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: . . . . . VIJAY M . . . . . .

Register No. : . . . . . .231501182 . . . . . .

Year / Branch / Section: . . . . . .II/B.Tech/AIML- 'FB' . . . . . .

Semester: . . . . . 4. . . . . .

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

INDEX

EXP.NO	Date	Title	Page No	Signature
1a	24/1/25	Installation and Configuration of Linux	4-5	
1b	24/1/25	Basic Linux Commands	6-17	
2	5/2/25	Study of Unix editors : sed,vi,emacs		
3 a)	5/2/25	Shell script a) Arithmetic Operation -using expr command b) Check leap year using if-else	18-21	
3 b)	7/2/25	a) Reverse the number using while loop b) Fibonacci series using for loop	22-25	
4	12/2/25	Text processing using Awk script a) Employee average pay b) Results of an examination	26-30	
5	12/2/25	System calls –fork(), exec(), getpid(),opendir(), readdir()	31-33	
6a	14/2/25	FCFS	34-37	
6b	5/3/25	SJF	38-44	
6c	5/3/25	Priority	45-48	
6d	5/3/25	Round Robin	49-54	
7.	7/3/25	Inter-process Communication using Shared Memory	55-59	

8	7/3/25	Producer Consumer using Semaphores	60-62	
9	19/3/25	Bankers Deadlock Avoidance algorithms	63-66	

10 a	19/3/25	Best Fit	67-68	
10 b	19/3/25	First Fit	69-71	
11a	26/3/25	FIFO	72-77	
11b	26/3/25	LRU	78-81	
11c	26/3/25	Optimal	82-85	
12	28/3/25	File Organization Technique- single and Two level directory	86-96	

Ex No: 1a)

Date: 24/1/25

INSTALLATION AND CONFIGURATION OF LINUX

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 Puppy Linux

Output:

Result :

Thus,installation and configuration of linux is done successfully.

Ex No: 1b)
Date: 24/1/25

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	Purpos	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

Categor	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 permissior	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**

EXAPLE: $ 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

18
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 6.18ms

Result:

Thus ,the basic linux commands program is executed successfully

Ex. no: 2a)

Date: 5/2/25

SHELL SCRIPT

Aim:

To write a Shellscript to to display basic calculator. Program:

```
#!/bin/bash

echo "Enter first number: "

read a

echo "Enter second number: "

read b

echo "Select operation:"

echo "1. Addition"

echo "2. Subtraction"

echo "3. Multiplication"

echo "4. Division"

echo "5. Modulus"
read choice
case $choice in

1) result=$((a + b))

echo "Addition = $result" ;;

2) result=$((a - b))

echo "Subtraction = $result" ;;

3) result=$((a * b))

echo "Multiplication = $result" ;;

4) if [ $b -ne 0 ]

then
```

```
result=$((a / b))

echo "Division = $result"

else

echo "Division by zero not allowed"

fi ;;

5) result=$((a % b))

echo "Modulus = $result" ;;

*) echo "Invalid choice" ;;

Esac
```

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:

Thus the basic calculator program is executed successfully

Ex. no: 2b)

Date: 5/2/25

SHELL SCRIPT

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

Program:

```
#!/bin/bash

echo "Enter a year:"
read year

if (( year % 400 == 0 )); then
echo "$year is a Leap Year"
elif (( year % 100 == 0 )); then
echo "$year is NOT a Leap Year"
elif (( year % 4 == 0 )); then
echo "$year is a Leap Year"
else
echo "$year is NOT a Leap Year"
fi
```

Sample Input and Output
Run the program using the below command [REC @ local host~]$ sh leap.sh

```
enter number
12
leap year
```

Result:

Thus the leap year program using linux commands is executed successfully

Ex. No.: 3a)

Date: 7/2/25

Shell Script – Reverse of Digit

Aim:

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

Program:

```
#!/bin/bash
echo "Enter a number:"
read num
reverse=0
while [ $num -gt 0 ]
do
remainder=$((num % 10))
reverse=$((reverse * 10 + remainder))
num=$((num / 10))
done
echo "Reversed number: $reverse"
```

Sample Input and Output

Run the program using the below command [REC@local host~]$sh indhu.sh

enter number
123
321

Result:

Thus the Shell script to reverse a given digit using looping is executed successfully

Ex. No.: 3b)

Date: 7/2/25

Shell Script – Fibbonacci Series

Aim:

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

Program:

```
#!/bin/bash
echo "Enter the number of terms:"
read n
a=0
b=1
echo "Fibonacci series:"
for (( i=0; i<n; i++ ))
do
echo -n "$a "
fn=$((a + b))
a=$b
b=$fn
done
echo
```

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:

Thus the fibonacci program using linux is executed successfully

Ex. No.: 4a)

Date: 12/2/25

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:

emp.data

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

emp.awk

```
BEGIN{total=0;count=0}
$2>6000 && $3>4 {
pay=$2*$3
print $1, pay
```

```
total+=pay

count+=1

}

END {

print "no of employees are=", count

print "total pay=", total

if(count>0)

print "average pay=", total/count

else

print "average pay= 0"

}
```

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:

Thus the program to find out the average pay of all employees whose salary is more than 6000 and no. of days worked is more than 4 is executed successfully

Ex. No.: 4b)

Date: 12/2/25

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 "NAME SUB1 SUB2 SUB3 SUB4 SUB5 SUB6 STATUS"}

