A.CPU SCHEDULING

CPU scheduling is a fundamental aspect of operating systems (OS) that deals with the efficient allocation of CPU resources to multiple processes. In a multitasking environment where several processes compete for CPU time, the scheduler decides which process gets to run next. The goal is to optimize system performance, throughput, and response time while ensuring fairness and maximizing resource utilization.

There are various CPU Scheduling Algorithms which are explained below.

1. First-Come, First-Served (FCFS) Scheduling

2. Shortest-Job-Next (SJN) Scheduling

3. Priority Scheduling

4. Shortest Remaining Time

5. Round Robin(RR) Scheduling

CPU Scheduling Algorithms with C Code:

1. First-Come, First-Served (FCFS) Scheduling

FCFS, or First-Come, First-Served, is one of the simplest CPU scheduling algorithms used in operating systems. In FCFS, the processes are executed in the order they arrive in the ready queue. The scheduler selects the process at the front of the queue and allocates the CPU to it until the process completes or gets blocked. FCFS is easy to implement and understand but can lead to poor average waiting times, especially if long processes arrive first, causing shorter processes to wait extensively.

C Code:

// C++ program for implementation of FCFS

#include<iostream>

using namespace std;

// Function to find the waiting time for all

void findWaitingTime(int processes[], int n,

int bt[], int wt[])

{

// waiting time for first process is 0

wt[0] = 0;

// calculating waiting time

for (int i = 1; i < n ; i++ )

wt[i] = bt[i-1] + wt[i-1] ;

}

// Function to calculate turn around time

void findTurnAroundTime( int processes[], int n,

int bt[], int wt[], int tat[])

{

// calculating turnaround time by adding

// bt[i] + wt[i]

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

tat[i] = bt[i] + wt[i];

}

//Function to calculate average time

void findavgTime( int processes[], int n, int bt[])

{

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

//Function to find waiting time of all processes

findWaitingTime(processes, n, bt, wt);

//Function to find turn around time for all processes

findTurnAroundTime(processes, n, bt, wt, tat);

//Display processes along with all details

cout << "Processes "<< " Burst time "

<< " Waiting time " << " Turn around time\n";

// Calculate total waiting time and total turn

// around time

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

{

total\_wt = total\_wt + wt[i];

total\_tat = total\_tat + tat[i];

cout << " " << i+1 << "\t\t" << bt[i] <<"\t "

<< wt[i] <<"\t\t " << tat[i] <<endl;

}

cout << "Average waiting time = "

<< (float)total\_wt / (float)n;

cout << "\nAverage turn around time = "

<< (float)total\_tat / (float)n;

}

// Driver code

int main()

{

//process id's

int processes[] = { 1, 2, 3};

int n = sizeof processes / sizeof processes[0];

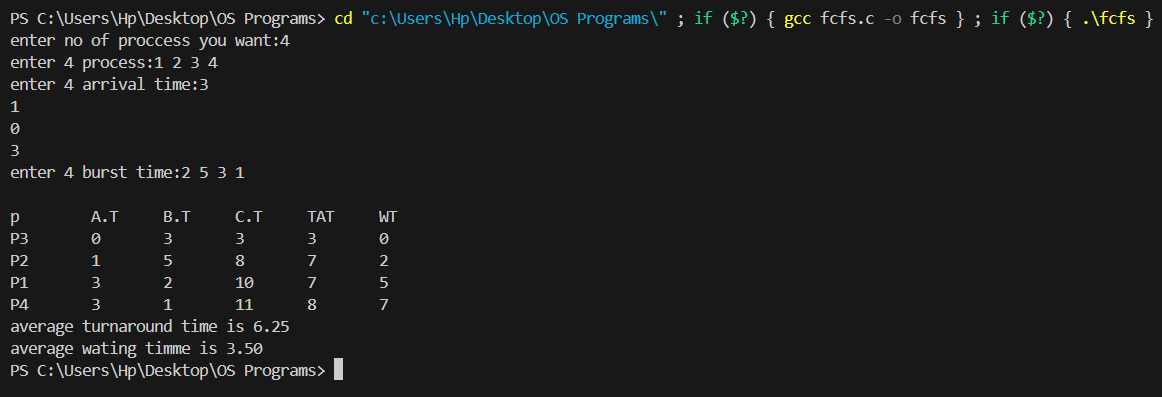
//Burst time of all processes

int burst\_time[] = {10, 5, 8};

findavgTime(processes, n, burst\_time);

return 0;

}



2. Shortest-Job-Next (SJN) Scheduling

SJN, or Shortest Job Next, also known as Shortest Job First (SJF), is a CPU scheduling algorithm that selects the process with the smallest execution time next. It prioritizes shorter jobs over longer ones to minimize the average waiting time and improve system throughput.In SJN scheduling, the scheduler examines all the available processes and selects the one with the shortest burst time (execution time) to execute next. If a new process arrives with a shorter burst time than the currently running process, the scheduler preempts the running process and switches to the new one.

C Code:

#include<stdio.h>

struct Process {

int id;

int arrival\_time;

int burst\_time;

int waiting\_time;

int turnaround\_time;

int completed;

};

void findWaitingTime(struct Process processes[], int n, int waiting\_time[]) {

int remaining\_time[n];

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

remaining\_time[i] = processes[i].burst\_time;

int completed\_processes = 0;

int current\_time = 0;

while (completed\_processes < n) {

int shortest\_job = -1;

int shortest\_time = 9999;

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

if (processes[i].arrival\_time <= current\_time && !processes[i].completed && remaining\_time[i] < shortest\_time) {

shortest\_job = i;

shortest\_time = remaining\_time[i]; }

}

if (shortest\_job == -1) {

current\_time++;

continue;

}

waiting\_time[shortest\_job] = current\_time - processes[shortest\_job].arrival\_time;

current\_time += remaining\_time[shortest\_job];

processes[shortest\_job].completed = 1;

completed\_processes++;

}

}

void findTurnaroundTime(struct Process processes[], int n, int waiting\_time[], int turnaround\_time[]) {

