**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**“JnanaSangama”, Belgaum -590014, Karnataka.**

****

**LAB REPORT**

**on**

**OPERATING SYSTEMS**

***Submitted by***

**PRASANNA KUMAR R (1BM21CS138)**

***in partial fulfillment for the award of the degree of***

**BACHELOR OF ENGINEERING**

***in***

**COMPUTER SCIENCE AND ENGINEERING**



**B.M.S. COLLEGE OF ENGINEERING**

**(Autonomous Institution under VTU)**

**BENGALURU-560019**

**June-2023 to September-2023**

**B. M. S. College of Engineering,**

**Bull Temple Road, Bangalore 560019**

(Affiliated To Visvesvaraya Technological University, Belgaum)

**Department of Computer Science and Engineering**



**CERTIFICATE**

This is to certify that the Lab work entitled “OPERATING SYSTEMS” carried out by **PRASANNA KUMAR R (1BM21CS138),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the academic semester June-2023 to September-2023. The Lab report has been approved as it satisfies the academic requirements in respect of a OPERATING SYSTEMS **(22CS4PCOPS)** work prescribed for the said degree.

Name of the Lab-In charge:Madhavi R.P. Dr. Jyothi S Nayak Designation: Assistant Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

**Index Sheet**

| **Lab Program No.** | **Program Details** | **Page No.** |
| --- | --- | --- |
| 1 | Write a C program to simulate the following non-pre-emptive CPU  scheduling algorithm to find turnaround time and waiting time.   * FCFS * SJF (pre-emptive & Non-pre-emptive) | 6 |
| 2 | Write a C program to simulate the following CPU scheduling  algorithm to find turnaround time and waiting time.   * Priority (pre-emptive & Non-pre-emptive) * Round Robin (Experiment with different quantum sizes for RR   algorithm) | 13 |
| 3 | Write a C program to simulate multi-level queue scheduling  algorithm considering the following scenario. All the processes in the  system are divided into two categories – system processes and user  processes. System processes are to be given higher priority than user  processes. Use FCFS scheduling for the processes in each queue. | 21 |
| 4 | Write a C program to simulate Real-Time CPU Scheduling  algorithms:  a) Rate- Monotonic  b) Earliest-deadline First  c) Proportional scheduling | 29 |
| 5 | Write a C program to simulate producer-consumer problem using  semaphores. | 46 |
| 6 | Write a C program to simulate the concept of Dining-Philosophers problem. | 50 |
| 7 | Write a C program to simulate Bankers algorithm for the purpose of  deadlock avoidance. | 56 |
| 8 | Write a C program to simulate deadlock detection | 61 |
| 9 | Write a C program to simulate the following contiguous memory  allocation techniques  a) Worst-fit  b) Best-fit  c) First-fit | 65 |
| 10 | Write a C program to simulate paging technique of memory  management. | 71 |
| 11 | Write a C program to simulate page replacement algorithms  a) FIFO  b) LRU  c) Optimal | 74 |
| 12 | Write a C program to simulate disk scheduling algorithms  a) FCFS  b) SCAN  c) C-SCAN | 83 |
| 13 | Write a C program to simulate disk scheduling algorithms  a) SSTF  b) LOOK  c) c-LOOK | 83 |

**Course Outcome**

| CO1 | Apply the different concepts and functionalities of Operating System |
| --- | --- |
| CO2 | Analyse various Operating system strategies and techniques |
| CO3 | Demonstrate the different functionalities of Operating System. |
| CO4 | Conduct practical experiments to implement the functionalities of Operating system. |

**Write a C program to simulate the following non-pre-emptive CPU**

**scheduling algorithm to find turnaround time and waiting time.**

**• FCFS**

**• SJF (pre-emptive & Non-pre-emptive)**

**CODE:**

#include <stdio.h>

// Structure to represent a process

struct Process {

int pid;

int arrivalTime;

int burstTime;

int remainingTime;

};

// Function to calculate FCFS scheduling

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

int currentTime = 0;

int totalWaitingTime = 0;

int totalTurnaroundTime = 0;

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

if (currentTime < processes[i].arrivalTime) {

currentTime = processes[i].arrivalTime;

}

totalWaitingTime += currentTime - processes[i].arrivalTime;

totalTurnaroundTime += currentTime + processes[i].burstTime - processes[i].arrivalTime;

currentTime += processes[i].burstTime;

}

printf("FCFS Scheduling:\n");

printf("Average Waiting Time: %.2f\n", (float)totalWaitingTime / n);

printf("Average Turnaround Time: %.2f\n", (float)totalTurnaroundTime / n);

}

// Function to calculate SJF (Non-preemptive) scheduling

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

// Sorting processes based on burst time (shortest first)

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

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

if (processes[j].burstTime > processes[j + 1].burstTime) {

struct Process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

}

int currentTime = 0;

int totalWaitingTime = 0;

int totalTurnaroundTime = 0;

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

if (currentTime < processes[i].arrivalTime) {

currentTime = processes[i].arrivalTime;

}

totalWaitingTime += currentTime - processes[i].arrivalTime;

totalTurnaroundTime += currentTime + processes[i].burstTime - processes[i].arrivalTime;

currentTime += processes[i].burstTime;

}

printf("SJF (Non-preemptive) Scheduling:\n");

printf("Average Waiting Time: %.2f\n", (float)totalWaitingTime / n);

printf("Average Turnaround Time: %.2f\n", (float)totalTurnaroundTime / n);

}

// Function to calculate SJF (Preemptive) scheduling

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

int currentTime = 0;

int totalWaitingTime = 0;

int totalTurnaroundTime = 0;

int completed = 0;

while (completed < n) {

int shortestIndex = -1;

int shortestBurst = -1;

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

if (processes[i].remainingTime > 0 && processes[i].arrivalTime <= currentTime &&

(shortestIndex == -1 || processes[i].remainingTime < shortestBurst)) {

shortestIndex = i;

shortestBurst = processes[i].remainingTime;

}

}

if (shortestIndex == -1) {

currentTime++;

continue;

}

processes[shortestIndex].remainingTime--;

currentTime++;

if (processes[shortestIndex].remainingTime == 0) {

completed++;

totalWaitingTime += currentTime - processes[shortestIndex].arrivalTime - processes[shortestIndex].burstTime;

totalTurnaroundTime += currentTime - processes[shortestIndex].arrivalTime;

}

}

printf("SJF (Preemptive) Scheduling:\n");

printf("Average Waiting Time: %.2f\n", (float)totalWaitingTime / n);

printf("Average Turnaround Time: %.2f\n", (float)totalTurnaroundTime / n);

}

int main() {

int n;

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

scanf("%d", &n);

struct Process processes[n];

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

processes[i].pid = i + 1;

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

scanf("%d", &processes[i].arrivalTime);

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

scanf("%d", &processes[i].burstTime);

processes[i].remainingTime = processes[i].burstTime;

}

int choice;

printf("Select scheduling algorithm:\n");

printf("1. FCFS\n");

printf("2. SJF (Non-preemptive)\n");

printf("3. SJF (Preemptive)\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

fcfs(processes, n);

break;

case 2:

sjfNonPreemptive(processes, n);

break;

case 3:

sjfPreemptive(processes, n);

break;

default:

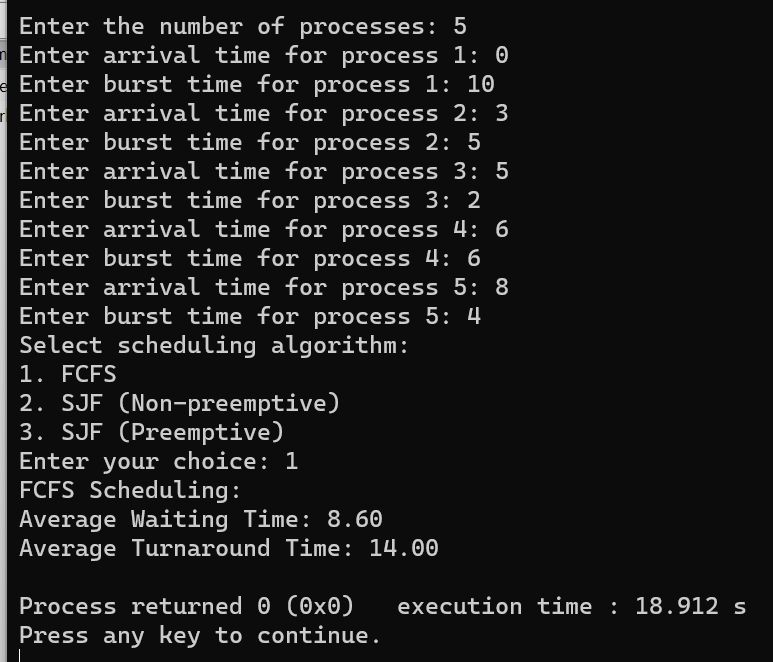
printf("Invalid choice.\n");