{

status="PASS"

for(i=2;i<=7;i++)

if($i<45) {status="FAIL";break}

print $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:

Thus,the program to print the pass/fail status of a student in a class is executed successfully

Ex. No.: 5
Date: 12/2/25

System Calls Programming

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

Algorithm:

1. Start
o Include the required header files (stdio.h and stdlib.h).
2. Variable Declaration
o Declare an integer variable pid to hold the process ID.
3. Create a Process
o 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
o 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
o If pid == -1:
▪ Print:

Copy code
CHILD PROCESS NOT CREATED

▪ Exit the program using exit(0).
6. Child Process Execution
o 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
o 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
o Print the statement:

Objective
Copy code
IT CAN BE EXECUTED TWICE

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

9. End

Program:

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

int main() {
pid_t pid = fork();  // Create a new process

if (pid < 0) {
// If fork fails
perror("Fork failed");
return 1;
}

if (pid == 0) {
// Child process
printf("Child process: PID = %d, Parent PID = %d\n", getpid(), getppid());

// Execute a command using execlp
execlp("ls", "ls", "-l", NULL);  // List files in the current directory
// If execlp fails
perror("execlp failed");
return 1;
} else {
// Parent process
wait(NULL);  // Wait for child process to complete
printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);
}

return 0;
}
```

Output:

Child process: PID = 12345, Parent PID = 12344

total 4

drwxrwxrwx 2 user user 4096 Apr 25 12:00 folder1

-rwxrwxrwx 1 user user 1732 Apr 25 12:00 testfile

Parent process: PID = 12344, Child PID = 12345

Result:

Thus the System Calls Programming using linux is executed successfully

Ex. No.: 6a)
Date: 14/2/25

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:

```
#include <stdio.h>

struct Process {
int pid;            // Process ID
int arrival_time; // Arrival time of the process
int burst_time;  // Burst time (time needed by the process to complete)
int waiting_time; // Waiting time for the process
int turn_around_time; // Turnaround time (waiting time + burst time)
};

void calculate_waiting_time(struct Process[], int, int);
void calculate_turnaround_time(struct Process[], int);
void find_average_times(struct Process[], int);

int main()
{ int n;

// Get the number of processes
printf("Enter the number of processes: ");
```

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

struct Process p[n];

// Input process details
for (int i = 0; i < n; i++) {
printf("\nEnter details for process %d:\n", i + 1);
p[i].pid = i + 1;  // Process ID
printf("Arrival time: ");
scanf("%d", &p[i].arrival_time);
printf("Burst time: ");
scanf("%d", &p[i].burst_time);
}

// FCFS Scheduling
calculate_waiting_time(p, n, 0);  // Calculate waiting times
calculate_turnaround_time(p, n);       // Calculate turnaround times
find_average_times(p, n);       // Find and print average times

return 0;
}

void calculate_waiting_time(struct Process p[], int n, int start_time)
{ p[0].waiting_time = 0; // First process has no waiting time

// Calculate waiting time for each process
for (int i = 1; i < n; i++) {
p[i].waiting_time = p[i - 1].waiting_time + p[i - 1].burst_time;
}
}
```

```
void calculate_turnaround_time(struct Process p[], int n) {
// Calculate turnaround time for each process
for (int i = 0; i < n; i++) {
p[i].turn_around_time = p[i].waiting_time + p[i].burst_time;
}
}

void find_average_times(struct Process p[], int n)
{  float total_waiting_time = 0, total_turnaround_time =
0;

// Display individual process times and calculate totals
printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
for (int i = 0; i < n; i++) {
printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].arrival_time, p[i].burst_time,
p[i].waiting_time, p[i].turn_around_time);
total_waiting_time += p[i].waiting_time;
total_turnaround_time += p[i].turn_around_time;
}

// Calculate and display average waiting time and turnaround time
printf("\nAverage waiting time: %.2f", total_waiting_time / n);
printf("\nAverage turnaround time: %.2f", total_turnaround_time / n);
}
```

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

Result:

Thus the linux program to implement First-come First- serve (FCFS) scheduling technique is executed successfully

Ex. No.: 6b)
Date: 5/3/25

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:

```
#include <stdio.h>



struct Process {

int pid;            // Process ID

int arrival_time; // Arrival time of the process

int burst_time;  // Burst time (time needed by the process to complete)

int waiting_time; // Waiting time for the process

int turn_around_time; // Turnaround time (waiting time + burst time)

};



// Function prototypes

void calculate_waiting_time(struct Process[], int);
```

```
void calculate_turnaround_time(struct Process[], int);

void find_average_times(struct Process[], int);

void sort_by_burst_time(struct Process[], int);


int main()

{ int n;


// Get the number of processes

printf("Enter the number of processes: ");

scanf("%d", &n);


struct Process p[n];


// Input process details

for (int i = 0; i < n; i++) {

printf("\nEnter details for process %d:\n", i + 1);

p[i].pid = i + 1;  // Process ID

printf("Arrival time: ");

scanf("%d", &p[i].arrival_time);
```

```
printf("Burst time: ");
scanf("%d", &p[i].burst_time);
}

// Sort processes by burst time (non-preemptive SJF)
sort_by_burst_time(p, n);

// Calculate waiting and turnaround times
calculate_waiting_time(p, n);
calculate_turnaround_time(p, n);

// Find and display average times
find_average_times(p, n);

return 0;
}

// Sort processes by burst time (non-preemptive SJF)
void sort_by_burst_time(struct Process p[], int n) {
```

```
struct Process temp;

for (int i = 0; i < n - 1; i++)

{ for (int j = i + 1; j < n; j++)

{

if (p[i].burst_time > p[j].burst_time) {

// Swap the processes

temp = p[i];

p[i] = p[j];

p[j] = temp;

}

}

}

}



// Calculate waiting time for each process

void calculate_waiting_time(struct Process p[], int n)

{ p[0].waiting_time = 0; // First process has no waiting time



for (int i = 1; i < n; i++) {
```

