

**EXPERIMENT-1****BASIC COMMANDS IN LINUX**

AIM: To study and execute the commands

File related commands

1. Pwd Command:

DESCRIPTION: It displays the present working directory

SYNTAX: pwd

OUTPUT:



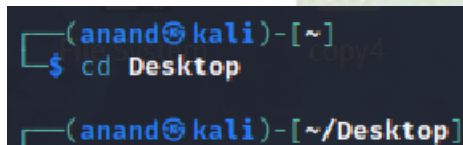
```
(anand@kali)-[~]  
$ pwd  
/home/anand
```

2. Cd Command:

DESCRIPTION: It changes the working directory i.e., change the current working to some other folder.

SYNTAX: cd [option] [directory]

OUTPUT:

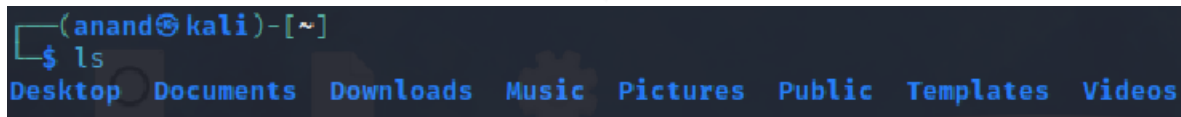


```
(anand@kali)-[~]  
$ cd Desktop  
  
(anand@kali)-[~/Desktop]
```

3. Ls Command:

DESCRIPTION: It lists the content/files listed in a particular UNIX directory SYNTAX: ls [options] [filename]

OUTPUT:



```
(anand@kali)-[~]  
$ ls  
Desktop Documents Downloads Music Pictures Public Templates Videos
```

4. Rm command:

DESCRIPTION: It removes files or directories

SYNTAX: rm [option ...] filelist

OUTPUT:

Before using rm command:

```
└─$ ls
a.out c1.c copy1 copy2 copy3 copy4 sample1 text.txt zombie.c
```

After using rm command:

```
└─$ rm text.txt

(anand@kali)-[~/Desktop]
└─$ ls
a.out c1.c copy1 copy2 copy3 copy4 sample1 zombie.c
```

#### 5. Mv command:

DESCRIPTION: It is used to move files or directories from one place to another or renames the files.

SYNTAX: mv [option] [sourceFile] [destFile]

If destination file doesn't exist then it will be created else it will overwrite and the source file is deleted.

OUTPUT:

```
└─$ mv zombie.c sample1

(anand@kali)-[~/Desktop]
└─$ ls
a.out c1.c copy1 copy2 copy3 copy4 sample1
```

#### 6. Cat Command:

DESCRIPTION: It allows us to create single or multiple files, view content of file, concatenate files and redirect output in terminal or files.

SYNTAX: cat [option ...] [file ...]

OUTPUT:

```
└─$ cat > text.txt
hi
^C
```

Text file is created

#### 7. Cmp Command:

DESCRIPTION: It compares two files and, if they differ, tells the first byte number where they differ.

SYNTAX: cmp [option ...] fromfile tofile

OUTPUT:

```
(anand@kali)-[~/Desktop]
$ cat > t1.txt
hi
^C

(anand@kali)-[~/Desktop]
$ cat > t2.txt
hello
^C

(anand@kali)-[~/Desktop]
$ cmp t1.txt t2.txt
t1.txt t2.txt differ: byte 2, line 1
```

## 8. Cp Command:

DESCRIPTION: It copies the files.

SYNTAX: cp [option] source destination

OUTPUT:

```
(anand@kali)-[~/Desktop]
$ cp t1.txt t2.txt

(anand@kali)-[~/Desktop]
$ cat t2.txt
hi
```

## 9. Echo Command:

DESCRIPTION: It is used for displaying a line of string/text that is passed as the arguments.

SYNTAX: echo [option] [string]

OUTPUT:

```
$ echo "hello world"
hello world
```

## 10. Mkdir Command:

DESCRIPTION: It creates a new directory.

SYNTAX: mkdir [options] DirectoryNames

OUTPUT:

```
(anand@kali)-[~/Desktop]
$ mkdir dir1

(anand@kali)-[~/Desktop]
$ ls
a.out  c1.c  copy1  copy2  copy3  copy4  dir1  sample1  t1.txt  t2.txt
```

## 11. Paste Command:

DESCRIPTION: It merges the lines from multiple files. It sequentially writes the corresponding lines from each file separated by a TAB delimiter on the UNIX terminal.

SYNTAX: paste [options] files-list

OUTPUT:

```
$ paste t1.txt t2.txt  
hi      hi
```

## 12. Rmdir Command:

DESCRIPTION: It removes the directories.

SYNTAX: rmdir [options] DirectoryNames

OUTPUT:

```
$ rmdir dir1  
  
(anand@kali)-[~/Desktop]  
$ ls  
a.out  c1.c  copy1  copy2  copy3  copy4  sample1  t1.txt  t2.txt
```

## 13. Head Command:

DESCRIPTION: It displays the beginning of a text file or piped data. It prints the first 10 lines of the specified files.

SYNTAX: head [option] [files]

OUTPUT:

```
$ head t1.txt  
hi
```

## 14. Tail Command:

DESCRIPTION: It displays the tail end of a text file or piped data. It prints the last 10 lines of the specified file.

SYNTAX: tail [option] [files]

OUTPUT:

```
$ tail t2.txt  
hi
```

## 15. Date Command:

DESCRIPTION: It is used to display date, time, time zone, etc. It is also used to set the date and time of the System. It is used to display the date in different formats and calculate dates over time.

SYNTAX: date [option] [+format]

OUTPUT:

```
$ date  
Saturday 04 November 2023 12:07:09 PM IST
```

## 16. Grep Command:

DESCRIPTION: It prints lines matching a pattern.

SYNTAX: grep [options] PATTERN [file]

OUTPUT:

```
$ cat num.txt
10
12
15
25
89

(anand@kali) - [~/Desktop]
$ grep "2" num.txt
12
25
```

#### 17. Touch Command:

DESCRIPTION: It creates a new file or updates its timestamp.

SYNTAX: touch [option ...] [File ...]

OUTPUT:

```
$ touch t3.txt

(anand@kali) - [~/Desktop]
$ ls
a.out  c1.c  copy1  copy2  copy3  copy4  num.txt  sample1  t1.txt  t2.txt  t3.txt
```

#### 18. Chmod Command:

DESCRIPTION: It is used to change the access permissions, change mode.

SYNTAX: chmod [options] Mode File

OUTPUT: Giving access of editing to anyone

```
$ ls -l
total 20
-rwxr-xr-x 1 anand anand 16008 Sep 20 21:47 a.out
-rw-r--r-- 1 anand anand 115 Sep 20 21:47 c1.c

(anand@kali) - [~/Desktop]
$ chmod 555 c1.c

(anand@kali) - [~/Desktop]
$ ls -l
total 20
-rwxr-xr-x 1 anand anand 16008 Sep 20 21:47 a.out
-r-xr-xr-x 1 anand anand 115 Sep 20 21:47 c1.c
```

#### 19. Man Command:

DESCRIPTION: It is the interface to view the system's reference manuals.

SYNTAX: man ls

OUTPUT:

```

File Actions Edit View Help
CHMOD(1)
NAME
  chmod - change file mode bits
SYNOPSIS
  chmod [OPTION] ... MODE[,MODE] ... FILE ...
  chmod [OPTION] ... OCTAL-MODE FILE ...
  chmod [OPTION] ... --reference=RFILE FILE ...

```

## 20. Clear Command:

DESCRIPTION: It clears the terminal screen.

SYNTAX: clear

OUTPUT: After entering the clear command, the entire terminal will be cleared

## PROCESS RELATED COMMANDS:

## 1. Top Command:

DESCRIPTION: To track the running processes on our machine.

SYNTAX: top

OUTPUT:

```

top - 12:15:22 up 59 min, 1 user, load average: 0.17, 0.27, 0.20
Tasks: 150 total, 1 running, 149 sleeping, 0 stopped, 0 zombie
%Cpu(s): 2.6 us, 3.3 sy, 0.0 ni, 94.1 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 2972.3 total, 1943.7 free, 751.7 used, 431.5 buff/cache
MiB Swap: 975.0 total, 975.0 free, 0.0 used, 2220.6 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR S  %CPU  %MEM    TIME+  COMMAND
  619 root        20   0 372952 117784 57928 S   2.7   3.9   0:50.23 Xorg
 16644 anand       20   0 443076 106672 87236 S   2.0   3.5   0:06.94 qterminal
    44 root        20   0      0      0      0 I   0.7   0.0   0:10.26 kworker/1:1-events
   951 anand      20   0 217340  4032   3508 S   0.7   0.1   0:00.06 VBoxClient

```

## 2. Ps Command:

DESCRIPTION: ps is short for Process status. It displays the currently-running processes.

SYNTAX: ps

OUTPUT:

```

$ ps
  PID TTY          TIME CMD
 16655 pts/0        00:00:10 zsh
  30332 pts/0        00:00:00 ps

```

## 3. Kill Command:

DESCRIPTION: It sends a signal to the process. There are different types of signals that we can send.



SYNTAX: kill [number]

OUTPUT: List of different types of signals

```
$ kill -l
HUP INT QUIT ILL TRAP IOT BUS FPE KILL USR1 SEGV USR2 PIPE ALRM TERM STKFLT CHLD CONT STOP TSTP
```

#### 4. Nice Command:

DESCRIPTION: The Nice command configures the priority of a Linux process before it is started. The renice command sets the priority of an already running process.

SYNTAX: nice -nice\_value command-arguments; renice -n nice\_value -p pid\_of\_the\_process

### NETWORK RELATED COMMANDS:

#### 1. Ping Command:

DESCRIPTION: It allows us to test the reachability of a device on a network.

SYNTAX: ping [host]

OUTPUT:

```
$ ping -c 5 google.com
PING google.com(maa03s44-in-x0e.1e100.net (2404:6800:4007:829::200e)) 56 data bytes
64 bytes from maa03s44-in-x0e.1e100.net (2404:6800:4007:829::200e): icmp_seq=1 ttl=59 time=117 ms
64 bytes from maa03s44-in-x0e.1e100.net (2404:6800:4007:829::200e): icmp_seq=3 ttl=59 time=44.6 ms
64 bytes from maa03s44-in-x0e.1e100.net (2404:6800:4007:829::200e): icmp_seq=4 ttl=59 time=33.3 ms
64 bytes from maa03s44-in-x0e.1e100.net (2404:6800:4007:829::200e): icmp_seq=5 ttl=59 time=31.8 ms

--- google.com ping statistics ---
5 packets transmitted, 4 received, 20% packet loss, time 4029ms
rtt min/avg/max/mdev = 31.845/56.683/117.011/35.180 ms
```

#### 2. Ifconfig Command:

DESCRIPTION: The command ifconfig stands for interface configurator. This command enables us to initialize an interface, assign an IP address, enable or disable an interface. It displays route and network interface.

