**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

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



**LAB REPORT**

## on

Operating Systems

## (22CS4PCOPS)

***Submitted by:***

**Tanisha Gotadke (1BM21CS229)**

***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 - August 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 **Tanisha Gotadke (1BM21CS229),** 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 year 2022-23. The Lab report has been approved as it satisfies the academic requirements in respect of **Operating Systems - (22CS4PCOPS)** work prescribed for the said degree.

**Sonika Sharma D Dr. Jyothi S Nayak**

Assistant Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

**Table Of Contents**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No.** | **Experiment Title** | | | **Page No.** |
| **1** | **Course Outcomes** | | | **5** |
| **2** | **Experiments** | | | **6 - 85** |
|  | **2.1** | **Experiment - 1** | | **6 - 10** |
|  | **2.1.1** | **Question:** | **6** |
|  |  | Write a C program to simulate the following non- |  |
|  |  | pre-emptive CPU scheduling algorithm to find |  |
|  |  | turnaround time and waiting time. |  |
|  |  | **(a)** FCFS |  |
|  |  | **(b)** SJF (pre-emptive & Non-pre-emptive) |  |
|  | **2.1.2** | **Code** | **6** |
|  | **2.1.3** | **Output** | **9** |
|  | **Experiment - 2** | | **11 - 16** |
| **2.2.1** | **Question:** | **11** |
|  | Write a C program to simulate the following CPU |  |
|  | scheduling algorithm to find turnaround time and |  |
|  | waiting time. |  |
|  | **(a)** Priority (pre-emptive & Non-pre-emptive) |  |
|  | **(b)** Round Robin (Experiment with different |  |
|  | quantum sizes for RR algorithm) |  |
| **2.2.2** | **Code** | **11** |
| **2.2.3** | **Output** | **16** |
| **2.3** | **Experiment - 3** | | **17 - 20** |
|  | **2.3.1** | **Question:** | **17** |
|  |  | 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. |  |
|  | **2.3.2** | **Code** | **17** |
|  | **2.3.3** | **Output** | **20** |
| **2.4.** | **Experiment - 4** | | **21 - 29** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | **2.4.1** | **Question:**  Write a C program to simulate Real-Time CPU Scheduling algorithms:   1. Rate- Monotonic 2. Earliest-deadline First 3. Proportional scheduling | **21** |
| **2.4.2** | **Code** | **21** |
| **2.4.3** | **Output** | **29** |
| **2.5.** | **Experiment - 5** | | **30 - 32** |
| **2.5.1** | **Question:**  Write a C program to simulate producer-consumer problem using semaphores. | **30** |
| **2.5.2** | **Code** | **30** |
| **2.5.3** | **Output** | **32** |
| **2.6** | **Experiment - 6** | | **33 - 35** |
| **2.6.1** | **Question:**  Write a C program to simulate the concept of Dining-Philosophers problem. | **33** |
| **2.6.2** | **Code** | **33** |
| **2.6.3** | **Output** | **35** |
| **2.7** | **Experiment - 7** | | **36 - 39** |
| **2.7.1** | **Question:**  Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance. | **36** |
| **2.7.2** | **Code** | **36** |
| **2.7.3** | **Output** | **38** |
| **2.8** | **Experiment - 8** | | **40 - 44** |
| **2.8.1** | **Question:**  Write a C program to simulate deadlock detection. | **40** |
| **2.8.2** | **Code** | **40** |
| **2.8.3** | **Output** | **43** |
| **2.9** | **Experiment - 9** | | **45 - 48** |
| **2.9.1** | **Question:** | **45** |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Write a C program to simulate the following contiguous memory allocation techniques:   1. Worst-fit 2. Best-fit 3. First-fit |  |
| **2.9.2** | **Code** | **45** |
| **2.9.3** | **Output** | **48** |
| **2.10** | **Experiment - 10** | | **49 - 51** |
| **2.10.1** | **Question:**  Write a C program to simulate paging technique of memory management. | **49** |
| **2.10.2** | **Code** | **49** |
| **2.10.3** | **Output** | **50** |
| **2.11** | **Experiment - 11** | | **52 - 58** |
| **2.11.1** | **Question:**  Write a C program to simulate page replacement algorithms:   1. FIFO 2. LRU 3. Optimal | **52** |
| **2.11.2** | **Code** | **52** |
| **2.11.3** | **Output** | **57** |
| **2.12** | **Experiment - 12** | | **59 - 64** |
| **2.12.1** | **Question:**  Write a C program to simulate the following file allocation strategies:   1. Sequential 2. Indexed 3. Linked | **59** |
|  | **2.12.2** | **Code** | **59** |
| **2.12.3** | **Output** | **63** |
| **2.13** | **Experiment - 13** | | **65 - 73** |
| **2.13.1** | **Question:**  Write a C program to simulate the following file organization techniques: | **65** |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 1. Single level directory 2. Two level directory 3. Hierarchical |  |
| **2.13.2** | **Code** | **65** |
| **2.13.3** | **Output** | **72** |
| **2.14** | **Experiment - 14** | | **74 - 79** |
| **2.14.1** | **Question:**  Write a C program to simulate disk scheduling algorithms:   1. FCFS 2. SCAN 3. C-SCAN | **74** |
| **2.14.2** | **Code** | **74** |
| **2.14.3** | **Output** | **79** |
| **2.15** | **Experiment - 15** | | **80 -85** |
| **2.15.1** | **Question:**  Write a C program to simulate disk scheduling algorithms:   1. SSTF 2. LOOK 3. c-LOOK | **80** |
| **2.15.2** | **Code** | **80** |
| **2.15.3** | **Output** | **85** |

# Course Outcomes

**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.

# Experiments

* 1. **Experiment - 1**

## Question:

Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.

1. FCFS
2. SJF

## Code:

#include<stdio.h>

int n, i, j, pos, temp, choice, Burst\_time[20], Waiting\_time[20], Turn\_around\_time[20], process[20], total=0;

float avg\_Turn\_around\_time=0, avg\_Waiting\_time=0;

int FCFS()

{

Waiting\_time[0]=0;

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

{

Waiting\_time[i]=0; for(j=0;j<i;j++)

Waiting\_time[i]+=Burst\_time[j];

}

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

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

{

Turn\_around\_time[i]=Burst\_time[i]+Waiting\_time[i]; avg\_Waiting\_time+=Waiting\_time[i]; avg\_Turn\_around\_time+=Turn\_around\_time[i];

printf("\nP[%d]\t\t%d\t\t\t%d\t\t\t\t%d",i+1,Burst\_time[i],Waiting\_time[i],Turn\_around\_time [i]);

}

avg\_Waiting\_time =(float)(avg\_Waiting\_time)/(float)i; avg\_Turn\_around\_time=(float)(avg\_Turn\_around\_time)/(float)i; printf("\nAverage Waiting Time:%.2f",avg\_Waiting\_time); printf("\nAverage Turnaround Time:%.2f\n",avg\_Turn\_around\_time);

return 0;

}

int SJF()

{

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

{

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

{

if(Burst\_time[j]<Burst\_time[pos]) pos=j;

}

temp=Burst\_time[i]; Burst\_time[i]=Burst\_time[pos]; Burst\_time[pos]=temp;

temp=process[i]; process[i]=process[pos]; process[pos]=temp;

}

Waiting\_time[0]=0;

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

{

Waiting\_time[i]=0;

for(j=0;j<i;j++) Waiting\_time[i]+=Burst\_time[j];

total+=Waiting\_time[i];

}

avg\_Waiting\_time=(float)total/n; total=0;

printf("\nProcess\t\tBurst Time\t\tWaiting Time\t\tTurnaround Time"); for(i=0;i<n;i++)

{

Turn\_around\_time[i]=Burst\_time[i]+Waiting\_time[i]; total+=Turn\_around\_time[i];

printf("\nP[%d]\t\t%d\t\t\t%d\t\t\t\t%d",process[i],Burst\_time[i],Waiting\_time[i],Turn\_aroun d\_time[i]);

}

avg\_Turn\_around\_time=(float)total/n;

printf("\n\nAverage Waiting Time=%f",avg\_Waiting\_time); printf("\nAverage Turnaround Time=%f\n",avg\_Turn\_around\_time);

}

int main()