// Calculate waiting time for each process

for (int i = 1; i < n; i++) {

```
p[i].waiting_time = p[i - 1].waiting_time + p[i - 1].burst_time;

}

}


// Calculate turnaround time for each process

void calculate_turnaround_time(struct Process p[], int n) {

// Calculate turnaround time for each process

for (int i = 0; i < n; i++) {

p[i].turn_around_time = p[i].waiting_time + p[i].burst_time;

}

}


// Calculate and display average waiting and turnaround times

void find_average_times(struct Process p[], int n) {

float total_waiting_time = 0, total_turnaround_time = 0;


// Display individual process times and calculate totals

printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

for (int i = 1; i < n; i++) {
```

```
printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].arrival_time, p[i].burst_time,

p[i].waiting_time, p[i].turn_around_time);

total_waiting_time += p[i].waiting_time;

total_turnaround_time += p[i].turn_around_time;

}



// Calculate and display average waiting time and turnaround time

printf("\nAverage waiting time: %.2f", total_waiting_time / n);

printf("\nAverage turnaround time: %.2f", total_turnaround_time / n);

}
```

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: Thus the program to implement the Shortest Job First (SJF) scheduling technique is executed successful

Ex. No.: 6c) **PRIORITY SCHEDULING**
Date: 5/3/25

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

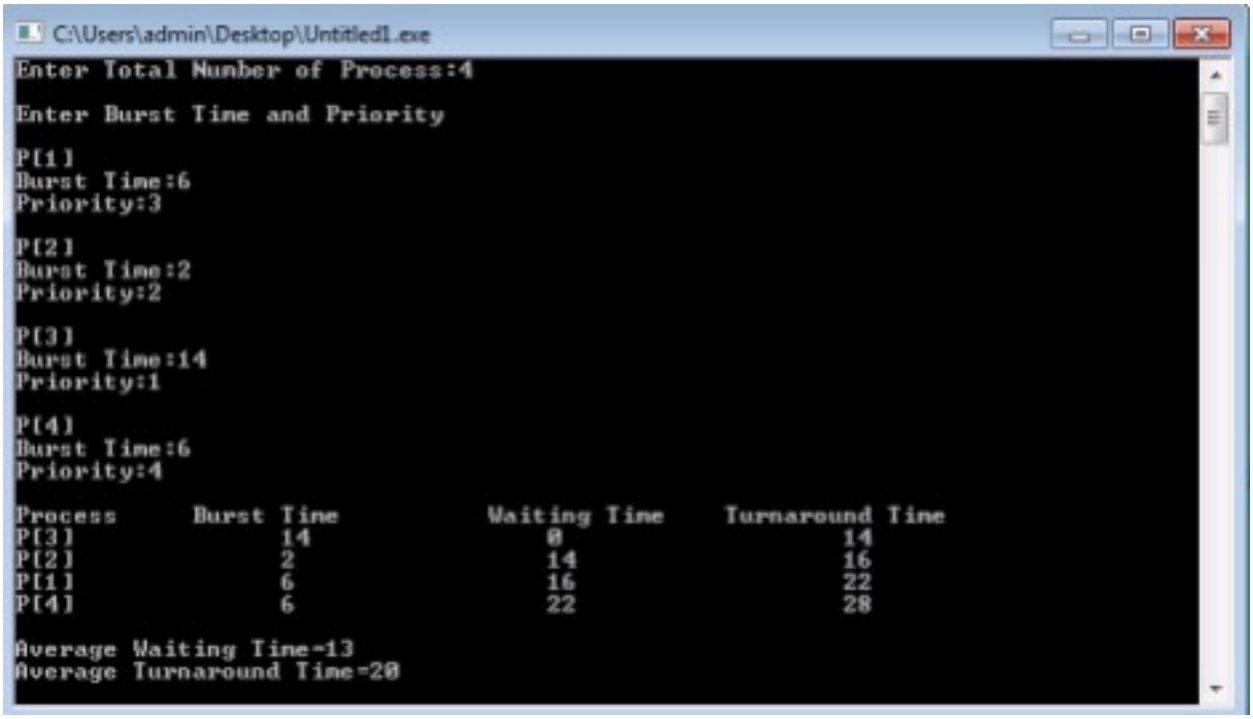
Program Code:
```
#include <stdio.h>
struct Process {
int pid;            // Process ID
int burst_time;           // Burst time of the process
int priority;     // Priority of the process
int waiting_time;         // Waiting time of the process
int turn_around_time; // Turnaround time of the process
};

void calculate_waiting_time(struct Process[], int);
void calculate_turnaround_time(struct Process[], int);
void find_average_times(struct Process[], int);
void sort_by_priority(struct Process[], int);

int main()
{ int n;

// Get the number of processes
printf("Enter the number of processes: ");
scanf("%d", &n);
```

```
struct Process p[n];

// Input process details
for (int i = 0; i < n; i++) {
printf("\nEnter details for process %d:\n", i + 1);
p[i].pid = i + 1;  // Process ID
printf("Burst time: ");
scanf("%d", &p[i].burst_time);
printf("Priority: ");
scanf("%d", &p[i].priority);
}

// Sort processes by priority (highest priority first)
sort_by_priority(p, n);

// Calculate waiting and turnaround times
calculate_waiting_time(p, n);
calculate_turnaround_time(p, n);

// Find and display average times
find_average_times(p, n);

return 0;
}

// Sort processes by priority (higher priority first)
void sort_by_priority(struct Process p[], int n)
{ struct Process temp;
for (int i = 0; i < n - 1; i++)
{ for (int j = i + 1; j < n; j++)
{
if (p[i].priority > p[j].priority) {
// Swap processes if the priority of the first is lower (higher priority value)
temp = p[i];
p[i] = p[j];
p[j] = temp;
}
}
}
}

// Calculate waiting time for each process
void calculate_waiting_time(struct Process p[], int n)
{ p[0].waiting_time = 0; // First process has no waiting time

// Calculate waiting time for each process
```

```
for (int i = 1; i < n; i++) {
```

```
p[i].waiting_time = p[i - 1].waiting_time + p[i - 1].burst_time;
}
}

// Calculate turnaround time for each process
void calculate_turnaround_time(struct Process p[], int n) {
// Calculate turnaround time for each process
for (int i = 0; i < n; i++) {
p[i].turn_around_time = p[i].waiting_time + p[i].burst_time;
}
}

// Calculate and display average waiting and turnaround times
void find_average_times(struct Process p[], int n) {
float total_waiting_time = 0, total_turnaround_time = 0;

// Display individual process times and calculate totals
printf("\nProcess\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");
for (int i = 0; i < n; i++) {
printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].burst_time, p[i].priority,
p[i].waiting_time, p[i].turn_around_time);
total_waiting_time += p[i].waiting_time;
total_turnaround_time += p[i].turn_around_time;
}

// Calculate and display average waiting time and turnaround time
printf("\nAverage waiting time: %.2f", total_waiting_time / n);
printf("\nAverage turnaround time: %.2f", total_turnaround_time / n);
}
```