SYNTAX: ifconfig

OUTPUT:

```
$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.0.105 netmask 255.255.255.0 broadcast 192.168.0.255
    inet6 2406:b400:b4:5e9c:98d0:dc7f:8fbb:83e7 prefixlen 64 scopeid 0<global>
    inet6 2406:b400:b4:5e9c:a00:27ff:fe8e:cba3 prefixlen 64 scopeid 0<global>
    inet6 fe80::a00:27ff:fe8e:cba3 prefixlen 64 scopeid 0<link>
    ether 08:00:27:8e:cb:a3 txqueuelen 1000 (Ethernet)
    RX packets 3496 bytes 253221 (247.2 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 1450 bytes 158368 (154.6 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 4 bytes 240 (240.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 4 bytes 240 (240.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

## 3. Netstat Command:

**DESCRIPTION:** The Netstat command displays active TCP connections, ports on which the computer is listening. The information this command provides can be useful in pinpointing problems in our network connections.

**SYNTAX:** netstat

**OUTPUT:**

```

L$ netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
udp        0      0 192.168.0.105:bootpc    192.168.0.1:bootps     ESTABLISHED
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags     Type       State           I-Node  Path
unix    3      [ ]       STREAM    CONNECTED      19595
unix    3      [ ]       STREAM    CONNECTED      20482
unix    3      [ ]       STREAM    CONNECTED      19816   @/tmp/.X11-unix/X0
unix    3      [ ]       STREAM    CONNECTED      19165   @/tmp/.X11-unix/X0
unix    2      [ ]       DGRAM     CONNECTED      17320
unix    3      [ ]       STREAM    CONNECTED      20496   /run/systemd/journal/stdout
unix    3      [ ]       STREAM    CONNECTED      20236

```

## 4. Nslookup Command:

**DESCRIPTION:** NsLookUp command is used to find all the IP addresses for a given domain name.

**SYNTAX:** nslookup host

**OUTPUT:**

```

L$ nslookup www.google.com
Server:         49.205.171.194
Address:        49.205.171.194#53

Non-authoritative answer:
Name:   www.google.com
Address: 142.250.196.36
Name:   www.google.com
Address: 2404:6800:4007:82a::2004

```

## 5. Telnet Command:

**DESCRIPTION:** It connects a destination via the telnet protocol. If a telnet connection establishes on any port, it means connectivity between two hosts is working fine.

**SYNTAX:** telnet hostname/IP address

## 6. Traceroute Command:

**DESCRIPTION:** A handy utility to view the number of hops and response time to get to a remote system or website is a traceroute.

**SYNTAX:** traceroute [OPTION...] HOST

**OUTPUT:**

```

L$ traceroute www.google.com
traceroute to www.google.com (142.250.196.36), 30 hops max, 60 byte packets
 1  192.168.0.1 (192.168.0.1)  5.012 ms  4.454 ms  3.777 ms
 2  * * *
 3  * * *
 4  broadband.actcorp.in (183.82.12.70)  8.919 ms  8.649 ms  8.390 ms
 5  broadband.actcorp.in (183.82.14.78)  19.393 ms  18.737 ms  18.459 ms
 6  72.14.223.26 (72.14.223.26)  18.191 ms  20.766 ms  17.040 ms
 7  * * *
 8  142.251.55.220 (142.251.55.220)  18.707 ms  74.125.242.129 (74.125.242.129)  21.304 ms  142.251.60.184 (142.251.60.184)  26.575 ms
 9  74.125.242.131 (74.125.242.131)  22.341 ms  142.251.55.29 (142.251.55.29)  23.896 ms  142.251.55.31 (142.251.55.31)  21.011 ms
10  108.170.253.113 (108.170.253.113)  19.423 ms  18.809 ms  maa03s45-in-f4.1e100.net (142.250.196.36)  18.084 ms

```

**CONCLUSION:** By executing the above commands, we have understood the command



**EXPERIMENT-2****SHELL SCRIPTING****SHELL:**

A shell is a special user program that provides an interface for the user to use operating system services. Shell accepts human-readable commands from users and converts them into something which the kernel can understand.

Ex: bash, ksh

**BASH SCRIPT:**

A bash script is a series of commands written in a file. These are read and executed by the bash program. The program executes line by line.

Creating a shell script :

We use “gedit” command

gedit:gedit is a text editor

```
cbit@cbit-VirtualBox:~/Desktop$ gedit shell1.sh
```

Changing the permission to the file:

```
cbit@cbit-VirtualBox:~/Desktop$ chmod +x shell1.sh
```

Running an .sh file

```
cbit@cbit-VirtualBox:~/Desktop$ ./shell1.sh  
Welcome to os lab
```

**Basic Commands:**

1. Declare variable and Printing  
for printing we use echo command

```
1 #!/bin/bash  
2  
3 n1=5  
4 n2=10  
5  
6 echo $n1 $n2
```

Output:

```
cbit@cbit-VirtualBox:~/Desktop$ ./shell2dp.sh
5 10
```

## 2. Input and Arithmetic

Ex1 :

```
1 #!/bin/bash
2
3 echo "enter Number1 : "
4 read a
5
6 echo "enter number2 : "
7 read b
8
9 echo "sum is : "$((a+b))
```

Output:

```
cbit@cbit-VirtualBox:~/Desktop$ ./sum.sh
enter Number1 :
10
enter number2 :
20
sum is : 30
```

Ex2:

```
1 #!/bin/bash
2
3 echo "enter Number1 : "
4 read a
5
6 echo "enter number2 : "
7 read b
8
9 m=$((a*b))
10 mod=$((a%b))
11 e=$((a**b))
12 d=$((a/b))
13
14 echo "mul : "$m "modulus : "$mod "expo : "$e "div : "$d
```

Output:

```
cbit@cbit-VirtualBox:~/Desktop$ ./arithmetic.sh
enter Number1 :
10
enter number2 :
20
mul : 200 modulus : 10 expo : 7766279631452241920 div :0
```

### 3. Conditions

Ifelse:

```
1 #!/bin/bash
2
3 read n1
4 read n2
5
6 if(($n1>$n2))
7 then
8 echo $n1 " > " $n2
9 else
10 echo $n1 " < " $n2
11 fi
```

Output:

```
cbit@cbit-VirtualBox:~/Desktop$ ./ifelse.sh
4
2
4 > 2
```

### 4. loops

for :

```
1 #!/bin/bash
2
3 echo "first 5 numbers: "
4 for i in {1..5}
5 do
6 echo $i
7 done
```

Output:

```
cbit@cbit-VirtualBox:~/Desktop$ ./for.sh
first 5 numbers:
1
2
3
4
5
```

While:

```
1 #!/bin/bash
2
3 i=1
4 while(($i<5))
5 do
6 echo "$i"
7 ((i+=1))
8 done
```

Output:

```
cbit@cbit-VirtualBox:~/Desktop$ ./while.sh
1
2
3
4
```

### Task Questions:

1.

AIM: To find the factorial of a number.

DESCRIPTION: In this program, we try to read a number from the user and find factorial for that number

PROGRAM:

```
1 #!/bin/bash
2
3 echo "number : "
4 read i
5 fact=1
6 for ((j=2;j<=i;j++))
7 do
8 fact=$((fact*j))
9 done
10 echo "factorial : "$fact
```

OUTPUT:

```
cbit@cbit-VirtualBox:~/Desktop$ ./factorial.sh
number :
5
factorial : 120
```

CONCLUSION: By executing the above shell script, we have successfully found factorial for the number given by user.

2.

AIM: To check if the given number is even or odd.

DESCRIPTION: In this program, we try to read a number from the user and check whether the given number is even or odd.

PROGRAM:

```
1 #!/bin/bash
2 echo "enetr a number : "
3 read i
4 m=$((i%2))
5 if ((m==0))
6 then
7 echo "$i is Even"
8 else
9 echo "$i is odd"
10 fi
```

OUTPUT:

```
cbit@cbit-VirtualBox:~/Desktop$ ./evenorodd.sh
enetr a number :
5
5 is odd
```

CONCLUSION: By executing the above shell script, we have successfully checked whether the given number is even or odd



3.

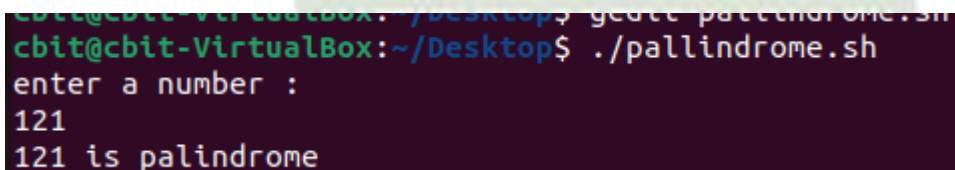
AIM: To check whether the given number is palindrome or not.

DESCRIPTION: In this program, we try to read the number from the user and check whether the given number is palindrome or not.

PROGRAM:

```
1 #!/bin/bash
2 echo "enter a number : "
3 read n
4 sum=0
5 num=$n
6 while(($n>0))
7 do
8 temp=$((n%10))
9 sum=$((sum*10+temp))
10 n=$((n/10))
11 done
12 if((num==sum))
13 then
14 echo "$num is palindrome"
15 else
16 echo "$num is not a palindrome"
17 fi
```

OUTPUT:



```
cbit@cbit-VirtualBox: ~/Desktop$ ./pallindrome.sh
enter a number :
121
121 is palindrome
```

CONCLUSION:

By executing the above shell script, we have successfully checked whether the given number is palindrome or not

4.

AIM: To print the Fibonacci Series.

DESCRIPTION: In this program, we try to print the Fibonacci Series

PROGRAM:

```
1#!/bin/bash
2echo "number of terms in fibonacci : "
3read i
4if ((i==1))
5then
6echo "0"
7elif ((i==2))
8then
9echo "0 1"
10else
11echo "0"
12echo "1"
13n1=0
14n2=1
15for ((j=3;j<=i;j++))
16do
17next=$((n1+n2))
18echo $next
19n1=$n2
20n2=$next
21done
22fi
```

OUTPUT:

```
cbit@cbit-VirtualBox:~/Desktop$ ./fibonacci.sh
number of terms in fibonacci :
6
0
1
1
2
3
5
```

CONCLUSION:

By executing the above shell script, we have successfully printed the Fibonacci Series

5.

AIM: To find power of number

DESCRIPTION: In this program, we try to get power of a number

PROGRAM:

```
#!/bin/sh
echo "Enter a number for base: "
read a
echo "Enter a number for power: "
read b
i=1
res=1
while [ $i -le $b ]
do
res=$(( $res * $a ))
i=$(( $i + 1 ))
done
echo "$a^$b = $res"
```

OUTPUT:

```
(anand@kali)-[~/Desktop]
$ ./power.sh
Enter a number for base:
10
Enter a number for power:
10
10^10 = 10000000000
```

CONCLUSION:

By executing the above shell script, we have successfully got power of a number

6.

AIM: To find whether a given number is prime or not

DESCRIPTION: In this program, we try to read a number from the user and check whether the given number is prime or not

PROGRAM:

```
#!/bin/sh
echo "Enter a number: "
read n
flag=0
if [ $n -le 1 ]
then
flag=1
else
i=2
a=$(( $n / 2 ))
while [ $i -le $a ]
do
rem=$(( $n % $i ))
```

```
if [ $rem -eq 0 ]
then
flag=1;
break;
fi
i=$(( $i + 1 ))
done
fi
if [ $flag -eq 0 ]
then
echo "$n is a prime number"
else
echo "$n is not a prime number."
fi
```

OUTPUT:

```
(anand@kali)-[~/Desktop]
$ ./prime.sh
Enter a number:
16
16 is not a prime number.
```

CONCLUSION: By executing the above shell script, we have successfully found whether the number given by the user is prime or not

7.

AIM: To demonstrate Mini Calculator

DESCRIPTION: In this program, we try to perform all basic arithmetic operations by demonstrating a mini calculator.

PROGRAM:

```
#!/bin/sh
echo "Enter two numbers: "
read a
read b
echo "Operations to be Performed: "
echo "1. Add"
echo "2. Sub"
echo "3. Multiply"
echo "4. Division"
echo "Enter your Choice: "
read c
while [ $c -le 4 ]
do
```

```
case "$c" in
    "1")
        sum=$(( $a + $b ))
        echo "$a + $b = $sum"
        ;;
    "2")
        diff=$(( $a - $b ))
        echo "$a - $b = $diff"
        ;;
    "3")
        mul=$(( $a * $b ))
        echo "$a * $b = $mul"
        ;;
    "4")
        div=$(( $a / $b ))
        echo "$a / $b = $div"
        ;;
esac
echo "Enter your choice: "
read c
done
```

## OUTPUT:

```
(anand@kali)-[~]
$ ./calc.sh
Enter two numbers:
10
20
Operations to be Performed:
1. Add
2. Sub
3. Multiply
4. Division
Enter your Choice:
1
10 + 20 = 30
Enter your choice:
^C
```



## EXPERIMENT-5

### PROCESS RELATED SYSTEM CALLS

1. Go to the manual pages and write in brief about fork(), getpid(), getppid(), exec() and wait() system call. Write syntax, header files required and explanation

System Calls:

#### 1. Fork():

Description: `fork()` creates a new process by duplicating the calling process. The new process is referred to as the child process, and the calling process becomes the parent process.

Header Files:

```
#include <sys/types.h>
#include <unistd.h>
```

Syntax: pid\_t fork(void);

Explanation:

- The `fork()` system call creates a new process, which is a copy of the current process, including its address space.
- On success, the PID of the child process is returned in the parent, and 0 is returned in the child.
- On failure, 1 is returned in the parent, no child process is created, and errno is set appropriately.

#### 2. getpid():

Description: `getpid()` is used to obtain the process identification (PID) of the calling process.

Header Files:

```
#include <sys/types.h>
#include <unistd.h>
```

Syntax:

```
pid_t getpid(void);
```

Explanation:

- `getpid()` returns the PID of the calling process.
- This function is always successful.

#### 3. getppid():

Description: `getppid()` is used to obtain the PID of the parent process of the calling process.

Header Files:

```
#include <sys/types.h>
```

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

Syntax:

```
pid_t getppid(void);
```

Explanation:

- `getppid()` returns the PID of the parent process of the calling process.
- This will be either the ID of the process that created this process using `fork()`, or if that process has already terminated, the ID of the process to which this process has been reparented.

#### 4. exec():

Description: The `exec()` family of functions are used to execute a file, replacing the current process image with a new one.

Header Files:

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

Syntax:

- `int execl (const char \*pathname, const char \*arg, ... /\* (char \*) NULL \*/);`
- `int execlp (const char \*file, const char \*arg, ... /\* (char \*) NULL \*/);`
- `int execl (const char \*pathname, const char \*arg, ..., (char \*) NULL, char \*const envp[]);`
- `int execv (const char \*pathname, char \*const argv[]);`
- `int execvp (const char \*file, char \*const argv[]);`
- `int execvpe (const char \*file, char \*const argv[], char \*const envp[]);`

Explanation:

- The `exec()` functions are used to execute a new process image.
- They replace the current process image with the new one specified by the function.
- The functions vary in their use of command-line arguments and environment variables.
- The functions return only if an error occurs, with a return value of 1, and errno is set to indicate the error.

#### 5. wait():

Description: The `wait()` system call suspends the execution of the calling thread until one of its child processes terminates.

Header Files:

```
#include <sys/types.h>
```

```
#include <sys/wait.h>
```

Syntax:

```
pid_t wait(int *wstatus);
```

Explanation:

- `wait()` is used to wait for the termination of a child process.
- It suspends the calling process until one of its child processes terminates.

- On success, it returns the PID of the terminated child process.
- On error, it returns 1.

2. Execute the following program and write the output and explain it.

AIM: To demonstrate the fork system call

DESCRIPTION: fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

PROGRAM:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    int a = 2;
    pid_t pid;
    pid = fork();
    printf("%d\n", pid);

    if (pid < 0) {
        printf("fork failed");
    } else if (pid == 0) {
        printf("child process \t a is: ");
        printf("%d\n", ++a);
    } else {
        printf("parent process \t a is: ");
        printf("%d\n", --a);
    }

    printf("exiting with x=%d\n", a);

    return 0;
}
```

OUTPUT:

```
(anand@kali)-[~/Desktop]
$ ./a.out
89628
parent process   a is: 1
exiting with x=1
0
child process   a is: 3
exiting with x=3
```

CONCLUSION:

By the executing the above program, we have successfully demonstrated the fork() System call

3. Execute the following program and write the output. You need to execute the following program twice once without using wait() and other one using wait() system call. Explain your Results.

AIM: To demonstrate the getpid(), getppid() and wait System calls.

DESCRIPTION: getpid() command is used to get process identification. This is often used by routines that generate unique temporary filenames. This function is always successful. getppid() command is used to get process identification. This function is always successful. The wait() system call suspends execution of the calling thread until one of its children terminates.

#### PROGRAM:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    int a = 2;
    pid_t pid;
    pid = fork();
    printf("%d\n", pid);
    if (pid < 0) {
        printf("Error");
    } else if (pid == 0) {
        printf("child process\n");
        printf("%d\n", ++a);
        printf("I am the child and my process id is %d\n", getpid());
        printf("I am the child and my parent process id is %d\n", getppid());
    } else {
        wait(NULL); // You should use wait(NULL) to wait for the child process to finish.
        printf("parent process\n");
        printf("%d\n", --a);
        printf("I am the parent and my process id is %d\n", getpid());
        printf("I am the parent and my child process id is %d\n", pid);
    }
    printf("exiting with x=%d\n", a);
    return 0;
}
```

#### OUTPUT:

```
(anand@kali)-[~/Desktop]
$ ./a.out
93481
0
child process
3
I am the child and my process id is 93481
I am the child and my parent process id is 93480
exiting with x=3
parent process
1
I am the parent and my process id is 93480
I am the parent and my child process id is 93481
exiting with x=1
```

**CONCLUSION:**

Here, in this program when wait() system call is used then the parent process waits until the child process is completed. Getpid() system call returns the process id of the that process and getppid() returns the process id of the parent process. When wait() is not used then first parent process will be completed then child process will come for the execution at that time the parent process is already completed then the child process will be taken by the another process depending upon the scheduler

4. Write a program to demonstrate the concept of orphan process and Zombie process

AIM: To demonstrate the demonstrate the concept of orphan process and Zombie process

**DESCRIPTION:**

Orphan Process: A child process that remains running even after its parent process is terminated or completed without waiting for the child process execution is called an orphan. A process becomes an orphan

unintentionally. Sometime intentionally becomes orphans due to long-running time to complete the assigned task without user attention. The orphan process has controlling terminals.

Zombie Process: A Zombie is a process that has completed its task but still, it shows an entry in a process table. The zombie process usually occurred in the child process. Very short time the process is a zombie. After the process has completed all of its tasks it reports the parent process that it has about to terminate. Zombie is unable to terminate itself because it is treated as a dead process. So parent process needs to execute to terminate the command to terminate the child.

**PROGRAM:**

Zombie Process:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    pid_t child_pid = fork();
    if (child_pid > 0) {
        sleep(10);
        printf("Hello\n");
    } else {
        exit(0);
    }
    return 0;
}
```

OUTPUT:



```
(anand@kali)-[~/Desktop]
$ ./a.out
Hello
```

### Orphan Process:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    int pid;
    pid = fork();
    if (pid == 0) {
        printf("I am the child, my process ID is %d\n", getpid());
        printf("My parent's process ID is %d\n", getppid());
        sleep(5);
        printf("\nAfter sleep\nI am the child, my process ID is %d\n", getpid());
        printf("My parent's process ID is %d\n", getppid());
        exit(0);
    } else {
        sleep(10);
        printf("I am the parent, my process ID is %d\n", getpid());
        printf("The parent's parent, process ID is %d\n", getppid());
        printf("Parent terminates\n");
    }
    return 0;
}
```

### OUTPUT:

```
(anand@kali)-[~/Desktop]
$ ./a.out
I am the child, my process ID is 99497
My parent's process ID is 99496

After sleep
I am the child, my process ID is 99497
My parent's process ID is 99496
I am the parent, my process ID is 99496
The parent's parent, process ID is 16655
Parent terminates
```

### CONCLUSION:

By executing the above program, we have successfully demonstrated the orphan and zombie process

5. Write a C program to create a child process and allow the parent to display “parent” and the child to display “child” on the screen.?

AIM: To create a child process and allow the parent to display “parent” and the child to display “child” on the screen

DESCRIPTION: Here in this program, we try to create a child process and allow the parent to display “parent” and the child to display “child” on the screen

PROGRAM:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    int pid = fork();
    if (pid > 0) {
        printf("I am Parent Process\n");
        printf("ID: %d\n", getpid());
    } else if (pid == 0) {
        printf("I am Child Process\n");
        printf("ID: %d\n", getpid());
    } else {
        printf("Failed to create Child Process\n");
    }
    return 0;
}
```

OUTPUT:

```
(anand@kali)-[~/Desktop]
$ ./a.out
I am Parent Process
I am Child Process
ID: 101896
ID: 101895
```

CONCLUSION:

By executing the above program, we have successfully created a child process and allowed child to print “child” and parent to print “parent”

## EXEC() SYSTEM CALL

1. Goto the manual pages and write in brief about exec () system call. Write syntax, header files required and explanation

DESCRIPTION: The exec() system call is used to make the processes. When the exec() function is used, the currently running process is terminated and replaced with the newly formed process. In other words, only the new process persists after calling exec(). The parent process is shut down. This

system call also substitutes the parent process's text segment, address space, and data segment with the child process.

