PREMIER UNIVERSITY CHATTOGRAM

DEPARTMENT OF **COMPUTER SCIENCE AND ENGINEERING**

LAB REPORT

COURSE NAME	Operating Systems Lab
COURSE CODE	CSE 3734
REPORTS ON	1.Unix File Manipulation Commands. 2.Unix Filtering Commands. 3.Shell Program:Leap Year Check 4.Shell Program:Factorial Calculation 5.CPU Scheduling Algorithms:FCFS.SJF,Priority,Round Robin
DATE OF REPORT	7-02-2025

SUBMITTED TO

Shatabdi Roy Moon

Lecturer

Department of Computer Science and Engineering

REMARKS	**SUBMITTED BY**			
	NAME	Rimjhim Dey		
	ID	0222220005101039		
	SEMESTER	5th		
	BATCH	42nd		
	SESSION	Fall 2024	**SECTION**	A

Experiment No: 01

Experiment Name: BASICS OF UNIX COMMANDS – File Manipulation

Objective:

The objective of this experiment is to understand and perform basic file manipulation operations in a UNIX environment. This includes creating, displaying, copying, moving, renaming, and deleting files and directories.

Introduction:

File manipulation is a fundamental operation in any operating system. In UNIX, file manipulation commands allow users to create, view, modify, copy, move, and delete files and directories efficiently. These operations are essential for organizing and managing data within a file system. This experiment introduces basic UNIX commands that enable users to perform these file management tasks.

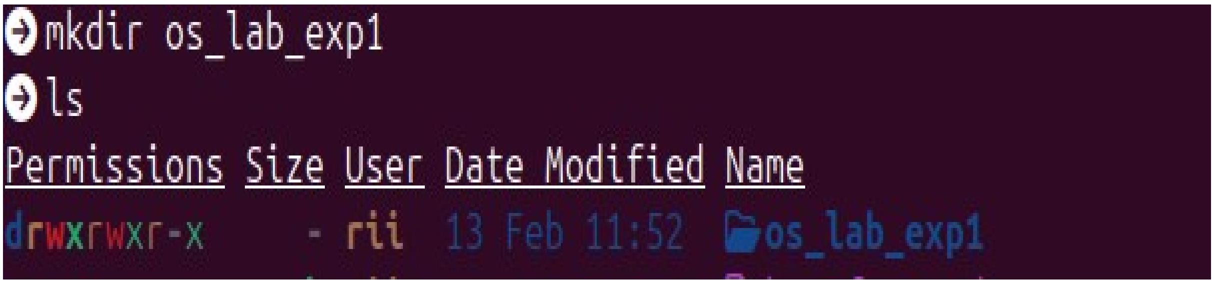
Code:

The following UNIX commands are used for basic file manipulation:

1. Create a Directory:

mkdir os_lab_exp1

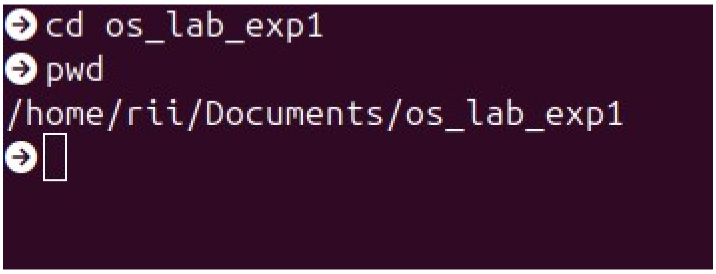
Output:



2. Navigate into the Directory:

```
cd os_lab_exp1
```

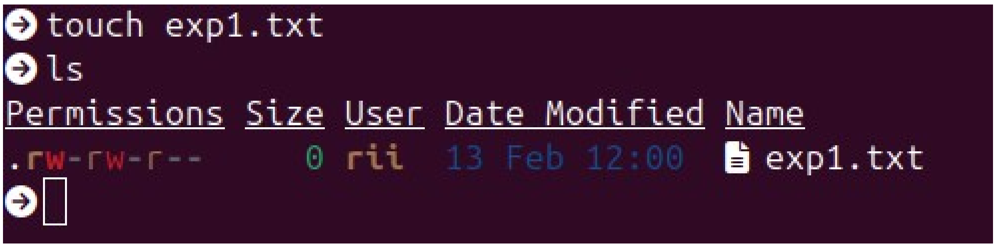
Output:



3. Create a file:

touch exp1.txt

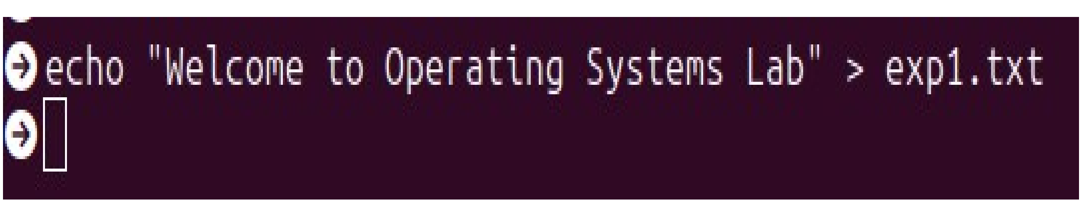
Output:



4. Write content to a file:

echo "Welcome to Operating Systems Lab" > exp1.txt

Output:



5. Display content of a file:

cat exp1.txt

Output:

```
cat exp1.txt
Welcome to Operating Systems Lab
```

6. Copy a file:

`cp exp1.txt copy_exp1.txt`

Output:

```
cp exp1.txt copy_exp1.txt
ls
Permissions Size User Date Modified Name
.rw-rw-r--    33 rii  13 Feb 12:16  copy_exp1.txt
.rw-rw-r--    33 rii  13 Feb 12:04  exp1.txt
```

7. Rename a file:

mv exp1.txt test_exp1.txt

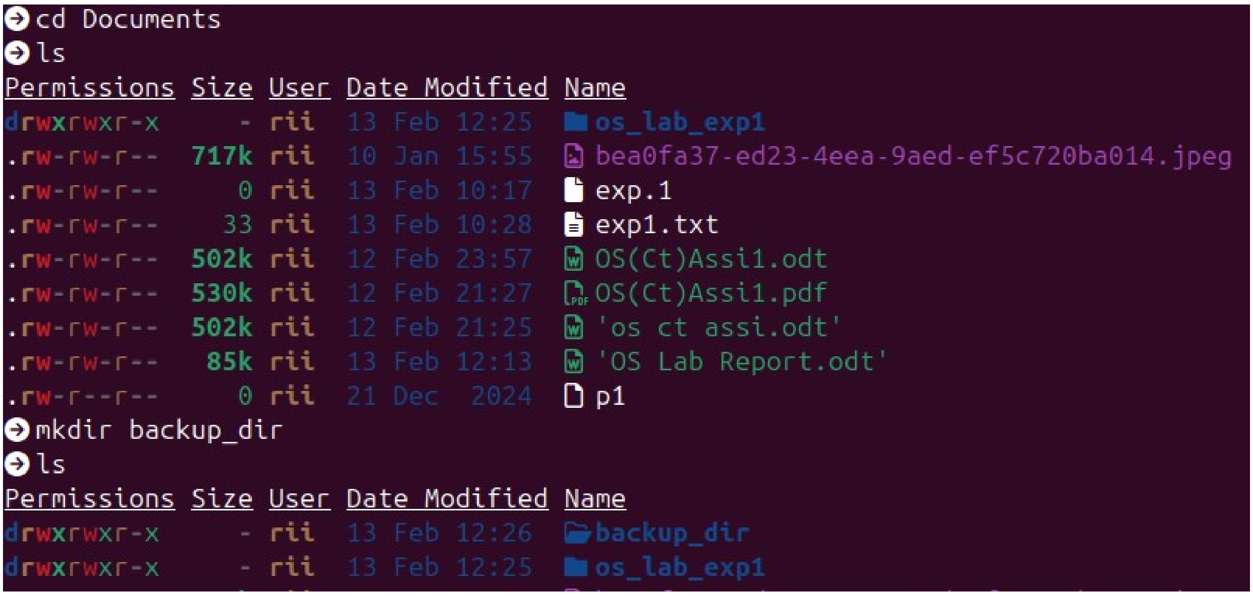
Output:

```
mv exp1.txt test_exp1.txt
ls
Permissions Size User Date Modified Name
.rw-rw-r--    33 rii  13 Feb 12:16  copy_exp1.txt
.rw-rw-r--    33 rii  13 Feb 12:04  test_exp1.txt
```

8. Create Another Directory

mkdir backup_dir

Output:



9.Move the 1st Created File to this Directory

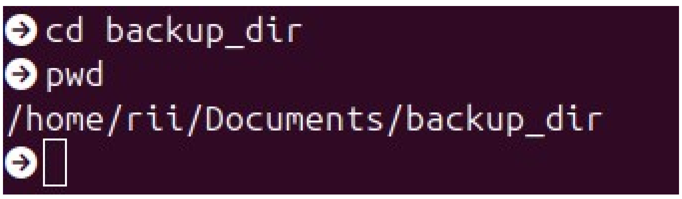
mv test_exp1.txt backup_dir/

Output:

10. Navigate into Directory:

cd backup_dir

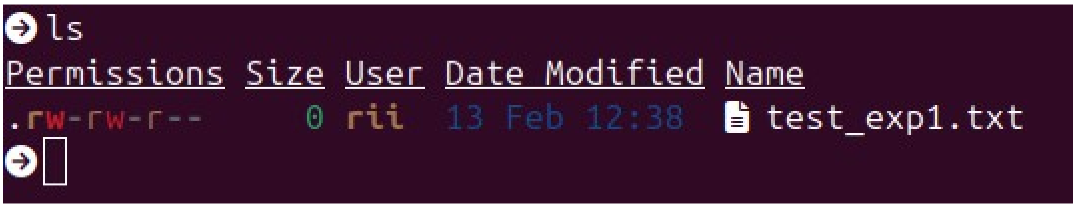
Output:



11. Display Files:

ls

Output:

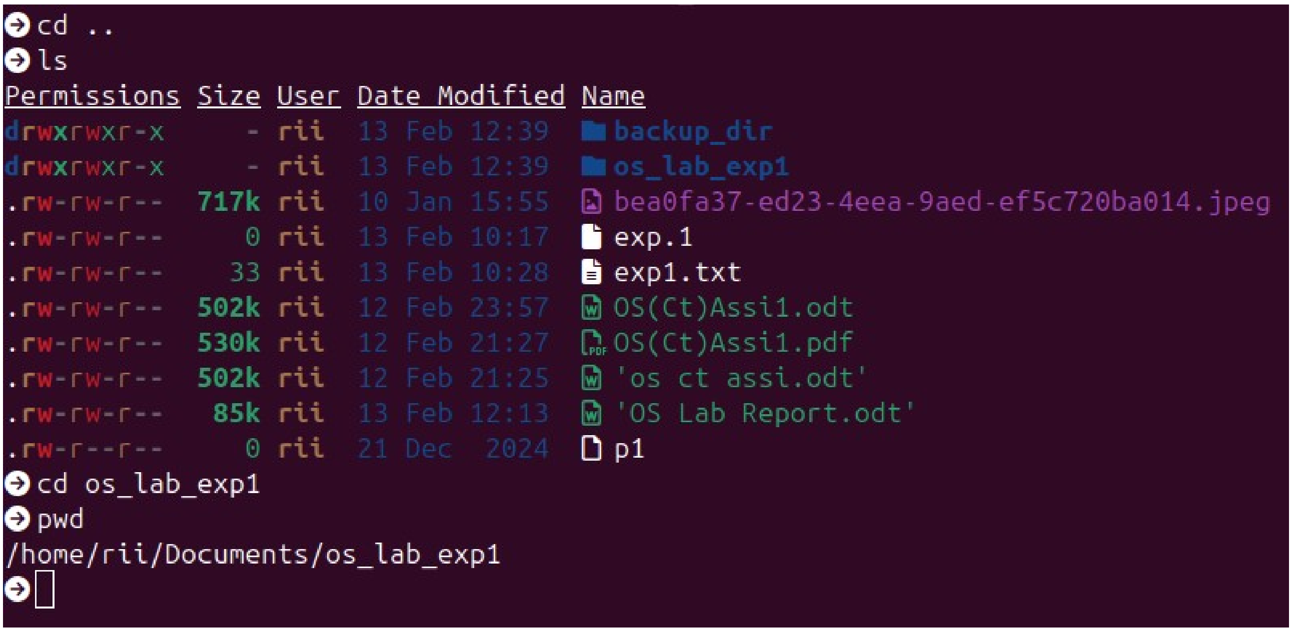


```
ls
Permissions Size User Date Modified Name
.rw-rw-r--     0 rii  13 Feb 12:38  test_exp1.txt
```

12. Navigate Back to Original Directory:

```
cd ..
```

Output:

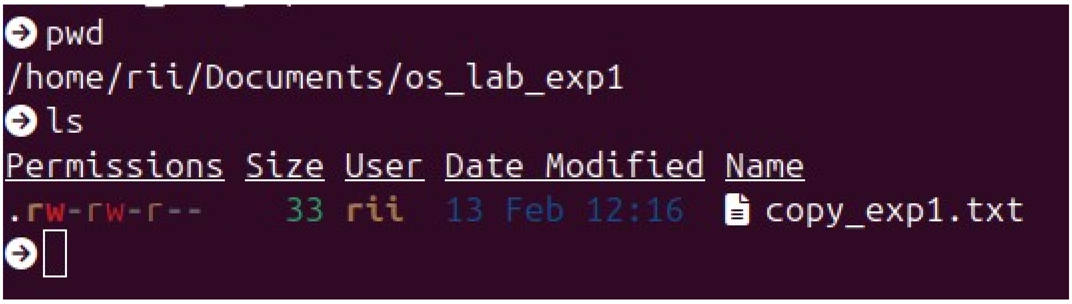


```
cd ..
ls
Permissions Size User Date Modified Name
drwxrwxr-x     - rii  13 Feb 12:39  backup_dir
drwxrwxr-x     - rii  13 Feb 12:39  os_lab_exp1
.rw-rw-r--  717k rii  10 Jan 15:55  bea0fa37-ed23-4eea-9aed-ef5c720ba014.jpeg
.rw-rw-r--     0 rii  13 Feb 10:17  exp.1
.rw-rw-r--    33 rii  13 Feb 10:28  exp1.txt
.rw-rw-r--  502k rii  12 Feb 23:57  OS(Ct)Assi1.odt
.rw-rw-r--  530k rii  12 Feb 21:27  OS(Ct)Assi1.pdf
.rw-rw-r--  502k rii  12 Feb 21:25  'os ct assi.odt'
.rw-rw-r--   85k rii  13 Feb 12:13  'OS Lab Report.odt'
.rw-r--r--     0 rii  21 Dec  2024  p1
cd os_lab_exp1
pwd
/home/rii/Documents/os_lab_exp1
```

