

BASIC COMMANDS IN LINUX

AIM: To study and execute the commands in UNIX

A.

FILE RELATED COMMANDS:

1. Pwd Command:

DESCRIPTION: It displays the present working directory

SYNTAX: pwd

OUTPUT:

```
[bel9cl-12@csl ~]$ pwd
/home/user/bel9cl/bel9cl-12
```

2. Cd Command:

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

SYNTAX: cd [option] [directory]

OUTPUT:

```
[bel9cl-12@csl ~]$ cd demo
[bel9cl-12@csl demo]$
```

3. Ls Command:

DESCRIPTION: It lists the content/files listed in a particular UNIX directory

SYNTAX: ls [options] [filename]

OUTPUT:

```
[bel9cl-12@csl ~]$ ls
demo test1.txt
```

4. Rm command:

DESCRIPTION: It removes files or directories

SYNTAX: rm [option ...] filelist

OUTPUT:

Before using rm command:

```
[bel9cl-12@csl ~]$ ls
demo test1.txt
```

After using rm command:

```
[bel9cl-12@csl ~]$ rm test1.txt
[bel9cl-12@csl ~]$ ls
demo
```

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:

Renaming the file present in directory

```
[bel9cl-12@csl ~]$ mv test1.txt sample.txt
[bel9cl-12@csl ~]$ ls
demo  sample.txt
```

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:

Text file is created

```
[bel9cl-12@csl ~]$ cat > test1.txt
Hello
CBIT
^C
```

7. Cmp Command:

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

SYNTAX: cmp [option ...] fromfile tofile

OUTPUT:

```
[bel9cl-04@csl ~]$ cat > sample.txt
jddbhfduifji
^C
[bel9cl-04@csl ~]$ cat > test1.txt
kjgfujherhb
^C
[bel9cl-04@csl ~]$ cmp test1.txt sample.txt
test1.txt sample.txt differ: byte 1, line 1
[bel9cl-04@csl ~]$
```

8. Cp Command:**DESCRIPTION:** It copies the files**SYNTAX:** cp [option] source destination**OUTPUT:**

```
[bel9cl-12@cs1 ~]$ cp test1.txt copy.txt
[bel9cl-12@cs1 ~]$ ls
copy.txt  demo  sample1.txt  test1.txt
[bel9cl-12@cs1 ~]$ cat copy.txt
Hello
CBIT
[bel9cl-12@cs1 ~]$ cat test1.txt
Hello
CBIT
```

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:**

```
[bel9cl-12@cs1 ~]$ echo "Operating Systems"
Operating Systems
```

10. Mkdir Command:**DESCRIPTION:** It created new directory**SYNTAX:** mkdir [options] DirectoryNames**OUTPUT:**

```
[bel9cl-12@cs1 ~]$ mkdir demo
[bel9cl-12@cs1 ~]$ ls
demo  test1.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:**

```
[bel9cl-12@cs1 ~]$ cat test1.txt
Hello
CBIT
[bel9cl-12@cs1 ~]$ cat sample1.txt
Hello
World
[bel9cl-12@cs1 ~]$ paste test1.txt sample1.txt
Hello  Hello
CBIT   World
```

12. Rmdir Command:**DESCRIPTION:** It removes the directories**SYNTAX:** rmdir [options] DirectoryNames**OUTPUT:**

```
[bel9cl-12@cs1 ~]$ rmdir demo
[bel9cl-12@cs1 ~]$ ls
copy.txt  sample1.txt  test1.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:**

```
[bel9cl-12@cs1 ~]$ cat test1.txt
Hello
CBIT
12
13
32
12
31
45
65
121
432
21
21
[bel9cl-12@cs1 ~]$ cat sample1.txt
Hi
Dog
bad
act
[bel9cl-12@cs1 ~]$ head test1.txt sample1.txt
==> test1.txt <==
Hello
CBIT
12
13
32
12
31
45
65
121

==> sample1.txt <==
Hi
Dog
bad
act
```

14. Tail Command:

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

SYNTAX: tail [option] [files]

OUTPUT:

```
[bel9cl-12@cs1 ~]$ cat test1.txt
Hello
CBIT
12
13
32
12
31
45
65
121
432
21
21
[bel9cl-12@cs1 ~]$ cat sample1.txt
Hi
Dog
bad
act
[bel9cl-12@cs1 ~]$ tail test1.txt sample1.txt
==> test1.txt <==
13
32
12
31
45
65
121
432
21
21

==> sample1.txt <==
Hi
Dog
bad
act
```

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:

```
[bel9cl-12@cs1 ~]$ date
Tue Aug 23 14:53:46 IST 2022
```

16. Grep Command:**DESCRIPTION:** It prints lines matching a pattern**SYNTAX:** grep [options] PATTERN [file]**OUTPUT:**

```
[bel9cl-12@cs1 ~]$ cat test1.txt
Hello
CBIT
12
13
32
12
31
45
65
121
432
21
21
[bel9cl-12@cs1 ~]$ grep "1" test1.txt
12
13
12
31
121
21
21
```

17. Touch Command:**DESCRIPTION:** It creates a new file or update its timestamp**SYNTAX:** touch [option ...] [File ...]**OUTPUT:**

```
[bel9cl-12@cs1 ~]$ ls
copy.txt  sample1.txt  test1.txt
[bel9cl-12@cs1 ~]$ touch file1
[bel9cl-12@cs1 ~]$ touch file2
[bel9cl-12@cs1 ~]$ ls
copy.txt  file1  file2  sample1.txt  test1.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

```
[bel9cl-12@cs1 ~]$ chmod 002 sample1.txt
```

19. Man Command:

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

SYNTAX: man ls

OUTPUT:

```
LS(1)                                User Commands                                LS(1)

NAME
  ls - list directory contents

SYNOPSIS
  ls [OPTION]... [FILE]...

DESCRIPTION
  List information about the FILES (the current directory by default).  Sort entries alphabetically if none of -ftuvSUX nor --sort.
  Mandatory arguments to long options are mandatory for short options too.

  -a, --all
      do not ignore entries starting with .

  -A, --almost-all
      do not list implied . and ..

  --author
      with -l, print the author of each file

  -b, --escape
      print octal escapes for nongraphic characters

  --block-size=SIZE
      use SIZE-byte blocks.  See SIZE format below

  -B, --ignore-backups
      do not list implied entries ending with ~

  -c      with -lt: sort by, and show, ctime (time of last modification of file status information) with -l: show ctime and sort by name otherwise: sort by ctime

  -C      list entries by columns

  --color[=WHEN]
      colorize the output.  WHEN defaults to 'always' or can be 'never' or 'auto'.  More info below

  -d, --directory
      list directory entries instead of contents, and do not dereference symbolic links

  -D, --dired
      generate output designed for Emacs' dired mode

  -f      do not sort, enable -aU, disable -ls --color

  -F, --classify
      append indicator (one of */>=) to entries

  --file-type
      likewise, except do not append '='
```

20. Clear Command:

DESCRIPTION: It clears the terminal Screen

SYNTAX: clear

OUTPUT:

After entering the clear command entire terminal will be cleared

```
[bel9cl-12@cs1 ~]$
```

PROCESS RELATED COMMANDS:**1. Top Command:****DESCRIPTION:** To track the running processes on our machine**SYNTAX:** top**OUTPUT:**

```
top - 15:27:56 up 22:53, 19 users, load average: 0.00, 0.00, 0.00
Tasks: 338 total, 1 running, 336 sleeping, 1 stopped, 0 zombie
Cpu(s): 0.0%us, 0.1%sy, 0.0%ni, 99.9%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 8061868k total, 2435568k used, 5626300k free, 59544k buffers
Swap: 8208376k total, 0k used, 8208376k free, 1766300k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1163	root	20	0	0	0	0	S	0.3	0.0	0:00.10	jbd2/dm-2-8
1	root	20	0	19356	1536	1228	S	0.0	0.0	0:03.44	init
2	root	20	0	0	0	0	S	0.0	0.0	0:00.03	kthreadd
3	root	RT	0	0	0	0	S	0.0	0.0	0:00.23	migration/0
4	root	20	0	0	0	0	S	0.0	0.0	0:00.01	ksoftirqd/0
5	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	migration/0
6	root	RT	0	0	0	0	S	0.0	0.0	0:00.11	watchdog/0
7	root	RT	0	0	0	0	S	0.0	0.0	0:00.23	migration/1
8	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	migration/1
9	root	20	0	0	0	0	S	0.0	0.0	0:00.09	ksoftirqd/1
10	root	RT	0	0	0	0	S	0.0	0.0	0:00.10	watchdog/1
11	root	RT	0	0	0	0	S	0.0	0.0	0:00.20	migration/2
12	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	migration/2
13	root	20	0	0	0	0	S	0.0	0.0	0:00.10	ksoftirqd/2
14	root	RT	0	0	0	0	S	0.0	0.0	0:00.11	watchdog/2
15	root	RT	0	0	0	0	S	0.0	0.0	0:00.24	migration/3
16	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	migration/3
17	root	20	0	0	0	0	S	0.0	0.0	0:00.02	ksoftirqd/3
18	root	RT	0	0	0	0	S	0.0	0.0	0:00.10	watchdog/3
19	root	RT	0	0	0	0	S	0.0	0.0	0:00.21	migration/4
20	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	migration/4
21	root	20	0	0	0	0	S	0.0	0.0	0:00.02	ksoftirqd/4
22	root	RT	0	0	0	0	S	0.0	0.0	0:00.10	watchdog/4
23	root	RT	0	0	0	0	S	0.0	0.0	0:00.22	migration/5
24	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	migration/5
25	root	20	0	0	0	0	S	0.0	0.0	0:00.02	ksoftirqd/5
26	root	RT	0	0	0	0	S	0.0	0.0	0:00.10	watchdog/5
27	root	RT	0	0	0	0	S	0.0	0.0	0:00.19	migration/6
28	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	migration/6
29	root	20	0	0	0	0	S	0.0	0.0	0:00.01	ksoftirqd/6
30	root	RT	0	0	0	0	S	0.0	0.0	0:00.10	watchdog/6
31	root	RT	0	0	0	0	S	0.0	0.0	0:00.25	migration/7
32	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	migration/7
33	root	20	0	0	0	0	S	0.0	0.0	0:00.01	ksoftirqd/7
34	root	RT	0	0	0	0	S	0.0	0.0	0:00.10	watchdog/7
35	root	20	0	0	0	0	S	0.0	0.0	0:02.63	events/0
36	root	20	0	0	0	0	S	0.0	0.0	0:02.22	events/1
37	root	20	0	0	0	0	S	0.0	0.0	0:02.39	events/2
38	root	20	0	0	0	0	S	0.0	0.0	0:02.22	events/3
39	root	20	0	0	0	0	S	0.0	0.0	0:02.00	events/4
40	root	20	0	0	0	0	S	0.0	0.0	0:02.34	events/5
41	root	20	0	0	0	0	S	0.0	0.0	0:01.59	events/6
42	root	20	0	0	0	0	S	0.0	0.0	0:02.82	events/7
43	root	20	0	0	0	0	S	0.0	0.0	0:00.00	cgroup
44	root	20	0	0	0	0	S	0.0	0.0	0:00.01	khelper