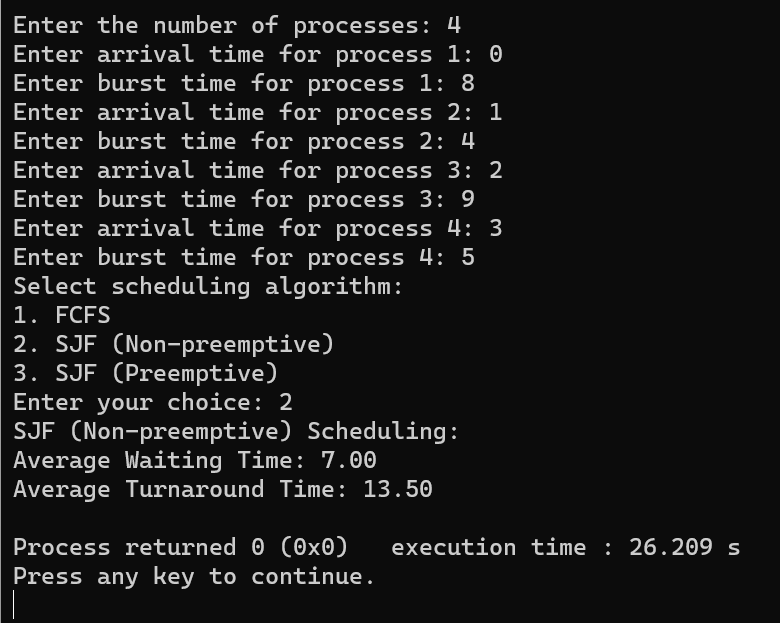
}

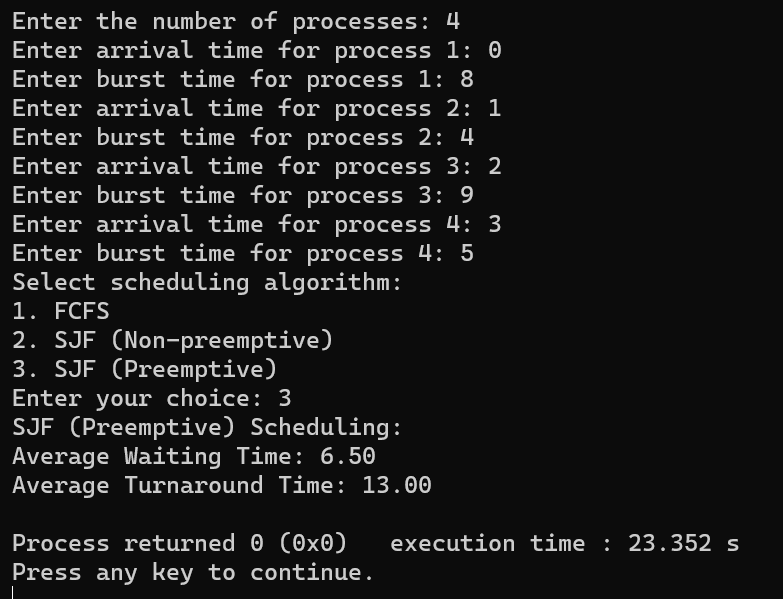
return 0;

}

**Result Screen shot**

****

****

****

**Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.**

* **Priority (pre-emptive & Non-pre-emptive)**
* **Round Robin (Experiment with different quantum sizes for RR algorithm)**

**CODE:**

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

#define MAX\_PROCESSES

struct Process {

int pid;

int arrival\_time;

int burst\_time;

int priority;

int remaining\_time;

int turnaround\_time;

int waiting\_time;

};

void priority\_nonpreemptive(struct Process processes[], int n) {

// Sort the processes based on priority in ascending order

int i,j,count=0,m;

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

{

if(processes[i].arrival\_time==0)

count++;

}

if(count==n||count==1)

{

if(count==n)

{

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

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

if (processes[j].priority > processes[j + 1].priority) {

struct Process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

}

}

else

{

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

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

if (processes[j].priority > processes[j + 1].priority) {

struct Process temp = processes[j];

processes[j] = processes[j + 1];

processes[j + 1] = temp;

}

}

}

}

}

int total\_time = 0;

double total\_turnaround\_time = 0;

double total\_waiting\_time = 0;

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

total\_time += processes[i].burst\_time;

processes[i].turnaround\_time = total\_time - processes[i].arrival\_time;

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

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

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

}

printf("Process\tTurnaround Time\tWaiting Time\n");

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

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

}

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

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

}

void priority\_preemptive(struct Process processes[], int n) {

int total\_time = 0,i;

int completed = 0;

while (completed < n) {

int highest\_priority = -1;

int next\_process = -1;

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

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

if (highest\_priority == -1 || processes[i].priority < highest\_priority) {

highest\_priority = processes[i].priority;

next\_process = i;

}

}

}

if (next\_process == -1) {

total\_time++;

continue;

}

processes[next\_process].remaining\_time--;

total\_time++;

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

completed++;

processes[next\_process].turnaround\_time = total\_time - processes[next\_process].arrival\_time;

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

}

}

double total\_turnaround\_time = 0;

double total\_waiting\_time = 0;

printf("Process\tTurnaround Time\tWaiting Time\n");

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

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

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

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

}

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

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

}

void round\_robin(struct Process processes[], int n, int quantum) {

int i, total\_time = 0, completed = 0;

while (completed < n) {

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

if (processes[i].remaining\_time > 0) {

if (processes[i].remaining\_time > quantum) {

total\_time += quantum;

processes[i].remaining\_time -= quantum;

} else {

total\_time += processes[i].remaining\_time;

processes[i].remaining\_time = 0;

processes[i].turnaround\_time = total\_time - processes[i].arrival\_time;

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

completed++;

}

}

}

}

double total\_turnaround\_time = 0;

double total\_waiting\_time = 0;

printf("Process\tTurnaround Time\tWaiting Time\n");

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

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

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

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

}

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

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

}

void main() {

int n, quantum, i, choice;

struct Process processes[MAX\_PROCESSES];

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

scanf("%d", &n);

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

printf("Process %d\n", i + 1);

printf("Enter arrival time: ");

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

printf("Enter burst time: ");

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

printf("Enter priority: ");

scanf("%d", &processes[i].priority);

processes[i].pid = i + 1;

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

processes[i].turnaround\_time = 0;

processes[i].waiting\_time = 0;

}

while (1) {

printf("\nSelect a scheduling algorithm:\n");

printf("1. Priority (Non-preemptive)\n");

printf("2. Priority (Preemptive)\n");

printf("3. Round Robin\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("\nPriority Non-preemptive Scheduling:\n");

priority\_nonpreemptive(processes, n);

break;

case 2:

printf("\nPriority Preemptive Scheduling:\n");

priority\_preemptive(processes, n);

break;

case 3:

printf("\nRound Robin Scheduling:\n");

printf("Enter the time quantum: ");

scanf("%d", &quantum);

round\_robin(processes, n, quantum);

break;

case 4:

exit(0);

default:

printf("Invalid choice!\n");

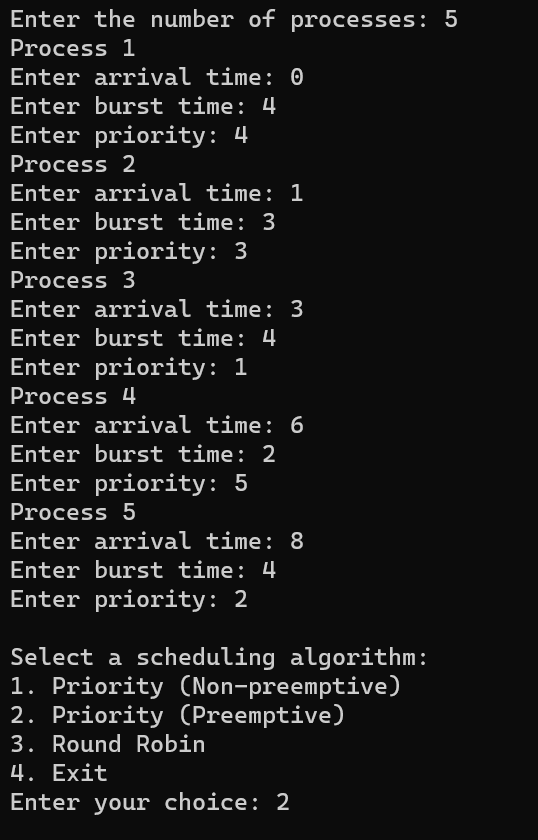
break;

