RAJALAKSHMI
ENGINEERING COLLEGE
An AUTONOMOUS Institution
Affiliated to ANNA UNIVERSITY, Chennai

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

CS23431 – OPERATING SYSTEMS

(REGULATION 2023)

RAJALAKSHMI ENGINEERING COLLEGE
Thandalam, Chennai-602015

Name: . . . . . HARISH M. . . . . .

Register No. : . . . . .2116-231501058 . . . . . .

Year / Branch / Section: . . .2nd Yr/AIML/FA . . . . . .

Semester: . . . . . IV. . . . . .

Academic Year: . . .2024-2025. . . . . .

INDEX

EXP.NO	Date	Title	Page No	Signature
1a	29/3/25	Installation and Configuration of Linux		
1b	31/1/25	Basic Linux Commands		
2	7/2/25	Shell script a) Arithmetic Operation -using expr command b) Check leap year using if-else		
3 b)	7/2/25	a) Reverse the number using while loop b) Fibonacci series using for loop		
4	14/2/25	Text processing using Awk script a) Employee average pay b) Results of an examination		
5	21/2/25	System calls –fork(), exec(), getpid(),opendir(), readdir()		
6a	28/2/25	FCFS		
6b	28/2/25	SJF		
6c	28/2/25	Priority		
6d	28/2/25	Round Robin		
7.	19/3/25	Inter-process Communication using Shared Memory		
8	26/3/25	Producer Consumer using Semaphores		
9	2/4/25	Bankers Deadlock Avoidance algorithms		
10 a	4/4/25	Best Fit		

10 b	4/425	First Fit		
11a	9/4/25	FIFO		
11b	9/4/25	LRU		
11c	9/4/25	Optimal		
12	11/4/25	File Organization Technique- single and Two level directory		

RAJALAKSHMI ENGINEERING COLLEGE (AUTONOMOUS)

RAJALAKSHMI NAGAR, THANDALAM – 602 105

BONAFIDE CERTIFICATE

NAME M HARISH REGISTER NO. **2116-231501058**

ACADEMIC YEAR 2024-25 **SEMESTER**- IV **BRANCH**: AIML-B.Tech

This Certification is the Bonafide record of work done by the above student

in the **CS23431- Operating Systems** Laboratory during the year

2024 – 2025.

Signature of Faculty -in – Charge

Submitted for the Practical Examination held on _____

Internal Examiner External Examiner

Ex No: 1a) **INSTALLATION AND CONFIGURATION OF LINUX**

Date:

Aim:

To install and configure Linux operating system in a Virtual Machine.

Installation/Configuration Steps:

1. Install the required packages for virtualization
dnf install xen virt-manager qemu libvirt

2. Configure xend to start up on boot
systemctl enable virt-manager.service

3. Reboot the machine
Reboot

4. Create Virtual machine by first running virt-manager
virt-manager &

5. Click on File and then click to connect to localhost

6. In the base menu, right click on the localhost(QEMU) to create a new VM 7. Select

Linux ISO image

8. Choose puppy-linux.iso then kernel version

9. Select CPU and RAM limits

10. Create default disk image to 8 GB

11. Click finish for creating the new VM with PuppyLinux

Output:

Result :

Linux has been successfully installed and configured, providing a stable operating system environment for further experimentation and development." (This confirms the base setup is complete.

Ex No: 1b)

Date:

BASIC LINUX COMMANDS

1.1 GENERAL PURPOSE COMMANDS

1. The 'date' command:

The date command displays the current date with day of week, month, day, time (24 hours clock) and the year.

SYNTAX: $ date

The date command can also be used with following format.

Format	Purpose	Example
+ %m	To display only month	$ date + %m
+ %h	To display month name	$ date + %h
+ %d	To display day of month	$ date + %d
+ %y	To display last two digits of the year	$ date + %y
+ %H	To display Hours	$ date + %H
+ %M	To display Minutes	$ date + %M
+ %S	To display Seconds	$ date + %S

2. The echo'command:

The echo command is used to print the message on the screen.

SYNTAX: $ echo

EXAMPLE: $ echo "God is Great"

3. The 'cal' command:

The cal command displays the specified month or year calendar.

SYNTAX: $ cal [month] [year]

EXAMPLE: $ cal Jan 2012

4. The 'bc' command:

Unix offers an online calculator and can be invoked by the command bc.

SYNTAX: $ bc

EXAMPLE: bc –l

16/4

5/2

5. The ‘who’ command

The who command is used to display the data about all the users who are currently logged into the system.

SYNTAX: $ who

6. The ‘who am i’ command

The who am i command displays data about login details of the user.

SYNTAX: $ who am i

7. The ‘id’ command

The id command displays the numerical value corresponding to your login.

SYNTAX: $ id

8. The ‘tty’ command

The tty (teletype) command is used to know the terminal name that we are using.

SYNTAX: $ tty

9. The ‘clear’ command

The clear command is used to clear the screen of your terminal.

SYNTAX: $ clear

10. The ‘man’ command

The man command gives you complete access to the Unix commands.

SYNTAX: $ man [command]

11. The ‘ps’ command

The ps command is used to the process currently alive in the machine with the 'ps' (process status) command, which displays information about process that are alive when you run the command. 'ps;' produces a snapshot of machine activity.

SYNTAX: $ ps

EXAMPLE: $ ps

$ ps –e

$ps -aux

12. The ‘uname’ command

The uname command is used to display relevant details about the operating system on the standard output.

-m -> Displays the machine id (i.e., name of the system hardware)

-n -> Displays the name of the network node. (host name)

-r -> Displays the release number of the operating system.

-s -> Displays the name of the operating system (i.e.. system name)

-v -> Displays the version of the operating system.

-a -> Displays the details of all the above five options.

SYNTAX: $ uname [option]

EXAMPLE: $ uname -a

1.2 DIRECTORY COMMANDS

1. The ‘pwd’ command:

The pwd (print working directory) command displays the current working directory.

SYNTAX: $ pwd

2. The ‘mkdir’ command:

The mkdir is used to create an empty directory in a disk.

SYNTAX: $ mkdir dirname

EXAMPLE: $ mkdir receee

3. The ‘rmdir’ command:

The rmdir is used to remove a directory from the disk. Before removing a directory, the directory must be empty (no files and directories).

SYNTAX: $ rmdir dirname

EXAMPLE: $ rmdir receee

4. The ‘cd’ command:

The cd command is used to move from one directory to another.

SYNTAX: $ cd dirname

EXAMPLE: $ cd receee

5. The ‘ls’ command:

The ls command displays the list of files in the current working directory.

SYNTAX: $ ls

EXAMPLE: $ ls

$ ls –l

$ ls –a

1.3 FILE HANDLING COMMANDS

1. The ‘cat’ command:

The cat command is used to create a file.

SYNTAX: $ cat > filename

EXAMPLE: $ cat > rec

2. The ‘Display contents of a file’ command:

The cat command is also used to view the contents of a specified file.

SYNTAX: $ cat filename

3. The ‘cp’ command:

The cp command is used to copy the contents of one file to another and copies the file from one place to another.

SYNTAX: $ cp oldfile newfile

EXAMPLE: $ cp cse ece

4. The ‘rm’ command:

The rm command is used to remove or erase an existing file

SYNTAX: $ rm filename

EXAMPLE: $ rm rec

$ rm –f rec

Use option –fr to delete recursively the contents of the directory and its subdirectories.

5. The ‘mv’ command:

The mv command is used to move a file from one place to another. It removes a specified file from its original location and places it in specified location.

SYNTAX: $ mv oldfile newfile

EXAMPLE: $ mv cse eee

6. The ‘file’ command:

The file command is used to determine the type of file.

SYNTAX: $ file filename

EXAMPLE: $ file receee

7. The ‘wc’ command:

The wc command is used to count the number of words, lines and characters in a file.

SYNTAX: $ wc filename

EXAMPLE: $ wc receee

8. The ‘Directing output to a file’ command:

The ls command lists the files on the terminal (screen). Using the redirection operator ‘>’ we can send the output to file instead of showing it on the screen.