2. Ps Command:

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

SYNTAX: ps

OUTPUT:

```
[bel9cl-12@csl ~]$ ps
  PID TTY          TIME CMD
 11974 pts/13    00:00:00 bash
 13351 pts/13    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

```
[bel9cl-12@csl ~]$ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGBUS     8) SIGFPE     9) SIGKILL    10) SIGUSR1
11) SIGSEGV    12) SIGUSR2   13) SIGPIPE   14) SIGALRM    15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD  18) SIGCONT   19) SIGSTOP   20) SIGSTP
21) SIGTTIN    22) SIGTTOU   23) SIGURG    24) SIGXCPU   25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF  28) SIGWINCH  29) SIGIO     30) SIGPWR
31) SIGSYS     34) SIGRTMIN  35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

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

OUTPUT:

```
[bel9cl-12@csl ~]$ renice +1 987 -u daemon root -p 32
renice: 987: getpriority: No such process
renice: 2: getpriority: No such process
0: old priority 0, new priority 1
renice: 32: setpriority: Operation not permitted
```

NETWORK RELATED COMMANDS:

1. Ping Command:

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

SYNTAX: ping [host]

OUTPUT:

```
[bel9c1-12@csl ~]$ ping -c 5 172.20.0.9
PING 172.20.0.9 (172.20.0.9) 56(84) bytes of data.
64 bytes from 172.20.0.9: icmp_seq=1 ttl=64 time=0.030 ms
64 bytes from 172.20.0.9: icmp_seq=2 ttl=64 time=0.024 ms
64 bytes from 172.20.0.9: icmp_seq=3 ttl=64 time=0.022 ms
64 bytes from 172.20.0.9: icmp_seq=4 ttl=64 time=0.021 ms
64 bytes from 172.20.0.9: icmp_seq=5 ttl=64 time=0.020 ms

--- 172.20.0.9 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 3999ms
rtt min/avg/max/mdev = 0.020/0.023/0.030/0.005 ms
```

2. Ifconfig Command:

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

SYNTAX: ifconfig

OUTPUT:

```
[bel9c1-12@csl ~]$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:50:56:BC:58:43
          inet addr:172.20.0.9  Bcast:172.20.15.255  Mask:255.255.240.0
          inet6 addr: fe80::250:56ff:febc:5843/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8513440 errors:0 dropped:0 overruns:0 frame:0
          TX packets:28820 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:550330097 (524.8 MiB)  TX bytes:6692512 (6.3 MiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:1038 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1038 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:84560 (82.5 KiB)  TX bytes:84560 (82.5 KiB)
```

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:

```
[bel9c1-12@csl ~]$ netstat -tnl
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:56750            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:111              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22               0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631            0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:25             0.0.0.0:*               LISTEN
tcp        0      0 :::111                   :::*                     LISTEN
tcp        0      0 :::22                    :::*                     LISTEN
tcp        0      0 :::1:631                 :::*                     LISTEN
tcp        0      0 :::1:25                   :::*                     LISTEN
tcp        0      0 :::49125                  :::*                     LISTEN
```

4. Nslookup Command:

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

SYNTAX: nslookup host

OUTPUT:

```
cse1ab11@cbit:~$ nslookup www.google.com
Server:         127.0.0.53
Address:        127.0.0.53#53

Non-authoritative answer:
Name:   www.google.com
Address: 142.250.183.36
Name:   www.google.com
Address: 2404:6800:4009:821::2004
```

5. Telnet Command:

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

SYNTAX: telnet hostname/IP address

OUTPUT:

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Ubuntu 18.04.2 LTS
javatpoint-Inspiron-3542 login: javatpoint
Password:
Last login: Sun Mar 29 22:39:18 IST 2020 from localhost on pts/1
Welcome to Ubuntu 18.04.2 LTS (GNU/Linux 5.3.0-40-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

 * Canonical Livepatch is available for installation.
   - Reduce system reboots and improve kernel security. Activate at:
     https://ubuntu.com/livepatch

304 packages can be updated.
29 updates are security updates.

Your Hardware Enablement Stack (HWE) is supported until April 2023.
```

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:

```
traceroute to javatpoint.com (194.169.80.121), 64 hops max
 1  192.168.1.1  2.955ms  108.083ms  2.823ms
 2  * * *
 3  10.72.222.105  82.520ms  10.72.222.117  40.085ms  10.72.222.93  39.574ms
 4  192.168.28.196  26.381ms  40.080ms  192.168.28.194  39.293ms
 5  192.168.28.197  53.927ms  42.535ms  43.258ms
 6  172.16.8.70  56.835ms  56.874ms  40.893ms
 7  * * *
 8  * * *
 9  * * *
10  * * *
11  * * *
12  * * *
13  * * *
14  103.198.140.54  309.762ms  319.848ms  328.998ms
15  103.198.140.43  309.751ms  197.310ms  188.902ms
16  195.66.225.162  203.018ms  370.983ms  319.810ms
17  78.110.166.98  319.668ms  320.027ms  321.521ms
```

CONCLUSION:

By executing the above commands, we have understood the commands.

To Study about UNIX vi Editor and its features:

- The default editor that comes with the UNIX operating system is vi editor (visual editor).
- Using vi editor, we can edit an existing file or create a new file from scratch.
- We can also use this editor to just read a text file
- Syntax: vi filename

Modes of Operation in vi editor:

- There are three modes of operation in vi:
 - **Command Mode:**
 - i. When vi starts up, it is in Command Mode.
 - ii. This mode is where vi interprets any characters we type as commands and thus does not display them in the window.
 - iii. This mode allows us to move through a file, and to delete, copy, or paste a piece of text.
 - iv. To enter into Command Mode from any other mode, it requires pressing the [Esc] key.
 - v. If we press [Esc] when we are already in Command Mode, then vi will beep or flash the screen
 - **Insert Mode:**
 - i. This mode enables you to insert text into the file.
 - ii. Everything that's typed in this mode is interpreted as input and finally, it is put in the file.
 - iii. The vi always starts in command mode.
 - iv. To enter text, you must be in insert mode. To come in insert mode you simply type i.
 - v. To get out of insert mode, press the Esc key, which will put you back into command mode.
 - **Last Line Mode (Escape Mode):**
 - i. Line Mode is invoked by typing a colon [:], while vi is in Command Mode.
 - ii. The cursor will jump to the last line of the screen and vi will wait for a command.
 - iii. This mode enables you to perform tasks such as saving files, executing commands.

Vi Commands:

- **To switch from command to insert mode:**
 - i – Start typing before the current character
 - I – Start typing at the start of current line
 - a – Start typing after the current character
 - A – Start typing at the end of current line
 - o – Start typing on a new line after the current line
 - O – Start typing on a new line before the current line
- **To move around a file:**
 - j – To move down
 - k – To move up
 - h – To move left
 - l – To move right
- **To delete:**
 - x – Delete the current character
 - X – Delete the character before the cursor
 - r – Replace the current character
 - xp – Switch two characters

- dd – Delete the current line
- D – Delete the current line from current character to the end of the line
- dG – Delete from the current line to the end of the file
- **To repeat or undo:**
 - u – Undo the last command
 - . – repeat the last command
- **To save and quit:**
 - :wq – Save and quit
 - :w – Save
 - :q – Quit
 - :w fname – Save as fname
 - ZZ – Save and quit
 - :q! – Quit discarding changes made
 - :w! – Save (and write to non-writable file)
- **Command to cut, copy and paste:**
 - dd – delete a line
 - yy – (yank yank) copy a line
 - p – Paste after the current line
 - P – Paste before the current line
 - <n>dd – Delete the specified n number of lines
 - <n>yy – Copy the specified n number of lines
- **Start and end of line:**
 - 0 – Bring at the start of the current line
 - ^ - Bring at the start of the current line
 - \$ - Bring at the end of the current line
 - d0 – Delete till start of a line
 - d\$ - Delete till end of a line
- **Joining lines:**
 - J – Join two lines
 - yyp – Repeat the current line
 - ddp – Swap two lines
- **Move forward or backward:**
 - w – Move one word forward
 - b – Move one word backward
 - <n>w – Move specified number of words forward
 - dw – Delete one word
 - yw – Copy one word
 - <n>dw – Delete specified number of words

SHELL SCRIPTING

SHELL:

- Shell is a user program or its environment provided for user interaction.
- It is a command language interpreter that executes commands.
- It reads from the input from the user and executes programs based on the input
- It displays the program output when the program finishes the execution
- Shell is not part of system kernel, but uses the system kernel to execute programs, create files, etc.

SHELL SCRIPT:

- A shell script is a list of commands in a computer program that is run by the UNIX shell which is a command line interpreter.
- It usually has comments that describes the steps
- Operations performed by shell scripts are program execution, File manipulation and text printing

VARIABLES:

- The shell enables us to create, assign and delete variables.
- Variable names must contain only letters, numbers or the underscore
- Defining Variables:
Variable_name = variable_value
- Variable can be accessed by using a dollar sign (\$) as a prefix for variable_name
- After a variable is marked read-only, its value cannot be changed
- Unsetting a variable directs the shell to remove the variable from the list of the variables.
- Unset command cannot be used on the variables which are marked read-only

READ STATEMENT:

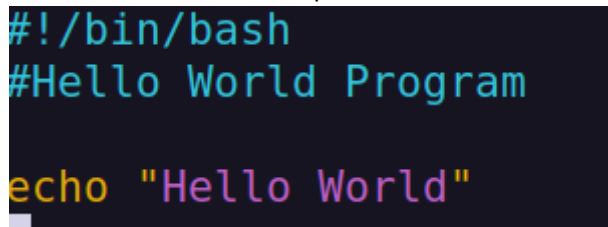
- The Read statement is used to get input from user and store the data to variable
- Syntax: read variable1, variable2, ..., variable

ECHO STATEMENT:

- The Echo statement is used to display text or value of variables on the screen.
- Syntax: echo [options] [string, variables...]

BASIC SHELL SCRIPT:

- touch HelloWorld.sh ---> creates a file
- nano HelloWorld.sh ----> opens the file in text editor



```
#!/bin/bash
#Hello World Program
echo "Hello World"
```

- chmod +x HelloWorld.sh ----> gives permission to execute the script
- ./HelloWorld.sh runs the script

Output:

```
cselab8@cselab8-OptiPlex-3060:~$ bash hello_world.sh
Hello World
```

ARITHMETIC OPERATIONS:

- Demonstration for basic Arithmetic operations:

Code:

```
#!/bin/sh
echo "Enter first number: "
read a
echo "Enter second number: "
read b
sum=$(( $a + $b ))
echo "Sum = $sum"
diff=$(( $a - $b ))
echo "Difference = $diff"
mul=$(( $a * $b ))
echo "Product = $mul"
div=$(( $a / $b ))
echo "Quotient when divided = $div"
rem=$(( $a % $b ))
echo "Remainder when divided = $rem"
```

Output:

```
cselab8@cselab8-OptiPlex-3060:~$ ./arithmetic.sh
Enter first number:
3
Enter second number:
2
Sum = 5
Difference = 1
Product = 6
Quotient when divided = 1
Remainder when divided = 1
```

DECISION MAKING STATEMENTS

- There are two types of decision-making statements within shell scripting. They are:
 - If-else statement
 - Case-esac statement

IF-ELSE STATEMENT:

- It is a conditional statement.
- There are couple of varieties present within the if-else statement. They are:
 - If-fi
 - If-else-fi
 - If-elif-else-fi
 - Nested if-else
- **Syntax:**
 - **If-fi**
if [expression]; then
statements
fi
 - **If-else-fi**
if [expression]; then
statement1
else
statement2
fi
 - **If-elif-else-fi**
if [expression]; then
statement1
elif [expression]
then
statement2
else
statement3
fi
 - **Nested if-else**
if [expression]
then
statement1
if [expression]
then
statement
else
statement
fi
else
statement
fi

- **Demo Program for If-else Statement:**

```
echo "Enter first number: "
read a
echo "Enter second number: "
read b
echo "Enter third number: "
read c
if [ $a -gt $b ]
then
if [ $a -gt $c ]
then
    echo "$a is greater"
else
    echo "$c is greater"
fi
else
if [ $b -gt $c ]
then
    echo "$b is greater"
else
    echo "$c is greater"
fi
fi
```

Output:

```
cselab11@cbit:~$ ./ifsample.sh
Enter first number:
3
Enter Second number:
4
Enter Third number:
5
5 is greater
```

THE CASE-ESAC STATEMENT:

- Case-esac is basically working the same as switch statement in programming.

- **Syntax:**

case \$var in

Pattern 1) Statement 1;;

Pattern n) Statement n;;

esac

- **Demo for case-esac statement:**

```
#!/bin/sh
echo "Enter a fruit name: "
read fruit
case "$fruit" in
    "Apple") echo "Apple a day keeps a doctor away"
    ;;
    "Mango") echo "Mango is a national fruit"
    ;;
    "Kiwi") echo "New Zealand is famous for kiwi"
    ;;
esac
```

Output:

```
cselab8@cselab8-OptiPlex-3060:~$ ./casedemo.sh
Enter a fruit name:
Apple
Apple a day keeps a doctor away
```

LOOPS:

- There are 3 looping statements which can be used in bash programming
 - While statement
 - For statement
 - Until statement

FOR STATEMENT:

- The for loop moves through a specified list of values until the list is exhausted.
- Syntax:
for varname in list
do
 statement
done

WHILE STATEMENT:

- Linux scripting while loop is similar to C language while loop
- There is a condition in while and commands are executed till the condition is valid. Once condition becomes false, loop terminates.
- Syntax:
while [condition]
do
 statement
done

UNTIL STATEMENT:

- The only difference is that until statement executes its code block while its conditional expression is false, and while statement executes its code block while its conditional expression is true
- Until loop always executes at least once.
- Loop while executes till it returns a zero value and until loop executes till it returns non-zero value.
- Syntax:
until [condition]
do
 commands
done

➤ **Demo program for all the loops:**

```
1#!/bin/sh
2#for loop
3echo "Demonstration of for loop"
4for var in 0 1 2 3 4 5 6 7 8 9
5do
6    echo $var
7done
8
9#While loop
10echo "Demonstration of while loop"
11a=0
12while [ $a -lt 5 ]
13do
14    echo $a
15    a=$(( $a + 1 ))
16done
17
18#until loop
19echo "Demonstration of Until Loop"
20a=0
21until [ $a -gt 7 ]
22do
23    echo $a
24    a=$(( $a + 1 ))
25done
26
```

Output:

```
cselab11@cbit:~$ ./loops_demo.sh
Demonstration of for loop
0
1
2
3
4
5
6
7
8
9
Demonstration of while loop
0
1
2
3
4
Demonstration of Until Loop
0
1
2
3
4
5
6
7
```

- To display Shell: echo \$0
- To find version of the shell: echo "\${BASH_VERSION}"
- To find path: echo \$PATH

```
cselab11@cbit:~$ echo $0
/bin/bash
cselab11@cbit:~$ echo "${BASH_VERSION}"
5.1.16(1)-release
cselab11@cbit:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

PROGRAMS:**Program-1: Write the Shell script to find Factorial of a number.**

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

ALGORITHM:

Step-1: Start

Step-2: Read the number

Step-3: a->1

Step-4: product->1

Step-5: While a<=n

 product->product*a

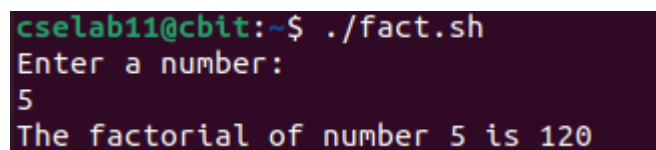
 a->a+1

Step-6: Print product

Step-7: End

PROGRAM:

```
#!/bin/sh
echo "Enter a number: "
read n
a=1
product=1
while [ $a -le $n ]
do
    product=$(( $product * $a ))
    a=$(( $a + 1 ))
done
echo "The factorial of number $n is $product"
```

OUTPUT:

```
cselab11@cbit:~$ ./fact.sh
Enter a number:
5
The factorial of number 5 is 120
```

CONCLUSION:

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

Program-2: Write the Shell Script to check if the given number is even or odd.

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

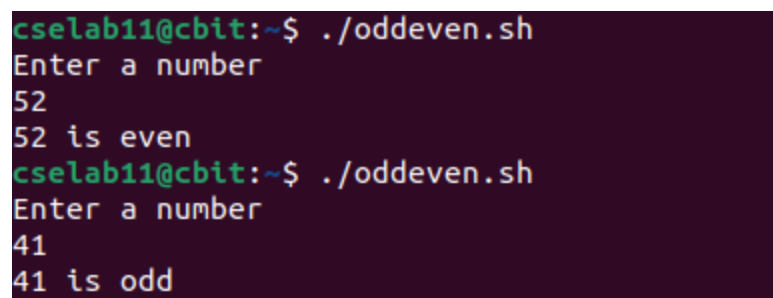
DESCRIPTION: In this program, we try to read a number from the user and check whether the given number is even or odd.

ALGORITHM:

Step-1: Start
Step-2: Read a number
Step-3: rem→n%2
Step-4: if rem=0
 Print("Even")
Step-5: else
 Print("Odd")
Step-6: End

PROGRAM:

```
#!/bin/sh
echo "Enter a number: "
read n
rem=$(( $n % 2 ))
if [ $rem -eq 0 ]
then
    echo "$n is even"
else
    echo "$n is odd"
fi
```

OUTPUT:

```
cselab11@cbit:~$ ./oddeven.sh
Enter a number
52
52 is even
cselab11@cbit:~$ ./oddeven.sh
Enter a number
41
41 is odd
```

CONCLUSION:

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

Program-3: Write a shell script to find power of a number.

AIM: To find power of a number.

DESCRIPTION: In this program, we try to read the base and power of a exponent and find the value of that exponent.

ALGORITHM:

Step-1: Start

Step-2: Read base and power as a, b

Step-3: i->1

Step-4: res->1

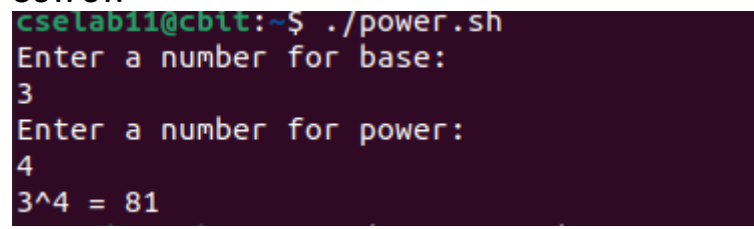
Step-5: while i<=b
 res->res*a
 i->i+1

Step-6: Print res

Step-7: End

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:

```
cselab11@cbit:~$ ./power.sh
Enter a number for base:
3
Enter a number for power:
4
3^4 = 81
```

CONCLUSION:

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

Program-4: Write a Shell script to check whether the given number is palindrome or not.

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.

ALGORITHM:

Step-1: Start

Step-2: Read a number n

Step-3: temp->n

Step-4: re->0

Step-5: while n!=0

 r=n%10

 re=re*10+r

 n=n/10

Step-6: if re=temp

 Print("Palindrome")

Step-7: else

 Print("Not a Palindrome")

Step-8: End

PROGRAM:

```
#!/bin/sh
```

```
echo "Enter a number: "
```

```
read n
```

```
temp=$n
```

```
re=0
```

```
while [ $n != 0 ]
```

```
do
```

```
    r=$(( $n % 10 ))
```

```
    re=$(( $re * 10 + $r ))
```

```
    n=$(( $n / 10 ))
```

```
done
```

```
if [ $re -eq $temp ]
```

```
then
```

```
    echo "The given number is a Palindrome"
```

```
else
```

```
    echo "The given number is not a Palindrome"
```

```
fi
```

OUTPUT:

```
Enter a number:
12321
The given number is a Palindrome
```

CONCLUSION:

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

Program-5: Write a Shell Script to print the Fibonacci Series.

AIM: To print the Fibonacci Series.

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

ALGORITHM:

Step-1: Start

Step-2: Read a number n

Step-3: a->0

Step-4: b->1

Step-5: Print a, b

Step-6: i->3

Step-7: while i<=n

 c=a+b

 print c

 a->b

 b->c

 i=i+1

Step-8: End

PROGRAM:

```
#!/bin/sh
```

```
echo "Enter a number: "
```

```
read n
```

```
a=0
```

```
b=1
```

```
echo "Fibonacci Series: "
```

```
echo $a
```

```
echo $b
```

```
i=3
```

```
while [ $i -le $n ]
```

```
do
```

```
    c=$(( $a + $b ))
```

```
    echo $c
```

```
    a=$b
```

```
    b=$c
```

```
    i=$(( $i + 1 ))
```

```
done
```

OUTPUT:

```
Enter a number:
10
Fibonacci Series:
0
1
1
2
3
5
8
13
21
34
```

CONCLUSION:

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

Program-6: Write a Shell Script to find whether a given number is prime or not.

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

ALGORITHM:

Step-1: Start

Step-2: Read a number n

Step-3: flag->0

Step-4: if n<=1

 flag->1

Step-5: else

 i->2

 a->n/2

 while i<=a

 rem=n%i

 if rem=0

 flag=1

 i=i+1

Step-6: if flag==0

 Print "Prime"

Step-7: else

 Print "not a prime"

Step-8: End

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
```

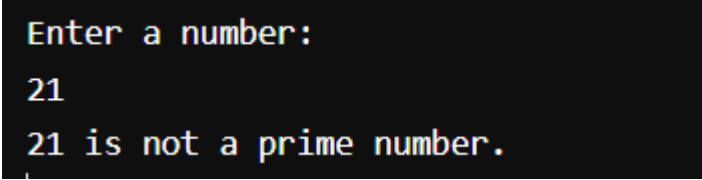
```
        i=$(( $i + 1 ))
```

```
    done
```

```
fi
```

```
if [ $flag -eq 0 ]
```

```
then
    echo "$n is a prime number"
else
    echo "$n is not a prime number."
fi
```

OUTPUT:

```
Enter a number:
21
21 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

Program-7: Write a shell script to check if the given number is an Armstrong number or not.

AIM: To find whether given number is an Armstrong number or not.

