**NAME: Meet Raut**

**DIV: S21**

**ROLL.NO: 2201084**

## Experiment 1:

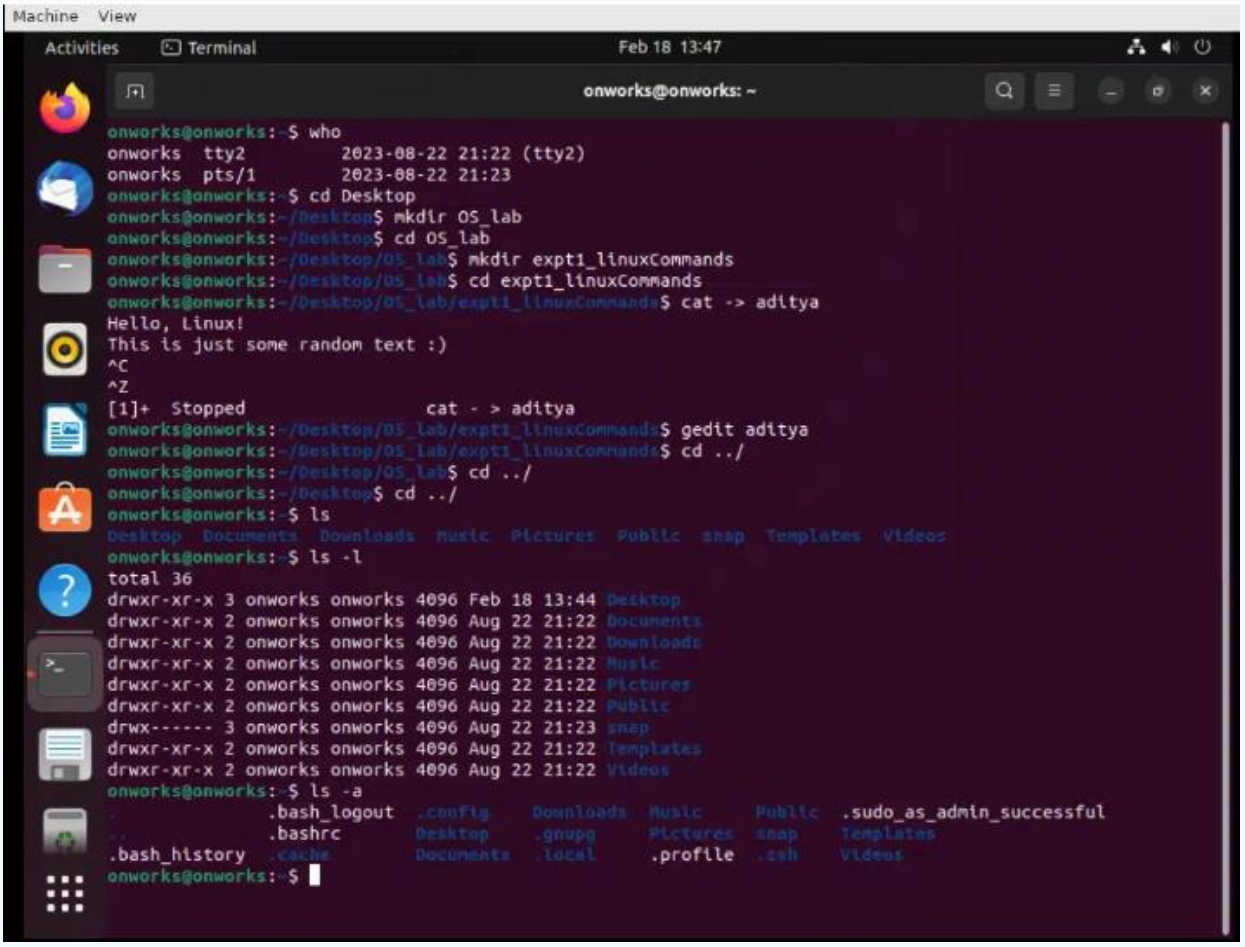
- **AIM: Explore usage of basic linux commands and system calls for files, directory and process management.**

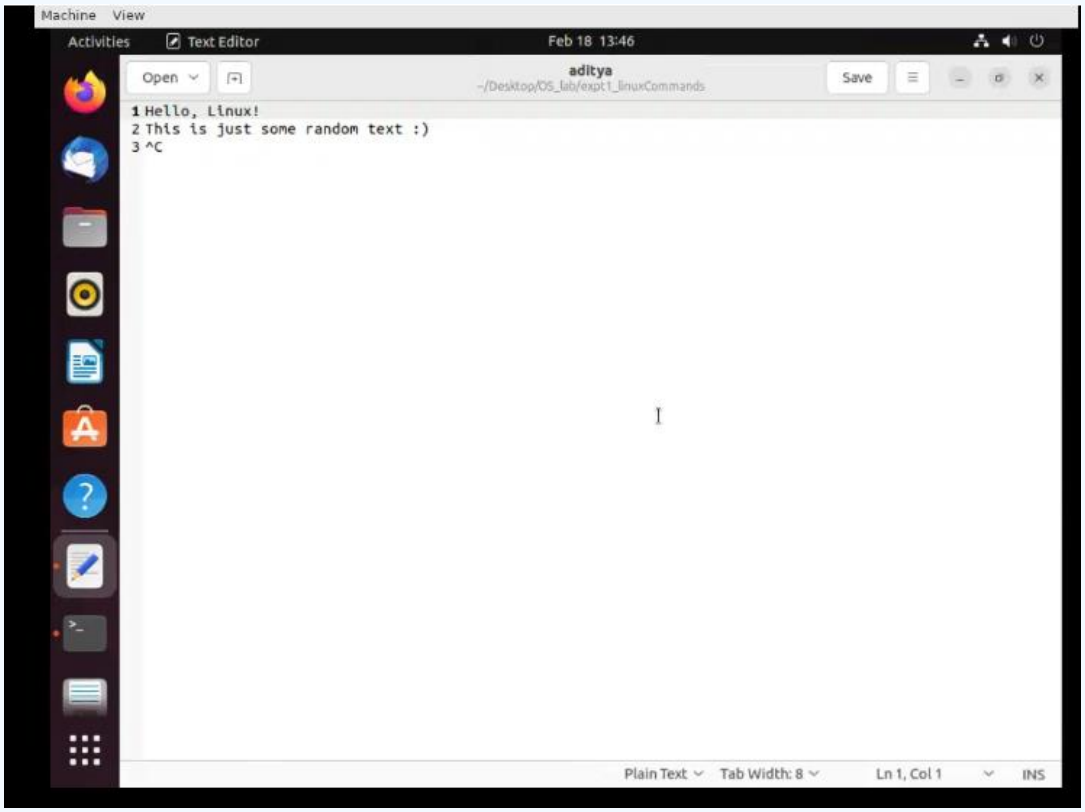
- **THEORY:**

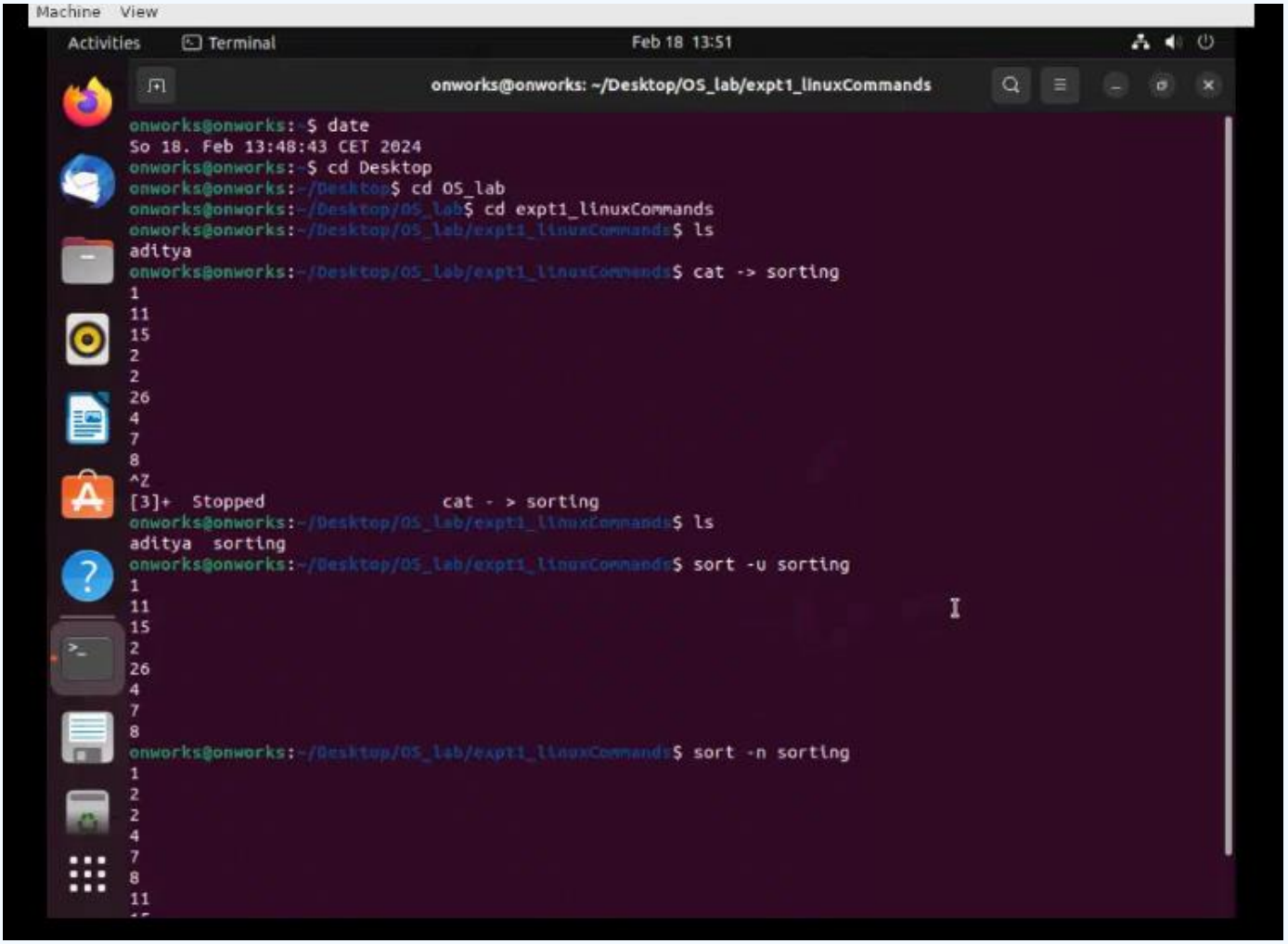
1. who : it is used to find out the current user who is logged into the system.
2. pwd : present working directory, lets you know the current directory you are in.
3. cal : show the calendar of the complete month.
4. date : It shows you the current date, along with the time, along with the day, along with the year.
5. mkdir: to create a new directory under any directory
6. chdir/cd : to change the current working directory
7. cat : to create the file and display the contents of the file
8. chmod: to change the mode of the file. There are three modes read(r), write(w) and execute(e)
9. ls : to list all directories and subdirectories
    a. ls-l : to show the long listing information about the directory
    b. ls-lh : human readable format.
    c. ls-ld : shows the details of the directory content.
    d. ls-d* : to show the sub directories in a directory
    e. ls-a : to show hidden files
    f. ls-lhs : show files in the descending order in which you have used your files.
10. sort-r file name.txt : sorts the list in reverse order
11. sort-n file name.txt : its sorts the numerical list in ascending order
12. sort nr file name.txt : its sorts the numerical list in reverse order
13. sort u file name.txt : to remove the duplicates
14. sort m file name.txt : Sorts the months in ascending order
15. awk : it is used for the user that defines text patterns that are to be searched for each line of the file.

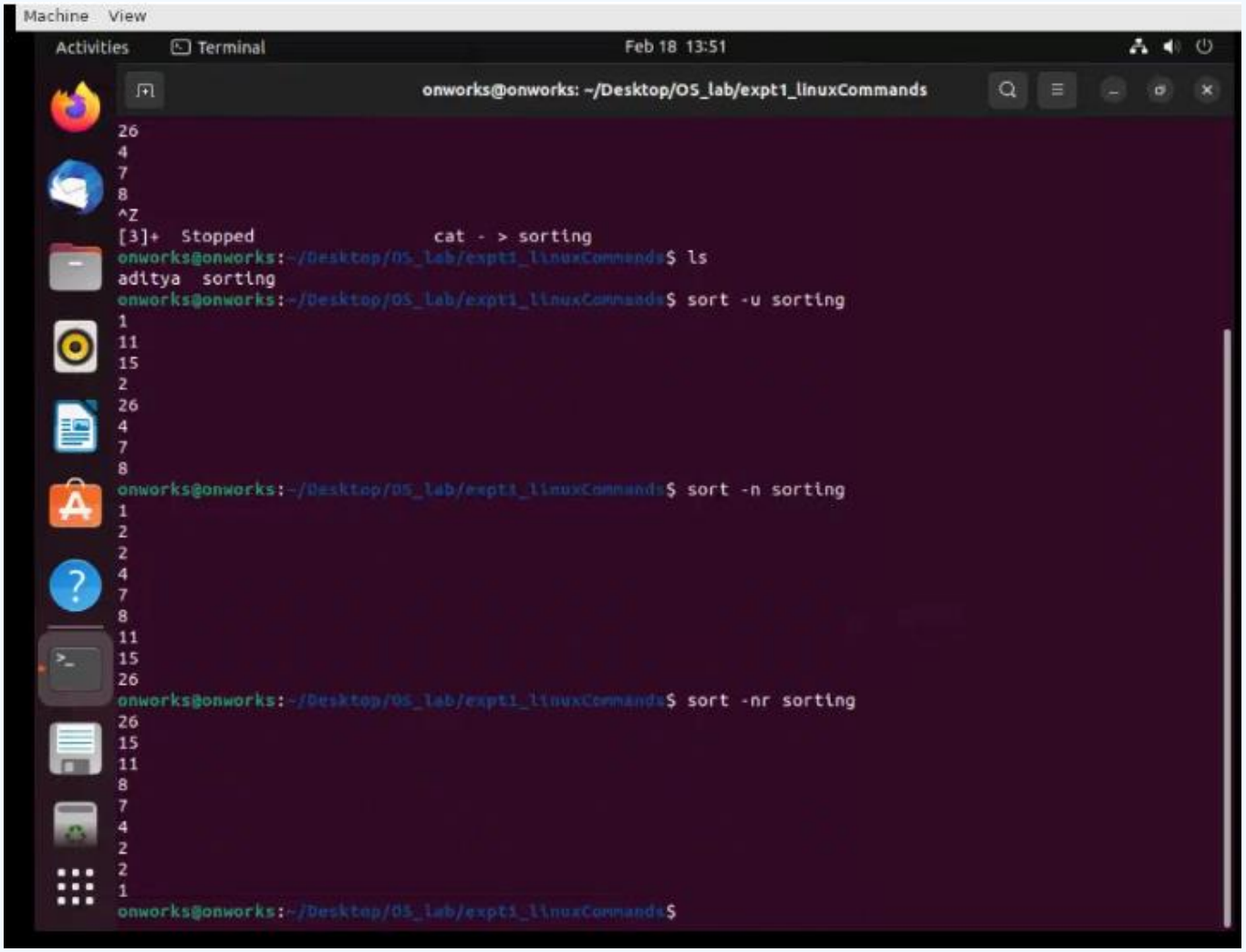
Syntax , awk '{print}' file name.txt awk
'/faculty/{print}' file name.txt : awk
'{print}NR, $0}' file name.txt :
NR - specifies the number of lines.

- **SCREENSHOTS:**




Machine View
Activities
Terminal
Feb 18 13:51
onworks@onworks: ~/Desktop/OS_lab/expt1_linuxCommands
onworks@onworks:~$ date
So 18. Feb 13:48:43 CET 2024
onworks@onworks:~$ cd Desktop
onworks@onworks:~/Desktop$ cd OS_lab
onworks@onworks:~/Desktop/OS_lab$ cd expt1_linuxCommands
onworks@onworks:~/Desktop/OS_lab/expt1_linuxCommands$ ls
aditya
onworks@onworks:~/Desktop/OS_lab/expt1_linuxCommands$ cat -> sorting
1
11
15
2
2
26
4
7
8
^Z
[3]+ Stopped cat - > sorting
onworks@onworks:~/Desktop/OS_lab/expt1_linuxCommands$ ls
aditya sorting
onworks@onworks:~/Desktop/OS_lab/expt1_linuxCommands$ sort -u sorting
1
11
15
2
26
4
7
8
onworks@onworks:~/Desktop/OS_lab/expt1_linuxCommands$ sort -n sorting
1
2
2
4
7
8
11




- **<u>CONCLUSION:</u> Thus, we have successfully explored and implemented usage of basic linux commands and system calls for files, directory and process management.**

**NAME: Meet Raut**

**DIV: S21**

**ROLL.NO: 2201084**

## Experiment 2:

- **AIM: To study the shell scripting language**
  **A display OS version release number kernel version**
  **B display top 10 processors in descending order**
  **C display processes with highest memory usage**
  **D display c+**

- **THEORY:**

**A]**

i) uname - r{username}
To show the OS version, release no. OS release version
ii) uname - v
Shows Kernel version (last when launched)
iii) uname - a
Shows all the details including OS used, PC and other details.
us - c + u - r → us - a

**B]**

i) ps-aux (i) sort : nk +41 tail
Sorts 10 processes in descending order. It shows logged in users, modes of memory & storage.
OR
ii) ps-aux | tail
Same as above
OR
iii) ps-aux | sort -nl+4 | tail - n15 (shows top 15 processes)
same with 15

**C]**

i) sudo apt install htop
shows date of processes with highest memory
ii) ps - eo pid, ppid, cmd, % mem, % cpu –sort =-% mem | head
same as above same as above