Sample Output:



Result: Thus the linux programming for priority scheduling is executed

Ex. No.: 6d)
Date : 5/3/25 **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:

```
#include <stdio.h>
struct Process {

int pid;            // Process ID

int burst_time;           // Burst time of the process

int remaining_time;  // Remaining time for the process

int waiting_time;        // Waiting time of the process

int turn_around_time; // Turnaround time of the process

};
```

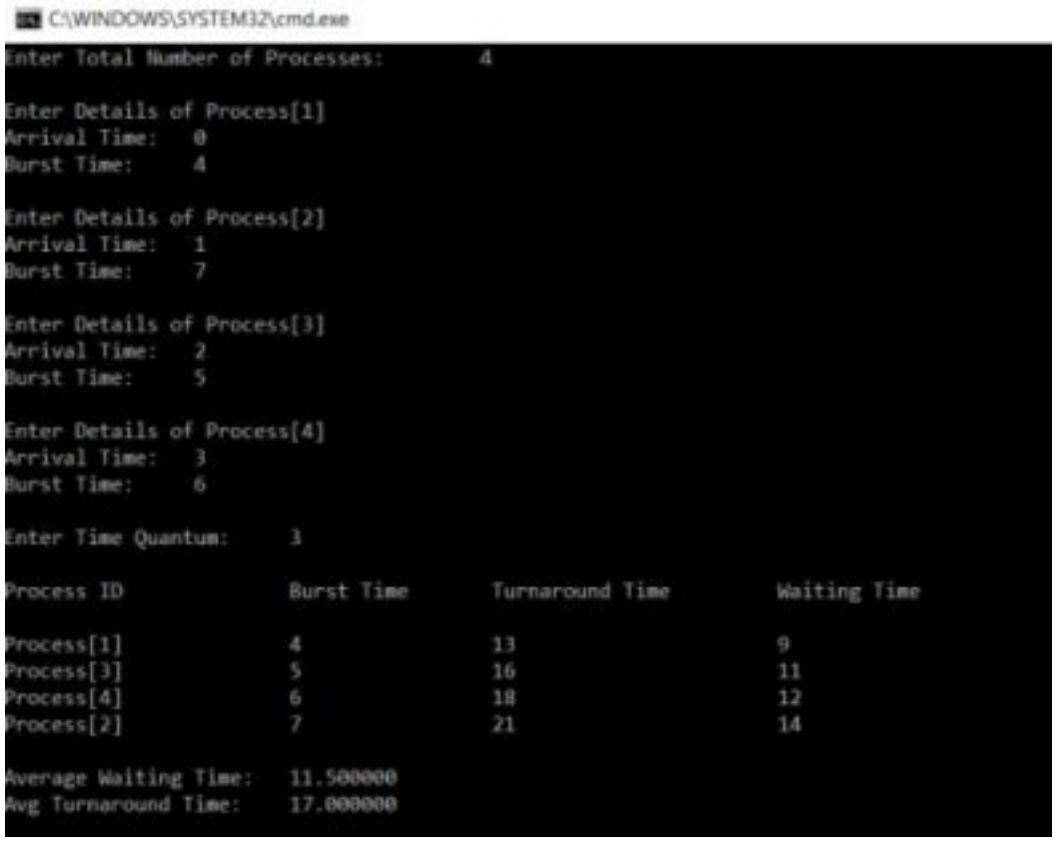
```
void calculate_waiting_time(struct Process[], int, int);
void calculate_turnaround_time(struct Process[], int);
void find_average_times(struct Process[], int);

int main() {
int n, quantum;

// Get the number of processes and time quantum
printf("Enter the number of processes: ");
scanf("%d", &n);
printf("Enter the time quantum: ");
scanf("%d", &quantum);

struct Process p[n];

// Input process details
for (int i = 0; i < n; i++) {
printf("\nEnter details for process %d:\n", i + 1);
p[i].pid = i + 1;  // Process ID
printf("Burst time: ");
scanf("%d", &p[i].burst_time);
p[i].remaining_time = p[i].burst_time; // Initially, remaining time is the burst time
}
```

```
// Calculate waiting and turnaround times
calculate_waiting_time(p, n, quantum);
calculate_turnaround_time(p, n);

// Find and display average times
find_average_times(p, n);

return 0;
}

// Calculate waiting time for each process
void calculate_waiting_time(struct Process p[], int n, int quantum)
{ int time = 0;
int remaining_processes = n;

while (remaining_processes > 0)
{ for (int i = 0; i < n; i++) {
if (p[i].remaining_time > 0) {
// If the process has remaining time
if (p[i].remaining_time > quantum)
{ time += quantum;
p[i].remaining_time -= quantum;
} else {
// If the process can finish within the quantum
time += p[i].remaining_time;
```

```
p[i].waiting_time = time - p[i].burst_time; // Calculate waiting time
p[i].remaining_time = 0;
remaining_processes--;
}
}
}
}
}

// Calculate turnaround time for each process
void calculate_turnaround_time(struct Process p[], int n)
{ for (int i = 0; i < n; i++) {
p[i].turn_around_time = p[i].waiting_time + p[i].burst_time;
}
}

// Calculate and display average waiting and turnaround times
void find_average_times(struct Process p[], int n) {
float total_waiting_time = 0, total_turnaround_time = 0;

// Display individual process times and calculate totals
printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
for (int i = 0; i < n; i++) {
printf("%d\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].burst_time, p[i].waiting_time,
p[i].turn_around_time);
```

```
total_waiting_time += p[i].waiting_time;
total_turnaround_time += p[i].turn_around_time;
}

// Calculate and display average waiting time and turnaround time