13. Display all the Files in this Directory:

```
ls
```

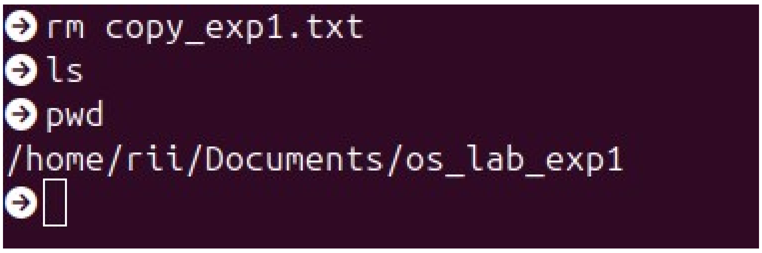
Output:



14.Delete a file (copy_exp1.txt):

```
rm copy_exp1.txt
```

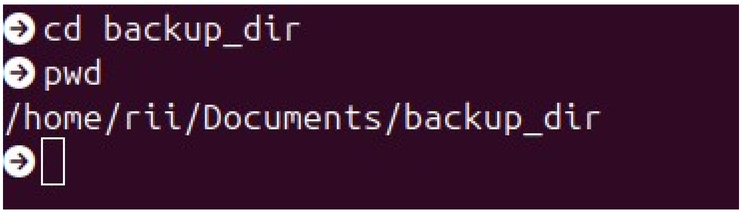
Output:



15.Navigate to Backup Directory(backup_dir):

cd backup_dir

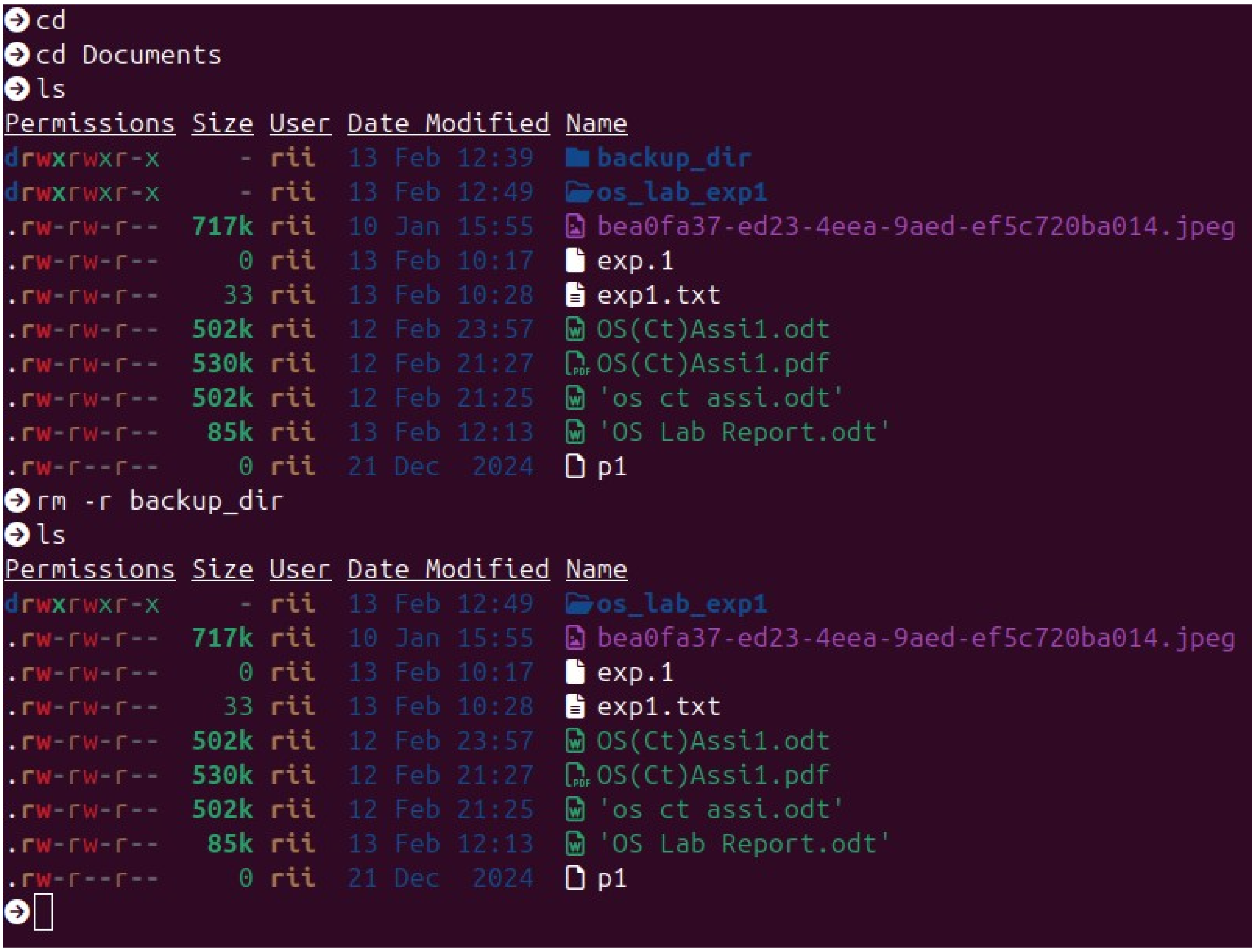
Output:



16.Delete `backup_dir` Directory and its Contents:

rm -r backup_dir

Output:

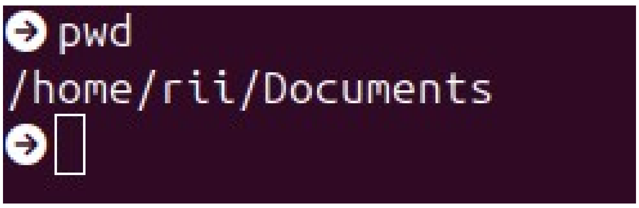


```
cd
cd Documents
ls
Permissions Size User Date Modified Name
drwxrwxr-x     - rii  13 Feb 12:39  backup_dir
drwxrwxr-x     - rii  13 Feb 12:49  os_lab_exp1
.rw-rw-r--  717k rii  10 Jan 15:55  bea0fa37-ed23-4eea-9aed-ef5c720ba014.jpeg
.rw-rw-r--     0 rii  13 Feb 10:17  exp.1
.rw-rw-r--    33 rii  13 Feb 10:28  exp1.txt
.rw-rw-r--  502k rii  12 Feb 23:57  OS(Ct)Assi1.odt
.rw-rw-r--  530k rii  12 Feb 21:27  OS(Ct)Assi1.pdf
.rw-rw-r--  502k rii  12 Feb 21:25  'os ct assi.odt'
.rw-rw-r--   85k rii  13 Feb 12:13  'OS Lab Report.odt'
.rw-r--r--     0 rii  21 Dec  2024  p1
rm -r backup_dir
ls
Permissions Size User Date Modified Name
drwxrwxr-x     - rii  13 Feb 12:49  os_lab_exp1
.rw-rw-r--  717k rii  10 Jan 15:55  bea0fa37-ed23-4eea-9aed-ef5c720ba014.jpeg
.rw-rw-r--     0 rii  13 Feb 10:17  exp.1
.rw-rw-r--    33 rii  13 Feb 10:28  exp1.txt
.rw-rw-r--  502k rii  12 Feb 23:57  OS(Ct)Assi1.odt
.rw-rw-r--  530k rii  12 Feb 21:27  OS(Ct)Assi1.pdf
.rw-rw-r--  502k rii  12 Feb 21:25  'os ct assi.odt'
.rw-rw-r--   85k rii  13 Feb 12:13  'OS Lab Report.odt'
.rw-r--r--     0 rii  21 Dec  2024  p1
```

17.Show current directory path:

pwd

Output:



```
pwd
/home/rii/Documents
```

Discussion:

In this experiment, we implemented basic UNIX commands to perform file and directory manipulation tasks. We began by creating a directory (`os_lab_exp1`) and navigating into it. Inside the directory, we created a file (`exp1.txt`), wrote content to it using the `echo` command, and displayed its contents with `cat`. We then performed file operations such as copying (`cp`), renaming (`mv`), and moving (`mv` again) files between directories. After that, we created another directory (`backup_dir`) and moved the file into it to demonstrate file organization. We used the `ls` command to list files and confirm their presence in the respective directories. We also demonstrated file deletion using the `rm` command and removed a directory along with its contents using `rm -r`. Lastly, we used the `pwd` command to check our current directory. These operations highlighted the fundamental file manipulation commands in UNIX, essential for managing files and directories efficiently in a UNIX environment.

Experiment No: 02

Experiment Name: Basics of UNIX Commands – Filtering

Objective:

The objective of this experiment is to learn and implement basic UNIX commands related to filtering data from files or command outputs using tools such as `grep`, `sort`, `uniq`, and `wc`. These commands allow users to process and manipulate text data efficiently in a UNIX environment.

Introduction:

In UNIX, filtering commands are used to process text or command output by extracting, sorting, or counting specific patterns. Filtering is a powerful technique in UNIX that allows users to quickly search for data, eliminate duplicates, and organize or count items based on certain criteria. In this experiment, we will use several filtering tools such as `grep`, `sort`, `uniq`, and `wc` to demonstrate how they can be applied to text data and command output.

Code:

1.Using “`grep`” to Search for a Pattern in a File:

The `grep` command is used to search for specific patterns in a file. It outputs lines that match the pattern specified.

