Shree Rahul Education Society's (Regd.)

SHREE L. R. TIWARI COLLEGE OF ENGINEERING

(Approved by AICTE & DTE, Maharashtra State & Affiliated to University of Mumbai)
NAAC Accredited, NBA Accredited Program, ISO 9001:2015 Certified | DTE Code No. : 3423
Minority Status (Hindi Linguistic)

Lab Manual

Subject: CSL403 : **Operating System Lab**

Semester: IV

Division: SECS-A

List of Experiments

Sr.No	Name of Practical
1.	Explore usage of basic Linux Commands and system calls for file, directory and process management.
2.	Write shell scripts to display various system information
3.	a. Create a child process in Linux using the fork system call. From the child process obtain the process ID of both child and parent by using getpid and getppid system call. b. Explore wait and waitpid before termination of process.
4.	a. Write a program to demonstrate the concept of non-preemptive scheduling algorithms. (FCFS) b. Write a program to demonstrate the concept of preemptive scheduling algorithms (Priority)
5.	Write a C program to implement solution of Producer consumer problem through Semaphore
6.	Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm
7.	Write a program demonstrate the concept of Dining Philospher's Problem
8.	Write a program to demonstrate the concept of dynamic partitioning placement algorithms i.e. Best Fit, First Fit,
9.	Write a program in C demonstrate the concept of page replacement policies for handling page faults eg: FIFO.
10.	a. Write a C program to simulate File allocation strategies typically sequential files b. Write a program in C to do disk scheduling - FCFS,
11.	Study of Round Robin Scheduling Algorithm in virtual lab

Experiment No. 1

Aim: Explore usage of basic Linux Commands and system calls for file, directory and process management.

For eg: (pwd,touch,cat,cp,rm,mv,mkdir, rmdir, cd, ls, chown, chmod, chgrp, ps. system calls: open, read, write, close, getpid, getppid etc.)

Theory:

1) **touch:** Create a new file or update its timestamp.

    Syntax: touch [OPTION]…[FILE]

    Example: Create empty files called 'file1' and 'file2'

    -$ touch file1 file2

2) **cat:** Concatenate files and print to stdout.

    Syntax: cat [OPTION]…[FILE]

    Example: Create file1 with entered content

    - $ cat > file1

    - Hello

    - ^D

3) **cp:** Copy files

    Syntax: cp [OPTION]source destination

    Example: Copies the contents from file1 to file2 and contents of file1 is retained

    - cp file1 file2

4) **mv:** Move files or rename files

Syntax: mv [OPTION]source destination

Example: Create empty files called 'file1' and 'file2'

-$ mv file1 file2

5) **rm:** Remove files and directories **Syntax:** rm [OPTION]…[FILE] Example: Delete file1

- $ rm file1

6) **mkdir:** Make directory

Syntax: mkdir [OPTION] directory

Example: Create directory called dir1

$ mkdir dir1

7) **rmdir:** Remove a directory

Syntax: rmdir [OPTION] directory

Example: Create empty files called 'file1' and 'file2'

- $ rmdir dir1

8) **cd:** Change directory

Syntax: cd [OPTION] directory

Example: Change working directory to dir1

- $ cd dir1

9) **pwd:** Print the present working directory

Syntax: pwd [OPTION]

Example: Print 'dir1' if a current working directory is dir1

-$ pwd

10) **ls:** ls is the list command in Linux. It will show the full list or content of your directory. Type ls and press the enter key. The whole content will be shown.

Syntax: ls

Example: - $ ls

11) **chown:** Linux chown command is used to change a file's ownership, directory, for a user or group.

The chown stands for change owner.

Syntax: chown [OPTION]... [OWNER][:[GROUP]] FILE...

sudo chown <username> <File name>

12) **chmod:** Linux chmod command is used to change the access permissions of files and directories. It stands for change mode.

Syntax: chmod <options> <permissions> <file name>

Example : To set the read and write permission for other users.

-$ chmod o+w *.txt

13) **ps:** The ps command is used to view currently running processes on the system. It helps us to determine which process is

doing what in our system, how much memory it is using, how much CPU space it occupies, user ID, command name, etc

Example: - $ ps

14) **open:** To open a particular file.

15) **read:** To read the contents in a file.

16) **write:** To write in a file.

17) **close:** To close the opened file.

18) **getpid:** It prints the process id of the current process running.

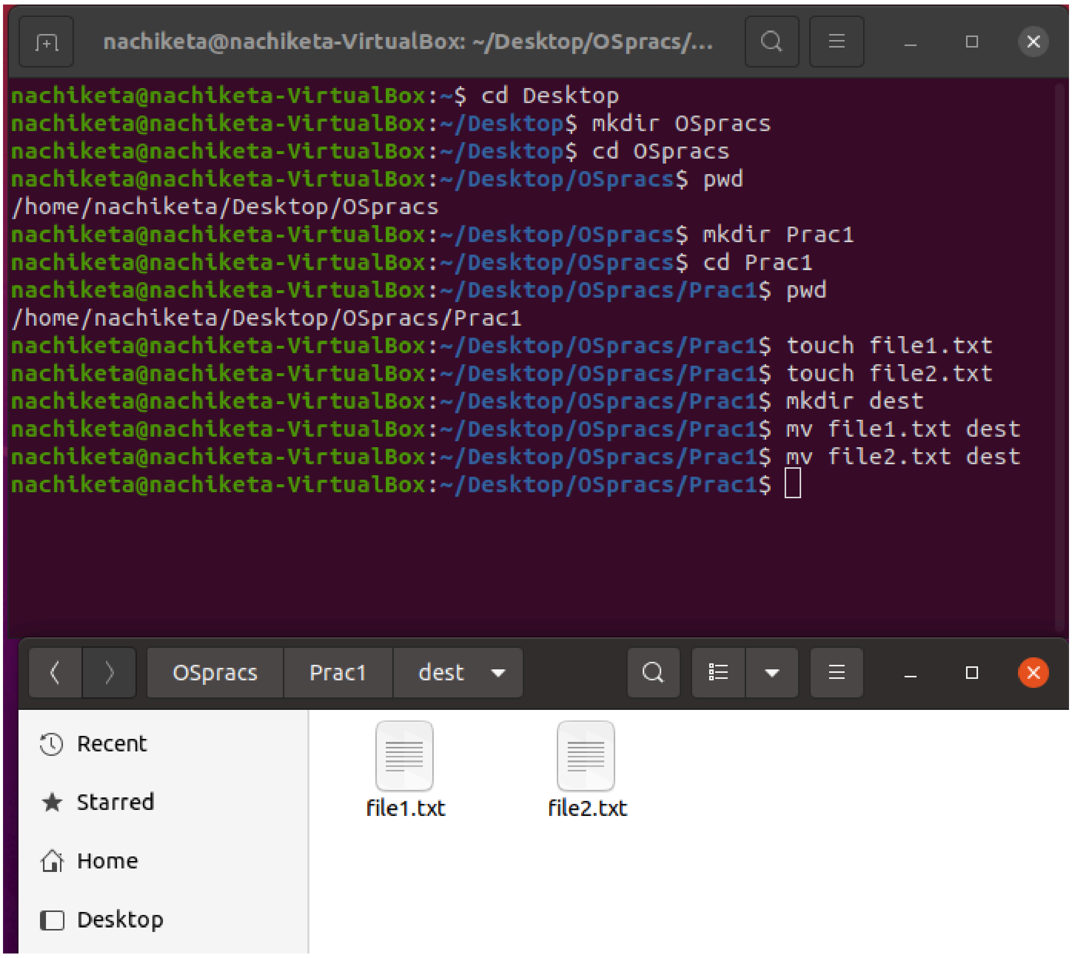
19) **setpid:** User can manually set process ID using setpid command in LINUX terminal.

20) **getppid:** If user has created a child process using a 'fork' system calls then using getppid command you can return its ID.

21) **getuid :** Linux users are assigned with certain unique id getuid command returns user ID .

Output:

1) mkdir, cd, pwd, touch, mv

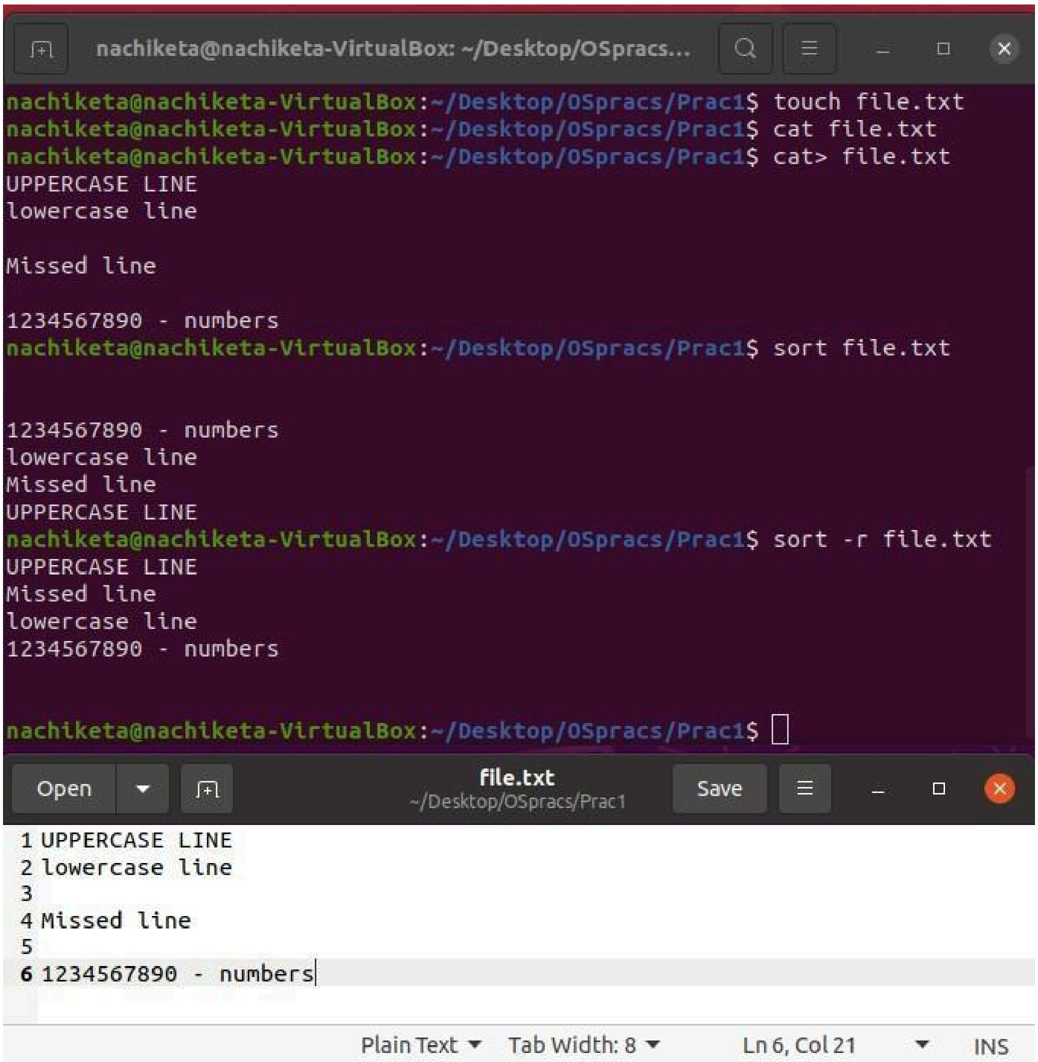


2) Performing sort operation on numerical data and searching operation in file using 'grep'

```
nachiketa@nachiketa-VirtualBox: ~/Desktop/OSpracs/Prac1
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$ touch file1.txt
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$ cat> file1.txt
05
10
7
5
2
589
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$ echo "Sorting data in ascending order"
Sorting data in ascending order
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$ sort -n file1.txt
2
05
5
7
10
589
49756
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$
```

```
nachiketa@nachiketa-VirtualBox: ~/Desktop/OSpracs/Prac1
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$ cat> file1.txt
Hello World
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$ echo "Searching a word in file using GR
EP command"
Searching a word in file using GREP command
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$ grep "Hello" file1.txt
Hello World
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$
```