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

ALGORITHM:

Step-1: Start

Step-2: Read a number n

Step-3: res->0; num->0; temp->n

Step-4: while temp!=0
 num->num+1
 temp->temp/10

Step-5: temp->n

Step-6: while temp!=0
 r->temp%10
 re=1
 i=1
 while i<=num
 re->re*r
 i->i+1
 res->res+re
 temp->temp/10

Step-7: if res=n
 Print "Armstrong Number"

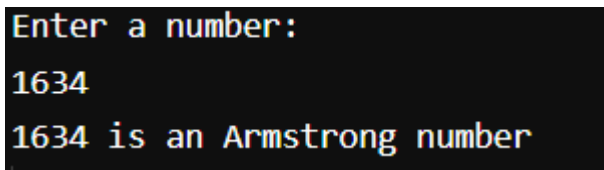
Step-8: else
 Print "not a Armstrong number"

Step-9: End

PROGRAM:

```
#!/bin/sh
echo "Enter a number: "
read n
res=0
num=0
temp=$n
while [ $temp -ne 0 ]
do
    num=$(( $num + 1 ))
    temp=$(( $temp / 10 ))
done
temp=$n
while [ $temp -ne 0 ]
do
    r=$(( $temp % 10 ))
    re=1
    i=1
    while [ $i -le $num ]
    do
```

```
    re=$(( $re * $r ))
    i=$(( $i + 1 ))
done
res=$(( $res + $re ))
temp=$(( $temp / 10 ))
done
if [ $res -eq $n ]
then
    echo "$n is an Armstrong number"
else
    echo "$n is not an Armstrong number"
fi
```

OUTPUT:

```
Enter a number:
1634
1634 is an Armstrong number
```

CONCLUSION:

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

Program-8: Write a Shell Script to demonstrate Mini Calculator.**AIM:** To demonstrate Mini Calculator**DESCRIPTION:** In this program, we try to perform all basic arithmetic operations by demonstrating a mini calculator.**ALGORITHM:**

Step-1: Start

Step-2: Read two numbers a, b

Step-3: Print "Operations to be Performed:

1. Add
2. Sub
3. Multiply
4. Division"

Step-4: Read Choice c

Step-5: while c<=4

case c in

"1" sum=a+b; Print sum

"2" diff=a-b; Print diff

"3" mul=a*b; Print mul

"4" div=a/b; Print div

Read c

Step-6: End

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"

;;

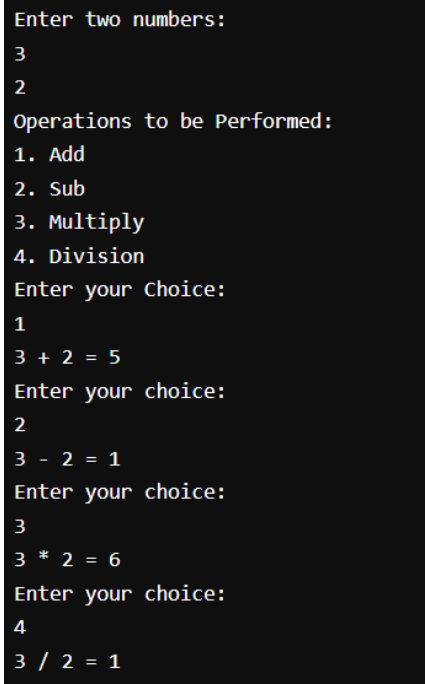
"3") mul=\$((\$a * \$b))

echo "\$a * \$b = \$mul"

;;

"4") div=\$((\$a / \$b))

```
        echo "$a / $b = $div"
    ;;
    esac
    echo "Enter your choice: "
    read c
done
```

OUTPUT:

```
Enter two numbers:
3
2
Operations to be Performed:
1. Add
2. Sub
3. Multiply
4. Division
Enter your Choice:
1
3 + 2 = 5
Enter your choice:
2
3 - 2 = 1
Enter your choice:
3
3 * 2 = 6
Enter your choice:
4
3 / 2 = 1
```

CONCLUSION:

By executing the above shell script, we have successfully demonstrated the Mini Calculator

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

1. **Fork():**

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.

HEADER FILES:

#include <sys/types.h>

#include <unistd.h>

SYNTAX:

pid_t fork(void);

RETURN VALUE: 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: This command is used to get process identification. This is often used by routines that generate unique temporary filenames. This function is always successful.

HEADER FILES:

#include <sys/types.h>

#include <unistd.h>

SYNTAX:

pid_t getpid(void);

RETURN VALUE: getpid() returns the process ID (PID) of the calling process.

3. **Getppid():**

DESCRIPTION: This command is used to get process identification. This function is always successful.

HEADER FILES:

#include <sys/types.h>

#include <unistd.h>

SYNTAX:

pid_t getppid(void);

RETURN VALUE: getppid() returns the process ID of the parent 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 the functions are used to execute a file. They replace the current process image with a new process image. Functions are execl, execlp, execl, execv, execvp, execvpe.

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[]);
```

RETURN VALUE: The exec() functions return only if an error has occurred. The return value is -1, and errno is set to indicate the error.

5. Wait():

DESCRIPTION: The wait() system call suspends execution of the calling thread until one of its children terminates.

HEADER FILES:

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

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

SYNTAX:

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

RETURN VALUE: On success, return the process ID of the terminated child; on error, -1 is returned.

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.

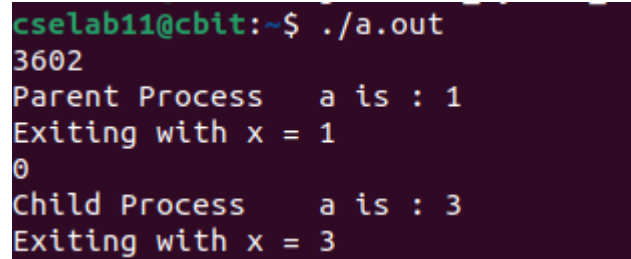
ALGORITHM:

Step-1: Start
Step-2: a->2
Step-3: pid_t pid
Step-4: pid->fork()
Step-5: Print pid
Step-6: if pid<0
 Print "Fork Failed"
Step-7: else if pid=0
 Print "Child Process"
 Print a
 Print ++a
Step-8: else
 Print "Parent Process"
 Print a
 Print --a
Step-9: Print a
Step-10: End

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);  
}
```

OUTPUT:

```
cse1ab11@cbit:~$ ./a.out  
3602  
Parent Process   a is : 1  
Exiting with x = 1  
0  
Child Process    a is : 3  
Exiting with x = 3
```

CONCLUSION:

Here, when `fork()` function is called, then a child process is created with variable `a = 2` and the existing process is called parent process with variable `a = 2`. The variables in child and parent process are independent of each other. Process ID displayed is the parent's process id and in that section a value has been decremented so the parent process exits with `a = 1`. Now, the Child's process ID is displayed and in that section a value is incremented so the child process exits with `a = 3`.

By 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.

ALGORITHM:

Step-1: Start

Step-2: a->2

Step-3: pid_t pid

Step-4: pid->fork()

Step-5: Print pid

Step-6: if pid<0

 Print "Error"

Step-7: else if pid=0

 Print "Child Process"

 Print ++a

 Print getpid()

 Print getppid()

Step-8: else

 Print "Parent Process"

 Print --a

 Print getpid()

 Print pid()

Step-9: Print a

Step-10: End

PROGRAM:

```
#include<stdio.h>
```

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

```
int main()
```

```
{
```

```
    int a=2;
```

```
    pid_t pid;
```

```
    pid=fork();
```

```
    printf("%d\n",pid);
```

```
    if(pid<0)
```

```
    {
```

```
        printf("Error");
```

```
    }
```

```
    else if(pid==0)
```

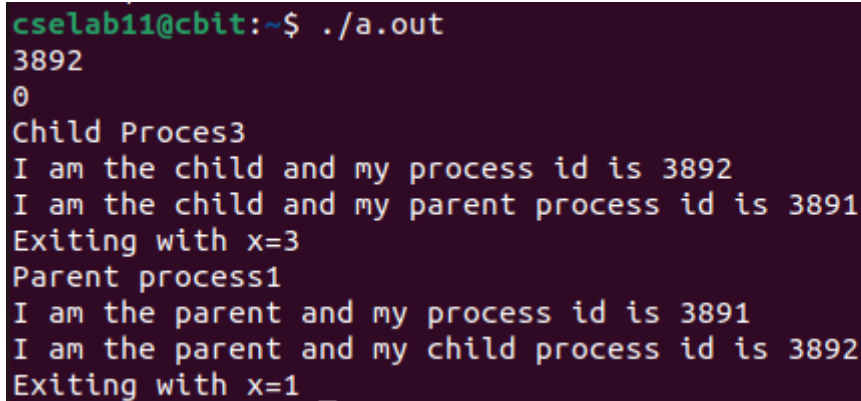
```
    {
```

```
        printf("child process");
```

```
        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();
        printf("parent process");
        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);
}
```

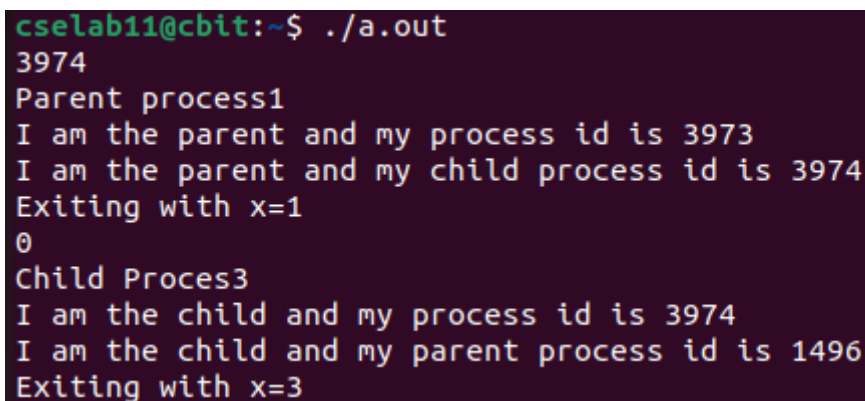
OUTPUT:

Using wait() system call:



```
cselab11@cbit:~$ ./a.out
3892
0
Child Proces3
I am the child and my process id is 3892
I am the child and my parent process id is 3891
Exiting with x=3
Parent process1
I am the parent and my process id is 3891
I am the parent and my child process id is 3892
Exiting with x=1
```

Without using System call:



```
cselab11@cbit:~$ ./a.out
3974
Parent process1
I am the parent and my process id is 3973
I am the parent and my child process id is 3974
Exiting with x=1
0
Child Proces3
I am the child and my process id is 3974
I am the child and my parent process id is 1496
Exiting with x=3
```

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.