**D]**

i) ps-p$$

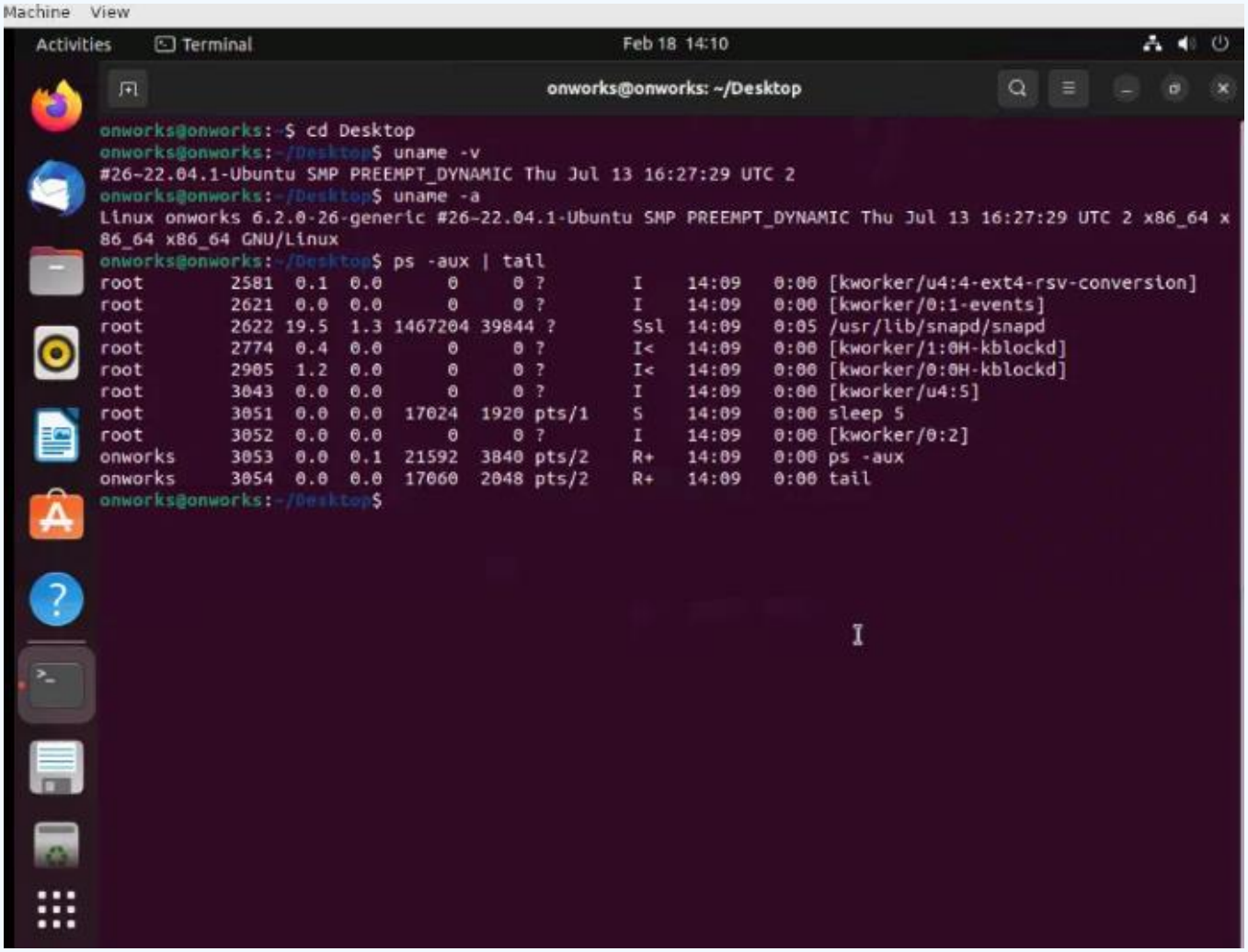
Shows process ID and current shell

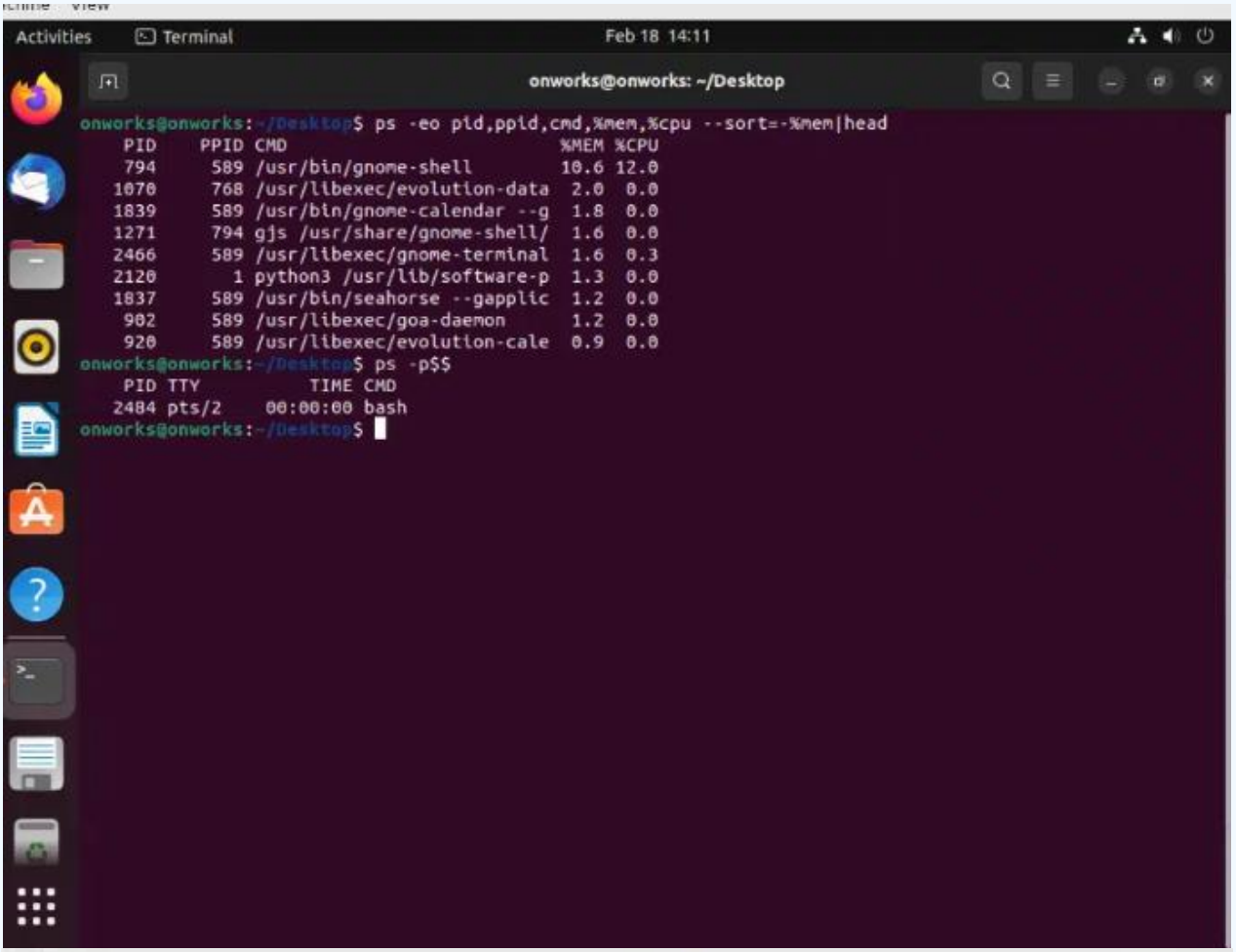
ii) echo display home directory

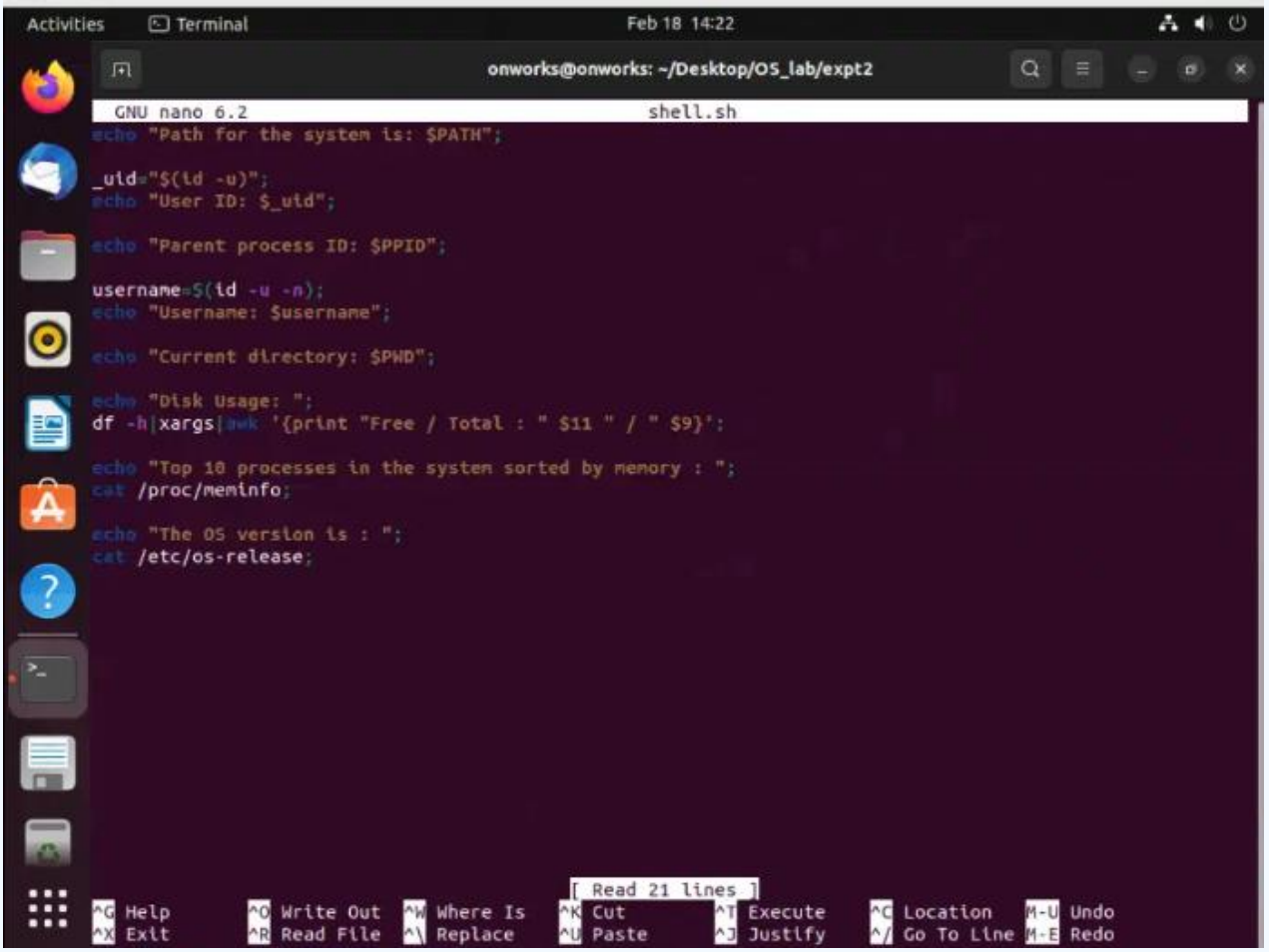
echo $ home

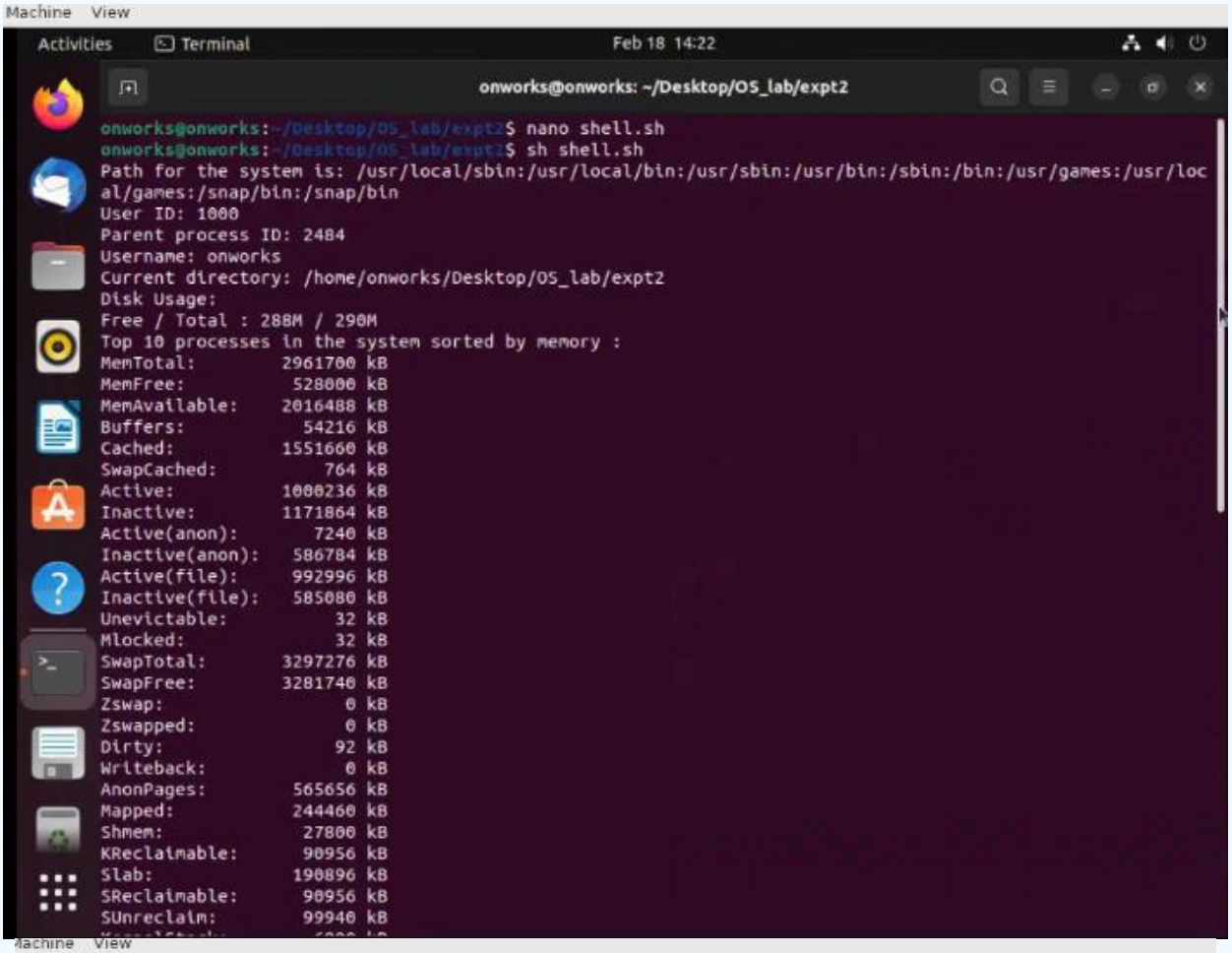
To display home directory

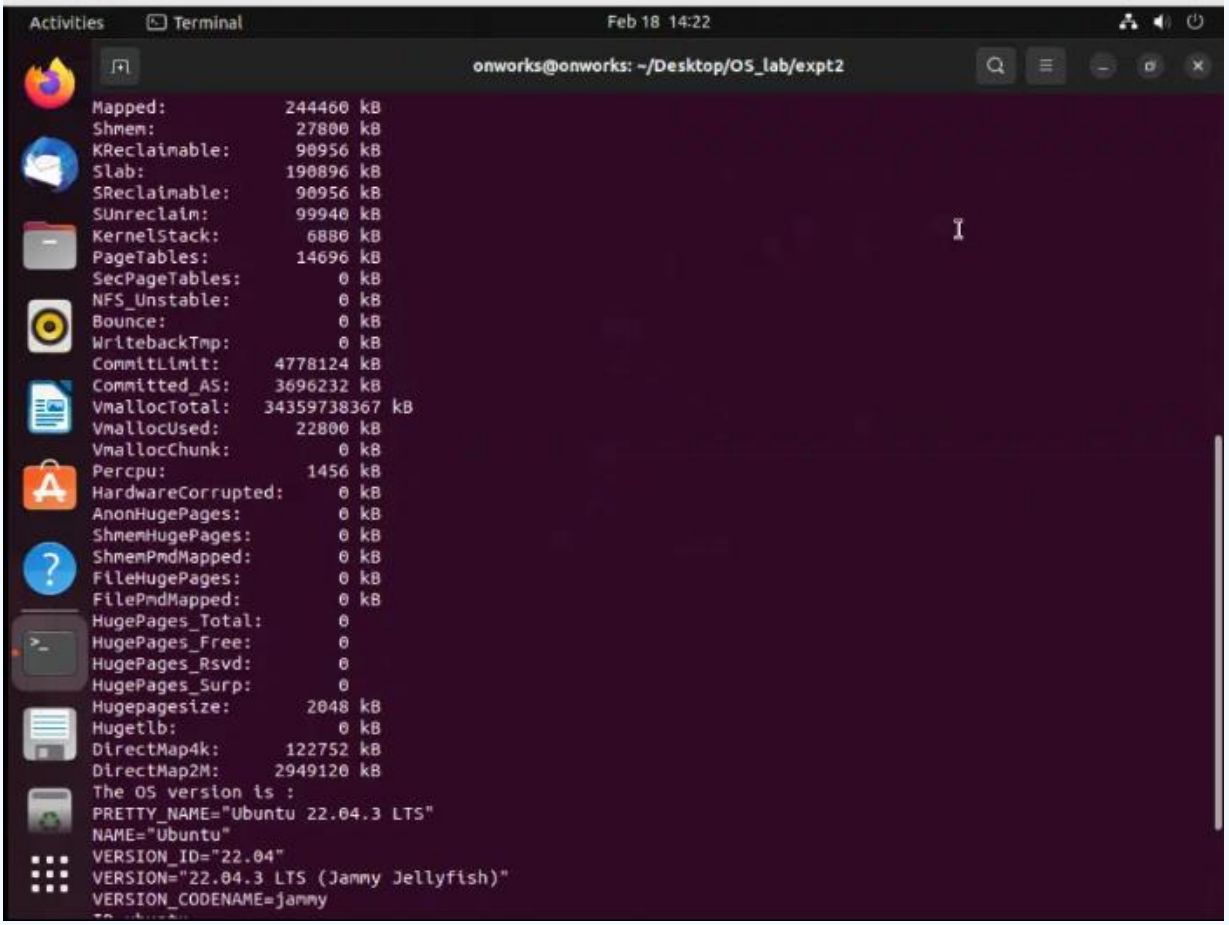
- **SCREENSHOTS:**

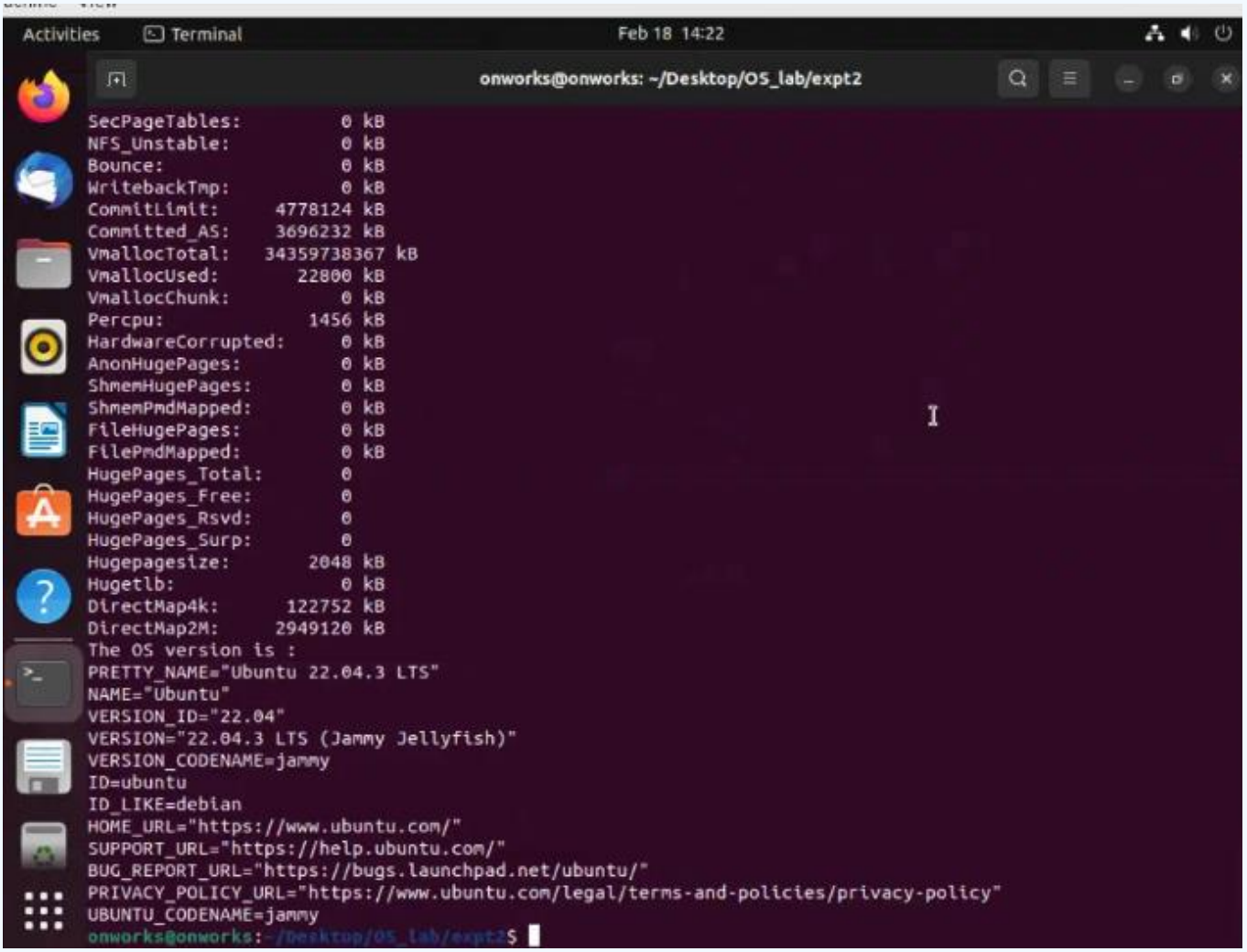



Activities
Terminal
Feb 18 14:11
onworks@onworks: ~/Desktop
onworks@onworks:~/Desktop$ ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem|head
PID PPID CMD %MEM %CPU
794 589 /usr/bin/gnome-shell 10.6 12.0
1070 768 /usr/libexec/evolution-data 2.0 0.0
1839 589 /usr/bin/gnome-calendar --g 1.8 0.0
1271 794 gjs /usr/share/gnome-shell/ 1.6 0.0
2466 589 /usr/libexec/gnome-terminal 1.6 0.3
2120 1 python3 /usr/lib/software-p 1.3 0.0
1837 589 /usr/bin/seahorse --gapplic 1.2 0.0
902 589 /usr/libexec/goa-daemon 1.2 0.0
920 589 /usr/libexec/evolution-cale 0.9 0.0
onworks@onworks:~/Desktop$ ps -p$$
PID TTY TIME CMD
2484 pts/2 00:00:00 bash
onworks@onworks:~/Desktop$


Activities
Terminal
Feb 18 14:22
onworks@onworks: ~/Desktop/OS_lab/expt2
GNU nano 6.2 shell.sh
echo "Path for the system is: $PATH";
_uid="$(id -u)";
echo "User ID: $_uid";
echo "Parent process ID: $PPID";
username=$(id -u -n);
echo "Username: $username";
echo "Current directory: $PWD";
echo "Disk Usage: ";
df -h|xargs|awk '{print "Free / Total : " $11 " / " $9}';
echo "Top 10 processes in the system sorted by memory : ";
cat /proc/meminfo;
echo "The OS version is : ";
cat /etc/os-release;
[ Read 21 lines ]
^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute ^C Location M-U Undo
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line M-E Redo


Machine View
Activities
Terminal
Feb 18 14:22
onworks@onworks: ~/Desktop/OS_lab/expt2
onworks@onworks:~/Desktop/OS_lab/expt2$ nano shell.sh
onworks@onworks:~/Desktop/OS_lab/expt2$ sh shell.sh
Path for the system is: /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/snap/bin
User ID: 1000
Parent process ID: 2484
Username: onworks
Current directory: /home/onworks/Desktop/OS_lab/expt2
Disk Usage:
Free / Total : 288M / 290M
Top 10 processes in the system sorted by memory :
MemTotal: 2961700 kB
MemFree: 528000 kB
MemAvailable: 2016488 kB
Buffers: 54216 kB
Cached: 1551660 kB
SwapCached: 764 kB
Active: 1000236 kB
Inactive: 1171864 kB
Active(anon): 7240 kB
Inactive(anon): 586784 kB
Active(file): 992996 kB
Inactive(file): 585080 kB
Unevictable: 32 kB
Mlocked: 32 kB
SwapTotal: 3297276 kB
SwapFree: 3281740 kB
Zswap: 0 kB
Zswapped: 0 kB
Dirty: 92 kB
Writeback: 0 kB
AnonPages: 565656 kB
Mapped: 244460 kB
Shmem: 27800 kB
KReclaimable: 90956 kB
Slab: 190896 kB
SReclaimable: 90956 kB
SUnreclaim: 99940 kB


Machine View
Activities
Terminal
Feb 18 14:22
onworks@onworks: ~/Desktop/OS_lab/expt2
Mapped: 244460 kB
Shmem: 27800 kB
KReclaimable: 90956 kB
Slab: 190896 kB
SReclaimable: 90956 kB
SUnreclaim: 99940 kB
KernelStack: 6880 kB
PageTables: 14696 kB
SecPageTables: 0 kB
NFS_Unstable: 0 kB
Bounce: 0 kB
WritebackTmp: 0 kB
CommitLimit: 4778124 kB
Committed_AS: 3696232 kB
VmallocTotal: 34359738367 kB
VmallocUsed: 22800 kB
VmallocChunk: 0 kB
Percpu: 1456 kB
HardwareCorrupted: 0 kB
AnonHugePages: 0 kB
ShmemHugePages: 0 kB
ShmemPmdMapped: 0 kB
FileHugePages: 0 kB
FilePmdMapped: 0 kB
HugePages_Total: 0
HugePages_Free: 0
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 2048 kB
Hugetlb: 0 kB
DirectMap4k: 122752 kB
DirectMap2M: 2949120 kB
The OS version is :
PRETTY_NAME="Ubuntu 22.04.3 LTS"
NAME="Ubuntu"
VERSION_ID="22.04"
VERSION="22.04.3 LTS (Jammy Jellyfish)"
VERSION_CODENAME=jammy




- **<u>CONCLUSION:</u> Thus, we have successfully studied and implemented shell scripting.**

**NAME: Meet Raut**

**DIV: S21**

**ROLL.NO: 2201084**

## Experiment 3:

- **AIM: Implement Basic Commands of Linux like ls, cp, mv using Kernel APIs.**
- **THEORY:**

cp → Copies the file.

cp -r → Copies recursively (directories).

cp -b → Copies the backups of the destination file in the source folder.

cp -i → Asks for confirmation before overwriting the destination file.

cp -f → Forces copying by deleting the destination file if necessary.

cp -v → Enables verbose mode, showing which files are being copied.

cp -p → Preserves file attributes such as modification time, access time, owners, and permissions.

Syntax: cp -p source_file destination_file.

cp -l → Creates a hard link file.

mv → Moves or renames a file.

mv -i → Asks for confirmation before moving the file.

mv -f → Forces moving by deleting the destination file if necessary.

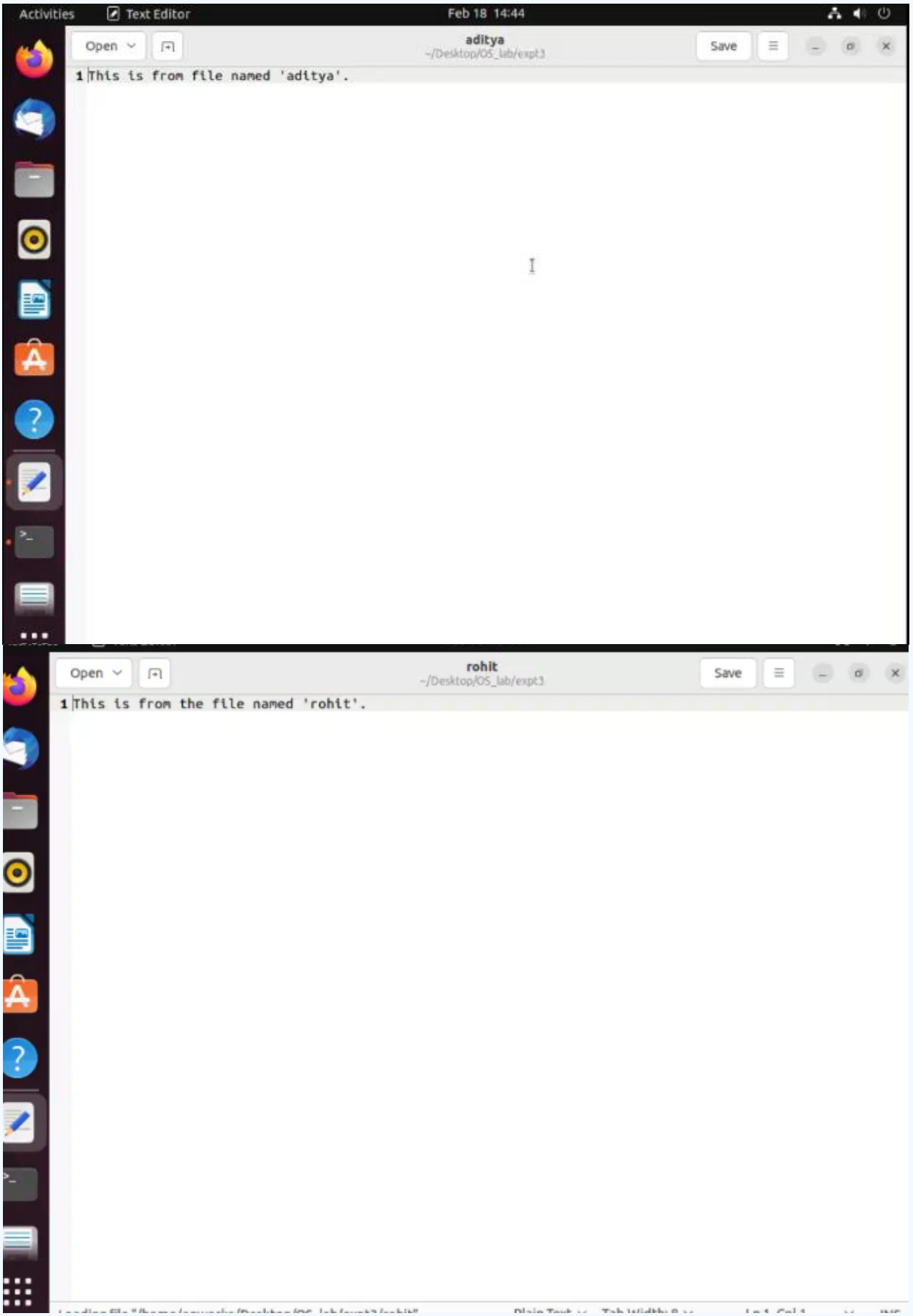
Syntax: mv -f source_file destination_file.

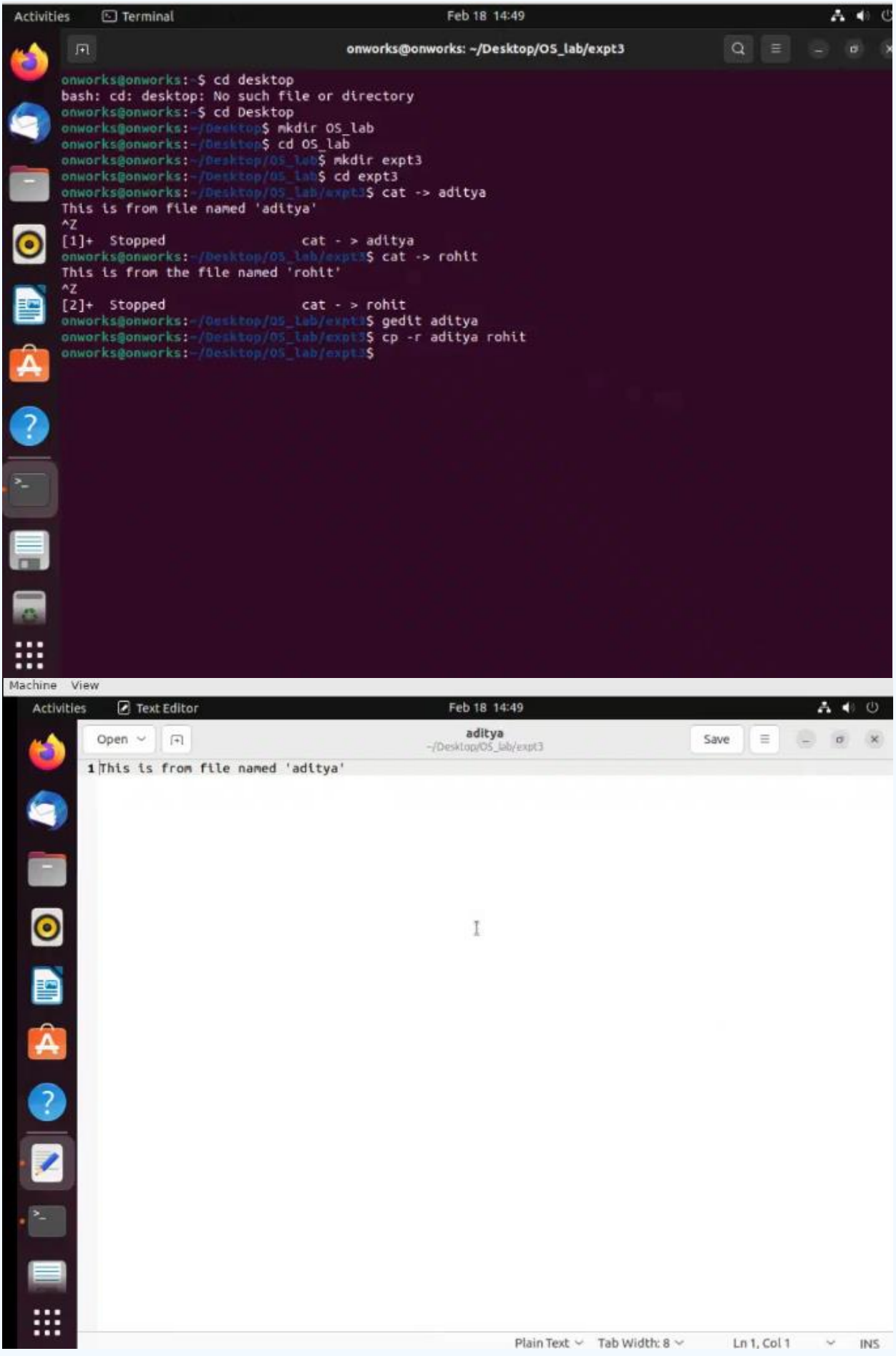
mv -n → Prevents overwriting an existing file.

