Experiment-1

Aim: Study and implement basic linux commands like ,MKDIR, CHDID, CAT, LS, CHOWN, CHMOD, CHGRP .

Output:

```
ictc@ictc-virtual-machine:~$ cd Destop
bash: cd: Destop: No such file or directory
ictc@ictc-virtual-machine:~$ cd Desktop
ictc@ictc-virtual-machine:~/Desktop$ mkdir harshit
ictc@ictc-virtual-machine:~/Desktop$ cd harshit
ictc@ictc-virtual-machine:~/Desktop/harshit$ touch sample.sh
ictc@ictc-virtual-machine:~/Desktop/harshit$ chmod +x sample.sh
ictc@ictc-virtual-machine:~/Desktop/harshit$ ./sample.sh
./sample.sh: line 1: unexpected EOF while looking for matching `"'
./sample.sh: line 3: syntax error: unexpected end of file
ictc@ictc-virtual-machine:~/Desktop/harshit$ ./sample.sh
harshit
ictc@ictc-virtual-machine:~/Desktop/harshit$ touch version.sh
ictc@ictc-virtual-machine:~/Desktop/harshit$ chmod +x version.sh
ictc@ictc-virtual-machine:~/Desktop/harshit$ ./version.sh
OS Name is
Linux
Release version name
Linux ictc-virtual-machine 5.15.0-131-generic #141~20.04.1-Ubuntu SMP Thu Jan 16 18:38:51 UTC 2025 x86_64 x86_64 x86_64 GNU/Linux
Kernal version name
5.15.0-131-generic
```

Conclusion:

In conclusion, mastering basic Linux commands such as mkdir, cd, cat, ls, chown, chmod, and chgrp is fundamental for efficiently managing files, directories, and system resources in a Linux environment. These commands enable users to create and navigate directories, view and manipulate file contents, and control access through file ownership and permissions. Whether organizing data, securing files, or configuring system settings, these commands provide the essential tools for system administration and day-to-day Linux operations, ensuring that users can maintain a structured, secure, and well-functioning environment.

Experiment-2

Aim: Study and implement basic Linux command like MV and CP.

Output:

```
ictc@ubuntu:~/Desktop/e/subject$ cd dbms
ictc@ubuntu:~/Desktop/e/subject/dbms$ touch marks.txt
ictc@ubuntu:~/Desktop/e/subject/dbms$ touch result.txt
ictc@ubuntu:~/Desktop/e/subject/dbms$ ls
marks.txt  result.txt
ictc@ubuntu:~/Desktop/e/subject/dbms$ mv marks.txt  result.txt
ictc@ubuntu:~/Desktop/e/subject/dbms$ ls
result.txt
ictc@ubuntu:~/Desktop/e/subject/dbms$ cp result.txt maths
ictc@ubuntu:~/Desktop/e/subject/dbms$ cd ..
ictc@ubuntu:~/Desktop/e/subject$ ls
dbms  e1.txt  maths  os
ictc@ubuntu:~/Desktop/e/subject$ cd maths
ictc@ubuntu:~/Desktop/e/subject/maths$ ls
ictc@ubuntu:~/Desktop/e/subject/maths$ cd ..
ictc@ubuntu:~/Desktop/e/subject$ cd dbms
ictc@ubuntu:~/Desktop/e/subject/dbms$ ls
maths  result.txt
ictc@ubuntu:~/Desktop/e/subject/dbms$ 
```

Conclusion:

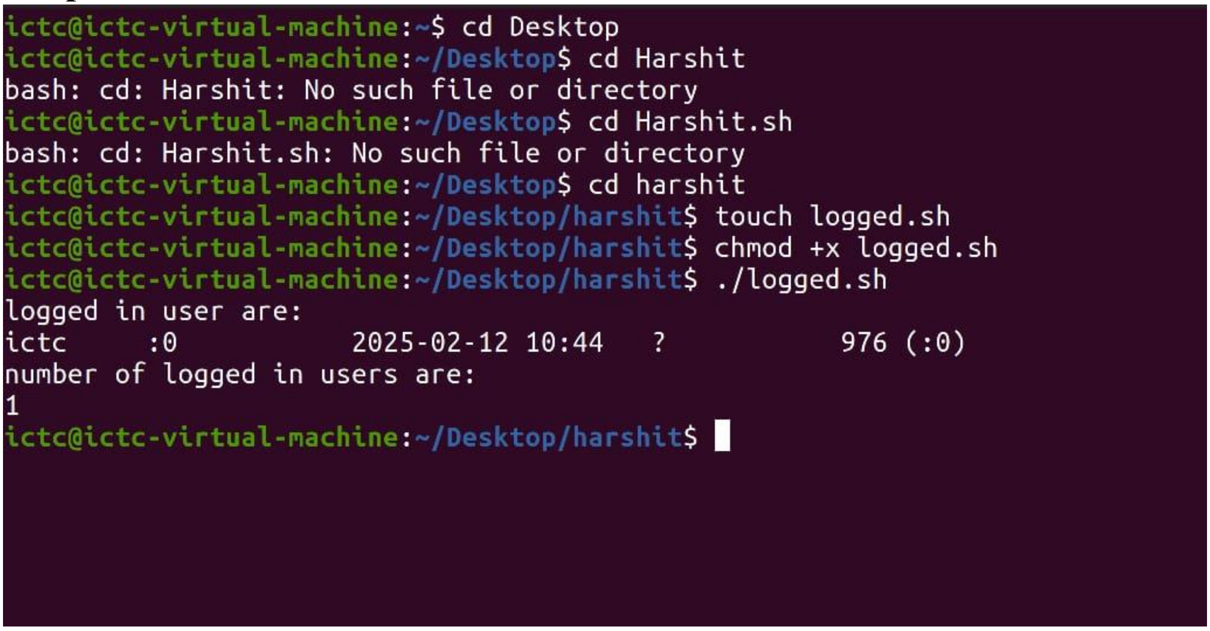
In this exercise, we studied and implemented two essential Linux commands: mv (move) and cp (copy). Both commands are fundamental tools for file and directory management in Linux. The mv command is primarily used for moving and renaming files or directories, allowing users to efficiently organize and restructure their filesystem. On the other hand, the cp command is crucial for copying files and directories, ensuring that data can be duplicated while preserving the original files. By understanding and using these commands, users gain the ability to manipulate files with flexibility and control, which is a core aspect of effective system administration. Mastering mv and cp enables users to streamline workflows, manage data, and safeguard important information through duplication or careful organization.

Experiment-3

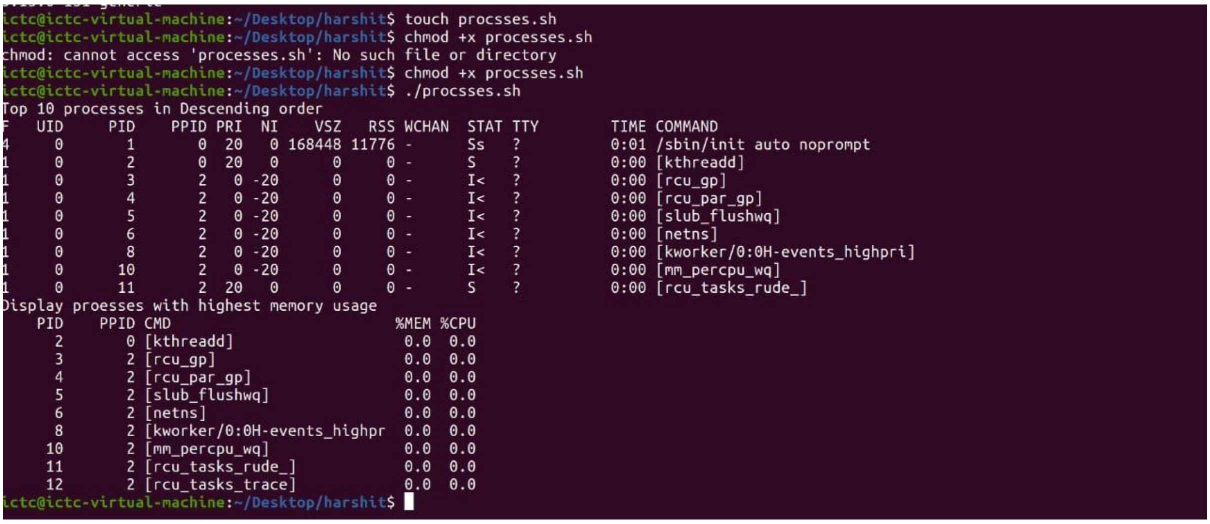
Aim: To study and implement the shell scripts.