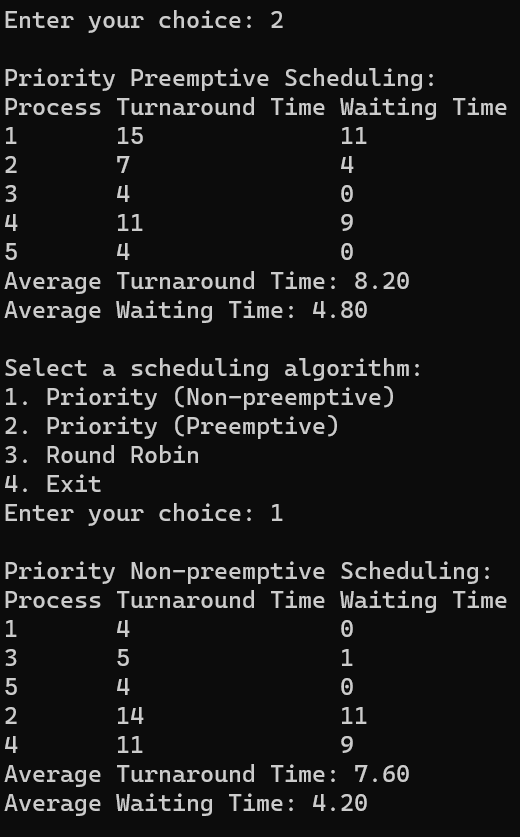
}

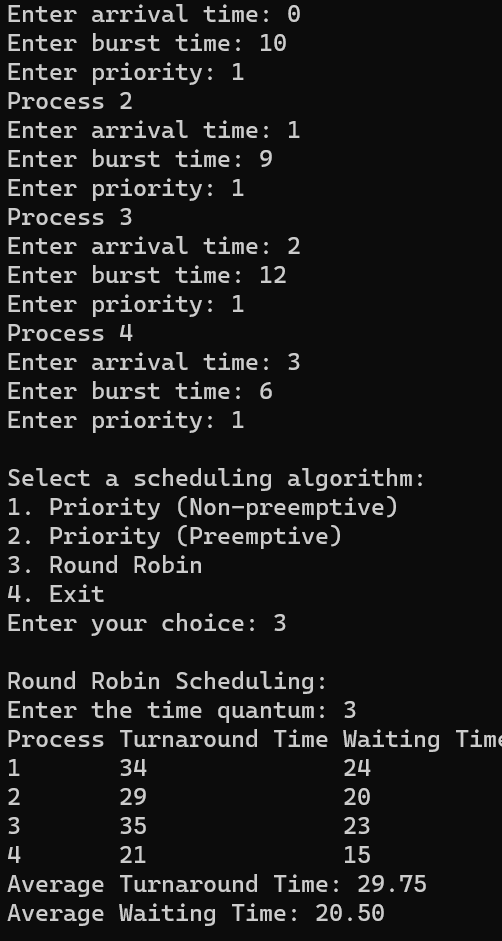
}

}

**Result Screen shot**

****

****

****

**Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue.**

**CODE:**

#include <stdio.h>

#include<stdlib.h>

#include <stdbool.h>

#define MAX\_QUEUE\_SIZE 100

int totalTime=0;

int userProcess=0,systemProcess=0;

// Structure to represent a process

typedef struct {

int processID;

int arrivalTime;

int burstTime;

int remainingTime;

int priority; // 0 for system process, 1 for user process

} Process;

// Function to execute a process

void executeProcess(Process process) {

int i;

printf("Executing Process %d\n", process.processID);

// Simulating the execution time of the process

for (i = 1; i <= process.burstTime; i++) {

printf("Process %d: %d/%d\n", process.processID, i,

process.burstTime);

}

printf("Process %d executed\n", process.processID);

}

// Function to perform FCFS scheduling for a queue of processes

void scheduleFCFS(Process system[],Process user[]) {

int i,j;

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

{

for(j=i+1;j<systemProcess;j++)

{

if(system[i].arrivalTime>system[j].arrivalTime)

{

Process temp=system[i];

system[i]=system[j];

system[j]=temp;

}

}

}

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

{

for(j=i+1;j<userProcess;j++)

{

if(user[i].arrivalTime>user[j].arrivalTime)

{

Process temp=user[i];

user[i]=user[j];

user[j]=temp;

}

}

}

int completed=0;

int currentProcess=-1;

bool isUserProcess=false;

int size=userProcess+systemProcess;

while(1)

{

int count=0;

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

{

if(system[i].remainingTime<=0)

{

count++;

}

}

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

{

if(user[j].remainingTime<=0)

{

count++;

}

}

if(count==size)

{

printf("\n end of processess");

exit(0);

}

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

{

if(totalTime>=system[i].arrivalTime &&

system[i].remainingTime>0)

{

currentProcess=i;

isUserProcess=false;

break;

}

}

if(currentProcess==-1)

{

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

{

if(totalTime>=user[j].arrivalTime &&

user[j].remainingTime>0)

{

currentProcess=j;

isUserProcess=true;

break;

}

}

}

if(currentProcess==-1)

{

totalTime++;

printf("\n %d idle time...",totalTime);

if(totalTime==1000)

{

exit(0);

}

continue;

}

if(isUserProcess==true)

{

user[currentProcess].remainingTime--;

printf("\n User process %d will excecute at %d

",user[currentProcess].processID,(totalTime));

totalTime++;

isUserProcess=false;

currentProcess=-1;

if(user[currentProcess].remainingTime==0)

{

}

}else{

completed++;

int temp=totalTime;

while(system[currentProcess].remainingTime--){

totalTime++;

}

if(system[currentProcess].remainingTime==0)

{

completed++;

}

printf("\n System process %d will excecute

from %d to %d ",system[currentProcess].processID,temp,(totalTime));

isUserProcess=false;

currentProcess=-1;

}

}

}

int main() {

int numProcesses,i;

Process processes[MAX\_QUEUE\_SIZE];

// Reading the number of processes

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

scanf("%d", &numProcesses);

// Reading process details

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

printf("Process %d:\n", i + 1);

printf("Arrival Time: ");

scanf("%d", &processes[i].arrivalTime);

printf("Burst Time: ");

scanf("%d", &processes[i].burstTime);

printf("System(0)/User(1): ");

scanf("%d", &processes[i].priority);

processes[i].processID = i + 1;

processes[i].remainingTime=processes[i].burstTime;

if(processes[i].priority==1)

{

userProcess++;

}else{

systemProcess++;

}

}

Process systemQueue[MAX\_QUEUE\_SIZE];

int systemQueueSize = 0;

Process userQueue[MAX\_QUEUE\_SIZE];

int userQueueSize = 0;

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

if (processes[i].priority == 0) {

systemQueue[systemQueueSize++] = processes[i];

} else {

userQueue[userQueueSize++] = processes[i];

}

}

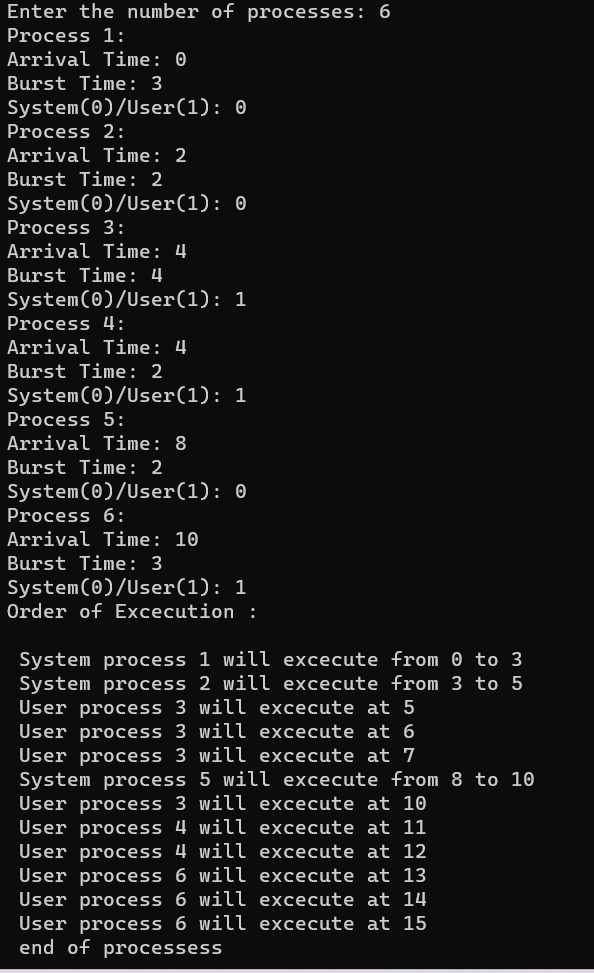
printf("Order of Excecution :\n");

scheduleFCFS(systemQueue,userQueue);

return 0;