3) cat operation, sort (ascending &descending) order



4) ps command – for process status

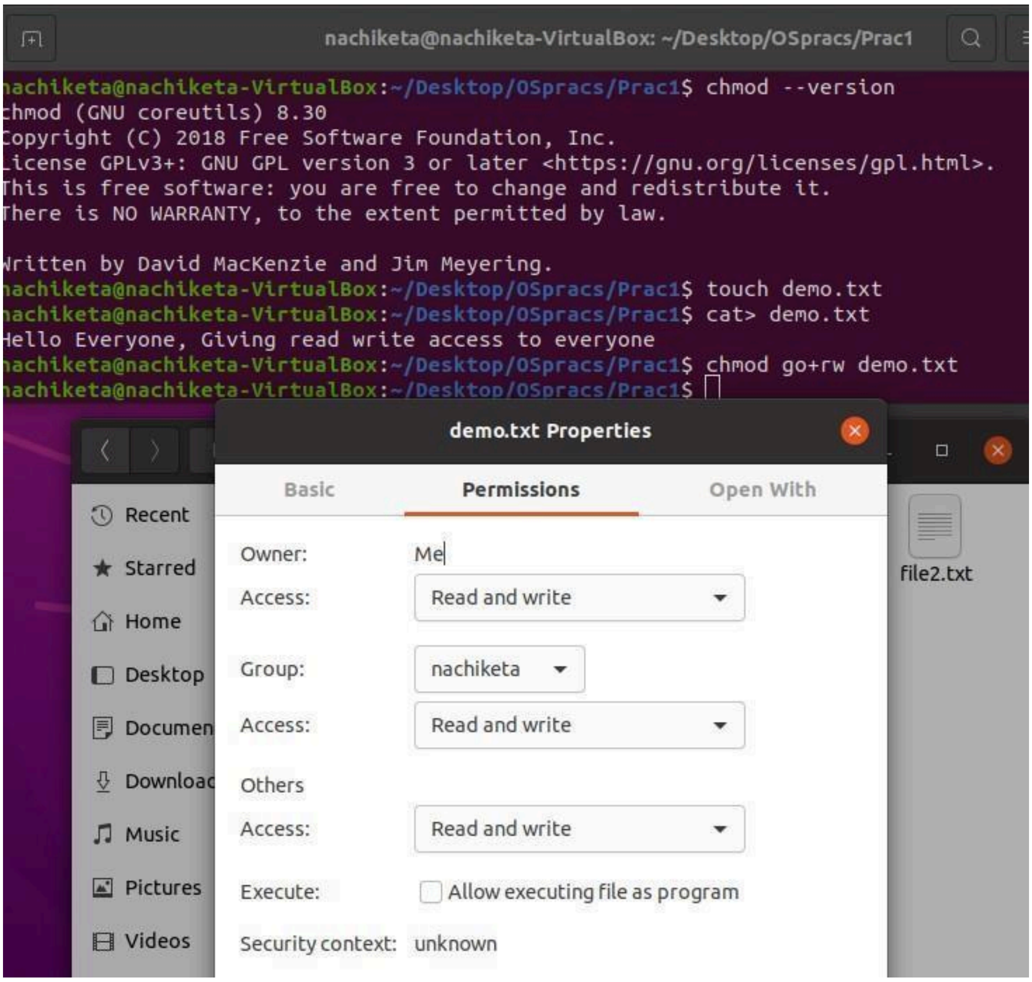
```
nachiketa@nachiketa-VirtualBox: ~/Desktop/OSpracs/Prac1
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$ echo "Currently running processes by th
e user"
Currently running processes by the user
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$ ps -u 1000
    PID TTY          TIME CMD
    952 ?        00:00:00 systemd
    953 ?        00:00:00 (sd-pam)
    958 ?        00:00:01 pulseaudio
    960 ?        00:00:00 tracker-miner-f
    963 ?        00:00:00 gnome-keyring-d
    967 tty2     00:00:00 gdm-x-session
    969 tty2     00:00:38 Xorg
    977 ?        00:00:01 dbus-daemon
    980 ?        00:00:00 gvfsd
    985 ?        00:00:00 gvfsd-fuse
    998 ?        00:00:00 gvfs-udisks2-vo
   1003 ?        00:00:00 gvfs-gphoto2-vo
   1017 ?        00:00:00 gvfs-afc-volume
   1022 ?        00:00:00 gvfs-mtp-volume
   1026 ?        00:00:00 gvfs-goa-volume
   1031 ?        00:00:00 goa-daemon
   1052 ?        00:00:00 goa-identity-se
   1071 tty2     00:00:00 gnome-session-b
   1151 ?        00:00:00 VBoxClient
   1153 ?        00:00:00 VBoxClient
   1163 ?        00:00:00 VBoxClient
   1164 ?        00:00:00 VBoxClient
   1168 ?        00:00:00 VBoxClient
   1169 ?        00:00:06 VBoxClient
   1173 ?        00:00:00 VBoxClient
   1174 ?        00:00:00 VBoxClient
   1187 ?        00:00:00 ssh-agent
   1219 ?        00:00:00 at-spi-bus-laun
   1224 ?        00:00:00 dbus-daemon
   1249 ?        00:00:00 gnome-session-c
   1256 ?        00:00:00 gnome-session-b
   1269 ?        00:01:45 gnome-shell
   1303 ?        00:00:00 ibus-daemon
   1307 ?        00:00:00 ibus-memconf
   1308 ?        00:00:04 ibus-extension-
   1312 ?        00:00:00 ibus-x11
   1315 ?        00:00:00 ibus-portal
   1324 ?        00:00:00 at-spi2-registr
   1332 ?        00:00:00 xdg-permission-
   1337 ?        00:00:00 gnome-shell-cal
   1345 ?        00:00:00 evolution-sourc
   1357 ?        00:00:00 evolution-calen
```

```
   1372 ?        00:00:00 dconf-service
   1376 ?        00:00:00 evolution-addre
   1379 ?        00:00:00 gjs
   1399 ?        00:00:00 gsd-a11y-settin
   1400 ?        00:00:01 gsd-color
   1402 ?        00:00:00 gsd-datetime
   1403 ?        00:00:00 gvfsd-trash
   1405 ?        00:00:00 gsd-housekeepin
   1407 ?        00:00:00 gsd-keyboard
   1411 ?        00:00:00 gsd-media-keys
   1412 ?        00:00:01 gsd-power
   1413 ?        00:00:00 gsd-print-notif
   1416 ?        00:00:00 gsd-rfkill
   1419 ?        00:00:00 gsd-screensaver
   1420 ?        00:00:00 gsd-sharing
   1424 ?        00:00:00 gsd-smartcard
   1425 ?        00:00:00 gsd-sound
   1426 ?        00:00:00 gsd-usb-protect
   1427 ?        00:00:00 gsd-wacom
   1428 ?        00:00:00 gsd-wwan
   1430 ?        00:00:01 gsd-xsettings
   1463 ?        00:00:00 gsd-disk-utilit
   1465 ?        00:00:01 evolution-alarm
   1478 ?        00:00:00 ibus-engine-sim
   1529 ?        00:00:00 gsd-printer
   1729 ?        00:00:00 gvfsd-metadata
   1732 ?        00:00:00 update-notifier
   2215 ?        00:00:11 nautilus
   2343 ?        00:00:04 gnome-terminal-
   2351 pts/0    00:00:00 bash
   2736 pts/0    00:00:00 ps
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$
```

5) Case insensitive searching, and using awk command.

```
nachiketa@nachiketa-VirtualBox: ~/Desktop/OSpracs/Prac1

nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$ echo "Case Insensitive Seach"
Case Insensitive Seach
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$ grep -i "hello" file2.txt
HELLO WORLD
hello world
Hello World
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$ echo "Printing line no. of searched string"
Printing line no. of searched string
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$ grep -n "hello" file2.txt
2:hello world
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$ awk '{print}' file2.txt
HELLO WORLD
hello world
Hello World

Hii how are you guys ? doing fine?
this is test case scenario
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$ awk '{print $1,$4}' file2.txt
HELLO
hello
Hello

Hii you
this case
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac1$
```

6) chmod – access permission



<u>Outcome:</u>

Demonstrate basic Operating system Commands, Shell scripts, System Calls and API w.r.t. Linux.

Experiment No. 2

Aim: Write Shell Scripts and Execute them in Linux.

Theory:

Write shell scripts to do the following:

a. Display OS version, release number, kernel version ANSWER: -

- To display OS version and kernel version:

  ~ $ cat /etc/os-release

- To display release number: -

  ~ $ lsb_release -a

b. Display top 10 processes in descending order ANSWER: -

- First, we open the vi editor by putting the command:

  ~$ vim Newtest3.sh

- Then you enter the following code in the vi editor:

```
- #!/bin/sh
echo " Top 10 processes in descending order are as follows";
ps aux | head -n 11
```

- For displaying the output in the terminal, we put the following command: -

  ~$ bash Newtest3.sh

c. Display processes with highest memory usage. ANSWER: -

- First, we open the vi editor by putting the command:

  ~$ vim Newtest2.sh

- Then you enter the following code in the vi editor:
  - #!/bin/sh
  echo "processes with highest memory usage are as follows";

ps -eo pid, ppid, cmd, %mem, %cpu --sort=-%mem|head

- For displaying the output in the terminal, we put the following command: -

  ~$ bash Newtest2.sh

d. Display current logged in user and log name. ANSWER: -

- First, we open the vi editor by putting the command: -

  ~$ vim Newtest4.sh

- Then you enter the following code in the vi editor:
  - #!/bin/sh
  echo " Logged in User";
  who -u
  echo "No of logged in
  users"; who -u | wc -l
- For displaying the output in the terminal, we put the following command: -

  ~$ bash Newtest4.sh

e. Display current shell, home directory, operating system type, current path setting, current working directory. ANSWER: -

- First, we open the vi editor by putting the command:

  ~$ vim newtest.sh

- Then you enter the following code in the vi editor:
  - #!/bin/sh

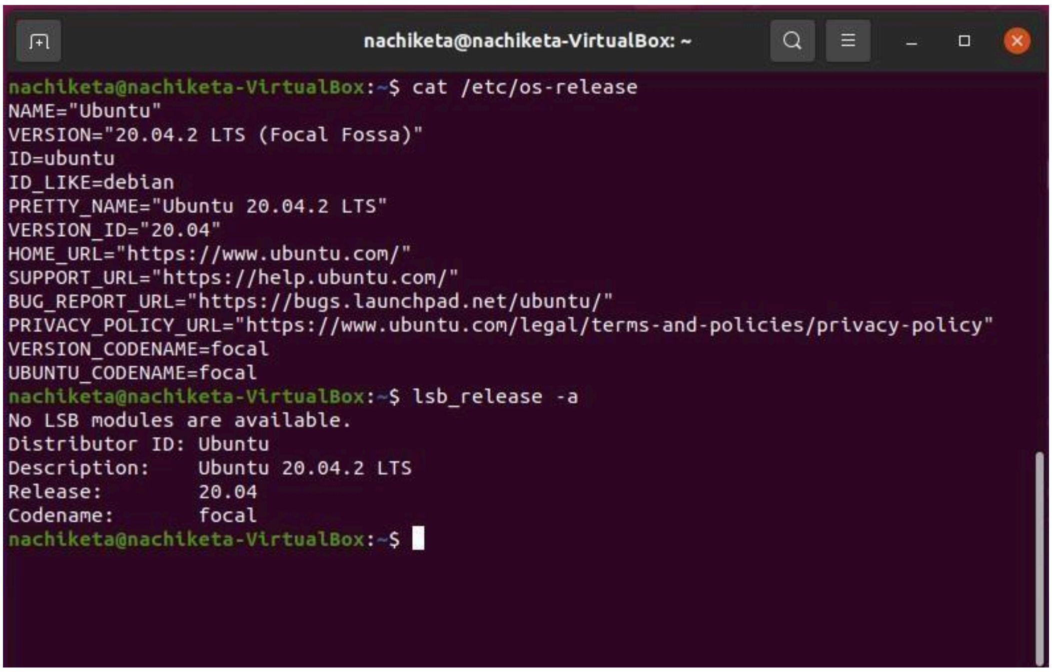
```
echo " Current Home Directory is:";
whoami
```

echo " Current Working Directory is:";
pwd
echo "Operating System
Type:"; uname
echo "Release no";
uname -a

- For displaying the output in the terminal, we put the following command: -
  ~$ bash newtest.sh

Output:

a)