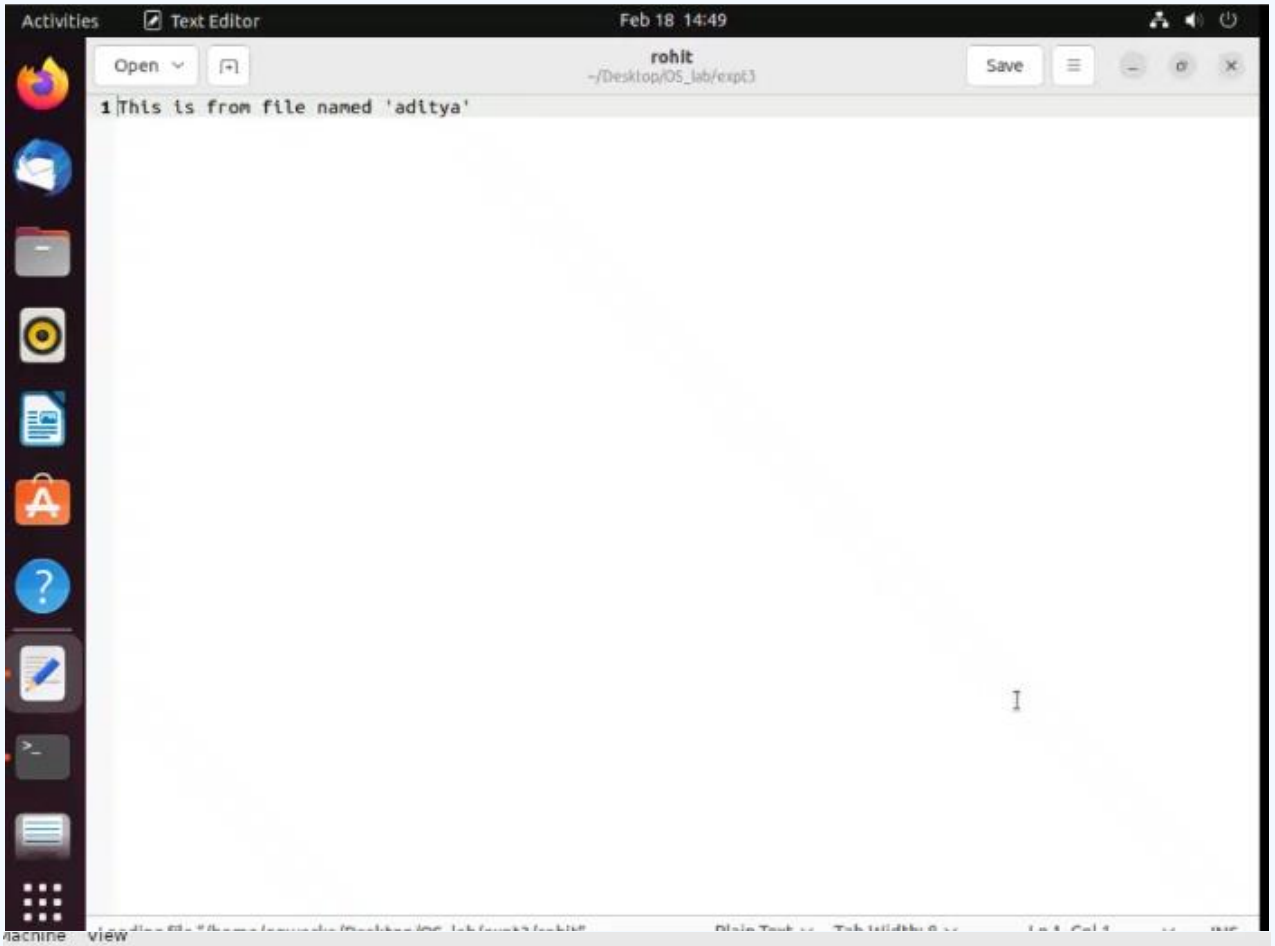
Syntax: mv -n source_file destination_file.

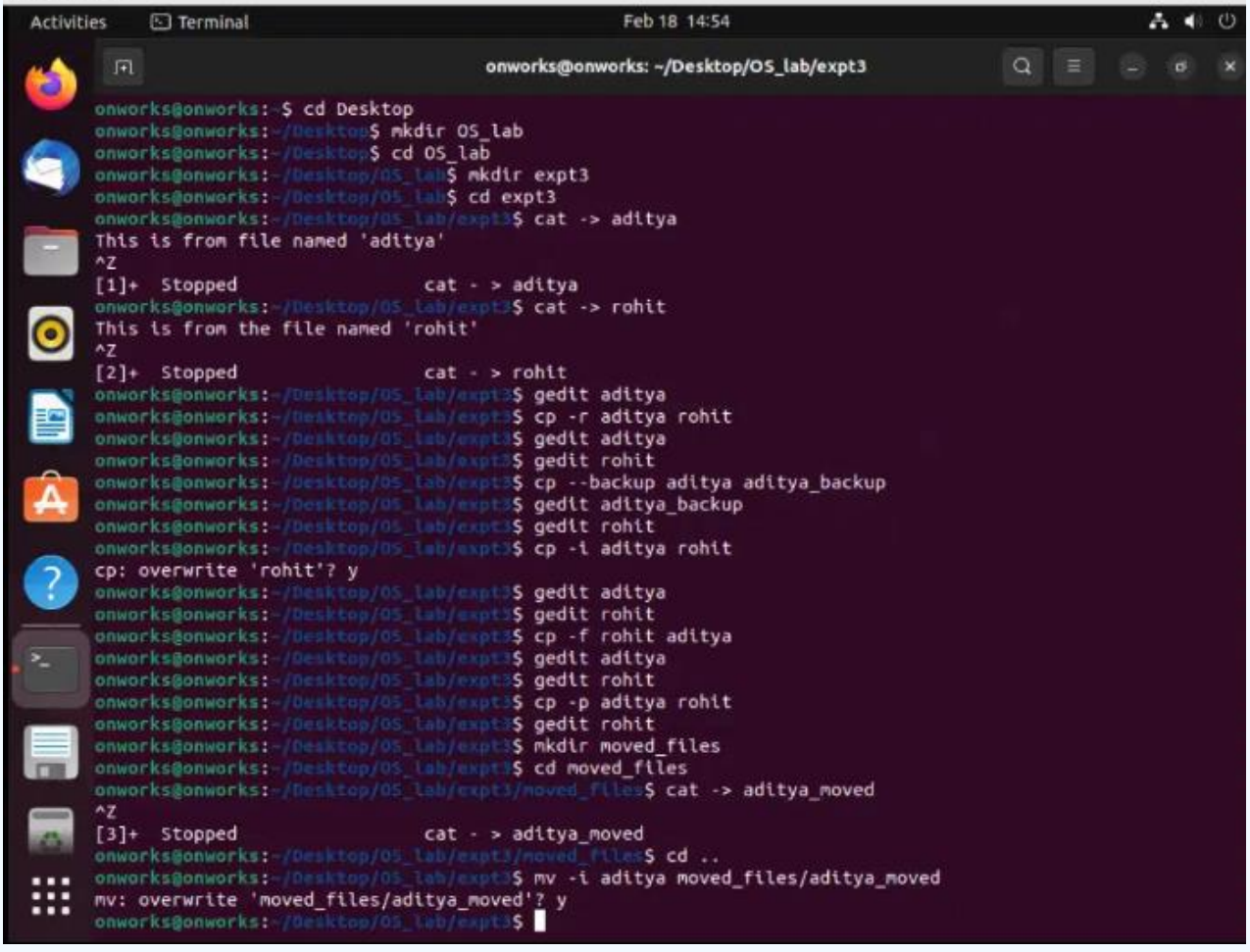
mv -b → Takes a backup of an existing file.

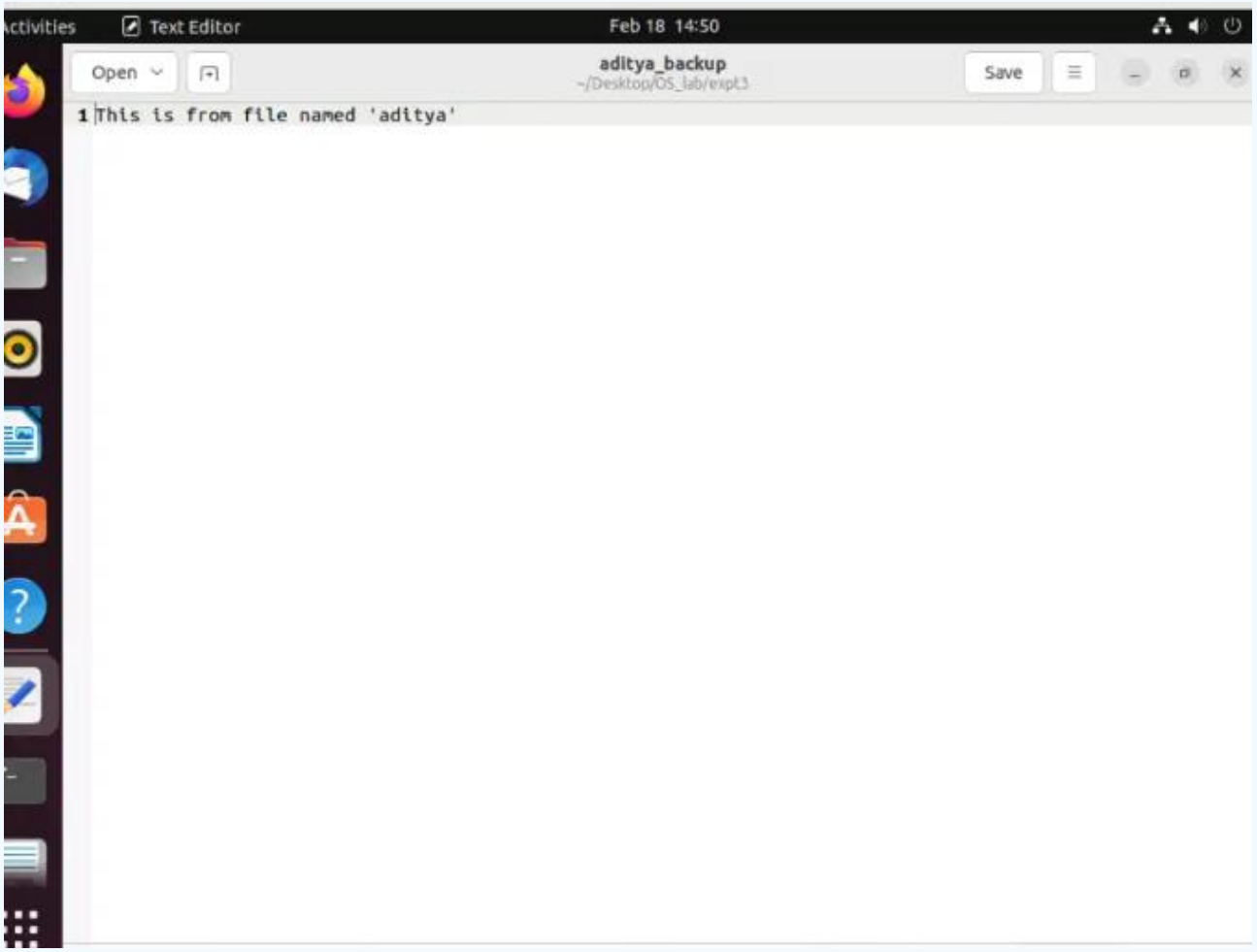
- **SCREENSHOTS:**




Activities
Terminal
Feb 18 14:49
onworks@onworks: ~/Desktop/OS_lab/expt3
onworks@onworks:~$ cd desktop
bash: cd: desktop: No such file or directory
onworks@onworks:~$ cd Desktop
onworks@onworks:~/Desktop$ mkdir OS_lab
onworks@onworks:~/Desktop$ cd OS_lab
onworks@onworks:~/Desktop/OS_lab$ mkdir expt3
onworks@onworks:~/Desktop/OS_lab$ cd expt3
onworks@onworks:~/Desktop/OS_lab/expt3$ cat -> aditya
This is from file named 'aditya'
^Z
[1]+ Stopped cat - > aditya
onworks@onworks:~/Desktop/OS_lab/expt3$ cat -> rohit
This is from the file named 'rohit'
^Z
[2]+ Stopped cat - > rohit
onworks@onworks:~/Desktop/OS_lab/expt3$ gedit aditya
onworks@onworks:~/Desktop/OS_lab/expt3$ cp -r aditya rohit
onworks@onworks:~/Desktop/OS_lab/expt3$
Machine View
Activities
Text Editor
Feb 18 14:49
Open
aditya
~/Desktop/OS_lab/expt3
Save
1 This is from file named 'adltya'
Plain Text
Tab Width: 8
Ln 1, Col 1
INS


Activities
Text Editor
Feb 18 14:49
Open
rohit
~/Desktop/OS_lab/expt3
Save
1 This is from file named 'aditya'


Activities
Terminal
Feb 18 14:54
onworks@onworks: ~/Desktop/OS_lab/expt3
onworks@onworks:~$ cd Desktop
onworks@onworks:~/Desktop$ mkdir OS_lab
onworks@onworks:~/Desktop$ cd OS_lab
onworks@onworks:~/Desktop/OS_lab$ mkdir expt3
onworks@onworks:~/Desktop/OS_lab$ cd expt3
onworks@onworks:~/Desktop/OS_lab/expt3$ cat -> aditya
This is from file named 'aditya'
^Z
[1]+ Stopped cat - > aditya
onworks@onworks:~/Desktop/OS_lab/expt3$ cat -> rohit
This is from the file named 'rohit'
^Z
[2]+ Stopped cat - > rohit
onworks@onworks:~/Desktop/OS_lab/expt3$ gedit aditya
onworks@onworks:~/Desktop/OS_lab/expt3$ cp -r aditya rohit
onworks@onworks:~/Desktop/OS_lab/expt3$ gedit aditya
onworks@onworks:~/Desktop/OS_lab/expt3$ gedit rohit
onworks@onworks:~/Desktop/OS_lab/expt3$ cp --backup aditya aditya_backup
onworks@onworks:~/Desktop/OS_lab/expt3$ gedit aditya_backup
onworks@onworks:~/Desktop/OS_lab/expt3$ gedit rohit
onworks@onworks:~/Desktop/OS_lab/expt3$ cp -i aditya rohit
cp: overwrite 'rohit'? y
onworks@onworks:~/Desktop/OS_lab/expt3$ gedit aditya
onworks@onworks:~/Desktop/OS_lab/expt3$ gedit rohit
onworks@onworks:~/Desktop/OS_lab/expt3$ cp -f rohit aditya
onworks@onworks:~/Desktop/OS_lab/expt3$ gedit aditya
onworks@onworks:~/Desktop/OS_lab/expt3$ gedit rohit
onworks@onworks:~/Desktop/OS_lab/expt3$ cp -p aditya rohit
onworks@onworks:~/Desktop/OS_lab/expt3$ gedit rohit
onworks@onworks:~/Desktop/OS_lab/expt3$ mkdir moved_files
onworks@onworks:~/Desktop/OS_lab/expt3$ cd moved_files
onworks@onworks:~/Desktop/OS_lab/expt3/moved_files$ cat -> aditya_moved
^Z
[3]+ Stopped cat - > aditya_moved
onworks@onworks:~/Desktop/OS_lab/expt3/moved_files$ cd ..
onworks@onworks:~/Desktop/OS_lab/expt3$ mv -i aditya moved_files/aditya_moved
mv: overwrite 'moved_files/aditya_moved'? y
onworks@onworks:~/Desktop/OS_lab/expt3$




- **<u>CONCLUSION:</u> Thus, we have successfully studied and implemented various basic commands of Linux like ls, cp and mv using kernel APIs.**

**NAME:** Meet Raut

**DIV:** S21

**ROLL.NO:** 2201084

# Experiment 4:

- **AIM: To create a child process in Linux using system calls; fork().**

- **THEORY:**

## 1] System call:

When a program is in user mode and requires access to Ram or hardware resources, it must ask the kernel to provide access to that resource. This is done via something called a system call. When a program makes a system call, the mode is switched from user mode to Kernel mode .This is called context switch. The kernel provides the resources which the program requested. System calls are made by user level programmes in following cases :

- Creating, opening, closing and delete files in the system
- Creating and managing new processes
- Creating a connection in the network, sending and receiving packets
- Requesting access to a hardware device like a mouse or a printer

## 2] Fork ( ) :

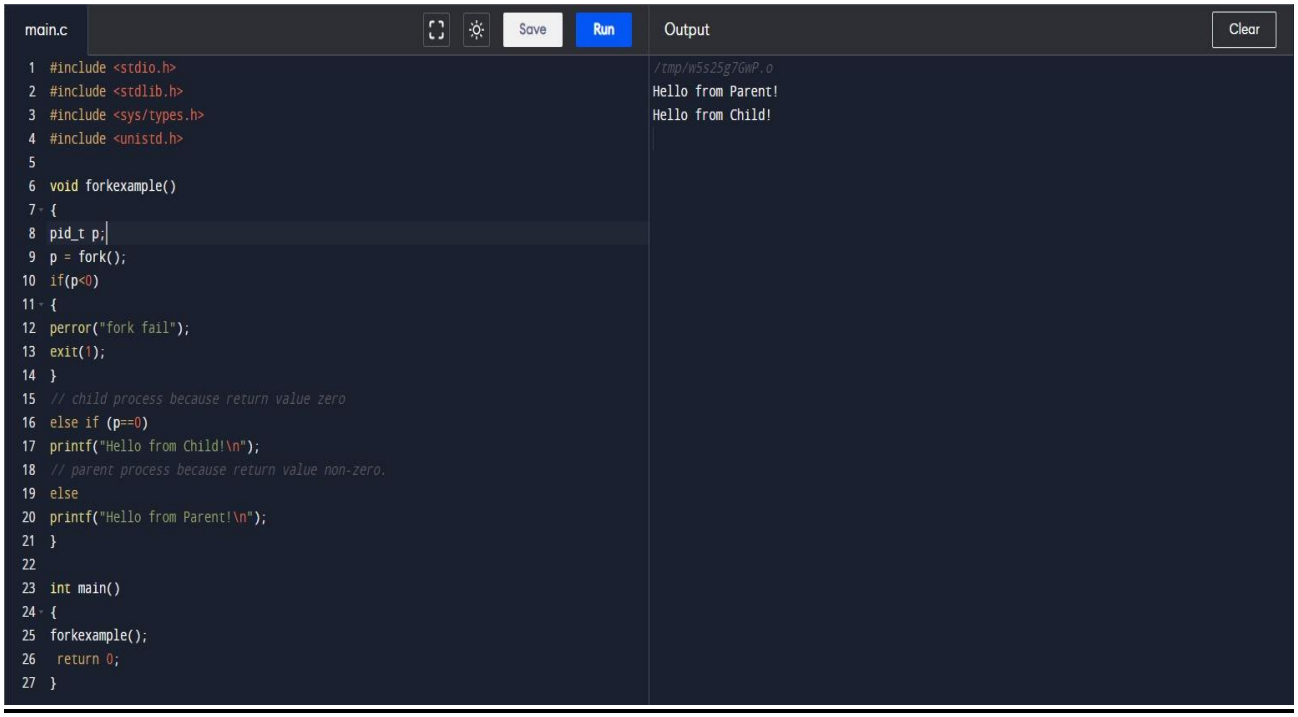
The fork system call is used to create processes. When a process makes a fork().call, an exact copy of the process is created.

There are now two processes, one being the parent process and the other being the child process. The process which is called fork()call is the parent process and the process which is created is called child process. The child process will be exactly the same as the parent. The process state of the parent i.e the address space, variables, open files  etc is copied into the child process. The change of values in the parent process doesn't affect the child and vice versa.

- **1) C PROGRAM:**

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

void forkexample()
{
pid_t p;
p = fork();
if(p<0)
{
perror("fork fail");
exit(1);
}
// child process because return value zero
else if (p==0)
printf("Hello from Child!\n");
// parent process because return value non-zero.
else
printf("Hello from Parent!\n");
}

int main()
{
forkexample();
 return 0;
}
```

- **OUTPUT:**

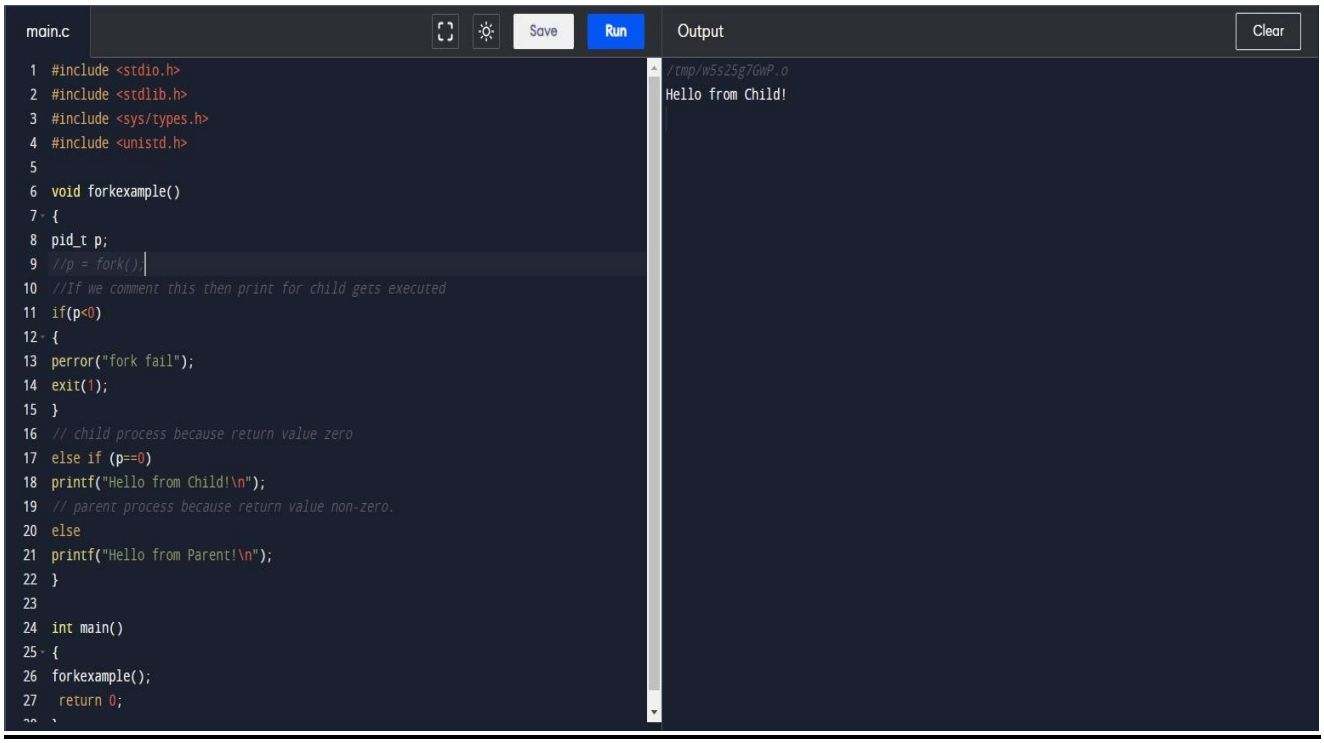



- **2) C PROGRAM:**

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

void forkexample()
{
pid_t p;
//p = fork();
//If we comment this then print for child gets executed
if(p<0)
{
perror("fork fail");
exit(1);
}
// child process because return value zero
```

```
else if (p==0)
printf("Hello from Child!\n");
// parent process because return value non-zero.
else
printf("Hello from Parent!\n");
}

int main()
{
forkexample();
 return 0;
}
```

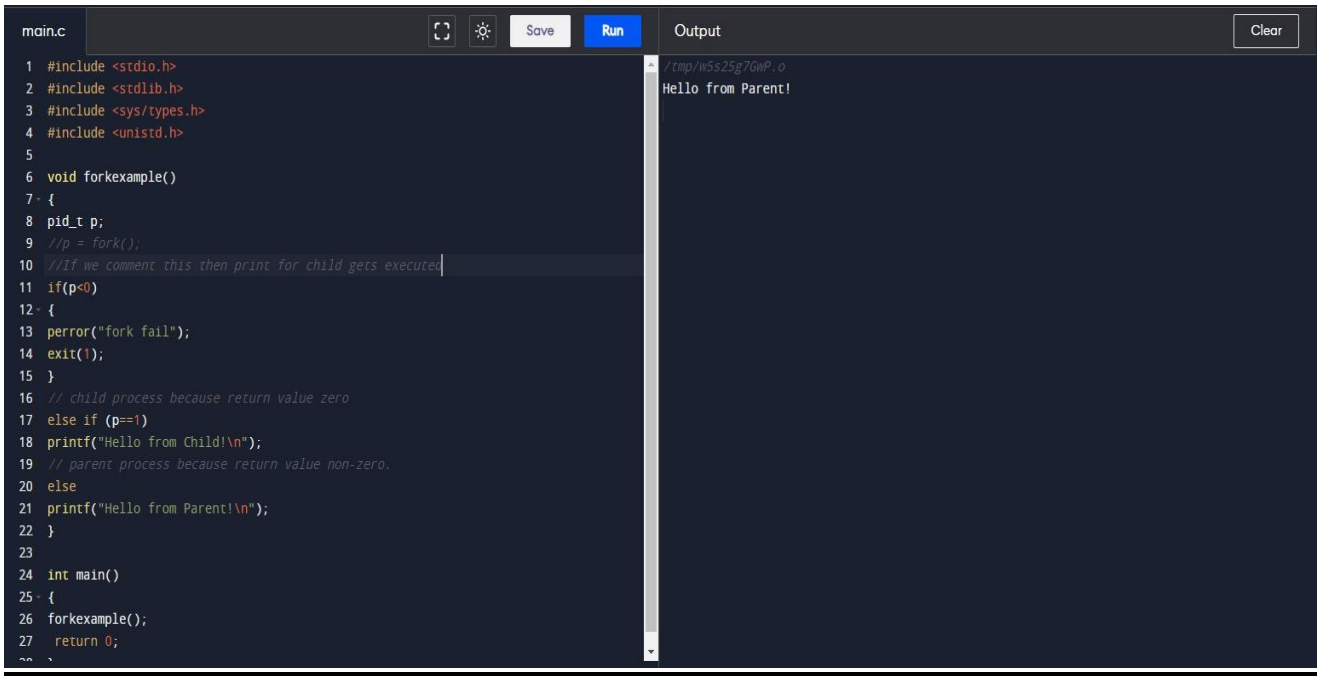
- **OUTPUT:**



- **3) C PROGRAM:**

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

void forkexample()
{
pid_t p;
//p = fork();
//If we comment this then print for child gets executed
if(p<0)
{
perror("fork fail");
exit(1);
}
// child process because return value zero
else if (p==1)
printf("Hello from Child!\n");
// parent process because return value non-zero.
else
printf("Hello from Parent!\n");
}

int main()
{
forkexample();
 return 0;
}
```

- **OUTPUT:**

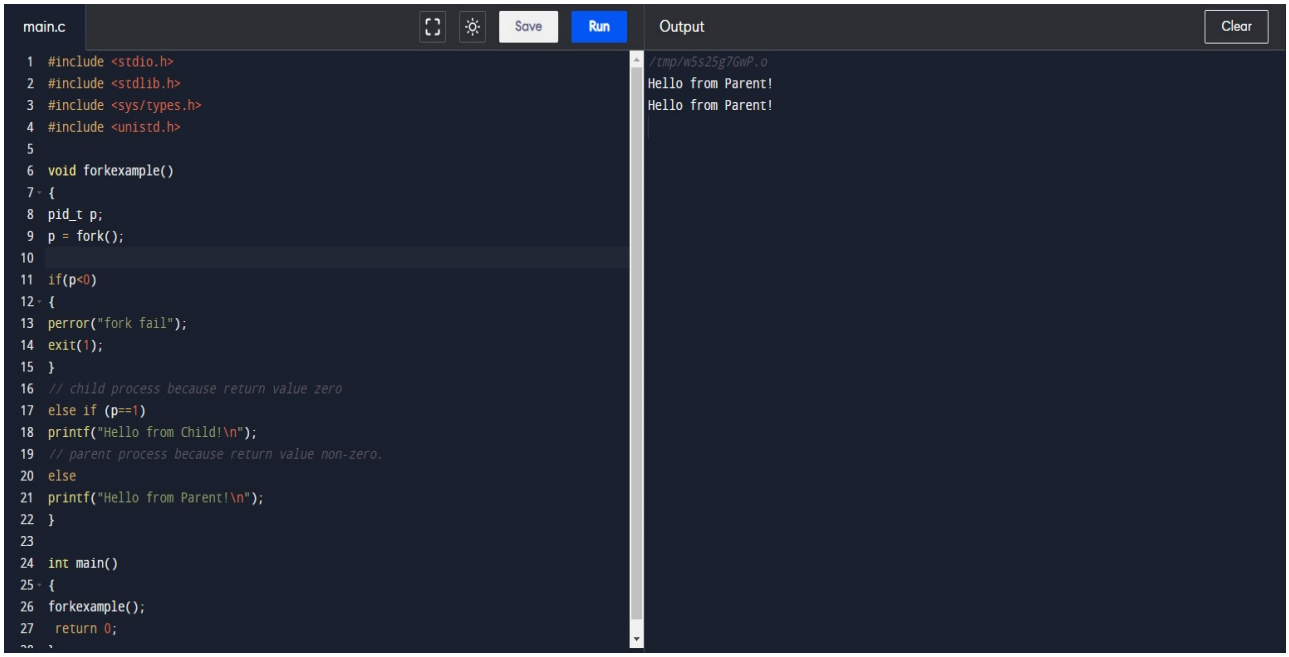



- **4) C PROGRAM:**

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

void forkexample()
{
pid_t p;
p = fork();

if(p<0)
{
perror("fork fail");
exit(1);
}
// child process because return value zero
else if (p==1)
printf("Hello from Child!\n");
```

```
// parent process because return value non-zero.
else
printf("Hello from Parent!\n");
}

int main()
{
forkexample();
 return 0;
}
```

- **OUTPUT:**




**<u>CONCLUSION:</u> Hence, we have successfully implemented and studied how to create child process in Linux using fork() system calls.**

**NAME:** Meet Raut

**DIV:** S21

**ROLL.NO:** 2201084

## Experiment 5:

- **AIM:** a) To study and implement non preemptive scheduling algorithm FCFS.

  b) To study and implement preemptive scheduling algorithm SRTF

- **THEORY:**

**Non preemptive algorithm : FCFS, SJF**

**1] FCFS : First Come First Serve**

It is a non-preemptive scheduling algorithm and the criteria for this is the arrival time of the process CPU is allotted to the process that requires it first. Jobs arriving later are placed at the end of the queue.