HEADER FILES: #include<unistd.h>

#### SYNTAX:

```
extern char **environ;
int execl(const char *pathname, const char *arg, ... /* (char *) NULL */);
int execlp(const char *file, const char *arg, ... /* (char *) NULL */);
int execlx(const char *pathname, const char *arg, ... /*, (char *) NULL, char *const envp[] */);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

2.

AIM: To demonstrate exec system call

DESCRIPTION: The exec() system call is used to make the processes. When the exec() function is used, the currently running process is terminated and replaced with the newly formed process. In other words, only the new process persists after calling exec(). The parent process is shut down. This system call also substitutes the parent process's text segment, address space, and data segment with the child process.

#### PROGRAM:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <errno>
int main(int argc, char *argv[]) {
    int pid, childpid, status;
    pid = fork();

    if (pid < 0) {
        printf("fork failed");
    } else if (pid == 0) {
        execl("/bin/ls", "ls", NULL);
        exit(0);
    } else {
        wait(NULL);
        printf("child process complete\n");
    }
    return 0;
}
```

OUTPUT:

```
(anand@kali)-[~/Desktop]
$ ./a.out
160120733012_Record.pdf a.out c1.c fork2.c fork.c orphan.c output.c power.sh prime.sh wait.c zombie.c
child process complete
return 0
```

### CONCLUSION:

By executing the above program, we have successfully demonstrated the exec system call

3. Write a program using the fork() system call that computes the factorial of a given integer in the child process. The integer whose factorial is to be computed will be provided in the command line. For example, if 5 is provided, the child will compute 5! and output 120. Because the parent and child processes have their own copies of the data it will be necessary for the child to output the factorial. Have the parent invoke the wait () call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a non-negative number is passed on the command line.

AIM: To create a child process and allow the child process to calculate factorial of a number

DESCRIPTION: Here in this program, we have to create a child process and we have to allow the child process to calculate factorial of a number and parent have to invoke the wait() call to wait for the child process to complete before exiting the program

### PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(int argc, char *argv[]) {
    pid_t ret;
    int i, num, sum = 0, fact = 1;
    ret = fork();
    for (i = 1; i < argc; i++) {
        num = atoi(argv[i]);
    }
    if (ret == 0 && num > 0) {
        for (i = 1; i <= num; i++) {
            fact *= i;
        }
        printf("\n[ID = %d] Factorial of %d is %d\n", getpid(), num, fact);
    } else {
        if (num < 0)
            printf("Invalid input");
        wait(NULL);
        printf("\n[ID = %d] DONE\n", getpid());
    }
    return 0;
}
```

### OUTPUT:

```
(anand@kali)-[~/Desktop]
$ ./a.out 9

[ID = 112478] Factorial of 9 is 362880
[ID = 112477] DONE
```

**CONCLUSION:**

By executing the above program, we have successfully created a child process and allowed the child factorial to calculate the factorial of a number.

4. Write a program that creates a child process and the child process checks whether a given number is palindrome or not. Make use of `exec()` system call for finding palindrome.

**AIM:** To create a child process and the child process checks whether a given number is palindrome or not

**DESCRIPTION:** Here in this program, we have to create a child process and the child process checks whether a given number is palindrome or not. Here `exec()` system call should be used for finding the palindrome

**PROGRAM:****EXEC.C**

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("PID of execute.c = %d\n", getpid());
    char *args[] = {"pal", NULL};
    execv("/home/anand/Desktop/pal", args);
    printf("The control never comes back to this line #unreachable");
    return 0;
}
```

**PAL.C**



```
#include <unistd.h>
#include <sys/wait.h>
void palindrome() {
    int n, r, sum = 0, temp;
    printf("Checking palindrome");
    printf("\nEnter the number: ");
    scanf("%d", &n);
    temp = n;
    while (n > 0) {
        r = n % 10;
        sum = (sum * 10) + r;
        n = n / 10;
    }
    if (temp == sum)
        printf("Palindrome number ");
    else
        printf("Not palindrome");
}

int main() {
    int pid;
    pid = fork();
    if (pid == 0) {
        printf("I am the child process");
        palindrome();
    } else {
        wait(NULL);
        printf("\nParent terminating after child");
    }
}
```

**OUTPUT:**

```
(anand@kali)~/Desktop
$ gcc -o pal /home/anand/Desktop/pal.c
```

```
(anand@kali)~/Desktop
$ gcc -o exec /home/anand/Desktop/exec.c

(anand@kali)~/Desktop
$ ./exec
PID of execute.c = 16584
I am the child processChecking palindrome
Enter the number: 121
Palindrome number
Parent terminating after child
```

**CONCLUSION:**

By executing the above program, we have successfully created the child process and allowed the child process to check whether the given number is a palindrome or not

**DEMONSTRATION OF LINUX/UNIX PROCESS RELATED SYSTEM CALLS**

AIM: To demonstrate the Linux/Unix Process Related System Calls – getuid, setuid, nice

**DESCRIPTION:****1. GETUID:**

DESCRIPTION: This system call is used to get user identity

**HEADER FILES:**

```
#include  
<unistd.h>  
#include  
<sys/types.h>
```

**SYNTAX:**

```
uid_t getuid(void)
```

**RETURN VALUE:**

getuid() returns the real user ID of the current process

**2. SETUID:**

**DESCRIPTION:** This system call is used to set user identity. It sets the effective user ID of the calling process

**HEADER FILES:**

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

**SYNTAX:**

```
int setuid(uid_t uid)
```

**RETURN VALUE:**

On success, zero is returned. On error, -1 is returned, and errno is set to indicate the error

**3. NICE:**

**DESCRIPTION:** The Nice command configures the priority of a Linux process before it is started. The renice command sets the priority of an already running process.

**SYNTAX:** nice -nice\_value command-arguments; renice -n nice\_value -p pid\_of\_the\_process

**PROGRAM:****GETUID():**

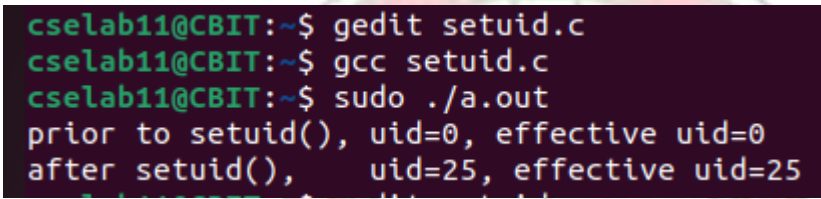
```
#include <stdio.h>  
#include <unistd.h>  
#include <stdlib.h>  
int main(void){  
    printf("The real user ID is %d\n", getuid());  
    printf("The effective user ID is %d\n", geteuid());  
    return 0;  
}
```

**OUTPUT:**

```
cse1ab11@CBIT:~$ gedit getuid.c  
cse1ab11@CBIT:~$ gcc getuid.c  
cse1ab11@CBIT:~$ ./a.out  
The real user ID is 1000  
The effective user ID is 1000
```

**SETUID():**

```
#define _POSIX_SOURCE
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
void main() {
printf("prior to setuid(), uid=%d, effective uid=%d\n", (int) getuid(), (int) geteuid());
if (setuid(25) != 0)
perror("setuid() error");
else
printf("after setuid(),uid=%d, effective uid=%d\n", (int) getuid(), (int) geteuid());
}
```

**OUTPUT:**

```
cselab11@CBIT:~$ gedit setuid.c
cselab11@CBIT:~$ gcc setuid.c
cselab11@CBIT:~$ sudo ./a.out
prior to setuid(), uid=0, effective uid=0
after setuid(), uid=25, effective uid=25
```

**NICE():**

```
#include <stdio.h> #include <sys/resource.h> int my_nice(int incr)
{
int prio = getpriority(PRIO_PROCESS, 0);
if (setpriority(PRIO_PROCESS, 0, prio + incr) == -1) return -1;
prio = getpriority(PRIO_PROCESS, 0); return prio;
}
int main(void)
{
int prio = getpriority(PRIO_PROCESS, 0); printf("Current priority = %d\n", prio); printf("\nAdding +5 to the priority\n"); my_nice(5);
prio = getpriority(PRIO_PROCESS, 0); printf("Current priority = %d\n", prio); printf("\nAdding -7 to the priority\n"); my_nice(-7);
prio = getpriority(PRIO_PROCESS, 0); printf("Current priority = %d\n", prio); return 0;
}
```

**OUTPUT:**

```
cselab11@CBIT:~$ gedit nice.c
cselab11@CBIT:~$ gcc nice.c
cselab11@CBIT:~$ ./a.out
Current priority = 0

Adding +5 to the priority
Current priority = 5

Adding -7 to the priority
Current priority = 5
```

**CONCLUSION:**

By executing the above program, we have successfully demonstrated the process related system calls – getuid(), setuid(), nice()



**EXPERIMENT-6****FILE RELATED SYSTEM CALLS:**

1. Go to manual pages and write in brief about creat(), open(), read(), write(), close(), lseek(), stat() system call. Write syntax, header files required and explanation

**1. Creat():**

DESCRIPTION: This command is used to create a file HEADER

FILES:

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

SYNTAX:

int creat(const char \*path, mode\_t mod);

RETURN VALUE: The function returns the file descriptor or in case of an error -1

**2. Open():**

DESCRIPTION: This command is used to open or create a file HEADER

FILES:

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

SYNTAX:

int open(const char \*path, int flags,... /\* mode\_t mod \*/);

RETURN VALUE: This function returns the file descriptor or in case of an error -1

**3. Read():**

DESCRIPTION: This command is used to read from file descriptor HEADER

FILES: #include <unistd.h> SYNTAX:

ssize\_t read(int fd, void\* buf, size\_t noct);

RETURN VALUE: The function returns the number of bytes read, 0 for end of file (EOF) and 1 in case an error occurred.

**4. Write():**

DESCRIPTION: This command is used to send a message to another user.

HEADER FILES: #include <unistd.h>

SYNTAX:

```
ssize_t write(int fd, const void* buf, size_t noct);
```

RETURN VALUE: The function returns the number of bytes written and the value -1 in case of an error.

#### 5. Close():

DESCRIPTION: This command is used to close a file descriptor HEADER

FILES: #include <unistd.h> SYNTAX:

```
int close(int fd);
```

RETURN VALUE: The function returns 0 in case of success and -1 in case of an error. At the termination of a process an open file is closed anyway.