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

turnaround\_time[i] = processes[i].burst\_time + waiting\_time[i];

}

void findAverageTime(struct Process processes[], int n) {

int waiting\_time[n], turnaround\_time[n];

findWaitingTime(processes, n, waiting\_time);

findTurnaroundTime(processes, n, waiting\_time, turnaround\_time);

int total\_waiting\_time = 0, total\_turnaround\_time = 0;

printf("Process ID\tBurst Time\tArrival Time\tWaiting Time\tTurnaround Time\n");

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

total\_waiting\_time += waiting\_time[i];

total\_turnaround\_time += turnaround\_time[i];

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst\_time, processes[i].arrival\_time, waiting\_time[i], turnaround\_time[i]);

}

float avg\_waiting\_time = (float)total\_waiting\_time / n; float avg\_turnaround\_time = (float)total\_turnaround\_time / n; printf("Average Waiting Time: %.2f\n", avg\_waiting\_time);

printf("Average Turnaround Time: %.2f\n", avg\_turnaround\_time);

}

int main() {

int n;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process processes[n];

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

printf("Enter arrival time and burst time for process %d: ", i + 1);

scanf("%d %d", &processes[i].arrival\_time, &processes[i].burst\_time);

processes[i].id = i + 1;

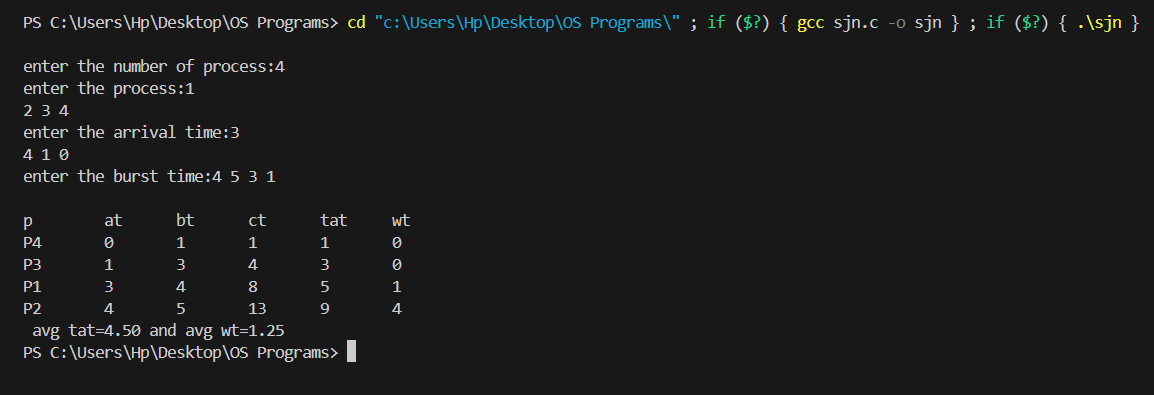
processes[i].completed = 0;

}

findAverageTime(processes, n);

return 0;

}



3. Priority Scheduling

Priority scheduling is a CPU scheduling algorithm where each process is assigned a priority. The scheduler selects the process with the highest priority for execution. In case of ties, other factors like arrival time or process ID may be used to break the tie.

C Code:

#include <stdio.h>

struct Process {

int id;

int arrival\_time;

int burst\_time;

int priority;

int completion\_time;

int turnaround\_time;

int waiting\_time;

};

void calculateTimes(struct Process processes[], int n) {

int currentTime = 0;

int completed = 0;

while (completed < n) {

int highestPriority = -1;

int nextProcess = -1;

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

if (processes[i].arrival\_time <= currentTime && processes[i].completion\_time == -1) {

if (highestPriority == -1 || processes[i].priority < highestPriority) {

highestPriority = processes[i].priority;

nextProcess = i;

}

}

} if (nextProcess != -1) {

processes[nextProcess].completion\_time = currentTime + processes[nextProcess].burst\_time;

processes[nextProcess].turnaround\_time = processes[nextProcess].completion\_time - processes[nextProcess].arrival\_time;

processes[nextProcess].waiting\_time = processes[nextProcess].turnaround\_time - processes[nextProcess].burst\_time;

currentTime = processes[nextProcess].completion\_time;

completed++;

} else {

currentTime++;

}

}

}

void displayResults(struct Process processes[], int n) {

printf("Process\tArrival Time\tBurst Time\tPriority\tCompletion Time\tTurnaround Time\tWaiting Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrival\_time, processes[i].burst\_time, processes[i].priority,

processes[i].completion\_time, processes[i].turnaround\_time, processes[i].waiting\_time);

}

}

int main() {

int n;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process processes[n];

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

printf("Enter arrival time, burst time and priority for process %d: ", i + 1);

scanf("%d %d %d", &processes[i].arrival\_time, &processes[i].burst\_time, &processes[i].priority);

processes[i].id = i + 1;

processes[i].completion\_time = -1;

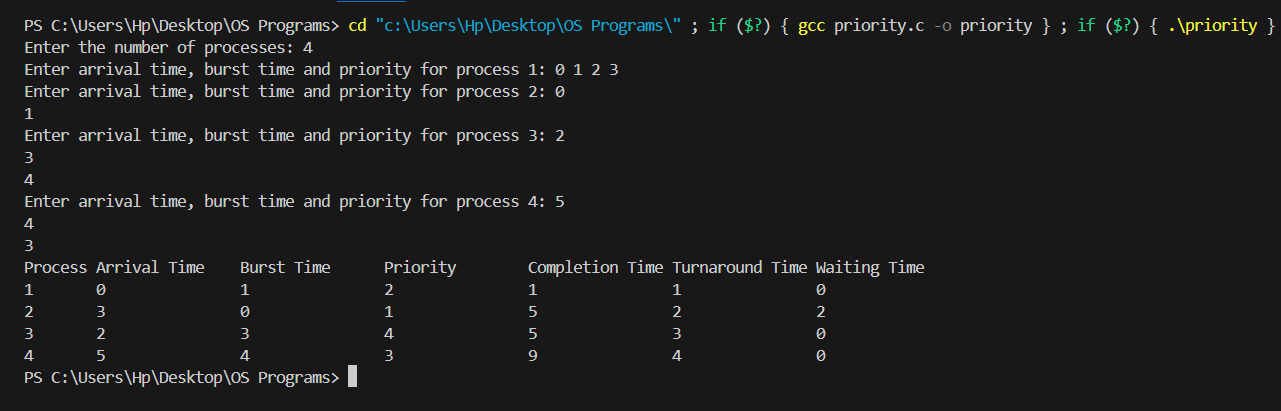
}

calculateTimes(processes, n);

displayResults(processes, n);

return 0;