ALGORITHM:**Zombie Process:**

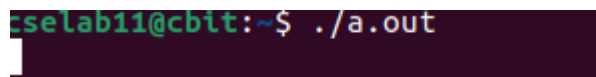
```
Step-1: Start
Step-2: pid_t child_pid = fork()
Step-3: if child_pid>0
        sleep(10)
        Print "Hello"
Step-4: else
        exit(0)
Step-5: End
```

Orphan Process:

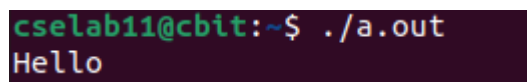
```
Step-1: Start
Step-2: pid = fork()
Step-3: if pid=0
        Print "Child Process"
        Print getpid()
        Print getppid()
        sleep(5)
        Print getpid()
        Print getppid()
Step-4: else
        sleep(10)
        Print getpid()
        Print getppid()
        Print "Parent Terminates"
Step-5: End
```

PROGRAM:

```
#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:

```
cselab11@cbit:~$ ./a.out
```



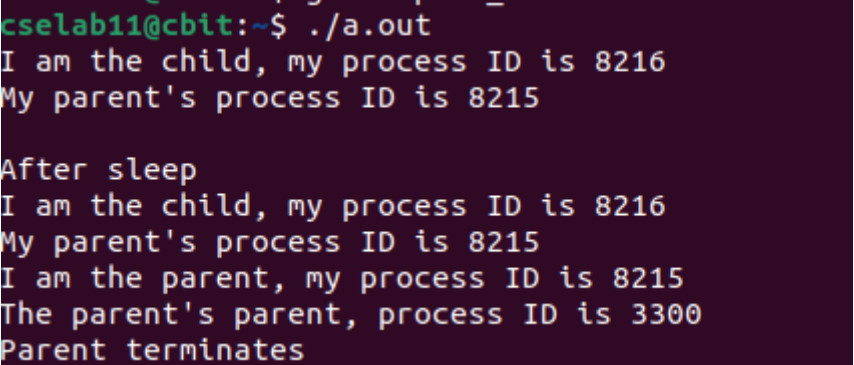
```
cselab11@cbit:~$ ./a.out
Hello
```

PROGRAM:

```
#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:A terminal window with a dark purple background and light green text. The prompt is 'cselab11@cbit:~\$'. The user enters './a.out'. The output is: 'I am the child, my process ID is 8216', 'My parent's process ID is 8215', a blank line, 'After sleep', 'I am the child, my process ID is 8216', 'My parent's process ID is 8215', 'I am the parent, my process ID is 8215', 'The parent's parent, process ID is 3300', and 'Parent terminates'.

```
cselab11@cbit:~$ ./a.out
I am the child, my process ID is 8216
My parent's process ID is 8215

After sleep
I am the child, my process ID is 8216
My parent's process ID is 8215
I am the parent, my process ID is 8215
The parent's parent, process ID is 3300
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

ALGORITHM:

Step-1: Start

Step-2: pid = fork()

Step-3: if pid>0

 Print “Parent”

 Print getpid()

Step-4: else if pid=0

 Print “Child”

 Print getpid()

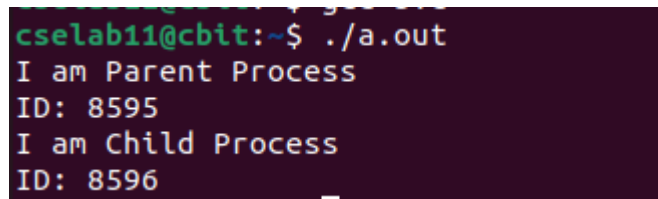
Step-5: else

 Print “Error”

Step-6: End

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:A terminal window with a dark background and light-colored text. The prompt is 'cselab11@cbit:~\$'. The command './a.out' has been executed. The output consists of four lines: 'I am Parent Process', 'ID: 8595', 'I am Child Process', and 'ID: 8596'.

```
cselab11@cbit:~$ ./a.out
I am Parent Process
ID: 8595
I am Child Process
ID: 8596
```

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 execlpe(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. Write the output of the following program**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.

ALGORITHM:

Step-1: Start

Step-2: pid=fork()

Step-3: if pid<0

Print "Fork Failed"

Step-4: else if pid=0

execl("/bin/ls", "ls", NULL);

exit(0)

Step-5: else

wait(NULL)

Print "Child process complete"

Step-6: End

PROGRAM:

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<string.h>

#include<errno.h>

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:

```

cse10p11@cblt:~$ ./a.out
0
012_internal_Exam
050
1
146.txt
'149 Bala'
160120733037_OS_LABINTERNAL.odt
166.txt
20_149
3.c
4.c
a.out
arithmetic.sh
arm.sh
calc.sh
case.sh
cp_sim.c
cse3-sree-batch3
dark.c
demo.txt
Desktop
Documents
Downloads
eo.sh
evenodd.sh
child process complete
example.c
example.txt
exec1.c
exithandler.c
factorial.sh
factorial.sh.save
fact.sh
fact.sh.save
fact.sh.save.1
fact.sh.save.2
fact.ssh
fcfs.c
fibb.sh
fib.sh
file.c
first.txt
fno.sh
fn.sh
hello.c
hello.sh
ifelse.sh
labexam
location.sh
mkdir.c
Music
name.sh
np.sh
oddeven.sh
odd.sh
orphan.c
oslab_315
oslab_315_2-09.zip
outputs.doc
outputs.odt
paging.c
palindrome.sh
pal.sh
peri.sh
Pictures
pidand_ppid.c
pno.sh
pn.sh
power.sh
pow.sh
prg2.c
prg3.c
prime.sh
prog1.c
prog2.c
prog3.c
prog5.c
Public
Q1.c
Question-1.doc
roshini
sample2.c
sample.c
sample.sh
second.txt
seid.txt
show.sh
sjf.c
snap
ssp.c
ssp.sh
temp
Templates
test.txt
Videos
vishnu.c
waipidexample2.c
week8
zombie.c

```

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

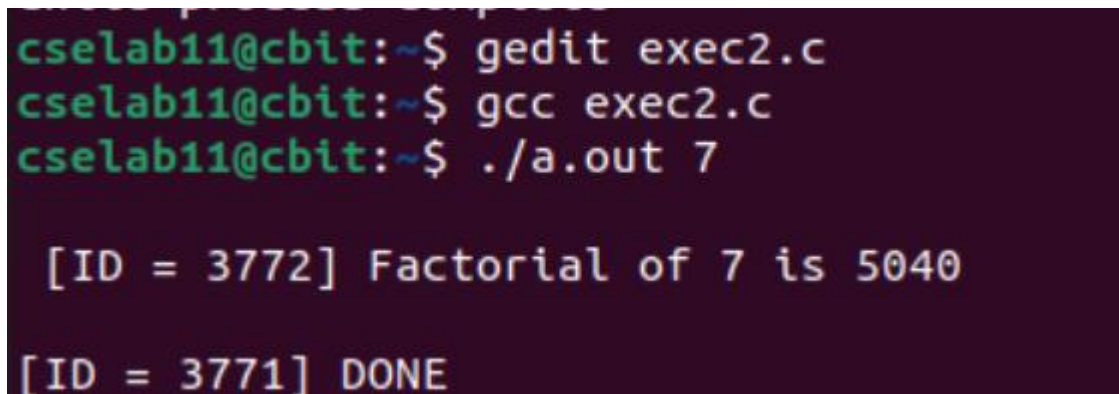
ALGORITHM:

```
Step-1: Start
Step-2: pid_t ret
Step-3: ret = fork()
Step-4: fact=1
Step-5: for i->1;i<argc;i++
        num=atoi(argv[1])
Step-6: if ret=0 and num>0
        for i->1;i<=num;i++
            fact->fact*i
        Print getpid()
        Print fact
Step-7: else
        if num>0
            print "Invalid input"
        wait(NULL)
        Print getpid()
Step-8: End
```

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[1]);
        }
    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:

```
cselab11@cbit:~$ gedit exec2.c
cselab11@cbit:~$ gcc exec2.c
cselab11@cbit:~$ ./a.out 7

[ ID = 3772] Factorial of 7 is 5040

[ ID = 3771] 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

ALGORITHM:

Step-1: Start

Create a function palindrome()

Step-2: Read a number n

Step-3: temp->n

Step-4: sum->0

Step-5: while n>0

 r->n%10

 sum->sum*10+r

 n->n/10

Step-6: if temp==sum

 Print "Palindrome"

Step-7: else

 Print "Not a Palindrome"

Main function:

Step-8: pid->fork()

Step-9: if pid=0

 Print "child"

 palindrome()

Step-10: else

 wait()

 Print "Parent"

Step-11: Create a new file with name exec.c

Step-12: Print pid of exec.c file

Step-13: execv("./pal", args)

Step-14: End

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("./pal", args);
```

```
    printf("The control never comes back to this line #unreachable");
```

```
    return 0;
```

```
}
```

Pal.c

```
#include <stdio.h>
#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:

```
cselab11@cbit:~$ gcc pal.c
cselab11@cbit:~$ gcc exec3.c
cselab11@cbit:~$ ./a.out
PID of execute.c = 3610
I am the child processChecking palindrome
Enter the number: 1234321
Palindrome number
Parent terminating after childcselab11@cbit:~$ ^C
cselab11@cbit:~$ ./a.out
PID of execute.c = 3624
I am the child processChecking palindrome
Enter the number: 123
Not palindrome
Parent terminating after childcselab11@cbit:~$ ^C
```

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

ALGORITHM:

getuid():

```
Step-1: Start
Step-2: Print "The Real user ID is "
Step-3: Print getuid()
Step-4: Print "The Effective user ID is"
Step-5: Print setuid()
Step-6: End
```

Setuid():

```
Step-1: Start
Step-2: Print getuid(), geteuid()
Step-3: if (setuid(25) != 0)
    perror "setuid() error"
Step-4: else
    Print getuid(), geteuid()
Step-5: End
```

Nice():

Step-1: Start

Step-2: Define a function my_nice(int incr):

```
prio -> getpriority(PRIO_PROCESS, 0)
if (setpriority(PRIO_PROCESS, 0, prio+incr)=-1)
    return -1
prio -> getpriority(PRIO_PROCESS, 0)
return prio
```

Step-3: prio -> getpriority(PRIO_PROCESS, 0)

Step-4: Print prio

Step-5: my_nice(5)

Step-6: prio -> getpriority(PRIO_PROCESS, 0)

Step-7: Print prio

Step-8: my_nice(-7)

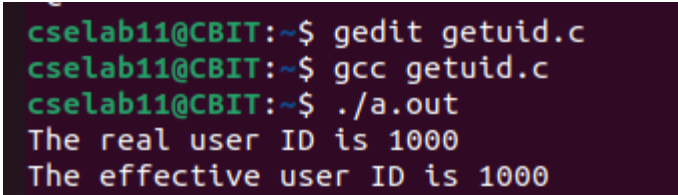
Step-9: prio -> getpriority(PRIO_PROCESS, 0)

Step-10: Print prio

Step-11: End

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:

```
cselab11@CBIT:~$ gedit getuid.c
cselab11@CBIT:~$ gcc getuid.c
cselab11@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;
}
```

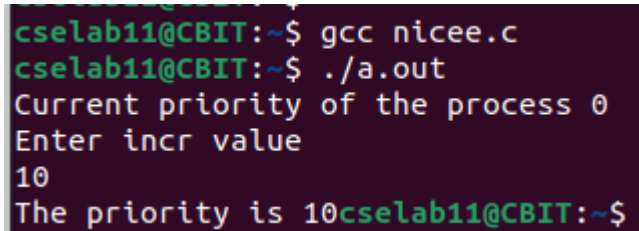
OUTPUT:

```
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
```

```
#include <stdio.h>
#include <unistd.h>
#include <sys/resource.h>
#include <sys/time.h>
int main()
{
    int incr;
    printf("Current priority of the process %d\n",getpriority(PRIO_PROCESS, 0));
    printf("Enter incr value\n");
    scanf("%d", &incr);
    int ret;
    ret = nice(incr);
    printf("The priority is %d", ret);
}
```