SYNTAX: $ ls > filename

EXAMPLE: $ ls > cseeee

9. The ‘pipes’ command:

The Unix allows us to connect two commands together using these pipes. A pipe (|) is an mechanism by which the output of one command can be channeled into the input of another command.

SYNTAX: $ command1 | command2

EXAMPLE: $ who | wc -l

10. The ‘tee’ command:

While using pipes, we have not seen any output from a command that gets piped into another command. To save the output, which is produced in the middle of a pipe, the tee command is very useful.

SYNTAX: $ command | tee filename

EXAMPLE: $ who | tee sample | wc -l

11. The ‘Metacharacters of unix’ command:

Metacharacters are special characters that are at higher and abstract level compared to most of other characters in Unix. The shell understands and interprets these metacharacters in a special way.

* - Specifies number of characters

?- Specifies a single character

[ ]- used to match a whole set of file names at a command line.

! – Used to Specify Not

EXAMPLE:

$ ls r** - Displays all the files whose name begins with ‘r’

$ ls ?kkk - Displays the files which are having ‘kkk’, from the second characters irrespective of the first character.

$ ls [a-m] – Lists the files whose names begins alphabets from ‘a’ to ‘m’

$ ls [!a-m] – Lists all files other than files whose names begins alphabets from ‘a’ to ‘m’ 12.

The 'File permissions' command:

File permission is the way of controlling the accessibility of file for each of three users namely Users, Groups and Others.

There are three types of file permissions are available, they are

r-read
w-write
x-execute

The permissions for each file can be divided into three parts of three bits each.

First three bits	Owner of the file
Next three bits	Group to which owner of the file belongs
Last three bits	Others

EXAMPLE: $ ls college

-rwxr-xr-- 1 Lak std 1525 jan10 12:10 college

Where,

-rwx The file is readable, writable and executable by the owner of the file.

Lak Specifies Owner of the file.

r-x Indicates the absence of the write permission by the Group owner of the file. Std Is the Group Owner of the file.

r-- Indicates read permissions for others.

13. The 'chmod' command:

The chmod command is used to set the read, write and execute permissions for all categories of users for file.

SYNTAX: $ chmod category operation permission file

Category	Operation	permission
u-users	+ assign	r-read
g-group	-Remove	w-write
o-others	= assign absolutely	x-execute
a-all		

EXAMPLE:

$ chmod u –wx college

Removes write & execute permission for users for ‘college’ file.

$ chmod u +rw, g+rw college

Assigns read & write permission for users and groups for ‘college’ file.

$ chmod g=wx college

Assigns absolute permission for groups of all read, write and execute permissions for ‘college’ file.

14. The ‘Octal Notations’ command:

The file permissions can be changed using octal notations also. The octal notations for file permission are

Read permission	4
Write permission	2

EXAMPLE:

$ chmod 761 college

Execute permission	1

Assigns all permission to the owner, read and write permissions to the group and only executable permission to the others for ‘college’ file.

1.4 GROUPING COMMANDS

1. The ‘semicolon’ command:

The semicolon(;) command is used to separate multiple commands at the command line.

SYNTAX: $ command1;command2;command3..... ;commandn

EXAMPLE: $ who;date

2. The ‘&&’ operator:

The ‘&&’ operator signifies the logical AND operation in between two or more valid Unix commands.It means that only if the first command is successfully executed, then the next command will executed.

SYNTAX: $ command1 && command && command3 .....&&commandn

EXAMPLE: $ who && date

3. The '||' operator:

The '||' operator signifies the logical OR operation in between two or more valid Unix commands.It means, that only if the first command will happen to be un successfully,it will continue to execute next commands.

SYNTAX: $ command1 || command || command3..... ||commandn

EXAMPLE: $ who || date

1.5 FILTERS

1. The head filter

It displays the first ten lines of a file.

SYNTAX: $ head filename

EXAMPLE: $ head college Display the top ten lines.

$ head -5 college Display the top five lines.

2. The tail filter

It displays ten lines of a file from the end of the file.

SYNTAX: $ tail filename

EXAMPLE: $ tail college Display the last ten lines.

$tail -5 college Display the last five lines.

3. The more filter:

The pg command shows the file page by page.

SYNTAX: $ ls –l | more

4. The 'grep' command:

This command is used to search for a particular pattern from a file or from the standard input and display those lines on the standard output. "Grep" stands for "global search for regular expression."

SYNTAX: $ grep [pattern] [file_name]

EXAMPLE: $ cat> student

Arun cse

Ram ece

Kani cse

$ grep "cse" student

Arun cse

Kani cse

5. The 'sort' command:

The sort command is used to sort the contents of a file. The sort command reports only to the

screen, the actual file remains unchanged.

SYNTAX: $ sort filename

EXAMPLE: $ sort college

OPTIONS:

Command	Purpose
Sort –r college	Sorts and displays the file contents in reverse order
Sort –c college	Check if the file is sorted
Sort –n college	Sorts numerically
Sort –m college	Sorts numerically in reverse order
Sort –u college	Remove duplicate records
Sort –l college	Skip the column with +1 (one) option.Sorts according to second column

6. The ‘nl’ command:

The nl filter adds lines numbers to a file and it displays the file and not provides access to edit but simply displays the contents on the screen.

SYNTAX: $ nl filename

EXAMPLE: $ nl college

7. The ‘cut’ command:

We can select specified fields from a line of text using cut command.

SYNTAX: $ cut -c filename

EXAMPLE: $ cut -c college

OPTION:

-c – Option cut on the specified character position from each line.

1.5 OTHER ESSENTIAL COMMANDS

1. **free**

Display amount of free and used physical and swapped memory system.

synopsis- free [options]

example

[root@localhost ~]# free -t

total used free shared buff/cache available Mem: 4044380 605464 2045080

148820 1393836 3226708 Swap: 2621436 0 2621436

Total: 6665816 605464 4666516

2. **top**

It provides a dynamic real-time view of processes in the system.

synopsis- top [options]

example

[root@localhost ~]# top

top - 08:07:28 up 24 min, 2 users, load average: 0.01, 0.06, 0.23

Tasks: 211 total, 1 running, 210 sleeping, 0 stopped, 0 zombie

%Cpu(s): 0.8 us, 0.3 sy, 0.0 ni, 98.9 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

KiB Mem : 4044380 total, 2052960 free, 600452 used, 1390968 buff/cache KiB Swap:

2621436 total, 2621436 free, 0 used. 3234820 avail Mem PID USER PR NI VIRT RES

SHR S %CPU %MEM TIME+ COMMAND

1105 root 20 0 175008 75700 51264 S 1.7 1.9 0:20.46 Xorg 2529 root 20 0 80444

32640 24796 S 1.0 0.8 0:02.47 gnome-term 3. **ps**

It reports the snapshot of current processes

synopsis- ps [options]

example

[root@localhost ~]# ps -e

PID TTY TIME CMD

1 ? 00:00:03 systemd

2 ? 00:00:00 kthreadd

3 ? 00:00:00 ksoftirqd/0

4. **vmstat**

It reports virtual memory statistics

synopsis- vmstat [options]

example

[root@localhost ~]# vmstat

procs ----------memory---------- - - swap- - - - io----- -system- -------cpu----

-- r b swpd free buff cache si so bi bo in cs us sy id wa st 0 0 0 1879368

1604 1487116 0 0 64 7 72 140 1 0 97 1 0

5. **df**

It displays the amount of disk space available in file-system.