}



4. Shortest Remaining Time

In the context of operating systems, SRT stands for Shortest Remaining Time, also known as Shortest Remaining Time First (SRTF) or Shortest Job Next (SJN) with preemption. In SRT scheduling, the scheduler selects the process with the smallest remaining burst time (time needed to complete execution) to run next. If a new process arrives with a shorter remaining burst time than the currently running process, the scheduler preempts the running process and switches to the new one.

C Code:

#include <stdio.h>

struct Process {

int id;

int arrival\_time;

int burst\_time;

int remaining\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

};

calculateTimes(struct Process processes[], int n) {

int currentTime = 0;

int completed = 0;

int shortestRemainingTimeIndex;

while (completed < n) {

shortestRemainingTimeIndex = -1;

int shortestRemainingTime = \_\_INT\_MAX\_\_;

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

if (processes[i].arrival\_time <= currentTime && processes[i].remaining\_time > 0) {

if (processes[i].remaining\_time < shortestRemainingTime) {

shortestRemainingTime = processes[i].remaining\_time;

shortestRemainingTimeIndex = i;

} }

if (shortestRemainingTimeIndex != -1) {

processes[shortestRemainingTimeIndex].remaining\_time--;

if (processes[shortestRemainingTimeIndex].remaining\_time == 0) {

completed++;

processes[shortestRemainingTimeIndex].completion\_time = currentTime + 1;

processes[shortestRemainingTimeIndex].turnaround\_time = processes[shortestRemainingTimeIndex].completion\_time - processes[shortestRemainingTimeIndex].arrival\_time; processes[shortestRemainingTimeIndex].waiting\_time = processes[shortestRemainingTimeIndex].turnaround\_time - processes[shortestRemainingTimeIndex].burst\_time;

} }

currentTime++;

}}

void displayResults(struct Process processes[], int n) {

printf("Process\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrival\_time, processes[i].burst\_time,

processes[i].completion\_time, processes[i].turnaround\_time, processes[i].waiting\_time);

}}

int main() {

int n;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process processes[n];

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

printf("Enter arrival time and burst time for process %d: ", i + 1);

scanf("%d %d", &processes[i].arrival\_time, &processes[i].burst\_time);

processes[i].id = i + 1;

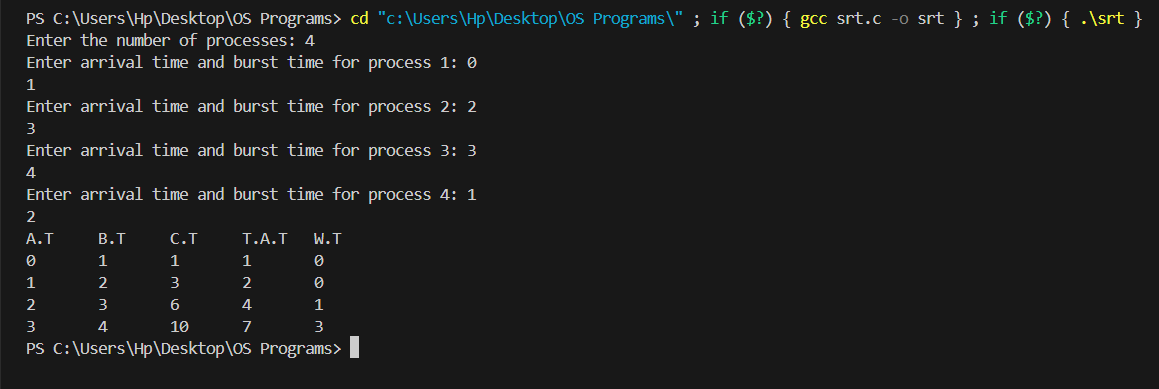
processes[i].remaining\_time = processes[i].burst\_time; }

calculateTimes(processes, n);

displayResults(processes, n);

return 0;

}



5. Round Robin(RR) Scheduling

Round Robin (RR) scheduling is a CPU scheduling algorithm that allocates CPU time to multiple processes in a fair and balanced manner. In RR scheduling, each process is assigned a fixed time quantum or time slice. The scheduler cycles through the ready queue, allocating each process the CPU for a time quantum, and then moving on to the next process in the queue. If a process does not complete within its time quantum, it is preempted, and the scheduler moves it to the end of the ready queue, allowing other processes to run.

C Code:

#include<stdio.h>

#include<conio.h>

int

main()

{

// initlialize the variable name

int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];

float avg\_wt, avg\_tat;

printf(" Total number of process in the system: ");

scanf("%d", &NOP);

y = NOP; // Assign the number of process to variable y

// Use for loop to enter the details of the process like Arrival time and the Burst Time

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

{

printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);

printf(" Arrival time is: \t"); // Accept arrival time

scanf("%d", &at[i]);

printf(" \nBurst time is: \t"); // Accept the Burst time

scanf("%d", &bt[i]);

temp[i] = bt[i]; // store the burst time in temp array

}

// Accept the Time qunat

printf("Enter the Time Quantum for the process: \t");

scanf("%d", &quant);

// Display the process No, burst time, Turn Around Time and the waiting time

printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");

for(sum=0, i = 0; y!=0; )

{

if(temp[i] <= quant && temp[i] > 0) // define the conditions

{

sum = sum + temp[i];

temp[i] = 0;

count=1;

}

else if(temp[i] > 0)

{

temp[i] = temp[i] - quant;

sum = sum + quant;

}

if(temp[i]==0 && count==1)