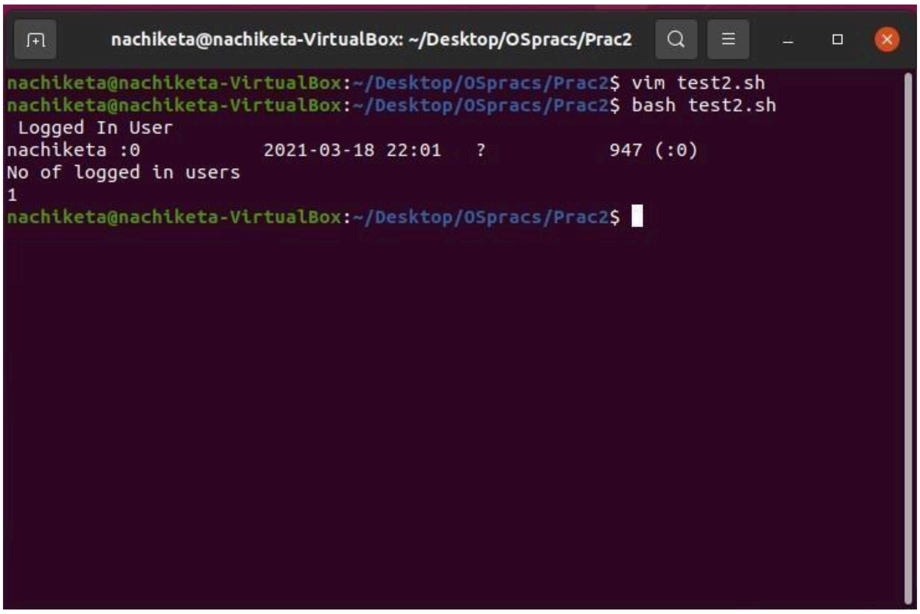
b)

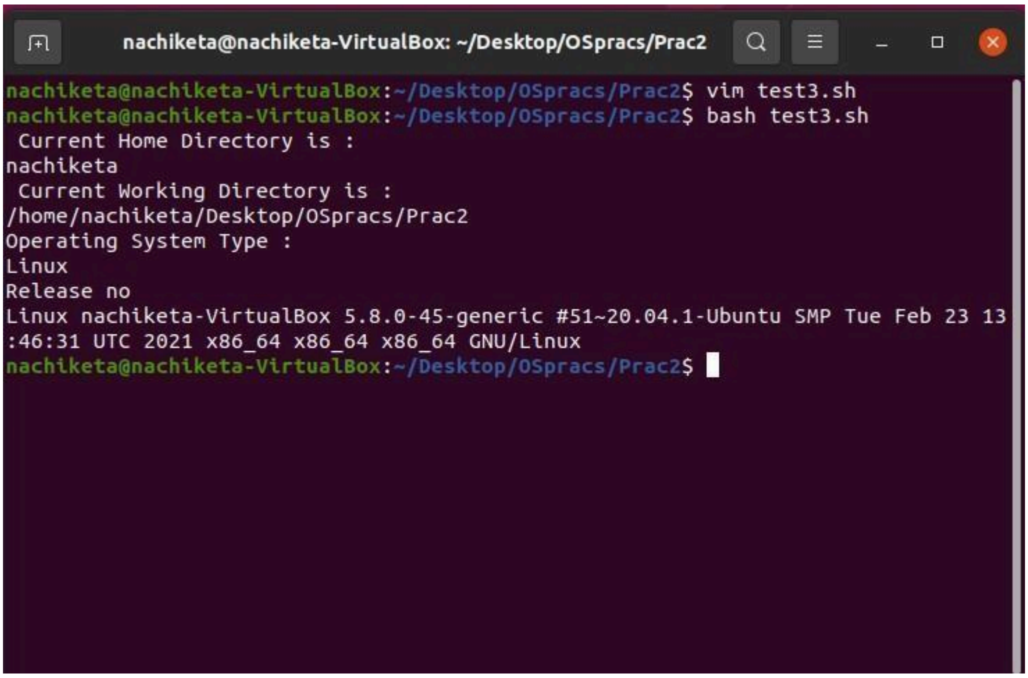
```
nachiketa@nachiketa-VirtualBox: ~/Desktop/OSpracs/Prac2
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac2$ vim test.sh
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac2$ bash test.sh
 Top 10 processes in descending order are as follows
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.3 167608 11420 ?        Ss   22:00   0:01 /sbin/init splash
root           2  0.0  0.0      0     0 ?        S    22:00   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   22:00   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   22:00   0:00 [rcu_par_gp]
root           6  0.0  0.0      0     0 ?        I<   22:00   0:00 [kworker/0:0H-kblockd]
root           9  0.0  0.0      0     0 ?        I<   22:00   0:00 [mm_percpu_wq]
root          10  0.0  0.0      0     0 ?        S    22:00   0:00 [ksoftirqd/0]
root          11  0.0  0.0      0     0 ?        I    22:00   0:00 [rcu_sched]
root          12  0.0  0.0      0     0 ?        S    22:00   0:00 [migration/0]
root          13  0.0  0.0      0     0 ?        S    22:00   0:00 [idle_inject/0]
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac2$
```

c)

```
nachiketa@nachiketa-VirtualBox: ~/Desktop/OSpracs/Prac2
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac2$ vim test1.sh
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac2$ bash test1.sh
test1.sh: line 1: !/bin/sh: No such file or directory
processes with highest memory usage are as follows
    PID    PPID CMD                         %MEM %CPU
   1251     932 /usr/bin/gnome-shell        12.1  3.4
    949     947 /usr/lib/xorg/Xorg vt2 -dis  2.5  1.2
   2707     932 /usr/bin/nautilus --gapplic  2.3  1.1
   1430    1237 /usr/libexec/evolution-data  1.9  0.0
   2491     932 /usr/bin/gnome-calendar --g  1.9  0.0
   3060     932 /usr/libexec/gnome-terminal  1.6  1.0
   2496     932 /usr/bin/seahorse --gapplic  1.4  0.0
   1008     932 /usr/libexec/goa-daemon      1.1  0.0
   3092     932 /usr/libexec/tracker-extrac  1.1  1.8
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac2$
```

d)





e)

Outcome: Demonstrated basic Operating system Commands, Shell scripts, System Calls and API w.r.t. Linux.

Experiment No. 3

Aim:

a. Create a child process in Linux using the fork system call. From the child process obtain the process ID of both child and parent by using getpid and getppid system call.

b. Explore wait and waitpid before termination of process.

Theory:

a. Create a child process in Linux using the fork system call.

fork() System Call

A Process can create a new child process using fork() system call. This new child process created through fork() call will have same memory image as of parent process i.e. it will be duplicate of calling process but will have different process ID.
For example,

Suppose there is a Process "Sample" with Process ID 1256 and parent ID 12. Now as soon as this process calls the fork() function, a new process will be created with same memory image but with different process ID.

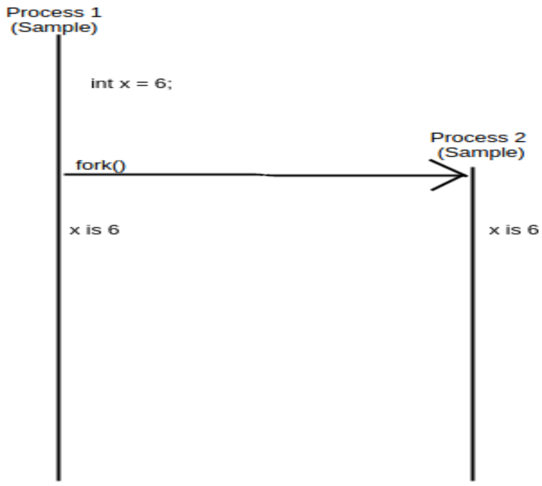
Also, process which has called this fork() function will become the parent process of this new process i.e.
Process 1: *Sample (pid=1341 | Parent Process ID = 12)*

After calling fork() system call,

Process 1: *Sample (pid=1341 | Parent Process ID = 12)*

Process 2: *Sample (pid= 4567 | Parent Process ID = 1341)*

As memory image of new child process will be the copy of parent process's memory image. So, all variables defined before fork() call will be available in child process with same values.



If fork() call is successful then code after this call will be executed in both the process. Therefore, fork() function's return value will be different in both the process's i.e.

If fork() call is successful then it will,

- Return 0 in child process.
- Return process id of new child process in parent process.

If fork() call is unsuccessful then it will return -1.

b. Explore wait and waitpid before termination of process

wait() and waitpid()

The wait() system call suspends execution of the current process until one of its children terminates. The call wait(&status) is equivalent to:
waitpid(-1, &status, 0);

The waitpid() system call suspends execution of the current process until a child specified by pid argument has changed state. By default, waitpid() waits only for terminated children, but this behaviour is modifiable via the options argument, as described below.
The value of pid can be:

Tag	Description
< -1	meaning wait for any child process whose process group ID is equal to the absolute value of *pid*.
-1	meaning wait for any child process.
0	meaning wait for any child process whose process group ID is equal to that of the calling process.
> 0	meaning wait for the child whose process ID is equal to the value of *pid*.

Program:

a)

```
#include <stdio.h>
#include
<stdlib.h>
#include
<unistd.h>
      //
Driver code
int   main()
{
    int pid, pid1, pid2;

    pid = fork();
```

```
if (pid == 0)
{
    sleep(3);

    printf("child[1] --> pid = %d and ppid = %d\n",
           getpid(), getppid());
```

```
    }
    else {
        pid1 = fork();
        if (pid1 == 0)
        {
            sleep(2);
            printf("child[2] --> pid = %d and ppid = %d\n",
                   getpid(), getppid());
        }
        else {
            pid2 = fork();
            if (pid2 == 0)
            {
                printf("child[3] --> pid = %d and ppid = %d\n",
                       getpid(), getppid());
            }
            else {

                sleep(3);
                printf("parent --> pid = %d\n", getpid());
            }
        }
    }
    return 0;
}
```

b)

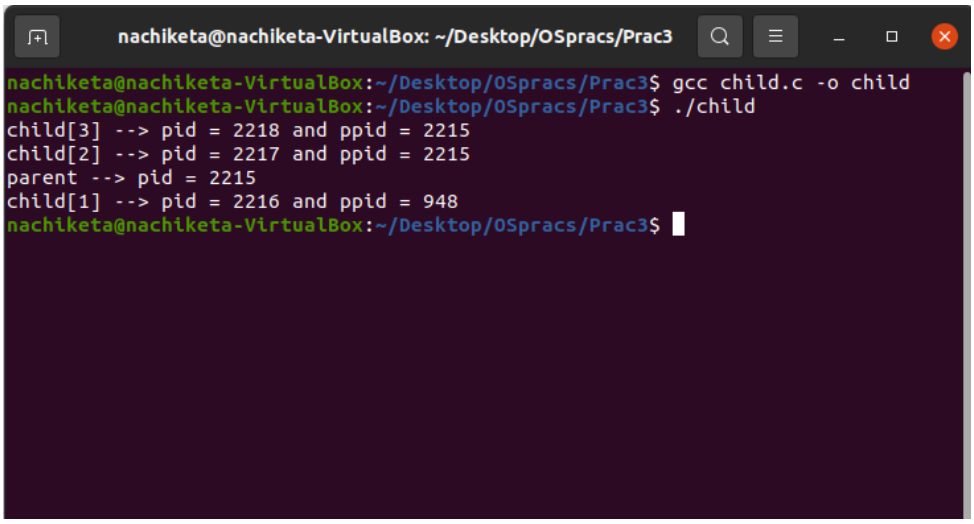
```
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h
>
#include<unistd.h>
void waitexample()
{
int i, stat;
pid_t
pid[5];
for (i=0; i<5; i++)
{
if ((pid[i] = fork()) == 0)
{
sleep(1);
exit(100 +
i);
}
}
```

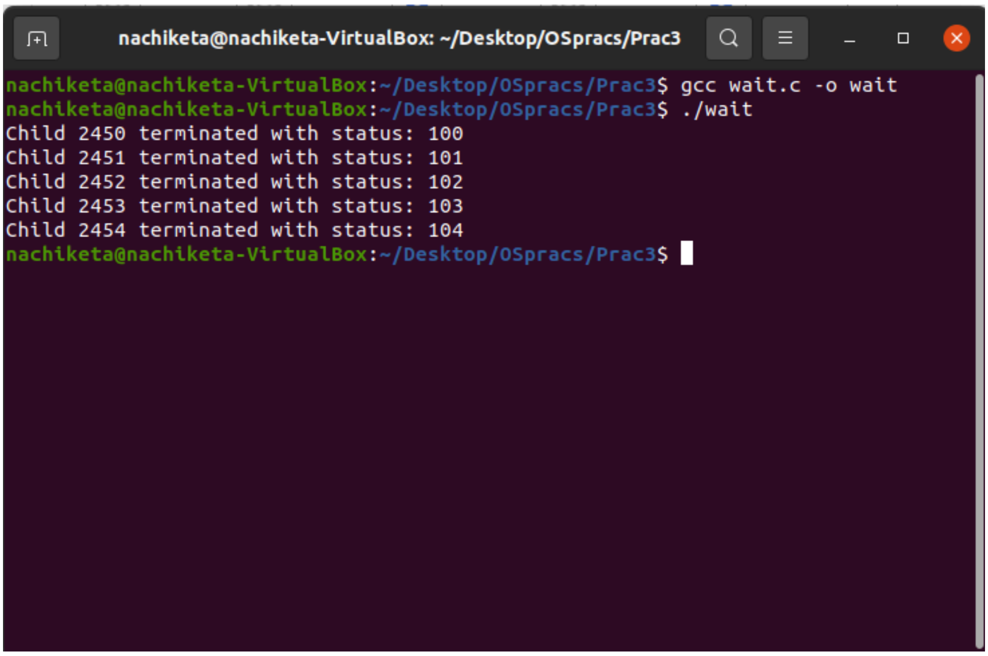
```
// Using waitpid() and printing exit status
// of children.
```

```
for (i=0; i<5; i++)
{
pid_t cpid = waitpid(pid[i], &stat,
0); if (WIFEXITED(stat))
printf("Child %d terminated with status: %d\n",cpid, WEXITSTATUS(stat));
}
}
// Driver
code int
main()
{
waitexample()
; return 0;
}
```

Output:

a)



b) 

<u>Outcome:</u>

hence , Study of process system calls has been done.

