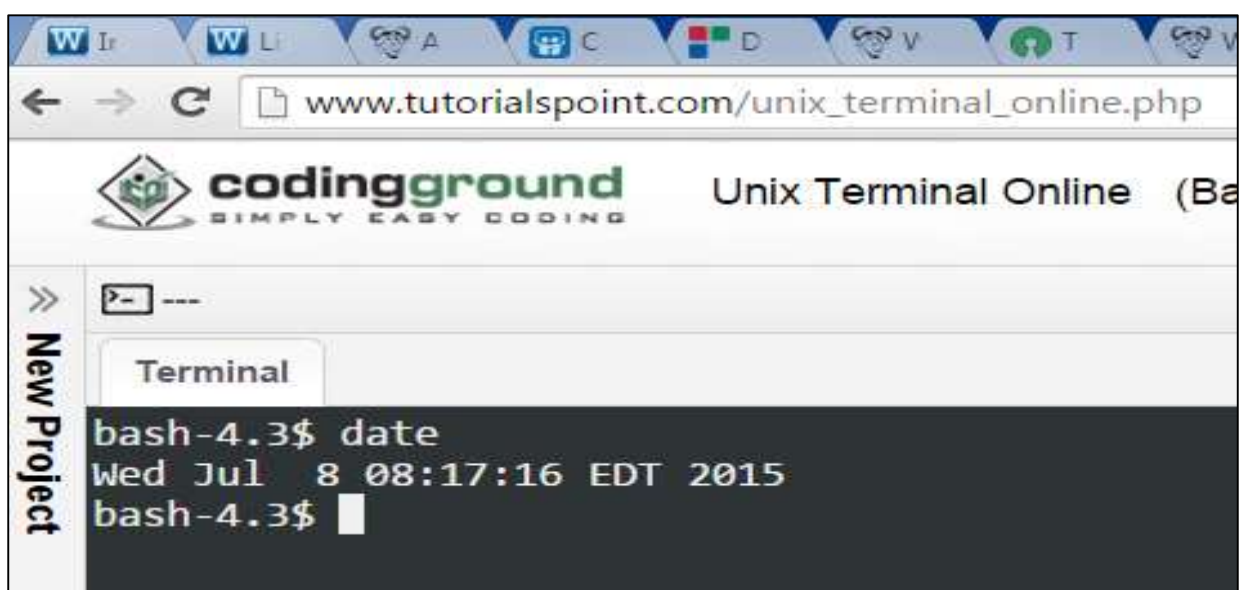
```
bash-4.3$ pwd
/home/cg/root/Lumia
bash-4.3$
```

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

1. Output



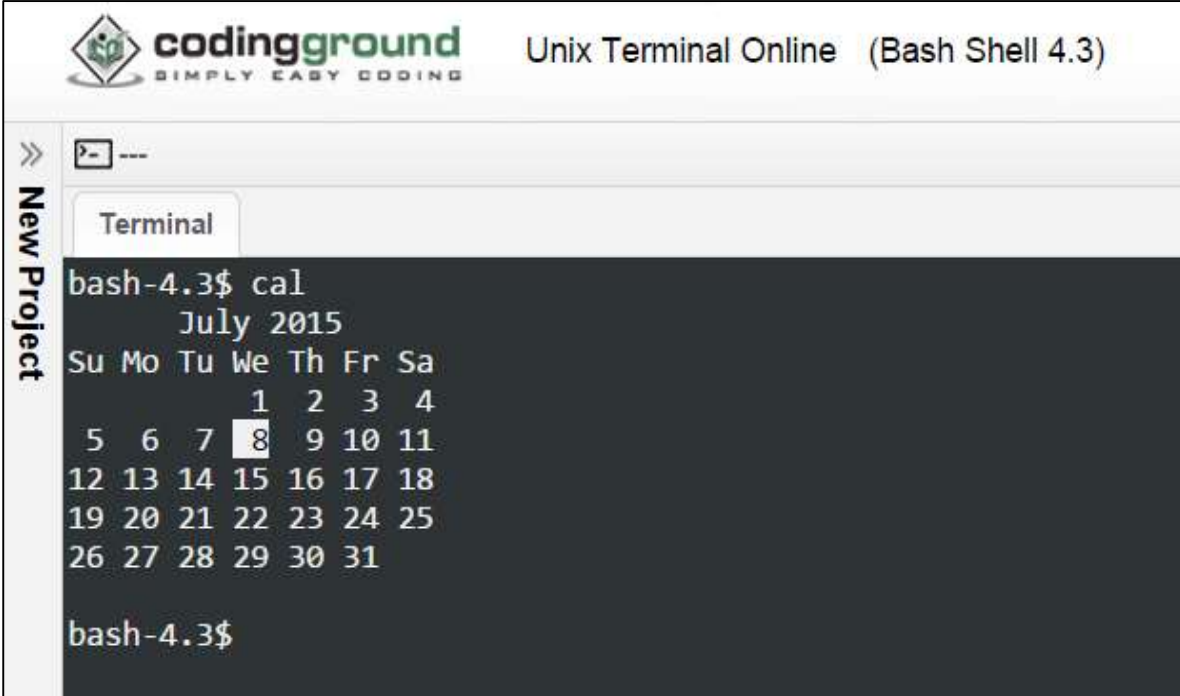
The screenshot shows a web browser window with the URL www.tutorialspoint.com/unix_terminal_online.php. The page features the 'codingground' logo and the text 'Unix Terminal Online (Bash)'. A terminal window titled 'New Project' is open, displaying the following text:

```
bash-4.3$ date
Wed Jul 8 08:17:16 EDT 2015
bash-4.3$
```

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 Current Month

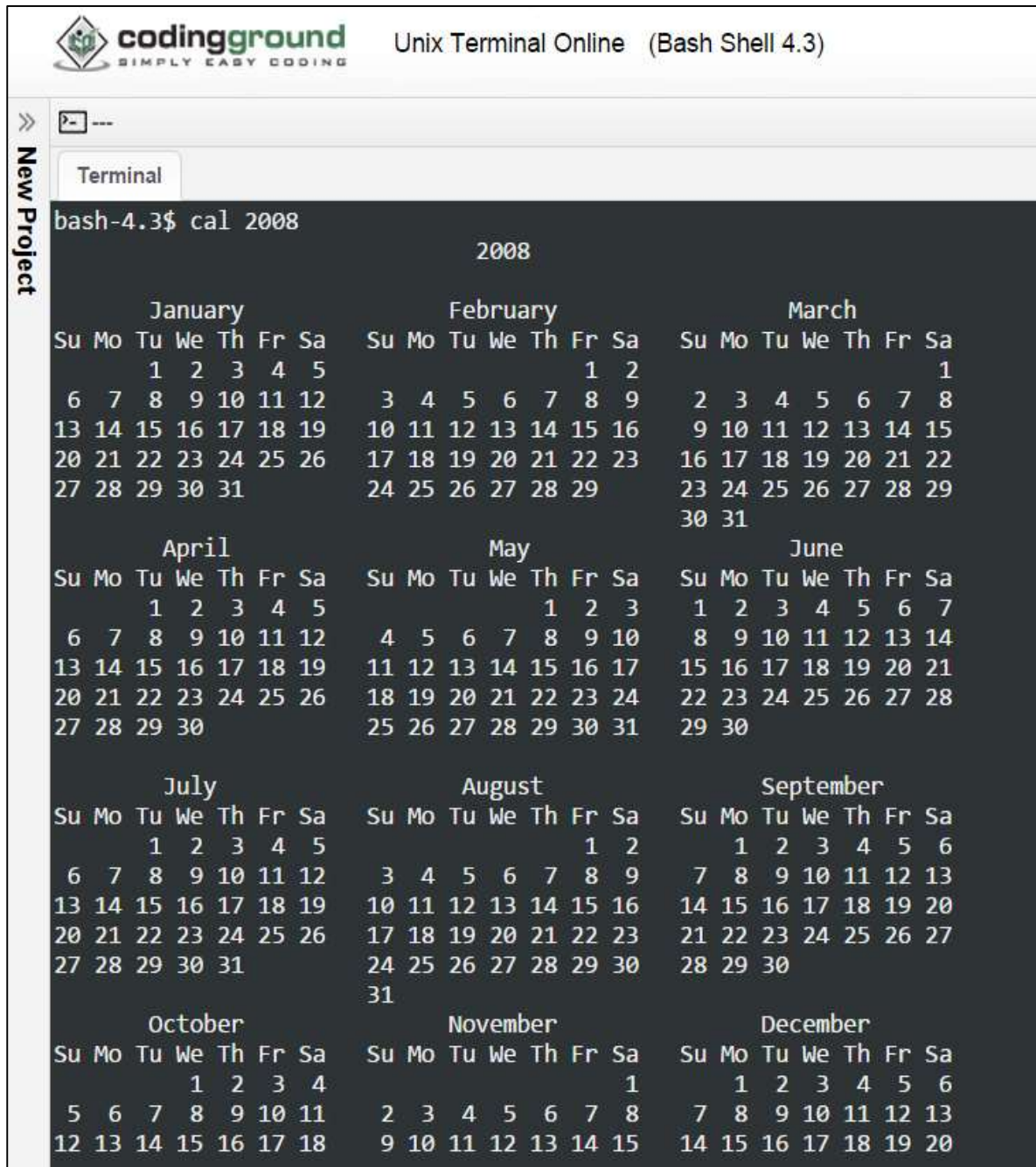


The screenshot shows a web-based terminal interface. At the top, there is a logo for 'codingground' with the tagline 'SIMPLY EASY CODING' and the text 'Unix Terminal Online (Bash Shell 4.3)'. Below the header, there is a sidebar on the left with a 'New Project' button. The main terminal area shows the following output:

```
bash-4.3$ cal
      July 2015
Su Mo Tu We Th Fr Sa
      1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

bash-4.3$
```

2.2 Displaying Entire Year



```
bash-4.3$ cal 2008

                2008

   January                February                March
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
   1  2  3  4  5                1  2                1
   6  7  8  9 10 11 12    3  4  5  6  7  8  9    2  3  4  5  6  7  8
  13 14 15 16 17 18 19    10 11 12 13 14 15 16    9 10 11 12 13 14 15
  20 21 22 23 24 25 26    17 18 19 20 21 22 23    16 17 18 19 20 21 22
  27 28 29 30 31          24 25 26 27 28 29    23 24 25 26 27 28 29
                                     30 31

   April                  May                  June
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
   1  2  3  4  5                1  2  3    1  2  3  4  5  6  7
   6  7  8  9 10 11 12    4  5  6  7  8  9 10    8  9 10 11 12 13 14
  13 14 15 16 17 18 19    11 12 13 14 15 16 17    15 16 17 18 19 20 21
  20 21 22 23 24 25 26    18 19 20 21 22 23 24    22 23 24 25 26 27 28
  27 28 29 30          25 26 27 28 29 30 31    29 30

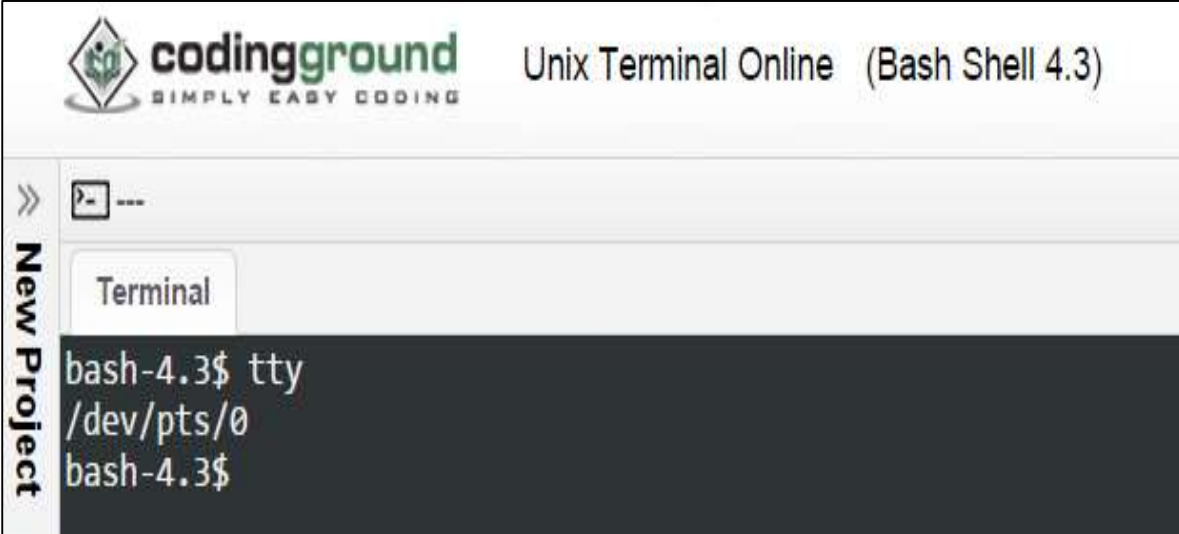
   July                  August                September
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
   1  2  3  4  5                1  2                1  2  3  4  5  6
   6  7  8  9 10 11 12    3  4  5  6  7  8  9    7  8  9 10 11 12 13
  13 14 15 16 17 18 19    10 11 12 13 14 15 16    14 15 16 17 18 19 20
  20 21 22 23 24 25 26    17 18 19 20 21 22 23    21 22 23 24 25 26 27
  27 28 29 30 31          24 25 26 27 28 29 30    28 29 30
                                     31

   October                November                December
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
   1  2  3  4                1                1  2  3  4  5  6
   5  6  7  8  9 10 11    2  3  4  5  6  7  8    7  8  9 10 11 12 13
  12 13 14 15 16 17 18    9 10 11 12 13 14 15    14 15 16 17 18 19 20
```

3. tty command

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

3.1 Displaying Terminal Name



The screenshot shows a web browser window with the following elements:

- Header: **codingground** SIMPLY EASY CODING | Unix Terminal Online (Bash Shell 4.3)
- Left sidebar: >> New Project
- Terminal window: Terminal
- Terminal output:

```
bash-4.3$ tty
/dev/pts/0
bash-4.3$
```

RESULT

Thus the linux commands have been studied and executed successfully.

EX.NO: 3 SHELL PROGRAMMING BASICS

AIM

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

EXISTING PROBLEM

- It is not possible to perform more than one task at a time using shell command

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

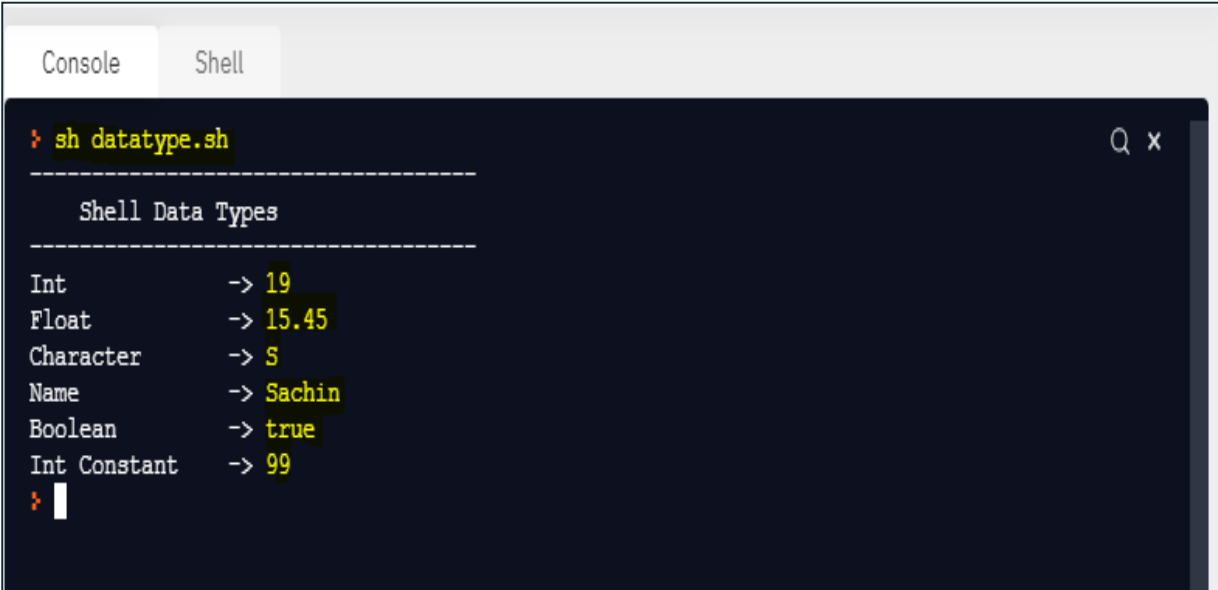
```
a=10                            # integer type
str="Sachin"                   # string type
c='G'                           # character type
f=true                         # boolean value
readonly id=14                # integer constant
```

I. EXAMPLE OF DATA TYPES IN SHELL CODE

SOURCE CODE

```
echo "-----"  
echo "\tShell Data Types"  
echo "-----"  
# variables definition  
a=19  
b=15.45  
c='S'  
str="Sachin"  
flag=true  
# constant variable definition  
readonly id=99  
echo "Int\t\t\t-> $a"  
echo "Float\t\t-> $b"  
echo "Character\t-> $c"  
echo "Name\t\t-> $str"  
echo "Boolean\t\t-> $flag"  
echo "Int Constant\t-> $id"
```

2. OUTPUT



```
Console Shell  
┆ sh datatype.sh  
-----  
Shell Data Types  
-----  
Int          -> 19  
Float       -> 15.45  
Character    -> S  
Name        -> Sachin  
Boolean     -> true  
Int Constant -> 99  
┆
```

COMMENT LINE STATEMENTS

- Usually these statements are ignored by compiler or interpreter
- Like c/c++, shell supports two types of comment line statements. They are
 1. Single line statement
 2. Multi line statements

1. Single line statement

- The single line statement is indicated by **# symbol** in shell program

Example

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

2. Multi line statements

- It is used to ignore more than one statements
- This is indicated by **:' symbol** in 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 an important to note that, the space should be given before and after the **operator symbol** []
- You can use test keyword instead of the operator symbols []

2. If else Statement

Syntax

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

3. If...elif...else Statement

Syntax

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

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

4. 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

Syntax

```
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)

1. While loop

Syntax

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

Infinite While loop

- It is an 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 []

II. EXAMPLE OF INFINITE LOOP USING WHILE LOOP

SOURCE CODE

```
while :  
do  
    echo "Hello World"  
done
```

(OR)

```
while true  
do  
    echo "Hello World"
```


2. Until loop

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

Syntax

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

III. EXAMPLE OF UNTIL LOOP

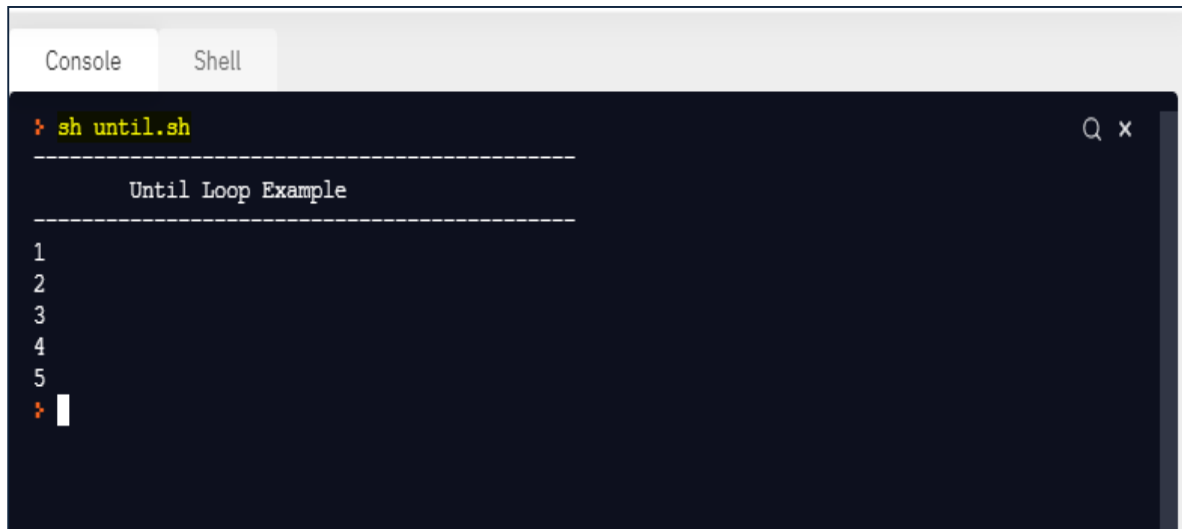
SOURCE CODE

```
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



```
Console Shell
sh until.sh
-----
Until Loop Example
-----
1
2
3
4
5
sh
```