{

y--; //decrement the process no.

printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);

wt = wt+sum-at[i]-bt[i];

tat = tat+sum-at[i];

count =0;

}

if(i==NOP-1)

{

i=0;

}

else if(at[i+1]<=sum)

{

i++;

}

else

{

i=0;

}

}

// represents the average waiting time and Turn Around time

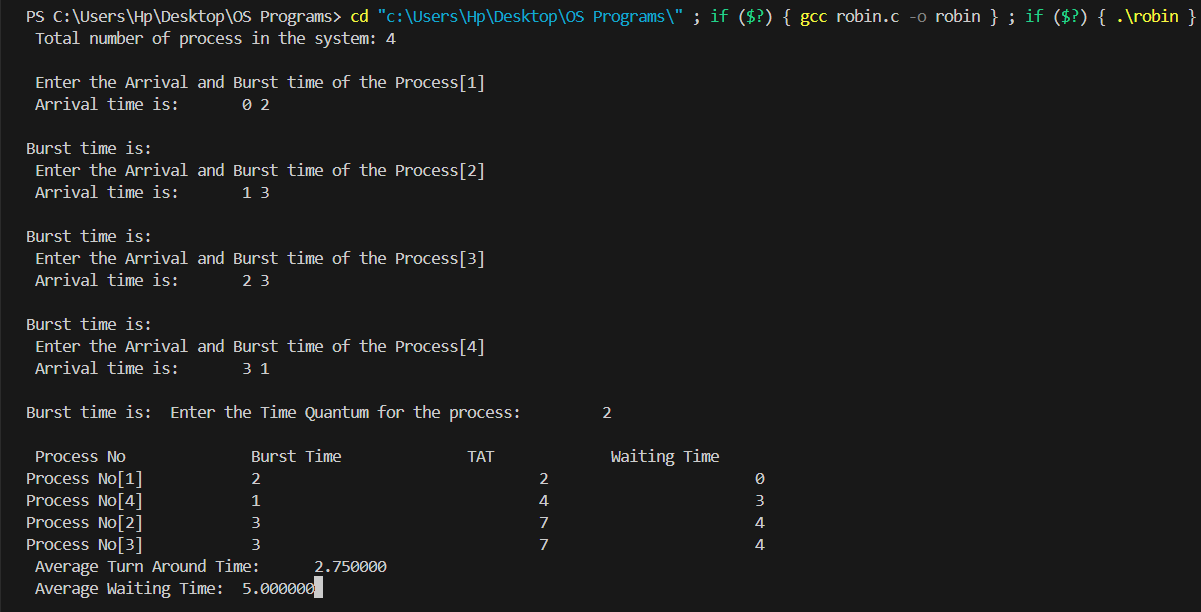
avg\_wt = wt \* 1.0/NOP;

avg\_tat = tat \* 1.0/NOP;

printf("\n Average Turn Around Time: \t%f", avg\_wt);

printf("\n Average Waiting Time: \t%f", avg\_tat);

getch();



B.Memory Management

Memory management in operating systems involves various mechanisms and techniques to efficiently manage system memory (RAM) to meet the demands of running processes and ensure optimal system performance. Operating systems allocate memory to processes dynamically. When a process is created, the OS allocates memory for its code, data, and stack segments. Memory allocation can be done using various techniques such as contiguous allocation, paging, or segmentation.

There are various Memory Management Algorithms in OS. Some of them are explained below:

1. First In First Out (FIFO) Page Replacement

2. Least Recently Used

Memory Management Algorithms with C program:

1. First In First Out (FIFO) Page Replacement

First-In-First-Out (FIFO) is one of the simplest page replacement algorithms used in memory management systems, particularly in virtual memory systems. In FIFO page replacement, the page that was brought into memory earliest is the one that is replaced when a new page needs to be brought in. It follows the principle of a queue: the page that has been in memory for the longest time is at the front of the queue, and the page that was brought in most recently is at the end.

C Code:

// C program for FIFO page replacement algorithm

#include<stdio.h>

int main()

{

int incomingStream[] = {4, 1, 2, 4, 5};

int pageFaults = 0;

int frames = 3;

int m, n, s, pages;

pages = sizeof(incomingStream)/sizeof(incomingStream[0]);

printf("Incoming \t Frame 1 \t Frame 2 \t Frame 3");

int temp[frames];

for(m = 0; m < frames; m++)

{

temp[m] = -1; }

for(m = 0; m < pages; m++)

{

s = 0;

for(n = 0; n < frames; n++)

{

if(incomingStream[m] == temp[n])

{

s++;

pageFaults--;

}

}

pageFaults++;

if((pageFaults <= frames) && (s == 0))

{

temp[m] = incomingStream[m];

}

else if(s == 0)

{

temp[(pageFaults - 1) % frames] = incomingStream[m];

}

printf("\n");

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

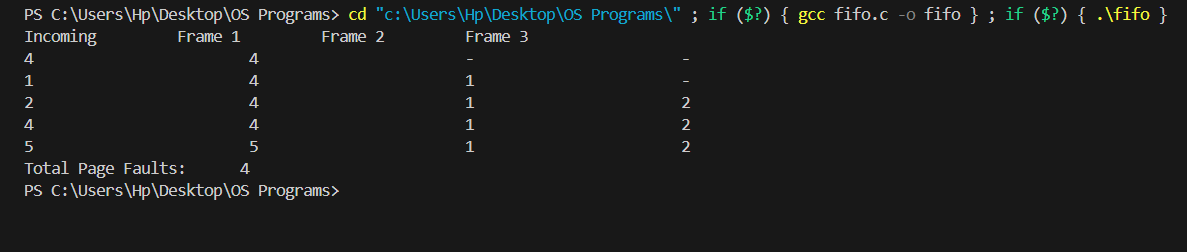
printf(" - \t\t\t");

}

}

printf("\nTotal Page Faults:\t%d\n", pageFaults);

return 0;

}

2. Least Recently Used

Least Recently Used (LRU) is a popular page replacement algorithm used in memory management systems, particularly in virtual memory systems. In LRU page replacement, the page that has not been used for the longest period of time is the one that is replaced when a new page needs to be brought into memory. The underlying principle is that pages that have not been accessed recently are less likely to be needed in the near future.