{

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

printf("\nEnter Burst Time:\n"); for(i=0;i<n;i++)

{

printf("P[%d]:",i+1); scanf("%d",&Burst\_time[i]); process[i]=i+1;

}

while(1)

{ printf("\n-----MAIN MENU \n");

printf("1. FCFS Scheduling\n2. SJF Scheduling\n"); printf("\nEnter your choice:");

scanf("%d", &choice); switch(choice)

{

case 1: FCFS(); break;

case 2: SJF(); break;

default: printf("Invalid Input!!!");

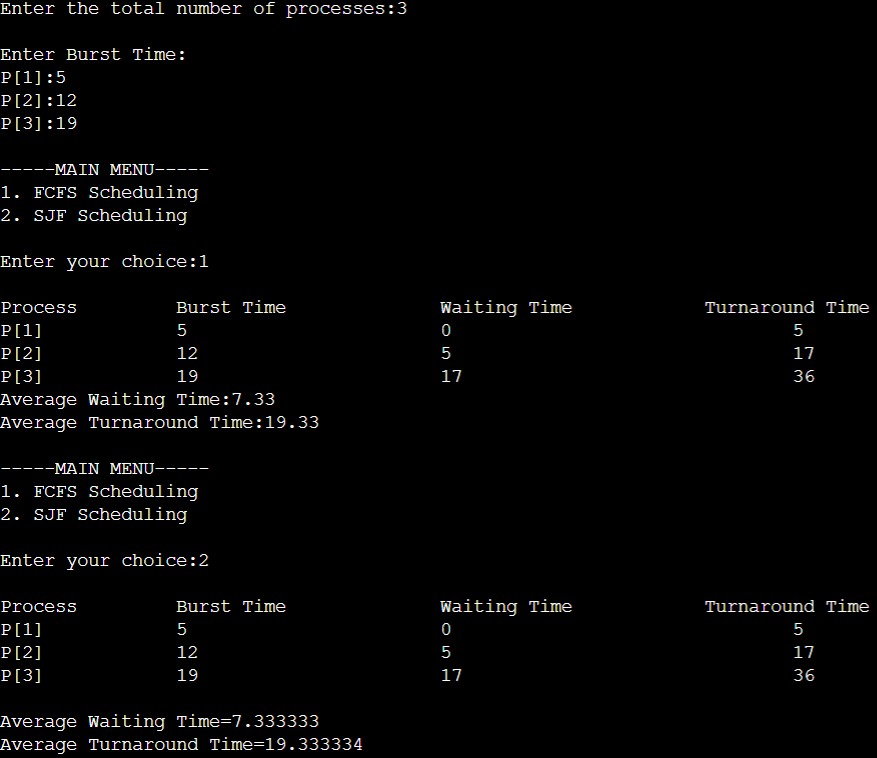
}

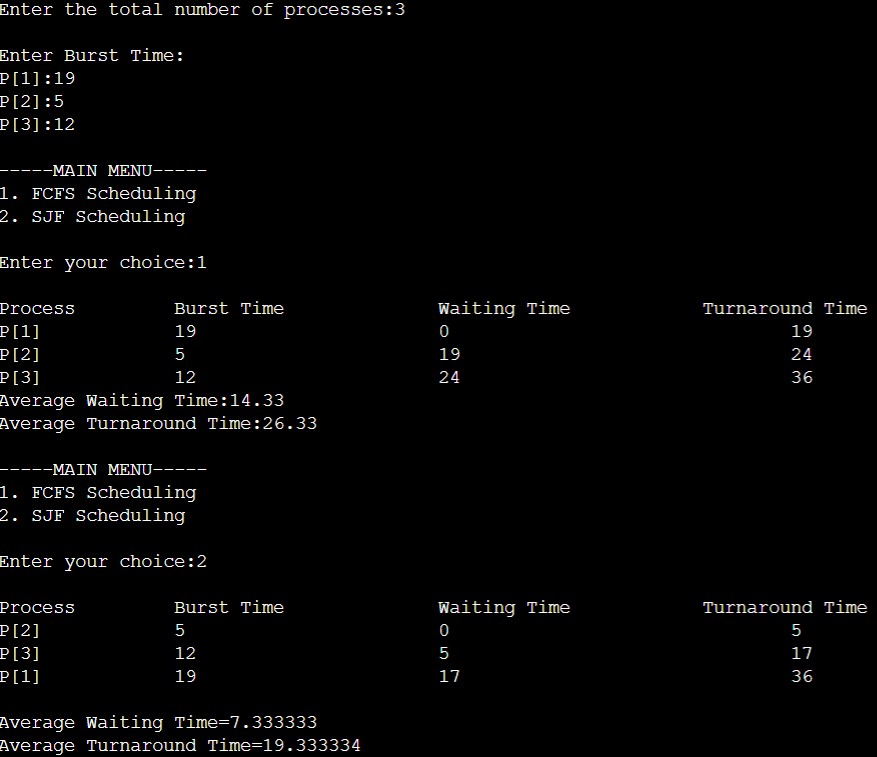
}

return 0;

}

* + 1. **Output:**





# Experiment - 2

## Question:

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

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

## Code:

1. **Priority (Non-pre-emptive)**

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

struct process { int process\_id; int burst\_time; int priority;

int waiting\_time;

int turnaround\_time;

};

void find\_average\_time(struct process[], int); void priority\_scheduling(struct process[], int); int main()

{

int n, i;

struct process proc[10];

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

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

{

printf("\nEnter the process ID: "); scanf("%d", &proc[i].process\_id);

printf("Enter the burst time: "); scanf("%d", &proc[i].burst\_time);

printf("Enter the priority: "); scanf("%d", &proc[i].priority);

}

priority\_scheduling(proc, n); return 0;

}

void find\_waiting\_time(struct process proc[], int n, int wt[])

{

int i; wt[0] = 0;

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

{

wt[i] = proc[i - 1].burst\_time + wt[i - 1];

}

}

void find\_turnaround\_time(struct process proc[], int n, int wt[], int tat[])

{

int i;

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

{

tat[i] = proc[i].burst\_time + wt[i];

}

}

void find\_average\_time(struct process proc[], int n)

{

int wt[10], tat[10], total\_wt = 0, total\_tat = 0, i;

find\_waiting\_time(proc, n, wt); find\_turnaround\_time(proc, n, wt, tat);

printf("\nProcess ID\tBurst Time\tPriority\tWaiting Time\tTurnaround Time"); for(i = 0; i< n; i++)

{

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

printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", proc[i].process\_id, proc[i].burst\_time, proc[i].priority, wt[i], tat[i]);

}

printf("\n\nAverage Waiting Time = %f", (float)total\_wt/n); printf("\nAverage Turnaround Time = %f\n", (float)total\_tat/n);

}

void priority\_scheduling(struct process proc[], int n)

{

int i, j, pos;

struct process temp; for(i = 0; i< n; i++)

{

pos = i;

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

{

if(proc[j].priority< proc[pos].priority) pos = j;

}

temp = proc[i]; proc[i] = proc[pos]; proc[pos] = temp;

}

find\_average\_time(proc, n);

}

1. **Round Robin (Non-pre-emptive)**

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

int turnarroundtime(int processes[], int n, int bt[], int wt[], int tat[]) { for (int i = 0; i < n ; i++)

tat[i] = bt[i] + wt[i]; return 1;

}

int waitingtime(int processes[], int n, int bt[], int wt[], int quantum)

{

int rem\_bt[n];

for (int i = 0 ; i < n ; i++) rem\_bt[i] = bt[i];

int t = 0;

while (1)

{

bool done = true;

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

{

if (rem\_bt[i] > 0)

{

done = false;

if (rem\_bt[i] > quantum)

{

t += quantum; rem\_bt[i] -= quantum;

}

else

{

t = t + rem\_bt[i]; wt[i] = t - bt[i]; rem\_bt[i] = 0;

}

}

}

if (done == true) break;

}

return 1;

}

int findavgTime(int processes[], int n, int bt[], int quantum) { int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

waitingtime(processes, n, bt, wt, quantum); turnarroundtime(processes, n, bt, wt, tat);

printf("\n\nProcesses\t\t Burst Time\t\t Waiting Time\t\t turnaround time\n"); for (int i=0; i<n; i++)