grep "commands" exp2.txt

Output:

```
➜ grep "commands" exp2.txt
This is Experiment 2 on UNIX commands.
Learn filtering commands like grep, sort, uniq, and wc.
We are learning about UNIX commands.
These commands are very useful for text processing.
➜
```

```
➜ cd
➜ ls
Permissions Size User Date Modified Name
drwxrwxr-x     - rii  16 Dec  2024  Android
drwxrwxr-x     - rii  10 Feb 20:48  AndroidStudioProjects
drwxr-xr-x     - rii  10 Dec  2024  Desktop
drwxr-xr-x     - rii  13 Feb 12:53  Documents
drwxr-xr-x     - rii  10 Feb 20:56  Downloads
drwxrwxr-x     - rii  13 Feb 13:18  labexp2
drwxr-xr-x     - rii  10 Dec  2024  Music
drwxr-xr-x     - rii  13 Feb 10:19  Pictures
drwxrwxr-x     - rii  15 Dec  2024  Programming
drwxr-xr-x     - rii  10 Dec  2024  Public
drwx------     - rii  16 Dec  2024  snap
drwxrwxr-x     - rii  29 Jan 20:50  StudioProjects
drwxr-xr-x     - rii  10 Dec  2024  Templates
drwxr-xr-x     - rii  12 Dec  2024  Videos
➜ cd labexp2
➜ touch exp2.txt
➜ ls
Permissions Size User Date Modified Name
.rw-rw-r--     0 rii  13 Feb 13:21  exp2.txt
➜ echo -e "Welcome to Operating Systems Lab\nThis is Experiment 2 on UNIX commands.\nLearn filtering commands like grep, sort, uniq, and wc.\ngrep command is used for searching patterns.\nWe
 are learning about UNIX commands.\nThese commands are very useful for text processing." > exp2.txt
➜ cat exp2.txt
Welcome to Operating Systems Lab
This is Experiment 2 on UNIX commands.
Learn filtering commands like grep, sort, uniq, and wc.
grep command is used for searching patterns.
We are learning about UNIX commands.
These commands are very useful for text processing.
➜ grep "commands" exp2.txt
This is Experiment 2 on UNIX commands.
Learn filtering commands like grep, sort, uniq, and wc.
We are learning about UNIX commands.
These commands are very useful for text processing.
➜
```

2.Using “`sort`” to Sort Lines in a File:

The `sort` command is used to sort the contents of a file or command output in ascending or descending order.

sort exp2.txt

Output:

```
➔ sort exp2.txt
grep command is used for searching patterns.
Learn filtering commands like grep, sort, uniq, and wc.
These commands are very useful for text processing.
This is Experiment 2 on UNIX commands.
We are learning about UNIX commands.
Welcome to Operating Systems Lab
➔
```

3.Using “`uniq`” to Remove Duplicate Lines:

The `uniq` command removes duplicate lines from a file or command output. It is typically used in conjunction with the `sort` command.

sort exp2.txt | uniq

Output:

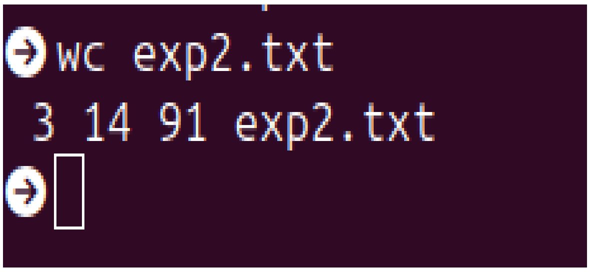
```
➔ echo -e " Welcome to Operating Systems Lab.\nWelcome to Operating Systems Lab.\nThis is Experiment 2." > exp2.txt
➔ cat exp2.txt
 Welcome to Operating Systems Lab.
Welcome to Operating Systems Lab.
This is Experiment 2.
➔ sort exp2.txt | uniq
This is Experiment 2.
 Welcome to Operating Systems Lab.
Welcome to Operating Systems Lab.
```

4.Using `wc` to Count Lines, Words, and Characters:

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

wc exp2.txt

Output:



Discussion:

In this experiment, we implemented basic UNIX filtering commands like `grep`, `sort`, and `uniq`, which are essential for processing text files efficiently. First, we prepared a text file containing multiple lines of data, including some duplicate lines, to demonstrate the functionality of these commands. The `grep` command was used to search for specific patterns in the file, helping us quickly locate lines containing the desired text. The `sort` command arranged the lines in alphabetical order, making it easier to organize and analyze data. Finally, the `uniq` command removed adjacent duplicate lines, ensuring the uniqueness of the content. Together, these filtering commands showcased how UNIX tools simplify text processing tasks, making them highly useful for handling large amounts of textual data.

Experiment No: 3

Experiment Name: SIMPLE SHELL PROGRAMS - A Shell program to check the given year is leap year or not.

Objective:

- To write a shell program that determines whether a given year is a leap year or not.
- To understand the use of conditional statements in shell programming.
- To gain familiarity with basic shell scripting syntax and execution.

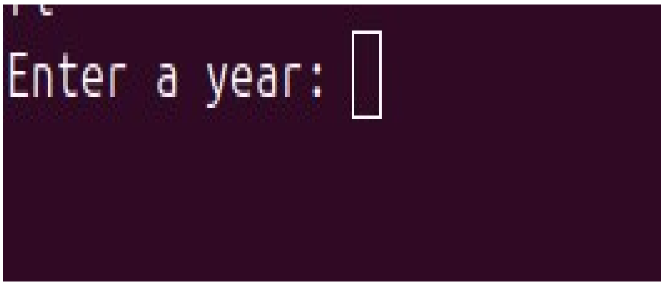
Introduction:

A leap year is a year that is divisible by 4, but century years are only leap years if they are divisible by 400. This concept is crucial for accurate date calculations in software development. Shell scripting allows automating tasks and performing logical operations efficiently using conditional statements.