printf("\nAverage waiting time: %.2f", total_waiting_time / n);
printf("\nAverage turnaround time: %.2f", total_turnaround_time / n);
}
```

Sample Output:



Result:

Thus the round robin program is executed successfully

Ex. No.: 7

Date: 7/3/25 **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 <sys/ipc.h>

#include <sys/shm.h>

#include <string.h>

#include <stdlib.h>



#define SHM_SIZE 1024  // Size of shared memory segment
```

```
int main() {

key_t key = 1234;  // Unique key for shared memory

int shmid;

char *shm_ptr;


// Create shared memory segment

shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT); // Creating shared memory

if (shmid == -1) {

perror("shmget failed");

exit(1);

}


// Attach shared memory segment to sender process

shm_ptr = shmat(shmid, NULL, 0);

if (shm_ptr == (char *) -1)

{ perror("shmat failed");

exit(1);

}
```

```
printf("Sender: Enter a message to send: ");

fgets(shm_ptr, SHM_SIZE, stdin);  // Writing message to shared memory


// Print confirmation that the message is written to shared memory

printf("Sender: Message written to shared memory: %s", shm_ptr);


// Detach shared memory

shmdt(shm_ptr);


return 0;

}
```

Receiver.c

```
// receiver.c

#include <stdio.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdlib.h>


#define SHM_SIZE 1024  // Size of shared memory segment
```

```
int main() {

key_t key = 1234;  // Unique key for shared memory

int shmid;

char *shm_ptr;


// Access shared memory segment

shmid = shmget(key, SHM_SIZE, 0666);

if (shmid == -1) {

perror("shmget failed");

exit(1);

}


// Attach shared memory segment to receiver process

shm_ptr = shmat(shmid, NULL, 0);

if (shm_ptr == (char *) -1)

{ perror("shmat failed");

exit(1);

}


// Read and print the message from shared memory
```

```
printf("Receiver: Message from shared memory: %s", shm_ptr);



// Detach shared memory

shmdt(shm_ptr);



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:

Thus IPC using shared memory is executed successfully

Ex. No.: 8
Date: 7/3/25

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 <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define SIZE 5  // Size of the buffer

int buffer[SIZE];         // Shared buffer
int in = 0, out = 0; // Indexes for producer and consumer

// Semaphores
sem_t empty;  // Counts empty slots
sem_t full;  // Counts full slots
pthread_mutex_t mutex; // Mutual exclusion for accessing buffer

void* producer(void* arg)
{ int item;
for (int i = 1; i <= 10; i++) {
item = rand() % 100;  // Produce an item
sem_wait(&empty);     // Decrease empty count
pthread_mutex_lock(&mutex);  // Lock buffer access

// Add item to buffer
buffer[in] = item;
printf("Producer produced: %d at position %d\n", item, in);
in = (in + 1) % SIZE;
```