3. For loop

Syntax

```
for variable in w1 w2 ... wn
do
    true statement
done
```

Where,

Variable can be any user defined name

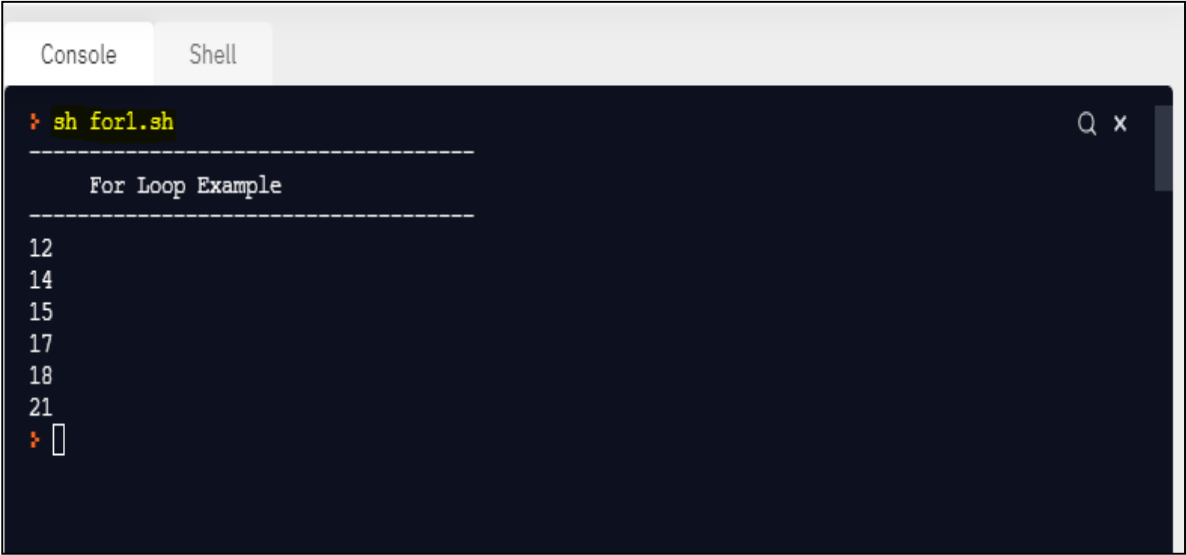
w1 w2 ...wn → list of the values separated by spaces

IV. EXAMPLE OF FOR LOOP

SOURCE CODE

```
echo "-----"  
echo "\t For Loop Example"  
echo "-----"  
for i in 12 14 15 17 18 21  
do  
    echo $i  
done
```

OUTPUT



```
Console  Shell  
sh for1.sh  
-----  
For Loop Example  
-----  
12  
14  
15  
17  
18  
21  
sh
```

V. CHARACTERS AND STRING RETRIEVAL USING FOR LOOP

SOURCE CODE

```
echo "-----"  
echo "\t Char & String Retrieval using for loop"  
echo "-----"  
for i in 'Sachin' 'B' 'C' 'D' 'E'  
do  
    echo $i  
done
```

OUTPUT

```
Console Shell
GNU bash, version 4.4.20(1)-release (x86_64-pc-linux-gnu)
> sh for2.sh
-----
Char & String Retrieval using for loop
-----
Sachin
B
C
D
E
>
```

VI. DISPLAYING FILES AND DIRECTORIES USING FOR LOOP

SOURCE CODE

```
echo "-----"
echo "\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
```

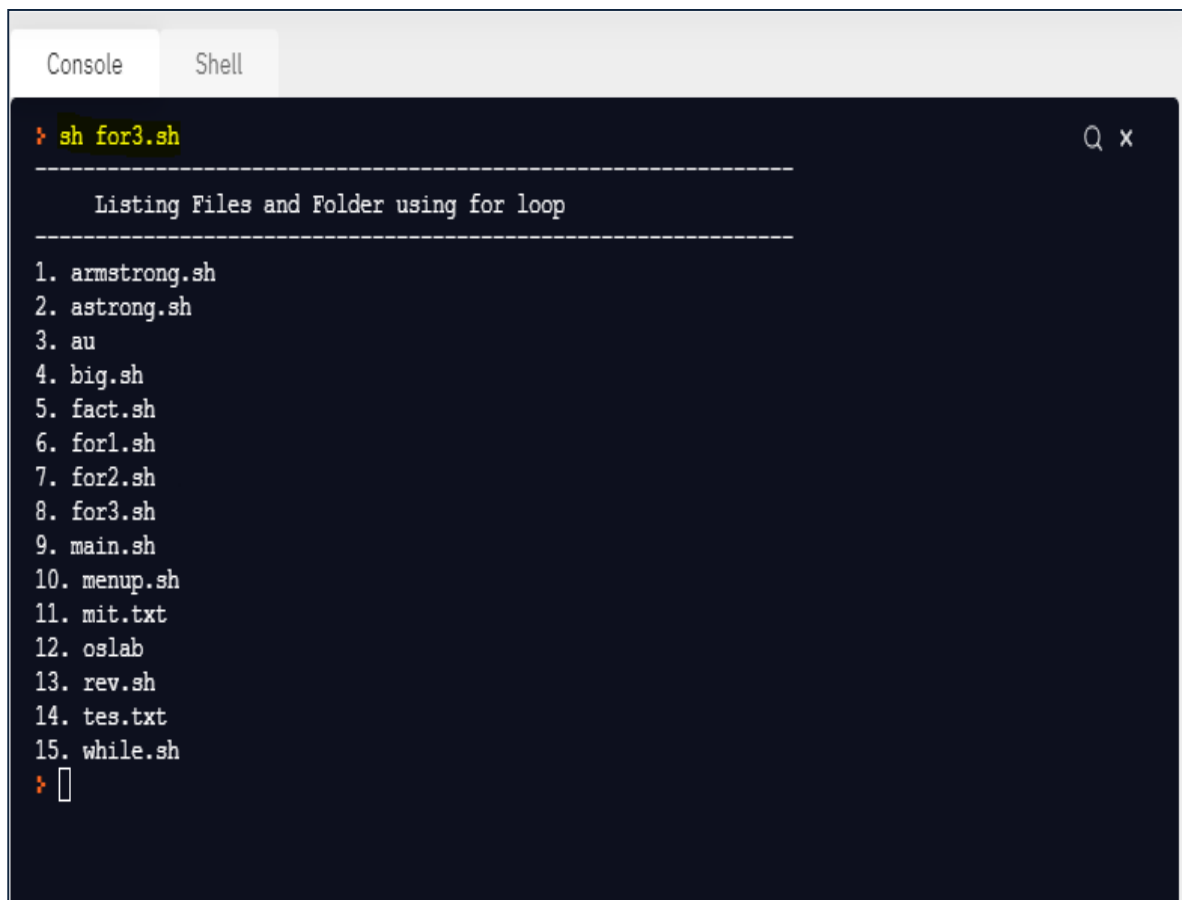
Store list of files to the variable fset using ls command.

k=k+1 or k++

Command Substitution:

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

OUTPUT



```
sh for3.sh
-----
Listing Files and Folder using for loop
-----
1. armstrong.sh
2. astrong.sh
3. au
4. big.sh
5. fact.sh
6. for1.sh
7. for2.sh
8. for3.sh
9. main.sh
10. menup.sh
11. mit.txt
12. oslab
13. rev.sh
14. tes.txt
15. while.sh
$
```

VII. FACTORIAL OF A NUMBER

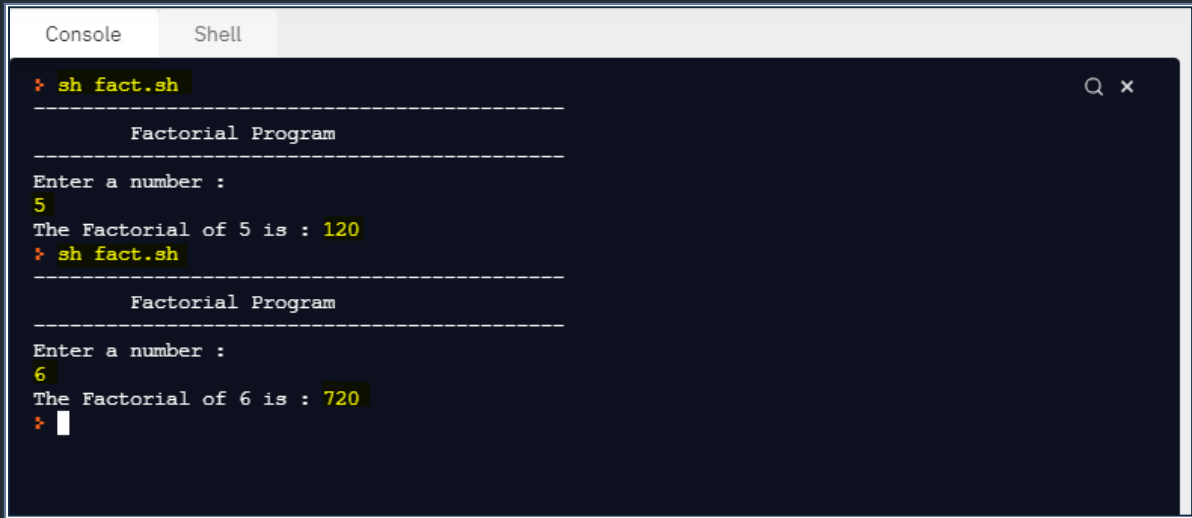
Language	:	shell (.sh)
Editor	:	replit.com (Online Linux Terminal)
OS	:	Windows 10

SOURCE CODE

```
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



```
Console Shell
> sh fact.sh
-----
Factorial Program
-----
Enter a number :
5
The Factorial of 5 is : 120
> sh fact.sh
-----
Factorial Program
-----
Enter a number :
6
The Factorial of 6 is : 720
> |
```

C STYLE CODING IN SHELL

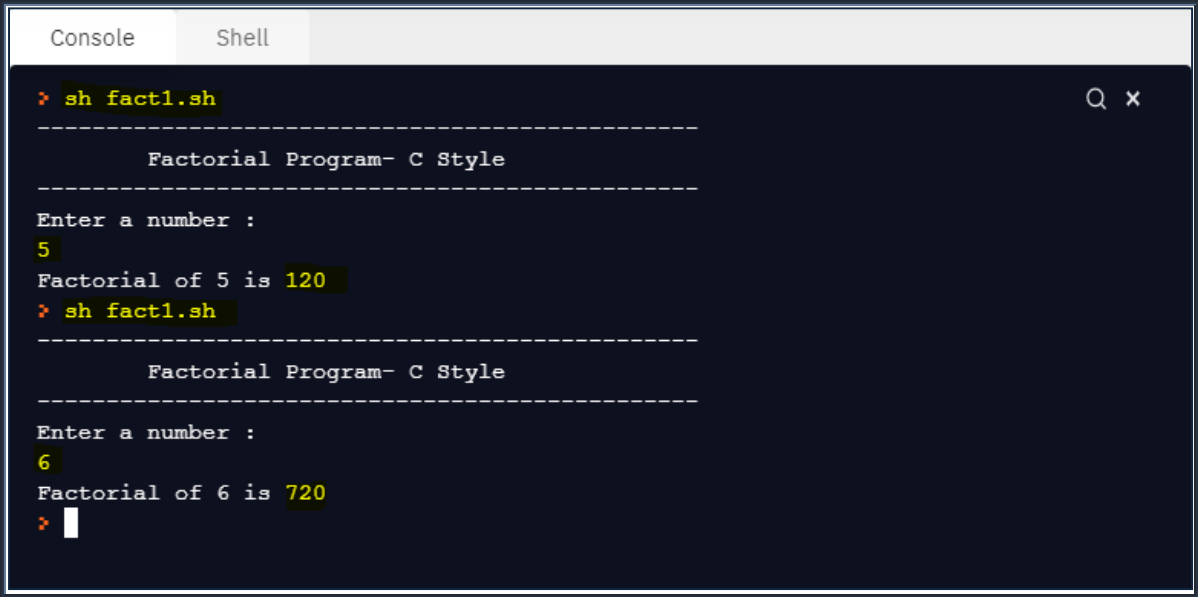
- In shell, we can use c style coding for looping statements and expressions
- The expressions are defined by **\$(exp)**. Here each is also closed by () symbol.

VIII. FACTORIAL OF A NUMBER USING C-STYLE

SOURCE CODE

```
echo "-----"
echo "\t\tFactorial Program- C Style"
echo "-----"
echo "Enter a number : "
read n
i=1
f=1
# while loop
while [ $i -le $n ]
do
# expression 1 in c-style
  f=$((f*i))
# expression 2 in c-style
  i=$((i+1))
done
echo "Factorial of $n is $f"
```

OUTPUT



```
Console Shell
> sh fact1.sh
-----
      Factorial Program- C Style
-----
Enter a number :
5
Factorial of 5 is 120
> sh fact1.sh
-----
      Factorial Program- C Style
-----
Enter a number :
6
Factorial of 6 is 720
> |
```

IX. REVERSE OF A NUMBER

SOURCE CODE

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

The diagram illustrates the execution flow of the shell script. Red arrows point from callout boxes to the corresponding code lines:

- Read a number**: Points to the `read n` line.
- while (n!=0)**: Points to the `while [$n -ne 0]` line.
- rem=n%10**: Points to the `rem=`expr $n % 10`` line.
- res=res*10**: Points to the `res=`expr $res * 10`` line.
- n=n/10**: Points to the `n=`expr $n / 10`` line.

OUTPUT

```
Console Shell
❯ sh rev.sh
-----
Reverse of Number
-----
Enter a number :
6724
The Reverse Number of 6724 is 4276
❯ sh rev.sh
-----
Reverse of Number
-----
Enter a number :
45
The Reverse Number of 45 is 54
❯ sh rev.sh
-----
Reverse of Number
-----
Enter a number :
12662
The Reverse Number of 12662 is 26621
❯
```

X. ARMSTRONG NUMBER

SOURCE CODE

```
echo "-----"
echo "\t\tArmstrong Number"
echo "-----"
echo "enter a number"
read n
# variables declarations
dup=$n
arm=0
# while loop
while test $n -ne 0
do
# find remainder of a number
rem=`expr $n % 10`
# multiply rem with three times
rem=`expr $rem \* $rem \* $rem`
# add the resultant remainder with arm variable
```

while (n!=0)

rem=n%10

rem=(rem*rem*rem)

```

arm=`expr $arm + $rem`
# divide n by 10
n=`expr $n / 10`
done
if [ $dup -eq $arm ]
then
    echo "Given Number $dup is Armstrong Number"
else
    echo "Given Number $dup is not Armstrong Number"
fi

```

n=n/10

OUTPUT

```

Console Shell
❯ sh armstrong.sh
-----
Armstrong Number
-----
enter a number
153
Given Number 153 is Armstrong Number
❯ sh armstrong.sh
-----
Armstrong Number
-----
enter a number
283
Given Number 283 is not Armstrong Number
❯

```

XI. MENU DRIVEN PROGRAM

SOURCE CODE

```

while [ true ]
do
    echo "-----"
    echo "\t\t Menu Program"
    echo "-----"
    echo "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 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
```

NOTE

- It is an 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
```

OUTPUT – (CON)

```
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
└─┘
```

COMMAND LINE ARGUMENTS (POSITIONAL PARAMETERS)

- Process of passing the input arguments to the program at the time of execution is called as command line arguments
- In shell, this is done with help of **\$** symbol

S.N	POSITIONAL PARAMETERS	DESCRIPTION
1.	\$0	Indicates the filename itself
2.	\$1	Indicates the first argument
3.	\$2	Indicates the second argument
	...	
4.	\$*	Represents the total number of input arguments which are submitted to the shell program
5.	\$#	Shows the count of total number of arguments passed to the shell program

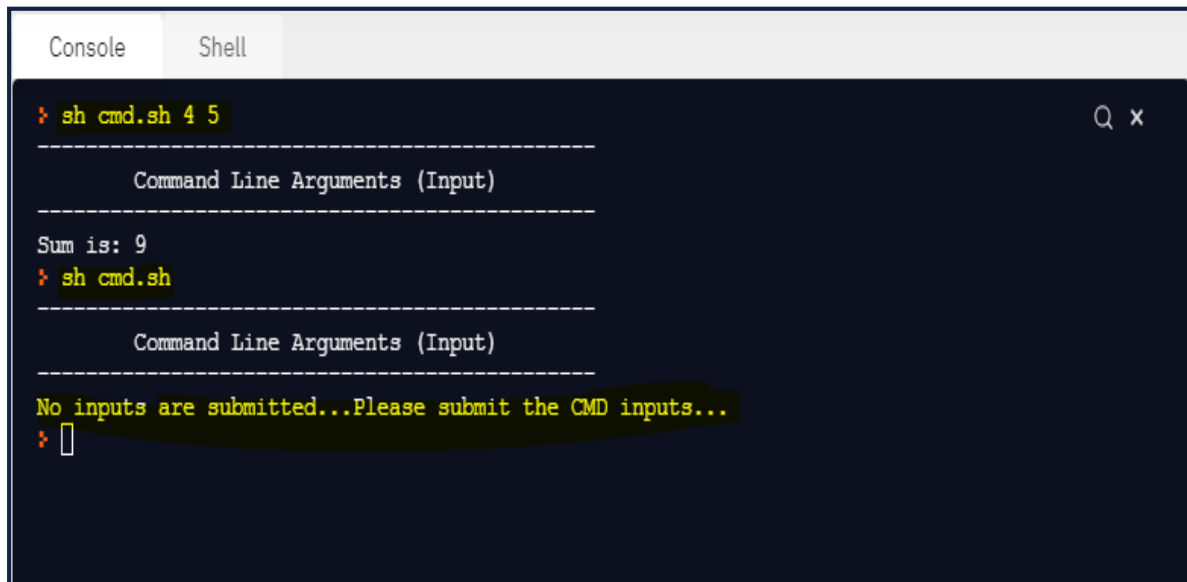
I. EXAMPLE OF COMMAND LINE INPUTS

SOURCE CODE

```
echo "-----"
echo "\t\tCommand Line Arguments (Input)"
echo "-----"
a=$1
b=$2
# check whether the command line arguments are submitted or not
if [ $# -ne 0 ]
then
    r=`expr $a + $b`
    echo "Sum is: $r"
else
    echo "No inputs are submitted...Please submit the CMD inputs..."
fi
```

This is equivalent to `if [$# != 0]`

OUTPUT

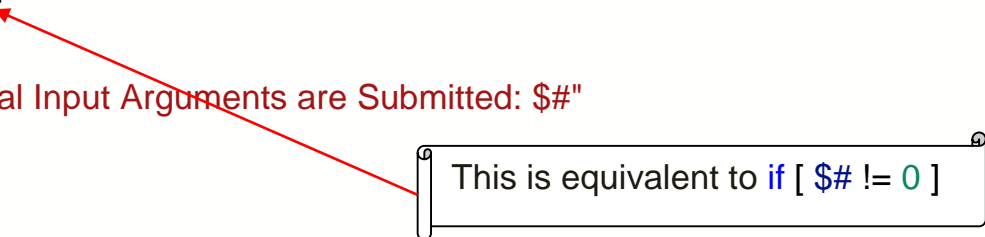


```
Console Shell
❯ sh cmd.sh 4 5
-----
Command Line Arguments (Input)
-----
Sum is: 9
❯ sh cmd.sh
-----
Command Line Arguments (Input)
-----
No inputs are submitted...Please submit the CMD inputs...
❯
```

II. EXAMPLE OF COMMAND LINE INPUTS – MUCH ARGUMENTS

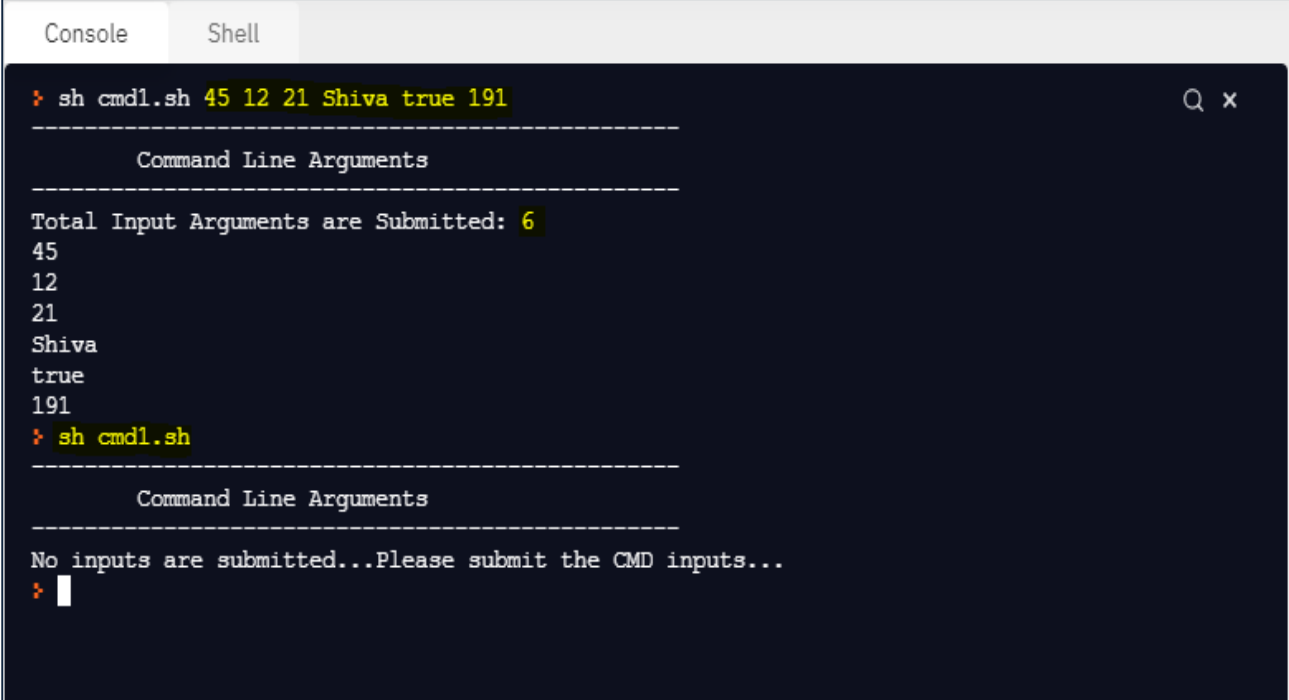
SOURCE CODE

```
echo "-----"
echo "\t\tCommand Line Arguments"
echo "-----"
# check whether the command line arguments are submitted or not
if [ $# -ne 0 ]
then
    echo "Total Input Arguments are Submitted: $#"
```



```
    for i in $*
    do
        echo $i
    done
else
    echo "No inputs are submitted...Please submit the CMD inputs..."
fi
```

OUTPUT



```
Console Shell
sh cmd1.sh 45 12 21 Shiva true 191
-----
Command Line Arguments
-----
Total Input Arguments are Submitted: 6
45
12
21
Shiva
true
191
sh cmd1.sh
-----
Command Line Arguments
-----
No inputs are submitted...Please submit the CMD inputs...
sh
```

RESULT

Thus the basics of shell programming was executed successfully.