1- Display OS version, Release version , Kernel version.
2- Display 10 prosses in descending order.
3- Display process with highest memory .
4- Display current logged in used and tag name.

Output:



```
ictc@ictc-virtual-machine:~$ cd Desktop
ictc@ictc-virtual-machine:~/Desktop$ cd Harshit
bash: cd: Harshit: No such file or directory
ictc@ictc-virtual-machine:~/Desktop$ cd Harshit.sh
bash: cd: Harshit.sh: No such file or directory
ictc@ictc-virtual-machine:~/Desktop$ cd harshit
ictc@ictc-virtual-machine:~/Desktop/harshit$ touch logged.sh
ictc@ictc-virtual-machine:~/Desktop/harshit$ chmod +x logged.sh
ictc@ictc-virtual-machine:~/Desktop/harshit$ ./logged.sh
logged in user are:
ictc     :0           2025-02-12 10:44   ?           976 (:0)
number of logged in users are:
1
ictc@ictc-virtual-machine:~/Desktop/harshit$
```



```
ictc@ictc-virtual-machine:~/Desktop/harshit$ touch procsses.sh
ictc@ictc-virtual-machine:~/Desktop/harshit$ chmod +x processes.sh
chmod: cannot access 'processes.sh': No such file or directory
ictc@ictc-virtual-machine:~/Desktop/harshit$ chmod +x procsses.sh
ictc@ictc-virtual-machine:~/Desktop/harshit$ ./procsses.sh
Top 10 processes in Descending order
F   UID     PID    PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
4     0       1       0  20   0 168448 11776 -      Ss   ?          0:01 /sbin/init auto noprompt
1     0       2       0  20   0      0     0 -      S    ?          0:00 [kthreadd]
1     0       3       2   0 -20      0     0 -      I<   ?          0:00 [rcu_gp]
1     0       4       2   0 -20      0     0 -      I<   ?          0:00 [rcu_par_gp]
1     0       5       2   0 -20      0     0 -      I<   ?          0:00 [slub_flushwq]
1     0       6       2   0 -20      0     0 -      I<   ?          0:00 [netns]
1     0       8       2   0 -20      0     0 -      I<   ?          0:00 [kworker/0:0H-events_highpri]
1     0      10       2   0 -20      0     0 -      I<   ?          0:00 [mm_percpu_wq]
1     0      11       2  20   0      0     0 -      S    ?          0:00 [rcu_tasks_rude_]
Display proesses with highest memory usage
    PID    PPID CMD                         %MEM %CPU
      2       0 [kthreadd]                   0.0  0.0
      3       2 [rcu_gp]                     0.0  0.0
      4       2 [rcu_par_gp]                 0.0  0.0
      5       2 [slub_flushwq]               0.0  0.0
      6       2 [netns]                      0.0  0.0
      8       2 [kworker/0:0H-events_highpr  0.0  0.0
     10       2 [mm_percpu_wq]               0.0  0.0
     11       2 [rcu_tasks_rude_]            0.0  0.0
     12       2 [rcu_tasks_trace]            0.0  0.0
ictc@ictc-virtual-machine:~/Desktop/harshit$
```

Conclusion: These scripts provide useful information about the system and running processes. By displaying the OS version, release version, and kernel version, we gain insight into the system's architecture. Listing the top 10 processes by memory usage allows for monitoring resource consumption, while identifying the process using the most memory helps in troubleshooting performance issues. Lastly, knowing the current user and terminal tag helps in identifying the active session and ensuring correct user-based operations. These commands are essential for system administrators and developers to maintain system health and diagnose issues effectively.