C Code:

#include<stdio.h>

main()

{

int q[20],p[50],c=0,c1,d,f,i,j,k=0,n,r,t,b[20],c2[20];

printf("Enter no of pages:");

scanf("%d",&n);

printf("Enter the reference string:");

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

scanf("%d",&p[i]);

printf("Enter no of frames:");

scanf("%d",&f);

q[k]=p[k];

printf("\n\t%d\n",q[k]);

c++;

k++;

for(i=1;i<n;i++)

{

c1=0;

for(j=0;j<f;j++)

{

if(p[i]!=q[j])

c1++;

}

if(c1==f)

{ c++;

if(k<f)

{ q[k]=p[i];

k++;

for(j=0;j<k;j++)

printf("\t%d",q[j]);

printf("\n");

}

else

{

for(r=0;r<f;r++)

{

c2[r]=0;

for(j=i-1;j<n;j--)

{

if(q[r]!=p[j])

c2[r]++;

else

break;

}

}

for(r=0;r<f;r++)

b[r]=c2[r];

for(r=0;r<f;r++)

{

for(j=r;j<f;j++)

{

if(b[r]<b[j])

{

t=b[r];

b[r]=b[j];

b[j]=t;

}

}

}

for(r=0;r<f;r++)

{

if(c2[r]==b[0])

q[r]=p[i];

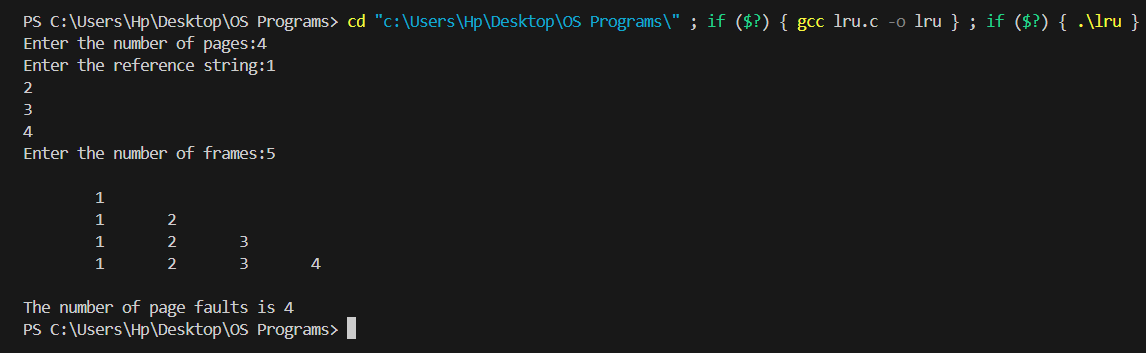
printf("\t%d",q[r]);

}

printf("\n");

} } }

printf("\nThe no of page faults is %d",c); }



C. Disk Scheduling

Disk scheduling in operating systems refers to the process of deciding the order in which disk I/O requests are serviced by the disk drive. Disk scheduling algorithms aim to optimize disk performance by minimizing seek time and maximizing throughput. Disk scheduling algorithms aim to balance between minimizing seek time, maximizing throughput, and ensuring fairness in servicing I/O requests. The choice of algorithm depends on factors such as workload characteristics, disk hardware, and performance requirements.

Top of Form

Some Disk Scheduling Algorithms are explained below:

1. First Come First Serve

2. Shortest Seek Time First (SSTF)

Disk Scheduling Algorithms with C code explained below:

1. First Come First Serve

FCFS disk scheduling is straightforward to implement and understand. However, it may not always result in optimal disk performance. Since FCFS does not consider the physical location of the data on the disk or the seek time required to move the disk head, it can lead to increased average seek time and reduced overall disk throughput, especially in scenarios with mixed workload patterns.

C Code:

#include <bits/stdc++.h>

using namespace std;

int size = 8;

void FCFS(int arr[], int head)

{

int seek\_count = 0;

int distance, cur\_track;

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

cur\_track = arr[i];

// calculate absolute distance

distance = abs(cur\_track - head);

seek\_count += distance;

head = cur\_track;

}cout << "Total number of seek operations = "

<< seek\_count << endl;

cout << "Seek Sequence is" << endl;

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

cout << arr[i] << endl;

}

}

int main()

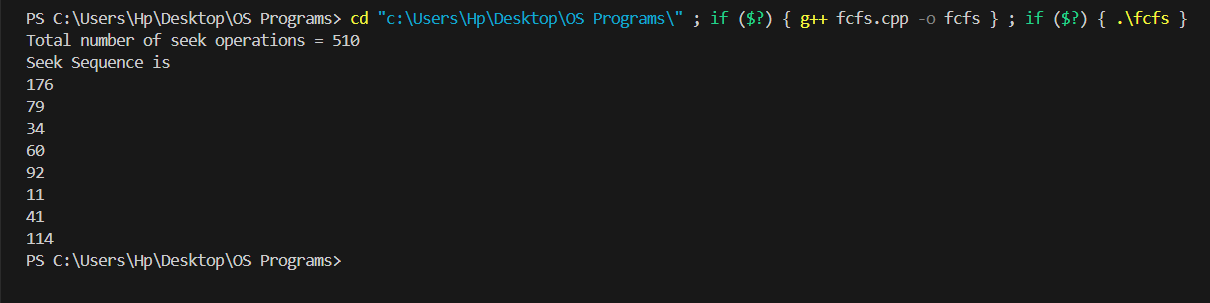
{

int arr[size] = { 176, 79, 34, 60, 92, 11, 41, 114 };

int head = 50;

FCFS(arr, head);

return 0;

}

2. Shortest Seek Time First (SSTF)

SSTF scheduling is more efficient than First-Come, First-Served (FCFS) scheduling because it minimizes the average seek time, leading to improved disk performance and throughput. However, SSTF may suffer from starvation issues if there are requests located far away from the current disk head position, causing nearby requests to be continually serviced while distant requests are delayed indefinitely.