FCFS (Example)

| Process | Duration | Oder | Arrival Time |
|---|---|---|---|
| P1 | 24 | 1 | 0 |
| P2 | 3 | 2 | 0 |
| P3 | 4 | 3 | 0 |

**Gantt Chart :**

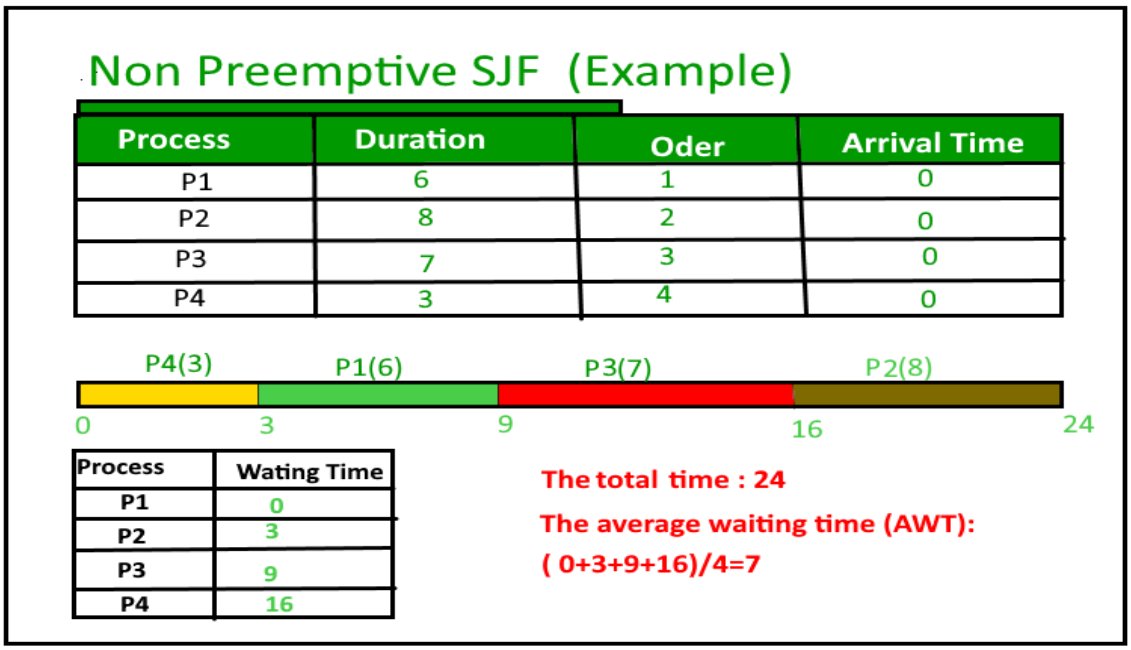

P1 waiting time : 0
P2 waiting time : 24
P3 waiting time : 27

The Average waiting time :

(0+24+27)/3 = 17

## 2] SJF : shortest job first

This is a non preemptive scheduling algorithm, which associates with each process the length of the processes next CPU first.
Criteria : Burst Time.
This algorithm assigns processes according to BT if BT of two processes is the same we use FCFS scheduling criteria.




## 3] Priority Scheduling

This is a non preemptive scheduling algorithm. Each process here has a priority that is either assigned already or externally done.

| Process | Burst Time | Priority |
|---|---|---|
| P1 | 10 | 2 |
| P2 | 5 | 0 |
| P3 | 8 | 1 |

| P1 | P3 | P2 |
|---|---|---|

0 10 18 23

## Preemptive Scheduling Algorithm : SRTF/STRN

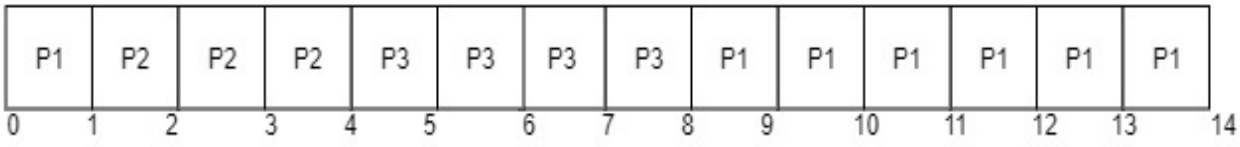
## Shortest Remaining Time First / Shortest Remaining Time Next scheduling.

It is a preemptive SJF Algorithm. The choice arrives when a new process arrives as the ready queue. While a previous process is still executing. The next CPU burst if the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF will preempt the current executing process and not allow the currently running process to finish its CPU burst.

Round Robin is also one preemptive algorithm.

| Process | Burst Time | Arrival Time |
|---|---|---|
| P1 | 7 | 0 |
| P2 | 3 | 1 |
| P3 | 4 | 3 |

**The Gantt Chart for SRTF will be:**



- **C PROGRAM FOR FCFS:**

```
//FCFS SCHEDULING
#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[])
{
   wt[0]=0;

   for (int i = 1; i<n; i++)
      wt[i]=bt[i-1] + wt[i-1];
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[])
{
   for (int i=0; i<n; i++)
   tat[i]=bt[i] + wt[i];
}

void findavgTime(int processes[], int n, int bt[])
{
   int wt[n], tat[n], total_wt=0, total_tat=0;
   findWaitingTime(processes, n, bt, wt);
   findTurnAroundTime(processes, n, bt, wt, tat);

   printf("Process BT      WT     TAT\n");

   for(int i=0; i<n; i++)
   {
      total_wt=total_wt+wt[i];
```

```
        total_tat=total_tat+tat[i];
        printf("\n %d ", (i+1));
        printf("     %d ", bt[i]);
        printf("     %d ", wt[i]);
        printf("     %d\n ", tat[i]);
    }
    float s=(float)total_wt / (float)n;
    float t=(float)total_tat / (float)n;
    printf("\n");
    printf("Average Waiting Time = %f",s);
    printf("\n");
    printf("Average Turn around Time = %f",t);

}

int main()
{
    printf("------------FCFS--------------\n\n\n");
    int processes[] = { 1, 2, 3};
    int n = sizeof processes / sizeof processes[0];

    int burst_time[]= {10, 5, 8};

    findavgTime(processes, n, burst_time);

    return 0;
}
```

- **OUTPUT:**

```
------------FCFS--------------


Process BT      WT      TAT

 1      10       0      10

 2      5       10      15

 3      8       15      23

Average Waiting Time = 8.333333
Average Turn around Time = 16.000000
```

- **C PROGRAM FOR NON-PREEMPTIVE SJF:**

```
#include<stdio.h>

int main()
{
   int A[100][4];
   int i, j, n, total = 0, index, temp;
   float avg_wt, avg_tat;

   printf("-----------SJF-----------\n\n");
   printf("Enter number of Processes: ");
   scanf("%d", &n);
   printf("Enter Burst Time:\n ");
   for(i=0; i<n; i++)
   {
      printf("P%d: ", i+1);
      scanf("%d", &A[i][1]);
      A[i][0] = i+1;
   }

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

```
        index=i;
        for(j=i+1; j<n; j++)
        if (A[j][1] < A[index][1])
        index=j;
        temp=A[i][1];
        A[i][1]=A[index][1];
        A[index][1]=temp;
    }

    A[0][2]=0;

    for(i=1; i<n; i++) {
        A[i][2]=0;
        for (j=0; j<i; j++)
        A[i][2]+=A[j][1];
        total+=A[i][2];
    }

    avg_wt=(float)total / n;
    total=0;
    printf("\n");
    printf("P      BT  WT  TAT \n");
    for(i=0; i<n; i++) {
        A[i][3]=A[i][1] + A[i][2];
        total+= A[i][3];
        printf("P%d      %d    %d   %d\n", A[i][0], A[i][1], A[i][2], A[i][3]);
    }
    avg_tat=(float)total / n;
    printf("Average Waiting Time = %f", avg_wt);
    printf("\nAverage Turn around Time = %f", avg_tat);
}
```

- **OUTPUT:**

```
-----------SJF-----------

Enter number of Processes: 5
Enter Burst Time:
 P1: 10
P2: 5
P3: 6
P4: 3
P5: 12

P      BT  WT  TAT
P1      3    0   3
P2      5    3   8
P3      6    8   14
P4      10    14   24
P5      12    24   36
Average Waiting Time = 9.800000
Average Turn around Time = 17.000000
```

**CONCLUSION: Hence, we have successfully implemented pre-emptive and non preemptive scheduling algorithms.**

Name: Meet Raut
Batch: S21
Roll number: 2201084

# Assignment 6

**AIM**- Write a C program to implement solution of Producer consumer problem through Semaphore

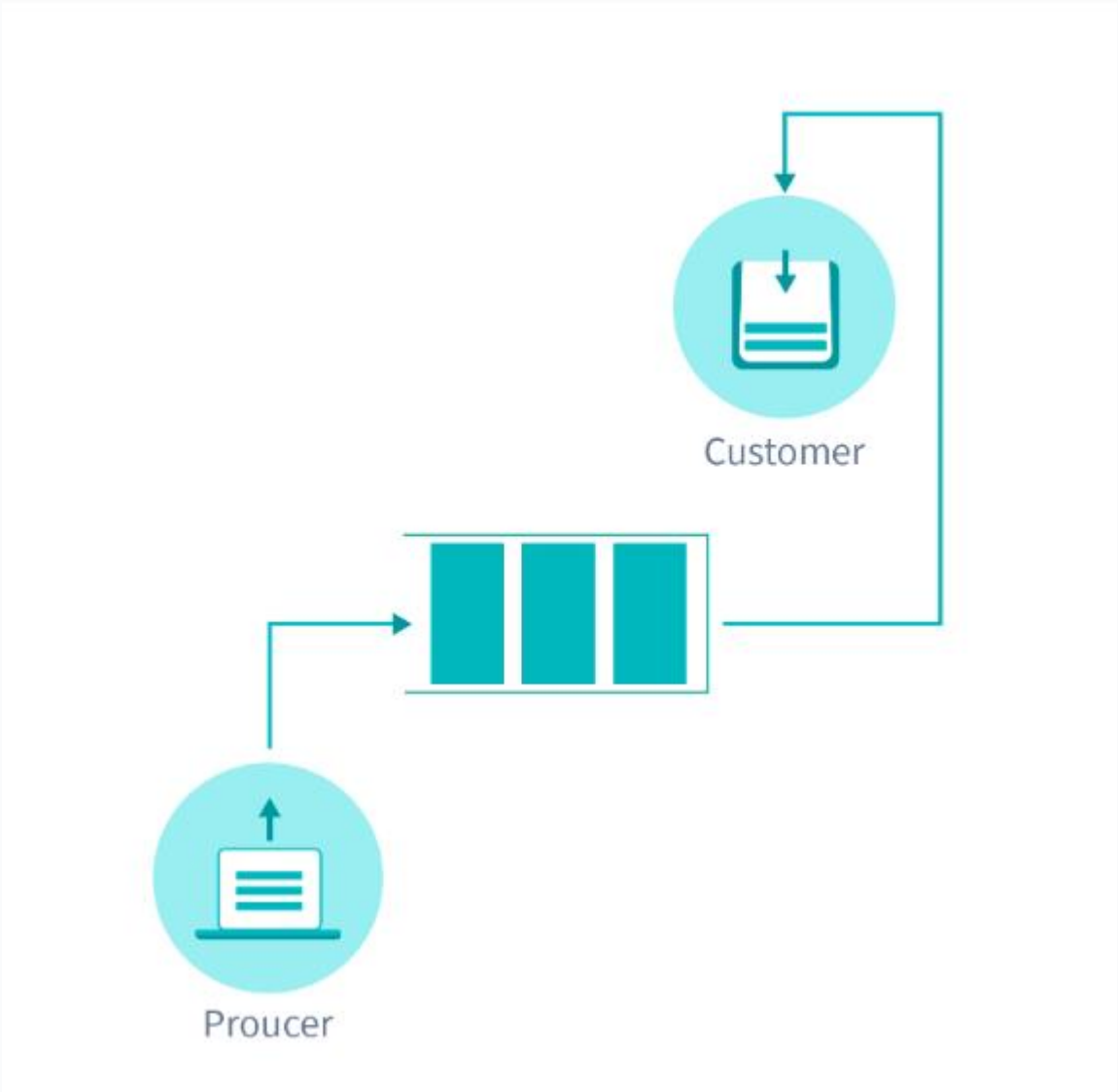
**THEORY**:-

Producer-Consumer problem is a classical synchronisation problem in the operating system. With the presence of more than one process and limited resources in the system the synchronisation problem arises. If one resource is shared between more than one process at the same time then it can lead to data inconsistency. In the producer-consumer problem, the producer produces an item and the consumer consumes the item produced by the producer.

## What is the Producer Consumer Problem?

Before knowing what is Producer-Consumer Problem we have to know what are Producer and Consumer.

- In operating System Producer is a process which is able to produce data/item.
- Consumer is a Process that is able to consume the data/item produced by the Producer.
- Both Producer and Consumer share a common memory buffer. This buffer is a space of a certain size in the memory of the system which is used for storage. The producer produces the data into the buffer and the consumer consumes the data from the buffer.




So, what are the Producer-Consumer Problems?

1. Producer Process should not produce any data when the shared buffer is full.
2. Consumer Process should not consume any data when the shared buffer is empty.
3. The access to the shared buffer should be mutually exclusive i.e at a time only one process should be able to access the shared buffer and make changes to it.

For consistent data synchronisation between Producer and Consumer, the above problem should be resolved.

### Solution For Producer Consumer Problem

To solve the Producer-Consumer problem three semaphores variable are used :

Semaphores are variables used to indicate the number of resources available in the system at a particular time. semaphore variables are used to achieve `Process Synchronisation.

## Full

The full variable is used to track the space filled in the buffer by the Producer process. It is initialised to 0 initially as initially no space is filled by the Producer process.

## Empty

The Empty variable is used to track the empty space in the buffer. The Empty variable is initially initialised to the BUFFER-SIZE as initially, the whole buffer is empty.

## Mutex

Mutex is used to achieve mutual exclusion. mutex ensures that at any particular time only the producer or the consumer is accessing the buffer.

Mutex - mutex is a binary semaphore variable that has a value of 0 or 1.

We will use the Signal() and wait() operation in the above-mentioned semaphores to arrive at a solution to the Producer-Consumer problem.

Signal() - The signal function increases the semaphore value by 1. Wait() - The wait operation decreases the semaphore value by 1.

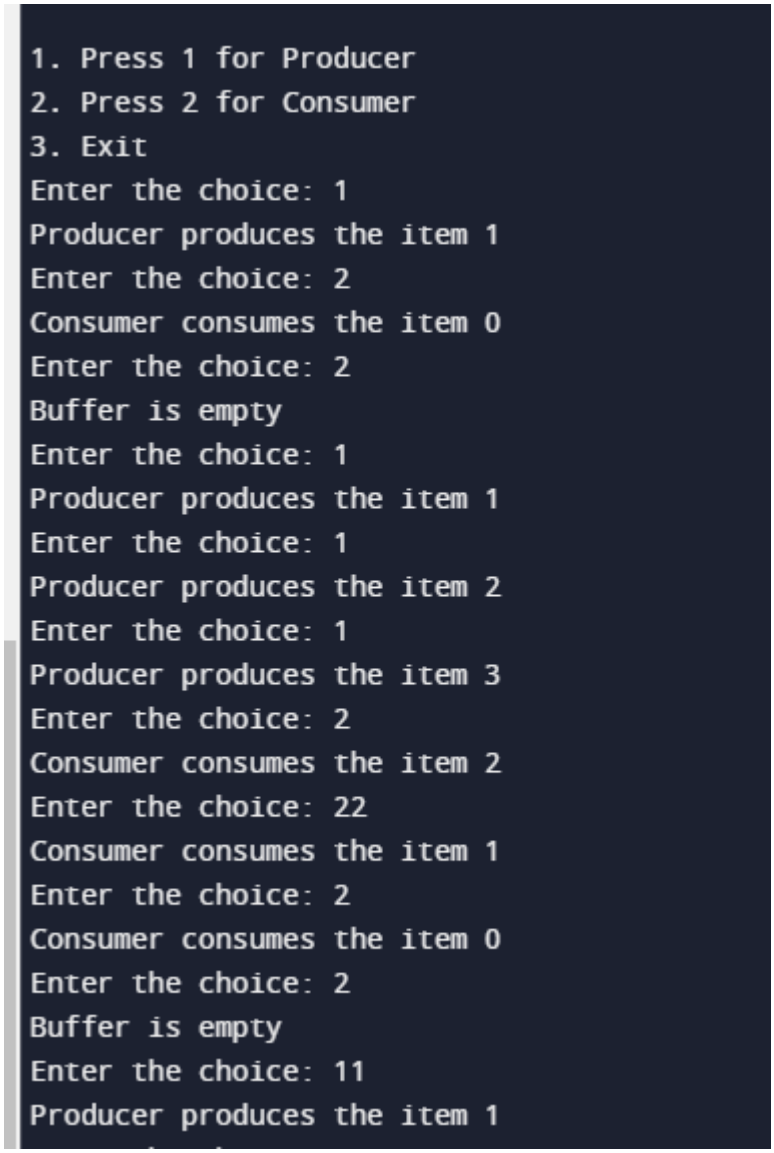
## CODE:

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

int full = 0, empty = 10, x = 0, mutex = 1;

void Producer()
{
   --mutex;
   ++full;
   --empty;
```

```
    x++;
    printf("Producer produces the item %d", x);
    ++mutex;
}

void Consumer()
{
    --mutex;
    --full;
    ++empty;
    x--;
    printf("Consumer consumes the item %d", x);
    ++mutex;
}

int main()
{

    int n, i;

    printf("\n1. Press 1 for Producer \n2. Press 2 for Consumer \n3. Exit");
    #pragma omp critical

    for (i = 1; i > 0; i++)
    {
        printf("\nEnter the choice: ");
        scanf("%d", &n);
        switch (n)
        {
            case 1:
                if ((mutex == 1) && (empty != 0))
                    Producer(full, empty, x, mutex);
                else
                    printf("Buffer is full");
                break;

            case 2:
```

```
            if ((mutex == 1) && (full != 0))
               Consumer(full, empty, x, mutex);
            else
               printf("Buffer is empty");
            break;
         case 3:
            printf("Exiting.....");
            exit(0);
            break;
         default:
            printf("Invalid choice!!!!");
            break;
      }

   }

}
```

**OUTPUT:**




## CONCLUSION:

In conclusion, employing semaphores in C to tackle the Producer-Consumer problem offers an elegant solution for managing shared resources and coordinating concurrent execution between producers and consumers. By using semaphores to enforce synchronisation and mutual exclusion, this approach ensures that producers and consumers operate safely and efficiently, preventing issues such as race conditions or data corruption. Through careful design and implementation, the Producer-Consumer problem can be effectively resolved in C, facilitating the development of robust and scalable multi-threaded applications.

**NAME:** Meet Raut

**DIV:** S2-1

**ROLL.NO:** 2201084

# Experiment 7:

- **AIM: To study and implement the concept of deadlock avoidance through Banker's Algorithm.**

- **THEORY:**

## a) BANKER'S ALGORITHM:

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for the 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?

The banker's algorithm is named so because it is used in the banking system to check whether a 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 the 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 come to withdraw their money then the bank can easily do it.
It also helps the OS to successfully share the resources between all the processes. It is called the banker's algorithm because bankers need a similar algorithm- they admit loans that collectively exceed the bank's funds and then release each borrower's loan in installments. The banker's algorithm uses the notation of a safe allocation state to ensure that granting a resource request cannot lead to a deadlock either immediately or in the future. 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 a safe state always.

The 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 resource 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 **Rj**

## 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 **Pi** may request at most **'k'** instances of resource type **Rj.**

## 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 **Pi** is currently allocated **'k'** instances of resource type **Rj**

## 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 **Pi** currently needs **'k'** instances of resource type **Rj**
- Need [ i, j ] = Max [ i, j ] – Allocation [ i, j ]

Allocation specifies the resources currently allocated to process Pi and Needi specifies the additional resources that process Pi may still request to complete its task.
Banker's algorithm consists of a Safety algorithm and a Resource request algorithm.

## Banker's Algorithm:-

1. **Active**:= Running U Blocked;
   for k=1…r
   New_ request[k]:= Requested_ resources[requesting_ process, k];
2. **Simulated_ allocation**:= Allocated_ resources;
   for k=1…..r //Compute projected allocation state
   Simulated_ allocation [requesting _process, k]:= Simulated_ allocation [requesting _process, k] + New_ request[k];
3. **feasible**:= true;
   for k=1….r // Check whether projected allocation state is feasible

if Total_ resources[k]< Simulated_ total_ alloc [k] then feasible:= false;

4. **if feasible**= true

then // Check whether projected allocation state is a safe allocation state

while set Active contains a process P1 such that

For all k, Total _resources[k] – Simulated_ total_ alloc[k]>= Max_ need [l ,k]-Simulated_ allocation[l, k]

Delete Pl from Active;

for k=1…..r

Simulated_ total_ alloc[k]:= Simulated_ total_ alloc[k]- Simulated_ allocation[l, k];

5. I**f set Active is empty**

then // Projected allocation state is a safe allocation state

for k=1….r // Delete the request from pending requests

Requested_ resources[requesting_ process, k]:=0;

for k=1….r // Grant the request

Allocated_ resources[requesting_ process, k]:= Allocated_ resources[requesting_ process, k] + New_ request[k];

Total_ alloc[k]:= Total_ alloc[k] + New_ request[k];

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

**Example:**

Considering a system with five processes P0 through P4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t0 following snapshot of the system has been

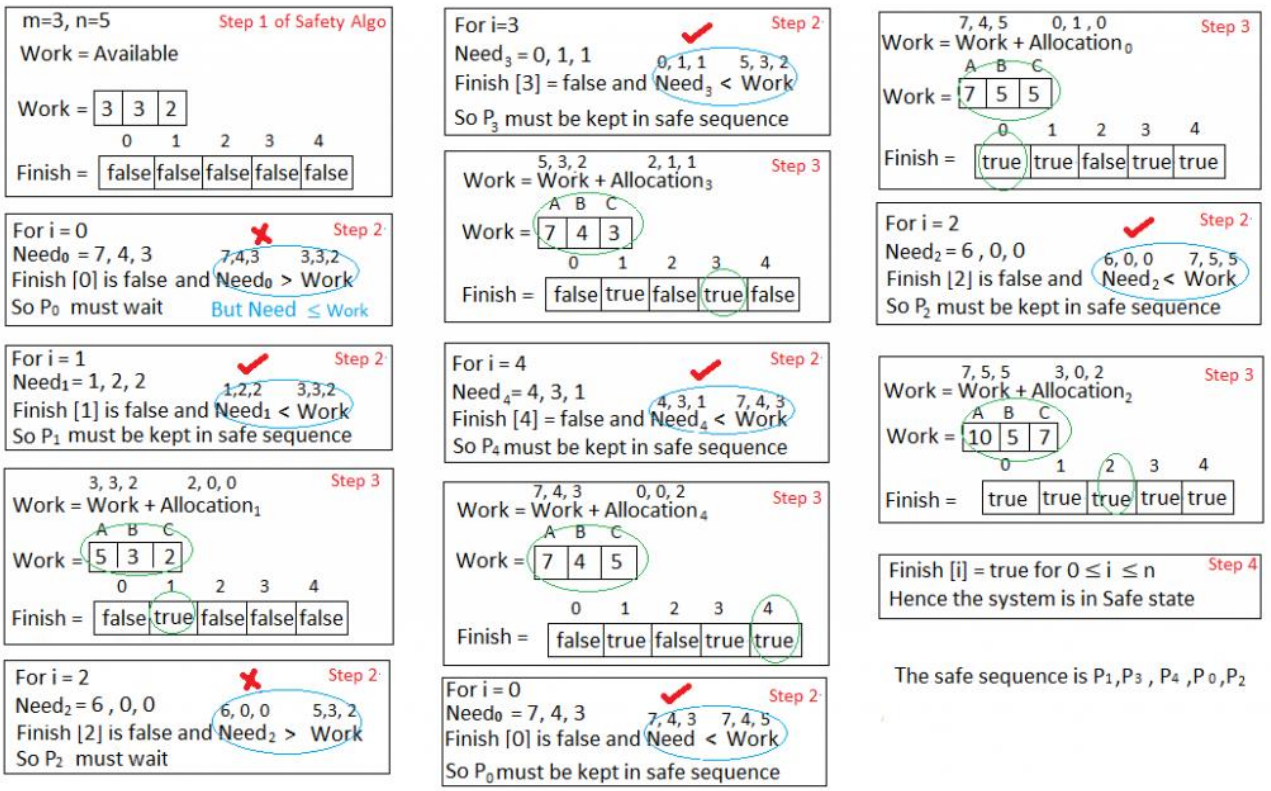
taken:

| Process | Allocation | Max | Available |
|---|---|---|---|
| | A B C | A B C | A B C |
| $P_0$ | 0 1 0 | 7 5 3 | 3 3 2 |
| $P_1$ | 2 0 0 | 3 2 2 | |
| $P_2$ | 3 0 2 | 9 0 2 | |
| $P_3$ | 2 1 1 | 2 2 2 | |
| $P_4$ | 0 0 2 | 4 3 3 | |

Q.1: What will be the content of the Need matrix?
Need [i, j] = Max [i, j] – Allocation [i, j]
So, the content of Need Matrix is:

| Process | Need | | |
|---|---|---|---|
| | A | B | C |
| $P_0$ | 7 | 4 | 3 |
| $P_1$ | 1 | 2 | 2 |
| $P_2$ | 6 | 0 | 0 |
| $P_3$ | 0 | 1 | 1 |
| $P_4$ | 4 | 3 | 1 |

Q.2: Is the system in a safe state? If Yes, then what is the safe sequence?
Applying the Safety algorithm on the given system,

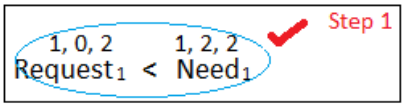


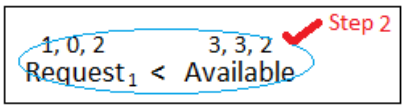

Q.3: What will happen if process P1 requests one additional instance of resource type A and two instances of resource type C?