Experiment No.-4

AIM:

a. Write a program to demonstrate the concept of non-pre-emptive scheduling algorithms (FCFS).

b. Write a program to demonstrate the concept of pre-emptive scheduling algorithms (Priority).

Theory:

a. Non- Pre-emptive Scheduling Algorithms (FCFS).

First Come First Serve (FCFS) is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival. It is the easiest and simplest CPU scheduling algorithm. In this type of algorithm, processes which requests the CPU first get the CPU allocation first. This is managed with a FIFO queue. The full form of FCFS is First Come First Serve.
As the process enters the ready queue, its PCB (Process Control Block) is linked with the tail of the queue and, when the CPU becomes free, it should be assigned to the process at the beginning of the queue.
Characteristics of FCFS method

- It supports non-pre-emptive and pre-emptive scheduling algorithm.
- Jobs are always executed on a first-come, first-serve basis.
- It is easy to implement and use.
- This method is poor in performance, and the general wait time is quite high.

Advantages of FCFS

Here, are pros/benefits of using FCFS scheduling algorithm:

- The simplest form of a CPU scheduling algorithm
- Easy to program
- First come first served

Disadvantages of FCFS

Here, are cons/ drawbacks of using FCFS scheduling algorithm:

- It is a Non-Pre-emptive CPU scheduling algorithm, so after the process has been allocated to the CPU, it will never release the CPU until it finishes executing.
- The Average Waiting Time is high.
- Short processes that are at the back of the queue have to wait for the long process at the front to finish.
- Not an ideal technique for time-sharing systems.
- Because of its simplicity, FCFS is not very efficient.

b. Pre-emptive Scheduling Algorithms (Priority)

The pre-emptive priority scheduling algorithm is a popular operating system process management and job scheduling algorithm.
Every job that enters the job queue is assigned a priority based on which its execution takes place. As simple it sounds, the processes with a higher priority will be executed first and then the processes with the lower priorities will be executed.
If there are multiple processes in the queue with the same priority, then such jobs are executed in the order of their arrival often called as **first come first served**.
In this pre-emptive implementation of priority scheduling program in C, we consider the **arrival time** of the processes.
Since this is a pre-emptive job scheduling algorithm, the CPU can leave the process midway. The current state of the process will be saved by the **context switch**.

The system can then search for another process with a higher priority in the ready queue or waiting queue and start its execution.
Once the CPU comes back to the previous incomplete process, the job is resumed from where it was earlier paused.

Advantages

- Pre-emptive priority scheduling is much more efficient as compared to the non-pre-emptive version.
- This priority job scheduling algorithm is quite simple to implement.
- The aging technique is implemented to reduce the starvation of lower priority processes.
- The average turnaround time and waiting time is efficient.

Disadvantages

- Indefinite blockage of the lower priority jobs.
- For a system failure occurs, the unfinished lower priority jobs are removed from the system and cannot be recovered.

Program:

a)

```
#include<stdio.h> int
main(){
int bt[10]={0},at[10]={0},tat[10]={0},wt[10]={0},ct[10]={0};
int n,sum=0;
float totalTAT=0,totalWT=0; printf("Enter number
of processes: "); scanf("%d",&n);
printf("Enter arrival time and burst time for each process\n"); for(int i=0;i<n;i++)
{
printf("Arrival time of process[%d]: ",i+1); scanf("%d",&at[i]);
printf("Burst time of process[%d]: ",i+1); scanf("%d",&bt[i]);
printf("\n");
}
//calculate completion time of processes for(int
j=0;j<n;j++)
```

```
{
sum+=bt[j];
ct[j]+=sum;
}
//calculate turnaround time and waiting times for(int
k=0;k<n;k++)
{
tat[k]=ct[k]-at[k];
totalTAT+=tat[k];
}
for(int k=0;k<n;k++)
{
wt[k]=tat[k]-bt[k];
totalWT+=wt[k];
}
printf("Solution: \n\n");
printf("P#\t AT\t BT\t CT\t TAT\t WT\t\n\n"); for(int
i=0;i<n;i++)
{
printf("P%d\t %d\t %d\t %d\t
%d\t%d\n",i+1,at[i],bt[i],ct[i],tat[i],wt[i]);
}
printf("\nAverage Turnaround Time = %f\n",totalTAT/n); printf("Average WT =
%f\n\n",totalWT/n);
return 0;
}
```

b)

```
#include<stdio.h>
struct process
{
        char process_name;
        int arrival_time, burst_time, ct, waiting_time, turnaround_time, priority;
int status;
}process_queue[10]; int
limit;
void Arrival_Time_Sorting(){ struct
        process temp; int i, j;
        for(i = 0; i < limit - 1; i++)
        {
                for(j = i + 1; j < limit; j++)
```

```
                {
                        if(process_queue[i].arrival_time >
process_queue[j].arrival_time)
                        {
                                temp = process_queue[i]; process_queue[i] =
                                process_queue[j]; process_queue[j] = temp;
                        }
                }
        }
}
void main()
{
        int i, time = 0, burst_time = 0, largest; char c;
        float wait_time = 0, turnaround_time = 0, average_waiting_time, average_turnaround_time;
        printf("\nEnter Total Number of Processes:\t"); scanf("%d", &limit);
        for(i = 0, c = 'A'; i < limit; i++, c++)
        {
                process_queue[i].process_name = c; printf("\nEnter Details For
                Process[%C]:\n",
process_queue[i].process_name);
                printf("Enter Arrival Time:\t");
                scanf("%d", &process_queue[i].arrival_time ); printf("Enter Burst
                Time:\t");
                scanf("%d", &process_queue[i].burst_time); printf("Enter
                Priority:\t");
                scanf("%d", &process_queue[i].priority);
                process_queue[i].status = 0;
                burst_time = burst_time + process_queue[i].burst_time;
        }
        Arrival_Time_Sorting();
        process_queue[9].priority = -9999;
        printf("\nProcess Name\tArrival Time\tBurst Time\tPriority\tWaiting Time");
        for(time = process_queue[0].arrival_time; time < burst_time;)
        {
                largest = 9;
                for(i = 0; i < limit; i++)
                {
```

```
                if(process_queue[i].arrival_time <= time && process_queue[i].status != 1
&& process_queue[i].priority > process_queue[largest].priority)
                {
                        largest = i;
                }
        }
        time = time + process_queue[largest].burst_time; process_queue[largest].ct = time;
        process_queue[largest].waiting_time =
process_queue[largest].ct - process_queue[largest].arrival_time - process_queue[largest].burst_time;
        process_queue[largest].turnaround_time = process_queue[largest].ct -
process_queue[largest].arrival_time;
        process_queue[largest].status = 1;
        wait_time = wait_time + process_queue[largest].waiting_time; turnaround_time =
        turnaround_time +
process_queue[largest].turnaround_time;
        printf("\n%c\t\t%d\t\t%d\t\t%d\t\t%d", process_queue[largest].process_name,
process_queue[largest].arrival_time, process_queue[largest].burst_time,
process_queue[largest].priority, process_queue[largest].waiting_time);
    }
    average_waiting_time = wait_time / limit; average_turnaround_time =
    turnaround_time / limit; printf("\n\nAverage waiting time:\t%f\n",
    average_waiting_time);
    printf("Average Turnaround Time:\t%f\n", average_turnaround_time);
}
```

Output:

```
nachiketa@nachiketa-VirtualBox: ~/Desktop/OSpracs/Prac4
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac4$ gcc fcfs.c -o fcfs
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac4$ ./fcfs
Enter number of processes: 3
Enter arrival time and burst time for each process
Arrival time of process[1]: 1
Burst time of process[1]: 4

Arrival time of process[2]: 2
Burst time of process[2]: 3

Arrival time of process[3]: 3
Burst time of process[3]: 5

Solution:

P#      AT      BT      CT      TAT     WT

P1      1       4       4       3       -1
P2      2       3       7       5       2
P3      3       5       12      9       4

Average Turnaround Time = 5.666667
Average WT = 1.666667
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac4$
```

a)

b)

```
nachiketa@nachiketa-VirtualBox: ~/Desktop/OSpracs/Prac4
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac4$ gcc prio.c -o prio
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac4$ ./prio

Enter Total Number of Processes:        3

Enter Details For Process[A]:
Enter Arrival Time:     1
Enter Burst Time:       23
Enter Priority: 2

Enter Details For Process[B]:
Enter Arrival Time:     2
Enter Burst Time:       54
Enter Priority: 1

Enter Details For Process[C]:
Enter Arrival Time:     3
Enter Burst Time:       12
Enter Priority: 3

Process Name    Arrival Time    Burst Time      Priority        Waiting Time
A               1               23              2               0
C               3               12              3               21
B               2               54              1               34

Average waiting time:   18.333334
Average Turnaround Time:        48.000000
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac4$
```

Outcome:

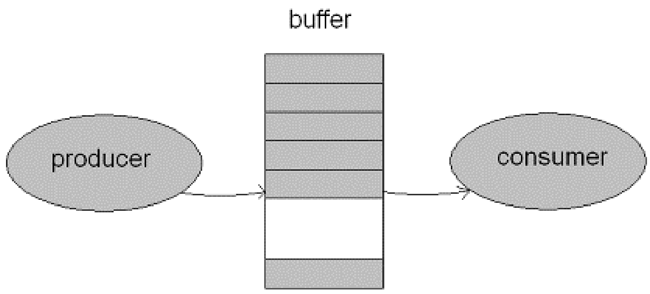
Implemented various process scheduling algorithms and evaluate their performance.

Experiment No. 5

Aim: Write a C program to implement solution of Producer Consumer Problem through Semaphore.

Theory:

The producer consumer problem is a synchronization problem. There is a fixed size buffer and the producer produces items and enters them into the buffer. The consumer removes the items from the buffer and consumes them.
A producer should not produce items into the buffer when the consumer is consuming an item from the buffer and vice versa. So the buffer should only be accessed by the producer or consumer at a time.
The producer should go to sleep when buffer is full. Next time when consumer removes data it notifies the producer and producer starts producing data again. The consumer should go to sleep when buffer is empty. Next time when producer add data it notifies the consumer and consumer starts consuming data. This solution can be achieved using semaphores.



A semaphore S is an integer variable that can be accessed only through two standard operations: wait () and signal ().
The wait () operation reduces the value of semaphore by 1 and the signal () operation increases its value by 1.

wait(S){

while(S<=0); // busy
waiting S--;
}

signal(S)
{ S++;
}

Semaphores are of two types:

Binary Semaphore – This is similar to mutex lock but not the same thing. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.