EX.NO: 4 SHELL SCRIPTING – OPERATORS, FUNCTIONS

AIM

To practice the different types of operators using shell programming

OPERATORS

- A special symbol which is used to perform the various tasks such as arithmetic operations, relational operations, logical operations, file testing operations, comparisons, etc, ...

ARITHMETIC OPERATORS

S.N	OPERATOR	DESCRIPTION
1.	+	Addition
2.	-	Subtraction
3.	*	Multiplication
4.	/	Division

EXPRESSION IN SHELLS

- Shell provides two options for performing expressions in shell scripts. They are
 1. Using expr command → Shell style `expr`
 2. Using double braces () → C Style `$(exp)`

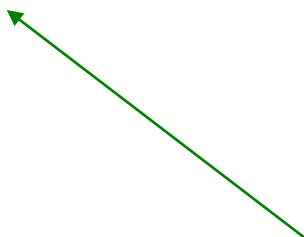
NOTE

- It is an important to note that, the operator `*` does **not provide the multiplication in shell expression using expr command**. Because in shell, the operator `*` means **wild card characters**.
- So, the backward slash followed by `*` symbol `*` is to provide the multiplication expression in shell expression using expr command.

I. EXAMPLE OF ARITHMETIC OPERATIONS USING EXPR COMMAND

SOURCE CODE

```
echo "-----"  
echo "\tArithmetic Operations using expr command"  
echo "-----"  
echo "Enter the number 1: "  
read a  
echo "Enter the number 2: "  
read b  
# performing arithmetic operations using expr command  
r1=`expr $a + $b`  
r2=`expr $a - $b`  
r3=`expr $a \* $b`  
r4=`expr $a / $b`  
# print the results  
echo "Add: \t$r1"  
echo "Sub: \t$r2"  
echo "Mul: \t$r3"  
echo "Div: \t$r4"
```



The usage of * in expr command does not directly support for multiplication. Because it gives different meanings. So the **backward slash with *** symbol provides the multiplication in the expr command.

OUTPUT

```
Console Shell
> sh arth1.sh
-----
Arithmetic Operations using expr command
-----
Enter the number 1:
15
Enter the number 2:
3
Add:    18
Sub:    12
Mul:    45
Div:    5
> 
```

II. EXAMPLE OF ARITHMETIC OPERATIONS USING C STYLE

SOURCE CODE

```
echo "-----"
echo "\tArithmetic Operations using C-Style"
echo "-----"
echo "Enter the number 1: "
read a
echo "Enter the number 2: "
read b
# performing arithmetic operations using expr command
r1=$((a+b))
r2=$((a-b))
r3=$((a*b))
r4=$((a/b))
# print the results
echo "Add: \t$r1"
echo "Sub: \t$r2"
echo "Mul: \t$r3"
echo "Div: \t$r4"
```

The Syntax is $\$(e1, e2, ..en)$.
Each expression e1, e2, ..en can be closed by ()

OUTPUT

```
Console Shell
> sh arth2.sh
-----
Arithmetic Operations using C-Style
-----
Enter the number 1:
12
Enter the number 2:
4
Add:    16
Sub:    8
Mul:    48
Div:    3
> 
```

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

III. EXAMPLE OF RELATIONAL OPERATORS USING C STYLE

SOURCE CODE

```
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: $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

```
Console Shell
> sh rel1.sh
-----
Relational Operators using C-Style
-----
Enter the number 1:
50
Enter the number 2:
21
Equal                : 0
Not Equal            : 1
Lesser than          : 0
Lesser than or Equal to : 0
Greater than         : 1
Greater than or Equal to : 1
> 
```

IV. EXAMPLE OF RELATIONAL OPERATORS FOR STRINGS

SOURCE CODE

```
echo "-----"
echo "\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

V. EXAMPLE OF LOGICAL AND OPERATOR

SOURCE CODE

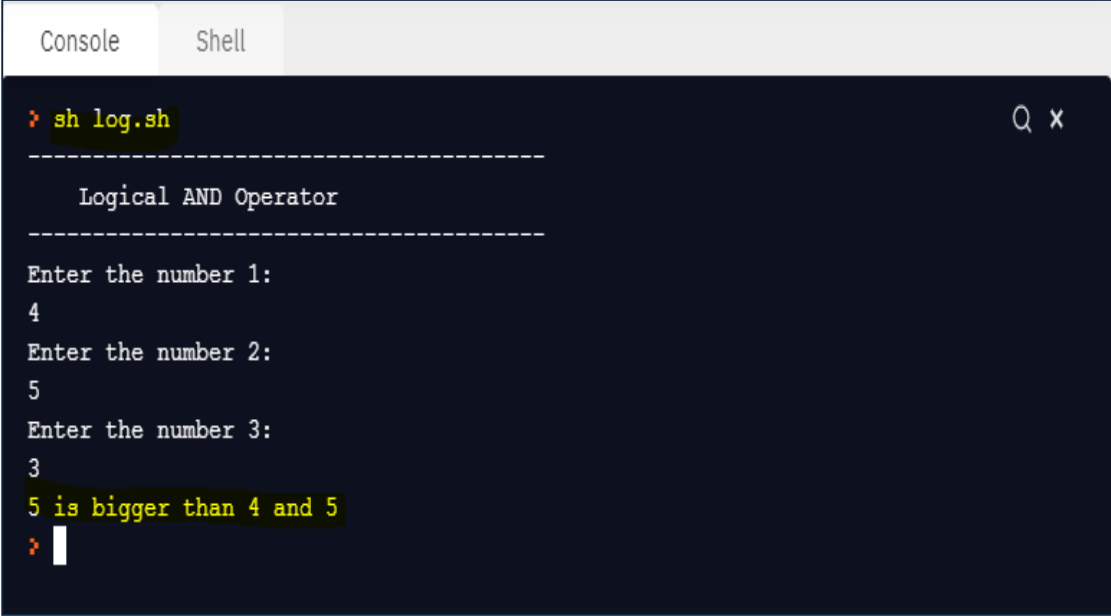
```
echo "-----"  
echo "\tLogical AND Operator"  
echo "-----"  
echo "Enter the number 1: "  
read a  
echo "Enter the number 2: "  
read b  
echo "Enter the number 3: "  
read c  
if [ $a -gt $b ] && [ $a -gt $c ]  
then  
    echo "$a is bigger than $b and $c"  
elif [ $b -gt $c ]  
then  
    echo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



```
echo "$c is bigger than $a and $b"
```

```
fi
```

OUTPUT



```
Console Shell
> sh log.sh
-----
Logical AND Operator
-----
Enter the number 1:
4
Enter the number 2:
5
Enter the number 3:
3
5 is bigger than 4 and 5
> |
```

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

VI. EXAMPLE OF FILE TEST OPERATORS

SOURCE CODE

```
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

```
Console Shell
> ls
5 arth2.sh city.txt hh.txt main.sh rel2.sh wel.txt
arth1.sh b filet.sh log.sh rel1.sh tt
> sh filet.sh
-----
File Testing Operator
-----
Enter the file name :
hh.txt
Object Exists...
It is a regular file...
Contents:
Good morning
> sh filet.sh
-----
File Testing Operator
-----
Enter the file name :
tt
Object Exists...
It is a directory...
/home/runner/Sh-Operators/tt
Contents of Directory:
hh.txt
wel.txt
> sh filet.sh
-----
File Testing Operator
-----
Enter the file name :
ffhsdjs
Object does not exists...
> |
```

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

VII. LISTING FILES AND FOLDERS IN ROOT DIRECTORY

SOURCE CODE

(test.sh)

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

Root Path: \$ ls /

OUTPUT

```
Console Shell
> sh test.sh
-----
Files and Folders in Root Directory
-----
bin
boot
config
dev
etc
gocode
home
inject
io
lib
lib32
lib64
media
mnt
nix
opt
phase2-assembly.tar.bz2
phase2-clisp.tar.bz2
phase2-cpp11.tar.bz2
phase2-cpp.tar.bz2
phase2-crystal.tar.bz2
phase2-csharp.tar.bz2
phase2-d.tar.bz2
phase2-elixir.tar.bz2
phase2-erlang.tar.bz2
phase2-express.tar.bz2
phase2-fortran.tar.bz2
phase2-fsharp.tar.bz2
phase2-guile.tar.bz2
phase2-haskell.tar.bz2
phase2-love2d.tar.bz2
phase2-mercury.tar.bz2
phase2-pascal.tar.bz2
phase2-php.tar.bz2
phase2-prolog.tar.bz2
phase2-react_native.tar.bz2
phase2-rlang.tar.bz2
proc
root
run
```

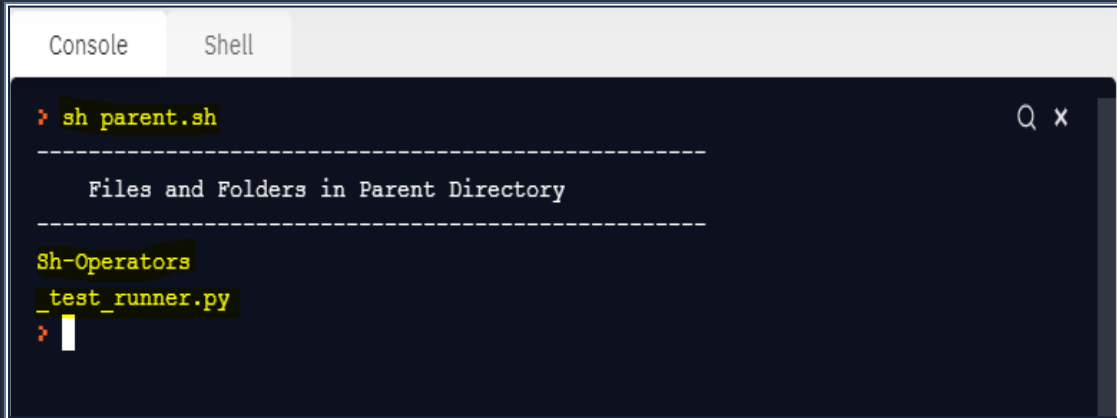
VIII. LISTING FILES AND FOLDERS IN PARENT DIRECTORY

SOURCE CODE

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

Path of Parent Directory: `$ ls ..`

OUTPUT



```
> sh parent.sh  
-----  
    \tFiles and Folders in Parent Directory  
-----  
Sh-Operators  
_test_runner.py  
>
```

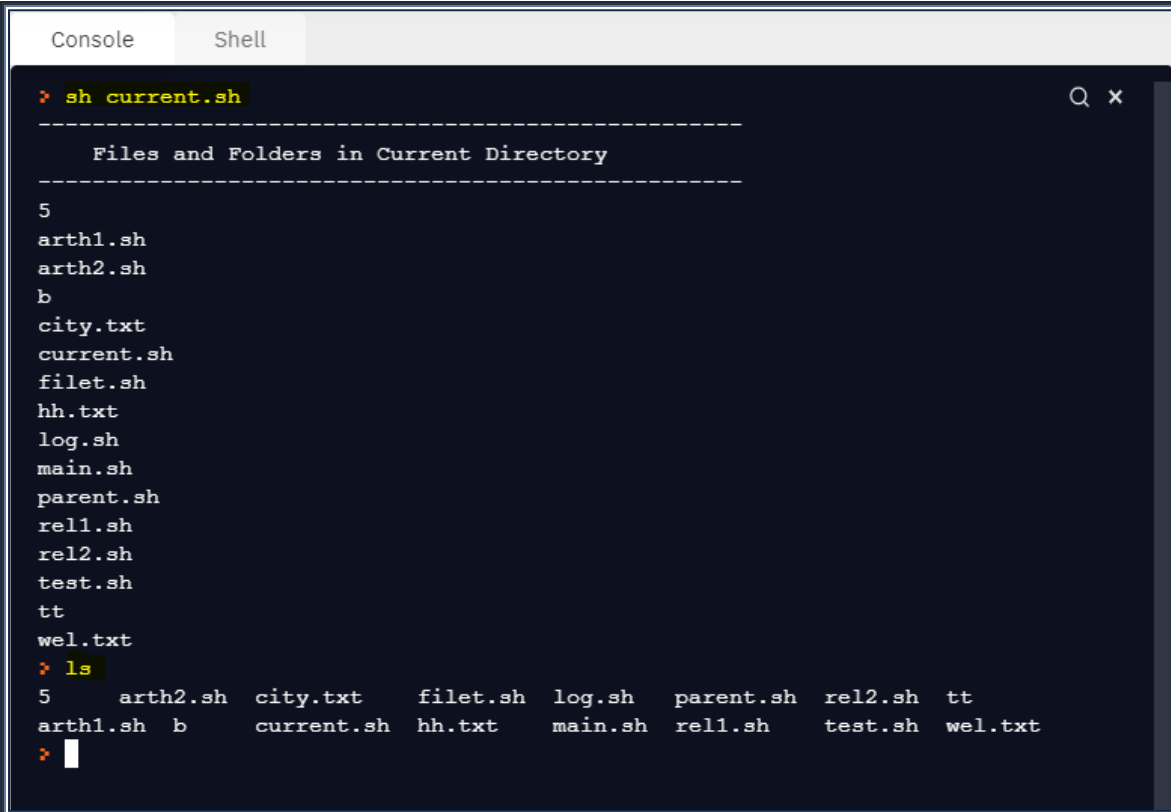
IX. LISTING FILES AND FOLDERS IN CURRENT DIRECTORY

SOURCE CODE

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

Path of Current Directory: `$ ls`

OUTPUT

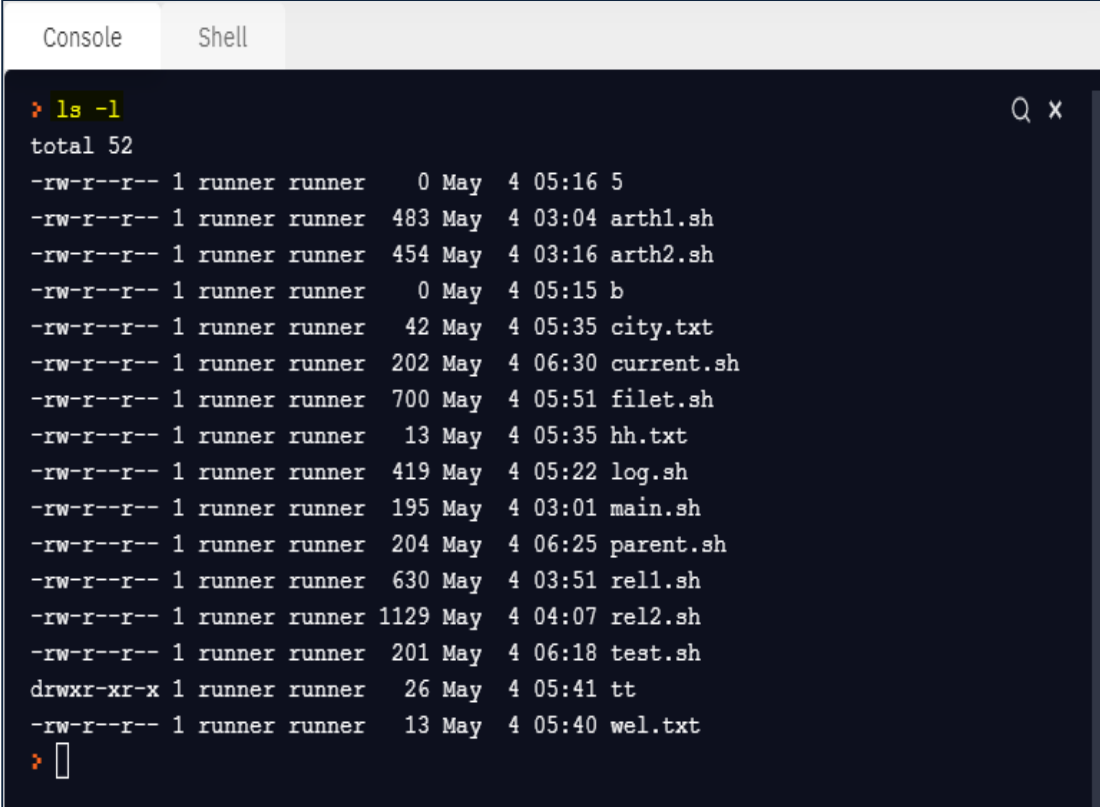


```
Console Shell
> sh current.sh
-----
Files and Folders in Current Directory
-----
5
arth1.sh
arth2.sh
b
city.txt
current.sh
filet.sh
hh.txt
log.sh
main.sh
parent.sh
rel1.sh
rel2.sh
test.sh
tt
wel.txt
> ls
5  arth2.sh  city.txt  filet.sh  log.sh  parent.sh  rel2.sh  tt
arth1.sh  b        current.sh  hh.txt  main.sh  rel1.sh  test.sh  wel.txt
> |
```

LIST FILES IN LONG FORMAT

- The command **ls -l** provides the detailed format of showing the files and folders in current file system.
 1. Content Permissions
 2. Number of links to the content
 3. Owner of the content
 4. Group owner of the content
 5. Size of the content (in bytes)
 6. Last modified date / time
 7. Name of the file / directory name

Long Listing of Files and Folders



```
Console Shell
> ls -l
total 52
-rw-r--r-- 1 runner runner  0 May  4 05:16 5
-rw-r--r-- 1 runner runner 483 May  4 03:04 arth1.sh
-rw-r--r-- 1 runner runner 454 May  4 03:16 arth2.sh
-rw-r--r-- 1 runner runner  0 May  4 05:15 b
-rw-r--r-- 1 runner runner  42 May  4 05:35 city.txt
-rw-r--r-- 1 runner runner 202 May  4 06:30 current.sh
-rw-r--r-- 1 runner runner 700 May  4 05:51 file1.sh
-rw-r--r-- 1 runner runner  13 May  4 05:35 hh.txt
-rw-r--r-- 1 runner runner 419 May  4 05:22 log.sh
-rw-r--r-- 1 runner runner 195 May  4 03:01 main.sh
-rw-r--r-- 1 runner runner 204 May  4 06:25 parent.sh
-rw-r--r-- 1 runner runner 630 May  4 03:51 rel1.sh
-rw-r--r-- 1 runner runner 1129 May  4 04:07 rel2.sh
-rw-r--r-- 1 runner runner 201 May  4 06:18 test.sh
drwxr-xr-x 1 runner runner  26 May  4 05:41 tt
-rw-r--r-- 1 runner runner  13 May  4 05:40 wel.txt
> █
```

CONTENT PERMISSION

- In content permission, the column 1 indicates the file type. They are
 - - means for regular file
 - d means for directory
 - b means for special block file
 - c means for special character file
- In content permission, the next column indicates the file permissions such as read, write and execute
- Read → represents the read mode which is code value 4
- Write → represents the write mode which is code value 2
- Execute → represents the execute mode which is code value 1.

SHELL FUNCTION

- Shell program supports the function which is used to implement the variables as well as execute the linux commands.

Syntax1

```
function <name>
{
    # user code
}
```

Example

```
function show
{
    # user code
}
```

Syntax2

```
<function-name>()
{
    # user code
}
```

Example

```
show()
{
    # user code
}
```

CALLING FUNCTION

- Shell function can be called **using its name only**.
- The operation **() should not be used while calling the function**.

Syntax

```
$ function-name
```

Example

```
$ show
```

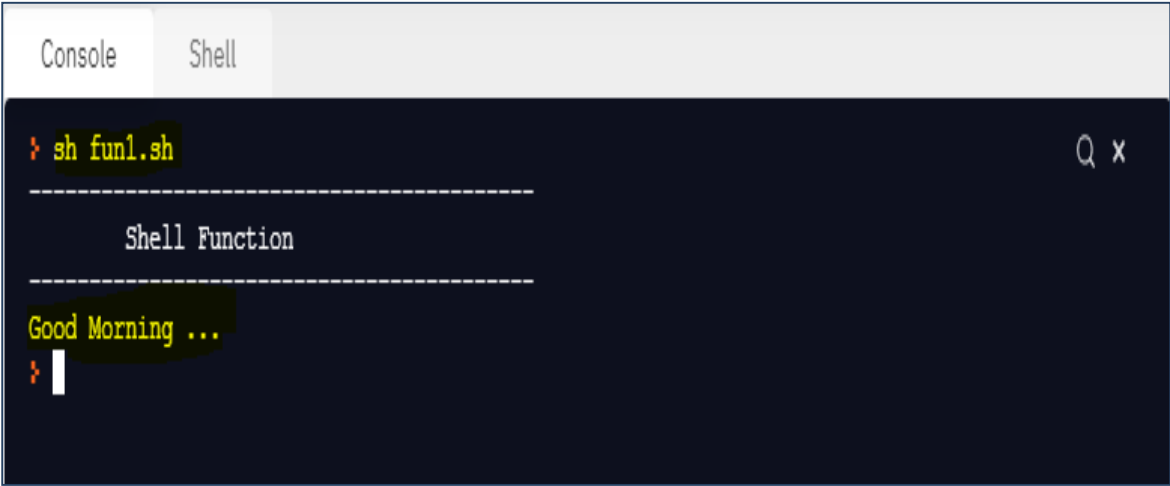
I. EXAMPLE OF SHELL FUNCTION

SOURCE CODE

```
echo "-----"  
echo "\t\tShell Function"  
echo "-----"  
disp()  
{  
    echo "Good Morning ..."  
}  
# calling function  
disp
```

The operator symbol () is not allowed while calling the shell function.