```
pthread_mutex_unlock(&mutex);  // Unlock buffer
sem_post(&full);               // Increase full count

sleep(1); // Simulate production time
}
pthread_exit(NULL);
}

void* consumer(void* arg)
{ int item;
for (int i = 1; i <= 10; i++)
{ sem_wait(&full);      // Decrease full count
pthread_mutex_lock(&mutex); // Lock buffer access

// Remove item from buffer
item = buffer[out];
printf("Consumer consumed: %d from position %d\n", item, out);
out = (out + 1) % SIZE;

pthread_mutex_unlock(&mutex); // Unlock buffer
sem_post(&empty);             // Increase empty count

sleep(2); // Simulate consumption time
}
pthread_exit(NULL);
}

int main()
{ pthread_t prod,
cons;

// Initialize semaphores and mutex
sem_init(&empty, 0, SIZE);
sem_init(&full, 0, 0);
pthread_mutex_init(&mutex, NULL);

// Create producer and consumer threads
pthread_create(&prod, NULL, producer, NULL);
pthread_create(&cons, NULL, consumer, NULL);

// Wait for both threads to finish
pthread_join(prod, NULL);
pthread_join(cons, NULL);

// Destroy semaphores and mutex
sem_destroy(&empty);
sem_destroy(&full);
pthread_mutex_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:

Thus the Producer-Consumer using Semaphores is executed successfully

Ex. No.: 9
Date: 19/3/25

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 Needi<= 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>

#define P 5 // Number of processes
#define R 3 // Number of resources

int main() {
int allocation[P][R] = {
{0, 1, 0},
{2, 0, 0},
{3, 0, 2},
{2, 1, 1},
{0, 0, 2}
};

int max[P][R] = {
{7, 5, 3},
{3, 2, 2},
{9, 0, 2},
{2, 2, 2},
{4, 3, 3}
```

```
};

int available[R] = {3, 3, 2};

int need[P][R];
bool finished[P] = {false};
int safeSequence[P];
int count = 0;

// Calculate need matrix
for (int i = 0; i < P; i++)
{ for (int j = 0; j < R; j++)
{
need[i][j] = max[i][j] - allocation[i][j];
}
}

while (count < P)
{ bool found =
false;
for (int p = 0; p < P; p++)
{ if (!finished[p]) {
bool canAllocate = true;
for (int r = 0; r < R; r++) {
if (need[p][r] > available[r])
{ canAllocate = false;
break;
}
}

if (canAllocate) {
for (int r = 0; r < R; r++)
{ available[r] += allocation[p][r];
}
```

```
safeSequence[count++] = p;
finished[p] = true;
```

```
found = true;
}
}
}

if (!found) {
printf("System is not in a safe state.\n");
return -1;
}
}

// Print safe sequence
printf("System is in a safe state.\nSafe sequence is: ");
for (int i = 0; i < P; i++) {
printf("P%d ", safeSequence[i]);
}
printf("\n");

return 0;
}
```

Sample Output:

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

Result: Thus the deadlock avoidance program is executed successfully

Ex. No.: 10a) **BEST FIT**
Date: 19/3/25

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(block_size, process_size):
n = len(block_size)
m = len(process_size)
allocation = [-1] * m  # Stores index of block allocated to process

for i in range(m):
best_idx = -1
for j in range(n):
if block_size[j] >= process_size[i]:
if best_idx == -1 or block_size[j] < block_size[best_idx]:
best_idx = j

if best_idx != -1:
allocation[i] = best_idx
block_size[best_idx] -= process_size[i]

print("\nProcess No.\tProcess Size\tBlock No.")
for i in range(m):
print(f" {i + 1}\t\t{process_size[i]}\t\t", end="")
if allocation[i] != -1:
print(f"{allocation[i] + 1}")
else:
print("Not Allocated")


# Example Inputs
block_size = [100, 500, 200, 300, 600]
process_size = [212, 417, 112, 426]

# Run Best Fit Allocation
best_fit(block_size, process_size)
```

Sample Output:

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

Result:

Thus the best fit program using python is execxuted successfully

Ex. No.: 10b) **FIRST FIT**
Date: 19/3/25

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_PARTITIONS 10
#define MAX_PROCESSES 10

int main() {
int partitions[MAX_PARTITIONS], processes[MAX_PROCESSES];
int partitionCount, processCount;
int allocation[MAX_PROCESSES];

printf("Enter number of memory partitions: ");
scanf("%d", &partitionCount);

printf("Enter size of each partition:\n");
for (int i = 0; i < partitionCount; i++)
{ printf("Partition %d: ", i + 1);
scanf("%d", &partitions[i]);
}

printf("Enter number of processes: ");
scanf("%d", &processCount);
```

```
printf("Enter size of each process:\n");
for (int i = 0; i < processCount; i++)
{ printf("Process %d: ", i + 1);
scanf("%d", &processes[i]);
allocation[i] = -1; // initially not allocated
}

// First Fit Allocation
for (int i = 0; i < processCount; i++)
{ for (int j = 0; j < partitionCount; j++)
{ if (partitions[j] >= processes[i])
{ allocation[i] = j;
partitions[j] -= processes[i]; // reduce partition size
break;
}
}
}