A B C

$Request_1$ = 1, 0, 2

To decide whether the request is granted we use Resource Request algorithm




Step 3

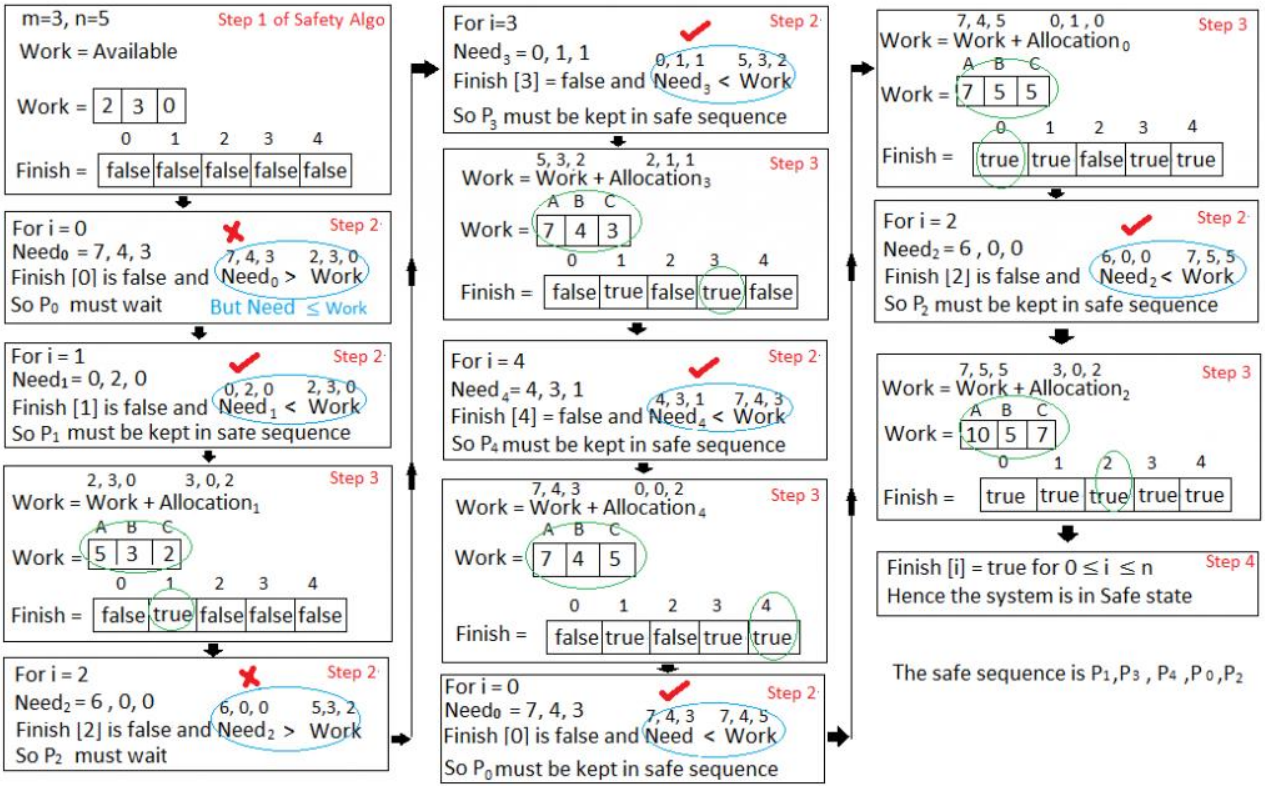
$Available = Available - Request_1$

$Allocation_1 = Allocation_1 + Request_1$

$Need_1 = Need_1 - Request_1$

| Process | Allocation | Need | Available |
|---|---|---|---|
| | A B C | A B C | A B C |
| $P_0$ | 0 1 0 | 7 4 3 | 2 3 0 |
| $P_1$ | 3 0 2 | 0 2 0 | |
| $P_2$ | 3 0 2 | 6 0 0 | |
| $P_3$ | 2 1 1 | 0 1 1 | |
| $P_4$ | 0 0 2 | 4 3 1 | |

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.




Hence the new system state is safe, so we can immediately grant the request for process P1 .

- As the processes enter the system, they must predict the maximum number of resources needed which is impractical to determine.
- In this algorithm, the number of processes remain fixed which is not possible in interactive systems.
- This algorithm requires that there should be a fixed number of resources to allocate. If a device breaks and becomes suddenly unavailable the algorithm would not work.
- Overhead cost incurred by the algorithm can be high when there are many processes and resources because it has to be invoked for every processes.

- **C PROGRAM:**

```
#include<stdio.h>

int main()
{
int p, c, count = 0, i, j, alc[5][3], max[5][3], need[5][3], safe[5], available[3],
finish[5], terminate = 0;

printf("Enter the number of process and resources");
scanf("%d %d", &p, &c);

printf("enter allocation of resource of all process %dx%d matrix",p,c);
for (i = 0; i < p; i++)
{
for (j = 0; j < c; j++)
{
scanf("%d", &alc[i][j]);
}
}

printf("enter the max resource process required %dx%d matrix", p, c);
for (i = 0; i < p; i++)
{
for (j = 0; j < c; j++)
{
scanf("%d", &max[i][j]);
}
}
```

```
printf("enter the available resource");
for (i = 0; i < c; i++)
scanf("%d", &available[i]);

printf("\n need resources matrix are\n");
for (i = 0; i < p; i++)
{
for (j = 0; j < c; j++)
{
need[i][j] = max[i][j] - alc[i][j];
printf("%d\t", need[i][j]);
}
printf("\n");
}

for (i = 0; i < p; i++)
{
finish[i] = 0;
}

while (count < p)
{
for (i = 0; i < p; i++)
{
if (finish[i] == 0)
{
for (j = 0; j < c; j++)
{
```

```
if (need[i][j] > available[j])
break;
}

if (j == c) {
safe[count] = i;
finish[i] = 1;
for (j = 0; j < c; j++) {
available[j] += alc[i][j];
}
count++;
terminate = 0;
}
else {
terminate++;
}
}
}

if (terminate == (p - 1))
{
printf("safe sequence does not exist");
break;
}
}

if (terminate != (p - 1))
{
printf("\n available resource after completion\n");
```

```
for (i = 0; i < c; i++)
{
printf("%d\t", available[i]);
}
printf("\n safe sequence are\n");
for (i = 0; i < p; i++)
{
printf("p%d\t", safe[i]);
}
}
return 0;
}
```

- **OUTPUT:**

```
Enter the number of process and resources5 3
enter allocation of resource of all process 5x3 matrix0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
enter the max resource process required 5x3 matrix7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
enter the available resource3 3 2

 need resources matrix are
7   4   3
1   2   2
6   0   0
0   1   1
```

```
 available resource after completion
10   5    7
 safe sequence are
p1   p3   p4   p0   p2
```

**<u>CONCLUSION:</u> Hence, we have successfully implemented the concept of deadlock avoidance through Banker's Algorithm.**

**NAME:** Meet Raut

**DIV:** S2-1

**ROLL.NO:** 2201084

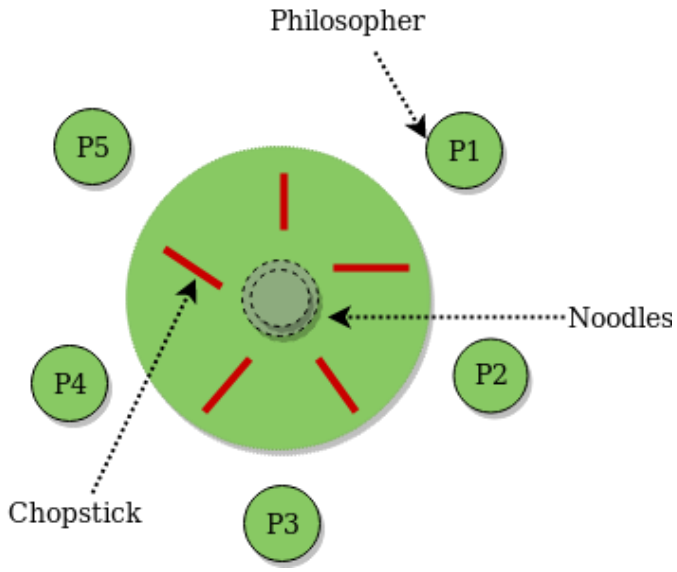
## Experiment 8:

- **AIM:** To study and implement the concept of Dining Philospher‟s Problem

- **THEORY:**

### ➢ DINING PHILOSPHER'S PROBLEM:

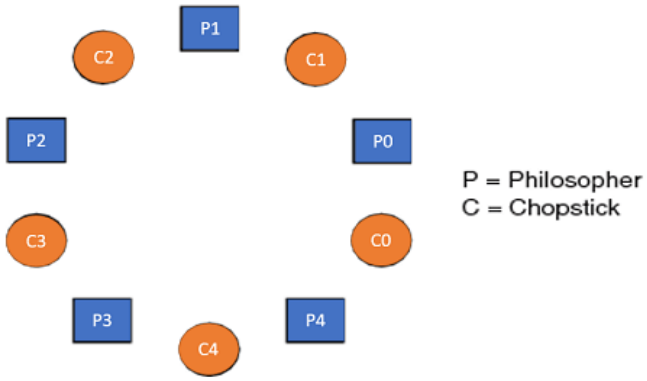
The Dining Philosopher Problem states that K philosophers are seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pick up the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.




The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers. To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available. In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.

The dining philosopher demonstrates a large class of concurrency control problems hence it's a classic synchronization problem.

**Dining Philosophers Problem**- Let's understand the Dining Philosophers Problem with the below code, we have used fig 1 as a reference to make you understand the problem exactly. The five Philosophers are represented as P0, P1, P2, P3, and P4 and five chopsticks by C0, C1, C2, C3, and C4.




```
Void Philosopher
 {
 while(1)
  {
   take_chopstick[i];
   take_chopstick[ (i+1) % 5] ;
   . .
   . EATING THE NOODLE
   .
   put_chopstick[i] );
   put_chopstick[ (i+1) % 5] ;
   .
   . THINKING
  }
}
```

Let's discuss the above code:

Suppose Philosopher P0 wants to eat, it will enter in Philosopher() function, and execute **take_chopstick[i];** by doing this it holds **C0 chopstick** after that it execute **take_chopstick[ (i+1) % 5];** by doing this it holds **C1 chopstick**( since i =0, therefore (0 + 1) % 5 = 1)

Similarly suppose now Philosopher P1 wants to eat, it will enter in Philosopher() function, and execute **take_chopstick[i];** by doing this it holds **C1 chopstick** after that it execute **take_chopstick[ (i+1) % 5];** by doing this it holds **C2 chopstick**( since i =1, therefore (1 + 1) % 5 = 2)

But Practically Chopstick C1 is not available as it has already been taken by philosopher P0, hence the above code generates problems and produces race condition.

## The solution of the Dining Philosophers Problem

We use a semaphore to represent a chopstick and this truly acts as a solution of the Dining Philosophers Problem. Wait and Signal operations will be used for the solution of the Dining Philosophers Problem, for picking a chopstick wait operation can be executed while for releasing a chopstick signal semaphore can be executed.

Semaphore: A semaphore is an integer variable in S, that apart from initialization is accessed by only two standard atomic operations - wait and signal, whose definitions are as follows:

```
1. wait( S )
{
while( S <= 0) ;
S--;
}

2. signal( S )
{
S++;
}
```

From the above definitions of wait, it is clear that if the value of S <= 0 then it will enter into an infinite loop(because of the semicolon; after while loop). Whereas the job of the signal is to increment the value of S.

The structure of the chopstick is an array of a semaphore which is represented as shown below –

```
semaphore C[5];
```

Initially, each element of the semaphore C0, C1, C2, C3, and C4 are initialized to 1 as the chopsticks are on the table and not picked up by any of the philosophers.

Let's modify the above code of the Dining Philosopher Problem by using semaphore operations wait and signal, the desired code looks like

```
void Philosopher
 {
 while(1)
  {
   Wait( take_chopstickC[i] );
   Wait( take_chopstickC[(i+1) % 5] ) ;
   . .
   . EATING THE NOODLE
   .
   Signal( put_chopstickC[i] );
   Signal( put_chopstickC[ (i+1) % 5] ) ;
   .
   . THINKING
  }
}
```

In the above code, first wait operation is performed on take_chopstickC[i] and take_chopstickC [ (i+1) % 5]. This shows philosopher i have picked up the chopsticks from its left and right. The eating function is performed after that.

On completion of eating by philosopher i the, signal operation is performed on take_chopstickC[i] and take_chopstickC [ (i+1) % 5]. This shows that the philosopher i have eaten and put down both the left and right chopsticks. Finally, the philosopher starts thinking again.

## Let's understand how the above code is giving a solution to the dining philosopher problem?

Let value of i = 0( initial value ), Suppose Philosopher P0 wants to eat, it will enter in Philosopher() function, and execute **Wait( take_chopstickC[i] );** by doing this it holds **C0**

**chopstick** and reduces semaphore C0 to 0**,** after that it execute **Wait( take_chopstickC[(i+1) % 5] );** by doing this it holds **C1 chopstick**( since i =0, therefore (0 + 1) % 5 = 1) and reduces semaphore C1 to 0

Similarly, suppose now Philosopher P1 wants to eat, it will enter in Philosopher() function, and execute **Wait( take_chopstickC[i] );** by doing this it will try to hold **C1 chopstick** but will not be able to do that**,** since the value of semaphore C1 has already been set to 0 by philosopher P0, therefore it will enter into an infinite loop because of which philosopher P1 will not be able to pick chopstick C1 whereas if Philosopher P2 wants to eat, it will enter in Philosopher() function, and execute **Wait( take_chopstickC[i] );** by doing this it holds **C2 chopstick** and reduces semaphore C2 to 0, after that, it executes **Wait( take_chopstickC[(i+1) % 5] );** by doing this it holds **C3 chopstick**( since i =2, therefore (2 + 1) % 5 = 3) and reduces semaphore C3 to 0.

Hence the above code is providing a solution to the dining philosopher problem, A philosopher can only eat if both immediate left and right chopsticks of the philosopher are available else philosopher needs to wait. Also at one go two independent philosophers can eat simultaneously (i.e., philosopher **P0 and P2, P1 and P3 & P2 and P4** can eat simultaneously as all are the independent processes and they are following the above constraint of dining philosopher problem)

## The drawback of the above solution of the dining philosopher problem

From the above solution of the dining philosopher problem, we have proved that no two neighboring philosophers can eat at the same point in time. The drawback of the above solution is that this solution can lead to a deadlock condition. This situation happens if all the philosophers pick their left chopstick at the same time, which leads to the condition of deadlock and none of the philosophers can eat.

To avoid deadlock, some of the solutions are as follows –

- Maximum number of philosophers on the table should not be more than four, in this case, chopstick C4 will be available for philosopher P3, so P3 will start eating and after the finish of his eating procedure, he will put down his both the chopstick C3 and C4, i.e. semaphore C3 and C4 will now be incremented to 1. Now philosopher P2 which was holding chopstick C2 will also have chopstick C3 available, hence similarly, he will put down his chopstick after eating and enable other philosophers to eat.
- A philosopher at an even position should pick the right chopstick and then the left chopstick while a philosopher at an odd position should pick the left chopstick and then the right chopstick.
- Only in case if both the chopsticks ( left and right ) are available at the same time, only then a philosopher should be allowed to pick their chopsticks
- All the four starting philosophers ( P0, P1, P2, and P3) should pick the left chopstick and then the right chopstick, whereas the last philosopher P4 should pick the right

chopstick and then the left chopstick. This will force P4 to hold his right chopstick first since the right chopstick of P4 is C0, which is already held by philosopher P0 and its value is set to 0, i.e C0 is already 0, because of which P4 will get trapped into an infinite loop and chopstick C4 remains vacant. Hence philosopher P3 has both left C3 and right C4 chopstick available, therefore it will start eating and will put down its both chopsticks once finishes and let others eat which removes the problem of deadlock.

The design of the problem was to illustrate the challenges of avoiding deadlock, a deadlock state of a system is a state in which no progress of system is possible. Consider a proposal where each philosopher is instructed to behave as follows:

- The philosopher is instructed to think till the left fork is available, when it is available, hold it.
- The philosopher is instructed to think till the right fork is available, when it is available, hold it.
- The philosopher is instructed to eat when both forks are available.
- then, put the right fork down first
- then, put the left fork down next
- repeat from the beginning.

We will implement the dining philosopher problem by using binary and counting semaphore. We can implement the semaphore as a room since there is one room which can accommodate the philosophers.

The function sem_init initialises the semaphores as it is initialised to 4 the value can vary from 0 to 4 ie P0 to P4 and so on.

For the 5 chopsticks we create 5 binary semaphores ie from C0 to C4

We use binary semaphores here as for C0 to C4 we have only one instance of it.

Thus in an empty room we have 5 chopsticks and the philosophers.

We create threads next. There can be a situation where all the threads started executing thus causing deadlock. Thus we allow some philosophers to enter room first so that atleast one of them finishes eating.

In the philosopher function we first convert the number passed as void * into int

We call sem_wait to check if resource is available and if available it is allocated to philosopher

We also know that it's a counting semaphore hence the number of semaphores is decremented ie when one of the semaphores is allocated and all resources are allocated all the remaining semaphores ie threads are placed on wait

The chopsticks are binary semaphores thus we can block the chopsticks ie we can block the chopsticks to the left and right. Finally we free the semaphore by using sem_post function so that other threads placed on the queue can use the resources for positive value of the semaphore and it is unlocked.

This happens for all the philosophers then we join them back to the main process using pthread_join

- **C PROGRAM:**

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
#include<unistd.h>
sem_t room;
sem_t chopstick[5];
void * philosopher(void *);
void eat(int);
int main()
{
int i,a[5];
pthread_t tid[5];
sem_init(&room,0,4);
for(i=0;i<5;i++)
sem_init(&chopstick[i],0,1);
for(i=0;i<5;i++){
a[i]=i;
pthread_create(&tid[i],NULL,philosopher,(void *)&a[i]);
}
for(i=0;i<5;i++)
pthread_join(tid[i],NULL);
```

```
}
void * philosopher(void * num)
{
int phil=*(int *)num;
sem_wait(&room);
printf("\nPhilosopher %d has entered room",phil);
sem_wait(&chopstick[phil]);
sem_wait(&chopstick[(phil+1)%5]);
eat(phil);
sleep(2);
printf("\nPhilosopher %d has finished eating",phil);
sem_post(&chopstick[(phil+1)%5]);
sem_post(&chopstick[phil]);
sem_post(&room);
}
void eat(int phil)
{
printf("\nPhilosopher %d is eating",phil);
}
```

- **OUTPUT:**

```
--------------- DINING PHILOSPHER'S PROBLEM ---------------


Philosopher 4 has entered room
Philosopher 4 is eating
Philosopher 3 has entered room
Philosopher 2 has entered room
Philosopher 1 has entered room
Philosopher 4 has finished eating
Philosopher 3 is eating
Philosopher 0 has entered room
Philosopher 3 has finished eating
Philosopher 2 is eating
Philosopher 2 has finished eating
Philosopher 1 is eating
Philosopher 1 has finished eating
Philosopher 0 is eating
Philosopher 0 has finished eating

...Program finished with exit code 0
Press ENTER to exit console.
```

```
--------------- DINING PHILOSPHER'S PROBLEM ---------------


Philosopher 0 has entered room
Philosopher 0 is eating
Philosopher 1 has entered room
Philosopher 2 has entered room
Philosopher 2 is eating
Philosopher 3 has entered room
Philosopher 0 has finished eating
Philosopher 4 has entered room
Philosopher 4 is eating
Philosopher 2 has finished eating
Philosopher 1 is eating
Philosopher 4 has finished eating
Philosopher 3 is eating
Philosopher 1 has finished eating
Philosopher 3 has finished eating

...Program finished with exit code 0
Press ENTER to exit console.
```


**<u>CONCLUSION:</u> Hence, we have successfully implemented the concept of Dining Philospher's Problem.**

## **Experiment 9:** FIFO Page Replacement Algorithm

**AIM:** To study & implement the FIFO Page Replacement Algorithm

**THEORY:**

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when a new page comes in. Page replacement becomes necessary when a page fault occurs and there are no free page frames in memory. However, another page fault would arise if the replaced page is referenced again. Hence it is important to replace a page that is not likely to be referenced in the immediate future. y. If no page frame is free, the virtual memory manager performs a page replacement operation to replace one of the pages existing in memory with the page whose reference caused the page fault. It is performed as follows:

The virtual memory manager uses a page replacement algorithm to select one of the pages currently in memory for replacement, accesses the page table entry of the selected page to mark it as "not present" in memory, and initiates a page-out operation for it if the modified bit of its page table entry indicates that it is a dirty page.

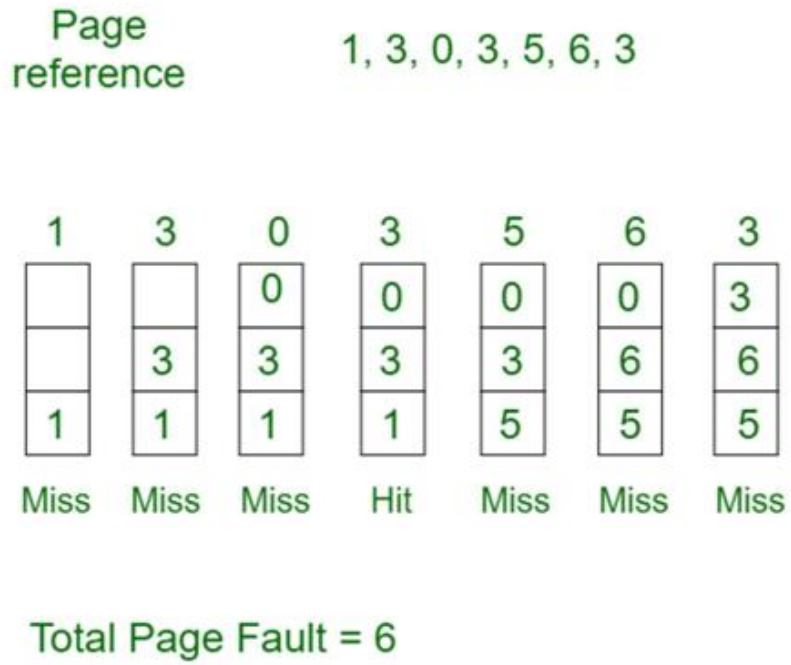
Page Fault: A page fault happens when a running program accesses a memory page that is mapped into the virtual address space but not loaded in physical memory. Since actual physical memory is much smaller than virtual memory, page faults happen. In case of a page fault, Operating System might have to replace one of the existing pages with the newly needed page.

Different page replacement algorithms suggest different ways to decide which page to replace.

The target for all algorithms is to reduce the number of page faults.

Page Replacement Algorithms: 1. First In First Out (FIFO): This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Example 1: Consider page reference string 1, 3, 0, 3, 5, 6, 3 with 3 page frames.Find the number of page faults




Initially, all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> 3 Page Faults. when 3 comes, it is already in memory so —> 0 Page Faults. Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>1 Page Fault. 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>1 Page Fault.

Finally, when 3 come it is not available so it replaces 0 1 page fault. Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example,

if we consider reference strings 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4, and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10-page faults. .

**CODE:**

```
#include<stdio.h>
int main()
{
   int incomingStream[] = {3, 0, 4, 3, 2, 1, 4, 6, 3, 0, 8, 9, 3,8,5};
   int pageFaults = 0;
   int frames = 3; //no. of frames
   int m, n, s, pages;
   pages = sizeof(incomingStream)/sizeof(incomingStream[0]);
   //Reference string / size of incoming
   printf("Incoming \t Frame 1 \t Frame 2 \t Frame 3\n");
   int temp[frames];
   for(m = 0; m < frames; m++)
   {
      temp[m] = -1;
   }
   //For entire reference string,
   for(m = 0; m < pages; m++)
```

```
    {
        s = 0;
        //evaluate for all pages ; set count s=0 (no.of hits) initially
        for(n = 0; n < frames; n++)
        {
            //If current page value of stream is present in temp,increment
hit (s) and decrease faults.
            if(incomingStream[m] == temp[n])
            {
                s++;
                pageFaults--;
            }
        }
        pageFaults++;
        if((pageFaults <= frames) && (s == 0))
        {
            temp[pageFaults - 1] = incomingStream[m];
        }
        //pagefaults are less than no.of frames and hits=0 then keep on
assigning m into temp array
        else if(s == 0)
        {
            temp[(pageFaults - 1) % frames] = incomingStream[m];
        }
        printf("%d\t\t\t", incomingStream[m]);
```