Experiment-4

Aim: Create a child process in Linux using the Fork system call. From the child process obtain the process ID of both child and parent by using get pid and get ppid system call.

Code:

```
#include <stdio.h>
#include <unistd.h>
int main() {
int pid;
pid = fork();
if (pid < 0) {
printf("Fork failed\n");
} else if (pid == 0) {
printf("Child process: PID = %d, Parent PID = %d\n", getpid(), getppid());
} else {
printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);
}return 0;
}
```

Output:

Conclusion:

The fork() system call is a fundamental concept in Unix-based systems, allowing a process to create a child process. Using getpid() and getppid() within the child process allows us to track and verify the relationship between the parent and child processes. This approach is essential for process management, inter-process communication, and building multi-process applications in Linux. It also serves as a key building block in understanding process control in operating systems.

Experiment-5

Aim: Write a program to demonstrate the concept of non-preemptive scheduling alogrithems,

1-FCFS [first come first serve].

2-SJF[shortest job first]

Code:

```
#include <stdio.h>
int main()
{
int n, bt[20], wt[20], tat[20], avwt = 0, avtat = 0, i, j;
printf("Enter total number of processes (maximum 20): ");
scanf("%d", &n);
printf("Enter Process Burst Time\n");
for(i = 0; i < n; i++)
{
printf("P[%d]: ", i + 1);
scanf("%d", &bt[i]);
}
wt[0] = 0; // Waiting time for the first process is 0
// Calculate waiting time
for(i = 1; i < n; i++)
{
wt[i] = 0;
for(j = 0; j < i; j++)
        wt[i] += bt[j];
}
// Calculate turnaround time
printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");
for(i = 0; i < n; i++)
{
```

```c
tat[i] = bt[i] + wt[i];
avwt += wt[i];
avtat += tat[i];
printf("\nP[%d]\t\t%d\t\t%d\t\t%d", i + 1, bt[i], wt[i], tat[i]);
}

avwt /= n;
avtat /= n;
printf("\n\nAverage Waiting Time: %d", avwt);
printf("\nAverage Turnaround Time: %d", avtat);

return 0;
}
```

Output:

```
Enter process Bust time
P[1]:2
P[2]:3
P[3]:4
P[4]:5
P[5]:6
P[6]:7

 Process                Burst Time       Waiting Time     Turnaround Time
P[1]            2               0               2
P[2]            3               27              30
P[3]            4               27              31
P[4]            5               27              32
P[5]            6               27              33
P[6]            7               27              34

 Average Waitng Time:22
 Average Turnaround Time:27ictc@ubuntu:~/Desktop$
```

Code:

```c
#include <stdio.h>
void main()
{
    int bt[20], p[20], wt[20], tat[20], i, j, n, total = 0, pos, temp;
```

```c
    float avg_wt, avg_tat;


    printf("Enter number of process: ");
    scanf("%d", &n);
  printf("\nEnter Burst Time:\n");
    for (i = 0; i < n; i++)
    {
        printf("p%d: ", i + 1);
        scanf("%d", &bt[i]);
        p[i] = i + 1;  // Contains process number
    }
  // Sorting burst time in ascending order using selection sort
    for (i = 0; i < n; i++)
    {
        pos = i;
        for (j = i + 1; j < n; j++)
        {
            if (bt[j] < bt[pos])
                pos = j;
        }
         temp = bt[i];
        bt[i] = bt[pos];
        bt[pos] = temp;


        temp = p[i];
        p[i] = p[pos];
        p[pos] = temp;
    }
   wt[0] = 0;  // Waiting time for first process will be zero
   // Calculate waiting time
```