}

**Result Screen shot**



**Write a C program to simulate Real-Time CPU Scheduling**

**algorithms:**

**a) Rate- Monotonic**

**b) Earliest-deadline First**

**c) Proportional scheduling**

**a)CODE:**

#include <stdio.h> #include <stdlib.h> #include <math.h> #include <stdbool.h>

#define MAX\_PROCESS 10

int num\_of\_process = 3, count, remain, time\_quantum;

int execution\_time[MAX\_PROCESS], period[MAX\_PROCESS], remain\_time[MAX\_PROCESS], deadline[MAX\_PROCESS], remain\_deadline[MAX\_PROCESS];

int burst\_time[MAX\_PROCESS], wait\_time[MAX\_PROCESS], completion\_time[MAX\_PROCESS], arrival\_time[MAX\_PROCESS];

// collecting details of processes

void get\_process\_info(int selected\_algo)

{

printf("Enter total number of processes (maximum %d): ", MAX\_PROCESS);

scanf("%d", &num\_of\_process); if (num\_of\_process < 1)

{

printf("Do you really want to schedule %d processes? -\_-", num\_of\_process);

exit(0);

}

if (selected\_algo == 2)

{

printf("\nEnter Time Quantum: "); scanf("%d", &time\_quantum);

if (time\_quantum < 1)

{

printf("Invalid Input: Time quantum should be greater than 0\n"); exit(0);

}

}

for (int i = 0; i < num\_of\_process; i++)

{

printf("\nProcess %d:\n", i + 1); if (selected\_algo == 1)

{

printf("==> Burst time: "); scanf("%d", &burst\_time[i]);

}

else if (selected\_algo == 2)

{

printf("=> Arrival Time: "); scanf("%d", &arrival\_time[i]); printf("=> Burst Time: "); scanf("%d", &burst\_time[i]); remain\_time[i] = burst\_time[i];

}

else if (selected\_algo > 2)

{

printf("==> Execution time: "); scanf("%d", &execution\_time[i]); remain\_time[i] = execution\_time[i]; if (selected\_algo == 4)

{

printf("==> Deadline: "); scanf("%d", &deadline[i]);

}

else

{

printf("==> Period: "); scanf("%d", &period[i]);

}

}

}

}

// get maximum of three numbers int max(int a, int b, int c)

{

int max;

if (a >= b && a >= c) max = a;

else if (b >= a && b >= c) max = b;

else if (c >= a && c >= b) max = c;

return max;

}

// calculating the observation time for scheduling timeline int get\_observation\_time(int selected\_algo)

{

if (selected\_algo < 3)

{

int sum = 0;

for (int i = 0; i < num\_of\_process; i++)

{

sum += burst\_time[i];

}

return sum;

}

else if (selected\_algo == 3)

{

return max(period[0], period[1], period[2]);

}

else if (selected\_algo == 4)

{

return max(deadline[0], deadline[1], deadline[2]);

}

}

// print scheduling sequence

void print\_schedule(int process\_list[], int cycles)

{

printf("\nScheduling:\n\n"); printf("Time: ");

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

{

if (i < 10)

printf("| 0%d ", i); else

printf("| %d ", i);

}

printf("|\n");

for (int i = 0; i < num\_of\_process; i++)

{

printf("P[%d]: ", i + 1);

for (int j = 0; j < cycles; j++)

{

if (process\_list[j] == i + 1) printf("|####");

else

printf("| ");

}

printf("|\n");

}

}

void rate\_monotonic(int time)

{

int process\_list[100] = {0}, min = 999, next\_process = 0;

float utilization = 0;

for (int i = 0; i < num\_of\_process; i++)

{

utilization += (1.0 \* execution\_time[i]) / period[i];

}

int n = num\_of\_process;

if (utilization > n \* (pow(2, 1.0 / n) - 1))

{

printf("\nGiven problem is not schedulable under the said scheduling algorithm.\n");

exit(0);

}

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

{

min = 1000;

for (int j = 0; j < num\_of\_process; j++)

{

if (remain\_time[j] > 0)

{

if (min > period[j])

{

min = period[j]; next\_process = j;

}

}

}

if (remain\_time[next\_process] > 0)

{

process\_list[i] = next\_process + 1; // +1 for catering 0 array index. remain\_time[next\_process] -= 1;

}

for (int k = 0; k < num\_of\_process; k++)

{

if ((i + 1) % period[k] == 0)

{

remain\_time[k] = execution\_time[k]; next\_process = k;

}

}

}

print\_schedule(process\_list, time);

}

int main(int argc, char \*argv[])

{

int option = 0;

printf("3. Rate Monotonic Scheduling\n"); printf("Select > ");

scanf("%d", &option);

printf(" \n");

get\_process\_info(option); // collecting processes detail int observation\_time = get\_observation\_time(option);

if (option == 3) rate\_monotonic(observation\_time);

return 0;

}

**Result Screen shot**



**b)CODE:**

#include <stdio.h>

#define arrival 0

#define execution 1

#define deadline 2

#define period 3

#define abs\_arrival 4

#define execution\_copy 5

#define abs\_deadline 6

typedef struct

{

int T[7],instance,alive;

}task;

#define IDLE\_TASK\_ID 1023

#define ALL 1

#define CURRENT 0

void get\_tasks(task \*t1,int n);

int hyperperiod\_calc(task \*t1,int n);

float cpu\_util(task \*t1,int n);

int gcd(int a, int b);

int lcm(int \*a, int n);

int sp\_interrupt(task \*t1,int tmr,int n);

int min(task \*t1,int n,int p);

void update\_abs\_arrival(task \*t1,int n,int k,int all);

void update\_abs\_deadline(task \*t1,int n,int all);

void copy\_execution\_time(task \*t1,int n,int all);

int timer = 0;

int main()

{

task \*t;

int n, hyper\_period, active\_task\_id;

float cpu\_utilization;

printf("Enter number of tasks\n");

scanf("%d", &n);

t = malloc(n \* sizeof(task));

get\_tasks(t, n);

cpu\_utilization = cpu\_util(t, n);

printf("CPU Utilization %f\n", cpu\_utilization);

if (cpu\_utilization < 1)

printf("Tasks can be scheduled\n");

else

printf("Schedule is not feasible\n");

hyper\_period = hyperperiod\_calc(t, n);

copy\_execution\_time(t, n, ALL);

update\_abs\_arrival(t, n, 0, ALL);

update\_abs\_deadline(t, n, ALL);

while (timer <= hyper\_period)

{

if (sp\_interrupt(t, timer, n))

{

active\_task\_id = min(t, n, abs\_deadline);

}

if (active\_task\_id == IDLE\_TASK\_ID)

{

printf("%d Idle\n", timer);

}

if (active\_task\_id != IDLE\_TASK\_ID)

{

if (t[active\_task\_id].T[execution\_copy] != 0)

{

t[active\_task\_id].T[execution\_copy]--;

printf("%d Task %d\n", timer, active\_task\_id + 1);

}

if (t[active\_task\_id].T[execution\_copy] == 0)

{

t[active\_task\_id].instance++;

t[active\_task\_id].alive = 0;

copy\_execution\_time(t, active\_task\_id, CURRENT);

update\_abs\_arrival(t, active\_task\_id,

t[active\_task\_id].instance, CURRENT);

update\_abs\_deadline(t, active\_task\_id, CURRENT);

active\_task\_id = min(t, n, abs\_deadline);

}

}

++timer;

}

free(t);

return 0;

}

void get\_tasks(task \*t1, int n)

{

int i = 0;

while (i < n)

{

printf("Enter Task %d parameters\n", i + 1);

printf("Arrival time: ");

scanf("%d", &t1->T[arrival]);

printf("Execution time: ");

scanf("%d", &t1->T[execution]);

printf("Deadline time: ");

scanf("%d", &t1->T[deadline]);

printf("Period: ");

scanf("%d", &t1->T[period]);

t1->T[abs\_arrival] = 0;

t1->T[execution\_copy] = 0;

t1->T[abs\_deadline] = 0;

t1->instance = 0;

t1->alive = 0;

t1++;

i++;

}

}

int hyperperiod\_calc(task \*t1, int n)

{

int i = 0, ht, a[10];

while (i < n)

{

a[i] = t1->T[period];

t1++;

i++;

}

ht = lcm(a, n);

return ht;

}

int gcd(int a, int b)

{

if (b == 0)

return a;

else

return gcd(b, a % b);