// Display Allocation
printf("\nProcess No.\tProcess Size\tPartition No.\n");
for (int i = 0; i < processCount; i++)
{ printf("%d\t\t%d\t\t", i + 1, processes[i]);
if (allocation[i] != -1)
printf("%d\n", allocation[i] + 1);
else
printf("Not Allocated\n");
}

return 0;
}
```

Sample Output:

Result: Thus the first fit program is executed successful

Ex. No.: 11a) **FIFO PAGE REPLACEMENT**
Date: 26/3/25

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:

```
#include <stdio.h>

#include <stdbool.h>



int main() {

int frames, pages;



printf("Enter number of frames: ");

scanf("%d", &frames);



printf("Enter number of pages: ");

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

```
int page[pages];

printf("Enter the page reference string:\n");

for (int i = 0; i < pages; i++) {

scanf("%d", &page[i]);

}


int memory[frames];

for (int i = 0; i < frames; i++)

memory[i] = -1;  // initialize frame content


int pageFaults = 0;

int pointer = 0;


printf("\nPage\tFrames\t\tStatus\n");

for (int i = 0; i < pages; i++) {

bool found = false;


// Check if page is already in memory

for (int j = 0; j < frames; j++) {

if (memory[j] == page[i]) {
```

```
found = true;
break;
}
}

if (!found)
{ memory[pointer] =
page[i];
pointer = (pointer + 1) % frames;
pageFaults++;

printf("%d\t", page[i]);
for (int k = 0; k < frames; k++)
{ if (memory[k] == -1)
printf("- ");
else
printf("%d ", memory[k]);
}
printf("\tPage Fault\n");
} else {
```

```
printf("%d\t", page[i]);
```

```
for (int k = 0; k < frames; k++)

{ if (memory[k] == -1)

printf("- ");

else

printf("%d ", memory[k]);

}

printf("\tNo Fault\n");

}

}


printf("\nTotal Page Faults = %d\n", pageFaults);


return 0;

}
```

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: Thus the FIFO program is executed successfully

Ex. No.: 11b) **LRU**
Date: 26/3/25

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 findLRU(int time[], int n)
{ int min = time[0], pos = 0;
for (int i = 1; i < n; ++i)
{ if (time[i] < min) {
min = time[i];
pos = i;
}
}
return pos;
}


int main() {
int frames, pages;


printf("Enter number of frames: ");
scanf("%d", &frames);


printf("Enter number of pages: ");
```

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

int page[pages];
printf("Enter the page reference string:\n");
for (int i = 0; i < pages; i++) {
scanf("%d", &page[i]);
}

int memory[frames];
int time[frames]; // To track last used time
int count = 0, pageFaults = 0;
int currentTime = 0;

for (int i = 0; i < frames; i++)
{ memory[i] = -1;
time[i] = 0;
}

printf("\nPage\tFrames\t\tStatus\n");

for (int i = 0; i < pages; i++)
{ int flag = 0;

for (int j = 0; j < frames; j++)
{ if (memory[j] == page[i])
{ currentTime++;
time[j] = currentTime;
flag = 1;
break;
}
}

if (!flag)
{ int pos;
```

```
if (count < frames)
{ pos = count;
count++;
} else {
pos = findLRU(time, frames);
}
memory[pos] = page[i];
currentTime++;
time[pos] = currentTime;
pageFaults++;
}

printf("%d\t", page[i]);
for (int k = 0; k < frames; k++)
{ if (memory[k] != -1)
printf("%d ", memory[k]);
else
printf("- ");
}

if (!flag)
printf("\tPage Fault\n");
else
printf("\tNo Fault\n");
}

printf("\nTotal Page Faults = %d\n", pageFaults);

return 0;
}
```

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:

Thus the LRU program has been executed successfully

Ex. No.: 11c) **Optimal**
Date: 26/3/25

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:

```
#include <stdio.h>

int search(int page, int frame[], int n)
{ for (int i = 0; i < n; i++) {
if (frame[i] == page)
return 1;
}
return 0;
}

int predict(int pages[], int frame[], int n, int index, int total_pages)
{ int res = -1, farthest = index;
for (int i = 0; i < n; i++)
{ int j;
for (j = index; j < total_pages; j++)
{ if (frame[i] == pages[j]) {
if (j > farthest) {
```

```
farthest = j;
res = i;
}
break;
}
}

if (j == total_pages)
return i; // If not found in future, return immediately
}

return (res == -1) ? 0 : res;
}

void optimalPageReplacement(int pages[], int total_pages, int capacity)
{ int frame[capacity];
int count = 0, page_faults = 0;

for (int i = 0; i < capacity; i++)
frame[i] = -1;

for (int i = 0; i < total_pages; i++) {
if (search(pages[i], frame, capacity))
{ printf("Page %d -> HIT\n", pages[i]);
continue;
}

if (count < capacity)
{ frame[count++] = pages[i];
} else {
int pos = predict(pages, frame, capacity, i + 1, total_pages);
frame[pos] = pages[i];
}

page_faults++;
printf("Page %d -> FAULT\tFrames: ", pages[i]);
```

```
for (int j = 0; j < capacity; j++)
printf("%d ", frame[j]);
printf("\n");
}

printf("\nTotal Page Faults = %d\n", page_faults);
}

int main() {
int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
int total_pages = sizeof(pages) / sizeof(pages[0]);
int capacity = 4;

optimalPageReplacement(pages, total_pages, capacity);

return 0;
}
```