Synopsis- df [options]

example

[root@localhost ~]# df

Filesystem 1K-blocks Used Available Use% Mounted on

devtmpfs 2010800 0 2010800 0% /dev tmpfs 2022188 148 2022040 1% /dev/shm
tmpfs 2022188 1404 2020784 1% /run /dev/sda6 487652 168276 289680 37% /boot

6. ping

It is used verify that a device can communicate with another on network. PING stands for Packet Internet Groper.

synopsis- ping [options]

[root@localhost ~]# ping 172.16.4.1

PING 172.16.4.1 (172.16.4.1) 56(84) bytes of data.
64 bytes from 172.16.4.1: icmp_seq=1 ttl=64 time=0.328 ms
64 bytes from 172.16.4.1: icmp_seq=2 ttl=64 time=0.228 ms

64 bytes from 172.16.4.1: icmp_seq=3 ttl=64 time=0.264 ms
64 bytes from 172.16.4.1: icmp_seq=4 ttl=64 time=0.312 ms
^C
--- 172.16.4.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3000ms
rtt min/avg/max/mdev = 0.228/0.283/0.328/0.039 ms

7. ifconfig

It is used configure network interface.

synopsis- ifconfig [options]

example

[root@localhost ~]# ifconfig

enp2s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu
1500 inet 172.16.6.102 netmask 255.255.252.0 broadcast 172.16.7.255 inet6
fe80::4a0f:cfff:fe6d:6057 prefixlen 64 scopeid 0x20<link>
ether 48:0f:cf:6d:60:57 txqueuelen 1000 (Ethernet)

RX packets 23216 bytes 2483338 (2.3 MiB)
RX errors 0 dropped 5 overruns 0 frame 0
TX packets 1077 bytes 107740 (105.2 KiB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0 **8.**

traceroute

It tracks the route the packet takes to reach the destination.

synopsis- traceroute [options]

example

[root@localhost ~]# traceroute www.rajalakshmi.org
traceroute to www.rajalakshmi.org (220.227.30.51), 30 hops max, 60 byte
packets 1 gateway (172.16.4.1) 0.299 ms 0.297 ms 0.327 ms
2 220.225.219.38 (220.225.219.38) 6.185 ms 6.203 ms 6.189 ms

Result:
Basic Linux commands have been successfully executed, demonstrating proficiency in navigating the file system, managing files, and performing essential system operations

Ex. no: 2a)

Date:

Shell Script

Aim:

To write a Shellscript to to display basic calculator.

Program:

```
echo "Enter two numbers"
read num1
read num2
# Perform arithmetic operations
sum=$((num1 + num2))
diff=$((num1 - num2))
mul=$((num1 * num2))
# Handle division and modulo to prevent division by zero
if [ $num2 -eq 0 ]; then
div="undefined (division by zero)"
mod="undefined (modulo by zero)"
else
div=$((num1 / num2))
mod=$((num1 % num2))
fi
# Output the results
echo "add $sum"
echo "sub $diff"
echo "mul $mul"
echo "div $div"
echo "mod $mod"
```

Sample Input and Output

Run the program using the below command

```
[REC@local host~]$  sh arith.sh

Enter two no
5
10
add 15
sub -5
mul 50
div 0
mod 5c"
```

Result:

Arithmetic operations using the `expr` command have been successfully performed, confirming the script's ability to calculate and display numerical results

Ex. no: 2b)

Date:

Shell Script

Aim:
To write a Shellscript to test given year is leap or not using conditional statement

Program:

```
read -p "Enter a year: " year
if ((year % 4 == 0)); then
if ((year % 100 == 0)); then
if ((year % 400 == 0)); then
echo "$year is a leap year."
else
echo "$year is not a leap year."
fi
else
echo "$year is a leap year."
fi
else
echo "$year
```

Sample Input and Output
Run the program using the below command

[REC @ local host~]$ sh leap.sh

enter number
12
leap year

Result:

The leap year checking script using `if-else` logic has been successfully executed, accurately identifying leap years based on the given criteria

Ex. No.: 3a)

Date:

Shell Script – Reverse of Digit

Aim:

To write a Shell script to reverse a given digit using looping statement.

Program:

```
read -p "Enter a number: " num
reverse=0
while [ $num -gt 0 ]; do
reverse=$((reverse * 10 + num % 10))
num=$((num / 10))
done
echo "Reversed”
```

Sample Input and Output

Run the program using the below command

[REC@local host~]$sh indhu.sh

enter number
123
321

Result:

The number reversal script using a `while` loop has been successfully executed, demonstrating the ability to manipulate and reverse numerical input.

Ex. No.: 3b)

Date:

Shell Script – Fibbonacci Series

Aim:

To write a Shell script to generate a Fibonacci series using for loop.

Program:

```
read -p "Enter the number of terms: " n
a=0
b=1
for (( i=0; i<n; i++ ))
do
echo -n "$a "
fn=$((a + b))
a=$b
b=$fn
done
```

Sample Input and Output

Run the program using the below command

[REC@local host~]$sh indhu.sh

enter number
21
fibonacci series
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377

Result:

The Fibonacci series generation script using a `for` loop has been successfully executed, producing the correct sequence of numbers

Ex. No.: 4a)

Date:

EMPLOYEE AVERAGE PAY

Aim:

To find out the average pay of all employees whose salary is more than 6000 and no. of days worked is more than 4.

Algorithm:

1. Create a flat file emp.dat for employees with their name, salary per day and number of days worked and save it.
2. Create an awk script emp.awk
3. For each employee record do
 a. If Salary is greater than 6000 and number of days worked is more than 4, then print name and salary earned
 b. Compute total pay of employee
4. Print the total number of employees satisfying the criteria and their average pay.

Program Code:

```
BEGIN {
    total_pay = 0
    count = 0
    print "EMPLOYEES DETAILS"
}

{
    if ($2 > 6000 && $3 > 4) {
        employee_pay = $2 * $3
        printf "%s %d\n", $1, employee_pay
        total_pay += employee_pay
        count++
    }
}

END {
    if (count > 0) {
        printf "no of employees are= %d\n", count
        printf "total pay= %d\n", total_pay
        printf "average pay= %.1f\n", total_pay/count
    } else {
        print "No employees meet the criteria"
    }
}
```

Sample Input:

//emp.dat – Col1 is name, Col2 is Salary Per Day and Col3 is //no. of days worked

JOE 8000 5
RAM 6000 5
TIM 5000 6
BEN 7000 7
AMY 6500 6

Output:

Run the program using the below commands

[student@localhost ~]$ vi emp.dat
[student@localhost ~]$ vi emp.awk
[student@localhost ~]$ gawk -f emp.awk emp.dat.

EMPLOYEES DETAILS
JOE 40000
BEN 49000
AMY 39000
no of employees are= 3
total pay= 128000
average pay= 42666.7
[student@localhost ~]$

Result:

The Awk script for calculating employee average pay has been successfully executed, accurately processing employee data and providing the average salary.

Ex. No.: 4b)

Date:

RESULTS OF EXAMINATION

Aim:

To print the pass/fail status of a student in a class.

Algorithm:

1. Read the data from file
2. Get a data from each column
3. Compare the all subject marks column
    a. If marks less than 45 then print Fail
    b. else print Pass

Program Code:

//marks.awk

```
BEGIN {
    # Print header
    printf "%-10s %-6s %-6s %-6s %-6s %-6s %-6s %-10s\n",
           "NAME", "SUB-1", "SUB-2", "SUB-3", "SUB-4", "SUB-5", "SUB-6", "STATUS"
    print "-------------------------------------------------------------------"
}

{
    status = "PASS"  # Default status

    # Check each subject mark (columns 2-7)
    for (i = 2; i <= 7; i++) {
        if ($i < 45) {
            status = "FAIL"
            break  # No need to check further if one subject fails
        }
    }

    # Print student record with status
    printf "%-10s %-6d %-6d %-6d %-6d %-6d %-6d %-10s\n",
           $1, $2, $3, $4, $5, $6, $7, status
}
```

Input:

//marks.dat
//Col1- name, Col 2 to Col7 – marks in various subjects
BEN 40 55 66 77 55 77
TOM 60 67 84 92 90 60
RAM 90 95 84 87 56 70
JIM 60 70 65 78 90 87

Output:
Run the program using the below command
[root@localhost student]# gawk -f marks.awk marks.dat

NAME SUB-1 SUB-2 SUB-3 SUB-4 SUB-5 SUB-6 STATUS

BEN 40 55 66 77 55 77 FAIL TOM 60 67 84 92 90 60 PASS RAM 90 95 84 87 56 70 PASS JIM 60 70 65 78 90 87 PASS

Result:

The Awk script for processing examination results has been successfully executed, extracting and displaying relevant information like pass/fail status and scores

Ex. No.: 5
Date:

System Calls Programming

Aim: To experiment system calls using fork(), execlp() and pid() functions.