{

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

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

}

printf("\nAverage waiting time = %f", (float)total\_wt / (float)n); printf("\nAverage turnaround time = %f", (float)total\_tat / (float)n); return 1;

}

int main()

{

int n, processes[n], burst\_time[n], quantum; printf("Enter the Number of Processes: "); scanf("%d",&n);

printf("\nEnter the quantum time: "); scanf("%d",&quantum);

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

{

printf("\nEnter the process: "); scanf("%d",&processes[i]); printf("Enter the Burst Time:"); scanf("%d",&burst\_time[i]);

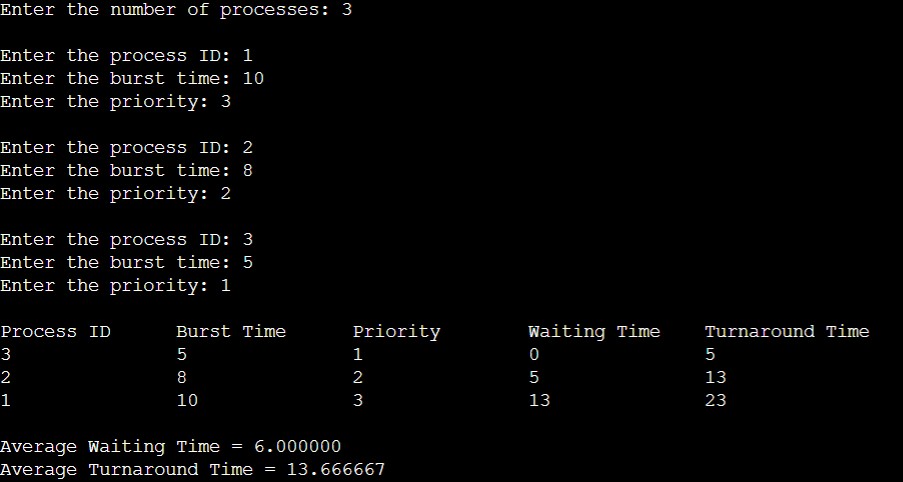
}

findavgTime(processes, n, burst\_time, quantum); return 0;

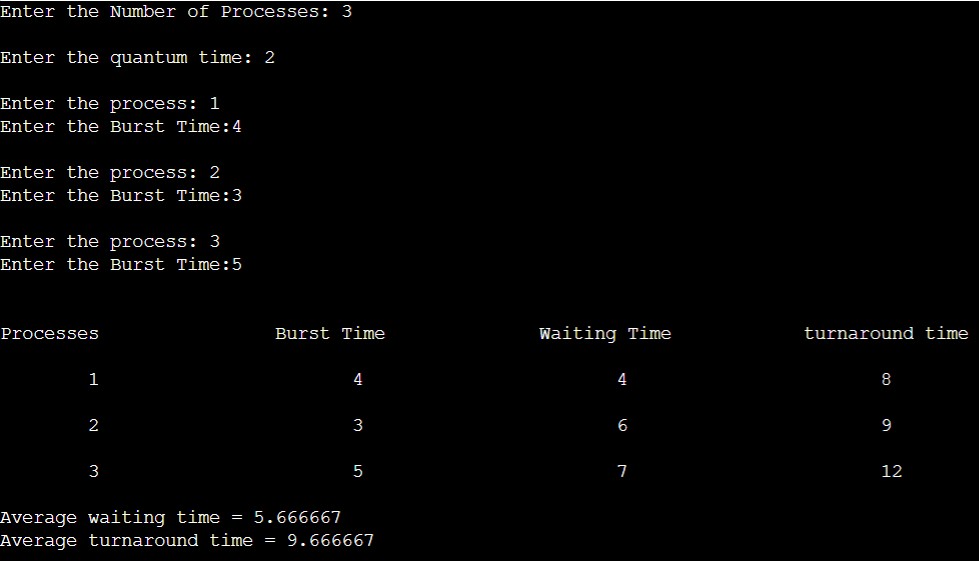
}

## Output:

1. **Priority (Non-pre-emptive)**



1. **Round Robin (Non-pre-emptive)**



# Experiment - 3

## Question:

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>

struct process { int pid;

int arrival\_time; int burst\_time; int priority;

int waiting\_time;

int turnaround\_time;

};

void FCFS(struct process \*queue, int n) { int i, j;

struct process temp; for (i = 0; i < n; i++) {

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

if (queue[i].arrival\_time > queue[j].arrival\_time) { temp = queue[i];

queue[i] = queue[j]; queue[j] = temp;

}

}

}

}

int main() { int n, i;

struct process \*system\_queue, \*user\_queue; int system\_n = 0, user\_n = 0;

float avg\_waiting\_time = 0, avg\_turnaround\_time = 0;

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

system\_queue = (struct process \*) malloc(n \* sizeof(struct process)); user\_queue = (struct process \*) malloc(n \* sizeof(struct process));

for (i = 0; i < n; i++) { struct process p;

printf("Enter arrival time, burst time, and priority (0-System/1-User) for process %d: ", i + 1);

scanf("%d %d %d", &p.arrival\_time, &p.burst\_time, &p.priority); p.pid = i + 1;

p.waiting\_time = 0;

p.turnaround\_time = 0; if (p.priority == 0) {

system\_queue[system\_n++] = p;

} else {

user\_queue[user\_n++] = p;

}

}

FCFS(system\_queue, system\_n); FCFS(user\_queue, user\_n);

int time = 0; int s=0,u=0;

while(s<system\_n || u<user\_n){ if(system\_queue[s].arrival\_time <= time){

if(user\_queue[u].arrival\_time <= time && user\_queue[u].arrival\_time < system\_queue[s].arrival\_time){

user\_queue[u].waiting\_time = time - user\_queue[u].arrival\_time; time += user\_queue[u].burst\_time; user\_queue[u].turnaround\_time = user\_queue[u].waiting\_time +

user\_queue[u].burst\_time;

avg\_waiting\_time += user\_queue[u].waiting\_time; avg\_turnaround\_time += user\_queue[u].turnaround\_time; u++;

}

else{

system\_queue[s].waiting\_time = time - system\_queue[s].arrival\_time; time += system\_queue[s].burst\_time; system\_queue[s].turnaround\_time = system\_queue[s].waiting\_time +

system\_queue[s].burst\_time;

avg\_waiting\_time += system\_queue[s].waiting\_time; avg\_turnaround\_time += system\_queue[s].turnaround\_time; s++;

}

}

else if(user\_queue[u].arrival\_time <= time){ user\_queue[u].waiting\_time = time - user\_queue[u].arrival\_time;

time += user\_queue[u].burst\_time;

user\_queue[u].turnaround\_time = user\_queue[u].waiting\_time + user\_queue[u].burst\_time;

avg\_waiting\_time += user\_queue[u].waiting\_time; avg\_turnaround\_time += user\_queue[u].turnaround\_time; u++;

}

else{

if(system\_queue[s].arrival\_time <= user\_queue[u].arrival\_time){ time = system\_queue[s].arrival\_time;

}

else{

time = user\_queue[u].arrival\_time;

}

}

}

avg\_waiting\_time /= n; avg\_turnaround\_time /= n;

printf("PID\tBurst Time\tPriority\tQueue Type\tWaiting Time\tTurnaround Time\n"); for (i = 0; i < system\_n; i++) {

printf("%d\t%d\t\t%d\t\tSystem\t\t%d\t\t%d\n", system\_queue[i].pid, system\_queue[i].burst\_time, system\_queue[i].priority, system\_queue[i].waiting\_time, system\_queue[i].turnaround\_time);

}

for (i = 0; i < user\_n; i++) { printf("%d\t%d\t\t%d\t\tUser\t\t%d\t\t%d\n", user\_queue[i].pid,

user\_queue[i].burst\_time, user\_queue[i].priority, user\_queue[i].waiting\_time, user\_queue[i].turnaround\_time);