}

int lcm(int \*a, int n)

{

int res = 1, i;

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

{

res = res \* a[i] / gcd(res, a[i]);

}

return res;

}

int sp\_interrupt(task \*t1, int tmr, int n)

{

int i = 0, n1 = 0, a = 0;

task \*t1\_copy;

t1\_copy = t1;

while (i < n)

{

if (tmr == t1->T[abs\_arrival])

{

t1->alive = 1;

a++;

}

t1++;

i++;

}

t1 = t1\_copy;

i = 0;

while (i < n)

{

if (t1->alive == 0)

n1++;

t1++;

i++;

}

if (n1 == n || a != 0)

{

return 1;

}

return 0;

}

void update\_abs\_deadline(task \*t1, int n, int all)

{

int i = 0;

if (all)

{

while (i < n)

{

t1->T[abs\_deadline] = t1->T[deadline] + t1->T[abs\_arrival];

t1++;

i++;

}

}

else

{

t1 += n;

t1->T[abs\_deadline] = t1->T[deadline] + t1->T[abs\_arrival];

}

}

void update\_abs\_arrival(task \*t1, int n, int k, int all)

{

int i = 0;

if (all)

{

while (i < n)

{

t1->T[abs\_arrival] = t1->T[arrival] + k \* (t1->T[period]);

t1++;

i++;

}

}

else

{

t1 += n;

t1->T[abs\_arrival] = t1->T[arrival] + k \* (t1->T[period]);

}

}

void copy\_execution\_time(task \*t1, int n, int all)

{

int i = 0;

if (all)

{

while (i < n)

{

t1->T[execution\_copy] = t1->T[execution];

t1++;

i++;

}

}

else

{

t1 += n;

t1->T[execution\_copy] = t1->T[execution];

}

}

int min(task \*t1, int n, int p)

{

int i = 0, min = 0x7FFF, task\_id = IDLE\_TASK\_ID;

while (i < n)

{

if (min > t1->T[p] && t1->alive == 1)

{

min = t1->T[p];

task\_id = i;

}

t1++;

i++;

}

return task\_id;

}

float cpu\_util(task \*t1, int n)

{

int i = 0;

float cu = 0;

while (i < n)

{

cu = cu + (float)t1->T[execution] / (float)t1->T[deadline];

t1++;

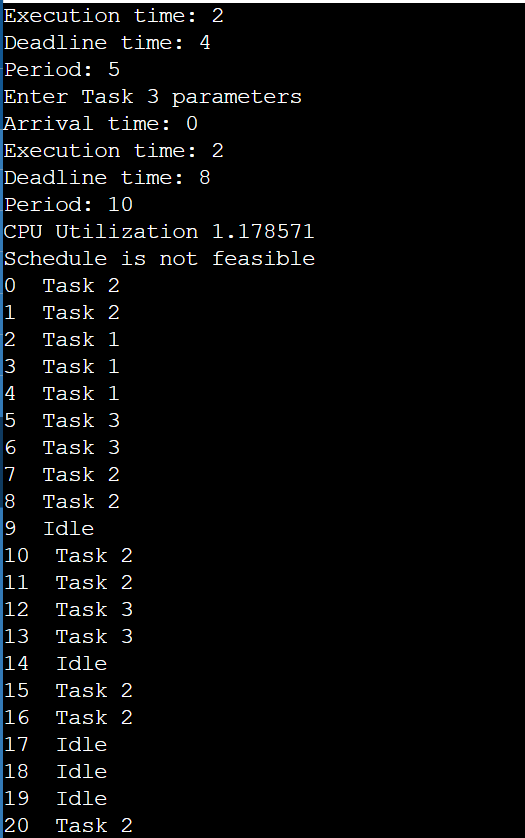
i++;

}

return cu;

}

**Result Screen shot**



**Write a C program to simulate producer-consumer problem using semaphores.**

**CODE:**

#include<stdio.h>

#include<stdlib.h>

int mutex=1,full=0,empty=3,x=0;

int main()

{

int n;

void producer();

void consumer();

int wait(int);

int signal(int);

printf("\n1.Producer \n 2.Consumer \n");

while(1)

{

printf("Enter your choice:");

scanf("%d",&n);

switch(n)

{

case 1:if((mutex==1)&&(empty!=0))

producer();

else

printf("Buffer is full \n");

break;

case 2:if((mutex==1)&&(full!=0))

consumer();

else

printf("Buffer is empty \n");

break;

case 3:exit(0);break;

}

}

return 0;

}//main

int wait(int s)

{

return(--s);

}

int signal(int s)

{

return(++s);

}

void producer()

{

mutex=wait(mutex);

full=signal(full);

empty=wait(empty);

x++;

printf("\n Producer produces item %d \n",x);

mutex=signal(mutex);

}

//producer

void consumer()

{

mutex=wait(mutex);

full=wait(full);

empty=signal(empty);

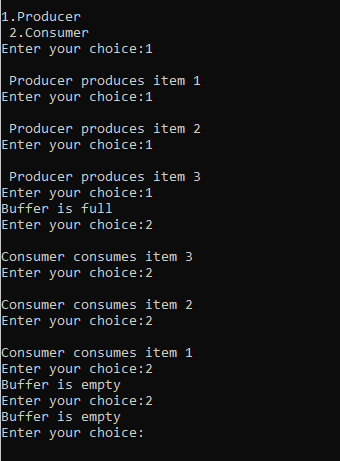
printf("\nConsumer consumes item %d \n",x);

x--;

mutex=signal(mutex);

}//consumer

**Result Screen shot**



**Write a C program to simulate the concept of Dining-Philosophers**

**problem.**

**CODE:**

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#define N 5

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (phnum + 4) % N

#define RIGHT (phnum + 1) % N

int state[N];

int phil[N] = { 0, 1, 2, 3, 4 };

sem\_t mutex;

sem\_t S[N];

void test(int phnum)

{

if (state[phnum] == HUNGRY

&& state[LEFT] != EATING

&& state[RIGHT] != EATING) {

// state that eating

state[phnum] = EATING;

sleep(2);

printf("Philosopher %d takes fork %d and %d\n",

phnum + 1, LEFT + 1, phnum + 1);

printf("Philosopher %d is Eating\n", phnum + 1);

// sem\_post(&S[phnum]) has no effect

// during takefork

// used to wake up hungry philosophers

// during putfork

sem\_post(&S[phnum]);

}

}

// take up chopsticks

void take\_fork(int phnum)

{

sem\_wait(&mutex);

// state that hungry

state[phnum] = HUNGRY;

printf("Philosopher %d is Hungry\n", phnum + 1);

// eat if neighbours are not eating

test(phnum);

sem\_post(&mutex);

// if unable to eat wait to be signalled

sem\_wait(&S[phnum]);

sleep(1);

}

// put down chopsticks

void put\_fork(int phnum)

{

sem\_wait(&mutex);

// state that thinking

state[phnum] = THINKING;

printf("Philosopher %d putting fork %d and %d down\n",

phnum + 1, LEFT + 1, phnum + 1);

printf("Philosopher %d is thinking\n", phnum + 1);

test(LEFT);

test(RIGHT);

sem\_post(&mutex);

}

void\* philosopher(void\* num)

{

while (1) {

int\* i = num;

sleep(1);

take\_fork(\*i);

sleep(0);

put\_fork(\*i);

}

}

int main()

{

int i;

pthread\_t thread\_id[N];

// initialize the semaphores

sem\_init(&mutex, 0, 1);

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

sem\_init(&S[i], 0, 0);

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

// create philosopher processes

pthread\_create(&thread\_id[i], NULL,

philosopher, &phil[i]);

printf("Philosopher %d is thinking\n", i + 1);

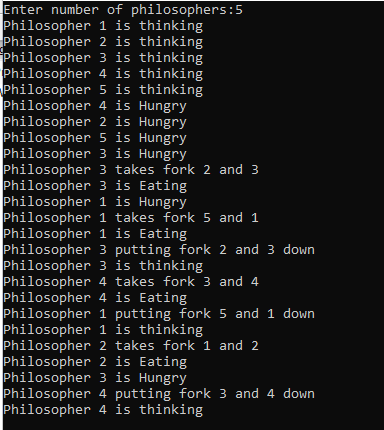
}

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

pthread\_join(thread\_id[i], NULL);

}

**Result Screen shot**



**Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.**

**CODE:**

#include<stdio.h>

struct file

{

int all[10];

int max[10];

int need[10];

int flag;

};

void main()

{

struct file f[10];

int fl;

int i, j, k, p, b, n, r, g, cnt=0, id, newr;

int avail[10],seq[10];

printf("Enter number of processes : ");

scanf("%d",&n);

printf("Enter number of resources : ");

scanf("%d",&r);