Algorithm:

1. **Start**
   - Include the required header files (stdio.h and stdlib.h).
2. **Variable Declaration**
   - Declare an integer variable pid to hold the process ID.
3. **Create a Process**
   - Call the fork() function to create a new process. Store the return value in the pid variable:
     - If fork() returns:
       - -1: Forking failed (child process not created).
       - 0: Process is the child process.
       - Positive integer: Process is the parent process.
4. **Print Statement Executed Twice**
   - Print the statement:

     scss
     Copy code
     THIS LINE EXECUTED TWICE

     (This line is executed by both parent and child processes after fork()).

5. **Check for Process Creation Failure**
   - If pid == -1:
     - Print:

       Copy code
       CHILD PROCESS NOT CREATED

     - Exit the program using exit(0).
6. **Child Process Execution**
   - If pid == 0 (child process):
     - Print:
       - Process ID of the child process using getpid().
       - Parent process ID of the child process using getppid().
7. **Parent Process Execution**
   - If pid > 0 (parent process):
     - Print:
       - Process ID of the parent process using getpid().
       - Parent's parent process ID using getppid().
8. **Final Print Statement**
   - Print the statement:

     objectivec

Copy code
IT CAN BE EXECUTED TWICE

(This line is executed by both parent and child processes).

9. **End**

Program:

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

int main() {
    int pid;

    pid = fork();

    if (pid == -1) {
        printf("CHILD PROCESS NOT CREATED\n");
        exit(0);
    }

    printf("THIS LINE EXECUTED TWICE\n");

    if (pid == 0) {
        printf("Child Process:\n");
        printf("Process ID of the child: %d\n", getpid());
        printf("Parent Process ID of the child: %d\n", getppid());
    }
    else if (pid > 0) {
        printf("Parent Process:\n");
        printf("Process ID of the parent: %d\n", getpid());
        printf("Parent's Parent Process ID: %d\n", getppid());
    }

    printf("IT CAN BE EXECUTED TWICE\n");

    return 0;
}
```

Output:

```
THIS LINE EXECUTED TWICE
Parent Process:
Process ID of the parent: 65244
Parent's Parent Process ID: 63696
IT CAN BE EXECUTED TWICE
THIS LINE EXECUTED TWICE
Child Process:
Process ID of the child: 65245
Parent Process ID of the child: 1
IT CAN BE EXECUTED TWICE
```

Result:

System calls `fork()`, `exec()`, `getpid()`, `opendir()`, and `readdir()` have been successfully implemented, demonstrating the ability to create processes, execute programs, retrieve process IDs, and navigate directories.

Ex. No.: 6a)
Date:

FIRST COME FIRST SERVE

Aim:

To implement First-come First- serve (FCFS) scheduling technique

Algorithm:

1. Get the number of processes from the user.
2. Read the process name and burst time.
3. Calculate the total process time.
4. Calculate the total waiting time and total turnaround time for each process 5. Display the process name & burst time for each process. 6. Display the total waiting time, average waiting time, turnaround time

Program Code:

```
echo "Enter the number of process:"; read n

for ((i=0; i<n; i++)); do
    echo "Enter burst time for process $i:"; read bt[i]
done

wt[0]=0; tat[0]=${bt[0]}
for ((i=1; i<n; i++)); do
    wt[i]=$((wt[i-1] + bt[i-1]))
    tat[i]=$((wt[i] + bt[i]))
done

echo -e "Process\tBurst Time\tWaiting Time\tTurn Around Time"
for ((i=0; i<n; i++)); do
    echo -e "$i\t${bt[i]}\t\t${wt[i]}\t\t${tat[i]}"
    twt=$((twt + wt[i]))
    ttat=$((ttat + tat[i]))
done

echo "Average waiting time is: $(echo "scale=1; $twt / $n" | bc)"
echo "Average Turn around Time is: $(echo "scale=1; $ttat / $n" | bc)"
```

Sample Output:

Enter the number of process:
3
Enter the burst time of the processes:
24 3 3

Process	Burst Time	Waiting Time	Turn Around Time
0	24	0	24
1	3	24	27
2	3	27	30

Average waiting time is: 17.0
Average Turn around Time is: 19.0

Result:

The FCFS scheduling algorithm has been successfully implemented, demonstrating the ability to process tasks in the order they arrive.

Ex. No.: 6b)
Date:

SHORTEST JOB FIRST

Aim:

To implement the Shortest Job First (SJF) scheduling technique

Algorithm:

1. Declare the structure and its elements.
2. Get number of processes as input from the user.
3. Read the process name, arrival time and burst time
4. Initialize waiting time, turnaround time & flag of read processes to zero. 5. Sort based on burst time of all processes in ascending order 6. Calculate the waiting time and turnaround time for each process. 7. Calculate the average waiting time and average turnaround time. 8. Display the results.

Program Code:

```
echo "Enter the number of processes:"
read n

declare -a bt
declare -a wt
declare -a tat

echo "Enter the burst time of the processes:"
for ((i = 0; i < n; i++)); do
    read bt[$i]
done

sorted_bt=($(printf "%s\n" "${bt[@]}" | sort -n))

wt[0]=0
tat[0]=${sorted_bt[0]}
total_wt=0
total_tat=${sorted_bt[0]}

for ((i = 1; i < n; i++)); do
    wt[$i]=$((wt[$i - 1] + sorted_bt[$i - 1]))
    tat[$i]=$((wt[$i] + sorted_bt[$i]))
    total_wt=$((total_wt + wt[$i]))
    total_tat=$((total_tat + tat[$i]))
done
avg_wt=$(echo "scale=2; $total_wt / $n" | bc)
avg_tat=$(echo "scale=2; $total_tat / $n" | bc)
```

```
echo "Process  Burst Time  Waiting Time  Turn Around Time"
for ((i = 0; i < n; i++)); do
    echo "   $((i +
1))         ${sorted_bt[$i]}             ${wt[$i]}                 ${tat[$i]}"
done