}

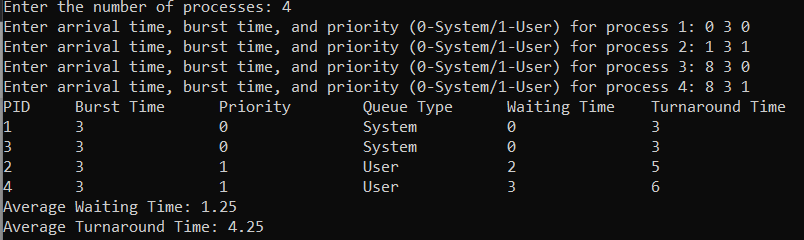
printf("Average Waiting Time: %.2f\n", avg\_waiting\_time); printf("Average Turnaround Time: %.2f\n", avg\_turnaround\_time);

free(system\_queue); free(user\_queue);

return 0;

}

* + 1. **Output:**



# Experiment - 4

## Question:

Write a C program to simulate Real-Time CPU Scheduling algorithms:

1. Rate- Monotonic
2. Earliest-deadline First
3. Proportional scheduling

## Code:

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

#define MAX\_PROCESS 10 typedef struct {

int id;

int burst\_time; float priority;

} Task;

int num\_of\_process;

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

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)

{

exit(0);

}

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

{

printf("\nProcess %d:\n", i + 1); printf("==> Execution time: "); scanf("%d", &execution\_time[i]); remain\_time[i] = execution\_time[i]; if (selected\_algo == 2)

{

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

}

else

{

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

}

}

}

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;

}

int get\_observation\_time(int selected\_algo)

{

if (selected\_algo == 1)

{

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

}

else if (selected\_algo == 2)

{

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

}

}

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;

int m = (float) (n \* (pow(2, 1.0 / n) - 1)); if (utilization > m)

{

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

}

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; 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);

}

void earliest\_deadline\_first(int time){ float utilization = 0;

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

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

}

int n = num\_of\_process;

int process[num\_of\_process];

int max\_deadline, current\_process=0, min\_deadline,process\_list[time]; bool is\_ready[num\_of\_process];

for(int i=0; i<num\_of\_process; i++){ is\_ready[i] = true;

process[i] = i+1;

}

max\_deadline=deadline[0];

for(int i=1; i<num\_of\_process; i++){ if(deadline[i] > max\_deadline) max\_deadline = deadline[i];

}

for(int i=0; i<num\_of\_process; i++){ for(int j=i+1; j<num\_of\_process; j++){

if(deadline[j] < deadline[i]){ int temp = execution\_time[j];

execution\_time[j] = execution\_time[i]; execution\_time[i] = temp;

temp = deadline[j]; deadline[j] = deadline[i]; deadline[i] = temp;

temp = process[j]; process[j] = process[i]; process[i] = temp;

}

}

}

for(int i=0; i<num\_of\_process; i++){ remain\_time[i] = execution\_time[i]; remain\_deadline[i] = deadline[i];

}

for (int t = 0; t < time; t++){ if(current\_process != -1){

--execution\_time[current\_process]; process\_list[t] = process[current\_process];

}

else

process\_list[t] = 0;

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

--deadline[i];

if((execution\_time[i] == 0) && is\_ready[i]){ deadline[i] += remain\_deadline[i]; is\_ready[i] = false;

}

if((deadline[i] <= remain\_deadline[i]) && (is\_ready[i] == false)){ execution\_time[i] = remain\_time[i];

is\_ready[i] = true;

}

}

min\_deadline = max\_deadline; current\_process = -1;

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

if((deadline[i] <= min\_deadline) && (execution\_time[i] > 0)){ current\_process = i;

min\_deadline = deadline[i];

}

}

}

print\_schedule(process\_list, time);

}

void proportionalScheduling() { int n;

printf("Enter the number of tasks: "); scanf("%d", &n);

Task tasks[n];

printf("Enter burst time and priority for each task:\n"); for (int i = 0; i < n; i++) {

tasks[i].id = i + 1;

printf("Task %d - Burst Time: ", tasks[i].id); scanf("%d", &tasks[i].burst\_time); printf("Task %d - Priority: ", tasks[i].id); scanf("%f", &tasks[i].priority);

}

// Sort tasks based on priority (ascending order) for (int i = 0; i < n - 1; i++) {

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

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

// Swap tasks

Task temp = tasks[j]; tasks[j] = tasks[j + 1]; tasks[j + 1] = temp;

}

}

}

printf("\nProportional Scheduling:\n");

int total\_burst\_time = 0; float total\_priority = 0.0;

for (int i = 0; i < n; i++) { total\_burst\_time += tasks[i].burst\_time; total\_priority += tasks[i].priority;

}

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

float time\_slice = (tasks[i].priority / total\_priority) \* total\_burst\_time; printf("Task %d executes for %.2f units of time\n", tasks[i].id, time\_slice);

}

}

int main()

{

int option;

int observation\_time;

while (1)

{

printf("\n1. Rate Monotonic\n2. Earliest Deadline first\n3. Proportional Scheduling\n\nEnter your choice: ");

scanf("%d", &option); switch(option)

{

case 1: get\_process\_info(option);

observation\_time = get\_observation\_time(option); rate\_monotonic(observation\_time);

break;

case 2: get\_process\_info(option);

observation\_time = get\_observation\_time(option); earliest\_deadline\_first(observation\_time);

break;

case 3: proportionalScheduling(); break;

case 4: exit (0);

default: printf("\nInvalid Statement");

}

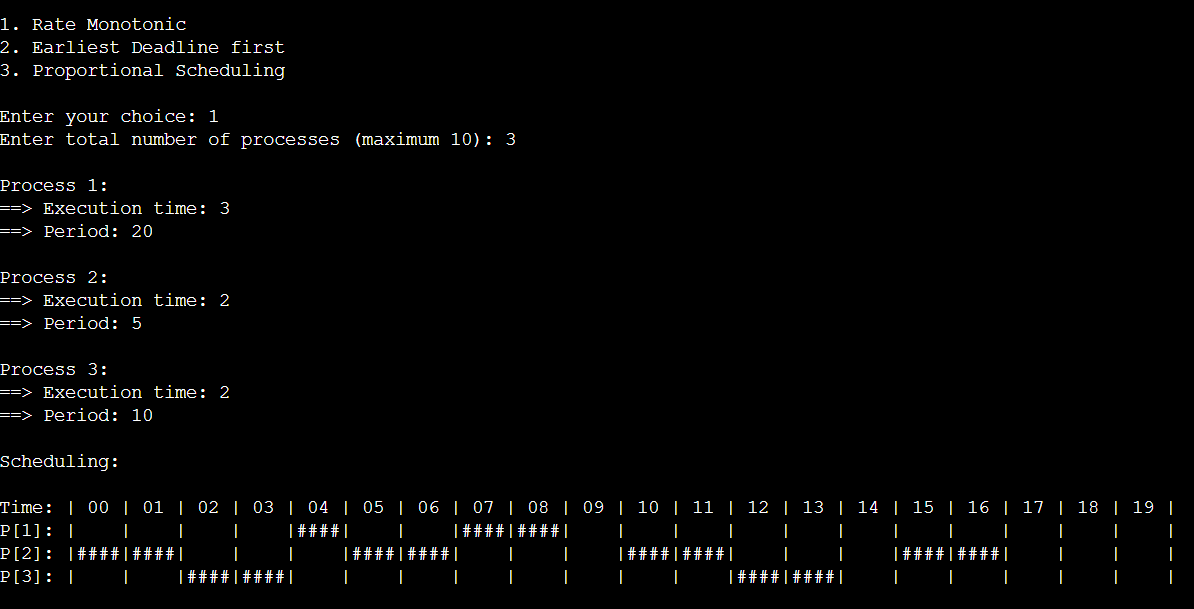
}

return 0;