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

{

printf("Enter details for P%d",i);

printf("\nEnter allocation\t : \t");

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

scanf("%d",&f[i].all[j]);

printf("Enter Max\t\t : \t");

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

scanf("%d",&f[i].max[j]);

f[i].flag=0;

}

printf("\nEnter Available Resources\t : \t");

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

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

printf("\nEnter New Request Details :");

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

scanf("%d",&id);

printf("Enter Request for Resources \t : \t");

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

{

scanf("%d",&newr);

f[id].all[i] += newr;

avail[i]=avail[i] - newr;

}

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

{

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

{

f[i].need[j]=f[i].max[j]-f[i].all[j];

if(f[i].need[j]<0)

f[i].need[j]=0;

}

}

cnt=0; fl=0;

while(cnt!=n)

{ g=0;

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

{

if(f[j].flag==0)

{ b=0;

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

{

if(avail[p]>=f[j].need[p]) b=b+1;

else b=b-1;

}

if(b==r)

{

printf("\nP%d is visited",j);

seq[fl++]=j;

f[j].flag=1;

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

avail[k]=avail[k]+f[j].all[k];

cnt=cnt+1;

printf("(");

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

printf("%3d",avail[k]);

printf(")");

g=1;

}

}

}

if(g==0)

{

printf("\n REQUEST NOT GRANTED -- DEADLOCK OCCURRED");

printf("\n SYSTEM IS IN UNSAFE STATE");

goto y;

}

}

printf("\nSYSTEM IS IN SAFE STATE");

printf("\nThe Safe Sequence is -- (");

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

printf("P%d ",seq[i]);

printf(")");

y: printf("\nProcess\t\tAllocation\t\tMax\t\tNeed\n");

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

{

printf("P%d\t",i);

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

printf("%5d",f[i].all[j]);

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

printf("%5d",f[i].max[j]);

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

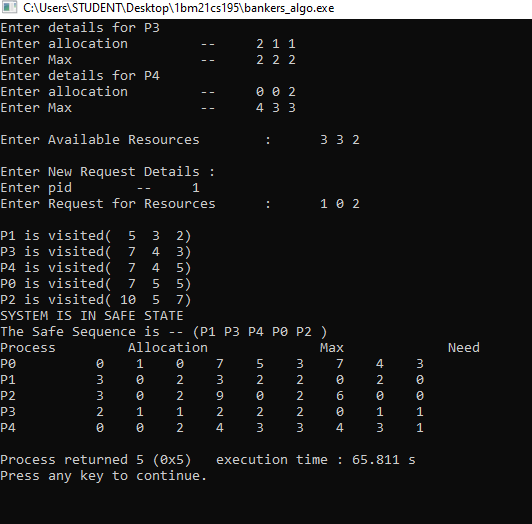
printf("%5d",f[i].need[j]);

printf("\n");

}

}

**Result Screen shot**



**Write a C program to simulate deadlock detection**

**CODE:**

#include <stdio.h>

#define MAX\_PROCESSES 5

#define MAX\_RESOURCES 3

int allocated[MAX\_PROCESSES][MAX\_RESOURCES];

int requested[MAX\_PROCESSES][MAX\_RESOURCES];

int available[MAX\_RESOURCES];

int work[MAX\_RESOURCES];

int finish[MAX\_PROCESSES];

void initialize()

{

// Initialize allocated and requested matrices

for (int i = 0; i < MAX\_PROCESSES; i++)

{

printf("Enter allocated resources for process P%d:\n", i);

for (int j = 0; j < MAX\_RESOURCES; j++)

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

printf("Enter requested resources for process P%d:\n", i);

for (int j = 0; j < MAX\_RESOURCES; j++)

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

finish[i] = 0; // Process is not finished yet

}

}

int checkSafety()

{

for (int i = 0; i < MAX\_RESOURCES; i++)

work[i] = available[i];

int count = 0;

while (count < MAX\_PROCESSES)

{

int found = 0;

for (int i = 0; i < MAX\_PROCESSES; i++)

{

if (!finish[i])

{

int j;

for (j = 0; j < MAX\_RESOURCES; j++)

{

if (requested[i][j] > work[j])

break;

}

if (j == MAX\_RESOURCES)

{

for (int k = 0; k < MAX\_RESOURCES; k++)

work[k] += allocated[i][k];

finish[i] = 1;

found = 1;

count++;

}

}

}

if (!found)

break;

}

return count == MAX\_PROCESSES;

}

int main()

{

initialize();

// Assume available resources are initially zero

for (int i = 0; i < MAX\_RESOURCES; i++)

available[i] = 0;

if (checkSafety())

printf("System is in safe state.\n");

else

printf("System is in unsafe state.\n");

return 0;

}

**Result Screen shot**

****

****

**Write a C program to simulate the following contiguous memory**

**allocation techniques**

**a) Worst-fit**

**b) Best-fit**

**c) First-fit**

**CODE:**

#include <stdio.h>

#include <conio.h>

#define max 25

int frag[max], b[max], f[max], nf, nb;

int bf[max], ff[max];

void firstfit() {

int i, j, temp;

static int bf[max];

for (i = 1; i <= nf; i++) {

for (j = 1; j <= nb; j++) {

if (bf[j] != 1) {

temp = b[j] - f[i];

if (temp >= 0) {

ff[i] = j;

break;

}

}

}

frag[i] = temp;

bf[ff[i]] = 1;

}

printf("\nFile\_size:\tBlock\_size:");

for (i = 1; i <= nf; i++) {

printf("\n%d\t\t%d", f[i], b[ff[i]]);

}

}

void bestfit() {

int i, j, temp, lowest = 10000;

static int bf[max];

for (i = 1; i <= nf; i++) {

for (j = 1; j <= nb; j++) {

if (bf[j] != 1) {

temp = b[j] - f[i];

if (temp >= 0 && lowest > temp) {

ff[i] = j;

lowest = temp;

}

}

}

frag[i] = lowest;

bf[ff[i]] = 1;

lowest = 10000;

}

printf("\nFile Size:\tBlock Size:");

for (i = 1; i <= nf && ff[i] != 0; i++) {

printf("\n%d\t\t%d", f[i], b[ff[i]]);

}

}

void worstfit() {

int i, j, temp, highest = 0;

static int bf[max];

for (i = 1; i <= nf; i++) {

for (j = 1; j <= nb; j++) {

if (bf[j] != 1) {

temp = b[j] - f[i];

if (temp >= 0 && highest < temp) {

ff[i] = j;

highest = temp;

}

}

}

frag[i] = highest;

bf[ff[i]] = 1;

highest = 0;

}

printf("\nFile\_size:\tBlock\_size:");

for (i = 1; i <= nf; i++) {

printf("\n%d\t\t%d", f[i], b[ff[i]]);

}

}

int main() {

int c;

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

scanf("%d", &nb);

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

scanf("%d", &nf);

printf("Enter the size of the blocks:\n");

for (int i = 1; i <= nb; i++) {

printf("Block %d:", i);

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

}

printf("Enter the size of the files:\n");

for (int i = 1; i <= nf; i++) {

printf("File %d:", i);

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

}

while (1) {

printf("\n1. First Fit 2. Best Fit 3. Worst Fit 4. Exit");

printf("\nEnter choice:");

scanf("%d", &c);

switch (c) {

case 1:

firstfit();

break;

case 2:

bestfit();

break;

case 3:

worstfit();

break;

case 4:

return 0;

default:

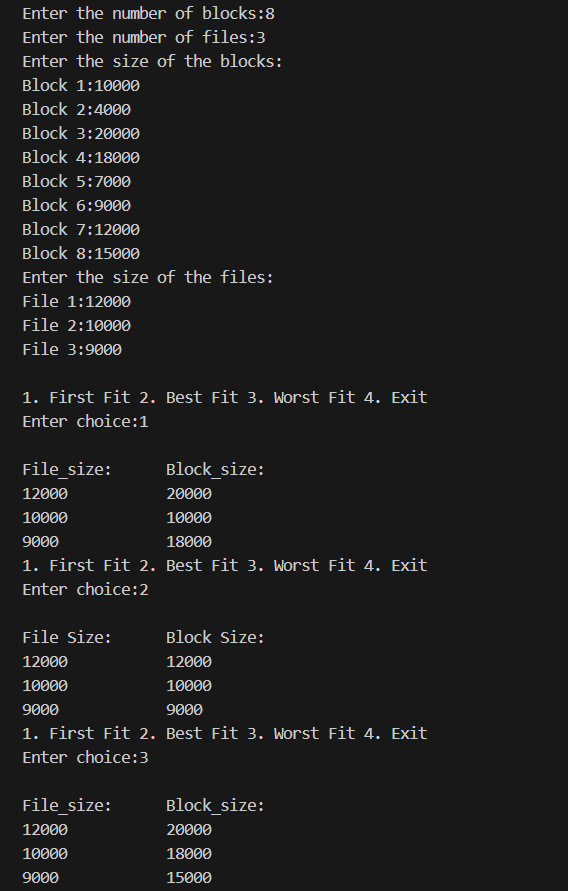
printf("Invalid choice");