Output:

```
Page 7 -> FAULT      Frames: 7 -1 -1 -1
Page 0 -> FAULT      Frames: 7 0 -1 -1
Page 1 -> FAULT      Frames: 7 0 1 -1
Page 2 -> FAULT      Frames: 7 0 1 2
Page 0 -> HIT
Page 3 -> FAULT      Frames: 3 0 1 2
...
Total Page Faults = X
```

Result:

Thus, a c program to implement Optimal page replacement is executed successfully

Ex. No.: 12 **File Organization Technique- Single and Two level directory**
Date: 28/3/25

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>


struct Directory {

char filename[20][20];

int file_count;

};


void singleLevelDirectory()

{ struct Directory dir;

dir.file_count = 0;

int choice;
```

```
char name[20];

printf("Single Level Directory Implementation\n");

while (1) {
printf("\n1. Create File\n2. Delete File\n3. Search File\n4. List Files\n5. Exit\nEnter choice: ");
scanf("%d", &choice);

switch (choice)
{ case 1:
printf("Enter file name to create: ");
scanf("%s", name);
int found = 0;
for (int i = 0; i < dir.file_count; i++) {
if (strcmp(name, dir.filename[i]) == 0)
{ found = 1;
break;
}
}
if (found)
printf("File already exists!\n");
else {
strcpy(dir.filename[dir.file_count], name);
dir.file_count++;
printf("File created successfully.\n");
}
break;
case 2:
printf("Enter file name to delete: ");
scanf("%s", name);
```

```
found = 0;
for (int i = 0; i < dir.file_count; i++) {
if (strcmp(name, dir.filename[i]) == 0)
{ found = 1;
for (int j = i; j < dir.file_count - 1; j++)
{ strcpy(dir.filename[j], dir.filename[j + 1]);
}
dir.file_count--;
printf("File deleted successfully.\n");
break;
}
}
if (!found)
printf("File not found!\n");
break;
case 3:
printf("Enter file name to search: ");
scanf("%s", name);
found = 0;
for (int i = 0; i < dir.file_count; i++) {
if (strcmp(name, dir.filename[i]) == 0)
{ found = 1;
printf("File found!\n");
break;
}
}
if (!found)
printf("File not found!\n");
break;
case 4:
```

```
printf("Files:\n");
for (int i = 0; i < dir.file_count; i++)
{ printf("%s\n", dir.filename[i]);
}
break;
case 5:
return;
default:
printf("Invalid choice.\n");
}
}
}
```

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:

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

struct UserDirectory
{ char username[20];
char files[10][20];
int file_count;
};
```

```
void twoLevelDirectory()
{ struct UserDirectory
users[5]; int user_count = 0;
int choice;
char username[20], filename[20];
int uIndex = -1;

printf("Two Level Directory Implementation\n");

while (1) {
printf("\n1. Create User Directory\n2. Create File\n3. Delete File\n4. List Files\n5. Exit\nEnter choice:
");
scanf("%d", &choice);

switch (choice)
{ case 1:
printf("Enter new username: ");
scanf("%s", username);
int exists = 0;
for (int i = 0; i < user_count; i++) {
if (strcmp(username, users[i].username) == 0)
{ exists = 1;
break;
}
}
if (exists)
printf("User already exists!\n");
```

```
else {
strcpy(users[user_count].username, username);
users[user_count].file_count = 0;
user_count++;
printf("User directory created.\n");
}
break;
case 2:
printf("Enter username: ");
scanf("%s", username);
uIndex = -1;
for (int i = 0; i < user_count; i++) {
if (strcmp(username, users[i].username) == 0)
{ uIndex = i;
break;
}
}
if (uIndex == -1)
{ printf("User not found.\n");
break;
}
printf("Enter filename to create: ");
scanf("%s", filename);
int fExists = 0;
for (int j = 0; j < users[uIndex].file_count; j++)
{ if (strcmp(filename, users[uIndex].files[j]) == 0)
{ fExists = 1;
```

```
break;
}
}
if (fExists)
printf("File already exists.\n");
else {
strcpy(users[uIndex].files[users[uIndex].file_count], filename);
users[uIndex].file_count++;
printf("File created.\n");
}
break;
case 3:
printf("Enter username: ");
scanf("%s", username);
uIndex = -1;
for (int i = 0; i < user_count; i++) {
if (strcmp(username, users[i].username) == 0)
{ uIndex = i;
break;
}
}
if (uIndex == -1)
{ printf("User not found.\n");
break;
}
printf("Enter filename to delete: ");
scanf("%s", filename);
```

```
int fIndex = -1;
for (int j = 0; j < users[uIndex].file_count; j++)
{ if (strcmp(filename, users[uIndex].files[j]) == 0)
{ fIndex = j;
break;
}
}
if (fIndex == -1)
printf("File not found.\n");
else {
for (int j = fIndex; j < users[uIndex].file_count - 1; j++)
{ strcpy(users[uIndex].files[j], users[uIndex].files[j + 1]);
}
users[uIndex].file_count--;
printf("File deleted.\n");
}
break;
case 4:
for (int i = 0; i < user_count; i++)
{ printf("User: %s\n", users[i].username);++
for (int j = 0; j < users[i].file_count; j++)
{ printf("  - %s\n", users[i].files[j]);
}
}
break;
case 5:
return;
```

```
default:
printf("Invalid choice.\n");
}
}
}
```

Result:

Thus ,the File Organization Technique- Single and Two level directory is executed successfully