C Code:

#include <bits/stdc++.h>

using namespace std;

void calculatedifference(int request[], int head,

int diff[][2], int n)

{

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

{

diff[i][0] = abs(head - request[i]);

}

}

int findMIN(int diff[][2], int n)

{

int index = -1;

int minimum = 1e9;

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

{

if (!diff[i][1] && minimum > diff[i][0])

{

minimum = diff[i][0];

index = i;

}

}

return index;

}void shortestSeekTimeFirst(int request[],

int head, int n)

{

if (n == 0)

{

return;

}

int diff[n][2] = { { 0, 0 } };

int seekcount = 0;

int seeksequence[n + 1] = {0};

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

{

seeksequence[i] = head;

calculatedifference(request, head, diff, n);

int index = findMIN(diff, n);

diff[index][1] = 1;

seekcount += diff[index][0];

head = request[index];

}

seeksequence[n] = head;

cout << "Total number of seek operations = "

<< seekcount << endl;

cout << "Seek sequence is : " << "\n";

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

{

cout << seeksequence[i] << "\n";

}

}

int main()

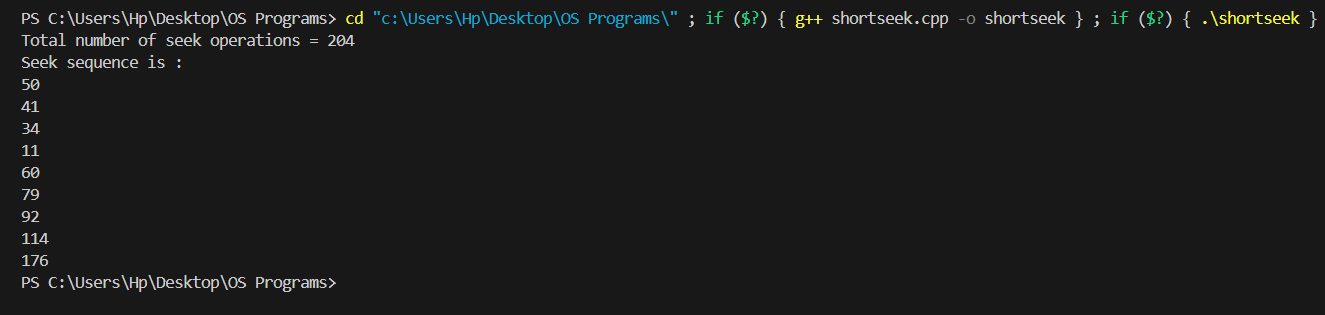
{int n = 8;

int proc[n] = { 176, 79, 34, 60, 92, 11, 41, 114 };

shortestSeekTimeFirst(proc, 50, n);

return 0;

}



D. Dead Lock Avoidance

Deadlock avoidance is a technique used in operating systems to prevent the occurrence of deadlocks, which are situations where two or more processes are unable to proceed because each is waiting for a resource held by the other. Deadlocks can lead to system instability and resource starvation if not properly managed.

One of the algorithm for Dead Lock Avoidance is explained with C program below:

1. Bankers Algorithm

The Banker's algorithm ensures that resource allocations will not lead to a deadlock by carefully managing the allocation of resources and checking for safety before making an allocation. However, it requires advance knowledge of the maximum resource needs of processes, which may not always be available or feasible to obtain. Additionally, the Banker's algorithm may introduce overhead due to the need for frequent safety checks and resource tracking.

C Code:

// Banker's Algorithm

#include <iostream>

using namespace std;

int main()

{

// P0, P1, P2, P3, P4 are the Process names here

int n, m, i, j, k;

n = 5; // Number of processes

m = 3; // Number of resources

int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix

{ 2, 0, 0 }, // P1

{ 3, 0, 2 }, // P2

{ 2, 1, 1 }, // P3

{ 0, 0, 2 } }; // P4

int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix

{ 3, 2, 2 }, // P1

{ 9, 0, 2 }, // P2

{ 2, 2, 2 }, // P3

{ 4, 3, 3 } }; // P4

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

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

for (k = 0; k < n; k++) {

f[k] = 0;

}

int need[n][m];

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

for (j = 0; j < m; j++)

need[i][j] = max[i][j] - alloc[i][j];

}

int y = 0;

for (k = 0; k < 5; k++) {

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

if (f[i] == 0) {

int flag = 0;

for (j = 0; j < m; j++) {

if (need[i][j] > avail[j]){

flag = 1;

break;

}

}

if (flag == 0) {

ans[ind++] = i;

for (y = 0; y < m; y++)

avail[y] += alloc[i][y];

f[i] = 1;

}

} } }

int flag = 1;

// To check if sequence is safe or not

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

{

if(f[i]==0)

{

flag = 0;

cout << "The given sequence is not safe";

break;

}

}

if(flag==1)

{

cout << "Following is the SAFE Sequence" << endl;

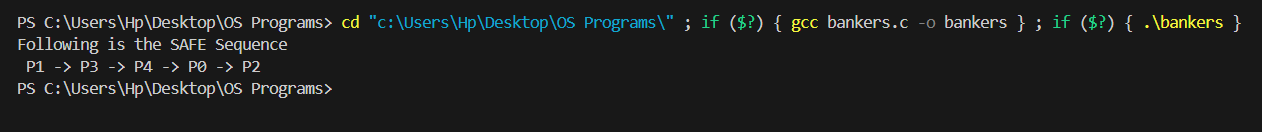
for (i = 0; i < n - 1; i++)

cout << " P" << ans[i] << " ->";

cout << " P" << ans[n - 1] <<endl;

}

return (0);

} 

E. File Management

File management in operating systems involves organizing, storing, and accessing files on storage devices such as hard drives, solid-state drives, or network storage. Effective file management is essential for organizing and accessing data efficiently, ensuring data integrity and security, and supporting various applications and user needs. Different operating systems provide different file management features and capabilities, tailored to their specific design goals, hardware platforms, and user requirements.