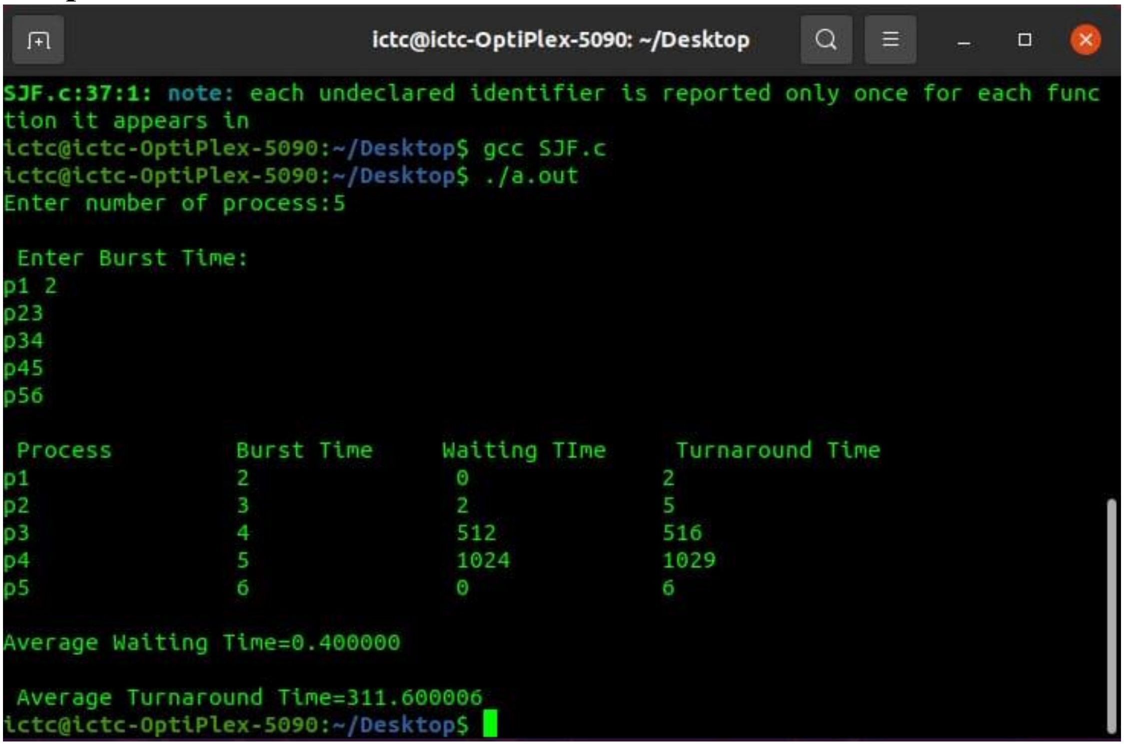
```
    for (i = 1; i < n; i++)
    wt[i] = 0;
        for (j = 0; j < i; j++)
            wt[i] += bt[j];

        total += wt[i];
    }
avg_wt = (float)total / n;  // Average waiting time
    total = 0;

    printf("\nProcess\t  Burst Time  \tWaiting Time\tTurnaround Time");
    for (i = 0; i < n; i++)
    {
        tat[i] = bt[i] + wt[i];  // Calculate turnaround time
        total += tat[i];
        printf("\n p%d\t\t %d\t\t %d\t\t%d", p[i], bt[i], wt[i], tat[i]);
    }

    avg_tat = (float)total / n;  // Average turnaround time
    printf("\n\nAverage Waiting Time = %.2f", avg_wt);
    printf("\nAverage Turnaround Time = %.2f\n", avg_tat);
}
```

Output:



Conclusion: The **FCFS (First Come First Serve)** and **SJF (Shortest Job First)** are two classic non-preemptive scheduling algorithms used in process management. FCFS executes processes in the order of their arrival, making it simple but prone to inefficiency, especially when long processes precede shorter ones, leading to high average waiting times. On the other hand, SJF improves performance by prioritizing processes with the shortest burst time, reducing waiting times overall, but it requires knowledge of burst times in advance and can lead to starvation for longer processes. Both algorithms illustrate the trade-off between simplicity and efficiency, with FCFS being easy to implement but less efficient, while SJF offers better performance but has its own limitations.

Experiment-6

Aim: Write a program to demonstrate the concept of preemptive scheduling alogrethims.For round robbin.