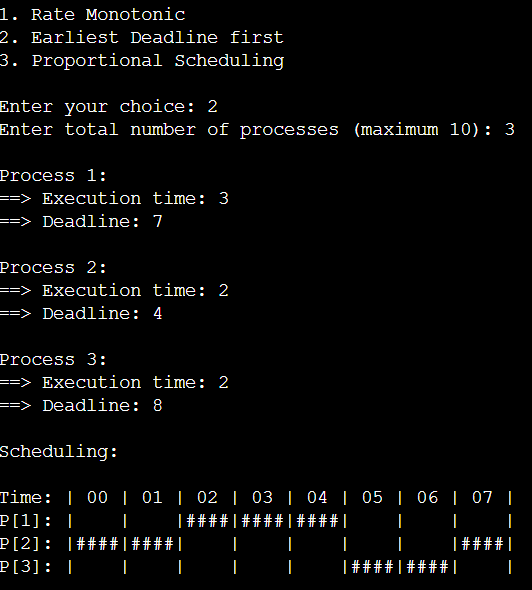
}

## Output:

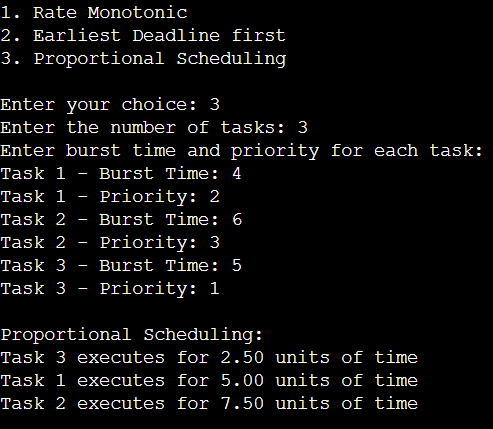
1. **Rate Monotonic:**



## Earliest Deadline First:



1. **Proportional Scheduling:**



# Experiment - 5

## Question:

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

* + 1. **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\n2.Consumer\n3.Exit"); while(1)

{

printf("\nEnter your choice: "); scanf("%d&",&n);

switch(n)

{

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

else

printf("Buffer is full!!"); break;

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

else

printf("Buffer is empty!!"); break;

case 3: exit(0); break;

}

}

return 0;

}

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("\nProducer produces the item %d",x); mutex=signal(mutex);

}

void consumer()

{

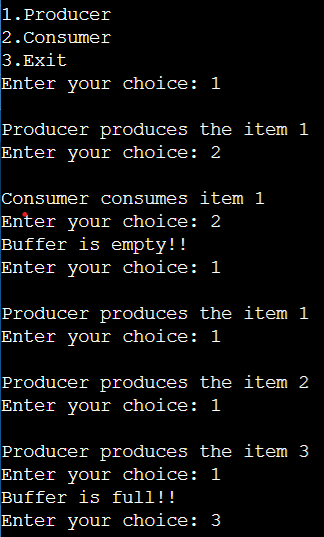
mutex=wait(mutex); full=wait(full); empty=signal(empty);

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

mutex=signal(mutex);

}

* + 1. **Output:**



# Experiment - 6

## Question:

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

## Code:

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

#define N 5

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (num\_of\_philosopher + 4) % N #define RIGHT (num\_of\_philosopher + 1) % N

int state[N];

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

sem\_t mutex; sem\_t S[N];

void test(int num\_of\_philosopher)

{

if (state[num\_of\_philosopher] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)

{

state[num\_of\_philosopher] = EATING; sleep(2);

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

+1, LEFT +1, num\_of\_philosopher +1);

printf("Philosopher %d is Eating\n", num\_of\_philosopher +1);

sem\_post(&S[num\_of\_philosopher]);

}

}

void take\_fork(int num\_of\_philosopher)

{

sem\_wait(&mutex); state[num\_of\_philosopher] = HUNGRY;

printf("Philosopher %d is Hungry\n", num\_of\_philosopher +1); test(num\_of\_philosopher);

sem\_post(&mutex); sem\_wait(&S[num\_of\_philosopher]); sleep(1);

}

void put\_fork(int num\_of\_philosopher)

{

sem\_wait(&mutex); state[num\_of\_philosopher] = THINKING;

printf("Philosopher %d putting fork %d and %d down\n",num\_of\_philosopher +1, LEFT +1, num\_of\_philosopher +1);

printf("Philosopher %d is thinking\n", num\_of\_philosopher +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];

sem\_init(&mutex,0,1); for (i =0; i < N; i++)

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

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

{

pthread\_create(&thread\_id[i],NULL,philosopher, &phil[i]);

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

}

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

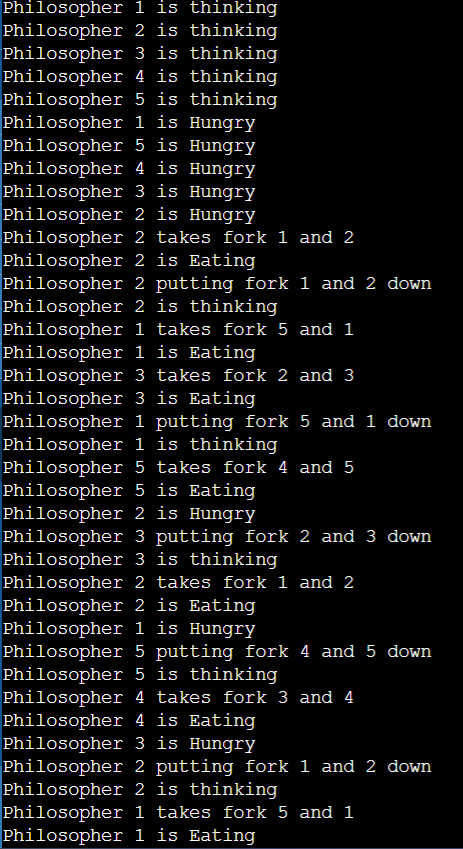
{

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

}

}

* + 1. **Output:**



# Experiment - 7

## Question:

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

* + 1. **Code**: #include <stdio.h>

int main()

{

int n, m, i, j, k;

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

printf("Enter the number of resources: "); scanf("%d", &m);

int allocation[n][m];

printf("Enter the Allocation Matrix:\n"); for (i = 0; i < n; i++)

{

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

{

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

}

}

int max[n][m];

printf("Enter the MAX Matrix:\n"); for (i = 0; i < n; i++)

{

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

{

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

}

}

int available[m];

printf("Enter the Available Resources:\n"); for (i = 0; i < m; i++)

{

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

}

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] - allocation[i][j];

}

}

int y = 0;

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

{

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

{

if (f[i] == 0)

{

int flag = 0;

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

{

if (need[i][j] > available[j])

{

flag = 1; break;

}

}

if (flag == 0)

{

ans[ind++] = i;

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

{

available[y] += allocation[i][y];

}

f[i] = 1;

}

}

}

}

int flag = 1;

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

{

if (f[i] == 0)

{

flag = 0;

printf("The following system is not safe\n"); break;

}

}

if (flag == 1)

{

printf("Following is the SAFE Sequence\n"); for (i = 0; i < n - 1; i++)

{

printf(" P%d ->", ans[i]);

}

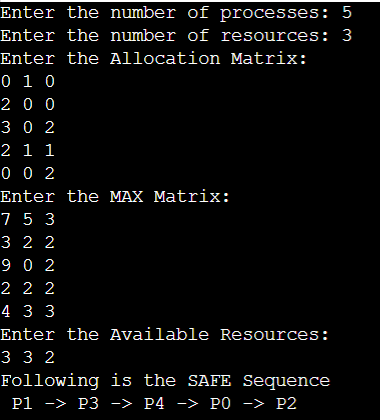
printf(" P%d\n", ans[n - 1]);

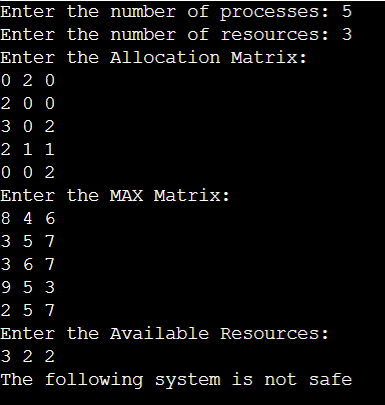
}

return 0;

}

* + 1. **Output:**





# \Experiment - 8

## Question:

Write a C program to simulate deadlock detection.