```
        for(n = 0; n < frames; n++)
        {
            if(temp[n] != -1)
                printf(" %d\t\t\t", temp[n]);
            else
                printf(" - \t\t\t");
        }
        printf("\n");
    }
    printf("Total Page Faults:\t%d\n", pageFaults);
    return 0;
}
```

**Output:**

```
/tmp/KjYeKCtaKB.o
Incoming     Frame 1     Frame 2     Frame 3
3            3           -           -
0            3           0           -
4            3           0           4
3            3           0           4
2            2           0           4
1            2           1           4
4            2           1           4
6            2           1           6
3            3           1           6
0            3           0           6
8            3           0           8
9            9           0           8
3            9           3           8
8            9           3           8
5            9           3           5
Total Page Faults:  12


=== Code Execution Successful ===
```

## CONCLUSION:

**Successfully Implemented Page Replacement policies for handling page faults.**

**NAME:** Meet Raut

**DIV:** S21

**ROLL.NO:** 2201084

# Experiment 10:

- **AIM: To study and implement the concept of Disk Scheduling.**

- **THEORY:**

As we know, a process needs two type of time, CPU time and IO time. For I/O, it requests the Operating system to access the disk.

However, the operating system must be fare enough to satisfy each request and at the same time, operating system must maintain the efficiency and speed of process execution.
The technique that operating system uses to determine the request which is to be satisfied next is called disk scheduling.

Let's discuss some important terms related to disk scheduling.

## Seek Time

Seek time is the time taken in locating the disk arm to a specified track where the read/write request will be satisfied.

## Rotational Latency

It is the time taken by the desired sector to rotate itself to the position from where it can access the R/W heads.

## Transfer Time

It is the time taken to transfer the data.

## Disk Access Time

Disk access time is given as,

Disk Access Time = Rotational Latency + Seek Time + Transfer Time

## Disk Response Time

It is the average of time spent by each request waiting for the IO operation.

## Purpose of Disk Scheduling

The main purpose of disk scheduling algorithm is to select a disk request from the queue of IO requests and decide the schedule when this request will be processed.

## Goal of Disk Scheduling Algorithm

- Fairness
- High throughout
- Minimal traveling head time

# Disk Scheduling Algorithms

The list of various disks scheduling algorithm is given below. Each algorithm is carrying some advantages and disadvantages. The limitation of each algorithm leads to the evolution of a new algorithm.

- FCFS scheduling algorithm
- SSTF (shortest seek time first) algorithm
- SCAN scheduling
- C-SCAN scheduling
- LOOK Scheduling
- C-LOOK scheduling

## FCFS Scheduling Algorithm

It is the simplest Disk Scheduling algorithm. It services the IO requests in the order in which they arrive. There is no starvation in this algorithm, every request is serviced.

## Disadvantages

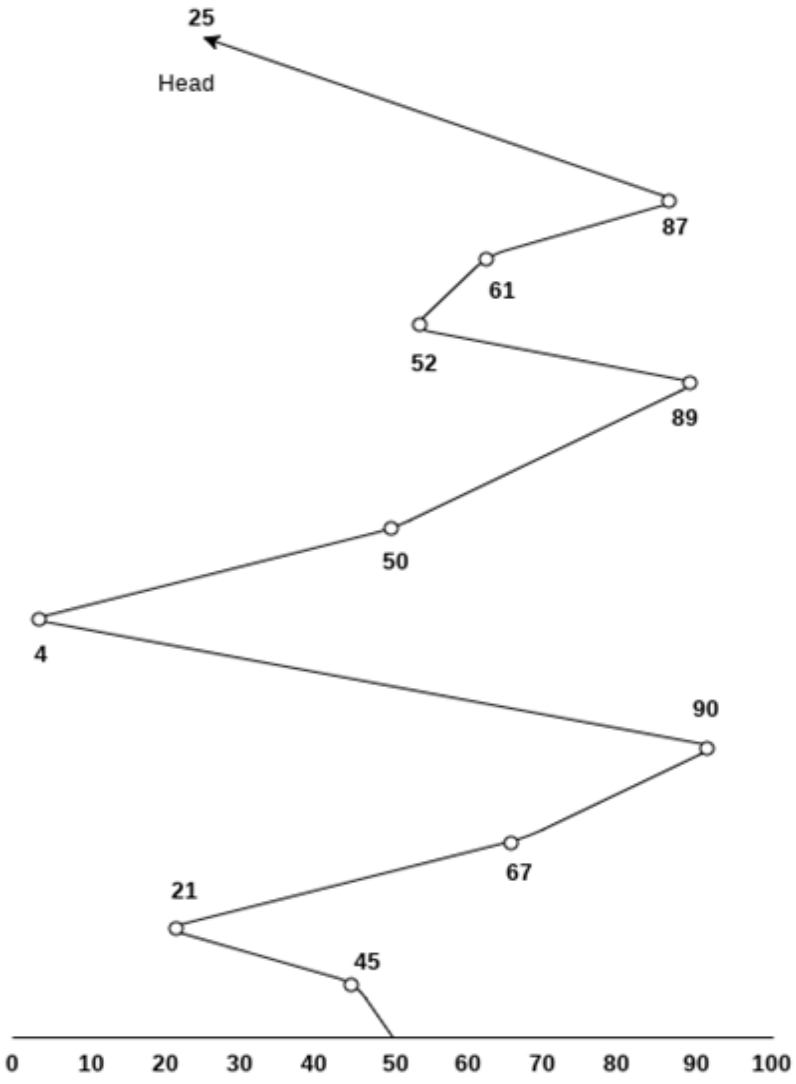
- The scheme does not optimize the seek time.
- The request may come from different processes therefore there is the possibility of inappropriate movement of the head.

## Example

Consider the following disk request sequence for a disk with 100 tracks 45, 21, 67, 90, 4, 50, 89, 52, 61, 87, 25

Head pointer starting at 50 and moving in left direction. Find the number of head movements in cylinders using FCFS scheduling.

## Solution




= (50-45)+(45-21)+(67-21)+(90-67)+(90-4)+(50-4)+(89-50)+(61-52)+(87-61)+(87-25)

= 5 + 24 + 46 + 23 + 86 + 46 + 49 + 9 + 26 + 62

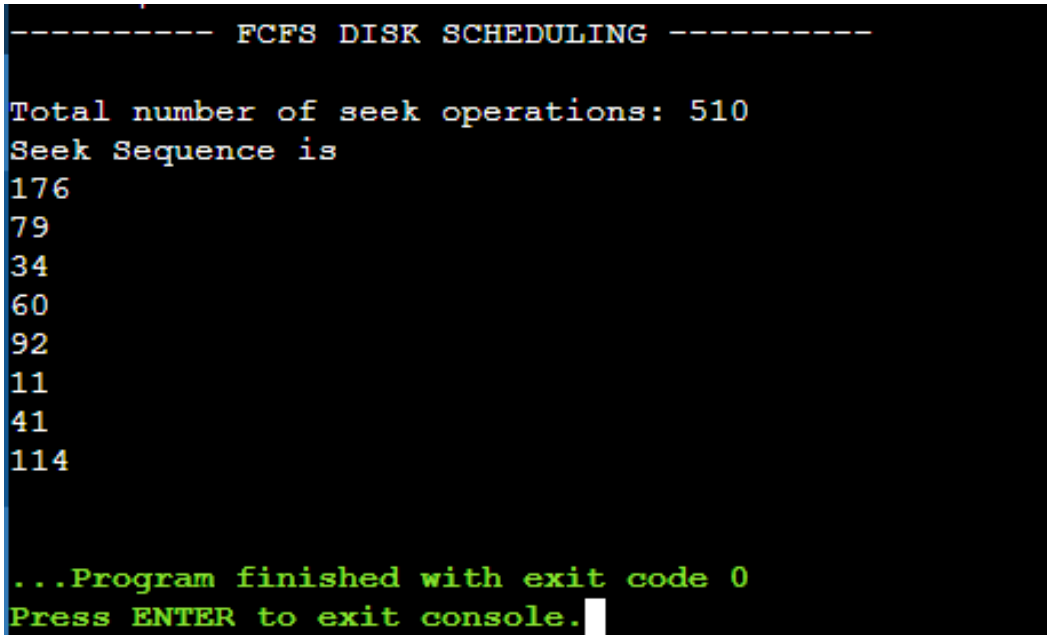
= 376

- **C PROGRAM (FCFS) :**

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

#define SIZE 8

void FCFS(int arr[], int head) {
    int seek_count = 0;
    int cur_track, distance;

    for (int i = 0; i < SIZE; i++) {
        cur_track = arr[i];

        // calculate absolute distance
        distance = abs(head - cur_track);

        // increase the total count
        seek_count += distance;

        // accessed track is now new head
        head = cur_track;
    }

    printf("Total number of seek operations: %d\n", seek_count);

    // Seek sequence would be the same
    // as request array sequence
```

```
    printf("Seek Sequence is\n");

    for (int i = 0; i < SIZE; i++) {
        printf("%d\n", arr[i]);
    }
}

//Driver code
int main() {
    // request array
    int arr[SIZE] = { 176, 79, 34, 60, 92, 11, 41, 114 };
    int head = 50;
    printf("---------- FCFS DISK SCHEDULING ----------\n\n");

    FCFS(arr, head);

    return 0;
}
```

- **OUTPUT:**




- **<u>CONCLUSION:</u> Hence, we have successfully implemented the concept of Disk Scheduling.**

Name : Meet Raut

Batch : S21

Roll no: 2201084

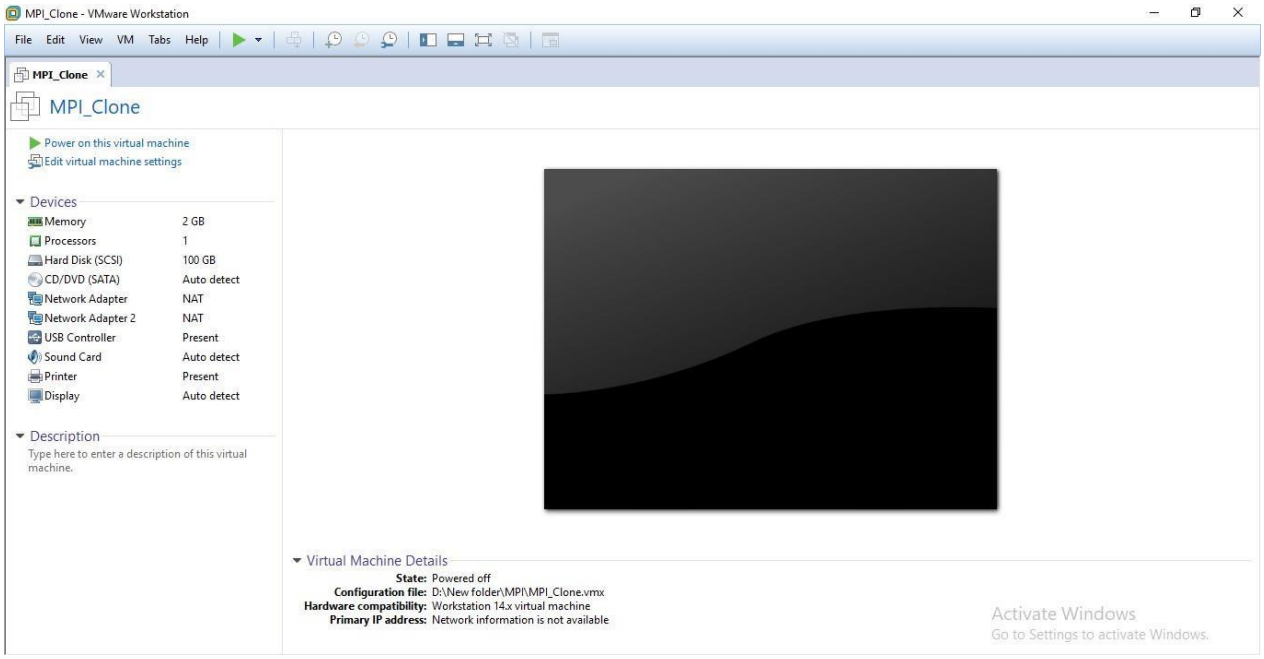
# Experiment 11

**Aim:** Installation of Linux distribution (RHEL) server

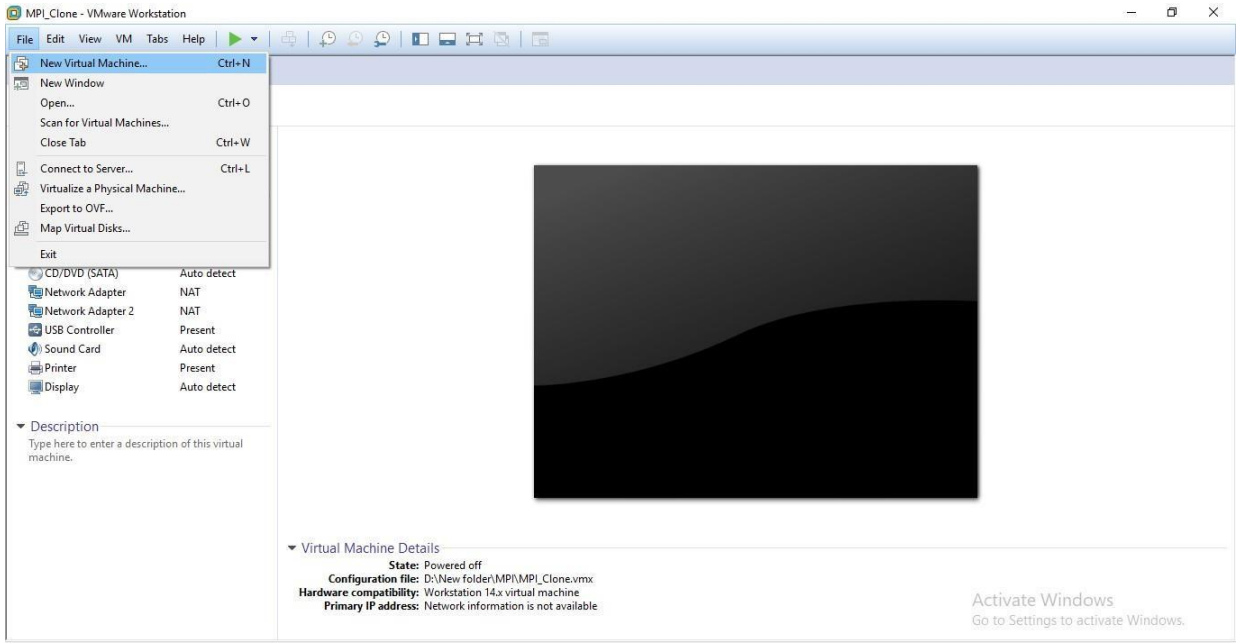
Linux can be installed either through your drive by using a DVD-ROM oris contemporarily available for installation even through USB stick. We will be considering installation of Linux via ISO files (which can be downloaded through the internet).

## Steps: Installation of Linux via ISO file

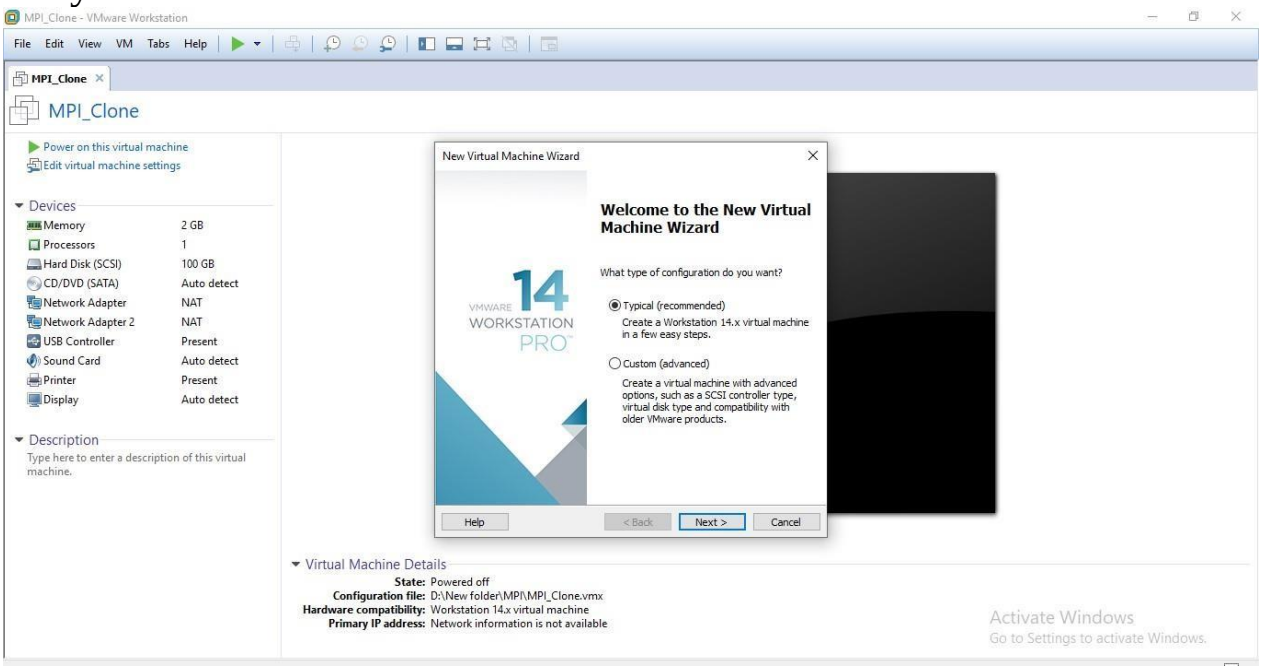
1) After installation in the chosen drive *open* **VMWare Workstation**from your machine. Make sure the drive (C: /D: /E:) wherever youinstall from the ISO file has adequate space in GBs prior to installation (this is the start screen after you open the VMWare Workstation application).



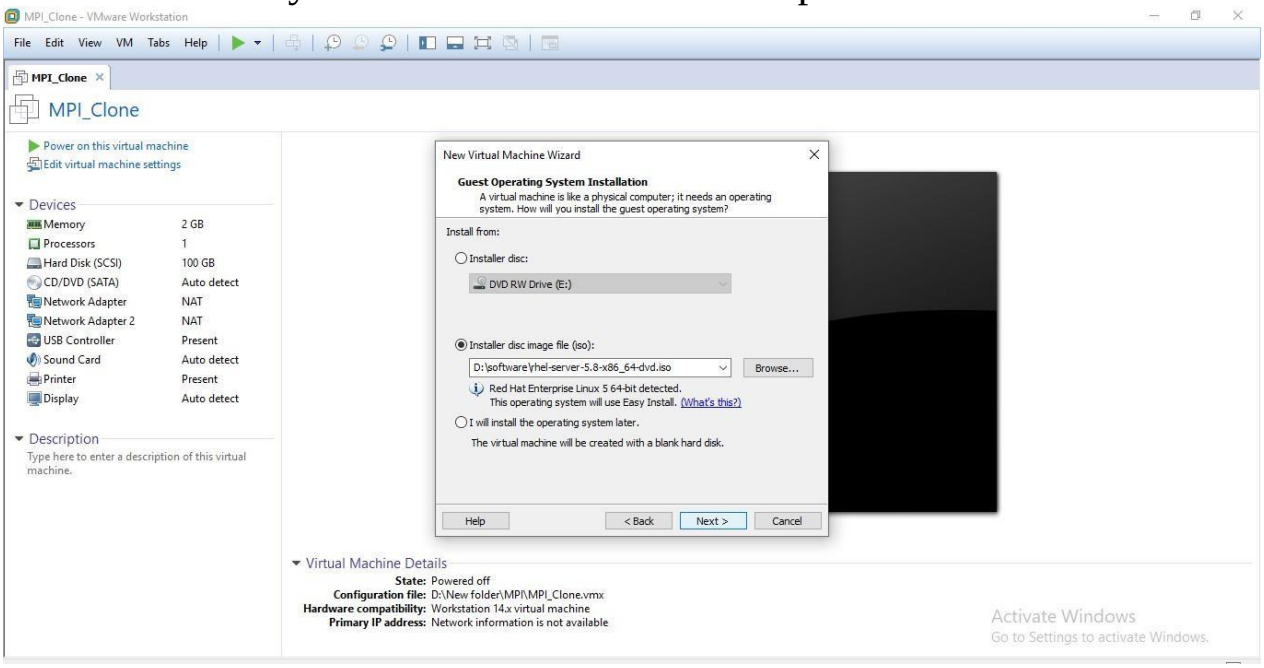

2) Whenever you want to use Linux you can create a VM (virtualmachine) in this way. After you *click* on "**FILE**" go to "**New Virtual Machine**"



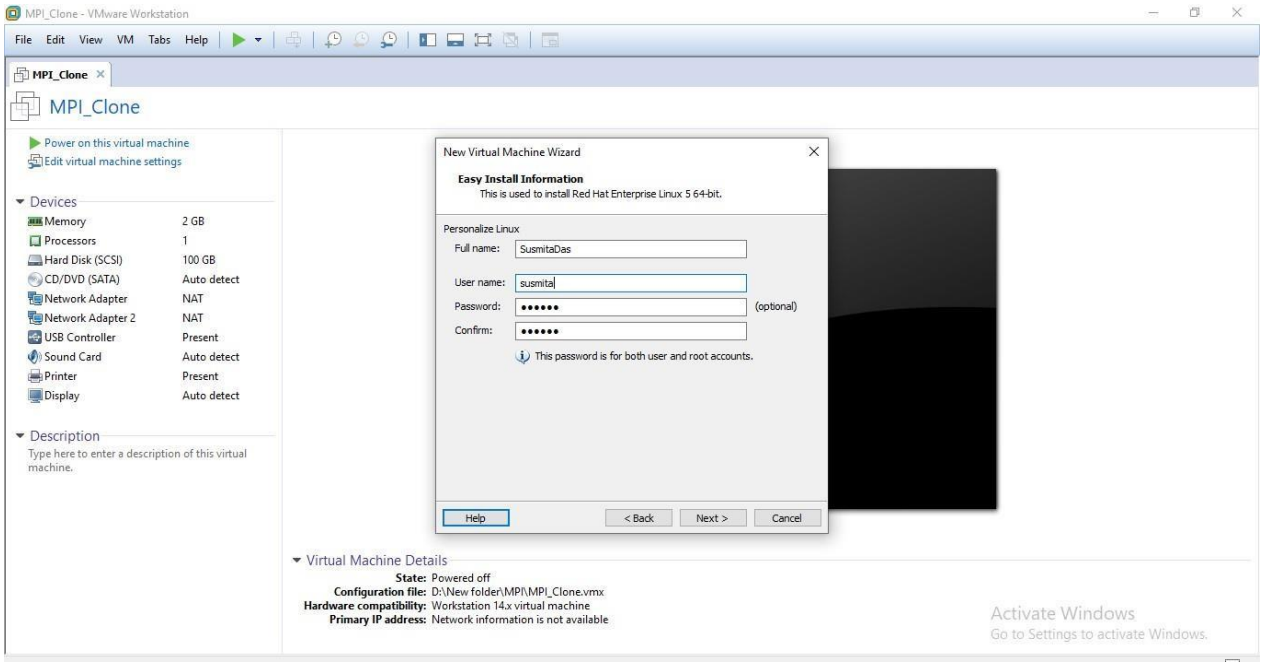
3) Follow the next set of steps for installing the VMWareWorkstation on your device.



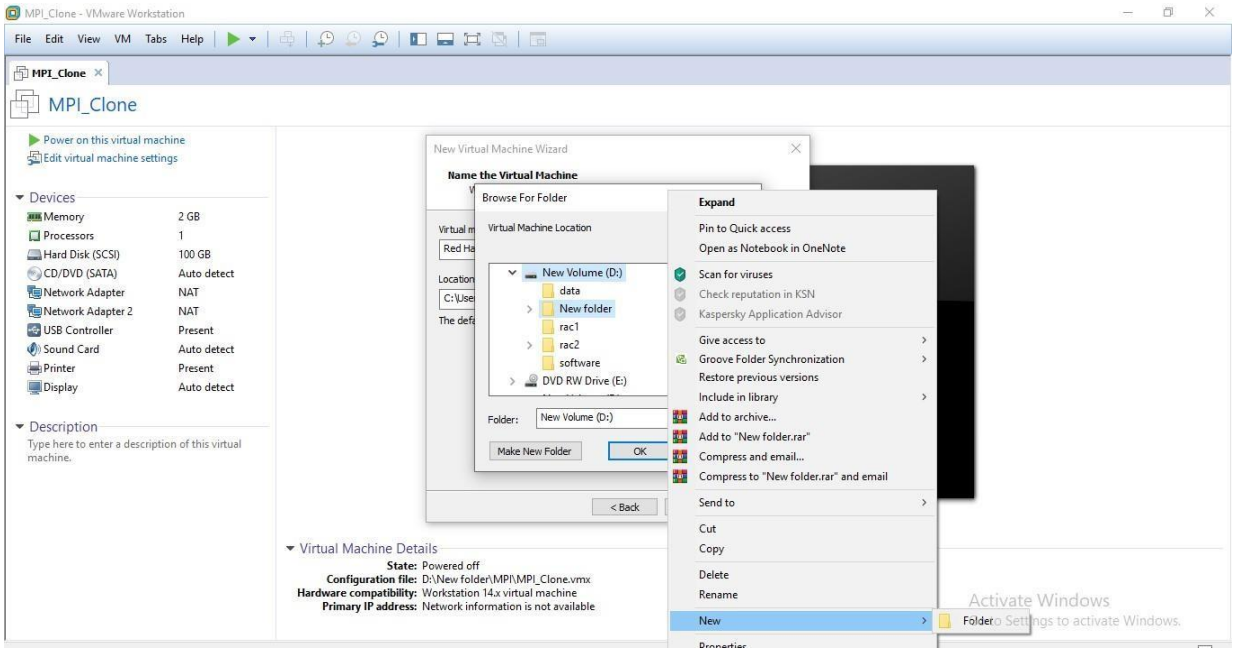

4) Choose from where you want to *copy* the ISO that you have downloaded on your machine & if saved on a particular drive.