Code:

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

struct Process {
    int id;
    int bst_time;
    int wt_time;
}
int main() {
    struct Process Proc_arr[20];
    float avg_wait_time = 0, total_turnaround_time = 0;
    int maxproc, time_qtm, time = 0, i, j, k = 0;

    printf("\n# ENTER HOW MANY PROCESSES ARE THERE: ");
    scanf("%d", &maxproc);

    for (i = 0; i < maxproc; i++) {
        printf("\n# ENTER THE BURST TIME FOR PROCESS P%d: ", i + 1);
        scanf("%d", &Proc_arr[i].bst_time);
        total_turnaround_time += Proc_arr[i].bst_time;
        Proc_arr[i].id = i + 1;
        Proc_arr[i].wt_time = 0;
    }

    printf("\n# ENTER THE TIME QUANTUM: ");
    scanf("%d", &time_qtm);
```

```
system("clear");  // Clears the console (for Linux/macOS)

printf("\n# TURNAROUND TIME: %.2f\n", total_turnaround_time);

i = 0;
k = 0;
time = 0;

while (k < maxproc) {
    if (Proc_arr[i].bst_time > 0) {
        if (Proc_arr[i].bst_time <= time_qtm) {
            time += Proc_arr[i].bst_time;
            Proc_arr[i].wt_time += (time - Proc_arr[i].bst_time);
            Proc_arr[i].bst_time = 0;
            k++;  // Process completed
        } else {
            time += time_qtm;
            Proc_arr[i].bst_time -= time_qtm;
            for (j = 0; j < maxproc; j++) {
                if (j != i && Proc_arr[j].bst_time > 0) {
                    Proc_arr[j].wt_time += time_qtm;
                }
            }
        }
    }
    i = (i + 1) % maxproc;  // Move to the next process in circular order
}
printf("\n\nPROCESS\tWAITING TIME\n");
for (i = 0; i < maxproc; i++)
```

```
        printf("P%d\t\t%d\n", Proc_arr[i].id, Proc_arr[i].wt_time);
       avg_wait_time += Proc_arr[i].wt_time;
       total_turnaround_time += Proc_arr[i].wt_time;
    }
    avg_wait_time /= maxproc;
   total_turnaround_time /= maxproc;
    printf("\n# AVERAGE WAITING TIME: %.2f", avg_wait_time);
   printf("\n# AVERAGE TURNAROUND TIME: %.2f\n", total_turnaround_time);
return 0;
}
```

Output:

```
ictc@ubuntu:~$ cd Desktop
ictc@ubuntu:~/Desktop$ gcc RR.c
ictc@ubuntu:~/Desktop$ ./a.out
Total number of processes in the system: 5

Enter the Arrival and Burst time of Process[1]
Arrival time: 0
Burst time: 5

Enter the Arrival and Burst time of Process[2]
Arrival time: 1
Burst time: 10

Enter the Arrival and Burst time of Process[3]
Arrival time: 2
Burst time: 15

Enter the Arrival and Burst time of Process[4]
Arrival time: 3
Burst time: 30

Enter the Arrival and Burst time of Process[5]
Arrival time: 4
Burst time: 40
Enter the Time Quantum: 5

Process No      Burst Time      TAT             Waiting Time

Process No[1]   5               5               0
Process No[2]   10              29              19
Process No[3]   15              48              33
Process No[4]   30              82              52
Process No[5]   40              96              56
Average Turnaround Time: 52.000000
Average Waiting Time: 32.000000ictc@ubuntu:~/Desktop$
```

Conclusion: The **Round Robin (RR)** scheduling algorithm is a preemptive algorithm designed to give each process an equal share of CPU time in a cyclic order, making it ideal for time-sharing systems. It ensures fairness by giving every process a fixed time slice (quantum), which helps avoid the starvation issue that can occur in other non-preemptive algorithms. However, the choice of time quantum is critical: a too-small quantum leads to excessive context switching, while a too-large quantum behaves similarly to **FCFS**. Although RR improves process response time, it may not always be the most efficient in terms of average waiting and turnaround times, particularly if processes have very different burst times.