* + 1. **Code**: #include<stdio.h>

int max[100][100];

int allocation[100][100]; int need[100][100];

int available[100]; int n,r;

int main()

{

int i,j;

printf("Deadlock Detection\n"); input();

show();

cal(); return 0;

}

void input()

{

int i,j;

printf("Enter the no of Processes: "); scanf("%d",&n);

printf("Enter the no of resource instances: "); scanf("%d",&r);

printf("Enter the Max Matrix:\n"); for(i=0;i<n;i++)

{

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

{

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

}

}

printf("Enter the Allocation Matrix:\n"); for(i=0;i<n;i++)

{

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

{

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

}

}

printf("Enter the available Resources:\n"); for(j=0;j<r;j++)

{

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

}

}

void show()

{

int i,j;

printf("Process\t Allocation\t Max\t Available\t"); for(i=0;i<n;i++)

{

printf("\nP%d\t ",i+1); for(j=0;j<r;j++)

{

printf("%d ",allocation[i][j]);

}

printf("\t"); for(j=0;j<r;j++)

{

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

}

printf("\t"); if(i==0)

{

for(j=0;j<r;j++) printf("%d ",available[j]);

}

}

}

void cal()

{

int finish[100],temp,need[100][100],flag=1,k,c1=0; int dead[100];

int safe[100]; int i,j; for(i=0;i<n;i++)

{

finish[i]=0;

}

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

{

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

{

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

}

}

while(flag)

{

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

{

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

{

if((finish[i]==0)&&(need[i][j]<=available[j]))

{

c++;

if(c==r)

{

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

{

available[k]+=allocation[i][j]; finish[i]=1;

flag=1;

}

if(finish[i]==1)

{

i=n;

}

}

}

}

}

}

j=0;

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

{

if(finish[i]==0)

{

dead[j]=i; j++;

flag=1;

}

}

if(flag==1)

{

printf("\n\nSystem is in Deadlock and the Deadlock process are\n"); for(i=0;i<n;i++)

{

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

}

}

else

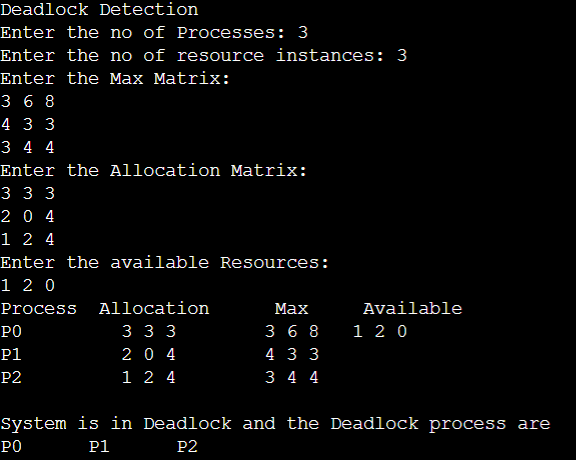
{

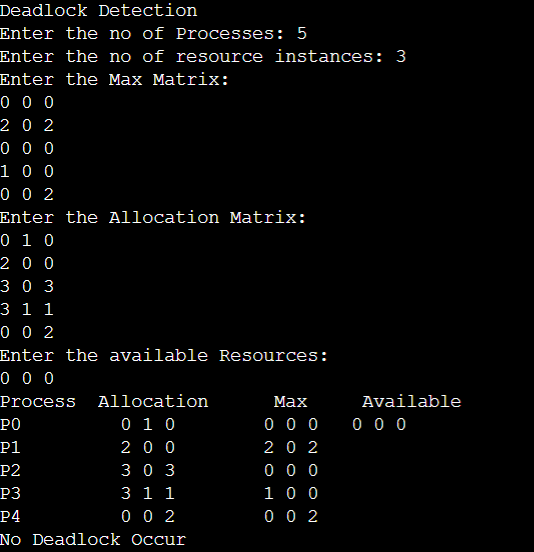
printf("\nNo Deadlock Occur");

}

}

* + 1. **Output:**





# Experiment - 9

## Question:

Write a C program to simulate the following contiguous memory allocation techniques:

1. Worst-fit
2. Best-fit
3. First-fit

## Code:

#include <stdio.h> #define max 25

void firstFit(int b[], int nb, int f[], int nf); void worstFit(int b[], int nb, int f[], int nf); void bestFit(int b[], int nb, int f[], int nf);

int main()

{

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

printf("Memory Management Schemes\n");

printf("\nEnter the number of blocks:"); scanf("%d", &nb);

printf("Enter the number of files:"); scanf("%d", &nf);

printf("\nEnter the size of the blocks:\n"); for (int i = 1; i <= nb; i++)

{

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

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

}

printf("\nEnter the size of the files:\n"); for (int i = 1; i <= nf; i++)

{

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

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

}

printf("\nMemory Management Scheme - First Fit"); firstFit(b, nb, f, nf);

printf("\n\nMemory Management Scheme - Worst Fit"); worstFit(b, nb, f, nf);

printf("\n\nMemory Management Scheme - Best Fit"); bestFit(b, nb, f, nf);

return 0;

}

void firstFit(int b[], int nb, int f[], int nf)

{

int bf[max] = {0};

int ff[max] = {0}; int frag[max], i, j;

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

{

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

{

if (bf[j] != 1 && b[j] >= f[i])

{

ff[i] = j;

bf[j] = 1;

frag[i] = b[j] - f[i]; break;

}

}

}

printf("\nFile\_no:\tFile\_size:\tBlock\_no:\tBlock\_size:\tFragment"); for (i = 1; i <= nf; i++)

{

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

}

}

void worstFit(int b[], int nb, int f[], int nf)

{

int bf[max] = {0};

int ff[max] = {0};

int frag[max], i, j, temp, highest = 0;

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\_no:\tFile\_size:\tBlock\_no:\tBlock\_size:\tFragment"); for (i = 1; i <= nf; i++)

{

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

}

}

void bestFit(int b[], int nb, int f[], int nf)

{

int bf[max] = {0};

int ff[max] = {0};

int frag[max], i, j, temp, lowest = 10000;

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\_no:\tFile\_size:\tBlock\_no:\tBlock\_size:\tFragment"); for (i = 1; i <= nf && ff[i] != 0; i++)

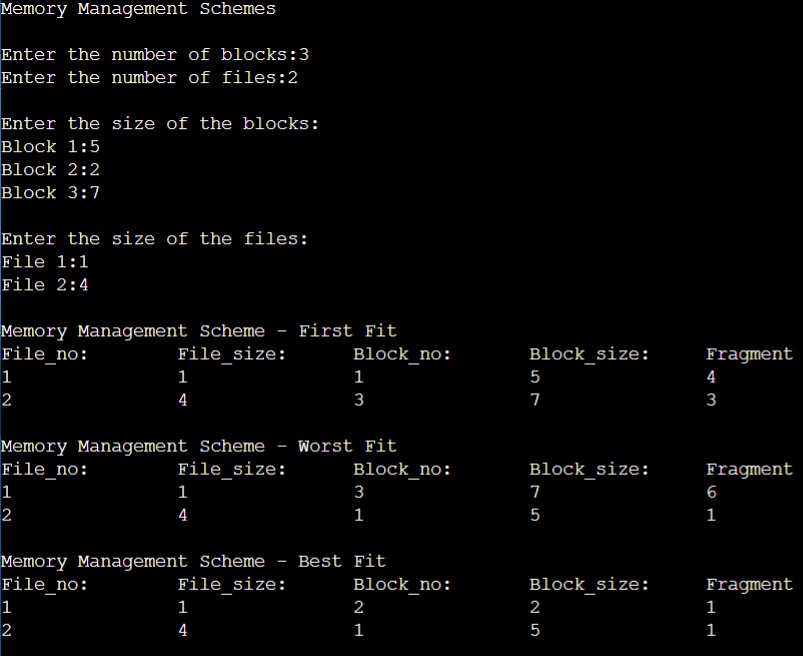
{

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

}

}

* + 1. **Output:**



# Experiment - 10

## Question:

Write a C program to simulate paging technique of memory management.