OUTPUT



```
Console Shell  
$ sh fun1.sh  
-----  
Shell Function  
-----  
Good Morning ...  
$
```

SHELL FUNCTION WITH ARGUMENTS

- Shell function supports the arguments
- The arguments are given after the function name while calling the function
- Each argument is separated by space
- The positional parameters like \$1, \$2, etc... will receive the values of function arguments inside the function definition

II. EXAMPLE OF SHELL FUNCTION WITH ARGUMENTS

SOURCE CODE

```
echo "-----"  
echo "\t\tShell Function with Arguments"  
echo "-----"  
# creating a shell function  
disp()  
{  
  a=$1  
  b=$2  
  rs=`expr $a + $b`  
  echo "Sum is: $rs"  
}  
# calling function with arguments  
disp 12 21
```

OUTPUT



```
Console Shell  
sh fun2.sh  
-----  
Shell Function with Arguments  
-----  
Sum is: 33  
sh
```

SHELL FUNCTION WITH RETURN STATEMENTS

- Like c language, shell function returns the values
- The return keyword is used to return the values in the shell function
- The returned result of calling function can be obtained using the special symbol \$? (by default, the return values of the function will be stored to the built-in variable \$?)

NOTE

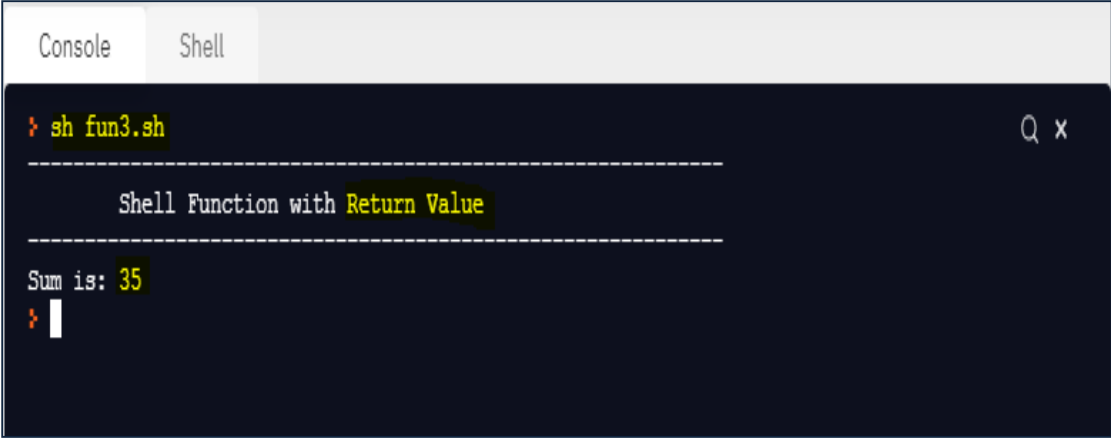
- It is an important to note that, the shell function will **return a single value**.

III. EXAMPLE OF SHELL FUNCTION WITH RETURN VALUE

SOURCE CODE

```
echo "-----"
echo "\tShell Function with Return Value"
echo "-----"
a=12
b=23
rs=0
# shell function
disp()
{
    rs=`expr $a + $b`
# return the variable
    return $rs
}
# calling function
disp
# get the return value from function using $?
k=$?
echo "Sum is: $k"
```

OUTPUT



```
Console Shell
❯ sh fun3.sh
-----
Shell Function with Return Value
-----
Sum is: 35
❯
```

COMMAND SUBSTITUTION

- In shell, assigning the built-in command to a user defined variable is called as command substitution
- This is done using the special symbol ``command-name`` or `$(command-name)`

Syntax1

```
Variable-Name=`command-name`
```

Example

```
files=`ls`           # the output of ls command is stored to the
                    # variable called files.
```

```
dirpath="test"
files=`ls $dirpath` # the contents of test directory are stored
                    # to the variable called files
```

Syntax 2

```
Variable-Name=$(command-name)
```

Example

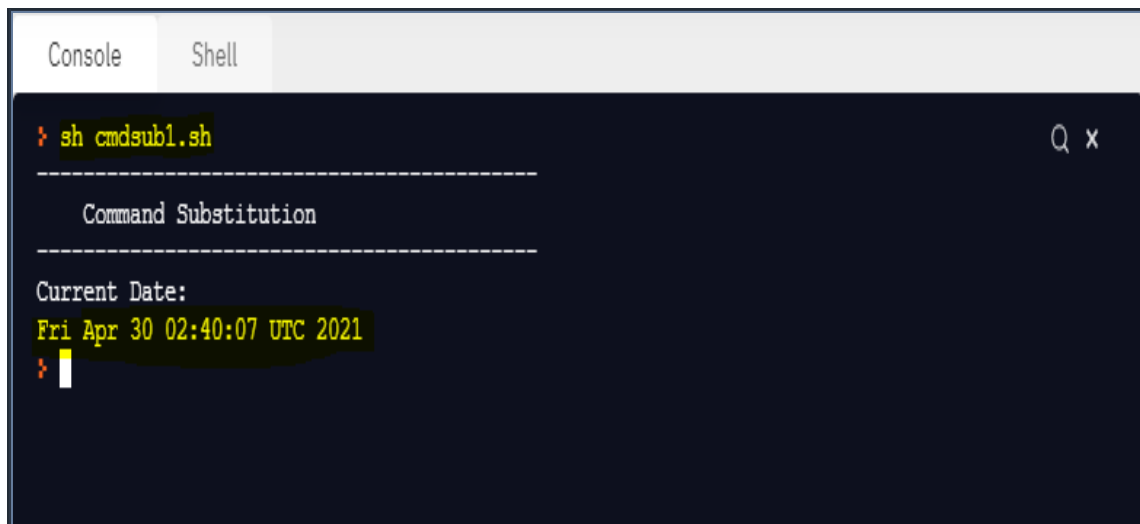
```
files=$(ls)         # the output of ls command is stored to the
                    # variable called files.
```

I. EXAMPLE OF COMMAND SUBSTITUTION

SOURCE CODE

```
echo "-----"  
echo "\tCommand Substitution"  
echo "-----"  
# date command Substitution  
rs=`date`  
# k=$(date)  
# print the output of date command via variable  
echo "Current Date: \n"$rs
```

OUTPUT



```
Console Shell  
sh cmdsub1.sh  
-----  
Command Substitution  
-----  
Current Date:  
Fri Apr 30 02:40:07 UTC 2021  
sh
```

II. COMMAND SUBSTITUTION-FILE COPYING BETWEEN DIRECTORIES

SOURCE CODE

```
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  
❯
```

RESULT

Thus the operators and shell functions of the shell programming have been executed successfully.

EX.NO: 5

SHELL ARRAYS

AIM

To practice the arrays in shell programming.

SHELL ARRAY

- Array is a set of elements which are having same type or different type
- It is an example of **linear data structures** (sequential data structure)
- It is accessed by the **index number** which starts from **0 to n-1**
- It supports the storage, retrieval, insertion, deletion and updation
- In shell, the array can be created with help of the special operator symbol **()** **instead of square operator []**
- Unlike other compiled languages **c/c++/c#/java**, the shell follows the dynamic data typed system. So no need to mention the data type in the creation of an array.

Syntax

```
User-defined-name = (Element 1 Element 2 ...Element n)
```

Where,

Element 1, Element 2, ...Element n can be Same Type or Different Type.

Example

```
arr=(12 34 59) // homogenous collection
arr=("Sachin" 12.41 55 true) // heterogeneous
```

Length

- Array length can be done by using the special symbol **#** followed by **@** or ***** symbol along with array name
- It is very important to note that, the curly braces operators **{}** are used for accessing the array in shell

Syntax

```
{#<array-name>[@]}
```

(OR)

```
{#<array-name>[*]}
```


Example

```
ls=(12 34 55 99)
len=${#ls[@]}
(OR)
len=${#ls[*]}
```

Accessing Individual array contents

- The contents of the array can have accessed by using its index location

Example

```
ls[0]    → 12
ls[1]    → 32
```

Printing Individual Element

- The individual element of an array can be displayed using the curly braces `{}` along with array name

Syntax

```
${arrayname[index-number]}
```

Example

```
echo ${ls[1]}    // display the second element
echo ${ls[3]}    // display the fourth element
```

Accessing Entire array contents

- The entire array elements can be called using the star symbol `*` or `@` with array name.
- It is an important to note that, **if the index number is `*` or `@`, the whole elements of the array are referenced.**

Example

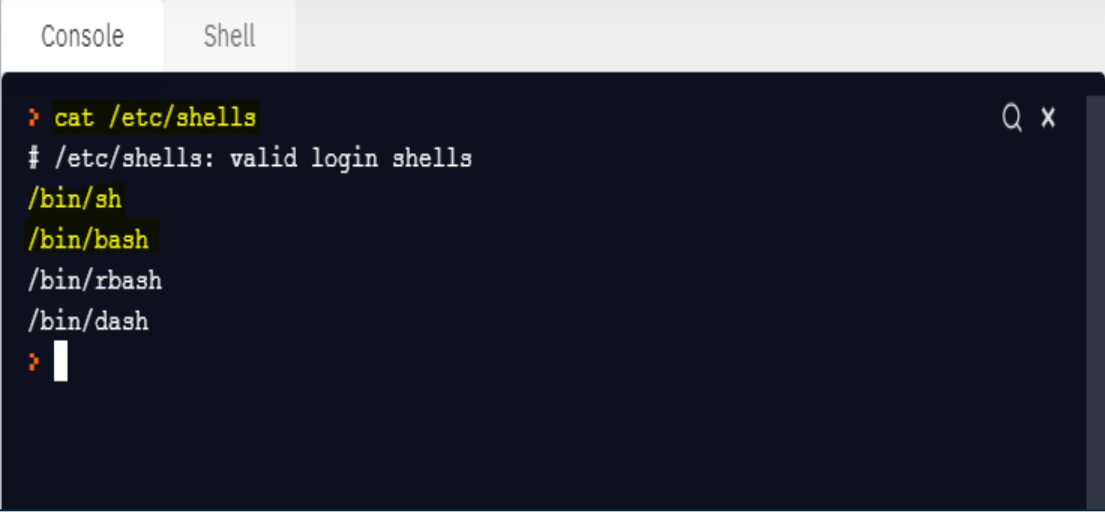
```
${ls[*]}    → contains the entire array elements
echo ${ls[*]}    → show the array contents at the same time.
```

OR

<code>\${ls[@]}</code>	→ contains the entire array elements
<code>echo \${ls[@]}</code>	→ show the array contents at the same time.

Commands to check the Shell Types

\$ cat /etc/shells



```
> cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/bash
/bin/rbash
/bin/dash
> |
```

I. EXAMPLE OF ARRAY CREATION FOR HOMOGENOUS TYPE

(/home/runner/Latest-Shells-21/[arr1.sh](#))

SOURCE CODE

```
echo "-----"
echo "Shell Array for Same Type of Elements"
echo "-----"
# Array Creation
lb=(12 45 77 99 88)
# print the contents of array at the same time
echo "Array Contents using name"
echo ${lb[*]}
# print the length of the array
echo "Length of the Array: ${#lb[@]}"
echo "Array Contents using for loop"
echo "-----"
```

Same type elements of array.

Display the array contents at the same time using the name.

```
# print the contents of array
for i in ${lb[*]}
do
    echo $i
done
```

Display the array contents one by one using for loop.

OUTPUT

```
> bash arr1.sh

-----
Shell Array for Same Type of Elements
-----
Array Contents using name
12 45 77 99 88
Length of the Array: 5
Array Contents using for loop
-----
12
45
77
99
88
> 
```

II. EXAMPLE OF ARRAY CREATION FOR HETEROGENEOUS TYPE

(/home/runner/Latest-Shells-21/arr2.sh)

SOURCE CODE

```
echo "-----"
echo "Shell Array for Different Type of Elements"
echo "-----"
# Array Creation
lb=("Shiva" 21 72.93 True)
# print the contents of array at the same time
echo "Array Contents using name"
echo ${lb[*]}
# print the length of the array
```

Different type elements of array.

Display the array contents at the same time using the name.

```

echo "Length of the Array: ${#lb[@]}"
echo "Array Contents using for loop"
echo "-----"
# print the contents of array
for i in ${lb[*]}
do
    echo $i
done

```

Display the array contents one by one using for loop.

OUTPUT

```

> bash arr2.sh
-----
Shell Array for Different Type of Elements
-----
Array Contents using name
Shiva 21 72.93 True
Length of the Array: 4
Array Contents using for loop
-----
Shiva
21
72.93
True
>

```

III. INDEX BASED ARRAY CREATION FOR HETEROGENEOUS TYPE

(/home/runner/Latest-Shells-21/arr3.sh)

SOURCE CODE

```

echo "-----"
echo "Array Creation using Index"
echo "-----"
# Empty array
ls=()
# store the elements to array using index number
ls[0]=12

```

Empty array creation

```
ls[1]=21
ls[2]='M'
ls[3]=True
ls[4]="Rohit"
```

Storing elements to array using index number.

```
# print the contents of array at the same time
```

```
echo "Array Contents using name"
```

```
echo ${ls[*]}
```

```
# print the length of the array
```

```
echo "Length of the Array: ${#ls[@]}"
```

```
echo "Array Contents using for loop"
```

```
echo "-----"
```

```
# print the contents of array
```

```
for i in ${ls[*]}
```

```
do
```

```
    echo $i
```

```
done
```

Printing the array length.

OUTPUT

```
Console Shell
> bash arr3.sh
-----
Array Creation using Index
-----
Array Contents using name
12 21 M True Rohit
Length of the Array: 5
Array Contents using for loop
-----
12
21
M
True
Rohit
> |
```

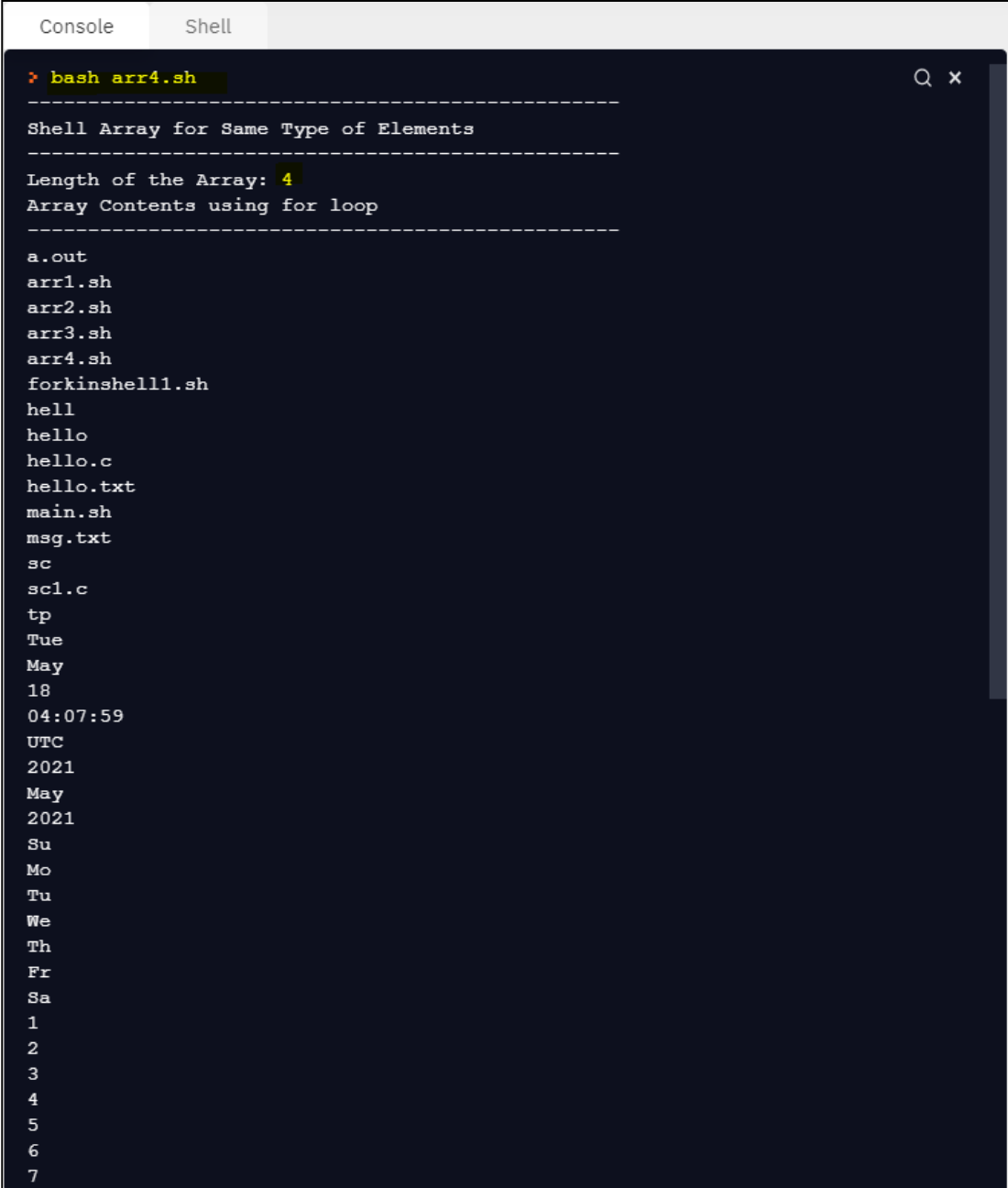
IV. DISPLAYING LINUX COMMANDS USING ARRAY

(/home/runner/Latest-Shells-21/arr4.sh)

SOURCE CODE

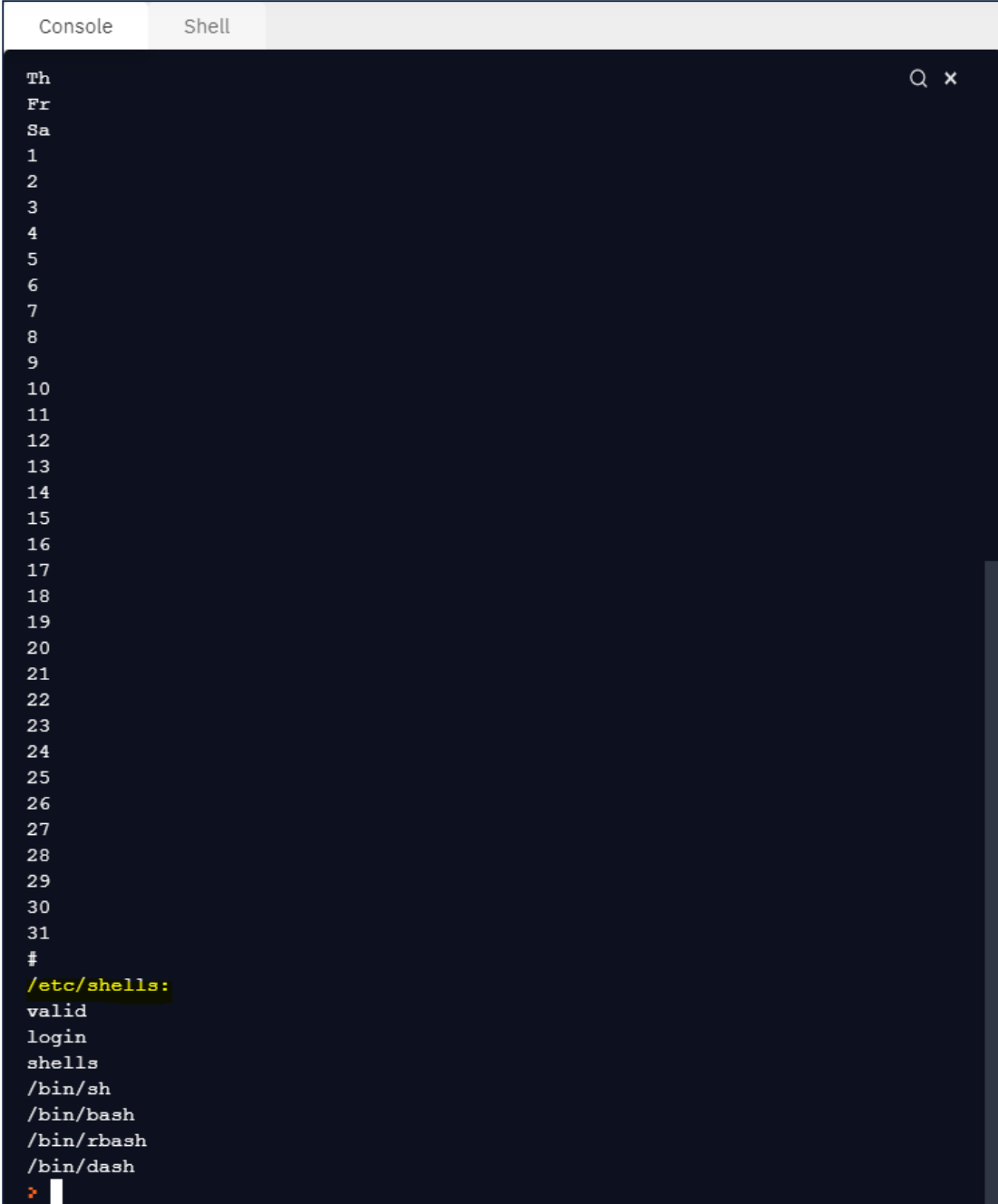
```
echo "-----"
echo "Shell Array for Same Type of Elements"
echo "-----"
# Array Creation
lb=()
# assigning linux commands to the each location of array
lb[0]=$ls)
lb[1]=$date)
lb[2]=`cal`
lb[3]=`cat /etc/shells`
# print the length of the array
echo "Length of the Array: ${#lb[@]}"
echo "Array Contents using for loop"
echo "-----"
# print the contents of array
for i in ${lb[*]}
do
    echo $i
done
```

DISPLAYING THE OUTPUT OF LINUX COMMANDS



```
Console Shell
> bash arr4.sh
-----
Shell Array for Same Type of Elements
-----
Length of the Array: 4
Array Contents using for loop
-----
a.out
arr1.sh
arr2.sh
arr3.sh
arr4.sh
forkinshell1.sh
hell
hello
hello.c
hello.txt
main.sh
msg.txt
sc
scl.c
tp
Tue
May
18
04:07:59
UTC
2021
May
2021
Su
Mo
Tu
We
Th
Fr
Sa
1
2
3
4
5
6
7
```

DISPLAYING THE OUTPUT OF LINUX COMMANDS – (CONTINUE)



```
Console Shell
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
#
/etc/shells:
valid
login
shells
/bin/sh
/bin/bash
/bin/rbash
/bin/dash
>
```

RESULT

Thus the arrays using shell programming have been executed successfully.