Counting Semaphore – Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Program:

```
#include<stdio.h>
#include<stdlib.h
>
int
mutex=1,full=0,empty=3,x=0;
int main()
{
    int n;
    void producer();
    void consumer();
    int wait(int);
    int signal(int);
    printf("\n1.Producer\n2.Consumer\n3.Exit")
    ; while(1)
    {
        printf("\nEnter your choice:");
        scanf("%d",&n);
```

```
switch(n)
{
```

```
            case 1:    if((mutex==1)&&(empty!=0))
                        producer();
                    else
                        printf("Buffer is full!!");
                    break;
            case 2:
                        if((mutex==1)&&(full!=0
                        )) consumer();
                    else
                        printf("Buffer is empty!!");
                    break;
                                    case 3: e
                                            x
                                            i
                    }                       t
                }                           (
                                            0
                                            )
                                            ;
                                            b
                                            r
                                            e
                                            a
                                            k
                                            ;

    return 0;
}

int wait(int s)
{
    return (--s);
}

int signal(int s)
{
    return(++s);
}
void producer()
{
    mutex=wait(mutex)
    ;
    full=signal(full)
    ;
    empty=wait(empty)
    ; x++;
    printf("\nProducer produces the item %d",x);
    mutex=signal(mutex);
}
void consumer()
{
    mutex=wait(mutex);
    full=wait(full);
    empty=signal(empty)
```

```
    ;
    printf("\nConsumer consumes item
    %d",x); x--;
    mutex=signal(mutex);
}
```

Output:

```
nachiketa@nachiketa-VirtualBox: ~/Desktop/OSpracs/Prac4
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac4$ gcc prog.c -o prog
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac4$ ./prog

1.Producer
2.Consumer
3.Exit
Enter your choice:1

Producer produces the item 1
Enter your choice:1

Producer produces the item 2
Enter your choice:2

Consumer consumes item 2
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!!
Enter your choice:3
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac4$
```

Outcome:

Implement and analyse concepts of synchronization and deadlocks.

Experiment No. 6

Aim: Write a program to demonstrate the concept of deadlock avoidance through Banker's Algorithm.

Theory:

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Why Banker's algorithm is named so?

Banker's algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan, then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders comes to withdraw their money then the bank can easily do it.
In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.
Following **Data structures** are used to implement the Banker's Algorithm:
Let **'n'** be the number of processes in the system and **'m'** be the number of resources types.

Available :

- It is a 1-d array of size **'m'** indicating the number of available resources of each type.
- Available[j] = k means there are **'k'** instances of resource type $\mathbf{R_j}$

Max :

- It is a 2-d array of size '**n*m'** that defines the maximum demand of each process in a system.
- Max[i, j] = k means process $\mathbf{P_i}$ may request at most **'k'** instances of resource type $\mathbf{R_j}$.

Allocation :

- It is a 2-d array of size **'n*m'** that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process $\mathbf{P_i}$ is currently allocated **'k'** instances of resource type $\mathbf{R_j}$

Need :

- It is a 2-d array of size **'n*m'** that indicates the remaining resource need of each process.
- Need [i, j] = k means process $\mathbf{P_i}$ currently need **'k'** instances of resource type $\mathbf{R_j}$ for its execution.
- Need [i, j] = Max [i, j] – Allocation [i, j]

$Allocation_i$ specifies the resources currently allocated to process P_i and $Need_i$ specifies the additional resources that process P_i may still request to complete its task.
Banker's algorithm consists of Safety algorithm and Resource request algorithm

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1) Let Work and Finish be vectors of length 'm' and 'n' respectively. Initialize: Work = Available
Finish[i] = false; for i=1, 2, 3, 4….n

2) Find an i such that both

a) Finish[i] = false

b) $Need_i$ <= Work

if no such i exists goto step (4)

3) Work = Work + Allocation[i]

Finish[i] =
true goto step
(2)
4) if Finish [i] = true for all i

then the system is in a safe state

Resource-Request Algorithm

Let $Request_i$ be the request array for process P_i. $Request_i$ [j] = k means process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1) If $Request_i$ <= $Need_i$

Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.
2) If $Request_i$ <= Available

Goto step (3); otherwise, P_i must wait, since the resources are not available.

3) Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

Available = Available – Requesti

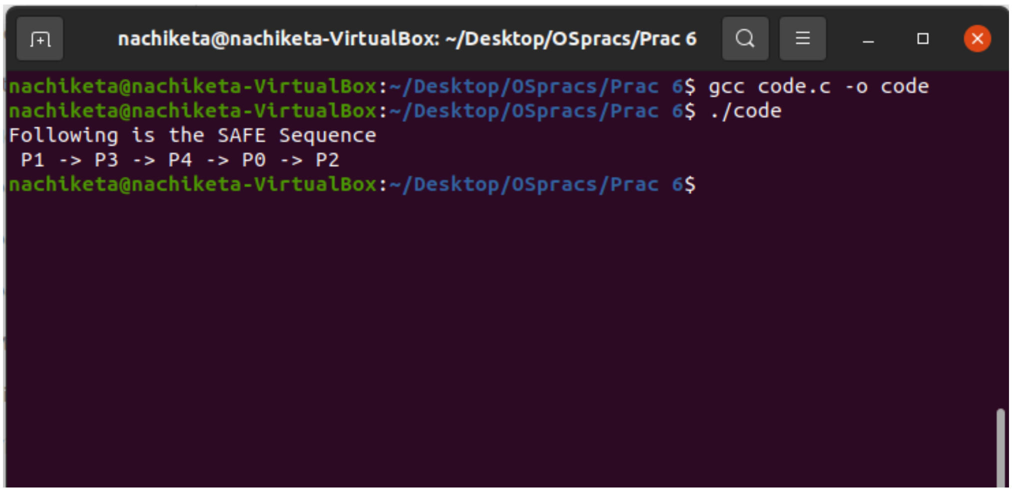
$Allocation_i = Allocation_i +$
$Request_i$ $Need_i = Need_i – Request_i$

Program:

```
#include
<stdio.h> int
main()
{
    // P0, P1, P2, P3, P4 are the Process names here
    int n, m, i, j, k;
    n = 5; // Number of processes
    m = 3; // Number of resources
    int alloc[5][3] = { { 0, 1, 0 }, // P0    // Allocation Matrix
                        { 2, 0, 0 }, // P1
                        { 3, 0, 2 }, // P2
                        { 2, 1, 1 }, // P3
                        { 0, 0, 2 } }; // P4

    int max[5][3] = { { 7, 5, 3 }, // P0    // MAX Matrix
                      { 3, 2, 2 }, // P1
                      { 9, 0, 2 }, // P2
                      { 2, 2, 2 }, // P3
                      { 4, 3, 3 } }; // P4

    int avail[3] = { 3, 3, 2 }; // Available

    Resources int f[n], ans[n], ind = 0;

for (k = 0; k < n; k++) {
        f[k] = 0;
    }
    int need[n][m];
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];
    }
    int y = 0;
    for (k = 0; k < 5; k++) {
        for (i = 0; i < n; i++)
        {
            if (f[i] == 0) {

                int flag = 0;
                for (j = 0; j < m; j++) {
                    if (need[i][j] >
                        avail[j]){ flag = 1;
                         break;
```

```
                    }
                }

                if (flag == 0) {
                    ans[ind++] =
                    i;
                    for (y = 0; y < m; y++)
                        avail[y] += alloc[i][y];
                    f[i] = 1;
                }
            }
        }
    }
    printf("Following is the SAFE
    Sequence\n"); for (i = 0; i < n - 1; i++)
        printf(" P%d ->", ans[i]);
    printf(" P%d", ans[n - 1]);
    printf("\n");
    return (0);
}
```

Output:



Outcome:

Hence Bankers Algorithm was studied successfully.

Experiment No. 7

Aim: Write a program to demonstrate the concept of Dining Philosopher's Problem.

Theory:

Implementation of dining philosophers using threads

Problem Description

Develop a program to implement the solution of the dining philosopher's problem using **threads**. The input to the program is the number of philosophers to be seated around the table. Output shows the various stages that each philosopher passes through within a certain time. A philosopher can be in anyone of the three stages at a time: thinking, eating or finished eating.

Data Structures and Functions

The main data structures used here are:

Arrays

The arrays represent the philosophers and corresponding chopsticks for them. Each element in the philosopher's array corresponds to a thread and each element in the chopstick's array corresponds to a mutex variable.
The functions used here are:

1. pthread_mutex_init (&mutex, NULL) – initialization of mutex variable
2. pthread_mutex_lock (&mutex) – attempt to lock a mutex
3. pthread_mutex_unlock (&mutex) – unlock a mutex
4. pthread_create (ptr to thread, NULL, (void*) func, (void*))
5. pthread_join (ptr to thread, &msg)-This function will make the main program wait until the called thread is finished executing it's task.
6. pthread_mutex_destroy (ptr to thread)-
7. pthread_exit(NULL)

Note: while compiling this program use the following:

$ gcc -o c code.c -pthread

Algorithm

Algorithm for process:

1. Start.
2. Declare and initialize the thread variables (philosophers) as required.
3. Declare and initialize the mutex variables (chopsticks) as required.

4. Create the threads representing philosophers.

5. Wait until the threads finish execution.

6. Stop.

Algorithm for thread (philosopher i) function:

1. Start.
2. Philosopher i is thinking.
3. Lock the left fork spoon.
4. Lock the right fork spoon.
5. Philosopher i is eating.
6. sleep
7. Release the left fork spoon.
8. Release the right fork spoon.
9. Philosopher i Finished eating. 10.Stop.

Program:

```
#include<stdio.h>
#include<fcntl.h>
#include<semaphore.h
>
#include<sys/wait.h>
#include<pthread.h>
#include<stdlib.h>
sem_t *sem[20];
int n;
int
main()
{
pid_t cpid[5];
char
semname[5]; int
i,j=0;
n = 5;
for(i=0;i<n;i++)
{
sprintf(semname,"%d",getpid()+i);
sem[i]=sem_open(semname,O_CREAT|O_EXCL,0666,1)
; if(sem[i]==SEM_FAILED)
perror("Unable to create semaphore");
}
for(i=0;i<n;i++)
```

```
{
cpid[i]=fork()
;
if(cpid[i]==0)
break;
}
if(i==n)
{
```

```
int status;
for(i=0;i<n;i++
)
waitpid(cpid[i],&status,WUNTRACED)
; for(i=0;i<n;i++)
{
sem_close(sem[i]);
sprintf(semname,"%d",getpid()+i)
; sem_unlink(semname);
}
}
else
reader(i)
;
}
int reader(int val)
{
printf("%d
Thinking\n",val+1); while(1)
{
sem_wait(sem[val%n]);
if(!sem_trywait(sem[(val+1)%n])
) break;
else
sem_post(sem[val%n])
;
}
printf("%d
Eating\n",val+1); sleep(2);
sem_post(sem[val%n]);
sem_post(sem[(val+1)%n]);
printf("%d Finished Eating\n",val+1);
}
```

Output:

```
nachiketa@nachiketa-VirtualBox: ~/Desktop/OSpracs/Prac 5
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac 5$ gcc -o c code.c -pthread
code.c: In function 'main':
code.c:23:52: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
   23 | k=pthread_create(&philosopher[i],NULL,(void *)func,(int *)i);
      |                                                    ^
code.c: In function 'func':
code.c:55:1: warning: implicit declaration of function 'sleep' [-Wimplicit-function-declaration]
   55 | sleep(3);
      | ^~~~~
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac 5$ ./c

Philosopher 1 is thinking
Philosopher 1 is eating
Philosopher 2 is thinking
Philosopher 4 is thinking
Philosopher 4 is eating
Philosopher 3 is thinking
Philosopher 5 is thinking
Philosopher 1 Finished eating
Philosopher 4 Finished eating
Philosopher 3 is eating
Philosopher 5 is eating
Philosopher 2 is eating
Philosopher 5 Finished eating
Philosopher 3 Finished eating
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac 5$ $
```

Outcome:

Hence, Dining Philosophers Problem was studied successfully.

Experiment No. 8

Aim: Write a program to demonstrate the concept of dynamic partitioning placement algorithms i.e. Best Fit, First Fit, Worst-Fit etc.

Theory:

There are various memory management schemes in operating system like first fit, best fit and worst fit.

First Fit:

What is First Fit Memory Management Scheme?

In this scheme we check the blocks in a sequential manner which means we pick the first process then compare its size with first block size if it is less than size of block it is allocated otherwise, we move to second block and so on.

When first process is allocated, we move on to the next process until all processes are allocated.

First Fit Algorithm

1. Get no. of Processes and no. of blocks.
2. After that get the size of each block and process requests.

   Now allocate processes
   if(block size >= process
   size)
   //allocate the process
   Else
   //move on to next block
3. Display the processes with the blocks that are allocated to a respective

process.

