

EXPERIMENT-3

Implement FCFS, SJF, priority scheduling, and RR scheduling algorithms in C language.

Purpose of Scheduling algorithm

- Maximum CPU utilization
- Fair allocation of CPU
- Maximum throughput
- Minimum turnaround time
- Minimum waiting time
- Minimum response time

There are four types of process scheduling algorithms:

1. First Come First Serve (FCFS) Scheduling
2. Shortest Job First (SJF) Scheduling
3. Round Robin Scheduling
4. Priority Scheduling

First Come First Serve (FCFS) Scheduling

First come first serve (FCFS) scheduling algorithm simply schedules the jobs according to their arrival time. The job which comes first in the readyqueue will get the CPU first. The lesser the arrival time of the job, the sooner will the job get the CPU. FCFS scheduling may cause the problem of starvation if the burst time of the first process is the longest among all the jobs.

Completion Time: Time at which process completes its execution.

Turn Around Time(TAT): Time Difference between completion time and arrival time. $\text{Turn Around Time} = \text{Completion Time} - \text{Arrival Time}$

Waiting Time(WT): Time Difference between turnaround time and bursttime. $\text{Waiting Time} = \text{Turn Around Time} - \text{Burst Time}$

Program Code

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

void main(){

clrscr();
int
bt[10]={0},at[10]={0},tat[10]={0},wt[10]={0},ct[10]={0};

int n,sum=0;

float totalTAT=0,totalWT=0;

printf("Enter number of processes: ");

scanf("%d",&n);
printf("Enter Arrival time and Burst time for each process:\n");

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

    printf("Arrival time of process[%d]:
    ",i+1); scanf("%d",&at[i]);
    printf("Burst time of process[%d]: ",i+1);
    scanf("%d",&bt[i]);
}

//calculate completion time of processes
for(int j=0;j<n;j++)
{

    sum+=bt[j];
    ct[j]+=sum;
}

//calculate turnaround time and waiting
times for(int k=0;k<n;k++)
```

```
{  
  
    tat[k]=ct[k]-at[k];  
  
    totalTAT+=tat[k];  
}  
  
for(k=0;k<n;k++)  
{  
  
    wt[k]=tat[k]-bt[k];  
  
    totalWT+=wt[k];  
}  
  
printf("P#\t AT\t BT\t CT\t TAT\t WT\t\n\n");  
  
for(i=0;i<n;i++)  
{  
  
    printf("P%d\t %d\t %d\t %d\t  
%d\t %d\n",i+1,at[i],bt[i],ct[i],tat[i],wt[i]);  
}  
  
printf("\nAverage Turnaround Time = %f\n",totalTAT/n);  
  
printf("Average WT = %f\n\n",totalWT/n);  
getch();  
}
```

OUTPUT :

Enter number of processes: 5

Enter Arrival time and Burst time for each process:

Arrival time of process[1]: 0

Burst time of process[1]: 4

Arrival time of process[2]:

Burst time of process[2]: 3

Arrival time of process[3]: 2

Burst time of process[3]: 1

Arrival time of process[4]: 3

Burst time of process[4]: 2

Arrival time of process[5]: 4

Burst time of process[5]: 5

P#	AT	BT	CT	TAT	WT
P1	0	4	4	4	0
P2	1	3	7	6	3
P3	2	1	8	6	5
P4	3	2	10	7	5
P5	4	5	15	11	6
Average Turnaround Time					= 6.800000

Average WT = 3.800000

Shortest job first(SJF) is a scheduling algorithm, that is used to schedule process in an operating system. It is a very important topic in scheduling when compared to round robin and FCFS Scheduling. In the following order:

- Types of SJF
- Non-Pre-emptive SJF
- Code for Non-Pre-emptive SJF Scheduling
- Code for Pre-emptive SJF Scheduling

There are two types of SJF

- Pre-emptive SJF
- Non-Preemptive SJF

These algorithm schedule processes in the order in which the shortest job is done first. It has a minimum average waiting time.

There are 3 factors to consider while solving SJF, they are

1. BURST Time
2. Average waiting time
3. Average turnaround time
- 4.

Non-Preemptive Shortest Job First

Here is an example

Processes Id	Burst Time	Waiting Time	TAT
4	3	0	
1	6	3	
3	7	9	
2	8	16	

Average waiting time = 7

Average turnaround time = 13

TAT = waiting time + burst time

Code for Shortest Job First Scheduling

```
#include<stdio.h>

int main()
{
    int bt[20], p[20], wt[20], tat[20], i, j, n, total=0,
    pos, temp; float avg_wt, avg_tat;

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

    // Get burst time for each process
```

```

printf("\nEnter Burst Time:\n");
for(i = 0; i < n; i++)
{
    printf("P%d: ", i + 1);
    scanf("%d", &bt[i]);
    p[i] = i + 1;
}

// Sort processes based on burst time using SJF algorithm
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;
}

// Calculate waiting time for each process
wt[0] = 0;

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

    total += wt[i];
}

// Calculate turnaround time for each process
for(i = 0; i < n; i++)
{
    tat[i] = bt[i] + wt[i];
    total += tat[i];
}

```

```
// Calculate average waiting time and average turnaround
time avg_wt = (float)total / n;
avg_tat = (float)total / n;

// Display results
printf("\nProcess\t Burst Time\t Waiting Time\t Turnaround Time\n");
for(i = 0; i < n; i++)
    printf("P%d\t\t %d\t\t %d\t\t %d\n", p[i], bt[i], wt[i], tat[i]);

printf("\nAverage Waiting Time = %f", avg_wt);
printf("\nAverage Turnaround Time = %f\n", avg_tat);

return 0;
}
```

Output

```
C:\WINDOWS\SYSTEM32\cmd.exe
Enter number of process:5
Enter Burst Time:
p1:4
p2:3
p3:7
p4:1
p5:2

Process    Burst Time    Waiting Time    Turnaround Time
p4         1             0              1
p5         2             1              3
p2         3             3              6
p1         4             6             10
p3         7            10             17

Average Waiting Time=4.000000
Average Turnaround Time=7.400000

-----
(program exited with code: 0)
Press any key to continue . . .
```

In the above program, we calculate the average waiting and average turnaround time of the jobs. We first ask the user to enter number of the processes and store it in n. We then accept the burst time from the user. It is stored in the bt array.

After this, the burst time are sorted in the next section so the shortest one can be executed first. Here selection sort is used to sort array of the burst time bt.

Waiting time of the first element is zero, the remaining waiting time is calculated by using two for loop that runs from 1 to n that controls the outer loop and the inner loop is controlled by another for loop that runs from j=0 to j<I inside the loop, the waiting time is calculated by adding the burst time to the waiting time.