* + 1. **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("\nEnter Page size: "); scanf("%d",&ps);

printf("\nEnter 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("\nPage No\t\tFrame No\n-------\t\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(page[pno]==-1)

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

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

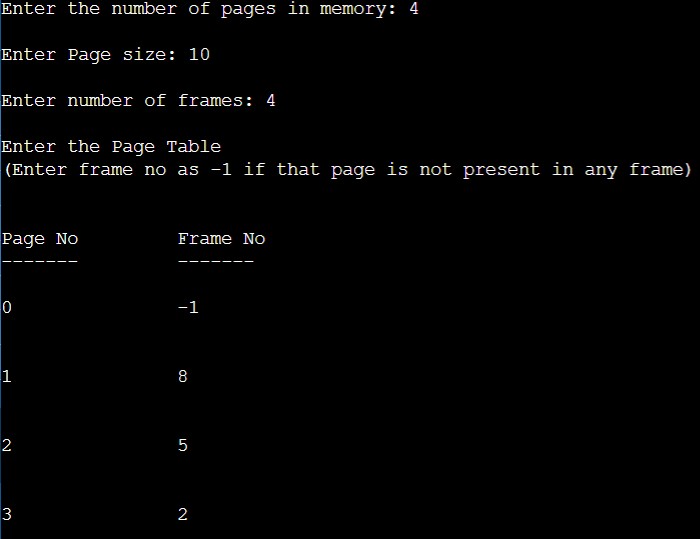
printf("\n\nDo you want to continue(1/0)?:"); scanf("%d",&choice);

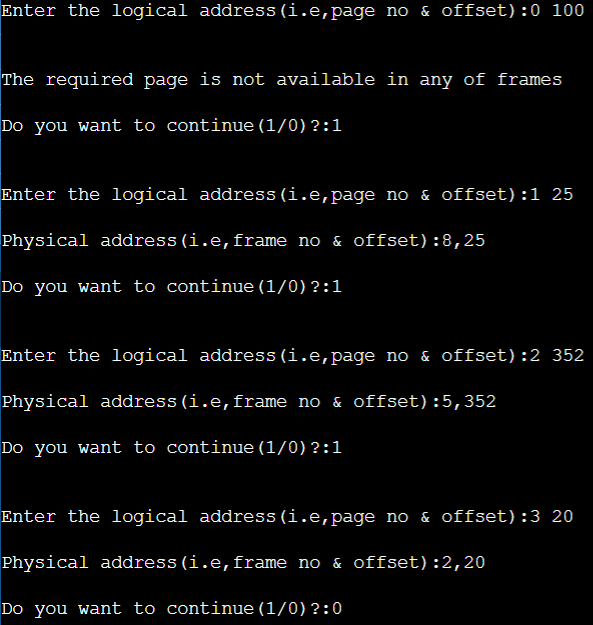
}while(choice==1);

return 1;

}

* + 1. **Output:**





# Experiment - 11

## Question:

Write a C program to simulate page replacement algorithms:

1. FIFO
2. LRU
3. Optimal
   * 1. **Code**: #include<stdio.h> int n, nf, i, j, k; int in[100];

int p[50]; int hit=0;

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()

{

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\n5.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;

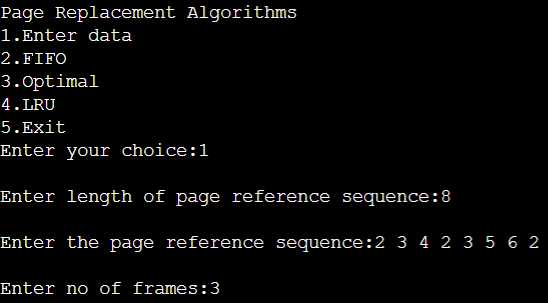
}

}

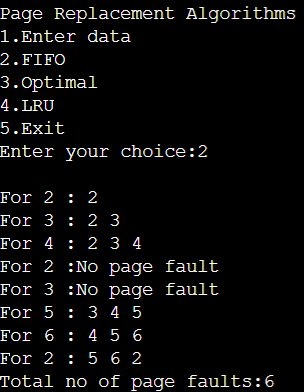
}

## Output:

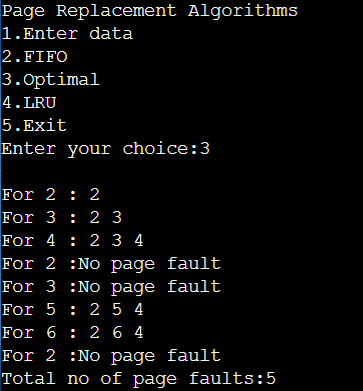
1. **Enter Data:**



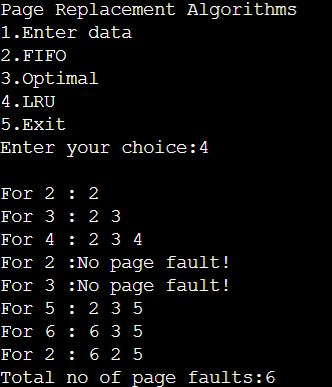
## FIFO:



1. **OPTIMAL:**



1. **LRU:**



# Experiment - 12

## Question:

Write a C program to simulate the following file allocation strategies:

1. Sequential
2. Indexed
3. Linked

## Code:

1. **Sequential:** #include<stdio.h> #include<string.h>

struct fileTable

{ char name[20]; int sb, nob; } ft[30];

void main() {

int i, j, n; char s[20]; printf("Enter no of files :"); scanf("%d",&n); for(i=0;i<n;i++)

{

printf("\nEnter file name %d :",i+1); scanf("%s",ft[i].name);

printf("Enter starting block of file %d :",i+1); scanf("%d",&ft[i].sb);

printf("Enter no of blocks in file %d :",i+1); scanf("%d",&ft[i].nob);

}

printf("\nEnter the file name to be searched -- "); scanf("%s",s);

for(i=0;i<n;i++) if(strcmp(s, ft[i].name)==0) break;

if(i==n)

printf("\nFile Not Found"); else

{

printf("\nFILE NAME START BLOCK NO OF BLOCKS BLOCKS OCCUPIED\n");

printf("\n%s\t\t%d\t\t%d\t",ft[i].name,ft[i].sb,ft[i].nob); for(j=0;j<ft[i].nob;j++)

printf("%d, ",ft[i].sb+j);

}

}

1. **Indexed:** #include<stdio.h> #include<conio.h>

struct fileTable

{

char name[20];

int nob, blocks[30];

}ft[30];

void main()

{

int i, j, n; char s[20]; printf("Enter no of files :"); scanf("%d",&n); for(i=0;i<n;i++)

{

printf("\nEnter file name %d :",i+1); scanf("%s",ft[i].name);

printf("Enter no of blocks in file %d :",i+1); scanf("%d",&ft[i].nob);

printf("Enter the blocks of the file :"); for(j=0;j<ft[i].nob;j++)

scanf("%d",&ft[i].blocks[j]);

}

printf("\nEnter the file name to be searched -- "); scanf("%s",s); for(i=0;i<n;i++)

if(strcmp(s, ft[i].name)==0) break;

if(i==n)

printf("\nFile Not Found");

else

{

printf("\nFILE NAME NO OF BLOCKS BLOCKS OCCUPIED");

printf("\n %s\t\t%d\t",ft[i].name,ft[i].nob); for(j=0;j<ft[i].nob;j++)

printf("%d, ",ft[i].blocks[j]);

}

}

1. **Linked:**

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

struct fileTable

{

char name[20]; int nob;

struct block \*sb;

}ft[30];

struct block

{

int bno;

struct block \*next;

};

void main()

{

int i, j, n; char s[20];

struct block \*temp; printf("Enter no of files :"); scanf("%d",&n); for(i=0;i<n;i++)

{

printf("\nEnter file name %d :",i+1); scanf("%s",ft[i].name);

printf("Enter no of blocks in file %d :",i+1); scanf("%d",&ft[i].nob);

ft[i].sb=(struct block\*)malloc(sizeof(struct block)); temp = ft[i].sb;

printf("Enter the blocks of the file :"); scanf("%d",&temp->bno);

temp->next=NULL; for(j=1;j<ft[i].nob;j++)

{

temp->next = (struct block\*)malloc(sizeof(struct block)); temp = temp->next;

scanf("%d",&temp->bno);

}

temp->next = NULL;

}

printf("\nEnter the file name to be searched -- "); scanf("%s",s);

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

if(strcmp(s, ft[i].name)==0) break;

if(i==n)

printf("\nFile Not Found");

else

{

printf("\nFILE NAME NO OF BLOCKS BLOCKS OCCUPIED");

printf("\n %s\t\t%d\t",ft[i].name,ft[i].nob); temp=ft[i].sb;

for(j=0;j<ft[i].nob;j++)

{

printf("%d->",temp->bno); temp = temp->next;