EX.NO: 6

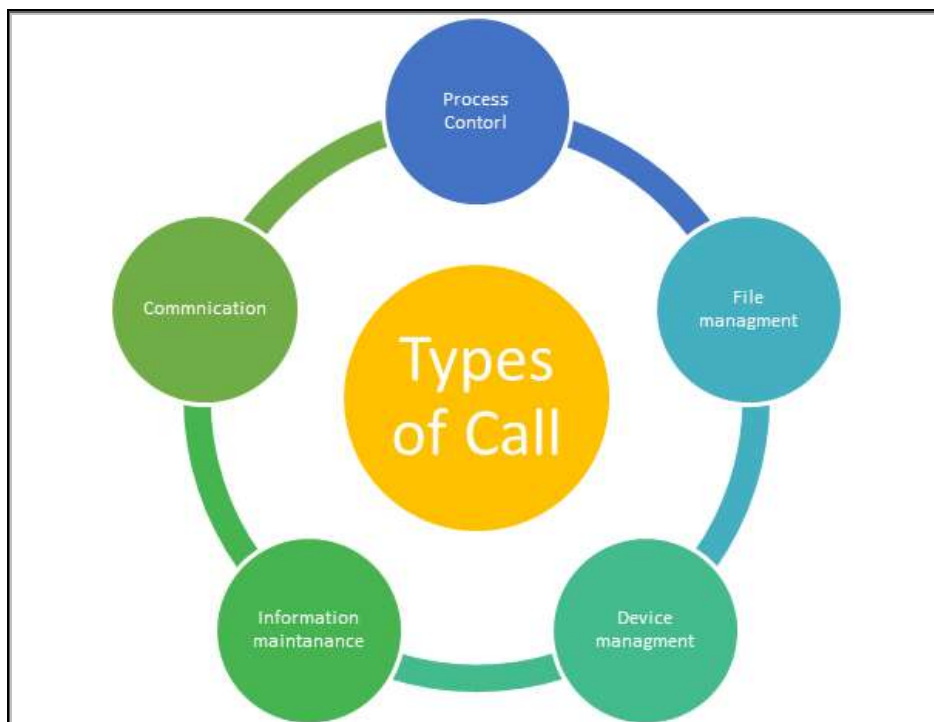
PROCESS SYSTEM CALLS – FORK, EXIT, WAIT

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



1. Process Control

- This system calls perform the task of process creation, process termination, etc.

Functions

- End and Abort
- Load and Execute
- Create Process and Terminate Process
- Wait and Signed Event
- Allocate and free memory

2. File Management

- It handles file manipulation jobs like creating a file, reading, and writing, etc.

Functions

- Create a file
- Delete file
- Open and close file
- Read, write, and reposition
- Get and set file attributes

3. Device Management

- It performs the job of device manipulation like reading from device buffers, writing into device buffers, etc.

Functions

- Request and release device
- Logically attach/ detach devices
- Get and Set device attributes

4. Information Management

- It handles information and its transfer between the OS and the user program.

Functions

- Get or set time and date

- Get process and device attributes

5. Communication

- These types of system calls are specially used for interprocess communications.

Functions

- Create, delete communications connections
- Send, receive message
- Help OS to transfer status information
- Attach or detach remote devices

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)**
- The new process gets a copy of the current program, but new process id (pid). The process id of the parent process (the process that called fork()) is registered as the new processes parent pid (ppid) to build a process tree.

- 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 **as child process**
- It is identified by the return code is **0**

Parent Process

- The caller of the newly created process is called as parent process
- It is identified by the return code is **positive value**

Formula

Total number of processes (T)	:	2^n
Total number of Child processes (C)	:	$2^n - 1$
Total number of Child processes (P)	:	$T - C$

NOTE

- **After the fork(), both parent and child processes are running simultaneously**
- The newly created process is called **as child process** which is identified by the return code is **0** and the caller of the child process is called as parent process which is identified by **positive value (>0)**
- **Program statements before fork() is common for both child and parent processes but after fork() call, the rest of the program instructions will be allocated separately for child and parent process**
- **Child and parent processes don't share common address space. They are having own memory address space. So if there are any changes in child process won't reflect the parent process. Similarly, if there are any changes in parent processes won't reflect the child process.**

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

- It is 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.).
- Depending on **how the process is added to the scheduler, either process may be scheduled first** after returning from fork.

REQUIRED HEADER FILES

1. #include<stdio.h>

- This header file is used for [printf\(\)](#), [scanf\(\)](#), etc, ...

2. #include<unistd.h>

- This header file is used for [fork\(\)](#), [getpid\(\)](#), [getppid\(\)](#), etc, ...

3. #include<sys/types.h>

- This header file is used for [pid_t](#), etc, ...

4. #include<sys/wait.h>

- This header file is used for [wait\(\)](#), etc, ...

5. #include<stdlib.h>

- This header file is used for [exit\(\)](#), etc, ...

TOOLS SUPPORT

Compiler	:	gcc compiler	(gcc <filename>.c)
Execution	:	./a.out	(assembly output)
OS	:	Linux OS platform	
Terminal	:	Online linux terminal	(www.replit.com)

I. EXAMPLE OF SINGLE FORK

(fork1.c)

SOURCE CODE

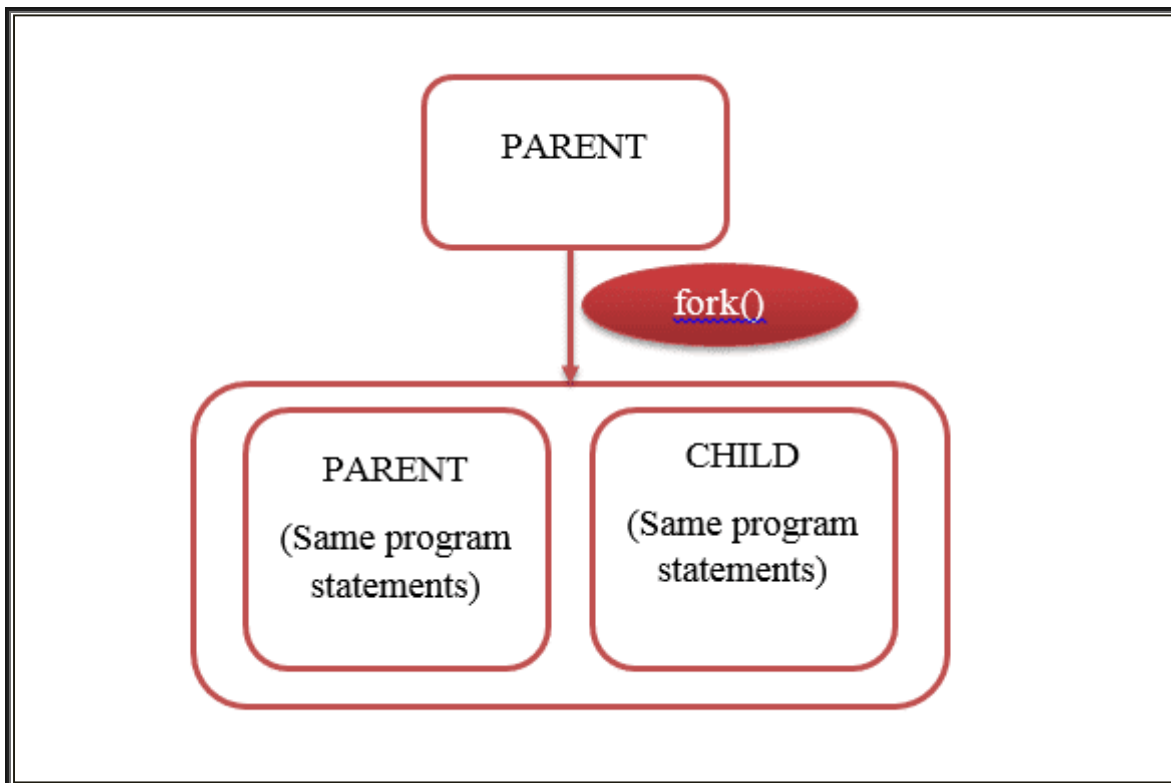
```
#include<unistd.h>
#include<stdio.h>
int main()
{
    printf("-----\n");
    printf("\tSimple Fork()\n");
    printf("-----\n");
    // calling fork()
    fork();
    printf("Hello World\n");
    return 0;
}
```

Total Number of Processes are $\rightarrow 2^n$
 $\rightarrow 2^1$
 $\rightarrow 2$

OUTPUT

```
Console Shell
> gcc fork1.c
> ./a.out
-----
\tSimple Fork()
-----
Hello World
Hello World
> █
```

Pictorial Representation



- When the **child process is created**, both the parent process and the child process will point to the next instruction (same Program Counter) after the fork().
- In this way the remaining instructions or C statements will be executed the total number of process times, that is **2^n times**, where **n** is the number of fork() system calls

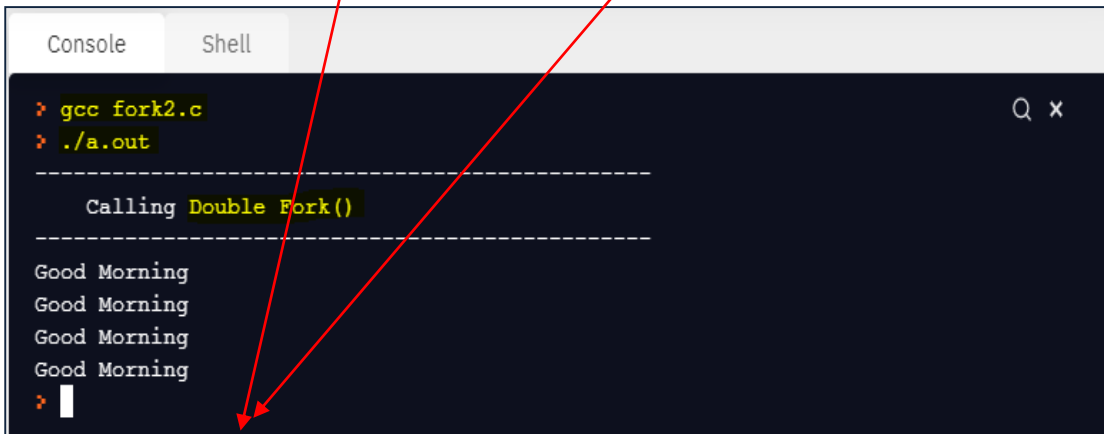
II. CALLING DOUBLE FORK

SOURCE CODE

```
#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;
}
```

Total Number of Processes are → 2^n
→ 2^2
→ 4

OUTPUT



III. CALLING TRIPLE FORK

SOURCE CODE

```
#include<unistd.h>
#include<stdio.h>
int main()
{
    printf("-----\n");
    printf("\tCalling Triple Fork()\n");
    printf("-----\n");
    // calling triple fork()
    fork();
    fork();
    fork();
    printf("Welcome to Chennai\n");
    return 0;
}
```

Total Number of Processes are → 2^n
→ 2^3
→ 8

OUTPUT

```
Console Shell
> gcc fork3.c
> ./a.out
-----
Calling Triple Fork()
-----
Welcome to Chennai
Welcome to Chennai
> Welcome to Chennai
Welcome to Chennai
Welcome to Chennai
Welcome to Chennai
Welcome to Chennai
Welcome to Chennai
Welcome to Chennai
```

IV. 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()
```

```
{
```

```
    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;
```

```
}
```

This header supports the `pid_t`

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
```

Usage of getpid() and getppid()

- Two major functions which are used to get the process ids. They are
 1. getpid()
 2. getppid()

1. getpid()

- It is a built-in function and available in `#include<unistd.h>` header file
- It is used to return the process ID of child process (newly created process)
- Return value: `pid_t`

2. getppid()

- It is a built-in function and available in `#include<unistd.h>` header file
- It is used to return the process ID of parent process (caller of the newly created process)
- Return value: `pid_t`

pid_t

- It stands for `process id type` and built-in variable to store the process ids of parent and child function
- It is the `type of the process ID` which returns an `unsigned integer value`.
- It is available in `#include<sys/types.h>`

V. 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;
}
```

OUTPUT

```
Console Shell
> gcc pid1_fork.c
> ./a.out
-----
Process ID of Parent & Child
-----
Parent Process is calling ...
Process ID (PID) of Parent Process : 15
Child Process is calling ...
Process ID (PID) of Child Process : 1037
> |
```

Wait()

- It is system call and available in `#include<sys/wait.h>` file
- It blocks the current process (calling process), until one of its child processes terminate 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 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 process ID of any terminated arbitrary child process.**

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

1. If **there are at least one child processes 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
- Wait(NULL) simply making the parent wait for the child.
- On success, wait() returns the process ID of terminated child process while on failure it returns **-1**
- **Once child process finishes, parent resumes and prints the rest of the statements of parent process**

exit()

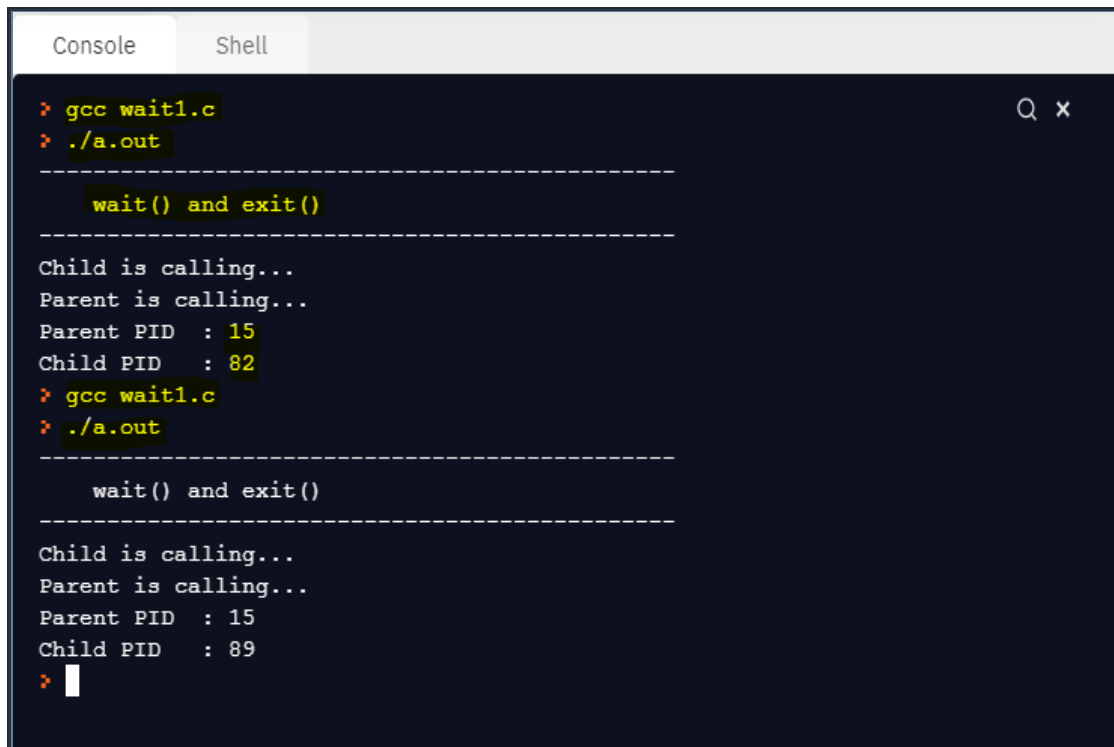
- It is system call and available in `#include<stdlib>` file
- It takes only one parameter which is exit status as a parameter
- It is used to close all files, sockets, frees all memory and then terminates the process.
- The parameter **0** indicates that the termination is normal.

VI. EXAMPLE OF WAIT AND EXIT SYSTEM CALLS

SOURCE CODE

```
#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", getpid());
        printf("Child PID\t: %d\n", pd);
    }
    return 0;
}
```

OUTPUT



```
Console Shell
> gcc wait1.c
> ./a.out
-----
wait() and exit()
-----
Child is calling...
Parent is calling...
Parent PID : 15
Child PID : 82
> gcc wait1.c
> ./a.out
-----
wait() and exit()
-----
Child is calling...
Parent is calling...
Parent PID : 15
Child PID : 89
> |
```

VII. SUM OF NUMBERS IN ARRAY USING CHILD AND PARENT PROCESS

Task

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

Used System calls:

fork(), wait()

SOURCE CODE

```
#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 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 ...
> |
```

RESULT

Thus the types of process system calls have been executed successfully.

EX.NO: 7

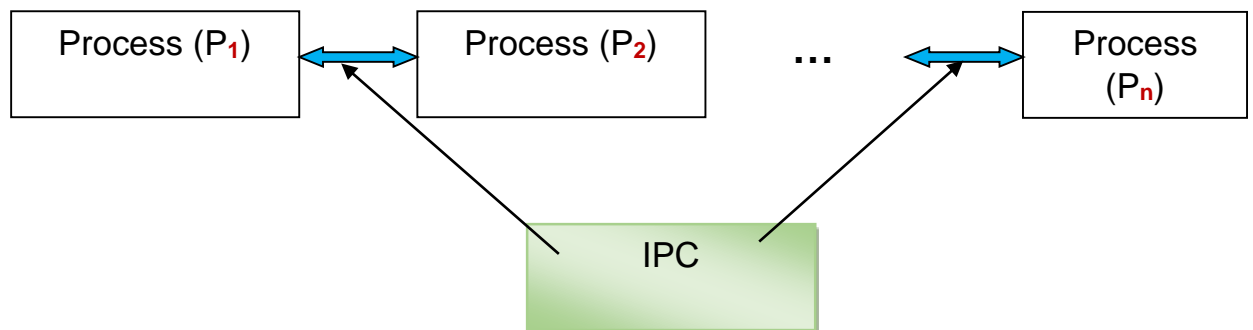
INTERPROCESS COMMUNICATION USING PIPE

AIM

To work with interprocess communication using pipe in linux c programming.

IPC

- IPC stands for **Inter Process Communication**
- It is one of the mechanism provided by the OS which **allows processes to communicate with each other** and synchronize their action
- Examples: **Message Queue, Signals, Shared Memory, Pipes**, etc,...



EXAMPLES TECHNIQUES OF IPC

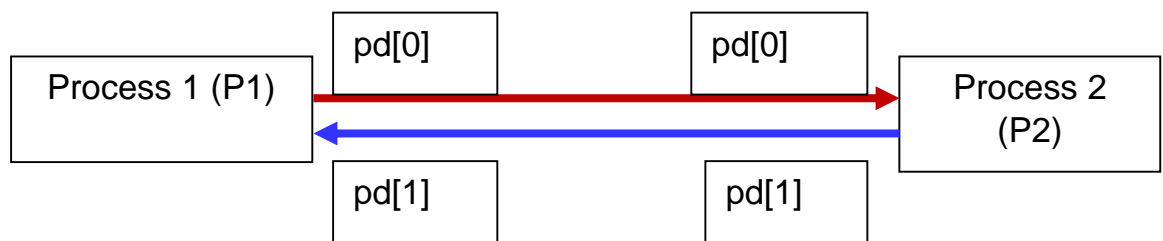
- Shared Files
- Shared Memory
- Pipes (Named and Unnamed Pipes)
- Message Queues
- Sockets
- Signals

PIPES

- Pipe provides the **communication between processes on same computer or different computer or across the network**
- It is classified as two types. They are
 1. Unnamed Pipe (PIPE)
 2. Named Pipe (FIFO)

Pipe Symbol

- The output of one command (read) can be given as the input of next command (write) that is called as pipe command. In linux terminal, it is indicated the symbol |
- **Two processes can be joined or communicated with help of the pipe symbol on the shell terminal**



- Both processes **P1, P2** are executed **simultaneously** and P1 will pass data / message to Process P2 as it executes.

Example of Pipe Symbol in Linux Terminal

```
Console Shell
> ls
a.out  arr2.sh  arr4.sh  hell  hello.c  main.sh  sc  tp
arr1.sh  arr3.sh  forkinshell1.sh  hello  hello.txt  msg.txt  sc1.c  wel.txt
> ls | wc -l
16
>
```

- Here the output of **ls** command is given as an input to the **wc -l** command using **pipe symbol (|)**.
- Hence, both **ls** command (**Process P1**) and **wc -l** command (**Process P2**) can be **communicated through pipe symbol (|)**.
- Using pipe symbol, 'n' of processes or commands can be communicated.

I. IPC USING PIPE

PIPE SYSTEM CALL

- In linux c, the pipe system call is created by using the built-in function called `pipe()`

Example

```
int pipe(int pd[2]);
```

- Here, pipe system call accepts **only one argument, which is an integer array of two pipe descriptors.**
- **pd[0]** is used for **reading data / message** from the pipe and **pd[1]** is used for **writing data / message** to the pipe.

Pipe (Unnamed Pipe or Anonymous Pipe)

- In linux, if the pipe **has no name** then it is called as **unnamed pipe or anonymous pipe** (gives communication between parent and child processes)
- It is a communication medium between two or more related or interrelated processes
- It is mainly used for interprocess communication. It has two ends. They are:
 1. First end is **fd[0] which is used for reading mode**
 2. Second end is **fd[1] which is used for writing mode**
- It is used for transferring data between two processes / commands / programs
- It acts like **queue data structure**. We can write 512 bytes at the same time but we can read **ONLY ONE BYTE** at the same time.
- It is an important to note that, pipe is an **uni-directional (half duplex or one-way communication)** that is **either from left to right or right to left**
- It is an important to note that, **whatever is written on one end** (Ex. write end) **might visible on other end** (Ex. read end)

Return Value of Pipe

- It returns **0** if successful otherwise it returns **-1**
- Return type: **int**

Required Header File