4. Stop.

Best Fit:

What is Best Fit Memory Management Scheme?

Best fit uses the best memory block based on the Process memory request. In best fit implementation the algorithm first selects the smallest block which can adequately fulfil the memory request by the respective process.

Because of this memory is utilized optimally but as it compares the blocks with the requested memory size it increases the time requirement and hence slower than other methods. It suffers from Internal Fragmentation which simply means that the memory block size is greater than the memory requested by the process, then the free space gets wasted.

Once we encounter a process that requests a memory which is higher than block size, we stop the algorithm.

Best Fit Algorithm

1. Get no. of Processes and no. of blocks.
2. After that get the size of each block and process requests.
3. Then select the best memory block that can be allocated using the above definition.
4. Display the processes with the blocks that are allocated to a respective process.
5. Value of Fragmentation is optional to display to keep track of wasted memory.
6. Stop.

<u>Program:</u>

First Fit

```
#include<stdio.h>
void main()
{
	int bsize[10], psize[10], bno, pno, flags[10], allocation[10], i, j;

	for(i = 0; i < 10; i++)
	{
		flags[i] = 0;
		allocation[i] = -1;
	}
	printf("Enter no. of blocks: ");
	scanf("%d", &bno);
	printf("\nEnter size of each block: ");
	for(i = 0; i < bno; i++)
		scanf("%d", &bsize[i]);
```

```
	printf("\nEnter no. of processes: ");
	scanf("%d", &pno);
	printf("\nEnter size of each process: ");
	for(i = 0; i < pno; i++)
		scanf("%d", &psize[i]);
	for(i = 0; i < pno; i++)		//allocation as per first
		fit for(j = 0; j < bno; j++)
			if(flags[j] == 0 && bsize[j] >= psize[i])
			{
				allocation[j] = i;
				flags[j] = 1;
				break;
			}
	//display allocation details
	printf("\nBlock no.\tsize\t\tprocess no.\t\tsize");
	for(i = 0; i < bno; i++)
	{
		printf("\n%d\t\t%d\t\t", i+1, bsize[i]);
		if(flags[i] == 1)
			printf("%d\t\t\t%d",allocation[i]+1,psize[allocation[i]]);
					else	printf("Not allocated");

			}
		}
```

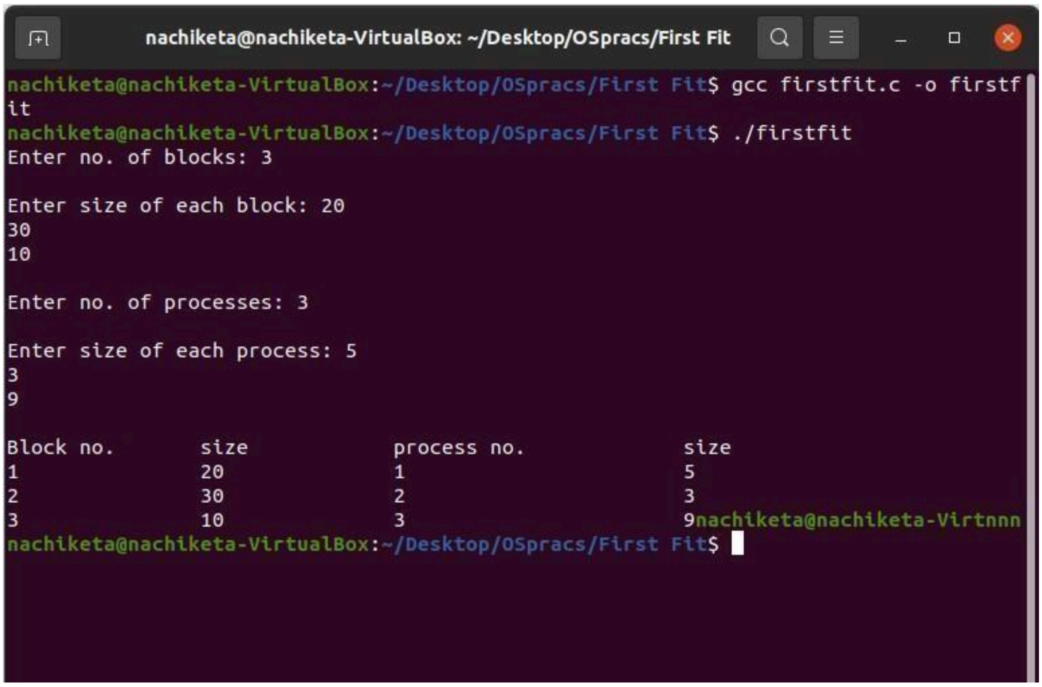
Best Fit

```
#include<stdio.h>
void main()
{
	int
	fragment[20],b[20],p[20],i,j,nb,np,temp,lowest=9999;
	static int barray[20],parray[20];

	printf("\n\t\t\tMemory Management Scheme - Best Fit");
	printf("\nEnter the number of blocks:");
	scanf("%d",&nb);
	printf("Enter the number of processes:");
	scanf("%d",&np);

	printf("\nEnter the size of the blocks:-\n");
	for(i=1;i<=nb;i++)
    {
		printf("Block no.%d:",i);
	 scanf("%d",&b[i]);
    }
	printf("\nEnter the size of the processes :-\n");
	for(i=1;i<=np;i++)
    {
	 printf("Process no.%d:",i);
	 scanf("%d",&p[i]);
    }
	for(i=1;i<=np;i++)
	{
		for(j=1;j<=nb;j++)
		{
			if(barray[j]!=1)
```

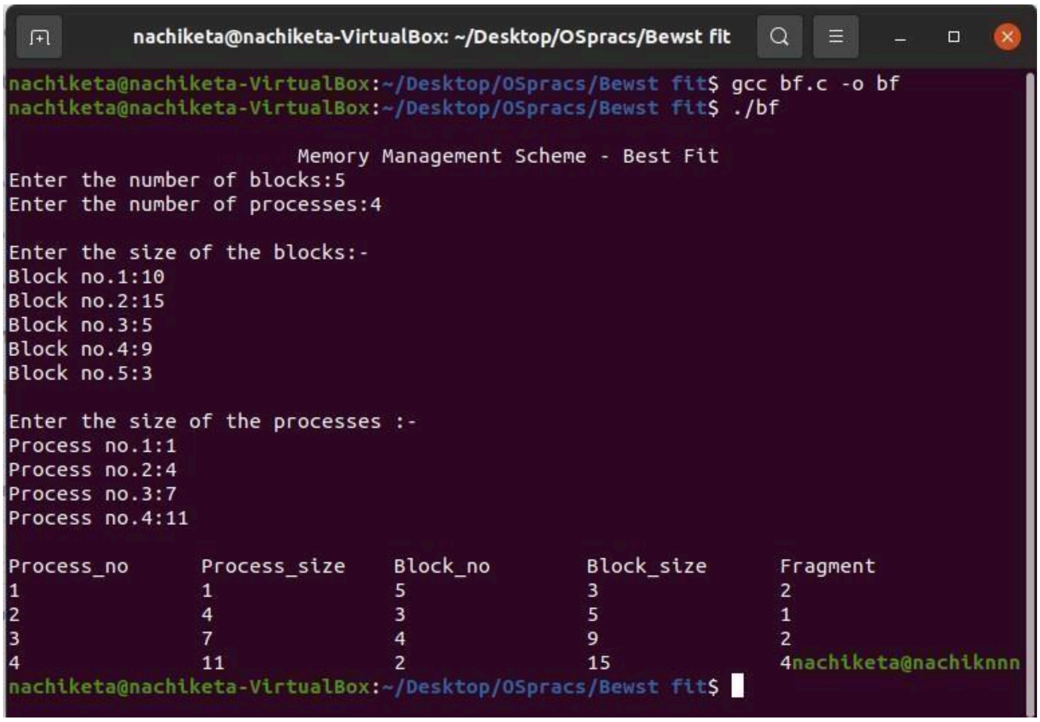
```
                        {
                                temp=b[j]-p[i];
                                if(temp>=0)
                                        if(lowest>temp)
                                        {
                                                parray[i]=j
                                                      ;
                                                lowest=temp
                                                      ;
                                        }
                        }
                }
                fragment[i]=lowest;
                barray[parray[i]]=1;
                lowest=10000;
        }
        printf("\nProcess_no\tProcess_size\tBlock_no\tBlock_size\tFragment");
        for(i=1;i<=np && parray[i]!=0;i++)
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,p[i],parray[i],b[parray[i]],fragment[i]);
}
```

Output:

First Fit



Best Fit



Outcome:

Implement various Memory Management techniques and evaluate their performance.

Experiment No. 9

Aim: Write a program in C demonstrate the concept of page replacement policies for handling page faults eg: FIFO, LRU.

Theory:

FIFO(First In First Out):

- The simplest page-replacement algorithm and work on the basis of first in first out (FIFO). It throws out the pages in the order in which they were brought in.
- The time is associated with each page when it was brought into main memory.
- This algorithm always chooses oldest page for replacement.
- Since replacement is FIFO, a queue can be maintained to hold all the pages in main memory.
- This algorithm doesn't care about which pages are accessed frequently and which are not. However, it is used in windows 2000.

LRU(Least Recently Used):

- The time of page's last use is associated with each page.
- When a page must be replaced, LRU chooses that page that was used farthest back in the past.
- LRU is a good approximation to the optimal algorithm.
- This algorithm looks backward in time while optimal replacement algorithm looks forward in time.
- This policy suggests that replace a page whose last usage is farthest from current time.
- This algorithm can be implemented with some hardware support and is considered to be a good solution for page replacement.

- This algorithm does not suffer through Belady's anomaly.

FIFO Algorithm:

Let capacity be the number of pages that memory can hold. Let set be the current set of pages in memory.

1. Start traversing the pages.

   i) If set holds less pages than capacity.

      a) Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.
      b) Simultaneously maintain the pages in the queue to perform FIFO.
      c) Increment page fault

   ii) Else

      If current page is present in set, do nothing.

      Else

      a) Remove the first page from the queue as it was the first to be entered in the memory
      b) Replace the first page in the queue with the current page in the string.
      c) Store current page in the queue.
      d) Increment page faults.

2. Return page faults.

LRU Algorithm:

Let capacity be the number of pages that memory can hold. Let set be the current set of pages in memory.

1. Start traversing the pages.

   i) If set holds less pages than capacity.

      a) Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.
      b) Simultaneously maintain the recent occurred index of each page in a map called indexes.
      c) Increment page fault

   ii) Else

      If current page is present in set, do nothing.

Else

a) Find the page in the set that was least recently used. We find it using index array. We basically need to replace the page with minimum index.
b) Replace the found page with current page.
c) Increment page faults.
d) Update index of current page.