#### 6. Lseek():

DESCRIPTION: This command is used for reposition read/write file offset HEADER FILES:

```
#include <sys/types.h>
```

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

```
off_t lseek(int fd, off_t offset, int ref);
```

RETURN VALUE: The function returns the displacement of the new current position from the beginning of the file or -1 in case of an error.

#### 7. Stat():

DESCRIPTION: This command is to display file or file system status.

HEADER FILES:

```
#include <sys/types.h>
```

```
#include <sys/stat.h> SYNTAX:
```

```
int stat(const char* path, struct stat* buf);
```

RETURN VALUE: The function returns 0 in case of success and -1 in case of an error.

## 2. WRITE A PROGRAM TO CREATE A FILE.

AIM: To demonstrate create system call

DESCRIPTION:

This command is used to create a file. Here in this program, we try to create a file using creat system call

ALGORITHM:

Step-1: Start



Step-2: fd -> creat("first.txt", S\_IREAD|S\_IWRITE) Step-3: fd1 -> creat("second.txt", S\_IREAD|S\_IWRITE) Step-4: Print fd  
Step-5: Print fd1 Step-6: if fd=-1  
Print "Error"  
Step-7: else  
Print "Success"  
Step-8: close(fd) Step-9: close(fd1) Step-10: End

## PROGRAM:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main()
{
    int fd;
    int fd1;
    fd = creat("first.txt", S_IREAD | S_IWRITE);
    fd1 = creat("second.txt", S_IREAD | S_IWRITE);
    printf("%d\n", fd);
    printf("%d\n", fd1);
    if (fd == -1)
        printf("ERROR\n");
    else
        printf("SUCCESS\n");
    close(fd);
    return 0;
}
```

## OUTPUT:

```
(anand@kali) - [~/Desktop]
$ ls -tl
total 2816
-rw-r--r-- 1 anand anand 0 Nov 26 20:17 first.txt
-rw-r--r-- 1 anand anand 0 Nov 26 20:17 second.txt
-rwxr-xr-x 1 anand anand 16104 Nov 26 20:17 a.out
```

## CONCLUSION:

By executing the above program, we have successfully created the files using Creat system call.

### 3. PROGRAM TO WRITE CONTENTS FROM FILE TO CONSOLE

AIM: To demonstrate write system call

**DESCRIPTION:**

This command is used to send a message to another user. Here in this program, we try to write contents from file to console

**PROGRAM:**

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
int main(int argc, char *argv[]) {
    int fd;
    int n_char = 0;
    char buffer[1];
    fd = open("first.txt", O_RDONLY);
    if (fd == -1) {
        exit(-1);
    }
    while ((n_char = read(fd, buffer, 1)) != 0) {
        write(1, buffer, n_char);
    }
    return 0;
}
```

**OUTPUT:**

```
(anand@kali)-[~/Desktop]
$ ./a.out
Hello world! This is text in First txt Document
```

**CONCLUSION:**

By executing the above program, we have successfully copied the contents from file to console

**4. PROGRAM TO READ FROM ONE FILE AND WRITE TO ANOTHER FILE**

AIM: To copy the contents of one file to another file

**DESCRIPTION:**

Here in program, we try to read the contents from one file and write that content into another file

**ALGORITHM:**

Step-1: Start

Step-2: fd1->open("first.txt",O\_RDONLY) Step-3: Print fd1

Step-4: fd2->creat("second.txt",S\_IRREAD|S\_IWRITE) Step-5: Print fd2

Step-6: if fd1<0 or fd2<0

Print "Error"

exit(1)

Step-7: while read(fd1,ch,1)>0 write(fd2,ch,1)

Print ch Step-8: close fd1 Step-9: close fd2 Step-10: End

#### PROGRAM:

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
int main() {
    int fd1, fd2;
    char ch[1];
    fd1 = open("first.txt", O_RDONLY);
    printf("%d\n", fd1);
    fd2 = creat("second.txt", S_IRREAD | S_IWRITE);
    printf("%d\n", fd2);
    if (fd1 < 0 || fd2 < 0) {
        printf("Error");
        exit(-1);
    }
    while((read(fd1, ch, 1)) > 0) {
        write(fd2, ch, 1);
        printf("%c", ch[0]);
    }
    close(fd1);
    close(fd2);
    return 0;
}
```

#### OUTPUT:

```
(anand@kali)-[~/Desktop]
$ ./a.out
3
4
Hello world! This is text in First txt Document

(anand@kali)-[~/Desktop]
$ cat second.txt
Hello world! This is text in First txt Document
```

#### CONCLUSION:

By executing the above program, we have successfully copied the contents of file to another file

## 5. PROGRAM TO SHOW THE WORKING OF THE LSEEK FUNCTION

AIM: To demonstrate the working of the lseek function

### DESCRIPTION:

This command is used for reposition read/write file offset. Here in this program, we try to demonstrate the lseek system call

### ALGORITHM:

Step-1: Start

Step-2: fd1->open("lseek\_example",O\_RDWR) Step-3: buffer[0]->'1'

Step-4: do

Read buffer[0]

if buffer[0]!='#'

write(fd1,buffer,1)

while (buffer[0]!='#')

Step-5: close(fd1)

Step-6: fd2->open("lseek\_example",O\_RDWR) Step-7: lseek(fd2,2\*sizeof(char),0)

Step-8: do

k=read(fd2,&buffer[0],1) if k!=0

Print buffer[0] while k!=0

Step-9: End

### PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fd = open("lseek_example.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    if (fd == -1) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }
    char data1[] = "This is position 0.";
    char data2[] = "This is position 10.";
    char data3[] = "This is position 20.";
    write(fd, data1, sizeof(data1) - 1);
    lseek(fd, 10, SEEK_SET);
    write(fd, data2, sizeof(data2) - 1);
    lseek(fd, 20, SEEK_SET);
    write(fd, data3, sizeof(data3) - 1);
    lseek(fd, 0, SEEK_SET);
    char buffer[100];
```

```
ssize_t bytesRead;
while ((bytesRead = read(fd, buffer, sizeof(buffer))) > 0) {
    write(STDOUT_FILENO, buffer, bytesRead);
}
close(fd);
return 0;
}
```

## OUTPUT:

```
(anand@kali)-[~/Desktop]
$ ./a.out
This is poThis is poThis is position 20.
```

## CONCLUSION:

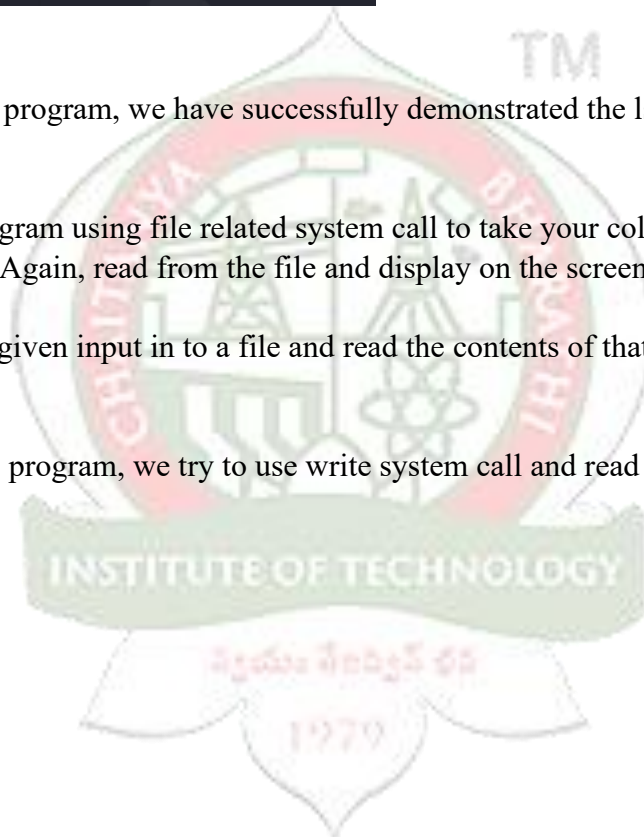
By executing the above program, we have successfully demonstrated the lseek function

**TASK-1:** Write a C program using file related system call to take your college name as input from the user, write it to the file. Again, read from the file and display on the screen

**AIM:** To write the user given input in to a file and read the contents of that file and display on the screen.

**DESCRIPTION:** In this program, we try to use write system call and read system calls.

## PROGRAM:



```
#include<stdio.h>
#include<string.h>
#include<fcntl.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>
int main() {
    int res;
    char ch[12];
    printf("To write the content entered by the user into a file:\n\n");
    printf("Enter data you want to enter into the file: ");
    scanf("%s", ch);
    printf("%s\n", ch);
    res = open("create.txt", O_RDWR);
    write(res, ch, strlen(ch));
    close(res);
    printf("Successfully updated the contents into the file\n");
    printf("\nTo read the contents from the same file:\n");
    res = open("create.txt", O_RDONLY, 0);
    read(res, &ch, 12);
    printf("%s", ch);
    return 0;
}
```

**OUTPUT:**

```
(anand@kali)-[~/Desktop]
$ ./a.out
To write the content entered by the user into a file:

Enter data you want to enter into the file: CBIT
CBIT
Successfully updated the contents into the file

To read the contents from the same file:
CBIT
```

**CONCLUSION:**

By executing the above program, we have successfully taken the input from user and written in to the file and again read from the file and displayed it on the screen.

**TASK-2:** Program for simulation of cp command using file related system calls.

**AIM:** To simulate cp command using file related system calls

**DESCRIPTION:** The command cp is used to copy a file into another file. In this program, we try to simulate the working of the cp command



## PROGRAM:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
void main()
{
    char buf;
    int fd_one, fd_two;
    fd_one = open("first.txt", O_RDONLY);
    if (fd_one == -1)
    {
        printf("Error opening first_file\n");
        close(fd_one);
        return;
    }
    fd_two = open("second.txt", O_WRONLY);
    while(read(fd_one, &buf, 1))
    {
        write(fd_two, &buf, 1);
    }
    printf("Successful copy");
    close(fd_one);
    close(fd_two);
}
```

## OUTPUT:

```
(anand@kali)-[~/Desktop]
$ cat first.txt
HELLO WORLD!

(anand@kali)-[~/Desktop]
$ cat second.txt

(anand@kali)-[~/Desktop]
$ gcc fcp.c

(anand@kali)-[~/Desktop]
$ ./a.out
Successful copy

(anand@kali)-[~/Desktop]
$ cat first.txt
HELLO WORLD!