```
#include<unistd.h>
```

OPERATIONS OF PIPE

- One Way Communication between processes
 - One pipe system call is required
- Two Way Communication between processes
 - Two pipe system calls are required
 - By default, it is a half-duplex method. So the first process will communicate with second process or second process will communicate with first process. However, in order to achieve a full-duplex, another pipe is needed.

BUILT-IN METHODS OF PIPE

1. write(pipe-descriptor[1], void * message, size_t count)

- This method is used to write the string message using the pipe end [1]
- It takes three arguments such as descriptor, message and size
- 1st argument is [pipe descriptor](#)
- 2nd argument is message which is [string type](#)
- 3rd argument is count which is [unsigned int type](#).
- This method will [return the number of bytes written on success](#) case and will return [-1 if the case is failure](#).
- Return type: [ssize_t](#)

2. read(pipe-descriptor[0], void * buffer, size_t count)

- This method is used to read the string message using the pipe end [0]
- It takes three arguments such as descriptor, message and size
- 1st argument is [pipe descriptor](#)
- 2nd argument is message which is [string type](#)

- 3rd argument is count which is `unsigned int` type.
- This method will `return the number of bytes read on success` case and will return `-1 if the case is failure`.
- Return type: `ssize_t`

3. `close(pipe-descriptor)`

- This method is used to close the pipe if pipe is already opened
- It takes only one argument which is the integer value of pipe descriptor
- It will return `0 on success case` and `-1 on failure case`.
- Return type: `int`

I. SENDING AND RECEIVING STATIC MESSAGES BETWEEN TWO PROCESSES USING PIPE

(/home/runner/Latest-Shells-21/pipeex1.c)

SOURCE CODE

```
#include<string.h>
#include<fcntl.h>
#include <unistd.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int r;
// pipe descriptors
    int p[2];
    printf("-----\n");
    printf("\tSending and Receiving Static Messages-Pipe\n");
    printf("-----\n");
// Text Messages (fixed messages)
    char *sms1="Good Morning\n";
    char *sms2="Hello World\n";
    unsigned int s=strlen(sms1);
    char buf[1024];
// create unnamed pipe using pipe system call
    r=pipe(p);
    if(r<0)
    {
        printf("Failed to created unnamed pipe...\n");
        exit(1);
    }
// send messages to another process
    write(p[1],sms1,strlen(sms1));
    write(p[1],sms2,strlen(sms1));
    printf("Two Messages are sent successfully...(Process 1)\n");
```

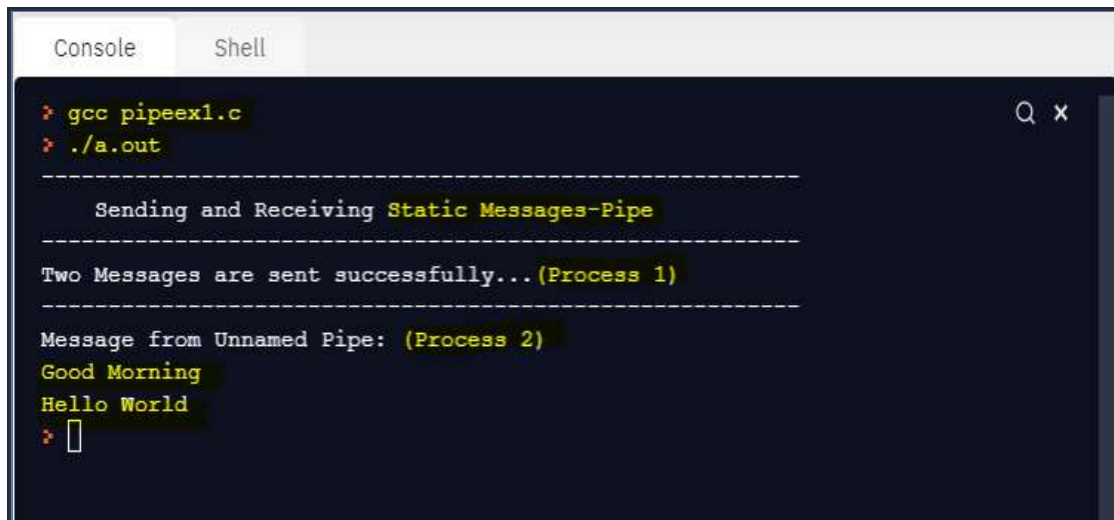


```

printf("-----\n");
printf("Message from Unnamed Pipe: (Process 2)\n");
// read messages from the pipe (other end) using single buffer
read(p[0],buf,sizeof(buf));
printf("%s",buf);
return 99;
}

```

OUTPUT



```

Console Shell
> gcc pipeex1.c
> ./a.out
-----
Sending and Receiving Static Messages-Pipe
-----
Two Messages are sent successfully... (Process 1)
-----
Message from Unnamed Pipe: (Process 2)
Good Morning
Hello World
> 

```

II. FINDING PRIME NUMBER USING PIPE (Static Input)

(/home/runner/Latest-Shells-21/pipeex3.c)

SOURCE CODE

```

#include<fcntl.h>
#include <unistd.h>
#include<stdio.h>
#include<stdlib.h>
// function returns string as a result
char * findprime(int n)
{
    int c=0;
    for(int i=0;i<n;i++)
    {

```

```

        if(n%(i+1)==0)
            c++;
    }
    if(c==2)
        return "The Given Number is a prime...\n";
    else
        return "The Given Number is NOT a prime...\n";
}
// main function
int main()
{
    int r,ele;
// pipe descriptors
    int p[2];
    printf("-----\n");
    printf("\tFinding Prime Number-Pipe\n");
    printf("-----\n");
    char buf[1024];
// create unnamed pipe using pipe system call
    r=pipe(p);
    if(r<0)
    {
        printf("Failed to created unnamed pipe...\n");
        exit(1);
    }
    char *msg="18";
// send single message (string type) to another process (one end)
    write(p[1],msg,sizeof(msg));
    printf("One Message is sent successfully...(Process 1)\n");
    printf("-----\n");
    printf("Message from Unnamed Pipe: (Process 2)\n");
// read message from pipe (other end)
    read(p[0],buf,sizeof(msg));

```

```

    printf("Received Data: %s\n",buf);
// convert received string data to integer
    int num=atoi(buf);
// pass integer number to function for the prime number detection
    char *rs=findprime(num);
// print the result
    printf("%s\n",rs);
    return 99;
}

```

IF INPUT IS 18

```

Console Shell
> gcc pipeex3.c
> ./a.out
-----
    Finding Prime Number-Pipe
-----
One Message is sent successfully... (Process 1)
-----
Message from Unnamed Pipe: (Process 2)
Received Data: 18
The Given Number is NOT a prime...
> █

```

IF INPUT IS 31

```

Console Shell
> gcc pipeex3.c
> ./a.out
-----
    Finding Prime Number-Pipe
-----
One Message is sent successfully... (Process 1)
-----
Message from Unnamed Pipe: (Process 2)
Received Data: 31
The Given Number is a prime...
> █

```

III. FINDING PRIME NUMBER USING PIPE

(Dynamic Input)

(/home/runner/Latest-Shells-21/pipeex4.c)

SOURCE CODE

```
#include<fcntl.h>
#include <unistd.h>
#include<stdio.h>
#include<stdlib.h>
char * findprime(int n)
{
    int c=0;
    for(int i=0;i<n;i++)
    {
        if(n%(i+1)==0)
        {
            c++;
        }
    }
    if(c==2)
        return "The Given Number is a prime...\n";
    else
        return "The Given Number is NOT a prime...\n";
}
int main()
{
    int r;
    // array of pipe descriptors for reading and writing
    int p[2];
    printf("-----\n");
    printf("\tFinding Prime Number-Pipe\n");
    printf("-----\n");
    char buf[1024];
```

```

// create unnamed pipe using pipe system call
r=pipe(p);
if(r<0)
{
    printf("Failed to created unnamed pipe...\n");
    exit(1);
}
int ele;
printf("Enter a number: ");
// read a dynamic input which is an integer type
scanf("%d",&ele);
// create a string and allocate a dynamic memory for this type
char *msg=(char *)malloc(sizeof(char *));
// convert integer number to string using sprintf(char*, type, base) method
sprintf(msg,"%d",ele);
// send single message to another process (one end)
write(p[1],msg,sizeof(msg));
printf("One Message is sent successfully...(Process 1)\n");
printf("-----\n");
printf("Message from Unnamed Pipe: (Process 2)\n");
// read message from sender using read() method (other end)
read(p[0],buf,sizeof(msg));
printf("Received Data: %s\n",buf);
// convert received string data to integer
int num=atoi(buf);
// pass integer number to function for the prime number detection
char *rs=findprime(num);
// display the results
printf("%s\n",rs);
return 99;
}

```

OUTPUT

```
Console Shell
> gcc pipeex4.c
> ./a.out

-----
Finding Prime Number-Pipe
-----
Enter a number: 13
One Message is sent successfully...(Process 1)
-----
Message from Unnamed Pipe: (Process 2)
Received Data: 13
The Given Number is a prime...

> gcc pipeex4.c
> ./a.out

-----
Finding Prime Number-Pipe
-----
Enter a number: 15
One Message is sent successfully...(Process 1)
-----
Message from Unnamed Pipe: (Process 2)
Received Data: 15
The Given Number is NOT a prime...

> gcc pipeex4.c
> ./a.out

-----
Finding Prime Number-Pipe
-----
Enter a number: 50
One Message is sent successfully...(Process 1)
-----
Message from Unnamed Pipe: (Process 2)
Received Data: 50
The Given Number is NOT a prime...

> |
```

IV. SENDING AND RECEIVING DYNAMIC MESSAGES BETWEEN TWO PROCESSES USING PIPE

(/home/runner/Latest-Shells-21/test2.c)

SOURCE CODE

```
#include<string.h>
#include<fcntl.h>
#include <unistd.h>
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int r,nl;
    static int c=0;
    // dynamic memory for sender
    char *msg=(char *)malloc(sizeof(char *));
    // fixed memory for receiver memory (max.memory)
    char bf[1024];
    // pipe descriptors for reading and writing data
    int p[2];
    printf("-----\n");
    printf("\tSending & Receiving Dynamic Messages using Pipe\n");
    printf("-----\n");
    // create unnamed pipe using pipe system call
    r=pipe(p);
    if(r<0)
    {
        printf("Failed to created unnamed pipe...\n");
        exit(1);
    }
    printf("Enter the number of messages to send: ");
    scanf("%d",&nl);
    printf("Enter the messages:\n");
    for(int i=0;i<nl;i++)
```

```

{
    printf("Message #%d: \n", (i+1));
    scanf("%s", msg);
    // write message to process 2
    write(p[1], msg, 50);
    // count the length of the message
    s += strlen(msg);
    printf("\tOne SMS is sent\n");
    printf("\tSMS Length: %d\n", (int)strlen(msg));
    // increment the SMS count by the variable c with 1
    c++;
}
printf("%d Messages are Sent Successfully\n", c);
printf("Message from Unnamed Pipe:\n");
printf("Process 2:\n");
printf("-----\n");
for(int i=0; i<n1; i++)
{
    read(p[0], bf, 50);
    printf("%s\n", bf);
}
return 99;
}

```


OUTPUT

```
Console Shell
> gcc test2.c
> ./a.out
-----
      Sending & Receiving Dynamic Messages using Pipe
-----
Enter the number of messages to send: 5
Enter the messages:
Message #1:
HelloWorld
    One SMS is sent
    SMS Length: 10
Message #2:
GoodMorning
    One SMS is sent
    SMS Length: 11
Message #3:
Welcome
    One SMS is sent
    SMS Length: 7
Message #4:
Super
    One SMS is sent
    SMS Length: 5
Message #5:
Nice
    One SMS is sent
    SMS Length: 4
5 Messages are Sent Successfully
Message from Unnamed Pipe:
Process 2:
-----
HelloWorld
GoodMorning
Welcome
Super
Nice
>
```

V. PARENT AND CHILD PROCESS COMMUNICATION (IPC) USING PIPE

[\(/home/runner/Latest-Shells-21/ipcpipe.c\)](#)

Number of used Pipes	:	1
Number of child processes	:	1
Number of parent processes	:	1
Major system calls used	:	pipe(), fork()
Types of Pipes	:	Unnamed Pipe()
Type of Communication	:	One way

SOURCE CODE

```
#include<fcntl.h>
#include <unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
int main()
{
    int r;
    // pipe descriptors for read and write
    int p[2];
    printf("-----\n");
    printf("\tIPC-Parent and Child using Pipe (One Way)\n");
    printf("-----\n");
    // variables declarations
    char *sms=(char *)malloc(sizeof(char *));
    char buf[1024];
    // create unnamed pipe using pipe system call
    r=pipe(p);
    if(r<0)
    {
        printf("Failed to created unnamed pipe...\n");
        exit(1);
    }
    // call fork() to create parent and child processes
    int f=fork();
    if(f<0)
    {
        printf("Error in creating the processes...\n");
        exit(0);
    }
}
```

```

// child process
else if(f==0)
{
    printf("Child Process (P1)\n");
    printf("Enter the message to send (50 characters maximum): ");
    fgets(sms,50,stdin);
    // send message from child to parent using pipe
    write(p[1],sms,50);
}
// parent process
else
{
    wait(NULL);
    printf("-----\n");
    printf("Parent Process (P2)\n");
    // receive message from child via pipe
    read(p[0],buf,50);
    printf("Received Message:\n");
    printf("%s\n",buf);
}
return 99;
}

```

It is indicated by keyboard input.

OUTPUT

```

Console  Shell
> gcc ipcpipe.c
> ./a.out

-----
IPC-Parent and Child using Pipe
-----

Child Process (P1)
Enter the message to send (50 characters maximum): Hello Hi How are you?
-----

Parent Process (P2)
Received Message:
Hello Hi How are you?

>

```

RESULT

Thus the interprocess communication using pipe has been executed successfully.

EX.NO: 8 INTERPROCESS COMMUNICATION USING NAMED PIPE

AIM

To work with interprocess communication using named pipe in linux c programming.

IPC USING NAMED PIPE (FIFO)

- FIFO stands for **First in First Out**
- A special kind of the file on the local storage that allows two or more number of processes to communicate with each other
- Named IPC object which provides communication between two unrelated processes
- Unlike pipe, **it has name (identified by the unique name)**
- Unlike pipe, **it is a full duplex method which means that first process can communicate with second process and second process can communicate with first process.**

DIFFERENCE BETWEEN PIPE AND FIFO

S.N	FEATURES	PIPE (Unnamed Pipe)	FIFO (Named Pipe)
1.	Description	It has no name (Unnamed IPC object)	It has a name (Named IPC object)
2.	Creation	It is created by the system call pipe()	It is created by the method mkfifo() or open()
3.	Existence	It does not exist in the file systems	It exists in the file system
4.	Nature	By default, it is an unidirectional	It is bidirectional , same FIFO can be used for reading and writing at the same time
5.	Read & Write	Here reader and writer operations are done at the same time	Here, it does not require that both read and write operations to happen at the same time.

6.	Processes	Here data transfer takes place between parent and child process.	It has multiple processes communicating through it, like multiple client server application
7.	Communication	Here communication is among the process having related process	In FIFO, it is not necessary for the process having the related process (unrelated process)
8.	Support	Pipe is local to the system and can't be used for the communication across the network	It is capable of communicating across different computers and network.

NOTE

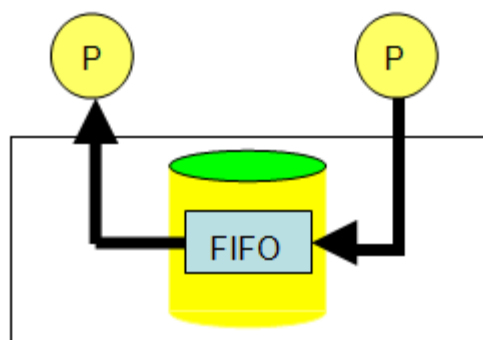
- Neither pipes nor FIFO allow file positioning. Both reading and writing operations happen sequentially that is reading from the beginning of the file and writing at the end of the file.

Required Header File

```
#include<sys/stat.h> // used for creating FIFO i.e. mkfifo()
```

CAPACITY OF FIFO

- It has [multiple readers or multiple writers](#)
- Bytes from each writer are written automatically up to a maximum size of PIPE_BUF (4KB on Linux OS)



BUILT-IN METHODS OF NAMED PIPE (FIFO)

1. mkfifo(const char *pathname, mode_t mode)

- It is used to create a FIFO file
- It takes two arguments. They are
 - First argument is pathname of the newly created FIFO file which is string type (identified by char *)
 - Second argument is mode of the FIFO's permissions.
- Return type: [int](#)

2. open(fpath, mode, permission)

- It is used to create a named pipe (FIFO)
- It takes two arguments. First argument is file name and second argument is mode. The mode can be
 - O_CREAT (create mode)
 - O_WRONLY (write mode only)
 - O_RDONLY (read mode only)
 - O_RDWR (read and write support)
- Return type: [int](#)

3. write(int file descriptor, void * message, size_t size)

- It is used to write / send the data from one end to another end
- It takes three arguments. They are
 1. First argument is the file object
 2. Second argument is data / message
 3. Third argument is the size of the data
- Return type: [ssize_t](#)

4. read(int file descriptor, void * message, size_t size)

- It is used to read / receive the data from named pipe
- It takes three arguments. They are
 1. First argument is the file object
 2. Second argument is data / message
 3. Third argument is the size of the data

- Return type: `ssize_t`

5. close(file descriptor)

- It is used to close the file descriptor if it is already opened
- It takes only one argument which is the file descriptor
- Return type: `int`

I. EXAMPLE OF IPC USING FIFO (NAMED PIPE)

(/home/runner/Latest-Shells-21/fifo1.c)

System Calls	:	File system calls
Type of Pipe	:	FIFO (Named Pipe)
Type of Communication	:	One way

SOURCE CODE

```
#include <stdio.h>
#include<sys/stat.h>
#include<string.h>
#include<fcntl.h>
#include <unistd.h>
// global variables
char m1[]="Hello World.";
char m2[]="Good Morning.";
char m3[]="Welcome to Chennai.\n";
void sendMessage()
{
    int fd;
    // create new file (FIFO) for create and write mode with necessary permission
    fd=open("sms.txt",O_CREAT| O_WRONLY,0777);
    // write the three messages to named pipe
    write(fd,m1,strlen(m1));
```

```

write(fd,m2,strlen(m2));
write(fd,m3,strlen(m3));
printf("Three Messages are successfully sent to named pipe...\n");
// close the file descriptor
close(fd);
}
void receiveMessage()
{
    int fp;
    char buf1[100],buf2[100],buf3[100];
    // open same file (FIFO) for read mode
    fp=open("sms.txt",O_RDONLY);
    // read three messages from named pipe
    read(fp,buf1,strlen(m1));
    read(fp,buf2,strlen(m2));
    read(fp,buf3,strlen(m3));
    printf("Messages from FIFO:\n");
    printf("-----\n");
    printf("\t%s\n",buf1);
    printf("\t%s\n",buf2);
    printf("\t%s\n",buf3);
    // close the file descriptor
    close(fp);
}

```

It is better to use length of sender's message in receiver side

```

int main()
{
    printf("-----\n");
    printf("\t\tIPC using FIFO (Named Pipe)\n");
    printf("-----\n");
    sendMessage();
}

```

Same FIFO file can be used for reading mode.