2. Return page faults

Program:

FIFO:

```
#include<stdio.h>
int main()
{
 int reference_string[10], page_faults = 0, m, n, s, pages, frames;
 printf("\nEnter Total Number of Pages:\t");
 scanf("%d", &pages);
 printf("\nEnter values of Reference String:\n");
 for(m = 0; m < pages; m++)
 {

 scanf("%d", &reference_string[m]);
 }
 printf("\nEnter Total Number of Frames:\t");
 {
 scanf("%d", &frames);
 }
 int temp[frames];
 for(m = 0; m < frames; m++)
 {
 temp[m] = -1;
 }
 for(m = 0; m < pages; m++)
 {
 s = 0;
 for(n = 0; n < frames; n++)
 {
 if(reference_string[m] == temp[n]) {
 s++;
 page_faults--;
 }
 }
 page_faults++;
 if((page_faults <= frames) && (s == 0))
 {
 temp[m] = reference_string[m];
 }
 else if(s == 0)
 {
 temp[(page_faults - 1) % frames] = reference_string[m];
 }
 printf("\n");
 for(n = 0; n < frames; n++)
```

```
 {
 printf("%d\t", temp[n]);
 }
 }
 printf("\nTotal Page Faults:\t%d\n",
 page_faults); return 0;
}
```

LRU:

```
#include<stdio.h>
int findLRU(int time[], int n){
int i, minimum = time[0], pos =
0; for(i = 1; i < n; ++i){
if(time[i] < minimum){
minimum =
time[i]; pos = i;
}
}
return pos;
}
int main()
{
int no_of_frames, no_of_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2,
i, j, pos, faults = 0;
printf("Enter number of frames: ");
scanf("%d", &no_of_frames);
printf("Enter number of pages: ");
scanf("%d", &no_of_pages);
printf("Enter reference string: ");
for(i = 0; i < no_of_pages; ++i){
scanf("%d", &pages[i]);
}
for(i = 0; i < no_of_frames; ++i){
frames[i] = -1;
}
for(i = 0; i < no_of_pages;
++i){ flag1 = flag2 = 0;
for(j = 0; j < no_of_frames; ++j){
if(frames[j] == pages[i]){
counter++;
time[j]         =
counter;  flag1  =
flag2     =     1;
break;}
}
if(flag1 == 0){
for(j = 0; j < no_of_frames; ++j){
if(frames[j] == -1){
counter++;
faults++;
frames[j] = pages[i];
time[j] = counter;
flag2 = 1;
break;
}
}
}
```

```
if(flag2 == 0){
pos = findLRU(time, no_of_frames);
counter++;
faults++;
frames[pos] =
pages[i]; time[pos] =
counter;
}
printf("\n");
for(j = 0; j < no_of_frames; ++j){
printf("%d\t", frames[j]);}
}
printf("\n\nTotal Page Faults = %d", faults);
printf("\n");
return 0;
}
```

Output:

FIFO:

```
nachiketa@nachiketa-VirtualBox: ~/Desktop/OSpracs/Prac 9
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac 9$ gcc fifo.c -o fifo
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac 9$ ./fifo

Enter Total Number of Pages:    8

Enter values of Reference String:
2
3
5
7
9
5
1
4

Enter Total Number of Frames:   3

2       -1      -1
2       3       -1
2       3       5
7       3       5
7       9       5
7       9       5
7       9       1
4       9       1
Total Page Faults:      7
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac 9$
```

LRU:

```
nachiketa@nachiketa-VirtualBox: ~/Desktop/OSpracs/Prac 9
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac 9$ gcc lru.c -o lru
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac 9$ ./lru
Enter number of frames: 3
Enter number of pages: 8
Enter reference string: 2
3
5
7
9
5
1
4

2       -1      -1
2       3       -1
2       3       5
7       3       5
7       9       5
7       9       5
1       9       5
1       4       5

Total Page Faults = 7
nachiketa@nachiketa-VirtualBox:~/Desktop/OSpracs/Prac 9$
```

Outcome:

Implemented various Page Replacement Algorithm and evaluated their performance.

Experiment No. 10

Aim: a. Write a C program to simulate File allocation strategies typically sequential files. b. Write a program in C to do FCFS disk scheduling.

Theory:

SEQUENTIAL FILE ALLOCATION IN THE OPERATING SYSTEM:

In the Sequential File Allocation method, the file is divided into smaller chunks and these chunks are then allocated memory blocks in the main memory. These smaller file chunks are stored one after another in a contiguous manner, this makes the file searching easier for the file allocation system.
The Contiguous (Sequential) File Allocation is one of the File Allocation Methods in the Operating System. The Other File Allocation Method is the Non-contiguous File Allocation which also has two types – first is the Linked File Allocation and the second is the Indexed File Allocation.

Why do we use the Sequential File Allocation method in the operating system?
The Sequential File Allocation or Contiguous File Allocation Method has an easy memory access advantage over the other two file allocation methods. In the contiguous File Allocation, the file is stored in sequential memory blocks and they are next to each other. So, when we have to search some files, we look into the directory (directory has the starting block address of each file) and reach the starting block where the file starts and from there, we will just read the next blocks in order to access the complete file. This access method also allows us to directly access the blocks of the memory as we can calculate easily where our required information is located.

FCFS DISK SCHEDULING:

Given an array of disk track numbers and initial head position, our task is to find the total number of seek operations done to access all the requested

tracks if First Come First Serve (FCFS) disk scheduling algorithm is used.
FCFS is the simplest disk scheduling algorithm. As the name suggests, this algorithm entertains requests in the order they arrive in the disk queue. The

algorithm looks very fair and there is no starvation (all requests are serviced sequentially) but generally, it does not provide the fastest service.

Algorithm: -

SEQUENTIAL FILE ALLOCATION:

STEP 1: Start the program.

STEP 2: Gather information about the number of files. STEP 3: Gather the memory requirement of each file.
STEP 4: Allocate the memory to the file in a sequential manner. STEP 5: Select any random location from the available location. STEP 6: Check if the location that is selected is free or not.
STEP 7: If the location is allocated set the flag = 1.

STEP 8: Print the file number, length, and the block allocated. STEP 9: Gather information if more files have to be stored.
STEP 10: If yes, then go to STEP 2. STEP 11: If no, Stop the program. **FCFS DISK SCHEDULING:**
STEP 1: Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. 'head' is the position of disk head.
STEP 2: Let us one by one take the tracks in default order and calculate the absolute distance of the track from the head.
STEP 3: Increment the total seek count with this distance.

STEP 4: Currently serviced track position now becomes the new head position. STEP 5: Go to step 2 until all tracks in request array have not been serviced.

Program:

SEQUENTIAL FILE ALLOCATION:

```
#include <stdio.h>
#include
<stdlib.h>
void recurse(int files[]){
```

```
    int flag = 0, startBlock, len, j, k, ch;
    printf("Enter the starting block and the length of the files: ");
    scanf("%d%d", &startBlock, &len);
    for (j=startBlock; j<(startBlock+len); j++){
        if (files[j] == 0)
            flag++;
    }
    if(len == flag){
        for (int k=startBlock; k<(startBlock+len); k++){
            if (files[k] == 0){
                files[k] = 1;
                printf("%d\t%d\n", k, files[k]);}
                                                      }
                                                      if (k != (startBlock+len-1))
                                                          printf("The file is allocated to the disk\n");
    }
    else
                                                      printf("The file is not allocated to the disk\n");
    printf("Do you want to enter more files?\n");
    printf("Press 1 for YES, 0 for NO: ");
    scanf("%d", &ch);
    if (ch == 1)
        recurse(files);
    else
        exit(0);
    return;
}
int main(){
int
files[50];
for(int i=0;i<50;i++)
files[i]=0;
printf("Files Allocated are :\n");
recurse(files);
return 0;}
```

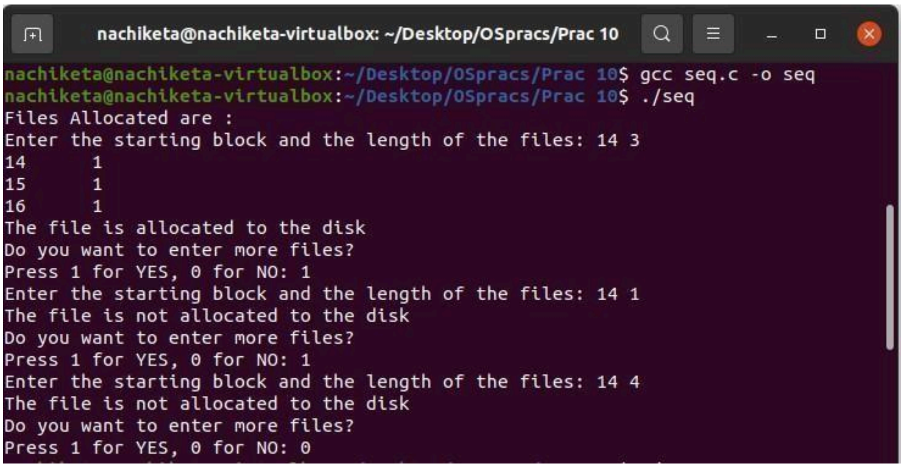
FCFS DISK SCHEDULING:

```
#include<stdio.h>
int main(){
            int queue[20],n,head,i,j,k,seek=0,max,diff;
            float avg;
            printf("Enter the max range of disk\n");
            scanf("%d",&max);
            printf("Enter the size of queue request\n");
            scanf("%d",&n);
            printf("Enter the queue of disk positions to be read\n");
            for(i=1;i<=n;i++)
            scanf("%d",&queue[i]);
            printf("Enter the initial head position\n");
            scanf("%d",&head);
            queue[0]=head;
            for(j=0;j<=n-1;j++)
            {
                        diff=abs(queue[j+1]-queue[j]);
                        seek+=diff;
                        printf("Disk head moves from %d to %d with seek
%d\n",queue[j],queue[j+1],diff);
            }
            printf("Total seek time is %d\n",seek);
```

```
avg=seek/(float)n;
printf("Average seek time is %f\n",avg);
return 0;
}
```

Output:

SEQUENTIAL FILE ALLOCATION:



FCFS DISK SCHEDULING:

```
nachiketa@nachiketa-virtualbox: ~/Desktop/OSpracs/Prac 10
nachiketa@nachiketa-virtualbox:~/Desktop/OSpracs/Prac 10$ gcc fcfs.c -o fcfs
fcfs.c: In function 'main':
fcfs.c:18:30: warning: implicit declaration of function 'abs' [-Wimplicit-functi
on-declaration]
   18 |                         diff=abs(queue[j+1]-queue[j]);
      |                              ^~~
nachiketa@nachiketa-virtualbox:~/Desktop/OSpracs/Prac 10$ ./fcfs
Enter the max range of disk
200
Enter the size of queue request
8
Enter the queue of disk positions to be read
23
45
67
89
23
56
78
12
Enter the initial head position
40
Disk head moves from 40 to 23 with seek
                                        17
Disk head moves from 23 to 45 with seek
                                        22
Disk head moves from 45 to 67 with seek
                                        22
Disk head moves from 67 to 89 with seek
                                        22
Disk head moves from 89 to 23 with seek
                                        66
Disk head moves from 23 to 56 with seek
                                        33
Disk head moves from 56 to 78 with seek
                                        22
Disk head moves from 78 to 12 with seek
                                        66
Total seek time is 270
Average seek time is 33.750000
nachiketa@nachiketa-virtualbox:~/Desktop/OSpracs/Prac 10$
```

Outcome: Demonstrated and analysed concepts of file management and I/O management techniques

Shree Rahul Education Society's (Regd.)

SHREE L. R. TIWARI

COLLEGE OF ENGINEERING

(Approved by AICTE & DTE, Maharashtra State & Affiliated to University of Mumbai)
NAAC Accredited, NBA Accredited Program, ISO 9001:2015 Certified | DTE Code No. : 3423
Minority Status (Hindi Linguistic)

Experiment No. 11

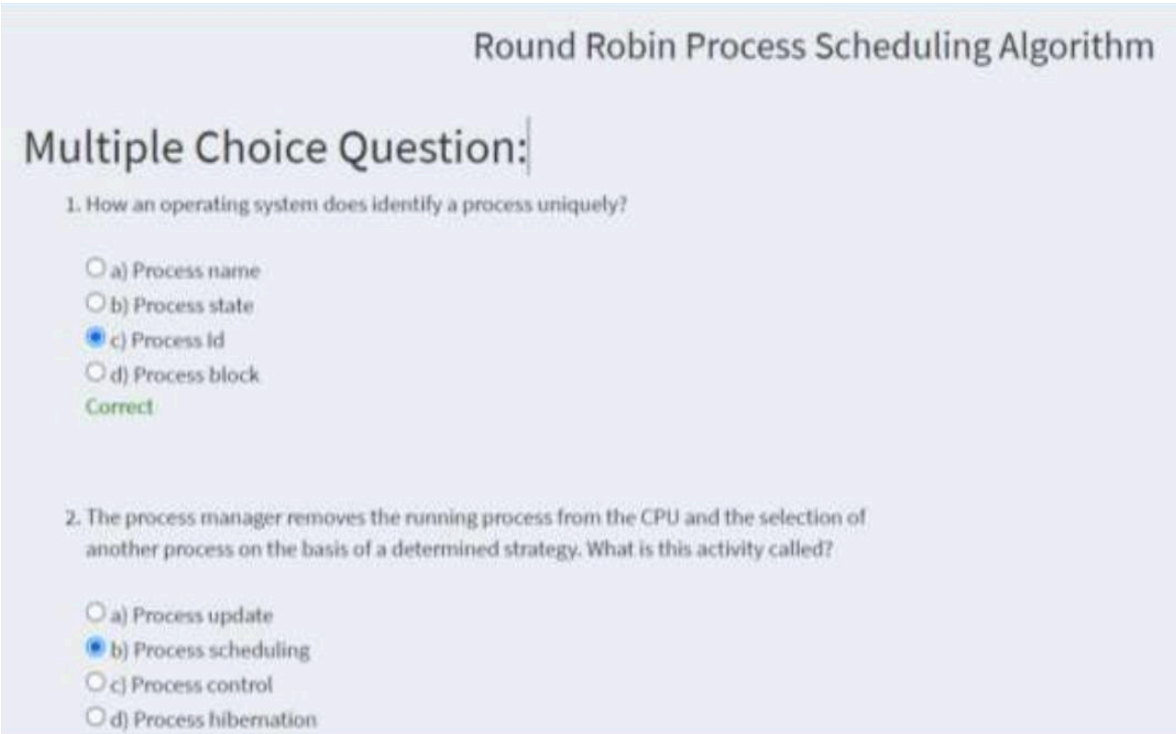
(V-lab)

Aim: How Round Robin Algorithm Schedules the Processes.