1. Sequential file with records

A sequential file with records is a type of file organization where records are stored sequentially, one after another, on the storage medium. Each record in the file is identified by its position or offset within the file, with records arranged in a linear order. sequential files with records offer a simple and straightforward way to store and access data sequentially. While they may not be suitable for applications requiring random or direct access to records, they are well-suited for applications where data is processed in a linear fashion.

C Code:

#include<stdio.h>

#include<conio.h>

main()

{

int n,i,j,b[20],sb[20],t[20],x,c[20][20];

printf("Enter no.of files:");

scanf("%d",&n);

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

{

printf("Enter no. of blocks occupied by file%d",i+1);

scanf("%d",&b[i]);

printf("Enter the starting block of file%d",i+1);

scanf("%d",&sb[i]);

t[i]=sb[i];

for(j=0;j<b[i];j++)

c[i][j]=sb[i]++;

}

printf("Filename\tStart block\tlength\n");

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

printf("%d\t %d \t%d\n",i+1,t[i],b[i]);

printf("Enter file name:");

scanf("%d",&x);

printf("File name is:%d",x);

printf("length is:%d",b[x-1]);

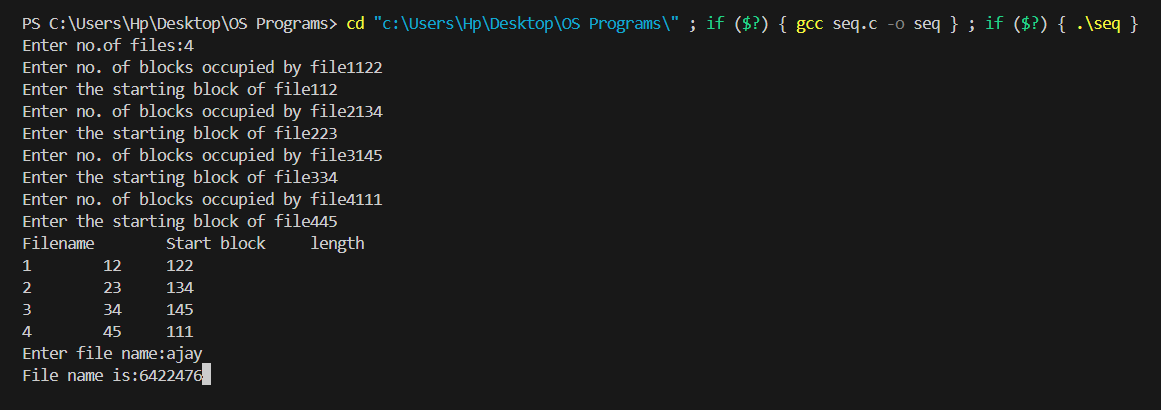
printf("blocks occupied:");

for(i=0;i<b[x-1];i++)

printf("%4d",c[x-1][i]);

getch();

}



F. OS InstallationTop of Form

Operating system installation is the process of setting up an operating system (OS) on a computer or computing device. It involves preparing the system, selecting and booting from installation media such as DVDs or USB drives, initiating the installation process, partitioning and formatting the disk, installing the OS, configuring settings such as language and time zone, and completing post-installation tasks such as installing drivers and software updates. Successful installation ensures that the computer is ready to be used with the newly installed OS, providing a platform for running applications and performing various tasks.

1. Linux Installation Screen Shot

Installing Ubuntu, a popular Linux distribution, involves downloading the Ubuntu ISO from the official website, creating a bootable USB drive with the ISO using software like Rufus or balenaEtcher, and booting the computer from the USB drive. Once booted, users can choose to install Ubuntu alongside an existing OS or perform a clean installation. During installation, users configure language, keyboard layout, disk partitioning, user details, and timezone settings. After installation, users can customize their Ubuntu system with additional software and configurations.

A. Linux

Linux is an open-source operating system renowned for its stability, security and flexibility. Developed by Linus Torvalds in 1991, it has since growth into a diverse ecosystem with various distributions catering to different needs. From servers to smartphones, Linux powers a wide range of devices and is favored by developers and enthusiasts alike for its customization options and extensive software repository. With its robust command-line interface and support for a multitude of programming languages, Linux continues to be a cornerstone of modern computing.

Some of the features of Linux are as follows:

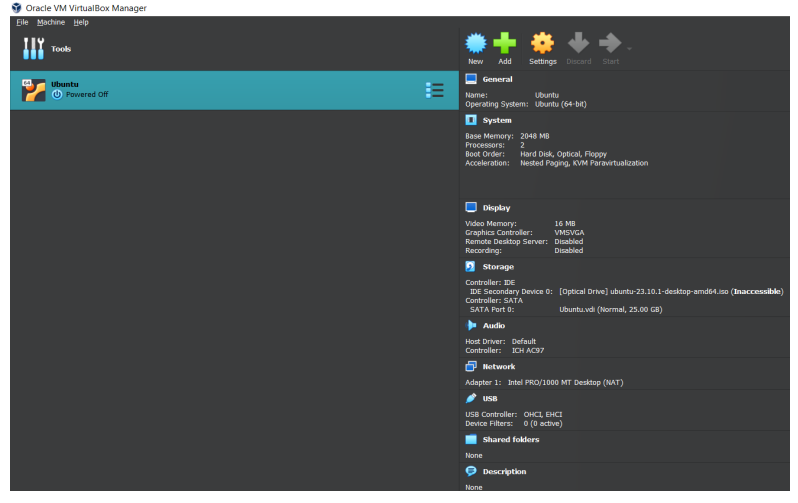
1.Open source

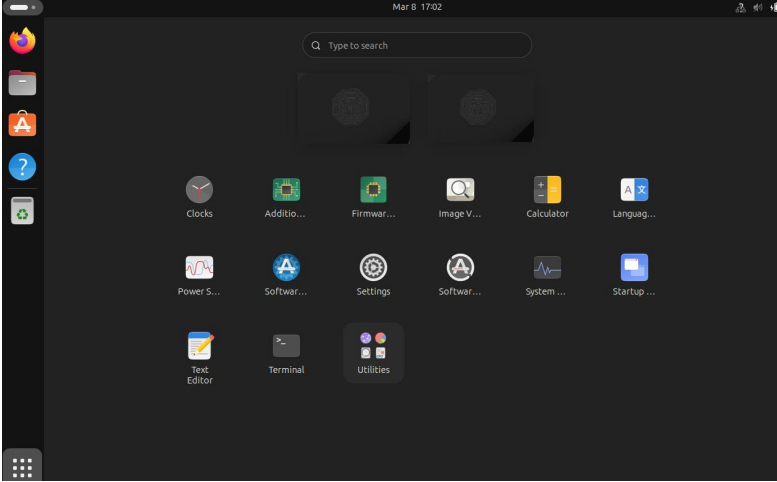
2.Stability and security

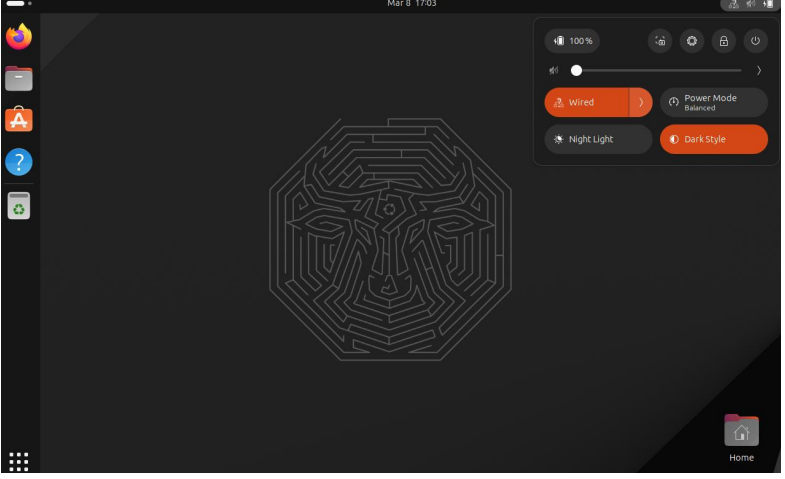
3.Powerful command-line interface

4.Community and support and many more

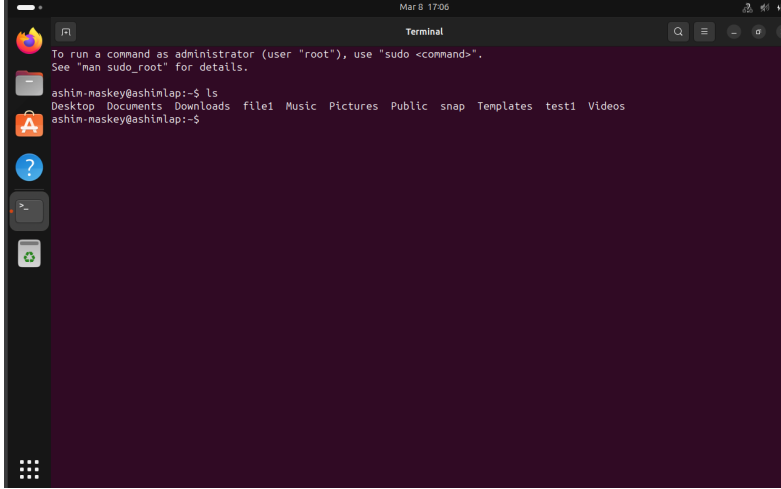
Some screenshots of Linux installation are as follows

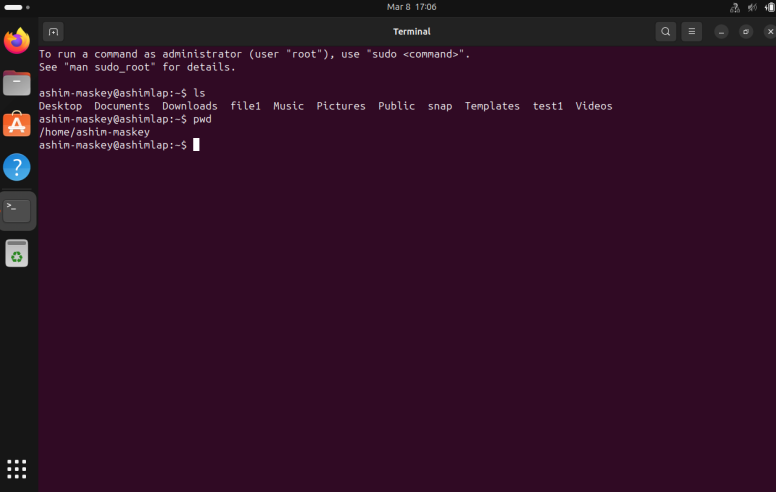


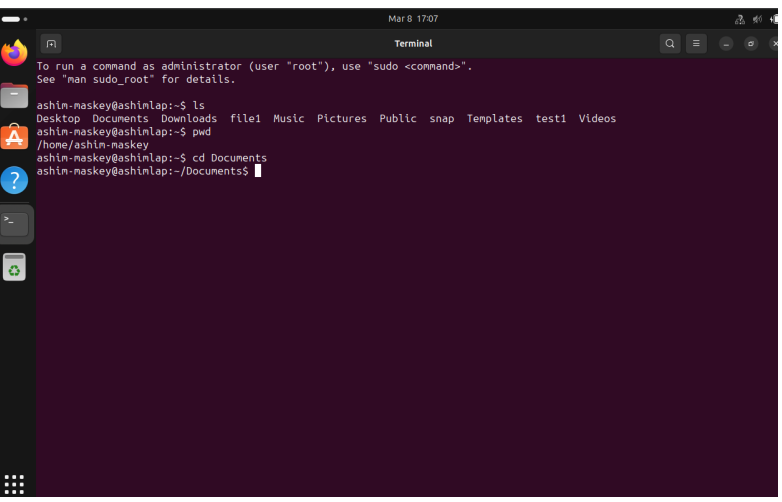




Some of the commands in Linux terminal:-







Top of Form