```

1 for(i=1;i<n;i++)
2 {
3     wt[i]=0;
4     for(j=0;j<i;j++)

5         wt[i]+=bt[j];
6     total+=wt[i];
7 }

```

Total is the addition of all the waiting time together. The average waiting time is calculated:

avg_wt=(float)total/n;

and it is printed.

Next, the turnaround time is calculated by adding the burst time and the waiting time

```

1 for(i=0;i<n;i++)
2 {
3     tat[i]=bt[i]+wt[i];
4     total+=tat[i];
5     printf("np%dt\t %dt\t %dttt%d", p[i] , bt[i] , wt[i] , tat[i]);
6 }

```

Again, here the for loop is used. And the total variable here holds the total turnaround time. After this the average turnaround time is calculated. This is how Non-Preemptive scheduling takes place.

Code for Preemptive SJF Scheduling

```

#include <stdio.h>
int main()
{
    int arrival_time[10], burst_time[10], temp[10];
    int i, smallest, count = 0, time, limit;
    double wait_time = 0, turnaround_time = 0, end;
    float average_waiting_time, average_turnaround_time;

    printf("Enter the Total Number of Processes:
    "); scanf("%d", &limit);

    printf("Enter Details of %d Processes\n", limit);

    for(i = 0; i < limit; i++)

```



```

{
    printf("Enter Arrival Time: ");
    scanf("%d", &arrival_time[i]);

    printf("Enter Burst Time: ");
    scanf("%d", &burst_time[i]);

    temp[i] = burst_time[i];
}

burst_time[9] = 9999;

for(time = 0; count != limit; time++)
{
    smallest = 9;

    for(i = 0; i < limit; i++)
    {
        if(arrival_time[i] <= time && burst_time[i] < burst_time[smallest]
&& burst_time[i] > 0)
        {
            smallest = i;
        }
    }

    burst_time[smallest]--;

    if(burst_time[smallest] == 0)
    {
        count++;
        end = time + 1;
        wait_time += end - arrival_time[smallest] - temp[smallest];
        turnaround_time += end - arrival_time[smallest];
    }
}

average_waiting_time = wait_time / limit;
average_turnaround_time = turnaround_time / limit;

printf("\nAverage Waiting Time: %f\n", average_waiting_time);
printf("Average Turnaround Time: %f\n", average_turnaround_time);

return 0;
}

```

Output

```

C:\WINDOWS\SYSTEM32\cmd.exe
Enter the Total Number of Processes: 4
Enter Details of 4 Processes
Enter Arrival Time: 1
Enter Burst Time: 4
Enter Arrival Time: 2
Enter Burst Time: 4
Enter Arrival Time: 3
Enter Burst Time: 5
Enter Arrival Time: 4
Enter Burst Time: 8
Average Waiting Time: 4.750000
Average Turnaround Time: 10.000000
-----
(program exited with code: 0)
Press any key to continue . . .

```

The only difference in preemptive and non-preemptive is that when two burst time are same the algorithm evaluates them on the first come first serve basis. Hence there is an arrival time variable.

With this, we come to an end of this Shortest Job Scheduling in C article. I hope you got an idea of how this scheduling works.

What is Priority Scheduling Algorithm?

In this priority scheduling, each process has a priority associated with it and as each process hits the queue, it is sorted based on its priority so that processes with higher priority is dealt with first. It should be noted that equal priority processes are scheduled in FCFS order.

To prevent high priority processes from running indefinitely the scheduler may decrease the priority of the currently running process at each clock tick (i.e., at each clock interrupt). If this action causes its priority to drop below that of the next highest process, a process switch occurs. Alternatively, each process may be assigned a maximum time quantum that it is allowed to run. When this quantum is used up, the next highest priority process is given a chance to run.

Example:

Here is an example of a Priority Scheduling algorithm.

Process	Arrival Time	Burst Time	Priority
P1	0	5	2
P2	1	3	1
P3	2	6	4
P4	4	4	3
P5	6	2	5

In this example, there are 5 processes with their arrival time, burst time, and priority. The execution order, waiting time, and turnaround time for each process will be as given below.

Process	Burst Time	Waiting Time	Turnaround Time
P2	3	0	3
P1	5	3	8
P4	4	8	12
P3	6	12	18
P5	2	18	20

- Average Waiting Time = $(0 + 3 + 8 + 12 + 18) / 5 = 8.2$
- Average Turnaround Time = $(3 + 8 + 12 + 18 + 20) / 5 = 12.2$

Limitations

The problem occurs when the operating system gives a particular task a very low priority, so it sits in the queue for a larger amount of time, not being dealt with by the CPU. If this process is something the user needs, there could be a very long wait, this process is known as “Starvation” or “Infinite Blocking”.

Solution

Many operating systems use a technique called “aging”, in which a low priority process slowly gains priority over time as it sits in the queue. Even if, the priority of the process is low, there is a surety of its execution.

C Program for Priority Scheduling

```
#include <stdio.h>
int main() {
```

```
int bt[20], p[20], wt[20], tat[20], pr[20], i, j, n, total = 0, pos, temp, avg_wt,
avg_tat; printf("Enter Total Number of Processes:"); scanf("%d", &n);
```

```
printf("\nEnter Burst Time and
Priority\n"); for (i = 0; i < n; i++) {
    printf("\nP[%d]\n", i + 1);
    printf("Burst Time:");
    scanf("%d", &bt[i]);
    printf("Priority:");
    scanf("%d", &pr[i]);
    p[i] = i + 1; //contains process number
}
```

```
//sorting burst time, priority and process number in ascending order using selection
sort
```

```
for (i = 0; i < n; i++) {
    pos = i;
    for (j = i + 1; j < n; j++) {
        if (pr[j] < pr[pos])
            pos = j;
    }
    temp = pr[i];
    pr[i] = pr[pos];
    pr[pos] = temp;
    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 is
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 = total / n; //average waiting time
total = 0;
```