Source:

http://vlabs.iitb.ac.in/vlabs-dev/vlab_bootcamp/bootcamp/CRUX/labs/ exp1/index.html

Pretest:



3. Which types of process schedulers are available?

- ○ a) Long term scheduler
- ○ b) Short term scheduler
- ○ c) Medium term scheduler
- ◉ d) All of the above

Correct

4. Which type of scheduler is responsible for suspending and resuming the process?

- ○ a) Long term scheduler
- ○ b) Short term scheduler
- ◉ c) Medium term scheduler
- ○ d) All of the above

Correct

5. Which queue keeps all processes of main memory, ready and waiting to execute?

- ◉ a) Ready Queue
- ○ b) Device Queue
- ○ c) Input Output queue
- ○ d) Waiting queue

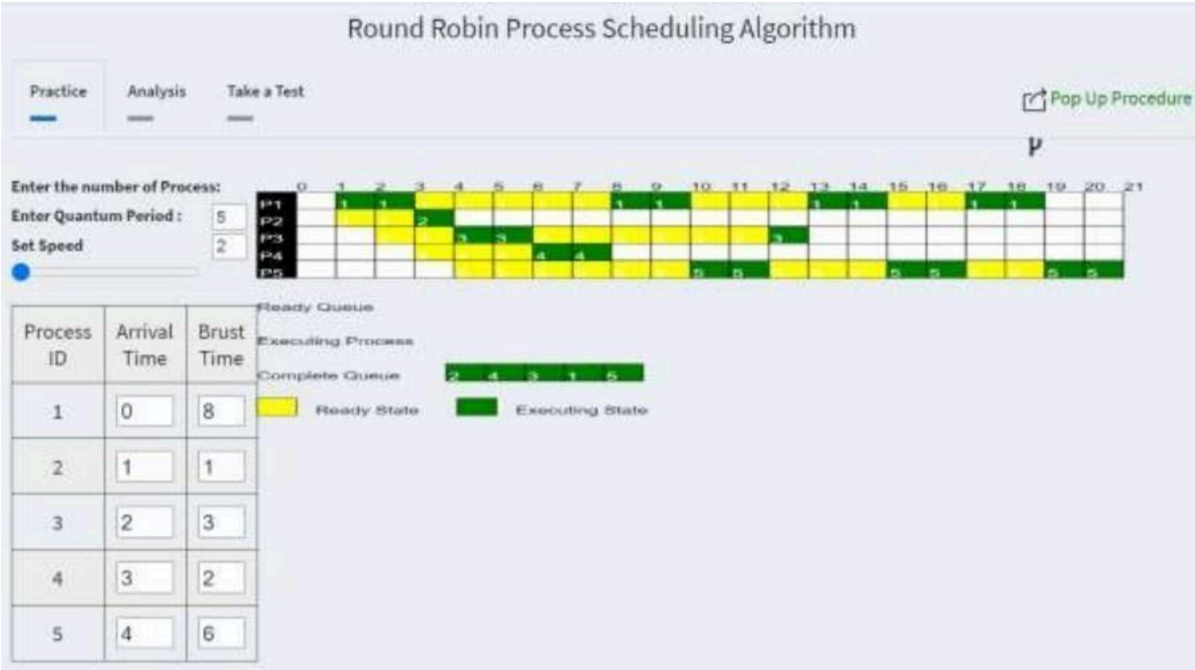
Correct

Submit

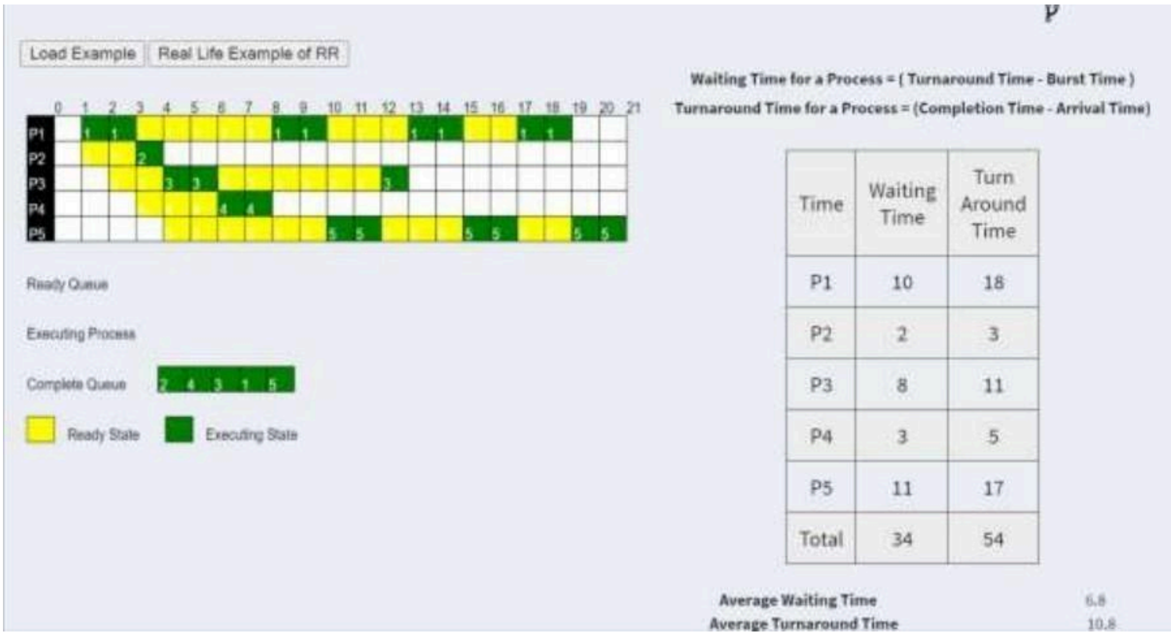
Your Score: 5/5

Simulation: -


Round Robin Process Scheduling Algorithm
Practice
Analysis
Take a Test
Enter the number of Process:
Enter Quantum Period :
5
Set Speed
2
Process ID
Arrival Time
Brust Time
1
0
8
2
1
1
3
2
3
4
3
2
5
4
6

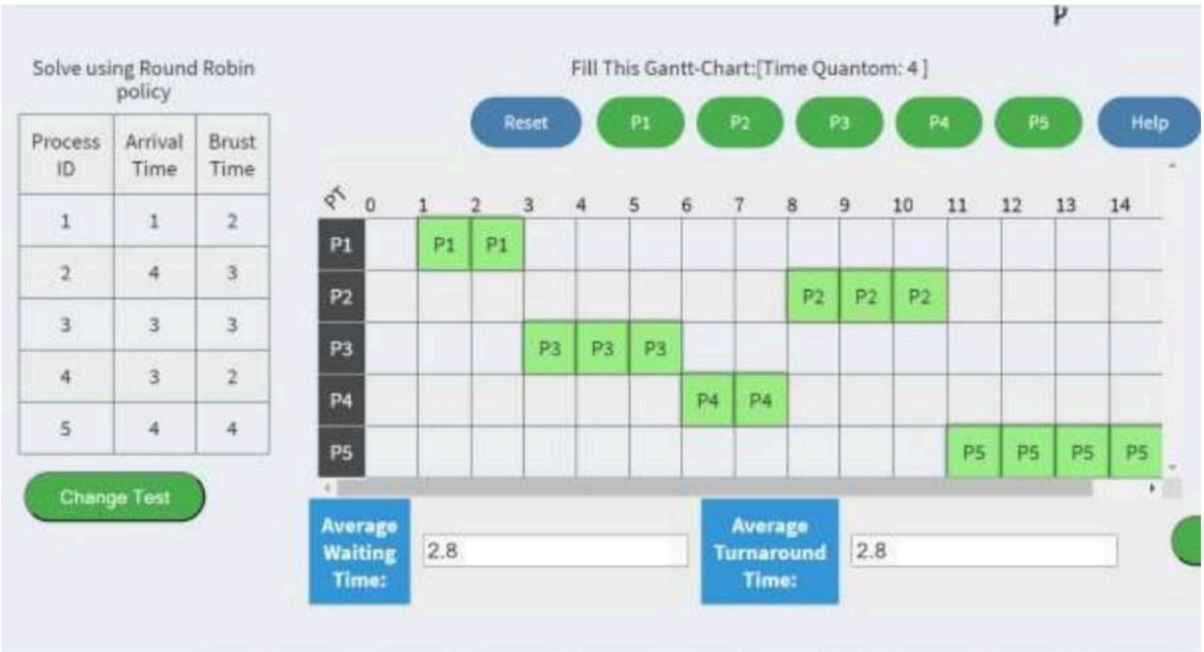


Process ID	Arrival Time	Brust Time
1	0	8
2	1	1
3	2	3
4	3	2
5	4	6



Time	Waiting Time	Turn Around Time
P1	10	18
P2	2	3
P3	8	11
P4	3	5
P5	11	17
Total	34	54

Average Waiting Time 6.8
Average Turnaround Time 10.8


Solve using Round Robin policy
Fill This Gantt-Chart:[Time Quantom: 4]
Reset
P1
P2
P3
P4
P5
Help
Process ID
Arrival Time
Brust Time
1 1 2
2 4 3
3 3 3
4 3 2
5 4 4
Change Test
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Average Waiting Time:
2.8
Average Turnaround Time:
2.8

Postetst:

Round Robin Process Scheduling Algorithm

Multiple Choice Question:

1. Scheduling is:

- (•) a) Allowing a job to use the processor
- ( ) b) Making proper use of processor
- ( ) c) All of the mentioned
- ( ) c) None of the mentioned

Correct

2. If the quantum time of round robin algorithm is very large, then it is equivalent to:

- (•) a) First in first out
- ( ) b) Shortest Job Next
- ( ) c) Lottery scheduling
- ( ) d) None of the above

Correct

3. An optimal scheduling algorithm in terms of minimizing the average waiting time of a given set of processes is ________.

- ( ) a) FCFS scheduling algorithm
- ( ) b) Round robin scheduling algorithm
- (•) c) Shortest job - first scheduling algorithm
- ( ) d) None of the above

Correct

4. Which of the following is a criterion to evaluate a scheduling algorithm?

- ( ) a) CPU Utilization: Keep CPU utilization as high as possible
- ( ) b) Throughput: number of processes completed per unit time
- ( ) c) Waiting Time: Amount of time spent ready to run but not running
- (•) d) All of the above

Correct

5. In interactive environments such as time-sharing systems, the primary requirement is to provide reasonably good response time and in general, to share system resources equitably. In such situations, the scheduling algorithm that is most popularly applied is ________.

- ( ) a) Shortest Remaining Time Next (SRTN) Scheduling
- ( ) b) Priority Based Preemptive Scheduling
- (•) c) Round Robin Scheduling
- ( ) d) None of the above

Correct

6. A scheduling algorithm is fair

- (selected) a) If no process faces starvation
- b) If a process is starved, detect it and run it with high priority
- c) If it uses semaphores
- d) Only if a queue is used for scheduling

Correct

7. Which of the following do not belong to queues for processes?

- a) Job Queue
- (selected) b) PCB queue
- c) Device Queue
- d) Ready Queue

Correct

8. Consider the following set of processes, the length of the CPU burst time given in milliseconds:

Process	Burst time
P1	6
P2	8
P3	7
P4	3

Assuming the above process being scheduled with the SJF scheduling algorithm:

- (•) a) The waiting time for process P1 is 3ms.
- ( ) b) The waiting time for process P1 is 0ms.
- ( ) c) The waiting time for process P1 is 16ms.
- ( ) d) The waiting time for process P1 is 9ms.

Correct

9. Shortest Job First executes first the job:

- (•) a) With the least processor needs
- ( ) b) That first entered the queue
- ( ) c) That has been in the queue for the longest
- ( ) d) That last entered the queue

Correct

10. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called:

- ( ) a) Job queue
- (•) b) Ready queue
- ( ) c) Execution queue
- ( ) d) Process queue

Correct

Submit

Outcome:

Hence, we have performed Round-Robin Scheduling Algorithm on virtual lab.