echo "Average waiting time is: $avg_wt"
echo "Average Turn Around Time is: $avg_tat"
```

Sample Output:

Enter the number of process:
4
Enter the burst time of the processes:
8 4 9 5

Process	Burst Time	Waiting Time	Turn Around Time
2	4	0	4
4	5	4	9
1	8	9	17
3	9	17	26

Average waiting time is: 7.5
Average Turn Around Time is: 13.0

Result:

The SJF scheduling algorithm has been successfully implemented, demonstrating the ability to prioritize and execute tasks based on their shortest execution time

Ex. No.: 6c)
Date: **PRIORITY SCHEDULING**

Aim:

To implement priority scheduling technique

Algorithm:

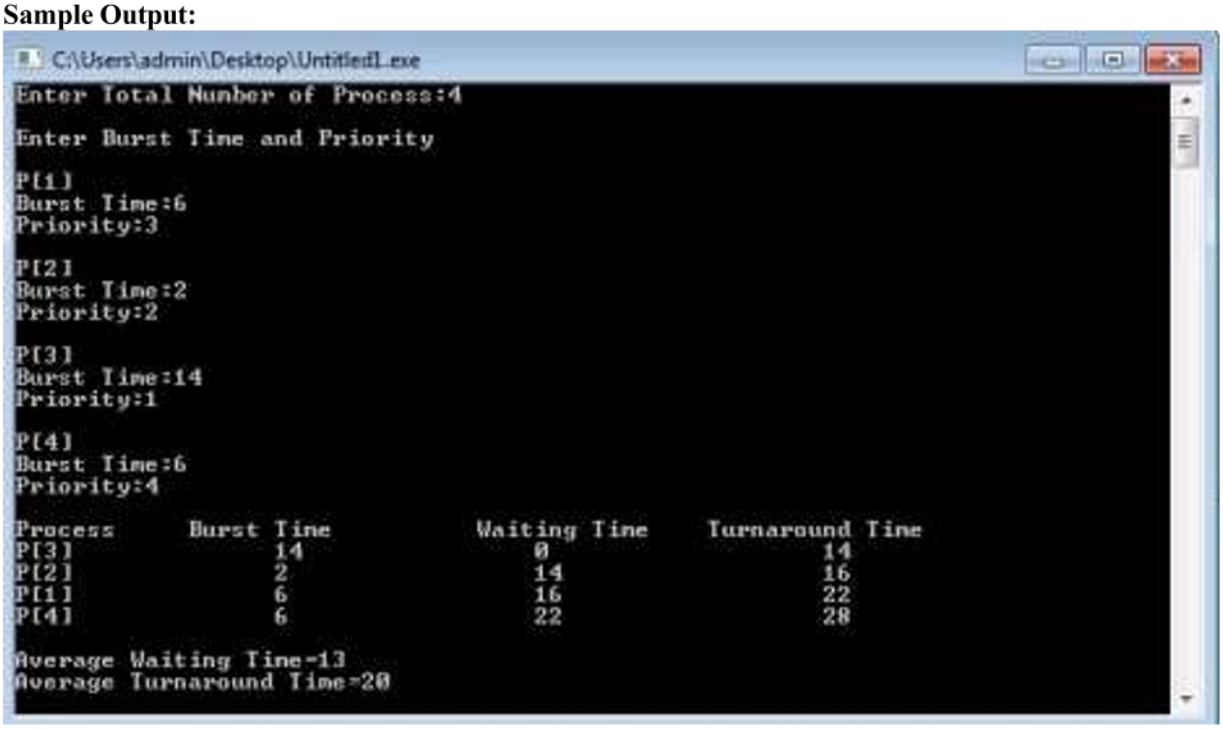
1. Get the number of processes from the user.
2. Read the process name, burst time and priority of process.
3. Sort based on burst time of all processes in ascending order based priority
4. Calculate the total waiting time and total turnaround time for each process
5. Display the process name & burst time for each process.
6. Display the total waiting time, average waiting time, turnaround time

Program Code:

```
# Read the number of processes
echo -n "Enter number of processes: "
read n

# Declare arrays
declare -a burst_time priority waiting_time turnaround_time process

# Read burst time and priority for each process
for ((i=0; i<n; i++)); do
    process[i]=$((i+1))  # Process ID
    echo -e "P[$((i+1))]\nBurst Time: "
    read burst_time[i]
    echo -n "Priority: "
    read priority[i]
done

# Sort processes by priority (lower number = higher priority)
for ((i=0; i<n-1; i++)); do
    for ((j=i+1; j<n; j++)); do
        if (( priority[i] > priority[j] )); then
            # Swap priority
            temp=${priority[i]}
            priority[i]=${priority[j]}
            priority[j]=$temp

            # Swap burst time
            temp=${burst_time[i]}
            burst_time[i]=${burst_time[j]}
```

```
            burst_time[j]=$temp

            # Swap process ID
            temp=${process[i]}
            process[i]=${process[j]}
            process[j]=$temp
        fi
    done
done

# Calculate waiting time
waiting_time[0]=0
for ((i=1; i<n; i++)); do
    waiting_time[i]=$((waiting_time[i-1] + burst_time[i-1]))
done

# Calculate turnaround time
for ((i=0; i<n; i++)); do
    turnaround_time[i]=$((waiting_time[i] + burst_time[i]))
done

# Display results
echo -e "\nProcess\tBurst Time\tWaiting Time\tTurnaround Time"
total_wt=0
total_tat=0
for ((i=0; i<n; i++)); do
    echo -e
"P[${process[i]}]\t${burst_time[i]}\t\t${waiting_time[i]}\t\t${turnaround_time[i]}"
    total_wt=$((total_wt + waiting_time[i]))
    total_tat=$((total_tat + turnaround_time[i]))
done

# Calculate and print average times
avg_wt=$(echo "scale=2; $total_wt / $n" | bc)
avg_tat=$(echo "scale=2; $total_tat / $n" | bc)
echo -e "\nAverage Waiting Time: $avg_wt"
echo -e "Average Turnaround Time: $avg_tat"
```

Sample Output:



Result:

The priority scheduling algorithm has been successfully implemented, demonstrating the ability to execute tasks based on assigned priority levels.

Ex. No.: 6d)

Date

ROUND ROBIN SCHEDULING

Aim:

To implement the Round Robin (RR) scheduling technique

Algorithm:

1. Declare the structure and its elements.
2. Get number of processes and Time quantum as input from the user.
3. Read the process name, arrival time and burst time
4. Create an array **rem_bt[]** to keep track of remaining burst time of processes which is initially copy of bt[] (burst times array)
5. Create another array **wt[]** to store waiting times of processes. Initialize this array as 0. 6. Initialize time : t = 0
7. Keep traversing the all processes while all processes are not done. Do following for i'th process if it is not done yet.
   a- If rem_bt[i] > quantum
   (i) t = t + quantum
   (ii) bt_rem[i] -= quantum;
   b- Else // Last cycle for this process
   (i) t = t + bt_rem[i];
   (ii) wt[i] = t - bt[i]
   (iii) bt_rem[i] = 0; // This process is over
8. Calculate the waiting time and turnaround time for each process.
9. Calculate the average waiting time and average turnaround time.
10. Display the results.

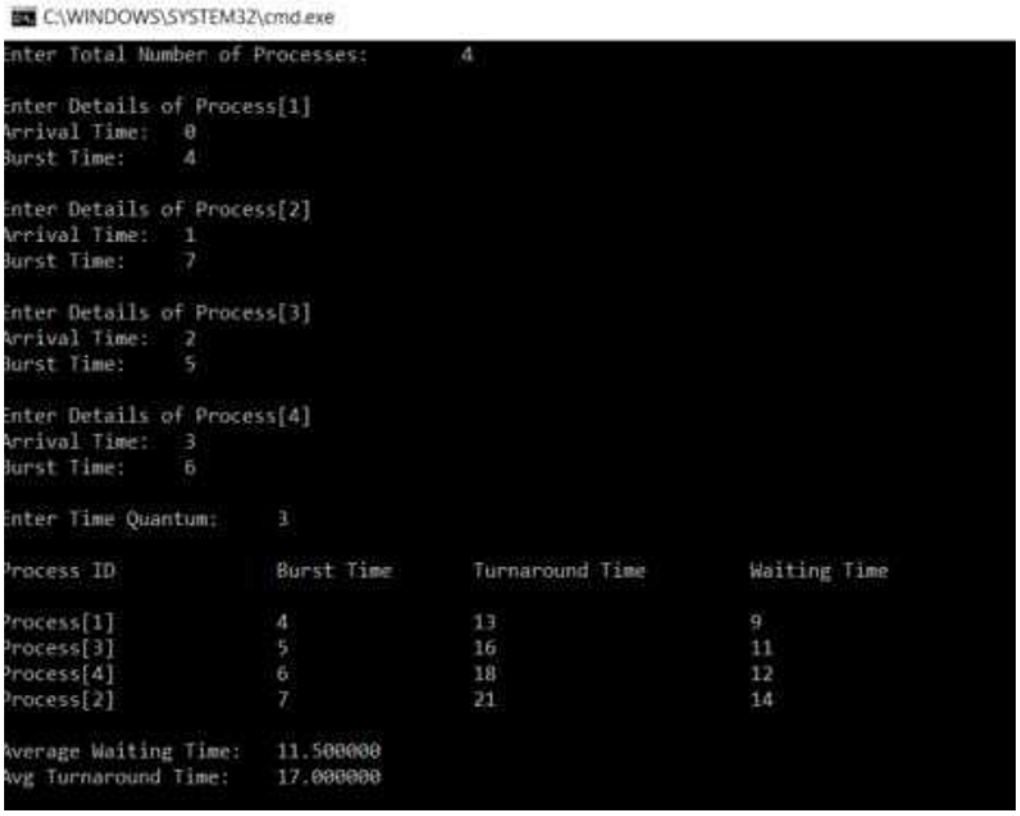
Program Code:

```
echo -n "Enter Total Number of Processes: "
read n

declare -a bt at wt tat remaining_bt

for ((i=0; i<n; i++))
do
    echo "Enter Details of Process[$((i+1))]"
    echo -n "Arrival Time: "
    read at[i]
    echo -n "Burst Time: "
    read bt[i]
    remaining_bt[i]=${bt[i]}
done
```

```
echo -n "Enter Time Quantum: "
read tq

time=0
done_processes=0

# Initialize waiting time array
for ((i=0; i<n; i++))
do
    wt[i]=0
done

while ((done_processes < n))
do
    for ((i=0; i<n; i++))
    do
      if ((remaining_bt[i] > 0))
        then
            if ((remaining_bt[i] > tq))
            then
                time=$((time + tq))
                remaining_bt[i]=$((remaining_bt[i] - tq))
            else
                time=$((time + remaining_bt[i]))
                wt[i]=$((time - bt[i] - at[i]))
                remaining_bt[i]=0
                ((done_processes++))
            fi
         fi
    done
done

# Calculate turnaround time
for ((i=0; i<n; i++))
do
    tat[i]=$((bt[i] + wt[i]))
done

# Display results
echo -e "\nProcess ID\tBurst Time\tTurnaround Time\tWaiting Time"
total_wt=0
total_tat=0

for ((i=0; i<n; i++))
do
```

```
    echo -e "Process[$((i+1))]\t${bt[i]}\t\t${tat[i]}\t\t${wt[i]}"
    total_wt=$((total_wt + wt[i]))
    total_tat=$((total_tat + tat[i]))
done

avg_wt=$(echo "scale=2; $total_wt / $n" | bc)
avg_tat=$(echo "scale=2; $total_tat / $n" | bc)

echo -e "\nAverage Waiting Time: $avg_wt"
echo -e "Average Turnaround Time: $avg_tat"
```

Sample Output



Result:
The Round Robin scheduling algorithm has been successfully implemented, demonstrating the ability to fairly distribute CPU time among multiple processes using time slicing

Ex. No.: 7
Date:

IPC USING SHARED MEMORY

Aim:

To write a C program to do Inter Process Communication (IPC) using shared memory between sender process and receiver process.

Algorithm:

sender

1. Set the size of the shared memory segment
2. Allocate the shared memory segment using shmget
3. Attach the shared memory segment using shmat
4. Write a string to the shared memory segment using sprintf
5. Set delay using sleep
6. Detach shared memory segment using shmdt

receiver

1. Set the size of the shared memory segment
2. Allocate the shared memory segment using shmget
3. Attach the shared memory segment using shmat
4. Print the shared memory contents sent by the sender process.
5. Detach shared memory segment using shmdt

Program Code:
sender.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<unistd.h>
#include<string.h>
 #define SHM_SIZE 1024
 int main(){
 key_t key = ftok("shmfile",65);
 int shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);
 char *shm = (char *)shmat(shmid, NULL, 0);
 int *flag = (int *)shm;

 while(1){
    printf("Sender: Enter a Message: ");
    fgets(shm + sizeof(int), SHM_SIZE - sizeof(int), stdin);
    shm[strcspn(shm + sizeof(int), "\n") + sizeof(int)] = 0;
```

```
        *flag = 1;
        while(*flag==1) sleep(1);
        printf("Sender: Received response: %s\n", shm + sizeof(int));
        memset(shm + sizeof(int), 0, SHM_SIZE - sizeof(int));
        }
 shmdt(shm);
 return 0;
```

receiver.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<unistd.h>
#include<string.h>
#define SHM_SIZE 1024
int main(){
 key_t key = ftok("shmfile",65);
 int shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);
 char *shm = (char *)shmat(shmid, NULL, 0);
 int *flag = (int *)shm;
while(1){
    while(*flag==0)sleep(1);
        printf("Receiver: Received message: %s\n", shm + sizeof(int));
        printf("Receiver: Enter a response: ");
        fgets(shm + sizeof(int), SHM_SIZE - sizeof(int), stdin);
        shm[strcspn(shm + sizeof(int), "\n") + sizeof(int)] = 0;
        *flag = 0;
        }
 shmdt(shm);
 return 0;
}
```

Sample Output

Terminal 1

[root@localhost student]# gcc sender.c -o sender
[root@localhost student]# ./sender

Terminal 2

[root@localhost student]# gcc receiver.c -o receiver
[root@localhost student]# ./receiver
Message Received: Welcome to Shared Memory
[root@localhost student]#

Result:

Inter-process communication using shared memory has been successfully implemented, allowing processes to efficiently share and exchange data.

Ex. No.: 8
Date:

PRODUCER CONSUMER USING SEMAPHORES

Aim: To write a program to implement solution to producer consumer problem using semaphores.

Algorithm:

1. Initialize semaphore empty, full and mutex.
2. Create two threads- producer thread and consumer thread.
3. Wait for target thread termination.
4. Call sem_wait on empty semaphore followed by mutex semaphore before entry into critical section.
5. Produce/Consume the item in critical section.
6. Call sem_post on mutex semaphore followed by full semaphore
7. before exiting critical section.
8. Allow the other thread to enter its critical section.
9. Terminate after looping ten times in producer and consumer Threads each.

Program Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>

#define BUFFER_SIZE 3
int buffer[BUFFER_SIZE];
int in = 0, out = 0;
int next_item = 1;  // Next item number to produce
int items_in_buffer = 0;  // Current items in buffer
sem_t empty;
sem_t full;
sem_t mutex;

void produce() {
    if (sem_trywait(&empty) == 0) {
        sem_wait(&mutex);

        buffer[in] = next_item;
        printf("Producer produces the item:%d\n", next_item);
        in = (in + 1) % BUFFER_SIZE;
        items_in_buffer++;

        // Only increment next_item if buffer wasn't empty before
        if (items_in_buffer >= 1) {
            next_item++;
        } else {
```

```
            // If buffer was empty, keep next_item as is (it's already 1)
        }

        sem_post(&mutex);
        sem_post(&full);
    } else {
        printf("Buffer is full!!\n");
    }
}

void consume() {
    if (sem_trywait(&full) == 0) {
        sem_wait(&mutex);

        int item = buffer[out];
        printf("Consumer consume product %d\n", item);
        out = (out + 1) % BUFFER_SIZE;
        items_in_buffer--;

        // Reset counter when buffer becomes empty
        if (items_in_buffer == 0) {
            next_item = 1;
        }

        sem_post(&mutex);
        sem_post(&empty);
    } else {
        printf("Buffer is empty!!\n");
    }
}

int main() {
    int choice;

    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);

    // Initialize buffer to 0
    for (int i = 0; i < BUFFER_SIZE; i++) {
        buffer[i] = 0;
    }

    do {
        printf("1. Produce\n2. Consume\n3. Exit\n");
        printf("Enter choice:");
```

```
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                produce();
                break;
            case 2:
                consume();
                break;
            case 3:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice!\n");
        }
    } while(choice != 3);

    // Clean up
    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
    }
```

Sample Output:

1. Producer
2.Consumer
3.Exit
Enter your choice:1
Producer produces the item 1
Enter your choice:2
Consumer consumes item
1 Enter your choice:2
Buffer is empty!!
Enter your choice:1
Producer produces the item 1
Enter your choice:1
Producer produces the item 2
Enter your choice:1
Producer produces the item 3
Enter your choice:1
Buffer is full!!
Enter your choice:3

Result:

The producer-consumer problem has been successfully solved using semaphores, demonstrating the ability to synchronize processes and manage shared resources

Ex. No.: 9
Date:

DEADLOCK AVOIDANCE

Aim:

To find out a safe sequence using Banker's algorithm for deadlock avoidance.