```
printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time");
```

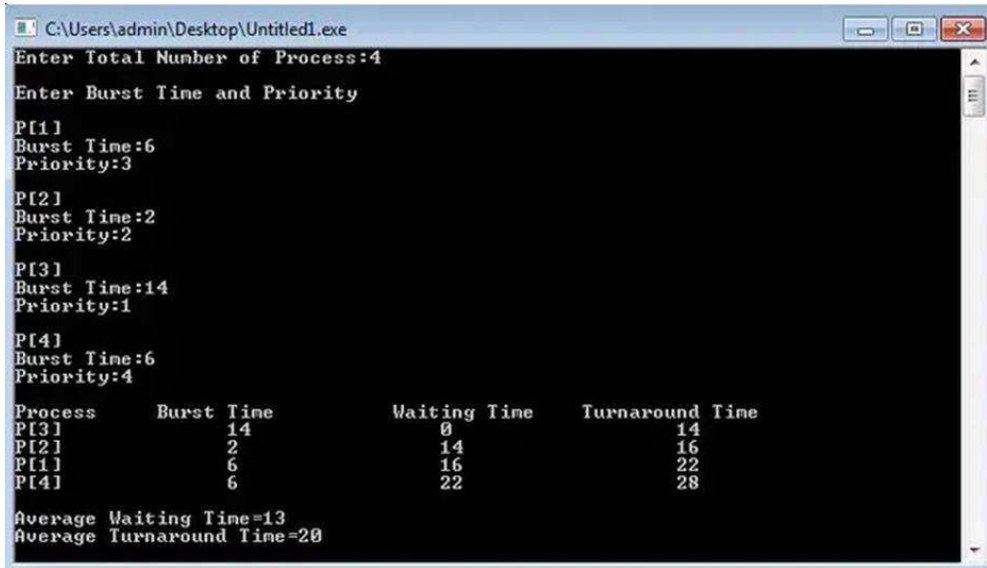
```
for (i = 0; i < n; i++) {
    tat[i] = bt[i] + wt[i]; //calculate turnaround
    time total += tat[i];
}
```

```

        printf("\nP[%d]\t\t%d\t\t%d\t\t%d", p[i], bt[i], wt[i], tat[i]);
    }
    avg_tat = total / n; //average turnaround time
    printf("\n\nAverage Waiting Time = %d", avg_wt);
    printf("\n\nAverage Turnaround Time = %d\n",
    avg_tat); return 0;
}

```

Output



```

C:\Users\admin\Desktop\Untitled1.exe
Enter Total Number of Process:4
Enter Burst Time and Priority
P[1]
Burst Time:6
Priority:3
P[2]
Burst Time:2
Priority:2
P[3]
Burst Time:14
Priority:1
P[4]
Burst Time:6
Priority:4
Process    Burst Time    Waiting Time    Turnaround Time
P[3]       14            0              14
P[2]       2            14             16
P[1]       6            16             22
P[4]       6            22             28
Average Waiting Time=13
Average Turnaround Time=20

```

Round Robin

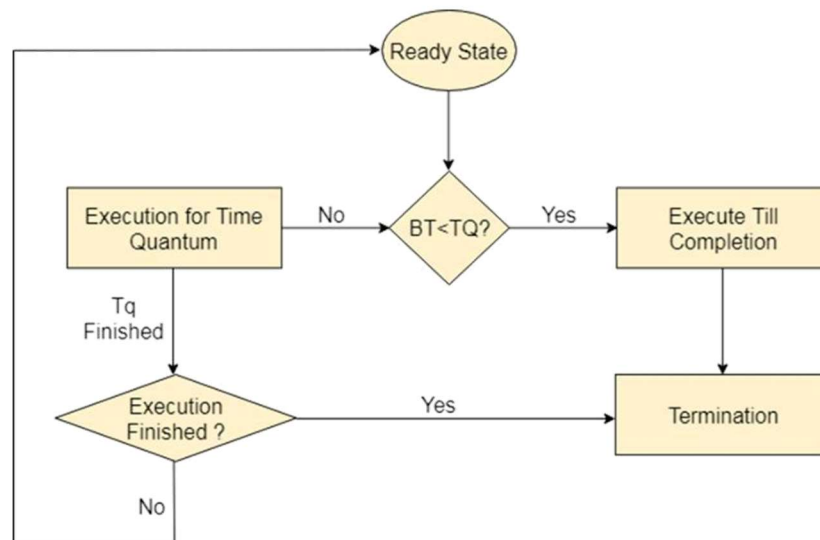
Round Robin CPU Scheduling is the most important CPU Scheduling Algorithm which is ever used in the history of CPU Scheduling Algorithms. Round Robin CPU Scheduling uses Time Quantum (TQ). The Time Quantum is something which is removed from the Burst Time and lets the chunk of process to be completed.

Time Sharing is the main emphasis of the algorithm. Each step of this algorithm is carried out cyclically. The system defines a specific time slice, known as a time quantum.

First, the processes which are eligible to enter the ready queue enter the ready queue. After entering the first process in Ready Queue is executed for a Time Quantum chunk of time. After execution is complete, the process is removed from the ready queue. Even now the process requires some time to complete its execution, then the process is added to Ready Queue.

The Ready Queue does not hold processes which already present in the Ready Queue. The Ready Queue is designed in such a manner that it does not hold non unique processes. By holding same processes Redundancy of the processes increases.

After, the process execution is complete, the Ready Queue does not take the completed process for holding.



Advantages

The advantages of Round Robin CPU Scheduling are:

1. A fair amount of CPU is allocated to each job.
2. Because it doesn't depend on the burst time, it can truly be implemented in the system.
3. It Is not affected by the convoy effect or the starvation problems as occurred in First Come First Serve CPU Scheduling Algorithm.

Disadvantages

The disadvantages of Round Robin CPU Scheduling are:

1. Low Operating System slicing times will result in decreased CPU output.
2. Round Robin CPU Scheduling approach takes longer to swap contexts.
3. Time quantum has a significant impact in its performance.
4. The procedures cannot have priorities established.

Examples:

1. S. No	Process ID	Arrival Time	Burst Time
2. _____	_____	_____	_____
3. 1	P 1	0	7
4. 2	P 2	1	4
5. 3	P 3	2	15
6. 4	P 4	3	11
7. 5	P 5	4	20
8. 6	P 6	4	9

Assume Time Quantum TQ = 5

Ready Queue:

P1, P2, P3, P4, P5, P6, P1, P3, P4, P5, P6, P3, P4, P5

Gantt Chart:

P1	P2	P3	P4	P5	P6	P1	P3	P4	P5	P6	P3	P4	P5
0	5	9	14	19	24	29	31	36	41	46	50	55	66

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
P1	0	7	31	31	24
P2	1	4	9	8	4
P3	2	15	55	53	38
P4	3	11	56	53	42
P5	4	20	66	62	42
P6	4	9	50	46	37

Average Completion Time

1. Average Waiting Time = $(5 + 26 + 5 + 42 + 42 + 37) / 6$
2. Average Waiting Time = $157 / 6$
3. Average Waiting Time = 26.16667

Average Turnaround Time

1. Average Turnaround Time = $(31 + 8 + 53 + 53 + 62 + 46) / 6$
2. Average Turnaround Time = $253 / 6$
3. Average Turnaround Time = 42.16667

EXPERIMENT -4

Implement the basic and user status commands like: su, sudo, man, help, history, who, whoami, id, uname, uptime, free, tty, cal, date, hostname, reboot, clear.

Introduction: Linux is famous for its powerful commands. To use Linux effectively, all users should know how to use terminal commands. Although the OS has a GUI, many functionalities work faster when run as commands through the terminal.

This guide showcases basic Linux commands all users should know.



Prerequisites

- A system running Linux.
- Access to the command line/terminal.

Basic Linux Commands

All Linux commands fall into one of the following four categories:

- **Shell builtins** - Commands built directly into the shell with the fastest execution.
- **Shell functions** - Shell scripts (grouped commands).
- **Aliases** - Custom command shortcuts.
- **Executable programs** - Compiled and installed programs or scripts.

1. pwd command

The **pwd** command (print working directory) is a shell builtin command that prints the current location. The output shows an absolute directory path, starting with the root directory (/)

The general syntax is:

```
pwd<options>
```

To see how the command works, run the following in the terminal:

```
pwd
```

```
kb@phoenixNAP:~$ pwd
/home/kb
kb@phoenixNAP:~$
```

The output prints the current location in the **/home/<username>** format.

2. ls command

The **ls** command (**list**) prints a list of the current directory's contents.

Run the following:

```
ls
```

```
kb@phoenixNAP:~$ ls
Desktop  Downloads  Pictures  snap      Videos
Documents Music      Public    Templates
kb@phoenixNAP:~$
```

Additional options provide flexibility with the display output. Typical usage includes combining the following options

- Show as a list:

```
ls -l
```

- Show as a list and include hidden files

```
ls-la
```

- Show sizes in a human-readable format:

```
ls -lah
```

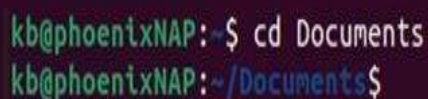
3. cd command

The **cd** command (change directory) is a shell builtin command for changing the current working directory:

```
cd <directory>
```

For example, to move to the *Document* directory, run:

```
Cd Documents
```



```
kb@phoenixNAP:~$ cd Documents
kb@phoenixNAP:~/Documents$
```

The working directory changes in the terminal interface. In a non-default interface, use the `pwd` command to check the current directory.

Use **cd** without any parameters to return to the home directory (~).

4. cat command

The **cat** command (concatenate) displays the contents of a file in the terminal (standard output or stdout). To use the command, provide a file name from the current directory:

```
cat <filename>
```

```
kb@phoenixNAP:~$ cat file.txt
Hello world!
kb@phoenixNAP:~$
```

Alternatively, provide a path to the file along with the file name:

```
cat <path>/<filename>
```

The command can also:

- Display contents of multiple files:

```
cat <file 1> <file 2>
```

- Create new files:

```
cat ><filename>
```

Add contents to the file and press **CTRL+D** to exit.

- Display line numbers:

```
cat -n <filename>
```

4. touch command

The **touch** command's primary purpose is to modify an existing file's timestamp. To use the command, run:

```
touch <filename>
```

```
kb@phoenixNAP:~$ touch new_file.txt
kb@phoenixNAP:~$ ls
Desktop  Downloads  Music      Pictures  Templates
Documents file.txt   new_file.txt Public     Videos
kb@phoenixNAP:~$
```

The command creates an empty file if it does not exist. Due to this effect, **touch** is also a quick way to make a new file (or a batch of files).

6. cp command

The main way to copy files and directories in Linux is through the **cp** (copy) command, Try the command with:

```
cp <source file> <target file>
```

```
kb@phoenixNAP:~$ cp file.txt file_copy.txt
kb@phoenixNAP:~$ ls
Desktop  Downloads  file.txt  Pictures  Templates
Documents file_copy.txt Music     Public    Videos
kb@phoenixNAP:~$
```

The source and target files must have different names since the command copies in the same directory. Provide a path before the file name to copy to another location.

7. mv command

Use the **mv** (move) command to move files or directories from one location to another. For example, to move a file from the current directory to `~/Documents`, run:

```
mv <filename> ~/Documents/<filename>
```

```
kb@phoenixNAP:~$ mv file.txt ~/Documents/file.txt
kb@phoenixNAP:~$ ls ~/Documents
file.txt
kb@phoenixNAP:~$
```

8. mkdir command

Use the **mkdir** (make directory) command creates a new directory in the provided location. Use the command in the following format:

```
mkdir <directory name>
```

```
kb@phoenixNAP:~$ mkdir New_directory
kb@phoenixNAP:~$ ls
Desktop  Downloads  New_directory  Public  Videos
Documents Music      Pictures       Templates
kb@phoenixNAP:~$
```

Provide a path to create a directory in the given location, or use a space or comma-separated list to create multiple directories simultaneously.

9. rmdir command

Use the **rmdir** (remove directory) command to delete an empty directory. For example:

```
rmdir <directory name>
```

```
kb@phoenixNAP:~$ ls
Desktop  Downloads  New_directory  Public  Videos
Documents Music      Pictures      Templates
kb@phoenixNAP:~$ rmdir New_directory
kb@phoenixNAP:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
kb@phoenixNAP:~$
```

If the directory is not empty, the command fails.

10. rm command

The **rm** command (remove) deleted files or directories. To use the command for non-empty directories, add the **-r** tag:

```
rm -r <file or directory>
```

```
kb@phoenixNAP:~$ ls Documents
file.txt
kb@phoenixNAP:~$ rm -r Documents
kb@phoenixNAP:~$ ls
Desktop  Downloads  Music  Pictures  Public  Templates  Videos
kb@phoenixNAP:~$
```

Unlike the **rmdir** command, **rm** also removes all the contents from the directory.

11. locate command

The **locate** command is a single Linux tool for finding a file. The command checks a file database on a system to perform the search quickly. However, the result is sometimes inaccurate if the database is not updated.

To use the command, install locate and try the following example:

```
locate <filename>
```

```
kb@phoenixNAP:~$ locate file.txt
/home/kb/Documents/file.txt
/usr/share/doc/alsa-base/driver/Procfile.txt.gz
kb@phoenixNAP:~$
```

The output prints the file's location path. The matching is unclear and outputs any file that contains the file name.

12. file command

Use the **find** command to perform a through search on the system. Add the **- name** tag to search for a file or directory by name:

```
find -name <file or directory>
```

```
kb@phoenixNAP:~$ find -name file.txt
./Documents/file.txt
kb@phoenixNAP:~$
```

The output prints the file's path and perform an exact match. Use additional options to control the search further.

13. grep command

The **grep** (global regular expression print) enables searching through text in a file or a standard output. The basic syntax is:

```
grep <search string> <filename>
```

```
kb@phoenixNAP:~$ grep world Documents/file.txt
Hello world!
kb@phoenixNAP:~$
```

The output highlights all matches. Advanced commands include using grep for multiple strings or writing grep regex statements.

14. sudo command

Use **sudo** (superuser do) elevates a user's permissions to administrator or root. Commands that change system configuration require elevated privilege.

Add **sudo** as a prefix to any command that requires elevated privileges:

```
sudo <command>
```

Use the command with caution to avoid making accidental changes permanent.

15. df command

The **df** (disk free) command shows the statistics about the available disk space on the filesystem. To see how **df** works, run the following:

df

```
kb@phoenixNAP:~$ df
Filesystem      1K-blocks    Used Available Use% Mounted on
tmpfs            848412      1580    846832   1% /run
/dev/sda3       245071124  9720620  222828728   5% /
tmpfs           4242056        0    4242056   0% /dev/shm
tmpfs            5120         4        5116   1% /run/lock
/dev/sda2       524252      5364    518888   2% /boot/efi
tmpfs           848408      2404    846004   1% /run/user/1000
kb@phoenixNAP:~$
```

The output shows the amount of space used by different drives. Add the **-h** tag to make the output in human-readable format (kilobytes, megabytes, and gigabytes).

16. du command

The **du** (disk usage) command helps show how much space a file or directory takes up. Run the command without any parameters:

du

```
kb@phoenixNAP:~$ du
8      ./Documents
4      ./Public
4      ./Downloads/firefox.tmp/Temp-ba964148-d1ac-4718-ab72-33eda062843d
508    ./Downloads/firefox.tmp
512    ./Downloads
4      ./Videos
4      ./Templates
4      ./Pictures
4      ./local/share/gnome-settings-daemon
```

The output shows the amount of space used by files and directories in the current directory. The size displays in blocks, and adding the tag changes the measure to human-readable format.

17. head command

Use the **head** command to truncate long outputs. The command can truncate files, for example:

head <filename>

Alternatively, pipe **head** to a command with a long output:

```
<command> | head
```

For example, to see the first ten lines of the **du** command, run:

```
du | head
```

```
kb@phoenixNAP:~$ du | head
8      ./Documents
4      ./Public
4      ./Downloads/firefox.tmp/Temp-ba964148-d1ac-4718-ab72-33eda062843d
508    ./Downloads/firefox.tmp
512    ./Downloads
4      ./Videos
4      ./Templates
4      ./Pictures
4      ./local/share/gnome-settings-daemon
4      ./local/share/icc
kb@phoenixNAP:~$
```

The output shows the first ten lines instead of everything.

18. tail command

The Linux tail command does the opposite of **head**. Use the command to show the last ten lines of a file:

```
tail <filename>
```

Or pipe **tail** to a command with a long output:

```
<command> | tail
```

For example, use **tail** to see the last of the **du** command:

```
du | tail
```



```
kb@phoenixNAP:~$ du | tail
12      ../config/evolution
12      ../config/ibus/bus
16      ../config/ibus
84      ../config/pulse
8       ../config/dconf
4       ../config/goa-1.0
4       ../config/update-notifier
8       ../config/gtk-3.0
156     ../config
42892   .
kb@phoenixNAP:~$
```

Both **head** and **tail** commands are helpful when reading Linux log files.

19. diff command

The **diff** (difference) command two files and prints the difference. To use the command, run:

```
diff <file 1> <file 2>
```

For example, to compare files *test1.txt* and *test2.txt*, run:

```
diff file1.txt file2.txt
```

```
kb@phoenixNAP:~$ diff file1.txt file2.txt
1c1
< Hello world!
---
> Hello world
kb@phoenixNAP:~$
```

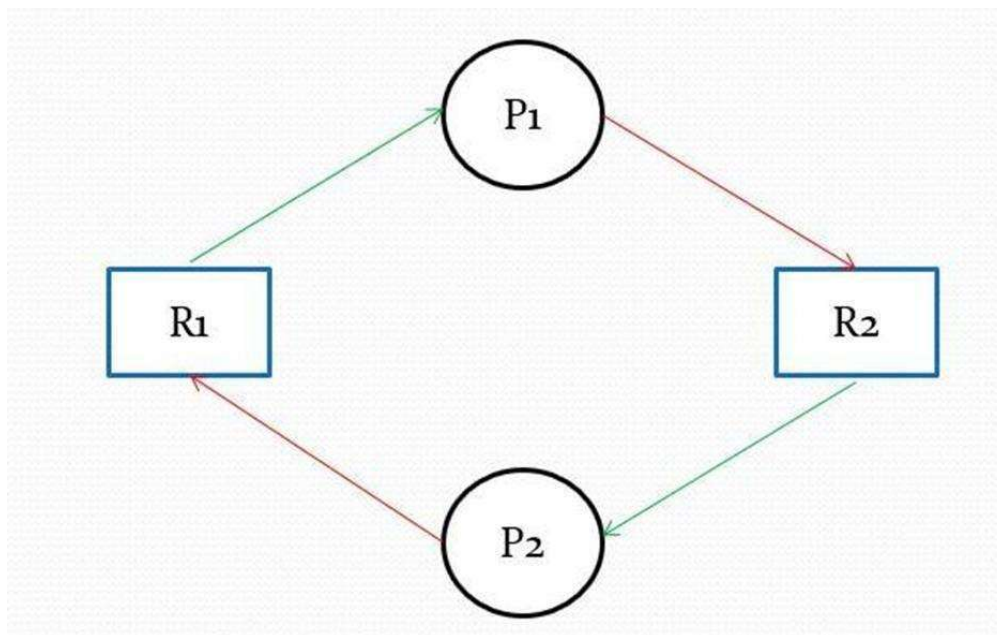
Developers often use **diff** to compare versions of the same code.

EXPERIMENT -5

Implement deadlock in C by using shared variable.

Deadlock in operating system is a situation which occurs when a process or thread enters a waiting state because a resource requested is being held by another waiting process, which in turn is waiting for another resource held by another waiting process. In a deadlock state a process is unable to change its state(waiting) indefinitely because the resources requested by it are being used by another waiting process.

Setup



To simulate deadlock in the system we will create the above shown situation.

P1 and P2 will be represented by two thread one and two. The two resources R1 and R2 will be represented the two lockvariables **first_mutex** and **second_mutex**.

First thread **one** will acquire lock **first_mutex** and then thread **two** will acquire lock **second_mutex**. Hence, both the threads have acquired oneresource each. Next, thread **one** will try to acquire lock **second_mutex** while the second thread, thread **two** will try to acquire lock **first_mutex**. Since the resources are already occupied bythe other thread, both the threads will get into a deadlock.

Note: You must know how to create Threads to understand this program

Program to create Deadlock Using C in Linux using Mutex Locks and threads

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