(anand@kali)-[~/Desktop]
$ cat second.txt
HELLO WORLD!
```

**TASK-3:** Program for simulation of grep command using file related system calls.

**AIM:** To simulate grep command using file related system calls.

**DESCRIPTION:** The command grep is used to print the lines matching a pattern. In this program, we try to simulate the grep command.

**PROGRAM:**

```
#include<stdio.h>
#include<string.h>
int main() {
    char fn[10], pat[10], temp[200];
    FILE *fp;
    printf("Enter file name: ");
    scanf("%s", fn);
    printf("Enter pattern to be searched: ");
    scanf("%s", pat);
    fp = fopen(fn, "r");
    while (!feof(fp)) {
        fgets(temp, 1000, fp);
        if (strstr(temp, pat))
            printf("%s", temp);
    }
    fclose(fp);
    return 0;
}
```

**OUTPUT:**

```
(anand@kali)-[~/Desktop]
$ ./a.out
Enter file name: first.txt
Enter pattern to be searched: H
HELLO WORLD!
HELLO WORLD!
```

**CONCLUSION:**

By executing the above program, we have successfully simulated the grep command using file related system calls.

**DEMONSTRATION OF LINUX/UNIX FILE RELATED SYSTEM CALLS**

AIM: To demonstrate Linux/Unix File Related System Calls – mkdir, chmod, chown

**DESCRIPTION:****1. MKDIR():**

Description: The mkdir() function shall create a new directory with name path

Header Files:

```
#include <sys/stat.h>
```

Syntax:

```
int mkdir(const char *path, mode_t mode);
```

Return Value:

Upon successful completion, mkdir() shall return 0. Otherwise, -1 shall be returned, no directory shall be created, and errno shall be set to indicate the error.

**2. CHMOD():**

Description: The chmod() and fchmod() system calls change a file's mode bits. (the mode consists of the file permission bits plus the set-user-ID, set-group-ID, and sticky bits)

Header files:

```
#include <sys/types.h> #include <sys/stat.h> Syntax:
```

```
int chmod(const char *path, mode_t mode); int fchmod(int fildes, mode_t mode); Return Value:
```

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

**3. CHOWN():**

Description: The chown() function sets the owner ID and group ID of the file that pathname specifies.

Header files:

```
#include <sys/types.h> #include <unistd.h> Syntax:
```

```
int chown(const char *pathname, uid_t owner, gid_t group); int fchown(int fildes, uid_t owner, gid_t group);
```

```
int lchown(const char *pathname, uid_t owner, gid_t group);
```

Return Value:

If successful, chown(), fchown(), and lchown() return zero. On failure, they return -1, make no changes to the owner or group of the file, and set errno

**PROGRAM:****Mkdir():**

```
#include<sys/stat.h>
```

```
#include<sys/types.h>
```

```
#include<stdio.h>
void main(){
int status;
char dir_name[10]; scanf("%s", dir_name);
status = mkdir(dir_name, 0755); if(status == 0)
printf("Directory creation successfull"); else
printf("Directory not created");
}
```

**OUTPUT:**

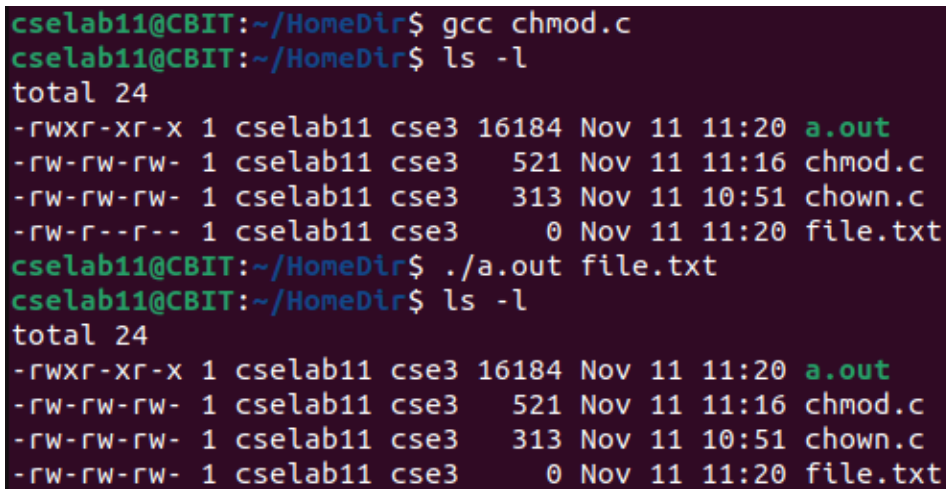
```
cselab11@CBIT:~$ gcc mkdir.c
cselab11@CBIT:~$ ./a.out
HomeDir
Directory creation successfullcselab11@CBIT:~$
```

```
Directory creation successfullcselab11@CBIT:~$ ls
050          evenodd.sh  fsf.c       pal.sh       Templates
add.sh        even.sh       fst.txt     parent_child.c test.sh
Anusha       exec.c        ghj         Pictures    untitled.sh
a.out        fac.sh        hello.sh    pow.sh       Videos
arm.sh       fact_fork.c   hgfsst     prime.sh    wait1.c
bankers.c    fact.sh       Home        Public       wait.c
cal.sh       fact.sh.save  HomeDir    reverse.sh   zombie.c
Desktop      fgt          mkdir.c    reverse.sh.save
Documents    fib.sh       Music      snap
Downloads   fork.c       pali.sh    stat.c
```

**Chmod():**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
int main(int argc, char *argv[])
{
const char *filename; struct stat fs;
int r;
filename = argv[1];
r = stat(filename,&fs);
if( r==-1)
{
fprintf(stderr,"Error reading \"%s\n",filename); exit(1);
}
r = chmod( filename, fs.st_mode | S_IWGRP+S_IWOTH ); if( r!=0)
{
```

```
fprintf(stderr,"Unable to reset permissions on \"%s\\n",filename);
exit(1);
}
stat(filename,&fs);
return(0);
}
```

**OUTPUT:**

```
cselab11@CBIT:~/HomeDir$ gcc chmod.c
cselab11@CBIT:~/HomeDir$ ls -l
total 24
-rwxr-xr-x 1 cselab11 cse3 16184 Nov 11 11:20 a.out
-rw-rw-rw- 1 cselab11 cse3 521 Nov 11 11:16 chmod.c
-rw-rw-rw- 1 cselab11 cse3 313 Nov 11 10:51 chown.c
-rw-r--r-- 1 cselab11 cse3 0 Nov 11 11:20 file.txt
cselab11@CBIT:~/HomeDir$ ./a.out file.txt
cselab11@CBIT:~/HomeDir$ ls -l
total 24
-rwxr-xr-x 1 cselab11 cse3 16184 Nov 11 11:20 a.out
-rw-rw-rw- 1 cselab11 cse3 521 Nov 11 11:16 chmod.c
-rw-rw-rw- 1 cselab11 cse3 313 Nov 11 10:51 chown.c
-rw-rw-rw- 1 cselab11 cse3 0 Nov 11 11:20 file.txt
```

**Chown():**

```
#include <stdio.h> #include <stdlib.h> #include <sys/types.h> #include <unistd.h>
int main( int argc, char** argv )
{
    int i;
    int ecode = 0;
    for( i = 1; i < argc; i++ )
    {
        if( chown( argv[i], 1000, 24 ) == 0 )
        {
            perror( argv[i] ); ecode++;
        }
    }
    exit( ecode );
}
```

**OUTPUT:**

```
cselab11@CBIT:~$ cd HomeDir
cselab11@CBIT:~/HomeDir$ gcc chown.c
cselab11@CBIT:~/HomeDir$ ls -l
total 20
-rwxr-xr-x 1 cselab11 cse3 16048 Nov 11 10:55 a.out
-rw-r--r-- 1 cselab11 cse3  313 Nov 11 10:51 chown.c
-rw-r--r-- 1 cselab11 cse3    0 Nov 11 10:53 file.txt
cselab11@CBIT:~/HomeDir$ ./a.out file.txt
file.txt: Success
cselab11@CBIT:~/HomeDir$ ls -l
total 20
-rwxr-xr-x 1 cselab11 cse3 16048 Nov 11 10:55 a.out
-rw-r--r-- 1 cselab11 cse3  313 Nov 11 10:51 chown.c
-rw-r--r-- 1 cselab11 cdrom  0 Nov 11 10:53 file.txt
```

**CONCLUSION:**

By executing the above program, we have successfully demonstrated the File Related System Calls – mkdir, chmod, chown.





**EXPERIMENT-3****DEMONSTRATION OF CPU SCHEDULING****1. FIRST COME FIRST SERVE SCHEDULING (FCFS)**

AIM: To demonstrate the First Come First Serve CPU Scheduling

**DESCRIPTION:**

First Come, First Served (FCFS) also known as First In, First Out (FIFO) is the CPU scheduling algorithm in which the CPU is allocated to the processes in the order they are queued in the ready queue. FCFS follows non-pre-emptive scheduling which mean once the CPU is allocated to a process it does not leave the CPU until the process will not get terminated or may get halted due to some I/O interrupt.

**PROGRAM:**

```
#include <stdio.h>
int waitingtime(int proc[], int n, int burst_time[], int wait_time[], int at[]) {
    int i;
    wait_time[0] = 0;
    for (i = 1; i < n; i++)
        wait_time[i] = burst_time[i - 1] + wait_time[i - 1] - (at[i] - at[i - 1]);
    return 0;
}
int turnaroundtime(int proc[], int n, int burst_time[], int wait_time[], int tat[]) {
    int i;
    for (i = 0; i < n; i++)
        tat[i] = burst_time[i] + wait_time[i];
    return 0;
}
int avgtime(int proc[], int n, int burst_time[], int at[]) {
    int wait_time[n], tat[n], total_wt = 0, total_tat = 0;
    int i;
    waitingtime(proc, n, burst_time, wait_time, at);
    turnaroundtime(proc, n, burst_time, wait_time, tat);
    printf("Processes \t Arrival time \t Burst \t Waiting \t Turn around \n");
    for (i = 0; i < n; i++) {
        total_wt += wait_time[i];
        total_tat += tat[i];
        printf(" %d\t %d\t\t %d\t\t %d \t %d\n", i + 1, at[i], burst_time[i], wait_time[i], tat[i]);
    }
    printf("Average waiting time = %f\n", (float)total_wt / (float)n);
    printf("Average turn around time = %f\n", (float)total_tat / (float)n);
    return 0;
}
```

```
int main() {
    int proc[50], burst_time[50], at[50];
    int n, i;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the arrival time of the processes: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &at[i]);
    }
    printf("Enter the burst time of the processes: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &burst_time[i]);
    }
    for (i = 0; i < n; i++) {
        proc[i] = i + 1;
    }
    avgtime(proc, n, burst_time, at);
    return 0;
}
```