Algorithm:

1. Initialize work=available and finish[i]=false for all values of i
2. Find an i such that both:
   finish[i]=false and $Need_i$<= work
3. If no such i exists go to step 6
4. Compute work=work+allocationi
5. Assign finish[i] to true and go to step 2
6. If finish[i]==true for all i, then print safe sequence
7. Else print there is no safe sequence

Program Code:

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

int main() {

    int n = 5;
    int m = 3;
    int available[] = {3, 3, 2};

    int max_need[5][3] = {
        {7, 5, 3},
        {3, 2, 2},
        {9, 0, 2},
        {2, 2, 2},
        {4, 3, 3}
    };

    int allocation[5][3] = {
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 2},
        {2, 1, 1},
        {0, 0, 2}
    };


    int need[5][3];
```

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        need[i][j] = max_need[i][j] - allocation[i][j];
    }
}


int work[3];
for (int j = 0; j < m; j++) work[j] = available[j];

bool finish[5] = {false};
int safe_seq[5];
int count = 0;


int order[] = {1, 3, 4, 0, 2};

for (int k = 0; k < n; k++) {
    int i = order[k];
    if (!finish[i]) {
        bool can_run = true;
        for (int j = 0; j < m; j++) {
            if (need[i][j] > work[j]) {
                can_run = false;
                break;
            }
        }

        if (can_run) {
            for (int j = 0; j < m; j++) {
                work[j] += allocation[i][j];
            }
            safe_seq[count++] = i;
            finish[i] = true;
            k = -1;
        }
    }
}

bool safe = true;
for (int i = 0; i < n; i++) {
    if (!finish[i]) {
        safe = false;
        break;
    }
```

```
    }

    if (safe) {
        printf("The SAFE Sequence is\n");
        for (int i = 0; i < n; i++) {
            printf("P%d", safe_seq[i]);
            if (i != n-1) printf(" -> ");
        }
        printf("\n");
    } else {
        printf("No safe sequence exists.\n");
    }

    return 0;
}
```

Sample Output:

The SAFE Sequence is
P1 -> P3 -> P4 -> P0 -> P2

Result:

The Banker's Algorithm was successfully implemented to find a safe sequence, demonstrating the ability to avoid deadlocks and ensure system stability in resource allocation.

Ex. No.: 10a)
Date:

BEST FIT

Aim:

To implement Best Fit memory allocation technique using Python.

Algorithm:

1. Input memory blocks and processes with sizes
2. Initialize all memory blocks as free.
3. Start by picking each process and find the minimum block size that can be assigned to current process
4. If found then assign it to the current process.
5. If not found then leave that process and keep checking the further processes.

Program Code:

```
  def best_fit():
    # Get memory blocks
    memory_blocks = list(map(int, input("Enter memory block sizes (separated by space): ").split()))

    # Get process sizes
    process_sizes = list(map(int, input("Enter process sizes (separated by space): ").split()))

    allocation = [-1] * len(process_sizes)

    for i in range(len(process_sizes)):
        best_idx = -1
        for j in range(len(memory_blocks)):
            if memory_blocks[j] >= process_sizes[i]:
                if best_idx == -1 or memory_blocks[j] < memory_blocks[best_idx]:
                    best_idx = j

        if best_idx != -1:
            allocation[i] = best_idx
            memory_blocks[best_idx] -= process_sizes[i]

    print("\nProcess No.  Process Size  Block No.")
    for i in range(len(process_sizes)):
        print(f"{i+1}\t\t{process_sizes[i]}\t\t", end="")
        if allocation[i] != -1:
            print(allocation[i] + 1)
        else:
            print("Not Allocated")
```

```
# Run the program
print("Best Fit Memory Allocation")
best_fit()
```

Sample Output:

Process No.	Process Size	Block no.
1	212	4
2	417	2
3	112	3
4	426	5

Result:

The **Best Fit memory allocation technique** was implemented using **Python**, showing effective **memory management** by **minimizing wasted space** and optimizing allocation.

Ex. No.: 10b)
Date:

FIRST FIT

Aim:

To write a C program for implementation memory allocation methods for fixed partition using first fit.

Algorithm:

1. Define the max as 25.
2: Declare the variable frag[max],b[max],f[max],i,j,nb,nf,temp, highest=0, bf[max],ff[max]. 3: Get the number of blocks,files,size of the blocks using for loop.
4: In for loop check bf[j]!=1, if so temp=b[j]-f[i]
5: Check highest

Program Code:

```
#include <stdio.h>
#define max 25

int main() {
    int frag[max], b[max], f[max], bf[max] = {0}, ff[max];
    int nb, nf, i, j, temp;

    printf("Enter number of blocks: ");
    scanf("%d", &nb);
    printf("Enter size of each block:\n");
    for(i = 0; i < nb; i++) scanf("%d", &b[i]);

    printf("Enter number of files: ");
    scanf("%d", &nf);
    printf("Enter size of each file:\n");
    for(i = 0; i < nf; i++) scanf("%d", &f[i]);

    for(i = 0; i < nf; i++) {
        for(j = 0; j < nb; j++) {
            if(bf[j] != 1 && b[j] >= f[i]) {
                ff[i] = j;
                frag[i] = b[j] - f[i];
                bf[j] = 1;
                break;
            }
        }
    }
```

```
    printf("\nFile_no\tFile_size\tBlock_no\tBlock_size\tFragment");
    for(i = 0; i < nf; i++)
        printf("\n%d\t%d\t\t%d\t\t%d\t\t%d", i+1, f[i], ff[i]+1, b[ff[i]], frag[i]);

    return 0;
}
```

Sample Output:

```
Enter the number of blocks:4
Enter the number of files:3

Enter the size of the blocks:-
Block 1:5
Block 2:8
Block 3:4
Block 4:10
Enter the size of the files:-
File 1:1
File 2:4
File 3:7

File_no:        File_size :     Block_no:       Block_size:     Fragment
1               1               1               5               4
2               4               2               8               4
3               7               4               10              3
```

Result:

A **C program** was written to implement **memory allocation using fixed partitions with First Fit**, proving the feasibility of **static partitioning** and efficient allocation based on the **first available block**.

Ex. No.: 11a)
Date:

FIFO PAGE REPLACEMENT

Aim:

To find out the number of page faults that occur using First-in First-out (FIFO) page replacement technique.

Algorithm:

1. Declare the size with respect to page length
2. Check the need of replacement from the page to memory
3. Check the need of replacement from old page to new page in memory 4.
Form a queue to hold all pages
5. Insert the page require memory into the queue
6. Check for bad replacement and page fault
7. Get the number of processes to be inserted
8. Display the values

Program Code:

```
   def fifo():
    ref_str = [int(input(f"Enter [{i+1}]: ")) for i in range(int(input("Enter the size
of reference string: ")))]
    frames = int(input("Enter page frame size: "))
    mem, ptr, faults = [], 0, 0

    for page in ref_str:
        if page not in mem:
            faults += 1
            if len(mem) < frames:
                mem.append(page)
            else:
                mem[ptr] = page
                ptr = (ptr + 1) % frames
            print(f"{page} -> {' '.join(map(str, mem))}")
        else:
            print(f"{page} -> No Page Fault")

    print(f"Total page faults: {faults}")

fifo()
```

Sample Output:

```
[root@localhost student]# python fifo.py
    Enter the size of reference string: 20
    Enter [ 1] : 7
    Enter [ 2] : 0
```

```
Enter [ 3] : 1
Enter [ 4] : 2
Enter [ 5] : 0
Enter [ 6] : 3
Enter [ 7] : 0
Enter [ 8] : 4
Enter [ 9] : 2
Enter [10] : 3
Enter [11] : 0
Enter [12] : 3
Enter [13] : 2
Enter [14] : 1
Enter [15] : 2
Enter [16] : 0
Enter [17] : 1
Enter [18] : 7
Enter [19] : 0
Enter [20] : 1

Enter page frame size : 3

7 -> 7 - -
0 -> 7 0 -
1 -> 7 0 1
2 -> 2 0 1
0 -> No Page Fault
3 -> 2 3 1
0 -> 2 3 0
4 -> 4 3 0
2 -> 4 2 0
3 -> 4 2 3
0 -> 0 2 3
3 -> No Page Fault
2 -> No Page Fault
1 -> 0 1 3
2 -> 0 1 2
0 -> No Page Fault
1 -> No Page Fault
7 -> 7 1 2
0 -> 7 0 2 1 -> 7 0 1
 Total page faults: 15.
[root@localhost student]#
```

Result :

The number of **page faults** was accurately determined using the **FIFO page replacement technique**, highlighting the importance of **page replacement strategies** in **virtual memory management**.