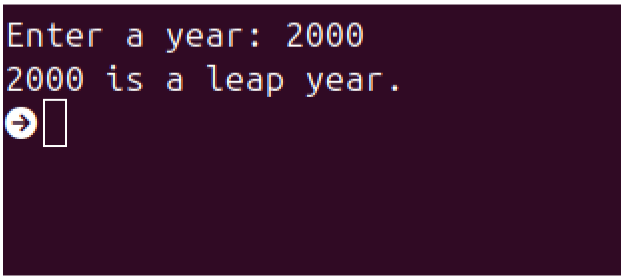
Code:

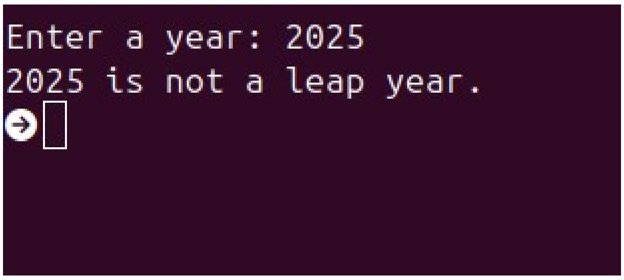
```
# Read year input from user
echo -n "Enter a year: "
read year
# Check if the year is a leap year
if [ $((year % 4)) -eq 0 ]; then
if [ $((year % 100)) -ne 0 ] || [ $((year % 400)) -eq 0 ]; then
echo "$year is a leap year."
else
echo "$year is not a leap year."
fi
else
echo "$year is not a leap year."
fi
```

Input:



Output:





Discussion:

In this experiment, a shell script was developed to determine whether a given year is a leap year. The program uses nested conditional statements and arithmetic operations to check the divisibility rules for leap years. If a year is divisible by 4 and not divisible by 100, or divisible by 400, it is classified as a leap year. User input is obtained using the `read` command, and modular arithmetic is performed to check the divisibility conditions. This program demonstrates the basic structure and control flow mechanisms in shell scripting, emphasizing the importance of logical evaluation in decision-making processes.

Experiment No: 4

Experiment Name: SIMPLE SHELL PROGRAMS – A Shell program to find the factorial of a number

Objective:

- To write a shell program to calculate the factorial of a given number.
- To understand the use of loops in shell scripting.
- To gain familiarity with arithmetic operations and control structures in shell programming.

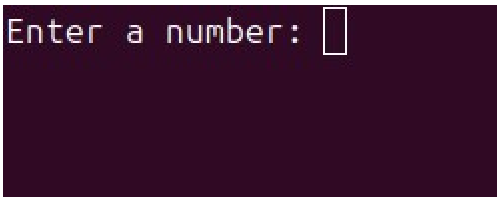
Introduction:

Factorial of a number is the product of all positive integers from 1 to that number. It is denoted by n! and is calculated as n! = n * (n-1) * (n-2) ... * 1. Shell scripting allows us to automate such mathematical calculations efficiently using loops and arithmetic operations.

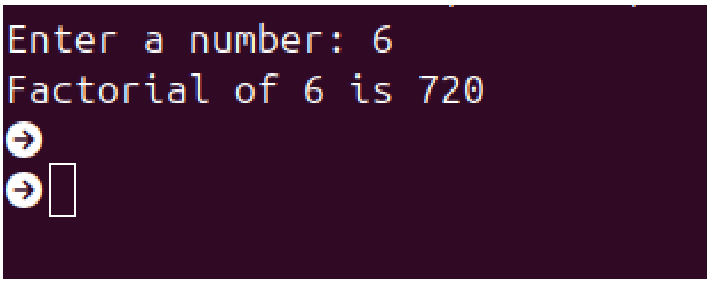
Code:

```
# Read number input from user
1.echo -n "Enter a number: "
2.read num
3.# Initialize factorial to 1
4.factorial=1
5.# Calculate factorial using a loop
6.for (( i=1; i<=num; i++ ))
7.do
8.factorial=$((factorial * i))
9.done
10.# Display the result
11.echo "Factorial of $num is $factorial"
```

Input:



Output:



Discussion:

In this experiment, a shell script was developed to find the factorial of a given number. The program takes user input using the `read` command and calculates the factorial using a `for` loop. The loop iterates from 1 to the entered number, multiplying the values to obtain the factorial. This experiment illustrates the use of loops and arithmetic operations in shell scripting, emphasizing the application of control structures for iterative calculations.

Experiment No: 5

Experiment Name: CPU Scheduling Algorithms – FCFS, SJF, Priority, Round Robin

Objective:

- To understand and implement different CPU scheduling algorithms.
- To compare the performance of FCFS, SJF, Priority, and Round Robin scheduling algorithms.
- To analyze and evaluate the efficiency of these algorithms based on waiting time and turnaround time.

Introduction:

CPU scheduling is a fundamental operating system concept that determines the order in which processes are executed by the CPU. Scheduling algorithms play a crucial role in ensuring fair and efficient CPU usage. The commonly used algorithms include:

1. **FCFS (First Come First Serve):** Processes are executed in the order of their arrival.
2. **SJF (Shortest Job First):** Processes with the shortest burst time are executed first.
3. **Priority Scheduling:** Processes are executed based on their priority level.
4. **Round Robin:** Processes are executed in a cyclic order with a fixed time quantum.

Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct Process {
int pid, burst_time, arrival_time, priority;
int waiting_time, turnaround_time;
};
void fcfs(vector<Process>& processes) {
int n = processes.size();
int current_time = 0;
for (int i = 0; i < n; ++i) {
if (current_time < processes[i].arrival_time)
current_time = processes[i].arrival_time;
processes[i].waiting_time = current_time - processes[i].arrival_time;
processes[i].turnaround_time = processes[i].waiting_time + processes[i].burst_time;
current_time += processes[i].burst_time;
}
}
void sjf(vector<Process>& processes) {
int n = processes.size();
vector<bool> completed(n, false);
int current_time = 0, completed_count = 0;
while (completed_count < n) {
int shortest = -1;
for (int i = 0; i < n; ++i) {
if (!completed[i] && processes[i].arrival_time <= current_time) {
if (shortest == -1 || processes[i].burst_time < processes[shortest].burst_time)
shortest = i;
}
}
if (shortest == -1) {
current_time++;
} else {
completed[shortest] = true;
processes[shortest].waiting_time = current_time - processes[shortest].arrival_time;
processes[shortest].turnaround_time = processes[shortest].waiting_time + processes[shortest].burst_time;
current_time += processes[shortest].burst_time;
completed_count++;
}
```

```
}
}
void priorityScheduling(vector<Process>& processes) {
sort(processes.begin(), processes.end(), [](Process a, Process b) {
return a.priority < b.priority;
});
fcfs(processes);
}
void roundRobin(vector<Process>& processes, int quantum) {
int n = processes.size();
vector<int> remaining_time(n);
for (int i = 0; i < n; ++i) remaining_time[i] = processes[i].burst_time;
int current_time = 0;
bool done;
do {
done = true;
for (int i = 0; i < n; ++i) {
if (remaining_time[i] > 0) {
done = false;
if (remaining_time[i] > quantum) {
current_time += quantum;
remaining_time[i] -= quantum;
} else {
current_time += remaining_time[i];
processes[i].waiting_time = current_time - processes[i].burst_time - processes[i].arrival_time;
processes[i].turnaround_time = processes[i].waiting_time + processes[i].burst_time;
remaining_time[i] = 0;
}
}
}
} while (!done);
}
void printResults(const vector<Process>& processes) {
cout << "PID\tWaiting Time\tTurnaround Time\n";
for (const auto& p : processes) {
cout << p.pid << "\t" << p.waiting_time << "\t\t" << p.turnaround_time << "\n";
}
}
int main() {
vector<Process> processes = {{1, 6, 0, 2}, {2, 8, 1, 1}, {3, 7, 2, 3}, {4, 3, 3, 4}};
cout << "FCFS:\n";
fcfs(processes);
printResults(processes);
cout << "\nSJF:\n";
```

```
sjf(processes);
printResults(processes);
cout << "\nPriority Scheduling:\n";
priorityScheduling(processes);
printResults(processes);
cout << "\nRound Robin (Quantum=2):\n";
roundRobin(processes, 2);
printResults(processes);
return 0;
}
```

Input:

The input to the program is defined as a vector of processes, where each process is represented by a structure containing the following attributes:

- Process ID (PID): Unique identifier for the process.
- Burst Time: The time required by the process to complete execution.
- Arrival Time: The time at which the process arrives in the ready queue.
- Priority: The priority level of the process (lower value indicates higher priority).

```
vector<Process> processes = {
   {1, 6, 0, 2},  // Process 1: Burst Time = 6, Arrival Time = 0, Priority = 2
   {2, 8, 1, 1},  // Process 2: Burst Time = 8, Arrival Time = 1, Priority = 1
   {3, 7, 2, 3},  // Process 3: Burst Time = 7, Arrival Time = 2, Priority = 3
   {4, 3, 3, 4}   // Process 4: Burst Time = 3, Arrival Time = 3, Priority = 4
};
```

PID(Process ID)	Burst Time	Arrival Time	Priority
1	6	0	2
2	8	1	1
3	7	2	3
4	3	3	4

Output:

The output of the program includes the waiting time and turnaround time for each process under different scheduling algorithms. The results are displayed in the following format:

```
FCFS:
PID Waiting Time    Turnaround Time
1   0       6
2   5       13
3   12      19
4   18      21

SJF:
PID Waiting Time    Turnaround Time
1   0       6
2   15      23
3   7       14
4   3       6

Priority Scheduling:
PID Waiting Time    Turnaround Time
2   0       8
1   9       15
3   13      20
4   19      22

Round Robin (Quantum=2):
PID Waiting Time    Turnaround Time
2   14      22
1   13      19
3   15      22
4   9       12


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

Discussion:

The experiment demonstrates the behavior and performance of four CPU scheduling algorithms: FCFS, SJF, Priority Scheduling, and Round Robin. FCFS is simple but suffers from high waiting times due to the convoy effect. SJF minimizes waiting time but can cause starvation for longer processes. Priority Scheduling ensures high-priority tasks are executed first but may starve low-priority processes. Round Robin, with its time quantum, provides fairness and prevents starvation, though it incurs higher overhead due to frequent context switching. Each algorithm has its trade-offs, and the choice depends on system requirements, such as fairness, efficiency, and responsiveness.