## OUTPUT:

```
(anand@kali) - [~/Desktop]
$ ./a.out
Enter the number of processes: 5
Enter the arrival time of the processes: 1 2 3 4 5
Enter the burst time of the processes: 3 7 4 4 5
Processes  Arrival time  Burst   Waiting  Turn around
1          1             3         0         3
2          2             7         2         9
3          3             4         8        12
4          4             4        11        15
5          5             5        14        19
Average waiting time = 7.000000
Average turn around time = 11.600000
```

## CONCLUSION:

By executing the above program, we have successfully demonstrated the FCFS CPU Scheduling

## 2. SHORTEST JOB FIRST SCHEDULING

AIM: To demonstrate Shortest Job First CPU Scheduling

### DESCRIPTION:

SJF scheduling algorithm, schedules the process according to their burst time. In SJF scheduling, the process with the lowest burst time, among the list of available processes in the ready queue, is going to be scheduled next

### PROGRAM:

```
#include <stdio.h>
int main() {
    int arrival_time[10], burst_time[10], temp[10];
    int i, smallest, count = 0, time, limit;
    int wt, tat;
    double wait_time = 0, turnaround_time = 0, end;
    float average_waiting_time, average_turnaround_time;
    printf("Enter the Number of Processes: ");
    scanf("%d", &limit);

    printf("Enter Arrival Time: ");
    for (i = 0; i < limit; i++) {
        scanf("%d", &arrival_time[i]);
    }
    printf("Enter Burst Time: ");
    for (i = 0; i < limit; i++) {
        scanf("%d", &burst_time[i]);
        temp[i] = burst_time[i];
    }
    burst_time[9] = 9999;
    printf("Processes Arrival time Burst Waiting Turn around \n");
    for (time = 0; count < limit; time++) {
        smallest = 9;
        for (i = 0; i < limit; i++) {
            if
            (arrival_time[i] <= time && burst_time[i]
             < burst_time[smallest] && burst_time[i] > 0)
            {
                smallest = i;
            }
        }
        burst_time[smallest]--;
    }
```

```
Code For if (burst_time[smallest] == 0) {  
    count++;  
    end = time + 1;  
    wt = end - arrival_time[smallest] - temp[smallest];  
    tat = end - arrival_time[smallest];  
    wait_time = wait_time + wt;  
    turnaround_time = turnaround_time + tat;  
    printf(" %d\t %d\t\t %d\t\t %d \t\t %d\n",  
        smallest+1, arrival_time[smallest], temp[smallest], wt, tat);  
    smallest++;  
}  
average_waiting_time = wait_time / limit;  
average_turnaround_time = turnaround_time / limit;  
printf("Average Waiting Time: %lf\n", average_waiting_time);  
printf("Average Turnaround Time: %lf\n", average_turnaround_time);  
return 0;  
}
```

## OUTPUT:

```
(anand@kali)-[~/Desktop]  
$ ./a.out  
Enter the Number of Processes: 4  
Enter Arrival Time: 1 2 3 4  
Enter Burst Time: 4 4 5 8  
Processes  Arrival time  Burst   Waiting  Turn around  
1          1           4        0         4  
2          2           4        3         7  
3          3           5        6        11  
4          4           8       10        18  
Average Waiting Time: 4.750000  
Average Turnaround Time: 10.000000
```

## CONCLUSION:

By executing the above program, we have successfully demonstrated the Shortest Job First CPU Scheduling.

### 3. PAGING MEMORY MANAGEMENT TECHNIQUE

AIM: To implement paging Memory Management Technique

#### DESCRIPTION:

Paging is the memory management technique in which secondary memory is divided into fixed-size blocks called pages, and main memory is divided into fixed-size blocks called frames. The Frame has the same size as that of a Page. The processes are initially in secondary memory, from where the processes are shifted to main memory (RAM) when there is a requirement. Each process is mainly divided into parts where the size of each part is the same as the page size. One page of a process is mainly stored in one of the memory frames. Paging follows no contiguous memory allocation. That means pages in the main memory can be stored at different locations in the memory.

#### PROGRAM:

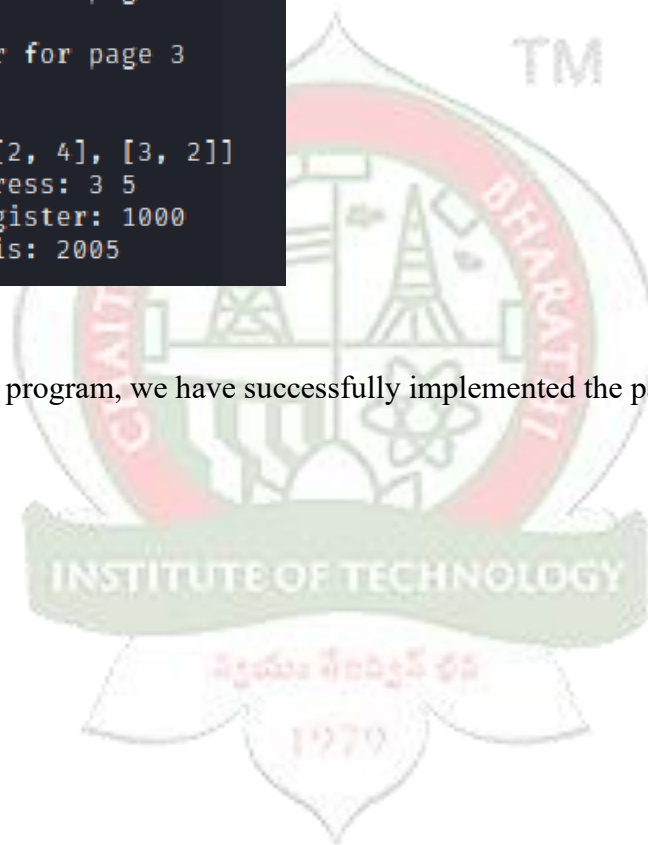
```
mem_size = int(input('Enter memory size: '))
page_size = int(input('Enter page size: '))
no_of_pages = mem_size // page_size
print('No of pages available in memory:', no_of_pages)
no_of_frames = int(input('Enter number of frames: '))
print('Total size of memory: ', page_size * no_of_frames)
page_table = []
for i in range(no_of_pages):
    print('Enter frame number for page', i)
    fno = int(input())
    page_table.append([i, fno])
print('Page Table:')
print(page_table)
inp_pno, inp_off = map(int, input('Enter logical address: ').split())
inp_fra = None
for i in page_table:
    if i[0] == inp_pno:
        inp_fra = i[1]
if inp_fra is None:
    print('Page not found')
else:
    base_reg = int(input('Enter the base register: '))
    phy_add = base_reg + inp_fra * page_size + inp_off
    print('Physical address is:', phy_add)
```

## OUTPUT:

```
(anand@kali)-[~/Desktop]
$ python3 paging.py
Enter memory size: 2000
Enter page size: 500
No of pages available in memory: 4
Enter number of frames: 5
Total size of memory: 2500
Enter frame number for page 0
1
Enter frame number for page 1
3
Enter frame number for page 2
4
Enter frame number for page 3
2
Page Table:
[[0, 1], [1, 3], [2, 4], [3, 2]]
Enter logical address: 3 5
Enter the base register: 1000
Physical address is: 2005
```

## CONCLUSION:

By executing the above program, we have successfully implemented the paging memory management technique.





#### 4. SEGMENTATION MEMORY MANAGEMENT TECHNIQUES

AIM: To implement Segmentation Memory Management Techniques

##### DESCRIPTION:

Segmentation divides the user program and the secondary memory into uneven-sized blocks known as Segments. Segmentation can be divided into two types namely - Virtual Memory Segmentation and Simple Segmentation.

A Segment Table is used to store the information of all segments of the currently executing process. The swapping of the segments of the process results in the breaking of the free memory space into small pieces. This breaking of free space into pieces is called External Fragmentation.

##### PROGRAM:

```
no_of_seg = int(input("Enter number of segments: "))
seg_table = []
for i in range(no_of_seg):
    print("Enter base and limit of segment", i)
    base, limit = map(int, input().split())
    seg_table.append([base, limit])
print("SEGMENT TABLE")
for i in seg_table:
    print(i)
print("Enter Logical Address [Segment number, Offset]")
inp_seg_no, offset = map(int, input().split())
if 0 ≤ inp_seg_no < no_of_seg and 0 ≤ offset < seg_table[inp_seg_no][1]:
    phy_add = seg_table[inp_seg_no][0] + offset
    print("The physical address of the given logical address is:", phy_add)
else:
    print("Invalid segment number or offset is greater than limit")
```

##### OUTPUT:

```
(anand@kali)-[~/Desktop]
$ python3 segmentation.py
Enter number of segments: 3
Enter base and limit of segment 0
1000 2000
Enter base and limit of segment 1
2000 3000
Enter base and limit of segment 2
3000 4000
SEGMENT TABLE
[1000, 2000]
[2000, 3000]
[3000, 4000]
Enter Logical Address [Segment number, Offset]
2 8
The physical address of the given logical address is: 3008
```

**CONCLUSION:**

By executing the above program, we have successfully implemented the Segmentation Memory Management Technique

**EXPERIMENT-4****IMPLEMENTATION OF FILE ALLOCATION METHODS****1. CONTIGUOUS FILE ALLOCATION**

AIM: To implement the Contiguous File Allocation Method

**DESCRIPTION:**

In Contiguous File Allocation Method, each file occupies a contiguous set of blocks on the disk. For example, if a file requires  $n$  blocks and is given a block  $b$  as the starting location, then the blocks assigned to the file will be:  $b, b+1, b+2, \dots, b+n-1$ . This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file.

**PROGRAM:**

```
#include <stdio.h>
int main() {
    int n, i, j, b[20], sb[20], t[20], x, c[20][20];
    printf("Enter no. of files: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        printf("Enter no. of blocks occupied by file-%d: ", i + 1);
        scanf("%d", &b[i]);
        printf("Enter the starting block of file-%d: ", i + 1);
        scanf("%d", &sb[i]);
        t[i] = sb[i];

        for (j = 0; j < b[i]; j++)
            c[i][j] = sb[i]++;
    }
    printf("Filename\tStart block\tlength\n");
    for (i = 0; i < n; i++)
        printf("%d\t\t\t %d \t\t%d\n", i + 1, t[i], b[i]);
    printf("Enter file name:");
    scanf("%d", &x);
    printf("File name is:%d\n", x);
    printf("Length is:%d\n", b[x - 1]);
    printf("Blocks occupied: ");
    for (i = 0; i < b[x - 1]; i++)
        printf("%4d", c[x - 1][i]);
    getch();
    return 0;
}
```

## OUTPUT:

```

(anand@kali)-[~/Desktop]
$ ./a.out
Enter no. of files: 2
Enter no. of blocks occupied by file-1: 4
Enter the starting block of file-1: 2
Enter no. of blocks occupied by file-2: 15
Enter the starting block of file-2: 4
Filename is the start block form length code:
1          2          4
2          4          15
Enter file name:2
File name is:2
Length is:15
Blocks occupied:  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18

```

## CONCLUSION:

By executing the above program, we have successfully implemented the Contiguous File Allocation Method

## 2. INDEXED FILE ALLOCATION

AIM: To implement the Indexed File Allocation Method

## DESCRIPTION:

In Indexed File Allocation Method, a special block known as the Index block contains the pointers to all the blocks occupied by a file. Each file has its own index block. The  $i$ th entry in the index block contains the disk address of the  $i$ th file block.