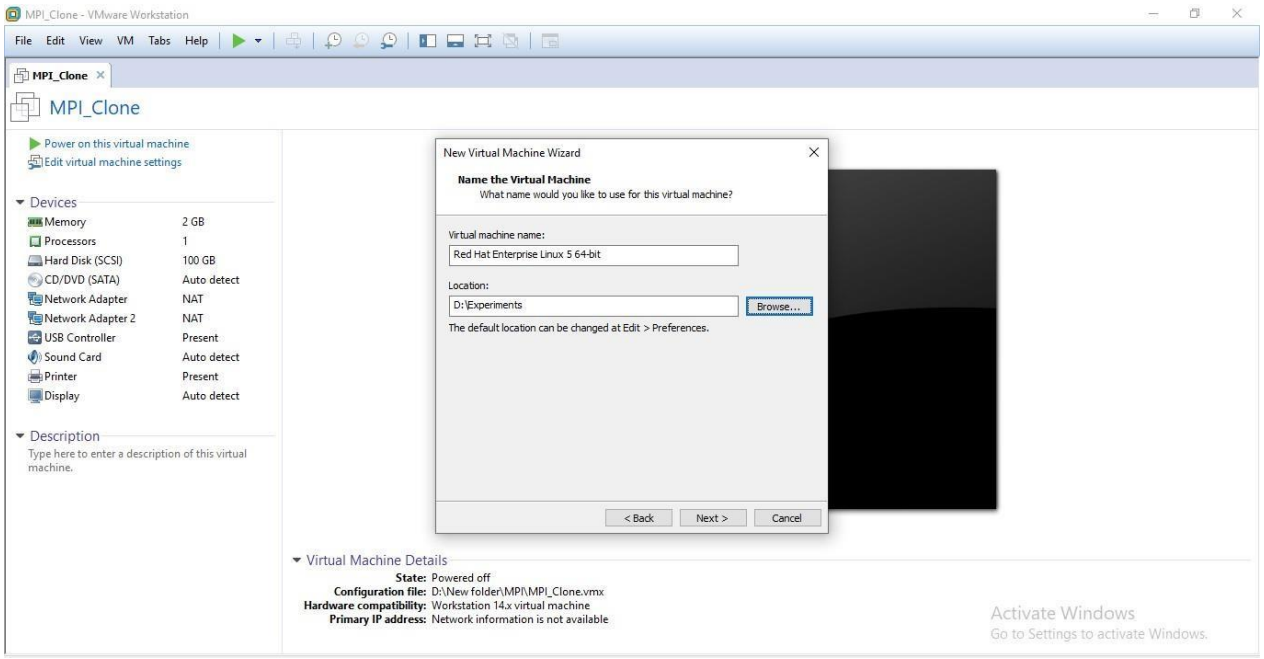



5) Provide details for the “personalized Linux set-up”|

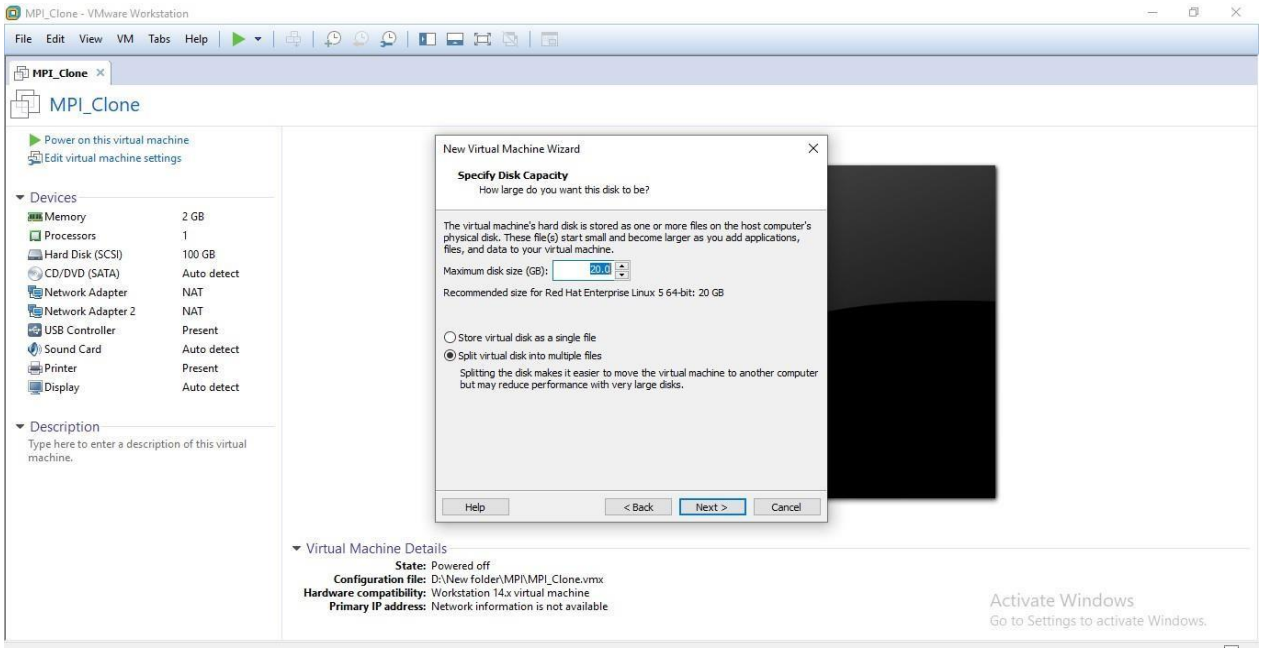



6) Provide a space (drive) to save your data for the VM created.

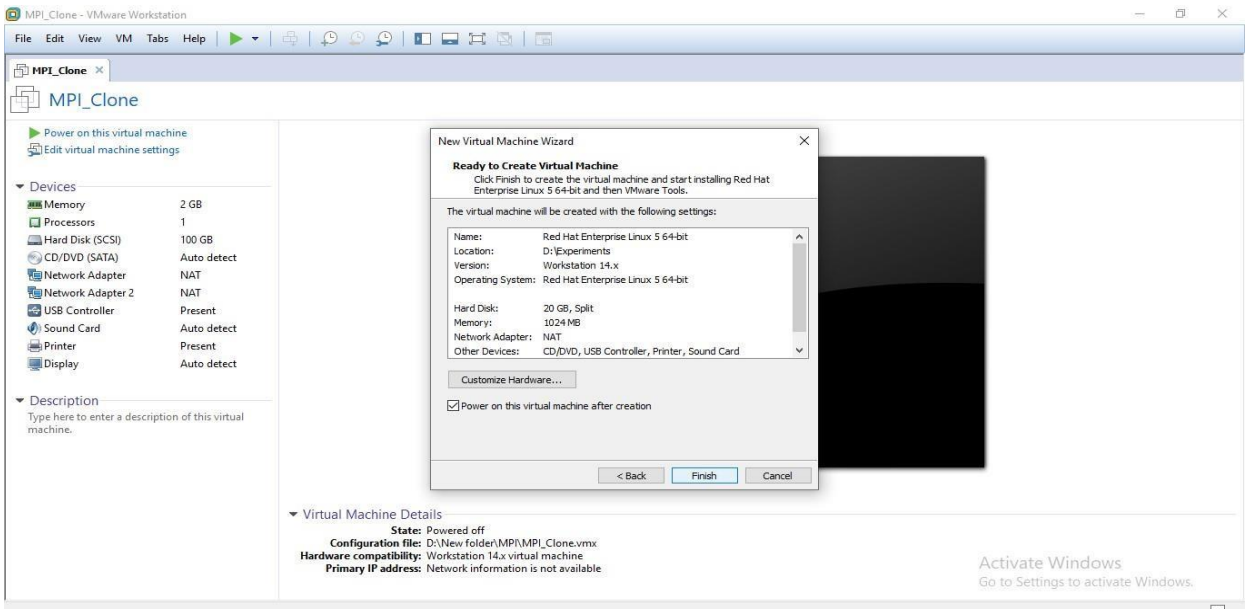




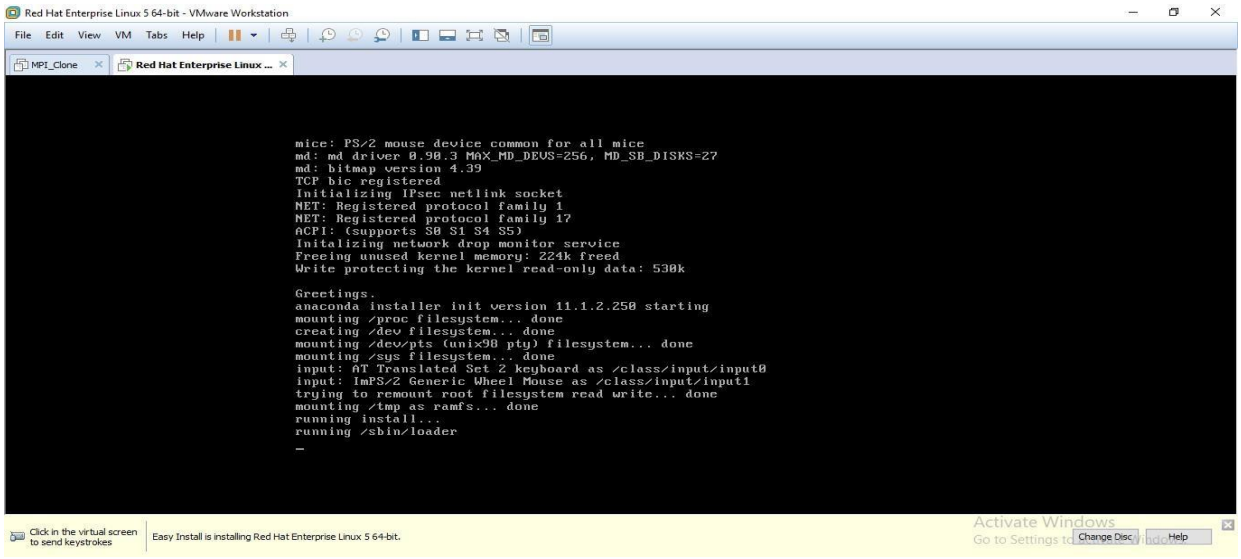

7) Specify the Disk capacity you want to allocate for the VM

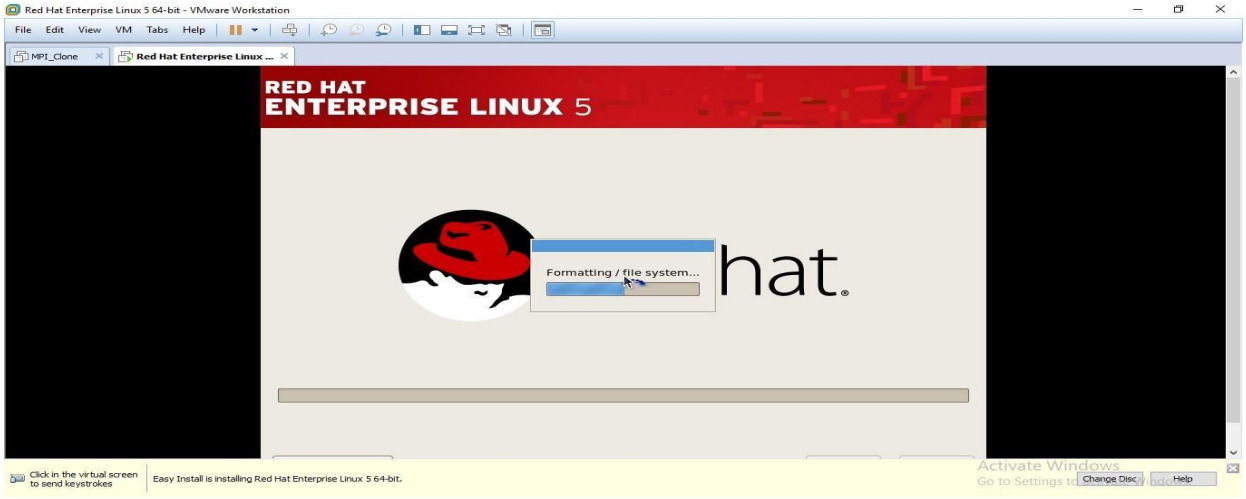


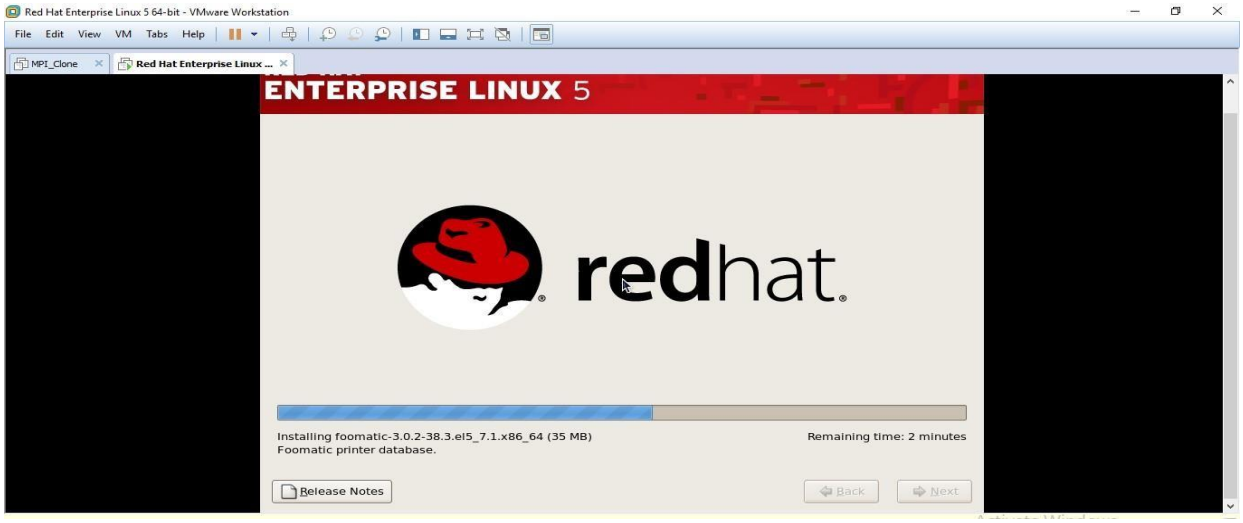

8) The VM is thus ready to be installed.



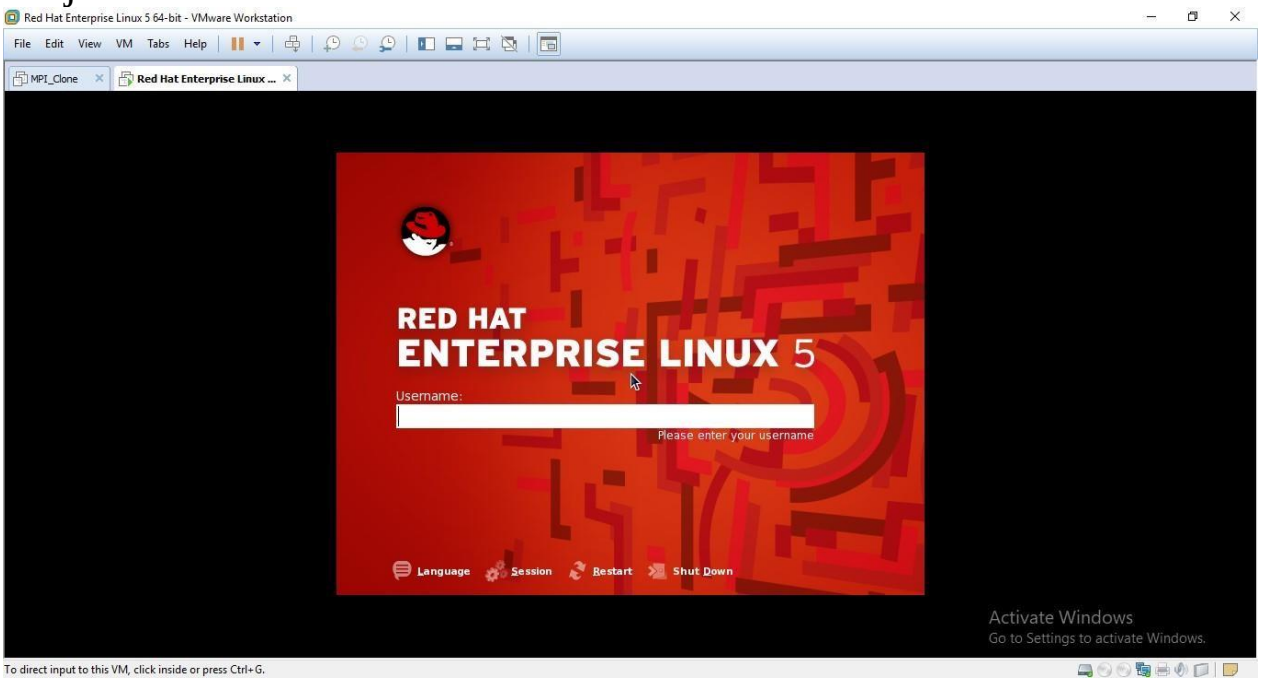

9) Following are the steps in auto-installing RHEL on your machine.


Red Hat Enterprise Linux 5 64-bit - VMware Workstation
File Edit View VM Tabs Help
MPI_Clone
Red Hat Enterprise Linux ...
mice: PS/2 mouse device common for all mice
md: md driver 0.90.3 MAX_MD_DEVS=256, MD_SB_DISKS=27
md: bitmap version 4.39
TCP bic registered
Initializing IPsec netlink socket
NET: Registered protocol family 1
NET: Registered protocol family 17
ACPI: (supports S0 S1 S4 S5)
Initalizing network drop monitor service
Freeing unused kernel memory: 224k freed
Write protecting the kernel read-only data: 530k
Greetings.
anaconda installer init version 11.1.2.250 starting
mounting /proc filesystem... done
creating /dev filesystem... done
mounting /dev/pts (unix98 pty) filesystem... done
mounting /sys filesystem... done
input: AT Translated Set 2 keyboard as /class/input/input0
input: ImPS/2 Generic Wheel Mouse as /class/input/input1
trying to remount root filesystem read write... done
mounting /tmp as ramfs... done
running install...
running /sbin/loader
Click in the virtual screen to send keystrokes
Easy Install is installing Red Hat Enterprise Linux 5 64-bit.
Change Disc
Help

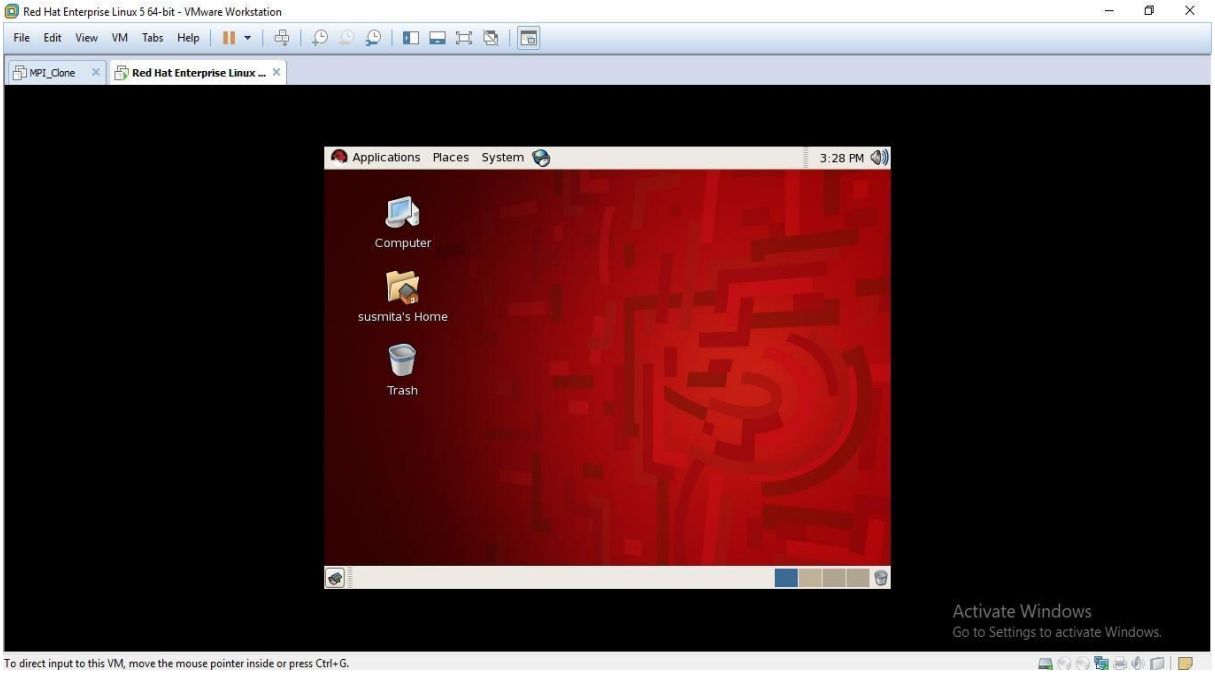

Red Hat Enterprise Linux 5 64-bit - VMware Workstation
File Edit View VM Tabs Help
MPI_Clone
Red Hat Enterprise Linux ...
RED HAT
ENTERPRISE LINUX 5
Formatting / file system...
Click in the virtual screen to send keystrokes
Easy Install is installing Red Hat Enterprise Linux 5 64-bit.
Change Disc
Help


Red Hat Enterprise Linux 5 64-bit - VMware Workstation
File Edit View VM Tabs Help
MPI_Clone
Red Hat Enterprise Linux ...
ENTERPRISE LINUX 5
redhat
Installing foomatic-3.0.2-38.3.el5_7.1.x86_64 (35 MB)
Foomatic printer database.
Remaining time: 2 minutes
Release Notes
Back
Next

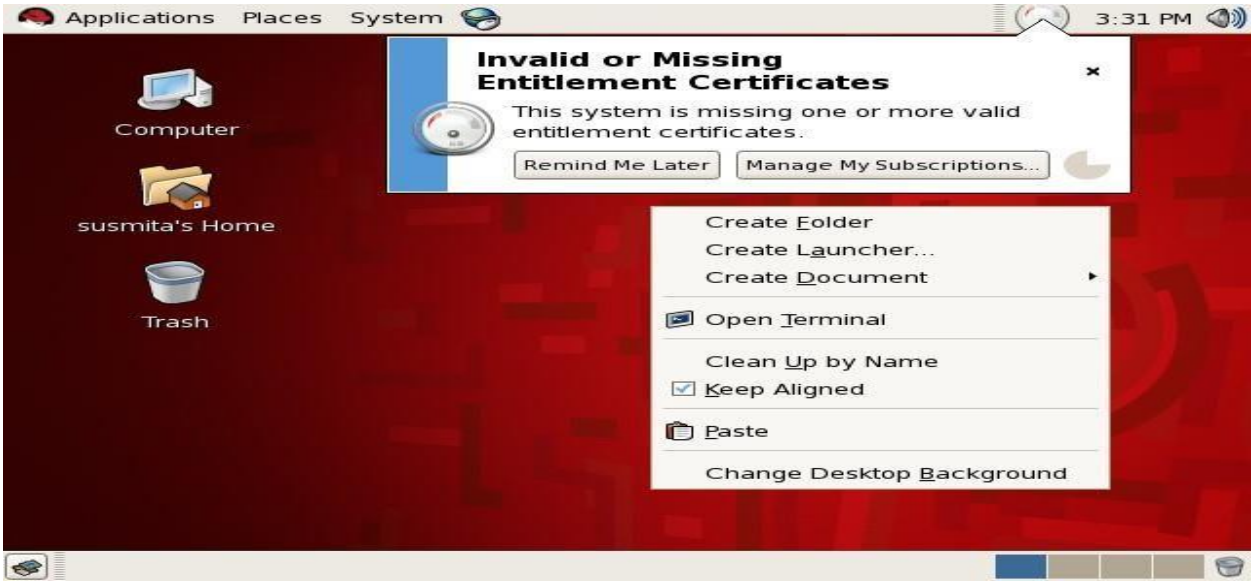
10) This is your **Homepage** for RHEL where you submit your details justcreated.



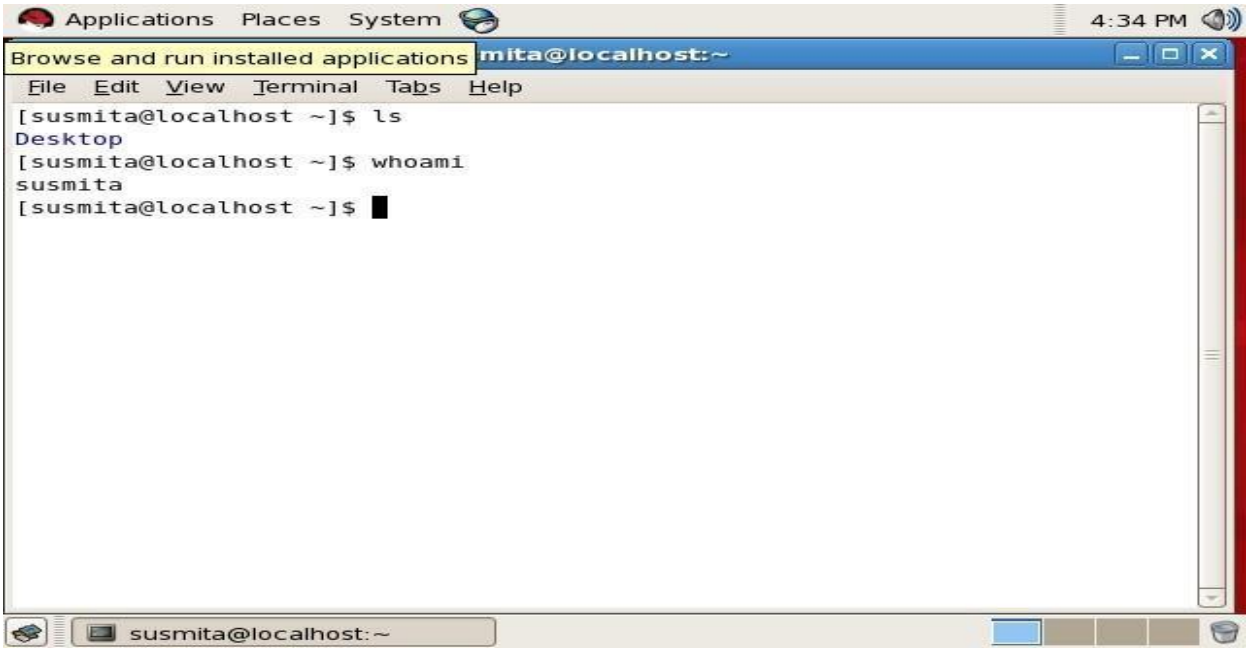

**11)** Enter your **Username** and **Password**




12) On your **Home** screen, Right click to get the next screen (**The CLIterminal screen for Linux**).




You can check a few commands (like "*whoami*") to identify the "localhost" of the current machine.




**Hence, we have understood and executed the steps for successfully installing Linux in our system for further usage.**

Name : Vaishnavi V. Poti
Batch : S21
Roll no :78

# Experiment no. 12

Aim: Configuration ofNFS (Network File System) Server and transfer files to a Linux Client.

Theory: NFS (Network File System) is a distributed file system. An NFS server has one or more file systems that are mounted by NFS clients. These file systems are mounted by using the standard UNIX mount commands. NFS is the standard method to share files between Linux and Unix-like systems.

NFS is a networking protocol for distributed file sharing. A file system defines the way files are stored and retrieved from storage devices, such as hard disk drives, solid-state drives and on tape drives across networks. It enables network administrators to share all or a portion of a file system on a networked server to make it accessible to remote computer users. NFS is one of the most widely used protocols for file servers. Cloud vendors also implement the NFS protocol for cloud storage, including Amazon Elastic File System, NFS file shares in Microsoft Azure and Google Cloud Filestore.