```

void *function1(); void
*function2();
pthread_mutex_t first_mutex; //mutex lock
pthread_mutex_t second_mutex;

int main() {
pthread_mutex_init(&first_mutex,NULL); //initialize the lock
pthread_mutex_init(&second_mutex,NULL);
pthread_t one, two;
pthread_create(&one, NULL, function1, NULL); // create thread
pthread_create(&two, NULL, function2, NULL);
pthread_join(one, NULL);
pthread_join(two, NULL);
printf("Thread joined\n");
}

void *function1( ) {

    pthread_mutex_lock(&first_mutex); // to acquire the resource/mutex lock

    printf("Thread ONE acquired first_mutex\n");

    sleep(1); pthread_mutex_lock(&second_mutex);

    printf("Thread ONE acquired second_mutex\n");

    pthread_mutex_unlock(&second_mutex); // to release the resource

    printf("Thread ONE released second_mutex\n");

    pthread_mutex_unlock(&first_mutex);

    printf("Thread ONE released first_mutex\n");
}

```

```
void *function2( ) {
    pthread_mutex_lock(&second_mutex);
    printf("Thread TWO acquired second_mutex\n");
    sleep(1);
    pthread_mutex_lock(&first_mutex);
    printf("Thread TWO acquired first_mutex\n");
    pthread_mutex_unlock(&first_mutex);
    printf("Thread TWO released first_mutex\n");
    pthread_mutex_unlock(&second_mutex);
    printf("Thread TWO released second_mutex\n");
}
```

Synchronization primitives Mutex locks • Used for exclusive access to shared resource (critical section) • Operation: Lock, unlock Semaphores • Generalization of mutexes: Count number of availability “resources” • Wait for an available resource (decrement), notify availability (increment) • Example: wait for free buffer space, signal more buffer space Condition variables

• Represent an arbitrary event • Operations: Wait for event, signal occurrence of event • Toed to a mutex for mutual exclusion 5 Condition variables Goal: Wait for a specific event to happen

• Event depends on state shared with multiple threads Solution: condition variables • “Names” an event • Internally, is a queue of threads waiting for the event Basic operations • Wait for event • Signal occurrence of event to one waiting thread •Signal occurrence of event to all waiting threads Signaling, not mutual exclusion • Condition variable is intimately tied to a mutex

EXPERIMENT -6

File system: Introduction to File system Architecture and File Types.

In a computer, a file system – sometimes written filesystem – is the way in which files are named and where they are placed logically for storage and retrieval. Without a file system, stored information wouldn't be isolated into individual files and would be difficult to identify and retrieve. As data capacities increase, the organization and accessibility of individual files are becoming even more important in data storage.

Digital file systems and files are named for and modeled after paper-based filing systems using the same logic-based method of storing and retrieving documents.

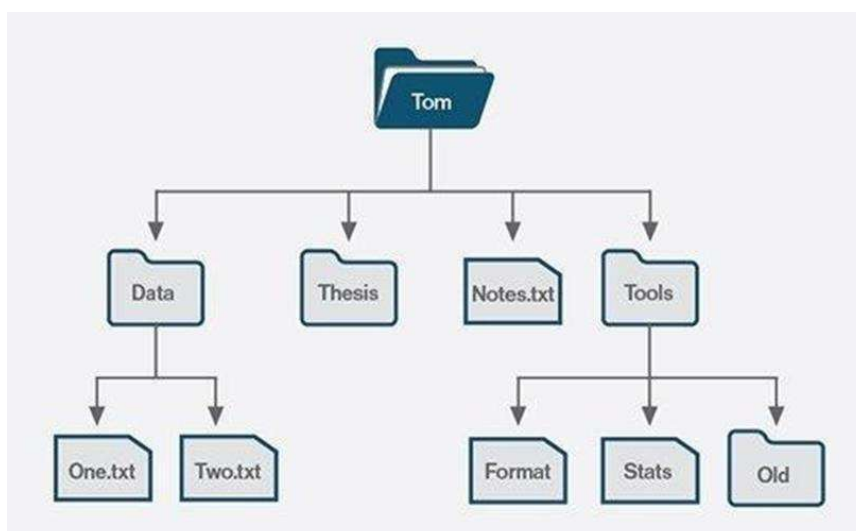
File systems can differ between operating systems (OS), such as Microsoft Windows, macOS and Linux-based systems. Some file systems are designed for specific applications. Major types of file systems include distributed file systems, disk-based file systems and special purpose file systems.

How file systems work ?

A file system stores and originates data and can be thought of as a type of index for all data contained in a storage device. These devices can include hard drives, optical drives and flash drives.

File system specify conventions for naming files, including the maximum number of characters in name, which characters can be used and, in some systems, how long the file name suffix can be. In many file systems, file names are not case sensitive.

Along with the file itself, file systems contain information such as the size of the file, as well as its attributes, location and hierarchy in the directory in the metadata. Metadata can also identify free blocks of available storage on the drive and how much space is available. Example of a file tree diagram



A file system also includes a format to specify the path to a file through the structures of directories. A file is placed in a directory – or a folder in Windows OS – or subdirectory at the desired place in the tree structure. PC and mobile OSes have file systems in which files are placed somewhere in a hierarchical tree structure.

Before files and directories are created on the storage medium, partitions should be put into place. A partition is a region of the hard disk or other storage that the OS manages separately. One file system is contained in the primary partition, and some OSes allow for multiple partitions on one disk. In this situation, if one file system gets corrupted, the data in a different partition will be safe.

File systems and the role of metadata

File systems use metadata to store and retrieve files. Example of metadata tags include:

- Data created
- Data modified
- Last date of access
- Last backup
- Used ID of the creator
- Access Permission
- File size

Metadata is stored separately from the contents of the file, with many file systems storing the file names in separate directory entries. Some metadata may be kept in the directory, whereas other metadata may be kept in a structure called an inode.

In Unix-like operating systems, an inode can store metadata unrelated to the content of the file itself. The inode indexes information by number, which can be used to access the location of the file and then the file itself.

An example of a file system that capitalizes on metadata is OS X, the OS used by Apple. It allows for a number of optimization features, including file names that can stretch to 255 characters.

File system access

File systems can also restrict read and write access to particular groups of users. Passwords are the easiest way to do this. Along with controlling who can modify or read files, restricting access can ensure that data modification is controlled and limited.

File permissions such as access or capability control lists can also be used to moderate file system access. These types of mechanisms are useful to prevent access by regular users, but not as effective against outside intruders.

Encrypting files can also prevent users' access, but it is focused more on protecting systems from outside attacks. An encryption key can be applied to unencrypted text to encrypt it, or the key can be used to decrypt encrypted text. Only users with the key can access the file. With encryption, the file system does not need to know the encryption key to manage the data effectively.

Types of file systems

There are number of types of file systems, all with different logical structures and properties, such as speed and size. The type of file system can differ by OS and the needs of that OS. The three most common PC operating systems are Microsoft Windows, Mac OS X and Linux. Mobile OSes include Apple iOS and Google Android.

Major file systems include the following:

File allocation table (FAT) is supported by the Microsoft Windows OS. FAT Is considered simple and reliable, and it is modified after legacy file systems. FAT was designed in 1977 for floppy disks, but was later adapted for hard disks. While efficient and compatible with most current OSes, FAT cannot match the performance and scalability of more modern file systems.

Global file system (GFS) is a file system for the Linux OS, and it is a shared disk file system. GFS offers direct access to shared block storage and can be used as a local file system.

GFS2 is an updated version with features not included in the original GFS, such as an updated metadata system. Under the terms of the GNU General Public License, both the GFS and GFS2 file systems are available as free software.

Hierarchical file system (HFS) was developed for use with Mac operating systems. HFS can also be referred to as Mac OS Standard, and it was succeeded by Mac OS Extended. Originally introduced in 1985 for floppy and hard disks, HFS replaced the original Macintosh file system. It can also be used on CD- ROMs.

The NT file system -- also known as the **New Technology File System (NTFS)** -- is the **default file system for Windows products from Windows NT 3.1 OS onward. Improvements from the previous FAT file system include better metadata support, performance and use of disk space. NTFS is also supported in the Linux OS through a free, open-source NTFS driver. Mac OSes have read- only support for NTFS.**