```
receiveMessage();
return 99;
}
```

OUTPUT



```
Console Shell
> ls
a.out  arr4.sh  hello    ipcpipe.c  pip1.c  pipeex3.c  test2.c
arr1.sh  fifol.c  hello.c  main.sh   pipel.c  pipeex4.c  tp
arr2.sh  forkinshell1.sh  hello.txt  msg.txt  pipeex1.c  sc        wel.c
arr3.sh  hell    ipcpipe1.c  npc.txt  pipeex2.c  sc1.c     wel.txt
> gcc fifol.c
> ./a.out
-----
IPC using FIFO (Named Pipe)
-----
Three Messages are successfully sent to named pipe...
Messages from FIFO:
-----
Hello World.
Good Morning.
Welcome to Chennai.
> ls
a.out  fifol.c  hello.txt  npc.txt  pipeex3.c  test2.c
arr1.sh  forkinshell1.sh  ipcpipe1.c  pip1.c  pipeex4.c  tp
arr2.sh  hell    ipcpipe.c  pipel.c  sc        wel.c
arr3.sh  hello  main.sh   pipeex1.c  sc1.c     wel.txt
arr4.sh  hello.c  msg.txt  pipeex2.c  sms.txt
> cat sms.txt
Hello World.Good Morning.Welcome to Chennai.
> ls -l sms.txt
-rwxr-xr-x 1 runner runner 45 May 26 05:42 sms.txt
>
```

RESULT

Thus the interprocess communication using named pipe has been executed successfully.

EX.NO: 9**MULTITHREADING USING PYTHON****AIM**

To practice the multithreading programming using python language.

THREADING

- Thread is a path of the execution within a process.

TASK BASED TYPES

1. Single Threading
2. Multithreading

DIFFERENCE BETWEEN SINGLE THREAD AND MULTIPLE THREADS

S.N	SINGLE THREAD	MULTITHREADING
1.	Performs single task	Performs multitasking. Doing more than one job at the same time
2.	It consists of only one thread	It consists of several threads
3.	It is used for experimental purpose	It is used for larger tasks

LIFE CYCLE OF THREADS

- Thread has five life cycles. It is always live in any of the state.
 1. New born state (New state)
 2. Runnable state
 3. Running state (execution state)
 4. Blocked state
 5. Dead state (End state)

BUILT-IN METHODS**1. start()**

- This is runnable state (waiting for the execution)
- This is method is used to start the thread

- Return type : nothing

2. threading.currentThread().getName()

- This method is used to display the name of currently executing or running thread
- Return type : **String**

Thread Creation

- Thread is created by using the super class Thread

Thread(name=<user-defined-name, target=<function-name>,)

- It is a built-in class which is used to create a new thread object
- It takes two or more arguments
- First argument is **name of thread**. It can be any name set by the user. It is optional argument
- Second argument is **target** which is used **to call the user defined function**

Required Module

threading

I. EXAMPLE OF SINGLE THREAD

Tools used : VSC Editor
Platform OS : Windows 10
Language : Python 3

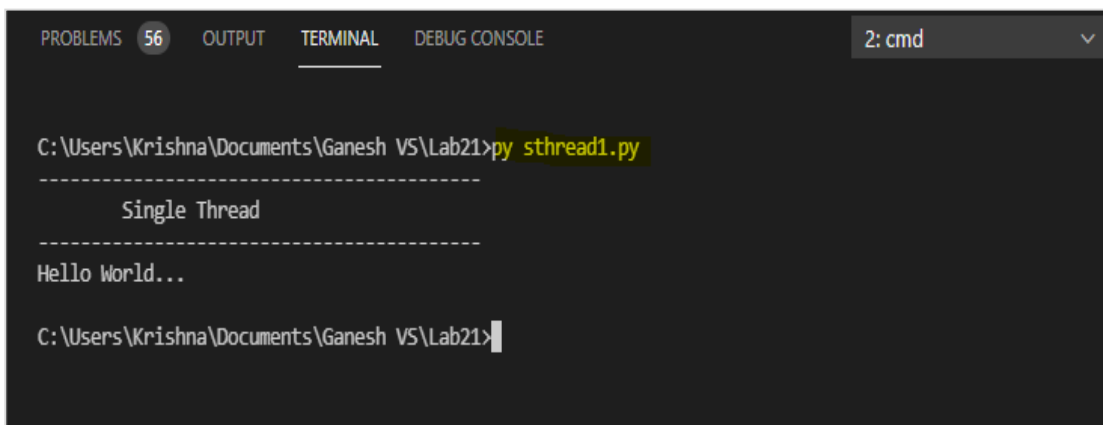
SOURCE CODE

```
from threading import *  
# user defined function  
def welcome():  
    print("Hello World...")  
print("-----")  
print("\tSingle Thread")  
print("-----")  
# creating object for single thread  
t1=Thread(target=welcome, name="Thread 1")  
# start the thread by calling start method  
t1.start()
```

Newborn state

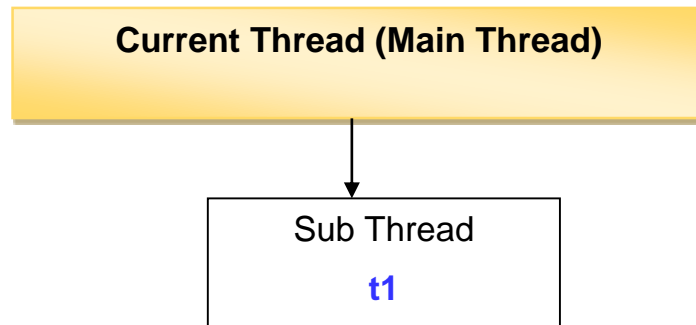
Runnable state

OUTPUT



```
PROBLEMS 56 OUTPUT TERMINAL DEBUG CONSOLE 2: cmd  
C:\Users\Krishna\Documents\Ganesh VS\Lab21>py sthread1.py  
-----  
Single Thread  
-----  
Hello World...  
C:\Users\Krishna\Documents\Ganesh VS\Lab21>
```

Pictorial Diagram



MULTITHREADING

- Process of executing more than one thread at the same time (Execution of multiple threads simultaneously)
- It is an important to note that, **we can't predict which thread will run first.**
- By all the threads will be running in parallel at the same time.
- It is an important to note that, **the main or current thread can randomly start any thread** from the list of threads.

II. EXAMPLE OF MULTITHREADING

(Asynchronous Processes)

Tools used : VSC Editor
Platform OS : Windows 10
Language : **Python 3**

SOURCE CODE

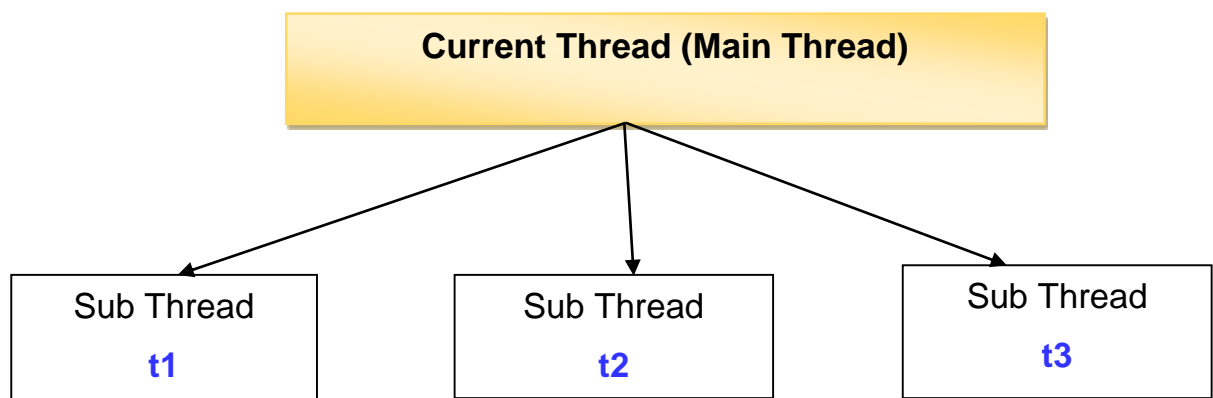
```
from threading import *  
# user defined function 1  
def m1():  
    for i in range(3):  
        print("Good Morning...")  
# user defined function 2  
def m2():
```

```

for i in range(3):
    print("Good Evening...")
# user defined function 3
def m3():
    for i in range(3):
        print("Good Night...")
print("-----")
print("\tMultithreading")
print("-----")
# creating objects for multiple threads
t1=Thread(target=m1,name="Morning")
t2=Thread(target=m2,name="Evening")
t3=Thread(target=m3,name="Night")
# start the threads by calling start method
t1.start()
t2.start()
t3.start()

```

Pictorial Diagram



OUTPUT

```
PROBLEMS 87 OUTPUT TERMINAL DEBUG CONSOLE 2: cmd
C:\Users\Krishna\Documents\Ganesh VS\Lab21>py mthread1.py
-----
Multithreading
-----
Good Morning...
Good Morning...
Good Morning...
Good Evening...
Good Evening...
Good Night...
Good Night...
Good Night...
Good Evening...

C:\Users\Krishna\Documents\Ganesh VS\Lab21>py mthread1.py
-----
Multithreading
-----
Good Morning...
Good Morning...
Good Evening...
Good Morning...
Good Evening...
Good Evening...
Good Night...
Good Night...
Good Night...

C:\Users\Krishna\Documents\Ganesh VS\Lab21>
```

III. EXAMPLE OF MULTITHREADING

(Synchronous Processes using join() method)

Tools used	:	VSC Editor
Platform OS	:	Windows 10
Language	:	Python 3

Existing Issues

- By default, the threads are running parallel in multithreading
- In the execution of multithreading using asynchronous methods, **the threads will be running in parallel**. That's why the output came differently in the previous example.
- So in order to execute thread one by one (sequential order) during the multithreading, **the join method will be used**.

SOURCE CODE

```
from threading import *
# user defined function 1
def m1():
    for i in range(3):
        print("Good Morning...")
# user defined function 2
def m2():
    for i in range(3):
        print("Good Evening...")

# user defined function 3
def m3():
    for i in range(3):
        print("Good Night...")
print("-----")
print("\tMultithreading")
print("-----")
```



```
# creating objects for multiple threads
t1=Thread(target=m1,name="Morning")
t2=Thread(target=m2,name="Evening")
t3=Thread(target=m3,name="Night")
# start thread 1
t1.start()
# wait until thread 1 is finished (main and sub threads t2, t3 should wait)
t1.join()
# start thread 2 after thread 1
t2.start()
# wait until thread 2 is finished (main and sub threads t1, t3 should wait)
t2.join()
# start thread 3 after thread 2
t3.start()
# wait until thread 3 is finished (main and sub threads t1, t2 should wait)
t3.join()
# end of the main thread
print("End of the main thread...")
```

OUTPUT

```
PROBLEMS 118 OUTPUT TERMINAL DEBUG CONSOLE 2: cmd +
C:\Users\Krishna\Documents\Ganesh VS\Lab21>py mthread2.py
-----
Multithreading
-----
Good Morning..
Good Morning.. ] T1
Good Morning.. ]
Good Evening.. ] T2
Good Evening.. ]
Good Night... ] T3
Good Night... ]
Good Night... ]
End of the main thread...

C:\Users\Krishna\Documents\Ganesh VS\Lab21>py mthread2.py
-----
Multithreading
-----
Good Morning..
Good Morning..
Good Morning..
Good Evening..
Good Evening..
Good Evening..
Good Night...
Good Night...
Good Night...
End of the main thread...

C:\Users\Krishna\Documents\Ganesh VS\Lab21>py mthread2.py
-----
Multithreading
-----
Good Morning..
Good Morning..
Good Morning..
Good Evening..
Good Evening..
Good Evening..
Good Night...
Good Night...
Good Night...
End of the main thread...

C:\Users\Krishna\Documents\Ganesh VS\Lab21>
```

IV. DETECTION OF CURRENTLY EXECUTING THREAD

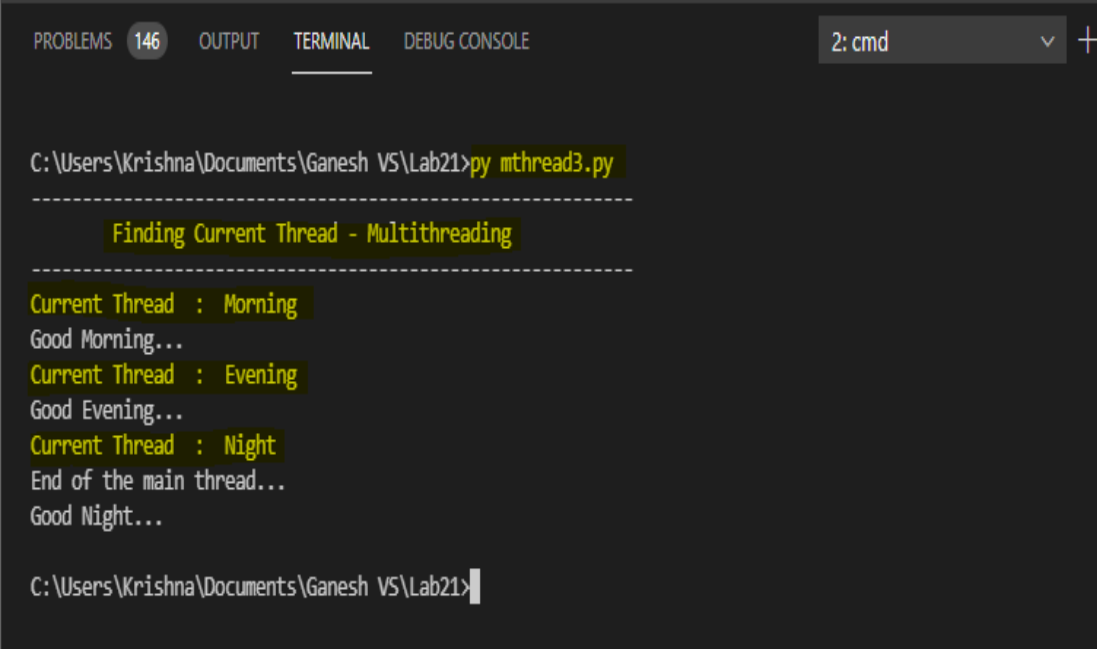
Tools used : VSC Editor
Platform OS : Windows 10
Language : **Python 3**

SOURCE CODE

```
from threading import *
import threading
# user defined function 1
def m1():
    tname=threading.currentThread().getName()
    print("Current Thread\t: ",tname)
    print("Good Morning...")
# user defined function 2
def m2():
    tname=threading.currentThread().getName()
    print("Current Thread\t: ",tname)
    print("Good Evening...")
# user defined function 3
def m3():
    tname=threading.currentThread().getName()
    print("Current Thread\t: ",tname)
    print("Good Night...")
# main thread
print("-----")
print("\tFinding Current Thread - Multithreading")
print("-----")
# creating objects for multiple threads
t1=Thread(target=m1,name="Morning")
t2=Thread(target=m2,name="Evening")
t3=Thread(target=m3,name="Night")
```

```
# start threads
t1.start()
t2.start()
t3.start()
# end of the main thread
print("End of the main thread...")
```

OUTPUT



```
PROBLEMS 146 OUTPUT TERMINAL DEBUG CONSOLE 2: cmd
C:\Users\Krishna\Documents\Ganesh VS\Lab21>py mthread3.py
-----
      Finding Current Thread - Multithreading
-----
Current Thread : Morning
Good Morning...
Current Thread : Evening
Good Evening...
Current Thread : Night
End of the main thread...
Good Night...

C:\Users\Krishna\Documents\Ganesh VS\Lab21>
```

GENERAL TYPES OF THREAD

- Like java, python supports two types of threads. They are
 1. Daemon thread
 2. Non daemon thread (User Thread)

1. Daemon Thread

- If a thread is running in **background mode**, then it is called as daemon thread
- It has low priority level than user thread
- This is created by **adding the boolean value to the daemon** argument of the Thread class.

2. Non Daemon Thread

- If a thread is running in **foreground mode**, then it is called as non daemon thread (user thread)
- It has high priority level than daemon thread
- This is created by the thread class.

DIFFERENCE BETWEEN DAEMON THREAD AND NON DAEMON THREAD

S.N	DAEMON THREAD	NON DAEMON THREAD
1.	It is always runs in background mode	It is always runs in foreground mode
2.	Main program does not wait for daemon thread to finish its task	Here main program waits for user threads have to terminate
3.	It has low priority	It has high priority
4.	It is not used for important task. It is generally used for some background tasks which are not important	Any important task is done by the user thread.
5.	It is created by the Python Virtual Machine (PVM)	It is created by the application or program.
6.	If all the threads are finished their execution, the PVM will force the daemon threads to finish their execution.	PVM won't force the user threads for termination. So it waits for user threads to terminate themselves.

I. EXAMPLE OF DAEMON THREAD

Tools used : VSC Editor
Platform OS : Windows 10
Language : **Python 3**

SOURCE CODE

```
from threading import *
from time import sleep
# user defined function 1
def m1():
    for i in range(3):
        print("Good Morning...")
        sleep(2)
# user defined function 2
def m2():
    for i in range(3):
        print("Good Evening...")
# user defined function 3
def m3():
    for i in range(3):
        print("Good Night...")
print("-----")
print("\tDaemon Thread")
print("-----")
# creating objects for multiple threads
# convert thread t1 to daemon thread by setting the boolean true to the
# daemon argument of thread class
t1=Thread(target=m1,name="Morning", daemon=True)
# non daemon threads
t2=Thread(target=m2,name="Evening")
t3=Thread(target=m3,name="Night")
```

Convert user thread to non-daemon thread.

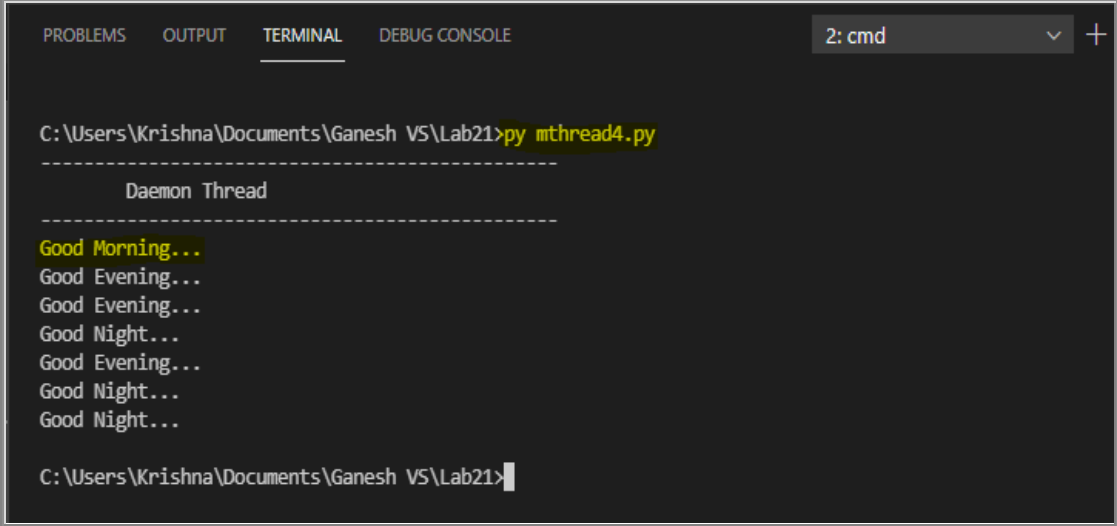
```
# start the thread by calling start method
```

```
t1.start()
```

```
t2.start()
```

```
t3.start()
```

OUTPUT



```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE 2: cmd +
C:\Users\Krishna\Documents\Ganesh VS\Lab21>py mthread4.py
-----
Daemon Thread
-----
Good Morning...
Good Evening...
Good Evening...
Good Night...
Good Evening...
Good Night...
Good Night...
C:\Users\Krishna\Documents\Ganesh VS\Lab21>
```

NOTE

- In the above output screenshot, the daemon thread **t1** → **Good Morning** is still running in the background mode.
- Eventhough all threads (including main thread) are terminated, the daemon thread is still aliving and running in the background.

II. EXAMPLE OF NON DAEMON THREAD

Tools used : VSC Editor
Platform OS : Windows 10
Language : **Python 3**

SOURCE CODE

```
from threading import *
from time import sleep
# user defined function 1
def m1():
    for i in range(3):
        print("Good Morning...")
        sleep(2)
# user defined function 2
def m2():
    for i in range(3):
        print("Good Evening...")
# user defined function 3
def m3():
    for i in range(3):
        print("Good Night...")
print("-----")
print("\tNon Daemon Thread(User Threads)")
print("-----")
# creating objects for multiple threads
# Non Daemon Threads
t1=Thread(target=m1,name="Morning")
t2=Thread(target=m2,name="Evening")
t3=Thread(target=m3,name="Night")
# start the threads by calling start method
t1.start()
```



```
t2.start()
t3.start()
```

OUTPUT

```
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE 2: cmd
C:\Users\Krishna\Documents\Ganesh VS\Lab21>py mthread4.py
-----
Daemon Thread
-----
Good Morning...
Good Evening...
Good Evening...
Good Night...
Good Evening...
Good Night...
Good Night...

C:\Users\Krishna\Documents\Ganesh VS\Lab21>py mthread5.py
-----
Non Daemon Thread(User Threads)
-----
Good Morning...
Good Evening...
Good Evening...
Good Evening...
Good Night...
Good Night...
Good Night...
Good Morning...
Good Morning...