}

}

}

**Result Screen shot**

****

**Write a C program to simulate the paging technique of memory**

**management.**

**CODE:**

#include <stdio.h>

#define MAX 50

int main() {

    int page[MAX], i, n, f, ps, off, pno;

    int choice = 0;

    printf("Enter the number of pages in memory: ");

    scanf("%d", &n);

    printf("Enter page size: ");

    scanf("%d", &ps);

    printf("Enter number of frames: ");

    scanf("%d", &f);

    for (i = 0; i < n; i++)

        page[i] = -1;

    printf("\nEnter the page table\n");

    printf("(Enter frame no as -1 if that page is not present in any frame)\n\n");

    printf("pageno\tframeno\n-------\t-------");

    for (i = 0; i < n; i++) {

        printf("\n\n%d\t\t", i);

        scanf("%d", &page[i]);

    }

    do {

        printf("\n\nEnter the logical address (i.e., page no & offset):");

        scanf("%d%d", &pno, &off);

        if (pno < 0 || pno >= n) {

            printf("\nInvalid page number\n");

            continue;

        }

        if (page[pno] == -1)

            printf("\n\nThe required page is not available in any of frames");

        else if (off < 0 || off >= ps)

            printf("\n\nInvalid offset\n");

        else

            printf("\n\nPhysical address (i.e., frame no & offset): %d,%d", page[pno], off);

        printf("\nDo you want to continue (1/0)?: ");

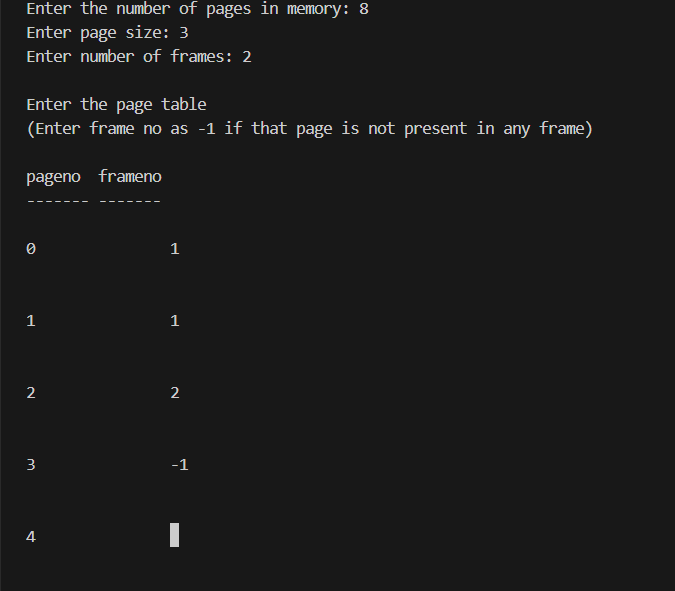
        scanf("%d", &choice);

    } while (choice == 1);

    return 0;

}

**Result Screen shot**

****

**Write a C program to simulate page replacement algorithms**

**a) FIFO**

**b) LRU**

**c) Optimal**

**CODE:**

#include<stdio.h>

int n,nf;

int in[100];

int p[50];

int hit=0;

int i,j,k;

int pgfaultcnt=0;

void getData()

{

printf("\nEnter length of page reference sequence:");

scanf("%d",&n);

printf("\nEnter the page reference sequence:");

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

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

printf("\nEnter no of frames:");

scanf("%d",&nf);

}

void initialize()

{

pgfaultcnt=0;

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

p[i]=9999;

}

int isHit(int data)

{

hit=0;

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

{

if(p[j]==data)

{

hit=1;

break;

}

}

return hit;

}

int getHitIndex(int data)

{

int hitind;

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

{

if(p[k]==data)

{

hitind=k;

break;

}

}

return hitind;

}

void dispPages()

{

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

{

if(p[k]!=9999)

printf(" %d",p[k]);

}

}

void dispPgFaultCnt()

{

printf("\nTotal no of page faults:%d",pgfaultcnt);

}

void fifo()

{

initialize();

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

{

printf("\nFor %d :",in[i]);

if(isHit(in[i])==0)

{

for(k=0; k<nf-1; k++)

p[k]=p[k+1];

p[k]=in[i];

pgfaultcnt++;

dispPages();

}

else

printf("No page fault");

}

dispPgFaultCnt();

}

void optimal() //replace the page that will be used in the most layer point of time

{

initialize();

int near[50];

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

{

printf("\nFor %d :",in[i]);

if(isHit(in[i])==0)

{

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

{

int pg=p[j];

int found=0;

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

{

if(pg==in[k])

{

near[j]=k;

found=1;

break;

}

else

found=0;

}

if(!found)

near[j]=9999;

}

int max=-9999;

int repindex;

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

{

if(near[j]>max)

{

max=near[j];

repindex=j;

}

}

p[repindex]=in[i];

pgfaultcnt++;

dispPages();

}

else

printf("No page fault");

}

dispPgFaultCnt();

}

void lru()

{

initialize();

int least[50];

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

{

printf("\nFor %d :",in[i]);

if(isHit(in[i])==0)

{

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

{

int pg=p[j];

int found=0;

for(k=i-1; k>=0; k--)

{

if(pg==in[k])

{

least[j]=k;

found=1;

break;

}

else

found=0;

}

if(!found)

least[j]=-9999;

}

int min=9999;

int repindex;

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

{

if(least[j]<min)

{

min=least[j];

repindex=j;

}

}

p[repindex]=in[i];

pgfaultcnt++;

dispPages();

}

else

printf("No page fault!");

}

dispPgFaultCnt();

}

int main()

{

int choice;

while(1)

{

printf("\nPage Replacement Algorithms\n1.Enter data\n2.FIFO\n3.Optimal\n4.LRU\n7.Exit\nEnter your choice:");

scanf("%d",&choice);

switch(choice)

{

case 1:

getData();

break;

case 2:

fifo();

break;

case 3:

optimal();

break;

case 4:

lru();

break;

default:

return 0;

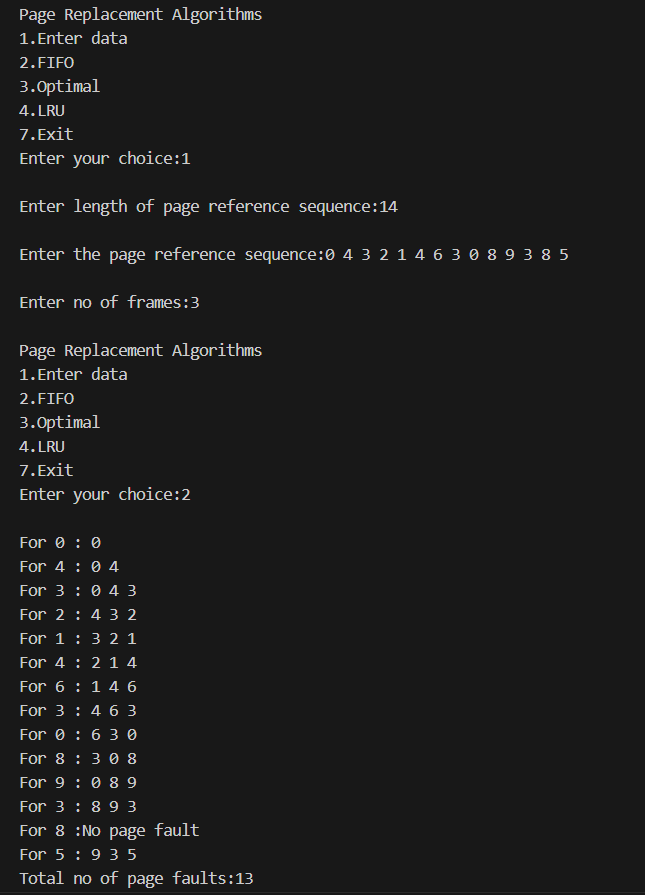
break;