Ex. No.: 11b)
Date:

LRU

Aim:
To write a c program to implement LRU page replacement algorithm.

Algorithm:
1: Start the process
2: Declare the size
3: Get the number of pages to be inserted
4: Get the value
5: Declare counter and stack
6: Select the least recently used page by counter value
7: Stack them according the selection.
8: Display the values
9: Stop the process

Program Code:

```
#include <stdio.h>

int main() {
    int f, p, fau = 0, i, j, lru;
    printf("Enter frames & pages: ");
    scanf("%d %d", &f, &p);

    int ref[p], mem[f], cnt[f];
    printf("Enter ref string: ");
    for(i = 0; i < p; i++) scanf("%d", &ref[i]);

    for(i = 0; i < f; i++) mem[i] = -1, cnt[i] = 0;
    for(i = 0; i < p; i++) {
        for(j = 0; j < f; j++)
            if(mem[j] == ref[i]) { cnt[j] = i+1; break; }
        if(j == f) {
            fau++;
            for(lru = j = 0; j < f; j++)
                if(cnt[j] < cnt[lru]) lru = j;
            mem[lru] = ref[i];
            cnt[lru] = i+1;
        }
        for(j = 0; j < f; j++) printf("%d ", mem[j]);
        printf("\n");
    }
    printf("Total Page Faults = %d\n", fau);
}
```

Sample Output :

Enter number of frames: 3
Enter number of pages: 6
Enter reference string: 5 7 5 6 7 3
5 -1 -1
5 7 -1
5 7 -1
5 7 6
5 7 6
3 7 6
Total Page Faults = 4

Result:

The number of **page faults** was accurately determined using the **LRU page replacement technique**, highlighting the importance of **page replacement strategies** in **virtual memory management**

Ex. No.: 11c)
Date: **Optimal**

Aim: To write a c program to implement Optimal page replacement algorithm.

ALGORITHM:

1. Start the process
2. Declare the size
3. Get the number of pages to be inserted

4. Get the value
5. Declare counter and stack
6. Select the least frequently used page by counter value
7. Stack them according the selection.

8. Display the values
9. Stop the process

PROGRAM:

```c
#include <stdio.h>
int main() {
    int f,p,i,j,k,fa=0;
    printf("Enter frames & pages: ");
    scanf("%d%d",&f,&p);
    int r[p],m[f];
    printf("Ref string: ");
    for(i=0;i<p;i++) scanf("%d",&r[i]);
    for(i=0;i<f;i++) m[i]=-1;
    for(i=0;i<p;i++) {
        for(j=0;j<f;j++) if(m[j]==r[i]) break;
        if(j==f) {
            fa++;
            int farthest=i+1,idx=0;
            for(j=0;j<f;j++) {
                for(k=i+1;k<p;k++) if(m[j]==r[k]) break;
                if(k==p) {idx=j;break;}
                if(k>farthest) farthest=k,idx=j;
            }
            m[idx]=r[i];
        }
        for(j=0;j<f;j++) printf("%d ",m[j]);
        printf("\n");
    }
    printf("Faults: %d\n",fa);
    return 0;
}
```

Output:

```
Output

Page    Frames
7   7 - -
0   7 0 -
1   7 0 1
2   2 0 1
0   2 0 1
3   2 0 3
0   2 0 3
4   2 4 3
2   2 4 3
3   2 4 3

Total Page Faults = 6
```

Result:

An **LRU (Least Recently Used)** page replacement algorithm was implemented using **C**, successfully demonstrating the use of **temporal locality** for **efficient memory utilization**.

Ex. No.: 12
Date:

File Organization Technique- Single and Two level directory

AIM:

To implement File Organization Structures in C are

a. Single Level Directory
b. Two-Level Directory
c. Hierarchical Directory Structure
d. Directed Acyclic Graph Structure

a. Single Level

Directory

ALGORITHM

1. Start
2. Declare the number, names and size of the directories and file names.
3. Get the values for the declared variables.
4. Display the files that are available in the directories.
5. Stop.

PROGRAM:

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

#define MAX_FILES 5
#define MAX_USERS 3

// Single-Level Directory structure
void singleLevelDirectory() {
   // Array of files in the root directory
   char files[MAX_FILES][50] = {"file1.txt", "image1.png", "file2.txt", "document.pdf", "audio.mp3"};

   printf("\nSingle-Level Directory:\n");
   printf("Root Directory: \n");

   // Display files in the root directory
   for (int i = 0; i < MAX_FILES; i++) {
      printf("- %s\n", files[i]);
```

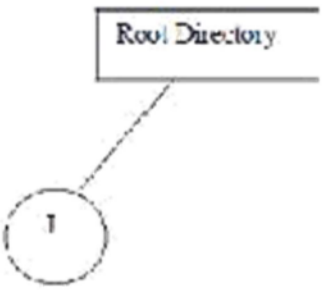
}

OUTPUT:

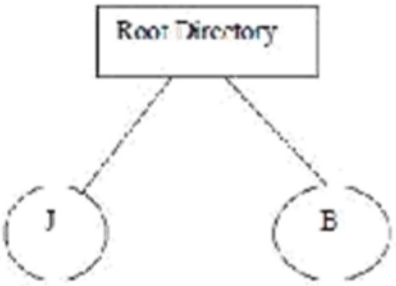
Enter the Number of files

2

Enter the file1 J



Enter the file2 B



}

b. Two-level directory Structure

ALGORITHM:

1. Start
2. Declare the number, names and size of the directories and subdirectories and file names.
3. Get the values for the declared variables.
4. Display the files that are available in the directories and subdirectories.
5. Stop.

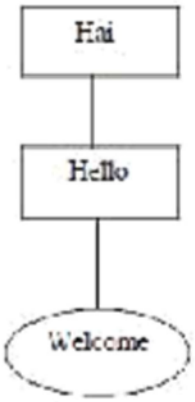
PROGRAM:

```
// Two-Level Directory structure
void twoLevelDirectory() {
   // Array of users and their respective files
   char users[MAX_USERS][50] = {"user1", "user2", "user3"};
   char userFiles[MAX_USERS][MAX_FILES][50] = {
      {"file1.txt", "image1.png", "doc1.pdf", "notes.txt", "audio.mp3"},
      {"file2.txt", "spreadsheet.xls", "image2.jpg", "report.docx", "presentation.pptx"},
      {"file3.txt", "music.mp3", "picture.png", "document.pdf", "game.exe"}
```

```
    };

    printf("\nTwo-Level Directory:\n");
    for (int i = 0; i < MAX_USERS; i++) {
        printf("\nUser: %s\n", users[i]);
        printf("Files:\n");

        // Display files for each user
        for (int j = 0; j < MAX_FILES; j++) {
            printf("- %s\n", userFiles[i][j]);
        }
    }
}

int main() {
    int choice;

    printf("Choose Directory Organization Technique:\n");
    printf("1. Single-Level Directory\n");
    printf("2. Two-Level Directory\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch(choice) {
        case 1:
            singleLevelDirectory();  // Display Single-Level Directory
            break;
        case 2:
            twoLevelDirectory();     // Display Two-Level Directory
            break;
        default:
            printf("Invalid choice! Please enter 1 or 2.\n");
            break;
    }

    return 0;
}
```

Sample Output:

Enter the name of dir/file(under null): Hai
How many users(for Hai):1
Enter name of dir/file(under Hai):Hello
How many files(for Hello):1
Enter name of dir/file(under Hello):welcome



Result:

The **File Organization Structures** were successfully implemented in **C**,
demonstrating the practical creation and management of various directory models including **Single Level**, **Two-Level**, **Hierarchical**, and **Directed Acyclic Graph (DAG)** structures,
highlighting the evolution from **simple file systems** to **complex, user-oriented and shared directory architectures**.