C:\Users\Krishna\Documents\Ganesh VS\Lab21>
```

In the previous example, main thread and user threads are terminated. But still daemon thread t1 → **Good Morning** is running in background jobs.

Here, main thread and user threads are terminated. No threads are running in background.

RESULT

Thus the multithreading programming using python has been executed successfully.

AIM

To practice the file allocation strategy in linux programming.

FILE ALLOCATION METHODS

- It defines, how the files are stored in the disk blocks.
- It supports three methods. They are
 1. Sequential File Allocation (Contiguous Allocation)
 2. Indexed File Allocation
 3. Linked File Allocation

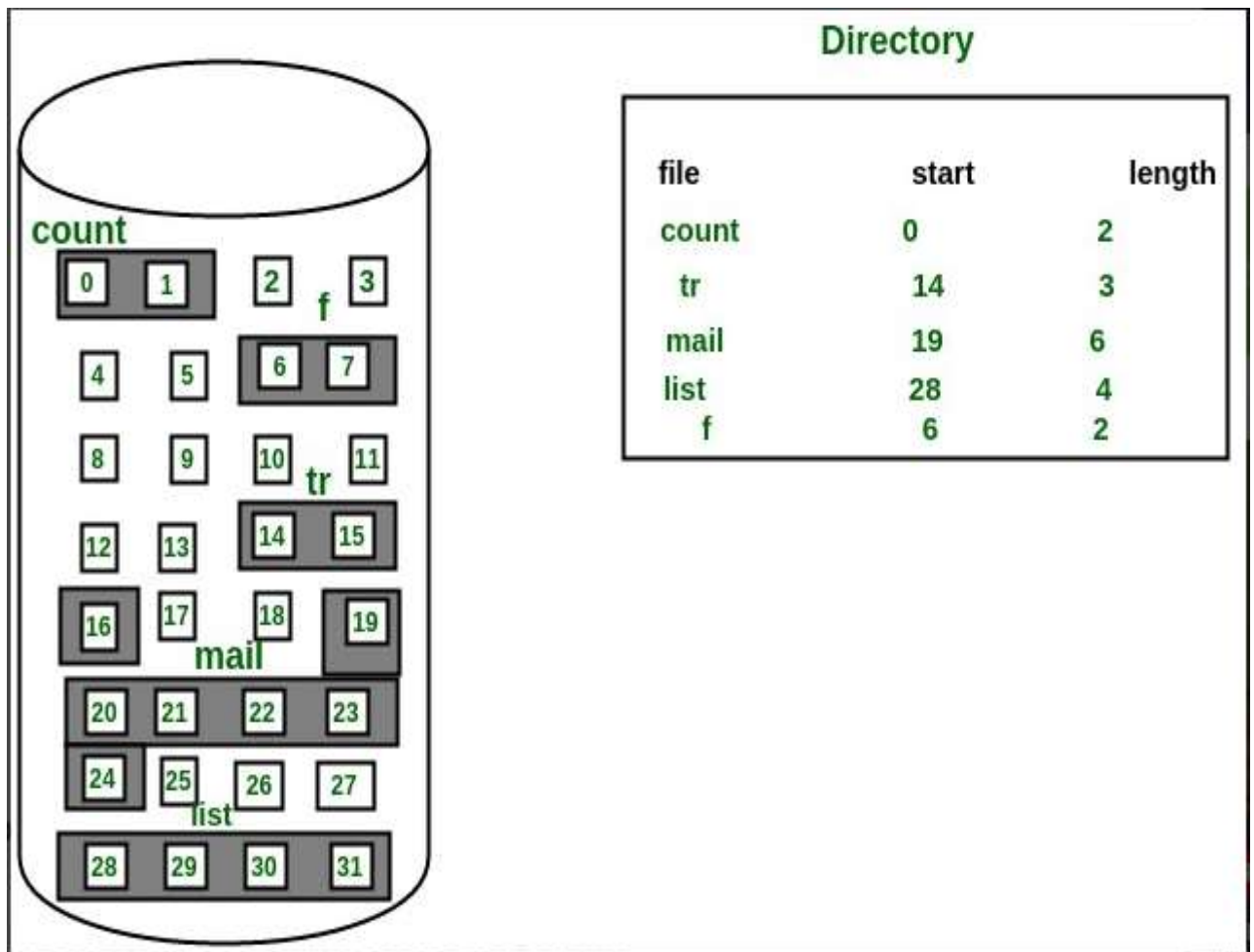
BENEFITS

- Efficient disk space utilization
- Fast access to the file blocks

1. SEQUENTIAL FILE ALLOCATION (CONTIGUOUS ALLOCATION)

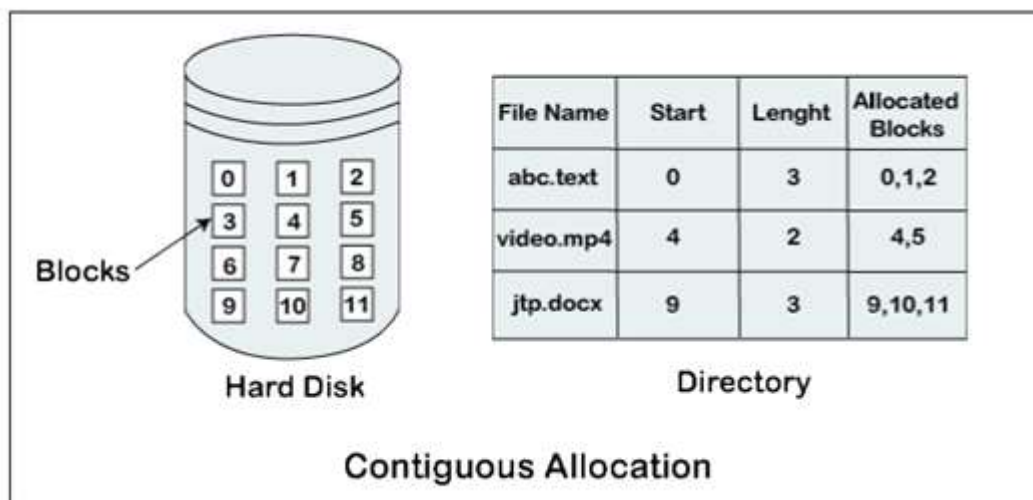
- Process of allocating resources to the contiguous blocks are called as sequential file allocation
- Both sequential and direct accesses are supported by this method.
- The directory entry for a file with contiguous allocation contains
 - Address of starting block
 - Length of the allocated portion.

Example 1



file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Example 2



File Name	Start	Lenght	Allocated Blocks
abc.text	0	3	0,1,2
video.mp4	4	2	4,5
jtp.docx	9	3	9,10,11

Contiguous Allocation

I. EXAMPLE OF SEQUENTIAL ALLOCATION METHOD

SOURCE CODE

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define BLOCKSIZE 50
int b[BLOCKSIZE+1];
int number,length;
void seq_fileallocation()
{
    printf("Enter the block number: ");
    scanf("%d",&number);
    // check for invalid number
    if(number>BLOCKSIZE)
    {
        printf("Invalid Block Number.\nPlease Enter the block number in the
range between 1 to 50\n");
    }
    else
    {
        // check the slot is free or not
        if(b[number]==0)
        {
            printf("Enter the number of lengths for the block %d: ", number);
            scanf("%d",&length);
            int c=0;
            // increment the counter for the blocks which are given by the user
            for(int i=number;i<number+length;i++)
            {
                if(b[i]==0)
                {
```

```

        c++;
    }
}
// if counter equals to exact length, then slot should be free
if(c==length)
{
// allocate file / block for the user request
    for(int i=number;i<number+length;i++)
    {
        b[i]=1;
        printf("block %d is allocated ...\n",i);
    }
    printf("File is allocated successfully for the block %d\n",number);
}
else if((number+length)>BLOCKSIZE)
{
    printf("File lengths are too out of range than MAX Capacity of Block
%d...\n",BLOCKSIZE);
}
else
{
    printf("File lengths are not free for the given block\n");
}
}
else
{
    printf("File / Block is already Allocated\nPlease try some block
%d\n",number);
}
}
}
// print the contents of table

```

```

void disp()
{
    for(int i=1;i<BLOCKSIZE+1;i++)
    {
        printf(" %d",b[i]);
        if(i%8==0)
            printf("\n");
    }
    printf("\n");
}

int main()
{
    // allot the inputs for the blocks
    for(int i=1;i<BLOCKSIZE+1;i++)
    {
        b[i]=0;
    }
    printf("Initial Block Table\n");
    disp();
    char ch[150];
    while(5)
    {
        printf("-----\n");
        printf("\tSequential File Allocation\n");
        printf("-----\n");
        printf("Before File Allocation, Table Contents:\n");
        disp();
        seq_fileallocation();
        printf("After File Allocation, Table Contents:\n");
        disp();
        printf("Do you want to continue: Press Yes / No : ");
    }
}

```

```

scanf("%s",ch);
if(strcmp(ch, "yes")==0||strcmp(ch, "Yes")==0||strcmp(ch, "YES")==0)
    continue;
else
    exit(1);
}
}

```

Allocation for Block Number 12 and its Lengths 5

```

Console Shell
> gcc seq.c
> ./a.out
Initial Block Table
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0
-----
Sequential File Allocation
-----
Before File Allocation, Table Contents:
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0
Enter the block number: 12
Enter the number of lengths for the block 12: 5
block 12 is allocated ...
block 13 is allocated ...
block 14 is allocated ...
block 15 is allocated ...
block 16 is allocated ...
File is allocated successfully for the block 12
After File Allocation, Table Contents:
0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0
Do you want to continue: Press Yes / No : Yes

```

Allocation for Block Number 3 and its Lengths 7 after that checking block number 9

```
Console Shell
-----
Sequential File Allocation
-----
Before File Allocation, Table Contents:
0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0
Enter the block number: 3
Enter the number of lengths for the block 3: 7
block 3 is allocated ...
block 4 is allocated ...
block 5 is allocated ...
block 6 is allocated ...
block 7 is allocated ...
block 8 is allocated ...
block 9 is allocated ...
File is allocated successfully for the block 3
After File Allocation, Table Contents:
0 0 1 1 1 1 1 1
1 0 0 1 1 1 1 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0
Do you want to continue: Press Yes / No : yes
-----
Sequential File Allocation
-----
Before File Allocation, Table Contents:
0 0 1 1 1 1 1 1
1 0 0 1 1 1 1 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0
Enter the block number: 9
File / Block is already Allocated
Please try some block 9
```


Allocation for Block Number 49 and its Lengths 5

```
Console Shell
Do you want to continue: Press Yes / No : yes
-----
Sequential File Allocation
-----
Before File Allocation, Table Contents:
0 0 1 1 1 1 1 1
1 0 0 1 1 1 1 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0
Enter the block number: 49
Enter the number of lengths for the block 49: 5
File lengths are too out of range than MAX Capacity of Block 50...
After File Allocation, Table Contents:
0 0 1 1 1 1 1 1
1 0 0 1 1 1 1 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0
Do you want to continue: Press Yes / No : yes
-----
Sequential File Allocation
-----
Before File Allocation, Table Contents:
0 0 1 1 1 1 1 1
1 0 0 1 1 1 1 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0
Enter the block number: 49
Enter the number of lengths for the block 49: 2
block 49 is allocated ...
block 50 is allocated ...
File is allocated successfully for the block 49
After File Allocation, Table Contents:
0 0 1 1 1 1 1 1
1 0 0 1 1 1 1 1
0 0 0 0 0 0 0 0
```

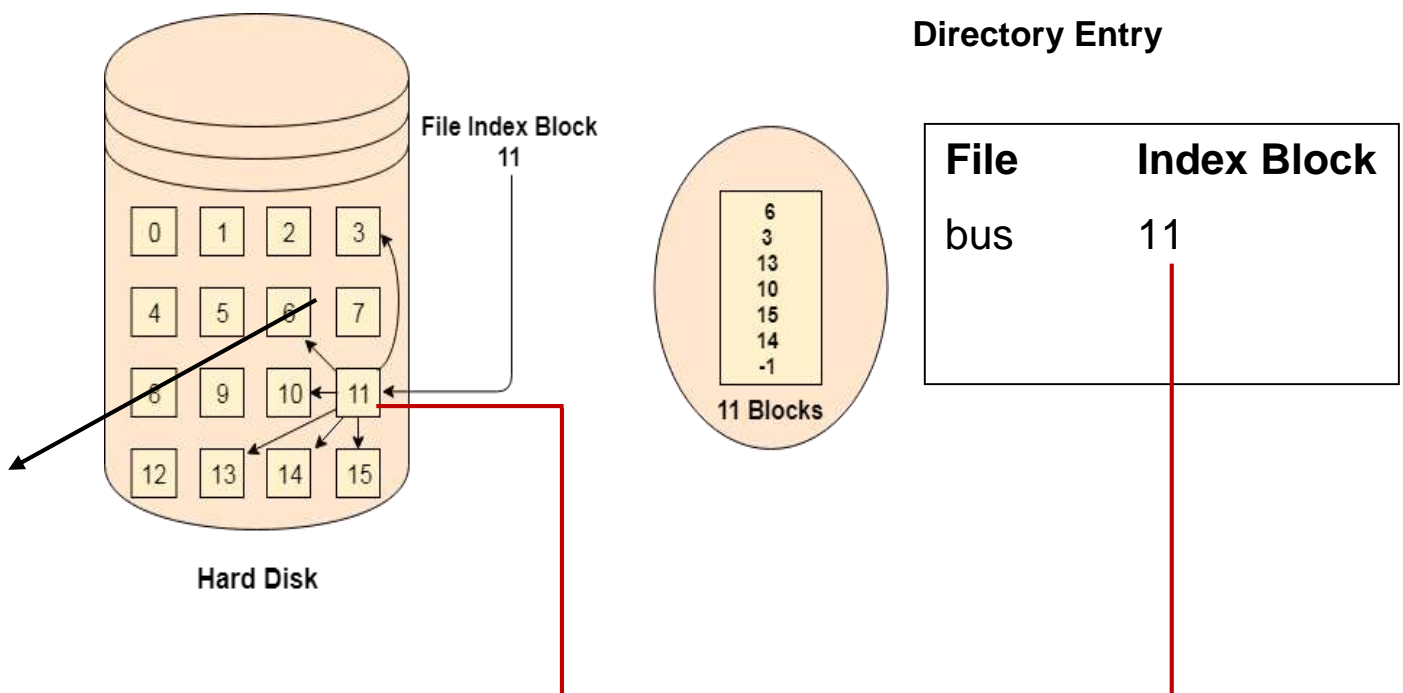
Allocation for Block Number 49 and its Lengths 5 (-Continue) after that checking block number 50

```
Console Shell
0 0 1 1 1 1 1 1
1 0 0 1 1 1 1 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0
Enter the block number: 49
Enter the number of lengths for the block 49: 2
block 49 is allocated ...
block 50 is allocated ...
File is allocated successfully for the block 49
After File Allocation, Table Contents:
0 0 1 1 1 1 1 1
1 0 0 1 1 1 1 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 1
Do you want to continue: Press Yes / No : yes
-----
Sequential File Allocation
-----
Before File Allocation, Table Contents:
0 0 1 1 1 1 1 1
1 0 0 1 1 1 1 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 1
Enter the block number: 50
File / Block is already Allocated
Please try some block 50
After File Allocation, Table Contents:
0 0 1 1 1 1 1 1
1 0 0 1 1 1 1 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 1
Do you want to continue: Press Yes / No : 
```

2. INDEXED FILE ALLOCATION

- Unlike sequential file allocation, this method uses an additional block called as the **index block** which is used to store all the disk pointers
- For each file, there is an individual index block.
- In the index block, the *i*th entry holds the disk address of the *i*th file block.
- It is an important to note that, **indexed block does not hold the file data**, but **it holds the pointers to all the disk blocks allotted to the particular file**.
- The directory entry contains the address of the index block as shown in the below image:

PICTORIAL REPRESENTATION



II. EXAMPLE OF INDEXED ALLOCATION METHOD

SOURCE CODE

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define BLOCKSIZE 50
int b[BLOCKSIZE+1], ib[BLOCKSIZE+1];
int number,length;
void indexed_allocation()
{
    printf("Enter the index block number: ");
    scanf("%d",&number);
    // check for invalid number
    if(number>BLOCKSIZE)
    {
        printf("Invalid index Block Number.\nPlease Enter the index block number
in the range between 1 to 50\n");
    }
    else
    {
        // check the slot is free or not
        if(b[number]==0)
        {
            printf("Enter the number of files for the block %d: ", number);
            scanf("%d",&length);
            printf("Enter the blocks:\n");
            for(int i=0;i<length;i++)
            {
                printf("Files #%d: ",(i+1));
                scanf("%d", &ib[i]);
            }
        }
    }
}
```

```

    int c=0;
// increment the counter for the blocks which are given by the user
    for(int i=0;i<length;i++)
    {
        if(b[ib[i]]>BLOCKSIZE)
        {
            printf("Index Block is Out of Range.\nPlease enter the blocks in
the range between 1-50\n");
            break;
        }
        else if(b[ib[i]]==0)
        {
            c++;
        }
    }
// if counter equals to exact length, then slot should be free
    if(c==length)
    {
// allocate file / block for the user request
        for(int i=0;i<length;i++)
        {
            b[ib[i]]=1;
            printf("Index block %d is allocated ...\n",ib[i]);
        }
        printf("File is allocated successfully for the block %d\n",number);
    }
// if the submitted index block is greater than MAXIMUM give message
    else
    {
        printf("File lengths are not free for the given block\n");
    }
}

```

```

    }
    else
    {
        printf("File / Block is already Allocated\nPlease try some block");
    }
}
}
// print the contents of table
void disp()
{
    for(int i=1;i<BLOCKSIZE+1;i++)
    {
        printf(" %d",b[i]);
        if(i%8==0)
            printf("\n");
    }
    printf("\n");
}
int main()
{
    // allot the inputs for the blocks
    for(int i=1;i<BLOCKSIZE+1;i++)
    {
        b[i]=0;
    }
    printf("Initial Block Table\n");
    disp();
    char ch[150];
    while(5)
    {
        printf("-----\n");
    }
}

```

```
printf("\tIndexed File Allocation\n");
printf("-----\n");
printf("Before File Allocation, Table Contents:\n");
disp();
indexed_allocation();
printf("After File Allocation, Table Contents:\n");
disp();
printf("Do you want to continue: Press Yes / No : ");
scanf("%s",ch);
if(strcmp(ch, "yes")==0||strcmp(ch, "Yes")==0||strcmp(ch, "YES")==0)
    continue;
else
    exit(1);
}
}
```

ALLOCATION OF INDEX BLOCK 5

```
Console Shell
> gcc ind.c
> ./a.out
Initial Block Table
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0
-----
Indexed File Allocation
-----
Before File Allocation, Table Contents:
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0
Enter the index block number: 5
Enter the number of files for the block 5: 7
Enter the blocks:
Files #1: 3
Files #2: 21
Files #3: 41
Files #4: 50
Files #5: 27
Files #6: 14
Files #7: 32
Index block 3 is allocated ...
Index block 21 is allocated ...
Index block 41 is allocated ...
Index block 50 is allocated ...
Index block 27 is allocated ...
Index block 14 is allocated ...
Index block 32 is allocated ...
File is allocated successfully for the block 5
```


ALLOCATION OF INDEX BLOCK 5 – (Continue)

```
Console Shell
Enter the blocks:
Files #1: 3
Files #2: 21
Files #3: 41
Files #4: 50
Files #5: 27
Files #6: 14
Files #7: 32
Index block 3 is allocated ...
Index block 21 is allocated ...
Index block 41 is allocated ...
Index block 50 is allocated ...
Index block 27 is allocated ...
Index block 14 is allocated ...
Index block 32 is allocated ...
File is allocated successfully for the block 5
After File Allocation, Table Contents:
0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 1
```

ALLOCATION OF INDEX BLOCK 35

```
Console Shell
-----
Indexed File Allocation
-----
Before File Allocation, Table Contents:
0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 1
Enter the index block number: 35
Enter the number of files for the block 35: 3
Enter the blocks:
Files #1: 9
Files #2: 44
Files #3: 18
Index block 9 is allocated ...
Index block 44 is allocated ...
Index block 18 is allocated ...
File is allocated successfully for the block 35
After File Allocation, Table Contents:
0 0 1 0 0 0 0 0
1 0 0 0 0 1 0 0
0 1 0 0 1 0 0 0
0 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0
1 0 0 1 0 0 0 0
0 1
Do you want to continue: Press Yes / No : yes
```

CHECKING ALREADY ALLOCATED INDEX BLOCKS FOR 32, 50

```
Console Shell
Do you want to continue: Press Yes / No : yes
-----
Indexed File Allocation
-----
Before File Allocation, Table Contents:
0 0 1 0 0 0 0 0
1 0 0 0 0 1 0 0
0 1 0 0 1 0 0 0
0 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0
1 0 0 1 0 0 0 0
0 1
Enter the index block number: 32
File / Block is already Allocated
Please try some blockAfter File Allocation, Table Contents:
0 0 1 0 0 0 0 0
1 0 0 0 0 1 0 0
0 1 0 0 1 0 0 0
0 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0
1 0 0 1 0 0 0 0
0 1
Do you want to continue: Press Yes / No : yes
-----
Indexed File Allocation
-----
Before File Allocation, Table Contents:
0 0 1 0 0 0 0 0
1 0 0 0 0 1 0 0
0 1 0 0 1 0 0 0
0 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0
1 0 0 1 0 0 0 0
0 1
Enter the index block number: 50
File / Block is already Allocated
Please try some blockAfter File Allocation, Table Contents:
0 0 1 0 0 0 0 0
1 0 0 0 0 1 0 0
0 1 0 0 1 0 0 0
0 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0
1 0 0 1 0 0 0 0
0 1
```

CHECKING INVALID INDEX NUMBER 59

```
Console Shell
-----
Indexed File Allocation
-----
Before File Allocation, Table Contents:
0 0 1 0 0 0 0 0
1 0 0 0 0 1 0 0
0 1 0 0 1 0 0 0
0 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0
1 0 0 1 0 0 0 0
0 1
Enter the index block number: 59
Invalid index Block Number.
Please Enter the index block number in the range between 1 to 50
After File Allocation, Table Contents:
0 0 1 0 0 0 0 0
1 0 0 0 0 1 0 0
0 1 0 0 1 0 0 0
0 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0
1 0 0 1 0 0 0 0
0 1
```

CHECKING ALREADY ALLOCATED INDEX NUMBERS 14, 25

```
Console Shell
Enter the index block number: 14
File / Block is already Allocated
Please try some blockAfter File Allocation, Table Contents:
0 0 1 0 0 0 0 0
1 0 0 0 0 1 0 0
0 1 0 0 1 0 0 0
0 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0
1 0 0 1 0 0 0 0
0 1
Do you want to continue: Press Yes / No : yes
-----
Indexed File Allocation
-----
Before File Allocation, Table Contents:
0 0 1 0 0 0 0 0
1 0 0 0 0 1 0 0
0 1 0 0 1 0 0 0
0 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0
1 0 0 1 0 0 0 0
0 1
Enter the index block number: 25
Enter the number of files for the block 25: 2
Enter the blocks:
Files #1: 21
Files #2: 41
File lengths are not free for the given block
After File Allocation, Table Contents:
0 0 1 0 0 0 0 0
1 0 0 0 0 1 0 0
0 1 0 0 1 0 0 0
0 0 1 0 0 0 0 1
0 0 0 0 0 0 0 0
1 0 0 1 0 0 0 0
0 1
Do you want to continue: Press Yes / No : yes
-----
Indexed File Allocation
-----
Before File Allocation, Table Contents:
0 0 1 0 0 0 0 0
1 0 0 0 0 1 0 0
0 1 0 0 1 0 0 0
```

RESULT

Thus the file allocation strategies like sequential and indexed file allocation methods have been executed successfully.