OUTPUT:

```
cse1ab11@CBIT:~$ gcc nicee.c
cse1ab11@CBIT:~$ ./a.out
Current priority of the process 0
Enter incr value
10
The priority is 10cse1ab11@CBIT:~$
```

CONCLUSION:

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

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>                /*header file for main function*/
#include<sys/types.h>
#include<sys/stat.h>             /*header files for creat() system call*/
#include<fcntl.h>
int main()
{
    int fd;                      /*creating 2 file descriptors*/
    int fd1;
    fd=creat("first.txt",S_IREAD|S_IWRITE); /*creating 2 files which */
    fd1=creat("second.txt",S_IREAD|S_IWRITE); //returns file descriptors
    printf("%d\n",fd);
    printf("%d\n",fd1);
    if(fd==-1)                   /*checking whether file descriptor is negative or not*/
        printf("ERROR\n");
    else
        printf("SUCCESS\n");
    close(fd);                   /*closing the file descriptors*/
    close(fd1);
}
```

OUTPUT:

```
cse1ab11@cbit:~$ ./a.out
3
4
SUCCESS
cse1ab11@cbit:~$ ls
133          Downloads          loops_demo.sh
152          exit          Music
185          f2          oddeven.sh
3_without_wait.c  factorial          orphan_demo.c
3_with_wait.c    factorial1.c       Pictures
5.c             fact.sh           power.sh
adi            fcfs.c            prime1.c
a.out          first.txt         Public
c3_173         fork.c            second.txt
cprime        'fork demonstrate' sjf.c
create_file.c  fork_system_call.c snap
demo          hello.c          Templates
demoprocess.c ifsample.sh       Videos
Desktop       lab2-5thsept2022  zombie_demo.c
Documents     LOCATION.SH
```

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

ALGORITHM:

Step-1: Start

Step-2: fd -> open(argv[1],O_RDONLY)

Step-3: if fd=-1

 exit(-1)

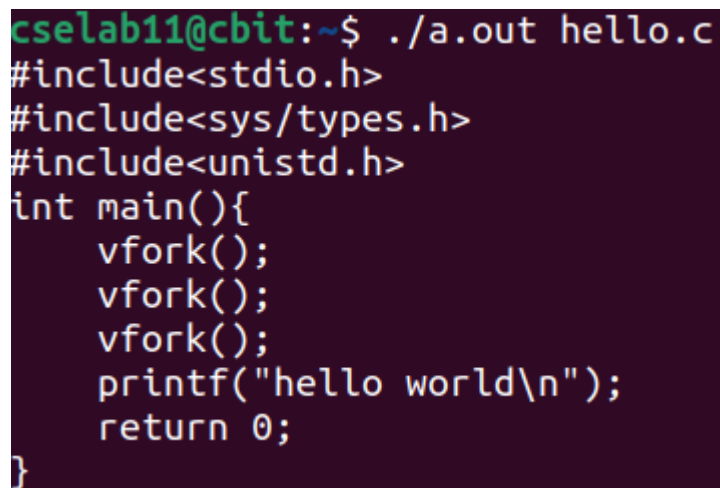
Step-4: while (n_char=read(fd, buffer, 1))!=0

 write(1,buffer,n_char)

Step-5: 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 argc,char *argv[])
{
    int fd;
    int n_char=0;
    char buffer[1];
    fd=open(argv[1],O_RDONLY);
    if(fd==-1)
    {
        exit(-1);
    }
    while((n_char=read(fd,buffer,1))!=0)
    {
        write(1,buffer,n_char);
    }
    return 0;
}
```

OUTPUT:A terminal window with a dark purple background. The prompt is 'cselab11@cbit:~\$'. The command './a.out hello.c' has been entered. The output shows the source code of 'hello.c' being printed to the console. The code includes standard headers and uses 'vfork()' three times before printing 'hello world\n' and returning 0.

```
cselab11@cbit:~$ ./a.out hello.c
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main(){
    vfork();
    vfork();
    vfork();
    printf("hello world\n");
    return 0;
}
```

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

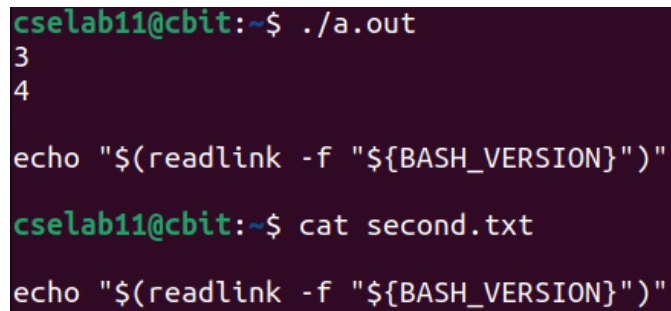
ALGORITHM:

Step-1: Start
Step-2: fd1->open("first.txt",O_RDONLY)
Step-3: Print fd1
Step-4: fd2->creat("second.txt",S_IREAD|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_IREAD|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:A terminal window with a dark purple background. The prompt is 'cselab11@cbit:~\$'. The user enters './a.out', and the output is '3' followed by '4' on the next line. Then, the user enters 'echo "\$(readlink -f "\${BASH_VERSION})"', and the output is 'cselab11@cbit:~\$ cat second.txt'. Finally, the user enters 'echo "\$(readlink -f "\${BASH_VERSION})"', and the output is 'cselab11@cbit:~\$ cat second.txt'.

```
cselab11@cbit:~$ ./a.out  
3  
4  
  
echo "$(readlink -f "${BASH_VERSION}")"  
  
cselab11@cbit:~$ cat second.txt  
  
echo "$(readlink -f "${BASH_VERSION}")"
```

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("1.txt",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("1.txt",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<unistd.h>
```

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

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

```
#include<fcntl.h>
```

```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
int main()
```

```
{
```

```
    int fd1=open("1.txt",O_RDWR);
```

```
    printf("%d",fd1);
```

```
    char buffer[1];
```

```
    buffer[0]='1';
```

```
    printf("enter the data : (press # to exit)");
```

```
    do
```

```
    {
```

```
        scanf("%c",&buffer[0]);
```

```
        if(buffer[0]!='#')
```

```
            write(fd1,buffer,1);
```

```
        }while(buffer[0]!='#');
```

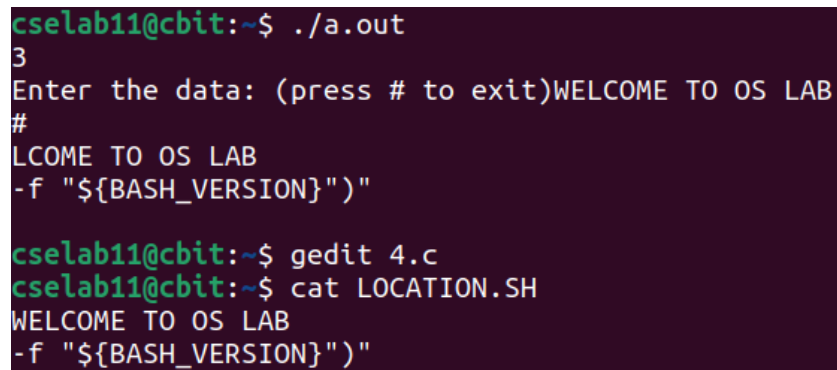
```
    close(fd1);
```

```
    int fd2=open("1.txt",O_RDWR);
```

```
    lseek(fd2,2*sizeof(char),0);int k;
```



```
do
{
k=read(fd2,&buffer[0],1);
if(k!=0)
printf("%c",buffer[0]);
}while(k!=0);
return 0;
}
```

OUTPUT:A terminal window with a dark purple background. The prompt is 'cselab11@cbit:~\$'. The user enters './a.out'. The program outputs '3' on the first line, 'Enter the data: (press # to exit)WELCOME TO OS LAB' on the second line, and '#' on the third line. Then it outputs 'L' on the fourth line, 'COM' on the fifth line, 'E' on the sixth line, and 'TO OS LAB' on the seventh line. Then it outputs '-f "\${BASH_VERSION}"' on the eighth line. The user then enters 'gedit 4.c' and 'cat LOCATION.SH'. The program outputs 'WELCOME TO OS LAB' and '-f "\${BASH_VERSION}"' again.

```
cselab11@cbit:~$ ./a.out
3
Enter the data: (press # to exit)WELCOME TO OS LAB
#
L
COM
E
TO OS LAB
-f "${BASH_VERSION}"
cselab11@cbit:~$ gedit 4.c
cselab11@cbit:~$ cat LOCATION.SH
WELCOME TO OS LAB
-f "${BASH_VERSION}"
```

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.

ALGORITHM:

Step-1: Start
Step-2: Read the data to be entered into a file
Step-3: Print ch
Step-4: res->open("create.txt",O_RDWR)
Step-5: write(res,ch,strlen(ch))
Step-6: close(res)
Step-7: res->open("create.txt",O_RDONLY,0)
Step-8: read(res,&ch,12)
Step-9: Print ch
Step-10: End

PROGRAM:

```
#include<stdio.h>
#include<string.h>
#include<fcntl.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>
void main()
{
    int res;
    char ch[12];
    printf("To write the content entered by the user in to a file: \n\n");
    printf("Enter data you want to enter into 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);
}
```

OUTPUT:

```
To write the content entered by the user in to a file:

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

To read the contents from the same file:
CBIT
-----
Process exited after 1.791 seconds with return value 4
Press any key to continue . . .
```

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

ALGORITHM:

Step-1: Start

Step-2: Define a createf function

Step-3: for i->0;i<2;i++

 Read n

 fptr->fopen(n,'w')

 if fptr=NULL

 Print "Unable to create file"

 Read the data to be entered in to the file

 Write the data into the file

 fclose(fptr)

Step-4: Define a copyfun function

Step-5: Read the name of the source file

Step-6: source->fopen(source_file,"r")

Step-7: if source=NULL

 Print "Press any key to exit"

 exit(EXIT_FAILURE)

Step-8: Read the name of the target file

Step-9: target->fopen(target_file,"w")

Step-10: if target=NULL

 fclose(source)

 print "Press any key to exit"

 exit(EXIT_FAILURE)

Step-11: while (ch->fgetc(source))!=EOF

 fputc(ch,target)

Step-12: fclose(source)

Step-13: fclose(target)

Step-14: End

PROGRAM:

```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
#include<dirent.h>
```

```
#define DATA_SIZE 1000
```

```
void createf()
```

```
{
```

```
    char data[DATA_SIZE];
```

```
    char n[100];
```

```
    FILE * fPtr;
```

```
    int i;
```

```
    printf("Create 2 files \nfile1: with data \nfile2: without data for copying\n");
```

```
    for ( i=0;i<2;i++)
```

```
    {
```

```
        printf("Enter a file name:");
        gets(n);
        fPtr = fopen(n,"w");
        if(fPtr == NULL)
        {
            printf("Unable to create file.\n");
            exit(EXIT_FAILURE);
        }
        printf("Enter contents to store in file: ");
        fgets(data, DATA_SIZE, stdin);
        fputs(data, fPtr);
        fclose(fPtr);
        printf("File created and saved successfully.\n");
    }
}

void copyfun()
{
    char ch, source_file[20], target_file[20];
    FILE *source, *target;
    printf("Enter name of file to copy\n");
    gets(source_file);
    source = fopen(source_file, "r");
    if (source == NULL)
    {
        printf("Press any key to exit...\n");
        exit(EXIT_FAILURE);
    }
    printf("Enter name of target file\n");
    gets(target_file);
    target = fopen(target_file, "w");
    if (target == NULL)
    {
        fclose(source);
        printf("Press any key to exit...\n");
        exit(EXIT_FAILURE);
    }
    while ((ch = fgetc(source)) != EOF)
        fputc(ch, target);
    printf("File copied successfully.\n");
    fclose(source);
    fclose(target);
}