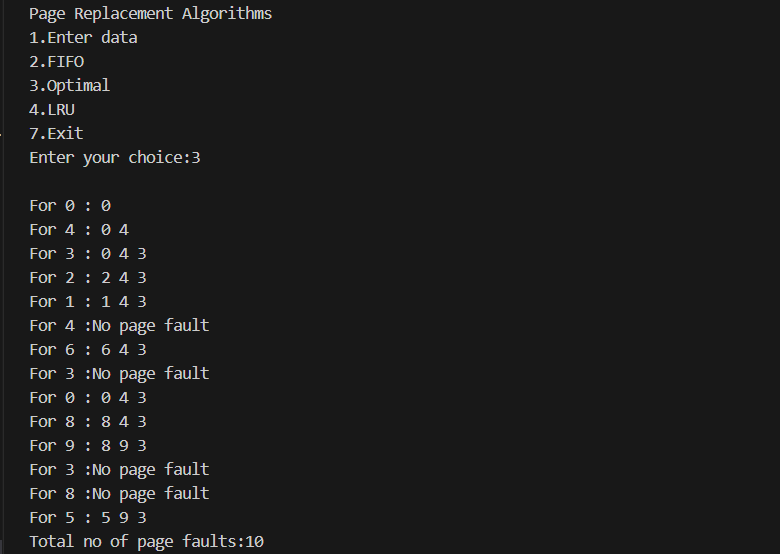
}

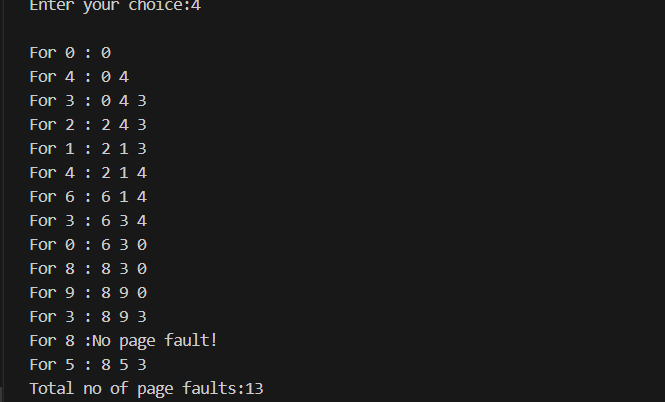
}

}

**Result Screen shot**







**Write a C program Write a C program to simulate disk scheduling algorithms**

**a) FCFS**

**b) SCAN**

**c) C-SCAN**

**a) SSTF**

**b) LOOK**

**c) c-LOOK**

**CODE:**

#include <stdio.h>

#include <stdlib.h>

int m, n, start; // Global variables for disk specifications

int a[15]; // Global array for the request queue

int absolute(int a, int b)

{

int c = a - b;

if (c < 0)

return -c;

else

return c;

}

void fcfs()

{

printf("\nFCFS:\n");

int count = 0;

int x = start;

printf("Scheduling services the request in the order that follows:\n%d\t", start);

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

{

x -= a[i];

if (x < 0)

x = -x;

count += x;

x = a[i];

printf("%d\t", x);

}

printf("\nTotal Head Movement: %d Cylinders\n", count);

}

void sstf()

{

printf("\nSSTF:\n");

int count = 0;

int x = start;

printf("Scheduling services the request in the order that follows:\n%d\t", start);

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

{

int min = absolute(a[i], x);

int pos = i;

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

{

if (min > absolute(x, a[j]))

{

pos = j;

min = absolute(x, a[j]);

}

}

count += absolute(x, a[pos]);

x = a[pos];

a[pos] = a[i];

a[i] = x;

printf("%d\t", x);

}

printf("\nTotal Head Movement: %d Cylinders\n", count);

}

//scan

void scan(int direction)

{

printf("\nSCAN:\n");

int count = 0;

int pos = 0;

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

{

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

{

if (a[j] > a[j + 1])

{

int temp = a[j];

a[j] = a[j + 1];

a[j + 1] = temp;

}

}

}

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

{

if (a[i] < start)

pos++;

}

int x = start;

if (direction == 1) // Right direction

{

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

{

count += absolute(a[i], x);

x = a[i];

printf("%d\t", x);

}

if (x != m - 1)

{

count += absolute(x, m - 1);

x = m - 1;

printf("%d\t", x);

}

for (int i = pos - 1; i >= 0; i--)

{

count += absolute(a[i], x);

x = a[i];

printf("%d\t", x);

}

}

else // Left direction

{

for (int i = pos - 1; i >= 0; i--)

{

count += absolute(a[i], x);

x = a[i];

printf("%d\t", x);

}

if (x != 0)

{

count += absolute(x, 0);

x = 0;

printf("%d\t", x);

}

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

{

count += absolute(a[i], x);

x = a[i];

printf("%d\t", x);

}

}

printf("\nTotal Head Movement: %d Cylinders\n", count);

}

void look(int direction)

{

printf("\nLOOK:\n");

int count = 0;

int pos = 0;

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

{

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

{

if (a[j] > a[j + 1])

{

int temp = a[j];

a[j] = a[j + 1];

a[j + 1] = temp;

}

}

}

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

{

if (a[i] < start)

pos++;

}

int x = start;

if (direction == 1) // Right direction

{

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

{

count += absolute(a[i], x);

x = a[i];

printf("%d\t", x);

}

for (int i = pos - 1; i >= 0; i--)

{

count += absolute(a[i], x);

x = a[i];

printf("%d\t", x);

}

}

else // Left direction

{

for (int i = pos - 1; i >= 0; i--)

{

count += absolute(a[i], x);

x = a[i];

printf("%d\t", x);

}

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

{

count += absolute(a[i], x);

x = a[i];

printf("%d\t", x);

}

}

printf("\nTotal Head Movement: %d Cylinders\n", count);

}

void cscan(int direction)

{

printf("\nC-SCAN:\n");

int count = 0;

int pos = 0;

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

{

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

{

if (a[j] > a[j + 1])

{

int temp = a[j];

a[j] = a[j + 1];

a[j + 1] = temp;

}

}

}

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

{

if (a[i] < start)

pos++;

}

int x = start;

if (direction == 1) // Right direction

{

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

{

count += absolute(x, a[i]);

x = a[i];

printf("%d\t", x);

}

count += absolute(m - 1, x);

x = 0;

printf("%d\t%d\t", m - 1, 0);

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

{

count += absolute(x, a[i]);

x = a[i];

printf("%d\t", x);

}

}

else // Left direction

{

for (int i = pos - 1; i >= 0; i--)

{

count += absolute(x, a[i]);

x = a[i];

printf("%d\t", x);

}

count += absolute(0, x);

x = m - 1;

printf("%d\t%d\t", 0, x);

for (int i = n - 1; i >= pos; i--)

{

count += absolute(x, a[i]);

x = a[i];

printf("%d\t", x);

}

}

printf("\nTotal Head Movement: %d Cylinders\n", count);

}

void clook(int direction)

{

printf("\nC-LOOK:\n");

int count = 0;

int pos = 0;

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

{

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

{

if (a[j] > a[j + 1])

{

int temp = a[j];

a[j] = a[j + 1];

a[j + 1] = temp;

}

}

}

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

{

if (a[i] < start)

pos++;

}

int x = start;

if (direction == 1) // Right direction

{

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

{

count += absolute(x, a[i]);

x = a[i];

printf("%d\t", x);

}

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

{

count += absolute(x, a[i]);

x = a[i];

printf("%d\t", x);

}

}

else // Left direction

{

for (int i = pos - 1; i >= 0; i--)

{

count += absolute(x, a[i]);

x = a[i];

printf("%d\t", x);

}

for (int i = n - 1; i >= pos; i--)

{

count += absolute(x, a[i]);

x = a[i];

printf("%d\t", x);

}

}

printf("\nTotal Head Movement: %d Cylinders\n", count);

}

int main()

{

int choice, direction;

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

scanf("%d", &m);

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

scanf("%d", &n);

printf("Enter current position: ");

scanf("%d", &start);

printf("Enter the request queue: ");

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

{

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

if (a[i] >= m)

{

printf("\nInvalid input, re-enter: ");

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

}

}

printf("Enter the direction (1 for Right, 0 for Left): ");

scanf("%d", &direction);

do

{

printf("\n\nDISK SCHEDULING ALGORITHMS\n1. FCFS\n2. SSTF\n3. SCAN\n4. C-SCAN\n5. LOOK\n6. C-LOOK\n");

printf("Enter choice: ");

scanf("%d", &choice);

switch (choice)

{

case 1:

fcfs();

break;

case 2:

sstf();

break;

case 3:

scan(direction);

break;

case 4:

cscan(direction);

break;

case 5:

look(direction);

break;

case 6:

clook(direction);

break;

default:

printf("Invalid choice\n");

}

printf("Do you want to continue? (1 to continue): ");

scanf("%d", &choice);

} while (choice == 1);

return 0;

}

**Result Screen shot**