How does the Network File System work?

NFS is a client-server protocol. An NFS Server is a host that meets the following requirements:

- has NFS Installed
- has at least one network connection for sharing NFS resources; and
- is configured to accept and respond to NFS requests over the network connection.

An NFS Client is a host that meets the following requirements:

- has NFS client software installed has network connectivity to an NFS server is authorized to access resources on the NFS server; and is configured to send and receive NFS requests over the network connection.

NFS was initially conceived as a method for sharing file systems across workgroups using UNIX. It is still often used for ad hoc sharing of resources.

Installation of NFS Server

Install the package for "nfs-kernel-server" on your machine.

## Installation of NYS Server

Install the package for "nfs-kernel-server" on your machine.

root@ubuntu: —

```
root apt install nfs-kernel-server Reading package
lists. Done
Bui Iding dependency tree
Reading state information. .. Done nfs-kernet-server is already the
newest version (1 : 1.3.4-2.5ubuntu3.4) .
The following packages were automatically installed and are no longer required :
  gimp-data 065-va-driver intel-media-va-drtver libaacso Iibamd2 Ii-baomo
  tibavcodec58 Iibavformat58 t ibavuti156 libbabt-O.1-O tibbdptuso tibbtas3
  libbluray2 ltbcamd2 IibccoIamd2 ltbcholmod3 libchromaprintl ltbcodec2-o.9
  Iibcotamd2 1tbde265-O IibgegI-O .4-0 tibgegt-common Iibgfortran5 tibgimp2.O
  IibgmeO Iibgsml t ibheifl I ibtgdgmmll I ibiImbase24 IibIapack3 Iibmetts5
      Iibmng2 Iibmypaint-I.5-I Iibmypaint-common Iibopenexr24 IibopenmptO
  IibquadmathO libraw19 1 tbsdt2-2.0-0 1 tbshtne3 tibsnappy1v5 Iibssh-gcrypt-4
  Iibsuttesparseconftg5 IibswresampIe3 IibswscaIe5 ItbumfpackS Iibva-drm2
  Iibva-xII-2 L tbva2 Itbvdpaul Iibx264-155 Iibx265-179 Iibxvtdcore4
Iibzvbt-common ItbzvbtO mesa-va-drivers mesa-vdpau-drtvers oct-
icd-libopencll 'yea-driver-all vdpau-drtver-all use ' apt auto remove'
to remove them .
  u raded O newt installed . O to remove and 47 not u raded r oot@ubuntu :
```

root@ubuntu: —

```
  Iibsuitesparseconfig5 IibswresampIe3 IibswscaIe5 Iibumfpack5 Iibva-
  drm2 Iibva-xII-2 1 ibva2 Iibvdpaul 11bx264-155 libx265-179
  11bxvtdcore4
Iibzvbi-common IibzvbiO mesa-va-drivers mesa-vdpau-drivers oct-icd-
ltbopencll va-drtver-all vdpau-drtver-all Use ' apt auto remove' to remove
them .
O upgraded, O newly installed, O to remove and 47 not upgraded .root@ubuntu
: root@ubuntu : root@ubuntu :
root          apt install nfs- common Reading
package       Done
```

Building dependency tree
Reading state information... Done nfs-common is already the newest version (I : I. 3.4-2. subuntu3.4) .
The following packages were automatically installed and are no longer required : gimp-data 1965-va-drtver intel-medta-va-drtver libaacsO I ibamd2 libaomO 1ibavcodec58 Iibavformat58 Iibavuti156 libbab1-O.1-O Iibbdp1usO Iibb1as3 I tbbIuray2 Iibcamd2 IibccoIamd2 IibchoImod3 Iibchromaprintl I tbcodec2-O.9 Iibcotamd2 Iibde265-O Iibgeg1-O.4-O Iibgegl-common Iibgfortran5 Iibgimp2.O I ibgmeO Iibgsml Iibheifl libigdgmmll libitmbase24 IibIapack3 Iibmetis5 I tbmng2 Iibmypatnt-1.5-I Iibmypatnt-common Iibopenexr24 LibopenmptO Iibquadmatho Iibraw19 Iibsdt2-2.0-0 Iibshine3 Iibsnappy1v5 Iibssh-gcrypt-4 I tbsuitesparseconfig5 t IbswresampIe3 IibswscaIe5 Iibumfpack5 Iibva-drm21ibva-x11-2 1 ibva2 Itbvdpaul libx264-155 libx265-179 11bxvidcore4 I ibzvbi-common I tbzvbiO mesa-va -drivers mesa -vdpau-drivers oct-tcd-ltbopencll va-drtver-all vdpau-drtver-all Use ' apt auto remove' to remove them .
O u raded O newt installed , O to remove and 47 not u raded root@ubuntu :

Create the folder that needs to be shared,

```
root@ubuntu:~#
root@ubuntu:~# cd /mnt
root@ubuntu:/mnt#
root@ubuntu:/mnt#
root@ubuntu:/mnt# ls -lrt
total 0
root@ubuntu:/mnt#
root@ubuntu:/mnt# mkdir nfsshare
root@ubuntu:/mnt#
root@ubuntu:/mnt# ls -lrt
total 4
drwxr-xr-x 2 root root 4096 Apr 27 07:35 nfsshare
root@ubuntu:/mnt#
```

Provide permissions as below,

```
root@ubuntu:/mnt#
root@ubuntu:/mnt# chown -R nobody:nogroup nfsshare
root@ubuntu:/mnt#
root@ubuntu:/mnt# ls -lrt
total 4
drwxr-xr-x 2 nobody nogroup 4096 Apr 27 07:35 nfsshare
root@ubuntu:/mnt#
root@ubuntu:/mnt#
root@ubuntu:/mnt#
root@ubuntu:/mnt# chmod -R 777 nfsshare/
root@ubuntu:/mnt#
root@ubuntu:/mnt#
root@ubuntu:/mnt# ls -lrt
total 4
drwxrwxrwx 2 nobody nogroup 4096 Apr 27 07:35 nfsshare
root@ubuntu:/mnt#
root@ubuntu:/mnt#
```

Edit "/etc/exports" with below entry with IP or ( * ) for all servers,

```
# /etc/exports: the access control list for filesystems which may be exported
#               to NFS clients.  See exports(5).
#
# Example for NFSv2 and NFSv3:
# /srv/homes       hostname1(rw,sync,no_subtree_check) hostname2(ro,sync,no_sub
tree_check)
#
# Example for NFSv4:
# /srv/nfs4        gss/krb5i(rw,sync,fsid=0,crossmnt,no_subtree_check)
# /srv/nfs4/homes  gss/krb5i(rw,sync,no_subtree_check)

/mnt/nfsshare  192.168.75.129(rw,sync,no_root_squash,insecure)
~
~
~
~
```

//add the IP address of server with details of the file being sharedCheck the status of NFS service after making these changes,

| r oot@ubuntu : —-4 root@ubuntu:—# systemctl [redacted] status nfs-kernel- server nfs-server . service NFS server and services |
|---|
| Loaded : loaded ( /lib/systemd/system/nfs -server . service; enabled; vendor p |
| Active : active (exited) since wed 2022-04-27 PDT ; 34s ago<br>Process : 6243 ExecStartPre=/usr/sbin/exportfs - r (code—exited, status=O/SU<br>Process : 6244 Execstart=/usr/sbin/rpc . nfsd SRPCNFSDARGS (code—exited, stat<br>main PID: 6244 (code—exited, status=O/SUCCESS)<br>Apr 27 07:44:40 ubuntu systemd[l]: Starting NFS server and services... A r 27 07:44 •.41 ubuntu systemd[l] •. F int shed NFS server and services. |

Check the firewall status,

```
root@ubuntu:~# ufw status
Status: inactive
root@ubuntu:~#
root@ubuntu:~#
```

Ex ort the NFS folder and mount the shared folder,

```
root@ubuntu:/mnt#
root@ubuntu:/mnt# exportfs -rav
exportfs: /etc/exports [3]: Neither 'subtree_check' or 'no_subtree_check' speci
fied for export "*:/mnt/nfsshare".
  Assuming default behaviour ('no_subtree_check').
  NOTE: this default has changed since nfs-utils version 1.0.x

exporting *:/mnt/nfsshare
root@ubuntu:/mnt#
root@ubuntu:/mnt#
root@ubuntu:/mnt# mount 192.168.75.129:/mnt/nfsshare  /mnt/nfs_client_Share
root@ubuntu:/mnt#
root@ubuntu:/mnt# ls -lrt
total 8
drwxrwxrwx 2 nobody nogroup 4096 Apr 27 07:35 nfsshare
drwxrwxrwx 2 nobody nogroup 4096 Apr 27 07:35 nfs_client_Share
root@ubuntu:/mnt#
```

root@ubuntu:/mnt#

Folder actin as server, has a "a.txt" file

cd nfsshare/

root@ubuntu : / mn t / nf s s ha

root@ubuntu : /mnt/nfsshare#

root@ubuntu : / mnt/nfsshare#

root@ubuntu : /mnt/nfsshare#

root@ubuntu:/mnt/nfsshare# Is -I rt

total [redacted]

I 27 08: 06

root root 28 Apr a. txt [redacted]

root@ubuntu : /mnt/ nf s sha

root@ubuntu : /mnt/nfsshare# cat [redacted]

| This | file is | a. txt shared on NFS. [redacted] |
|---|---|---|

root@ubuntu : /mnt /nfsshare#

Text file acting as a server has a txt file and is also now visible via Client share machine

```
root@ubuntu:/mnt#
root@ubuntu:/mnt#
root@ubuntu:/mnt# cd nfs_client_Share/
root@ubuntu:/mnt/nfs_client_Share#
root@ubuntu:/mnt/nfs_client_Share#
root@ubuntu:/mnt/nfs_client_Share# ls -lrt
total 4
-rw-r--r-- 1 root root 28 Apr 27 08:06 a.txt
root@ubuntu:/mnt/nfs_client_Share#
root@ubuntu:/mnt/nfs_client_Share#
root@ubuntu:/mnt/nfs_client_Share# cat a.txt
This file is shared on NFS.
```

Conclusion: Hence, we have studied about the configuration of NFS Server and transferred files across Linux clients and servers.

Meet A. Raut
S21 84

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

# Assignment 1

Q.1) Given a system with n processes, how many possible ways processes can be scheduled?

Ans. The number of possible ways n processes can be scheduled depends on the scheduling algo used. For eg. in preemption scheduling algo., each process can be interrupted and switched out leading to different interleaving. The number of permutations can be calculated as n factorial where $n! = n(n-1)(n-2)\ldots \times 2 \times 1$ where n is a processes

1. FCFS :

In this method the process are executed in the order they arrive. Only one scheduling time can be determined. Here it may lead to starvation.

2. SJF :

Here the process having smallest burst time or execution time are executed first. In non premptive SJF, a process with shorter burst time or execution time. This method of scheduling usually maximizes waiting & turnaround time.

3. RR :

The processes are assigned a fixed time quantum for execution. Processes are excited in a cyclic manner & the scheduling process depends on factors such as the time quantum & process order.

4. Priority scheduling :

Processes are executed based on their priority. If the higher priority process are executed first. However this may lead to starvation for lower priority processes.

Q.2) Define OS. Enlist functions of OS.

Ans. An OS acts as an interface between the user and the hardware of the computer and also controls the execution of the application programs. OS is also called as resource manager. The most important functions of the OS are:

i) Process Management:

Creation, execution and deletion of user and system processor syncronization, inter process communication and deadlock handling for processes.

ii) Memory management:

Allocating primary as well as secondary memory to user and system processes

iii) File management:

Creation and deletion of files and directories and backup of file.

iv) Device management:

Keep an eye on device driver communicate, control and monitor the device drive

v) Protection and security:

Provide user authentication, files attributes such as read, write, encryption and back-up.

Q. State 2 advantages of kernel level threads among the two thread types which one is better?

Ans.

Two advantages of kernel - level threads are :

(i) Concurrency handlings : kernel level threads can run in parallel on multiple processors or core, allowing for efficient utilization of system resources and improved overall system performance.

(ii) Independance from user-level code :

kernel level threads are managed by the OS kernel making them less dependent on user level code. This leads to better stability and reliability because the kernel can manage thread execution and resources more effectively

kernel level threads offer advantages in terms of concurrency and system control by may have higher overhead. User level thread provide more flexibility are often eighter in terms of resource usage but might not take full advantage of multi core systems.

Q. Define PCB. Enlist 3 components of PCB.

Ans. Any process is identified by its process control block (PCB). PCB is data structure used for keeping track on the process by OS. All the informetic associated with the process is kept in its PCB. These is a seperate PCB for each process. The three components of PCB are :

i) Pointer :

This field points to other processes PCB. The scheduling list is maintained by pointer.

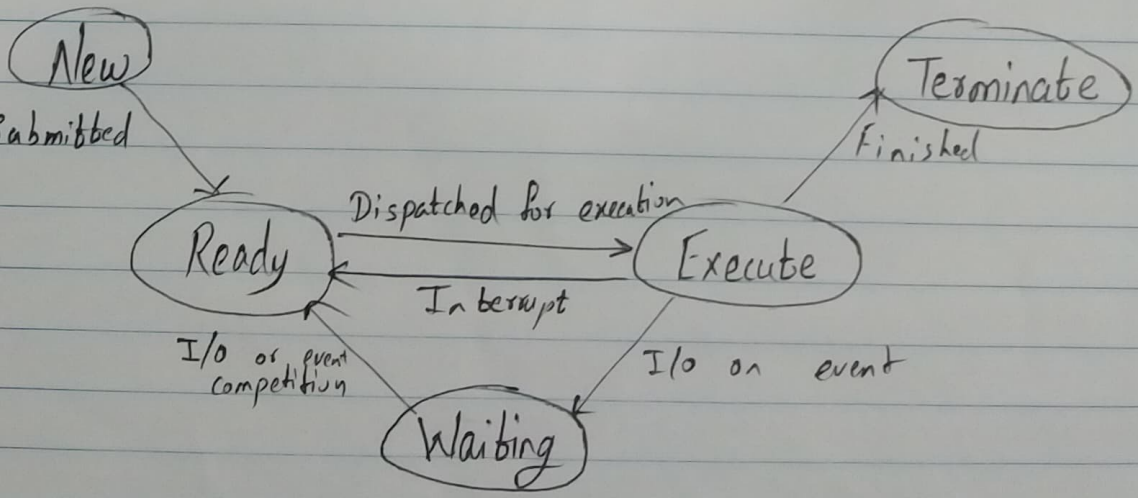
ii) Current State :

Currently process can be in any of the state from new ready, executing, waiting, etc.

iii) Process IW :

Identification number of process. OS assign this number to process to distingwish it from other processes.

Q. Sketch 5 state process state transition model.



THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Q for each of the following indicate wether transition is possible or not if yes, explain.

Ans.

- Executing/Running → Ready

yes, this transition is possible, whenever an interrupt is generated the processes goes back in the Ready Queue and stops executing.

- Executing / ~~Running~~ waiting →

yes, whenever the process in running and require an I/O or a event to occur to continue the process then it enters waiting state to wait for I/O or event to occur.

- Waiting → Execute

No, when as explained above the process after event/I/O competition will transition to ready state first and then to execution state.

- Terminated → waiting

No, after termination it cant change to any state.

Q. State one principle of concurrency in detail.

Ans.

The ~~process~~ principle of process synchronization :

In a single processor multiprogramming system processors are not executed concurrently. In order to get the appearance of concurrent execution, a fixed time slot is allocated to each process. After utilization of each time slot, CPU gets allocated to another process. Such switching of CPU back and forth between processes is called as context switch. At a time single process gets executed so parallel processing cannot be accomplished. Also there is definite amount of overhead drawn in switching back and forth between processes. Apart from above limitation, interleaved execution ~~often~~ offers major benifit in processing efficiency and in program-structuring in a multi processor system, interleaving and overlapping of multiple processes is achievable. The comparative speed of execution of processes depends on activities and behaviour of other processes.

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

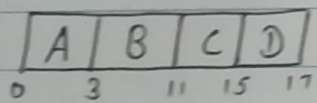
Q. For the processes listed, draw gantt chart of these processes using FCFS, SJF. Find avg waiting and avg turnaround time.

Ans.

For FCFS:

| Process | Arrival Time | Burst Time | Completion Time | TAT | WT |
|---|---|---|---|---|---|
| A | 0.001 | 3 | 3 | 3 | 0 |
| B | 1.001 | 8 | 11 | 10 | 2 |
| C | 4.001 | 4 | 15 | 11 | 7 |
| D | 6.001 | 2 | 17 | 11 | 9 |

Gantt chart:



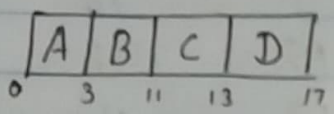

Avg turnaround time = (3 + 10 + 11 + 11)/4
= 8.75

Avg waiting time = (0 + 2 + 7 + 9)/4
= 4.5

ii) SJF

| Process | AT | BT | CT | TAT | WT |
|---|---|---|---|---|---|
| A | 0.001 | 3 | 3 | 3 | 0 |
| B | 1.001 | 8 | 11 | 10 | 2 |
| C | 4.001 | 4 | 17 | 13 | 9 |
| D | 6.001 | 2 | 13 | 7 | 5 |

gantt chart:




Avg. TAT = (3+10+13+7) /4
= 5.75

~~Avg.~~

Avg. WT = (0+2+9+5)/4
= 4

Meet Raut
S21 84

# Assignment 2

Q.1)

Ans a) Deadlock avoidance and prevention.

| Deadlock avoidance | Deadlock prevention |
|---|---|
| i) It involves dynamically examining resource allocation request to ensure deadlock do not occur | i) It focuses on structually negating one of the four necessary cond$^n$ for deadlock: Mutual exclusion, hold and wait, no preemption, circular wait |
| ii) This allows processes to request resource while ensuring system is in safe state. | ii) This allows designing the system in such a way that deadlock can't occur by eliminating one or more of necessary cond$^n$ |
| iii) Algo can be complex and may require overhead | iii) Algo are generally less complex |
| iv) It leads to better resource utilisation as resources are dynamically being allocated | iv) It may not optimise the resource utilization since it relies on defined rules. |

⑥ RAG and banker's algo.

| Resource Allocation Graph | Banker's Algorithm |
|---|---|
| i) It is deadlock prevention method used in operating system to represent resource allocation | i) It is a deadlock avoidance method to allocate resource available dynamically by checking its |
| ii) RAG works by representing processes and resources as nodes in a graph with edges denoting resource | ii) Banker's algo operates by maint the max demand of each proces and the available resources in the sy |
| iii) The system overload is reduced but the time delay may increase due to checking safety cond$^n$ for every proces | iii) System overload may occur as dynamic allocation is carried out but it may increase the system sta |
| iv) RAG may not optimize the utilization of resource as process may need to wait for safety cond$^n$ check | iv) Since the resources are alloc dynamically, bankers algo optimizes the resources utilizatio |

(Q.2)

Ans $Need[i,j] = Max[i,j] - Allocation[i,j]$

| Processes | Need | | | |
|---|---|---|---|---|
| | A | B | C | D |
| $P_0$ | 2 | 1 | 0 | 3 |
| $P_1$ | 1 | 0 | 0 | 1 |
| $P_2$ | 0 | 2 | 0 | 0 |
| $P_3$ | 4 | 1 | 0 | 2 |
| $P_4$ | 2 | 3 | 2 | 0 |

Need < Available

$i=0$ : Need = 2,1,0,3
Available = 0,3,0,1
Condⁿ is false
∴ Finish[0] = F

$i=1$ Need 1 ≤ Work
(1,0,0,1) ≤ (0,3,0,1)
Condⁿ is false
Finish[1] = F

$i=3$ Need 2 ≤ Work
(0,2,0,0) ≤ (0,3,0,1)
Condⁿ True
Finish[3] = T

i = 4 Need 3 ≤ Work
(4,1,0,2) ≤ (3,4,2,2)
Cond^n false

i = 5 Need 0 ≤ Work
(4,1,0,2) (3,4,2,2)
Cond^n is false

Now, Work = Allocated (0) + Work = (3,0,1,4) + (7,6,3,4)
Work = (10,6,4,8)
Release process

For process $P_1$:

Need 1 ≤ Work ⇒ cond^n true
Finish [1] = T
Work = (12,8,5,8)

Now, For process $P_3$:
Is Need 3 ≤ Work
(4,1,0,2) (12,8,5,8)

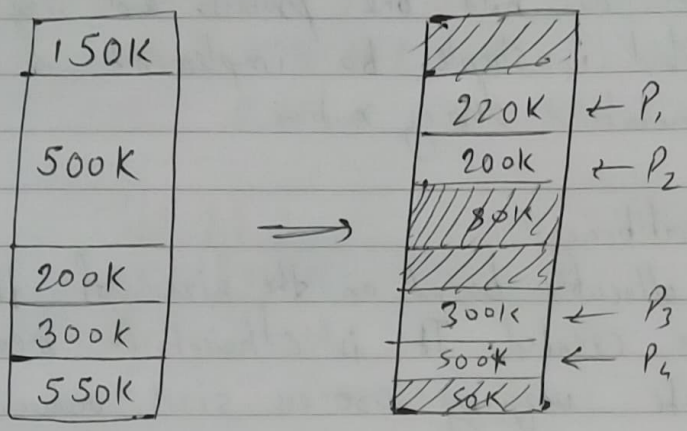
$P_2 \rightarrow P_4 \rightarrow P_0 \rightarrow P_1 \rightarrow P_3$

Q.3)
Ans. a) Partitioning: Virtual memory management brings process into main memory for executing by the processor involves virtual memory based on segmentation and paging.

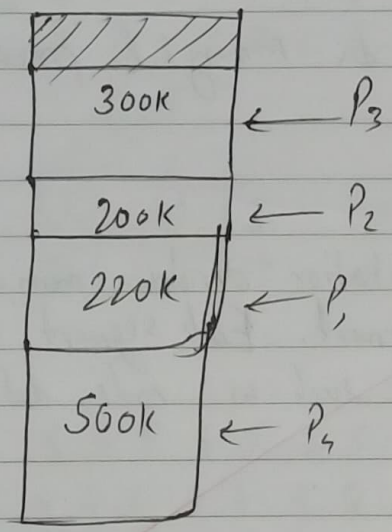
Types of memory partitioning:

Q.4)

Ans. Next Fit:




Best fit:



I) Fixed partitioning:

In this, memory is divided into fixed size partitions. Each partition can hold one process, and size of partition can. This method is simple to implement & there is no fragmentation during runtime

II) dynamic partitioning:

It allocates based on the size of the process partitions are created. It is efficient in memory usage can accomodate varying process sizes. reduces usage of memory.

III) paging:

It divides memory & processes into fixed sizes block called frames.

IV) Segmentation:

Segmentation divides memory & processes. It has variable size segment. Each segment represents a logical unit of a program. such as codes, data or stack.

Q.5) FIFO

Ans.

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $F_1$ | 1 | 1 | 1 | 4 | 4 | 4 | 6 | 6 | 6 | 6 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 2 |
| $F_2$ | | 2 | 2 | 2 | 5 | 5 | 5 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 5 | 5 | 5 | 5 | 5 |
| $F_3$ | | | 3 | 3 | 3 | 3 | 1 | 1 | 1 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 4 | 4 | 4 | 4 |
| | | | | H | H | | | | | H | H | | | H | H | H | | | H | H |

Total hit: 9
Total fault: 13
Page size: 22

LRU:

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $F_1$ | 1 | 1 | 1 | 4 | 4 | 4 | 4 | 4 | 4 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 5 | 5 | 5 | 5 |
| $F_2$ | | 2 | 2 | 2 | 5 | 5 | 5 | 1 | 1 | 1 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 4 | 4 | 4 | 4 |
| $F_3$ | | | 3 | 3 | 3 | 3 | 3 | 6 | 7 | 6 | 6 | 6 | 6 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 2 |
| | | | | | H | H | | | | | H | H | | | H | H | H | | | H | H |

Total hit = 9
Total fault = 13

Optimal:

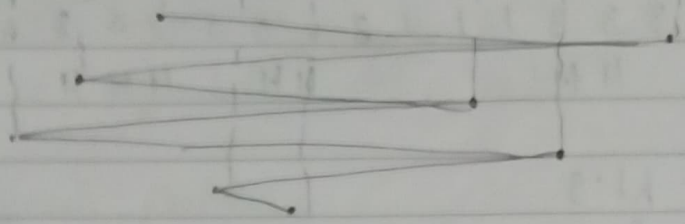
| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $F_1$ | 1 | 1 | 1 | 1 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 9 | 9 | 9 | 9 | 5 | 5 | 5 | 5 | 5 |
| $F_2$ | | 2 | 2 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 4 | 4 | 4 | 4 |
| $F_3$ | | | 3 | 3 | 3 | 3 | 3 | 1 | 6 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 2 |
| | | | | | H | H | | | | | H | H | | H | | H | H | | | H | H | |

Total hit = 9
Total fault = 13

(Q.6)

Ans. FCFS

0 14 37 53 65 67 98 122 124 183 199

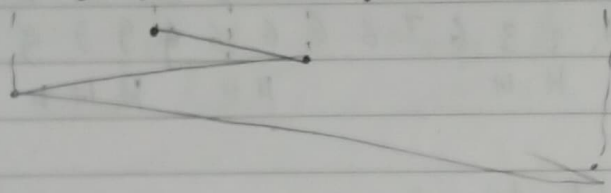


Total seek time = 45 + 85 + 146 + 85 + 108 + 110 + 59 + 2

= 640

SSTF:

0 14 37 53 65 67 98 122 124 183 199

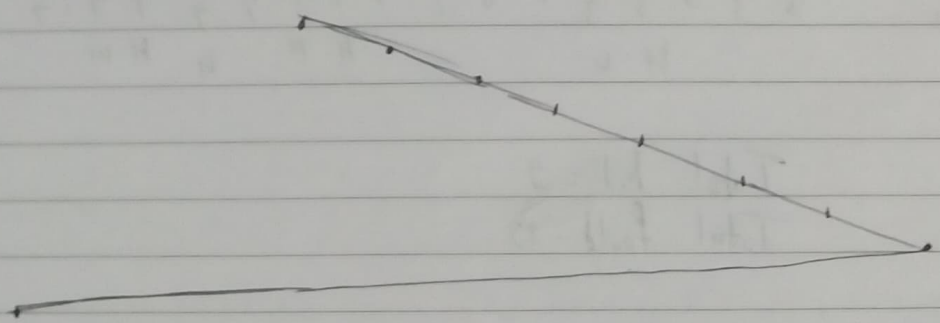


Total seek time = 12 + 2 + 30 + 23 + 84 + 24 + 2 + 59

= 220

SCAN:

0 14 37 53 65 67 98 122 124 183 199



Total seek time = 12 + 2 + 31 + 24 + 2 + 59 + 16 + 162 + 23

= 331