Universal Disk Format (UDF) is a vendor-neutral file system used on optical media and DVDs. UDF replaces the ISO 9660 file system and is the official filesystem for DVD video and audio as chosen by the DVD Forum.

File system vs. DBMS

Like a file system, a database management system (DBMS) efficiently stores data that can be updated and retrieved. The two are not interchangeable, however. While a file system stores unstructured, often unrelated files, a DBMS is used to store and manage structured, related data.

A DBMS creates and defines the restraints for a database. A file system allows access to single files at a time and addresses each file individually. Because of this, functions such as redundancy are performed on an individual level, not by the file system itself. This makes a file system a much less consistent form of data storage than a DBMS, which maintains one repository of data that is defined once.

The centralized structure of a DBMS allows for easier file sharing than a file system and prevents anomalies that can occur when separate changes are made to files in a file system.

There are methods to protect files in a file system, but for heavy-duty security, a DBMS is the way to go. Security in a file system is determined by the OS, and it can be difficult to maintain over time as files are accessed and authorization is granted to users.

A DBMS keeps security constraints high, relying on password protection, encryption and limited authorization. More security does result in more obstacles when retrieving data, so in terms of general, simple-to-use file storage and retrieval, a file system may be preferred.

File systems definition evolves

While previously referring to physical, paper files, the term *file system* was used to refer to digital files as early as 1961. By 1964, it had entered general use to refer to computerized file systems.

The term *file system* can also refer to the part of an OS or an add-on program that supports a file system. Examples of such add-on file systems include the Network File System (NFS) and the Andrew File System (AFS).

In addition, the term has evolved to refer to the hardware used for nonvolatile storage, the software application that controls the hardware and architecture of both hardware and software

EXPERIMENT -7

Implement the commands that is used for Creating and Manipulating files: cat, cp, mv, rm, ls, and its options, touch and their options, which is, where is, what is.

The basics of the Unix file system.

We will also cover the commands that are used to work with the file system such as



1. **touch:** Create a new file or update its timestamp.
 - **Syntax:** touch [OPTION]...[FILE]
 - **Example:** Create empty files called 'file1' and 'file2'
\$ touch file1 file2
2. **cat:** Concatenate files and print to stdout
 - **Syntax:** cat [OPTION]...[FILE]
 - **Example:** Create file1 with entered content
\$ cat > file1
Hello
3. **cp:** Copy files
 - **Syntax:** cp [OPTION]source destination
 - **Example:** Copies the content from file1 to file2 and the contents of file1 are retained
\$ cp file1 file2

4. **mv:** Move files or rename files
 - **Syntax:** mv [OPTION] source destination
 - **Example:** Create empty files called 'file1' and 'file2'
\$ mv file1 file2
5. **rm:** Remove files and directories
 - **Syntax:** rm [OPTION]...[FILE]
 - **Example:** Delete file1
\$ rm file1
6. **mkdir:** Make a directory
 - **Syntax:** mkdir [OPTION] directory
 - **Example:** Create a directory
called dir1 \$ mkdir dir1
7. **rmdir:** Remove a directory
 - **Syntax:** rmdir [OPTION] directory
 - **Example:** Create empty files called 'file1' and 'file2'
\$ rmdir dir1
8. **cd:** Change directory
 - **Syntax:** cd [OPTION] directory
 - **Example:** Change working directory to
dir1 \$ cd dir1
9. **pwd:** Print the present working directory
 - **Syntax:** pwd[OPTION]
 - **Example:** Print 'dir1' if a current working directory is dir1
\$ pwd

EXPERIMENT -8

Implement Directory oriented commands: `cs`, `pwd`, `mkdir`, `rmdir`

Managing Directories: `mkdir`, `rmdir`, `ls`, `cd`, and `pwd`

You can create and remove your own directories, as well as change your working directory, with the `mkdir`, `rmdir`, and `cd` commands. Each of these commands can take as its arguments the pathname for a directory. The `pwd` command displays the absolute pathname of your working directory. In addition to these commands, the special characters represented by a single dot, a double dot, and a tilde can be used to reference the working directory, respectively. Taken together, these commands enable you to manage your directories. You can create nested directories, move from one directory to another, and use pathnames to reference any of your directories. Those commands commonly used to manage directories are listed in Table 10-3.

Table 10-3: Directory Commands	
Command	Execution
<code>mkdir directory</code>	Creates a directory.
<code>rmdir directory</code>	Erases a directory.
<code>ls -F</code>	Lists directory name with a preceding slash.
<code>ls -R</code>	Lists working directory as well as all subdirectories.
<code>cd directoryname</code>	Changes to the specified directory, making it the working directory. <code>cd</code> without a directory name changes back to the home directory: <code>\$ cd reports</code>
<code>Pwd</code>	Displays the pathname of the working directory.
<code>directory name/filename</code>	A slash is used in pathnames to separate each directory name. In the case of pathnames for files, a slash separates the preceding directory names from the filename.
<code>..</code>	References the parent directory. You can use it as an argument or as part of a pathname: <code>\$ cd ..</code> <code>\$ mv ../larisa oldletters</code>
Table 10-3: Directory Commands	
Command	Execution

.	References the working directory. You can use it as an argument or as part of a pathname: \$ ls .
~/pathname	The tilde is a special character that represents the pathname for the home directory. It is useful when you need to use an absolute pathname for a file or directory: \$ cp monday ~/today

Creating and Deleting Directories .

You create and remove directories with the **mkdir** and **rmdir** commands. In either case, you can also use pathnames for the directories. In the next example, the user creates the directory **reports**. Then the user creates the directory **letters** using a pathname:

\$ mkdir reports

\$ mkdir /home/chris/letters

You can remove a directory with the **rmdir** command followed by the directory name. In the next example, the user removes the directory **reports** with the **rmdir** command:

\$ rmdir reports

To remove a directory and all its subdirectories, you use the **rm** command with the **-r** option. This is a very powerful command and could easily be used to erase all your files. You will be prompted for each file. To simply remove all files and subdirectories without prompts, add the **-f** option. The following examples delete the **reports** directory and all its subdirectories: **rm -rf reports**

Display Directory Contents

You have seen how to use the **ls** command to list the files and directories within your working directory. To distinguish between file and directory names, however, you need to use the **ls** command with the **-F** option. A slash is then placed after each directory name in the list.

\$ ls

Weather reports letters

\$ ls -F

Weather reports/ letters/

The **ls** command also takes as an argument any directory name or directory pathname. This enables you to list the files in any directory without first having to change to that directory. In the next example, the **ls** command takes as its argument the name of a directory, **reports**. Then the **ls** command is executed again, only this time the absolute pathname of **reports** is used.

\$ ls reports

monday tuesday

\$ ls /home/chris/reports

Monday Tuesday

\$

Moving Through directories

The **cd** command takes as its arguments the name of the directory to which you want to change. The name of the directory can be the name of a subdirectory in your working directory or the full pathname of any directory on the system. If you want to change back to your home directory, you only need to enter the **cd** command by itself, without a filename argument.