## PROGRAM:

```

#include <stdio.h>
int main() {
    int n, m[20], i, j, sb[20], s[20], b[20][20], x;
    printf("Enter no. of files: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        printf("Enter starting block and size of file-%d: ", i + 1);
        scanf("%d%d", &sb[i], &s[i]);
        printf("Enter blocks occupied by file-%d: ", i + 1);
        scanf("%d", &m[i]);
        printf("Enter blocks of file-%d: ", i + 1);
        for (j = 0; j < m[i]; j++)
            scanf("%d", &b[i][j]);
    }
}

```

```
printf("\nFile\t index\tlength\n");
for (i = 0; i < n; i++) {
    printf("%d\t%d\t%d\n", i + 1, sb[i], m[i]);
}
printf("\nEnter file name: ");
scanf("%d", &x);
printf("File name is:%d\n", x);
i = x - 1;
printf("Index is:%d\n", sb[i]);
printf("Block occupied are:");
for (j = 0; j < m[i]; j++)
    printf("%3d", b[i][j]);
getchar();
return 0;
}
```

## OUTPUT:

```
(anand@kali)-[~/Desktop]
$ ./a.out
Enter no. of files: 2
Enter starting block and size of file-1: 2 5
Enter blocks occupied by file-1: 10
Enter blocks of file-1: 3 2 5 4 6 7 2 6 4 7
Enter starting block and size of file-2: 3 4
Enter blocks occupied by file-2: 5
Enter blocks of file-2: 3 4 5 6 7

File      index  length
1         2      10
2         3       5

Enter file name: 2
File name is:2
Index is:3
Block occupied are:  3  4  5  6  7
```

## CONCLUSION:

By executing the above program, we have successfully implemented the Indexed File Allocation Method.

### 3. LINKED FILE ALLOCATION

AIM: To implement the Linked File Allocation Method

#### DESCRIPTION:

In Linked File Allocation Method, each file is a linked list of disk blocks which need not be contiguous. The disk blocks can be scattered anywhere on the disk. The directory entry contains a pointer to the starting and the ending file block. Each block contains a pointer to the next block occupied by the file.

#### PROGRAM:

```
#include <stdio.h>
struct file {
    char fname[10];
    int start, size, block[10];
} f[10];
int main() {
    int i, j, n;
    printf("Enter no. of files:");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        printf("Enter file name:");
        scanf("%s", &f[i].fname);
        printf("Enter starting block:");
        scanf("%d", &f[i].start);
        f[i].block[0] = f[i].start;
        printf("Enter no. of blocks:");
        scanf("%d", &f[i].size);
        printf("Enter block numbers:");
        for (j = 1; j <= f[i].size; j++) {
            scanf("%d", &f[i].block[j]);
        }
    }
    printf("File\tstart\tsize\tblock\n");
    for (i = 0; i < n; i++) {
        printf("%s\t%d\t%d\t", f[i].fname, f[i].start, f[i].size);
        for (j = 1; j <= f[i].size - 1; j++) {
            printf("%d→", f[i].block[j]);
        }
        printf("%d", f[i].block[j]);
        printf("\n");
    }
    getchar();
    return 0;
}
```

OUTPUT:

```
(anand@kali)-[~/Desktop]
$ ./a.out
Enter no. of files:2
Enter file name:OS
Enter starting block:10
Enter no. of blocks:4
Enter block numbers:6 7 9 0
Enter file name:os
Enter starting block:20
Enter no. of blocks:3
Enter block numbers:7 8 9
File      start    size    block
OS        10         4       6→7→9→0
os        20         3       7→8→9
```

CONCLUSION:

By executing the above program, we have successfully implemented the Linked File Allocation Method





**EXPERIMENT-7****SOCKET PROGRAMMING**

**AIM:** To implement echo server using Socket Programming

**DESCRIPTION:**

**Sockets:** Provide a standard interface between the network and application. Allow communication between processes on the same machine or different machines. A socket is a communication end point. It basically contains an IP address and Port number. Socket in Unix/Linux domain provide communication between the processes in the same machine.

**Types of sockets:** There are four types of sockets, they are

1. **Stream Sockets:** Provide reliable byte stream transport service. These sockets guarantee the delivery of packets and the order in a network environment. Uses TCP (Transmission Control Protocol). Data records do not have any boundaries.
2. **Datagram Sockets:** Delivery in a network environment is not guaranteed. Uses UDP (User Datagram Protocol)

**Client Process:** Typically, a process which makes a request for information. After getting the response, a client process may terminate or may do some other processing. Ex. Web browser

**Server Process:** Is a process which takes a request from the clients and serves it. After getting a request from the client, this process will perform the required processing, gather the requested information and send it to the client. Once done, it becomes ready to serve another client. Server processes are always alert and ready to serve incoming requests. Ex: HTTP server, SMTP server, DNS, mail server etc.

**Types of Servers:** Servers can be:

1. **Iterative Server:** Serves its clients one after other. They cannot serve clients simultaneously and may be implemented when the service time is finite and short time then they can be implemented as iterative servers. Ex: Time server, DNS etc.
2. **Concurrent Server:** Is a server that can serve multiple client requests simultaneously. Simplest way to create concurrent servers, we use `fork()` system call.

**Socket system calls:** In a client server environment, both the client and server have to create sockets for communication. The standard API for network programming in C is Berkeley Sockets. This API was first introduced in 4.3BSD UNIX and now available on all Unix-like platforms including Linux, MacOS X, Free BSD and Solaris. A very similar network API is available on Windows.

**Server side:** `socket()`, `bind()`, `listen()`, `accept()`, `close()`

**Client side:** `socket()`, `connect()` and `close()`

**Both sides:** `recv()/read()`, `send()/write()`

**PROGRAM:**

client.py



```
import socket
# Create a socket object
client_socket = socket.socket()

# Connect to the server
client_socket.connect(('127.0.0.1', 9000))

# Receive data from the server
data = client_socket.recv(1024)
print(f"Received from server: {data.decode()}")

# Send data to the server
client_socket.send(b"Hello, server! This is the client.")

# Close the socket
client_socket.close()
```

Server.py

```
import socket

# Create a socket object
server_socket = socket.socket()

# Bind the socket to a specific address and port
server_socket.bind(('127.0.0.1', 9000))

# Listen for incoming connections
server_socket.listen(5)
print("Server is listening for connections ...")

try:
    # Your code or process that you want to run
    while True:
        # Accept a connection from a client
        client_socket, client_address = server_socket.accept()
        print(f"Connection from {client_address}")

        # Send data to the client
        client_socket.send(b"Hello, client! This is the server.")

        # Receive data from the client
        data = client_socket.recv(1024)
        print(f"Received from client: {data.decode()}")

except KeyboardInterrupt:
    print("\nKeyboard interrupt received. Ending the process.")

finally:
    client_socket.close()
    server_socket.close()
```

## OUTPUT:

```
(anand@kali)-[~/Desktop]
$ python3 server.py
Server is listening for connections ...
Connection from ('127.0.0.1', 60012)
Received from client: Hello, server! This is the client.
```

```
(anand@kali)-[~/Desktop]
$ python3 client.py
Received from server: Hello, client! This is the server.
```

## CONCLUSION:

By executing the above program, we have successfully implemented the echo programming.



**EXPERIMENT-8****BANKER'S (SAFETY) ALGORITHM**

AIM: To execute the banker's algorithm and find whether a safe sequence exists or not.

**DESCRIPTION:**

Banker's algorithm is used for resources which are of multiple instance type. It generally consists of some data structures like:

1. Available: a vector of length m which keeps a track of the available resources
2. Max: a  $n \times m$  matrix which keeps a track of maximum allocated resources for a process
3. Allocation: a  $n \times m$  matrix which keeps a track of the currently allocated resources to a particular process
4. Need: a  $n \times m$  matrix which keeps a track of additionally required resources for a particular process apart from the allocated resources.

If the Banker's algorithm generates a safe sequence, then there exists no deadlock else there exists a deadlock.

**PROGRAM:**

```
n = int(input("Enter the number of processes: "))
m = int(input("Enter the number of resources: "))
print("Enter the Allocated matrix: ")
alloc = []
for i in range(n):
    l = list(map(int, input().strip().split()))
    alloc.append(l)
print("Enter the Max matrix: ")
max_matrix = []
for i in range(n):
    l = list(map(int, input().strip().split()))
    max_matrix.append(l)
print("Enter the Avail matrix: ")
avail = list(map(int, input().strip().split()))
print("Allocated Matrix:", alloc, end="\n")
print("Max Matrix:", max_matrix, end="\n")
print("Avail Matrix:", avail, end="\n")
f = [0] * n
ans = [0] * n
ind = 0
for k in range(n):
    f[k] = 0
need = [[0 for i in range(m)] for i in range(n)]
for i in range(n):
    for j in range(m):
        need[i][j] = max_matrix[i][j] - alloc[i][j]
```

```

for k in range(n):
    for i in range(n):
        if f[i] == 0:
            flag = 0
            for j in range(m):
                if need[i][j] > avail[j]:
                    flag = 1
                    break
            break
        if flag == 0:
            ans[ind] = i
            ind += 1
            for y in range(m):
                avail[y] += alloc[i][y]
            f[i] = 1
print("Safe Sequence:")
for i in range(n - 1):
    print(" P", ans[i], " → ", sep="", end="")
print(" P", ans[n - 1], sep="")

```

## OUTPUT:

```

(anand@kali) - [~/Desktop]
$ python3 banker.py
Enter the number of processes: 5
Enter the number of resources: 3
Enter the Allocated matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the Max matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter the Avail matrix:
3 3 2
Allocated Matrix: [[0, 1, 0], [2, 0, 0], [3, 0, 2], [2, 1, 1], [0, 0, 2]]
Max Matrix: [[7, 5, 3], [3, 2, 2], [9, 0, 2], [2, 2, 2], [4, 3, 3]]
Avail Matrix: [3, 3, 2]
Safe Sequence:
P1 → P3 → P4 → P0 → P2

```

## CONCLUSION:

By executing the program, we have successfully the banker's Algorithm and found whether a safe sequence exists or not.