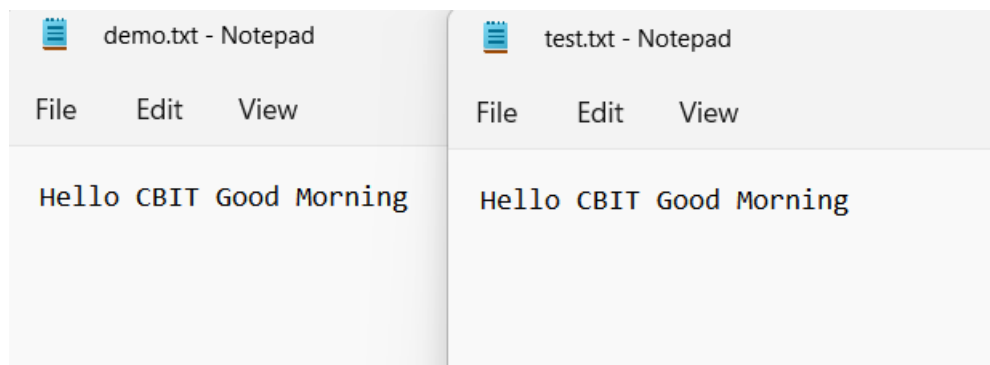
int main(){
    createf();
    copyfun();
}
```

OUTPUT:

```
Create 2 files
file1: with data
file2: without data for copying
Enter a file name:demo.txt
Enter contents to store in file: Hello CBIT Good Morning
File created and saved successfully.
Enter a file name:test.txt
Enter contents to store in file: Hi
File created and saved successfully.
Enter name of file to copy
demo.txt
Enter name of target file
test.txt
File copied successfully.

-----
Process exited after 34.75 seconds with return value 0
Press any key to continue . . .
```

Contents of demo.txt file and test.txt file after executing the above program:

**CONCLUSION:**

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

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.

ALGORITHM:

Step-1: Start

Step-2: Read the file name

Step-3: Read the pattern to be searched

Step-4: fp->fopen(fn,"r")

Step-5: while !feof(fp)
 fgets(temp,1000,fp)
 if strstr(temp,pat)
 Print temp

Step-6: fclose(fp)

Step-7: End

PROGRAM:

```
#include<stdio.h>
#include<string.h>
void 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);
}
```

OUTPUT:

```
Enter file name: FCFS.c
Enter pattern to be searched: int
int waitingtime(int proc[], int n, int burst_time[], int wait_time[], int at[])
    int i;
int turnaroundtime( int proc[], int n, int burst_time[], int wait_time[], int tat[]) {
    int i;
int avgttime( int proc[], int n, int burst_time[], int at[])
    int wait_time[n], tat[n], total_wt = 0, total_tat = 0;
    int i;
    printf("Processes  Arrival time  Burst   Waiting Turn around \n");
        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);
int main() {
    int proc[50], burst_time[50], at[50];
    int n,i;
    printf("Enter the number of processes: ");
    printf("Enter the arrival time of the processes: ");
    printf("Enter the burst time of the processes: ");

-----
Process exited after 6.584 seconds with return value 0
Press any key to continue . . .
```

CONCLUSION:

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

STAT SYSTEM CALL

AIM: To demonstrate the Stat System Call

DESCRIPTION:

Stat System call is a system call in Linux to check the status of a file such as to check when the file was accessed. The stat() system call actually returns file attributes. The file attributes of an inode are basically returned by Stat() function. An inode contains the metadata of the file. An inode contains: the type of the file, when the file was accessed (modified, deleted) that is timestamps, and the path of the file, the user ID and the group ID, links of the file, and physical address of file content.

SYNTAX:

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

ALGORITHM:

Step-1: Start
Step-2: struct stat sfile
Step-3: stat("file.c", &sfile)
Step-4: Print sfile.st_mode
Step-5: Print sfile.st_uid
Step-6: Print sfile.st_size
Step-7: Print sfile.st_gid
Step-8: Print sfile.st_nlink
Step-9: End

PROGRAM:

```
#include <stdio.h>
#include <sys/stat.h>
int main()
{
    struct stat sfile;
    stat("file.c", &sfile);
    printf("st_mode = %o\n", sfile.st_mode);
    printf("user id: %d\n", sfile.st_uid);
    printf("File size: %ld\n", sfile.st_size);
    printf("File group id: %d\n", sfile.st_gid);
    printf("st_nlink: %u\n", (unsigned int)sfile.st_nlink);
    return 0;
}
```

OUTPUT:

```
cse1ab11@CSELAB:~$ gedit stat.c
cse1ab11@CSELAB:~$ gcc stat.c
cse1ab11@CSELAB:~$ ./a.out
st_mode = 16300072151
user id: 1819177588
File size: 96
File group id:3480
st_nlink:1819435365
```

CONCLUSION:

By executing the above program, we have successfully demonstrated the Stat System Call.

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*

ALGORITHM:

Mkdir():

Step-1: Start

Step-2: Read the name of the directory to be created

Step-3: status -> mkdir(dir_name, 0755)

Step-4: if (status == 0):

 Print "Directory creation successful"

Step-5: else:

 Print "Directory not created"

Step-6: End

Chmod():

Step-1: Start

Step-2: filename -> argv[1]

Step-3: r -> stat(filename, &fs)

Step-4: if r=-1:

Print stderr

Print "Error Reading"

Print filename

Step-5: r -> chmod(filename, fs.st_mode|S_IWGRP+S_IWOTH)

Step-6: if r!=0:

Print stderr

Print "Unable to reset permissions"

Print filename

Step-7: stat(filename, &fs)

Step-8: End

Chown():

Step-1: Start

Step-2: ecode -> 0

Step-3: for (i->1;i<argc;i++)

if (chown(argv[i],1000,24)=0)

perror(argv[i])

ecode++

Step-4: exit(ecode)

Step-5: End

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:

```
cse1ab11@CBIT:~$ gcc mkdir.c
```

```
cse1ab11@CBIT:~$ ./a.out
```

```
HomeDir
```

```
Directory creation successfullcse1ab11@CBIT:~$
```

```
Directory creation successfulcselab11@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         hgfs         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:

```
cse1ab11@CBIT:~$ cd HomeDir
cse1ab11@CBIT:~/HomeDir$ gcc chown.c
cse1ab11@CBIT:~/HomeDir$ ls -l
total 20
-rwxr-xr-x 1 cse1ab11 cse3 16048 Nov 11 10:55 a.out
-rw-r--r-- 1 cse1ab11 cse3  313 Nov 11 10:51 chown.c
-rw-r--r-- 1 cse1ab11 cse3    0 Nov 11 10:53 file.txt
cse1ab11@CBIT:~/HomeDir$ ./a.out file.txt
file.txt: Success
cse1ab11@CBIT:~/HomeDir$ ls -l
total 20
-rwxr-xr-x 1 cse1ab11 cse3  16048 Nov 11 10:55 a.out
-rw-r--r-- 1 cse1ab11 cse3    313 Nov 11 10:51 chown.c
-rw-r--r-- 1 cse1ab11 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.

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 means 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.

ALGORITHM:

Step-1: Start

Step-2: In function int waitingtime (int proc[], int n, int burst_time[], int wait_time[], int at[])

Set wait_time[0] = 0

Loop For i = 1 and i < n and i++

Set wait_time[i] = burst_time[i-1] + wait_time[i-1] - (at[i] - a[i-1])

End For

Step-3: In function int turnaroundtime(int proc[], int n, int burst_time[], int wait_time[], int tat[])

Loop For i = 0 and i < n and i++

Set tat[i] = burst_time[i] + wait_time[i]

End For

Step-4: In function int avgtime(int proc[], int n, int burst_time[], int at[])

Declare and initialize wait_time[n], tat[n], total_wt=0, total_tat = 0

Call waitingtime(proc, n, burst_time, wait_time, tat)

Call turnaroundtime(proc, n, burst_time, wait_time, tat)

Loop For i=0 and i<n and i++

Set total_wt = total_wt + wait_time[i]

Set total_tat = total_tat + tat[i]

Print process number, arrival time, burst time, wait time and turnaround time

End For

Print Average waiting time = total_wt/n

Print Average turnaround time = total_tat/n

Step-5: In int main()

Read the input from user n, burst_time[], at[]

Call avgtime(proc, n, burst_time, at)

Step-6: End

PROGRAM:

```
#include <stdio.h>
```

```
//Function to calculate waiting time
```

```
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;
```

```
}
```



```
// Function to calculate turn around time
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;
}

//Function to Calculation Average waiting and Turn around time
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 Arrival time Burst Waiting Turn around \n");
    for ( i=0; i<n; i++) {
        total_wt = total_wt + wait_time[i];
        total_tat = 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;
}

// main function
int main() {
    //process id's
    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:

```
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

-----
Process exited after 17.36 seconds with return value 0
Press any key to continue . . .
```

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

ALGORITHM:

Step-1: Start

Step-2: Take process, arrival time, burst time input from the user

Step-3: Sort the process according to arrival time and if the process has the same arrival time then sort them having less burst time

Step-4: Swap the process one above one in the order of execution

Step-5: Find the turnaround time (tat) and waiting time (wt)

Step-6: Find average tat and average wt

Step-7: End

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]--;
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%d\n", smallest+1, arrival_time[smallest],
temp[smallest], wt, tat);
}
}
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:

```
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
-----
Process exited after 6.16 seconds with return value 0
Press any key to continue . . .
```

CONCLUSION:

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

BANKER'S 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.

ALGORITHM:

1. Let Work and Finish be vectors of length ' m ' and ' n ' respectively.
Initialize: Work = Available
Finish[i] = false; for $i=1, 2, 3, 4, \dots, n$
2. Find an i such that both
 - a) Finish[i] = false
 - b) Need[i] \leq Workif no such i exists goto step (4)
3. Work = Work + Allocation[i]
Finish[i] = true
goto step (2)
4. if Finish [i] = true for all i then the system is in a safe state

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=[]
for i in range(n):
    l=list(map(int, input().strip().split()))
    max.append(l)
print("Enter the Avail matrix: ")
avail=list(map(int, input().strip().split()))
print("Allocated Matrix: ", alloc, end="\n")
print("Max Matrix: ", max, 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[i][j] - alloc[i][j]
y=0
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
            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:

```
= RESTART: D:/Engineering/SEM-5/Operating System/Lab Work/Programs/Banker's Algo
rithm.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.

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.

ALGORITHM:

Step-1: Start
Step-2: Read the memory size
Step-3: Read the Page size
Step-4: Calculate the number of pages
Step-5: Read the number of Frames
Step-6: Print the total size of the memory
Step-7: Read the frame number for each page and store them in page table
Step-8: Print the page table
Step-9: Read the logical address i.e., the page number and offset
Step-10: Search for the input page number if found go to step-12
Step-11: If not found print page not found go to step 15
Step-12: Read the base register
Step-13: Calculate Physical Address i.e., $\text{phy_add} = \text{base_reg} + \text{inp_fra} * \text{page_size} + \text{inp_off}$
Step-14: Print physical address
Step-15: End

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 = list()
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==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:

```
===== RESTART: C:\Users\CBIT\Desktop\012\pagging.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.