```
$ cd prop  
$ pwd  
/home/Dylan/prop
```

Referencing the Parent Directory

A directory always has a parent (except, of course, for the root). For example, in the preceding listing, the parent for **thankyou** is the **letters** directory. When a directory is created, two entries are made: one represented with a dot (**.**), and the other represented by double dots (**..**). The dot represents the pathnames of the directory, and the double dots represent the pathname of its parent directory. Double dots, used as an argument in a command, reference a parent directory. The single dot references the directory itself.

You can use the single dot to reference your working directory, instead of using its pathname. For example, to copy a file to the working directory retaining the same name, the dot can be used in place of the working directory's pathname. In the next example, the user copies the weather file from the **chris** directory to the **reports** directory.

The **reports** directory is the working directory and can be represented with the single dot.

```
$ cd reports  
$ cp /home/chris/weather
```

The **..** symbol is often used to reference files in the present directory. In the next example, the **cat** command displays the weather file in the parent directory. The pathname for the **..** symbol followed by a slash and the filename.

```
$ cat ../weather  
raining and warm
```

EXPERIMENT -9

Implement File using diff, cmp, comm the different ways involved for comparing two files.

The files comparison command helps us to compare the files and find the similarities and difference between these files. The different file comparison commands used in Unix are cmp, comm, diff, dircmp, and uniq.



Different ways of comparing two files in Unix

1. **cmp:** This command is used to compare two files character by character.
 - **Syntax:** cmp [option] file1 file2
 - **Example:** Add write permission for user, group and others for file1. \$ cmp file1 file2
2. **comm:** This command is used to compare two sorted files.
 - **Syntax:** comm [option] file1 file2
 - One set of options allow selection of 'columns' to suppress. -1: suppress lines unique to file1 (column 1)
-2: suppress lines unique to file2 (column 2)
-3: suppress lines common to file1 and file2 (column 3)
 - **Example:** Only show column -3 that contains lines common between file1 and file2
\$ comm -12 file1 file2
3. **diff:** This command is used to compare two files line by line.
 - **Description:** The output indicates how the lines in each file are different, and the steps involved to change file1 to file2. The 'patch' command can be used to make the suggested changes. The output is formatted as blocks of:

Changes commands

```
< lines from file1
--
> lines from file2
```

The change commands are in the format [range][acd][range]. The range on the left may be a lone number or a comma-separated range of line numbers referring to file1, and the range on the right similarly refer to file2. The character in the middle indicates the action i.e., add, change or delete.

- ‘LaR’ – Add lines in range ‘R’ from file2 after line ‘L’ in file1.
- ‘FcT’ – Change lines in range ‘F’ of file1 to lines in range ‘T’ of file2.
- ‘RdL’ – Delete lines in range ‘R’ from file1 that would have appeared at line ‘L’ in file2
- **Syntax:** diff [option] file1 file2
- **Example:** Add write permission for user, group and others for file1
\$ diff file1 file2

4. **diff:** This command is used to compare the contents of directories.

- **Description:** This command works older versions of Unix. In order to compare the directories in the newer versions of Unix, we can use diff -r
- **Syntax:** diff [option] dir1 dir2
- **Example:** Compare contents of dir1 dir2
\$ diff dir1 dir2

5. **uniq:** This command is used to filter the repeated lines in a file which are adjacent to each other.

- **Syntax:** uniq [option] [input [output]]
- **Example:** Omit repeated lines which are adjacent to each other in file1 and print the repeated lines only once
\$ using file1