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.

ALGORITHM:

```
Step-1: Start
Step-2: Read number of segments no_of_seg
Step-3: seg_table -> []
Step-4: for i in range (no_of_seg):
    Read the base and limit of segment
    Seg_table.append([base, limit])
Step-5: for i in seg_table
    Print i
Step-6: Read the logical address [Segment number, Offset]
Step-7: if (offset < seg_table[inp_seg_no][1]):
    phy_add = seg_table[inp_seg_no][0] + offset
    Print phy_add
Step-8: else
    Print "offset is greater than limit"
```

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(offset < seg_table[inp_seg_no][1]):
    phy_add = seg_table[inp_seg_no][0] + offset
    print("The physical address of given logical address is: ", phy_add)
else:
    print("Offset is greater than limit")
```

OUTPUT:

```
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 given logical address is: 3008
```

CONCLUSION:

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

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.

ALGORITHM:

Step 1: Start the program.

Step 2: Get the number of memory partition and their sizes.

Step 3: Get the number of processes and values of block size for each process.

Step 4: First fit algorithm searches all the entire memory block until a hole which is big enough is encountered. It allocates that memory block for the requesting process.

Step 5: Best-fit algorithm searches the memory blocks for the smallest hole which can be allocated to requesting process and allocates it.

Step 6: Worst fit algorithm searches the memory blocks for the largest hole and allocates it to the process.

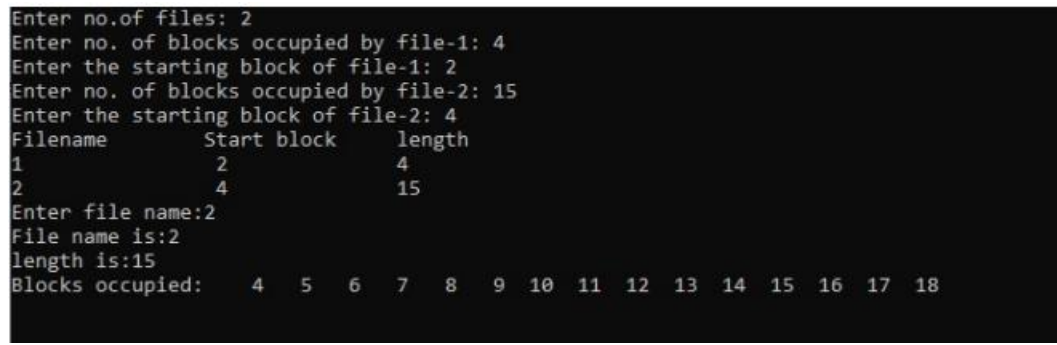
Step 7: Analyses all the three memory management techniques and display the best algorithm which utilizes the memory resources effectively and efficiently.

Step 8: Stop the program.

PROGRAM:

```
#include <stdio.h>
#include <conio.h>
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 %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();
}
```

OUTPUT:

```
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      Start block  length
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 ith entry in the index block contains the disk address of the ith file block.

ALGORITHM:

Step 1: Start.

Step 2: Let n be the size of the buffer

Step 3: check if there are any producer

Step 4: if yes check whether the buffer is full

Step 5: If no the producer item is stored in the buffer

Step 6: If the buffer is full the producer has to wait

Step 7: Check there is any consumer. If yes check whether the buffer is empty

Step 8: If no the consumer consumes them from the buffer

Step 9: If the buffer is empty, the consumer has to wait.

Step 10: Repeat checking for the producer and consumer till required

Step 11: Terminate the process.

PROGRAM:

```
#include <stdio.h>
#include <conio.h>
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]);  
getch();  
}
```

OUTPUT:

```
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.

ALGORITHM:

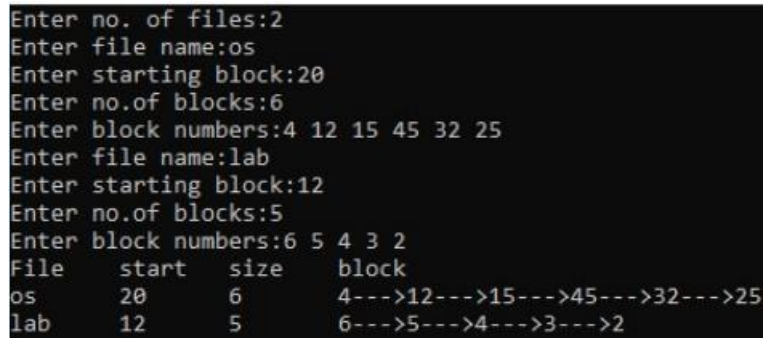
- Step 1: Create a queue to hold all pages in memory
- Step 2: When the page is required replace the page at the head of the queue
- Step 3: Now the new page is inserted at the tail of the queue
- Step 4: Create a stack
- Step 5: When the page fault occurs replace page present at the bottom of the stack
- Step 6: Stop the allocation.

PROGRAM:

```
#include <stdio.h>
#include <conio.h>
struct file
{
    char fname[10];
    int start,size,block[10];
}f[10];
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");
    }
    getch();
}
```

OUTPUT:

```
Enter no. of files:2
Enter file name:os
Enter starting block:20
Enter no.of blocks:6
Enter block numbers:4 12 15 45 32 25
Enter file name:lab
Enter starting block:12
Enter no.of blocks:5
Enter block numbers:6 5 4 3 2
File   start   size   block
os     20       6     4--->12--->15--->45--->32--->25
lab    12       5     6--->5--->4--->3--->2
```

CONCLUSION:

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

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 Berkely 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()

ALGORITHM:

Step-1: Start
Step-2: sfd = socket(AF_INET, SOCK_STREAM, 0)
Step-3: Print sfd, strerror(errno)
Step-4: struct sockaddr_in addr
Step-5: addr.sin_family = AF_INET
Step-6: addr.sin_port = htons(8090)
Step-7: status = inet_pton(AF_INET, "127.0.1", &addr.sin_addr)
Step-8: Print status, strerror(errno)
Step-9: cfd = connect(sfd, (struct sockaddr*)&addr, addrlen)
Step-10: Print cfd, strerror(errno)
Step-11: assert(cfd != -1)
Step-12: Read a buffer Message from the user
Step-13: status = send(sfd, buf, strlen(buf), 0)
Step-14: Print status, strerror(errno)
Step-15: assert(status != -1)
Step-16: status = read(sfd, buf, 1024)
Step-17: Print status, strerror(errno)
Step-18: assert(status != -1)
Step-19: Print buf
Step-20: close(cfd)
Step-21: End

PROGRAM:**client.c:**

```
#include<assert.h>
#include<errno.h>
#include<stdio.h>
#include<string.h>

#include<arpa/inet.h>
#include<sys/socket.h>
#include<sys/types.h>

int main() {
    int status;
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    printf("sfd: %d, %s\n", sfd, strerror(errno));
    assert(sfd != -1);

    struct sockaddr_in addr;
    socklen_t addrlen = sizeof(addr);
    addr.sin_family = AF_INET;
    addr.sin_port = htons(8090);

    status = inet_pton(AF_INET, "127.0.0.1", &addr.sin_addr);
    printf("ip convert status: %d, %s\n", status, strerror(errno));
    assert(status != -1);

    int cfd = connect(sfd, (struct sockaddr*)&addr, addrlen);
```

```
printf("cfd: %d, %s\n", cfd, strerror(errno));
assert(cfd != -1);

char buf[1024];
printf("Enter message: ");
scanf("%[^\n]", buf);

status = send(sfd, buf, strlen(buf), 0);
printf("send status: %d, %s\n", status, strerror(errno));
assert(status != -1);

status = read(sfd, buf, 1024);
printf("server status: %d, %s\n", status, strerror(errno));
assert(status != -1);

printf("Got: %s\n", buf);

close(cfd);
return 0;
}
```

server.c:

```
#include<assert.h>
#include<errno.h>
#include<stdio.h>
#include<string.h>

#include<arpa/inet.h>
#include<sys/socket.h>
#include<sys/types.h>

int main() {
    int status;
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    printf("sfd: %d, %s\n", sfd, strerror(errno));
    assert(sfd != -1);

    struct sockaddr_in addr;
    socklen_t addrlen = sizeof(addr);
    addr.sin_family = AF_INET;
    addr.sin_port = htons(8090);
    addr.sin_addr.s_addr = INADDR_ANY;
    status = bind(sfd, (struct sockaddr *)&addr, addrlen);
    printf("status: %d, %s\n", status, strerror(errno));
    assert(status != -1);

    status = listen(sfd, 5);
    printf("status: %d, %s\n", status, strerror(errno));
    assert(status != -1);
}
```

```
int nsfd = accept(sfd, (struct sockaddr *)&addr, &addrlen);
printf("nsfd: %d, %s\n", nsfd, strerror(errno));
assert(nsfd != -1);

char buf[1024];
status = read(nsfd, buf, 1024);
printf("status: %d, %s\n", status, strerror(errno));
assert(status != -1);
printf("Got: %s\n", buf);

ssize_t send_status = send (nsfd, buf, strlen(buf), 0);
printf("send: %lu, %s\n", send_status, strerror(errno));
assert(status != -1);

close(nsfd);
shutdown(sfd, SHUT_RDWR);
return 0;
}
```

OUTPUT:

```
server.c: In function 'main':
server.c:34:11: warning: implicit declaration of function 'read'; did you mean 'freadd'? [-Wimplicit-function-declaration]
  34 |     status = read(nsfd, buf, 1024);
      |               ^~~~~~
      |               freadd
server.c:43:2: warning: implicit declaration of function 'close'; did you mean 'pclose'? [-Wimplicit-function-declaration]
   43 |     close(nsfd);
      |     ^~~~~~
      |     pclose
sfd: 3, Success
status: 0, Success
status: 0, Success
nsfd: 4, Success
status: 13, Success
Got: Hello, World!
send: 14, Success
cbit@DESKTOP-049FT1G:/mnt/c/Users/CBIT/Documents/301/w-0212$

cbit@DESKTOP-049FT1G:/mnt/c/Users/CBIT/Documents/301/w-0212$ cc client.c -o client && ./client
client.c: In function 'main':
client.c:37:11: warning: implicit declaration of function 'read'; did you mean 'freadd'? [-Wimplicit-function-declaration]
   37 |     status = read(sfd, buf, 1024);
      |               ^~~~~~
      |               freadd
client.c:43:2: warning: implicit declaration of function 'close'; did you mean 'pclose'? [-Wimplicit-function-declaration]
   43 |     close(cfd);
      |     ^~~~~~
      |     pclose
sfd: 3, Success
ip convert status: 1, Success
cfd: 0, Success
Enter message: Hello, World!
send status: 13, Success
server status: 14, Success
Got: Hello, World!
cbit@DESKTOP-049FT1G:/mnt/c/Users/CBIT/Documents/301/w-0212$
```

CONCLUSION:

By executing the above program, we have successfully implemented the echo-server using Socket Programming.