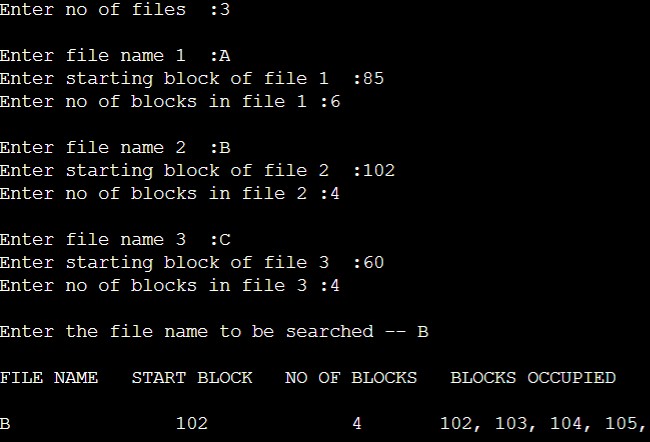
}

}

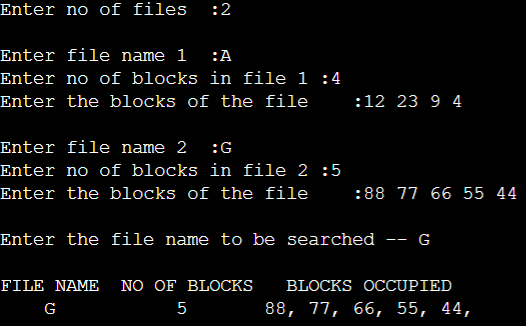
}

## Output:

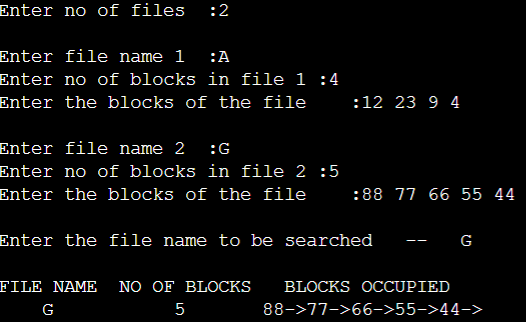
1. **Sequential:**



## Indexed:



1. **Linked:**



# Experiment - 13

## Question:

Write a C program to simulate the following file organization techniques:

1. Single level directory
2. Two level directory
3. Hierarchical

## Code:

1. **Single Level Directory:**

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

struct

{

char dname[10],fname[10][10]; int fcnt;

} dir;

void main()

{

int i,ch; char f[30]; dir.fcnt = 0;

printf("\nEnter name of directory -- "); scanf("%s", dir.dname);

while(1)

{

printf("\n\n1. Create File\t2. Delete File\t3. Search File \n4. Display Files\t5.

Exit\nEnter your choice -- "); scanf("%d",&ch);

switch(ch)

{

case 1: printf("\nEnter the name of the file -- "); scanf("%s",dir.fname[dir.fcnt]); dir.fcnt++;

break;

case 2: printf("\nEnter the name of the file -- "); scanf("%s",f);

for(i=0;i<dir.fcnt;i++)

{

if(strcmp(f, dir.fname[i])==0)

{

printf("File %s is deleted ",f); strcpy(dir.fname[i],dir.fname[dir.fcnt-1]); break;

}

}

if(i==dir.fcnt)

printf("File %s not found",f); else

dir.fcnt--; break;

case 3: printf("\nEnter the name of the file -- "); scanf("%s",f);

for(i=0;i<dir.fcnt;i++)

{

if(strcmp(f, dir.fname[i])==0)

{

printf("File %s is found ", f); break;

}

}

if(i==dir.fcnt)

printf("File %s not found",f); break;

case 4: if(dir.fcnt==0)

printf("\nDirectory Empty"); else

{

printf("\nThe Files are -- "); for(i=0;i<dir.fcnt;i++)

printf("\t%s",dir.fname[i]);

}

break;

default: exit(0);

}

}

}

1. **Two Level Directory:**

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

struct

{

char dname[10],fname[10][10]; int fcnt;

}dir[10];

void main()

{

int i,ch,dcnt,k; char f[30], d[30]; dcnt=0;

while(1)

{

printf("\n1. Create Directory\t2. Create File\t3. Delete File"); printf("\n4. Search File\t\t5. Display\t6. Exit\nEnter your choice --"); scanf("%d",&ch);

switch(ch)

{

case 1: printf("\nEnter name of directory -- "); scanf("%s", dir[dcnt].dname); dir[dcnt].fcnt=0;

dcnt++;

printf("Directory created"); break;

case 2: printf("\nEnter name of the directory -- "); scanf("%s",d);

for(i=0;i<dcnt;i++) if(strcmp(d,dir[i].dname)==0)

{

printf("Enter name of the file -- "); scanf("%s",dir[i].fname[dir[i].fcnt]); dir[i].fcnt++;

printf("File created"); break;

}

if(i==dcnt)

printf("Directory %s not found",d); break;

case 3: printf("\nEnter name of the directory -- "); scanf("%s",d);

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

{

if(strcmp(d,dir[i].dname)==0)

{

printf("Enter name of the file -- "); scanf("%s",f); for(k=0;k<dir[i].fcnt;k++)

{

if(strcmp(f, dir[i].fname[k])==0)

{

printf("File %s is deleted ",f); dir[i].fcnt--;

strcpy(dir[i].fname[k],dir[i].fname[dir[i].fcnt]); goto jmp;

}

}

printf("File %s not found",f); goto jmp;

}

}

printf("Directory %s not found",d); jmp : break;

case 4: printf("\nEnter name of the directory -- "); scanf("%s",d);

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

{

if(strcmp(d,dir[i].dname)==0)

{

printf("Enter the name of the file -- "); scanf("%s",f); for(k=0;k<dir[i].fcnt;k++)

{

if(strcmp(f, dir[i].fname[k])==0)

{

printf("File %s is found ",f); goto jmp1;

}

}

printf("File %s not found",f); goto jmp1;

}

}

printf("Directory %s not found",d); jmp1: break;

case 5: if(dcnt==0)

printf("\nNo Directory's "); else

{

printf("\nDirectory\tFiles"); for(i=0;i<dcnt;i++)

{

printf("\n%s\t\t",dir[i].dname); for(k=0;k<dir[i].fcnt;k++) printf("\t%s",dir[i].fname[k]);

}

}

break; default:exit(0);

}

}

}

1. **Hierarchical:** #include<stdio.h> #include<string.h> #include<stdlib.h>

//#include<graphics.h>

struct tree\_element

{

char name[20];

int x,y,ftype,lx,rx,nc,level; struct tree\_element \*link[5];

};

typedef struct tree\_element node;

void main()

{

int gm; node \*root;

root=NULL; create(&root,0,"root",0,639,320);

//initgraph(&gd,&gm,"c:\\tc\\BGI"); display(root);

//closegraph();

}

create(node \*\*root,int lev,char \*dname,int lx,int rx,int x)

{

int i,gap; if(\*root==NULL)

{

(\*root)=(node \*)malloc(sizeof(node));

printf("Enter name of dir/file(under %s) :",dname); fflush(stdin);

gets((\*root)->name);

printf("enter 1 for Dir/2 forfile :"); scanf("%d",&(\*root)->ftype);

(\*root)->level=lev; (\*root)->y=50+lev\*50; (\*root)->x=x;

(\*root)->lx=lx; (\*root)->rx=rx;

for(i=0;i<5;i++)

(\*root)->link[i]=NULL;

if((\*root)->ftype==1)

{

printf("No of sub directories/files(for %s):",(\*root)->name); scanf("%d",&(\*root)->nc);

if((\*root)->nc==0) gap=rx-lx;

else

gap=(rx-lx)/(\*root)->nc; for(i=0;i<(\*root)->nc;i++)

create(&((\*root)->link[i]),lev+1,(\*root)->name,lx+gap\*i,lx+gap\*i+gap,lx+gap\*i+gap/2);

}

else (\*root)->nc=0;

}

}

/\*display(node \*root)

{

int i; settextstyle(2,0,4); settextjustify(1,1); setfillstyle(1,BLUE); setcolor(14);

if(root!=NULL)

{

for(i=0;i<root->nc;i++)

{

line(root->x,root->y,root->link[i]->x,root->link[i]->y);

}

if(root->ftype==1)

bar3d(root->x-20,root->y-10,root->x+20,root->y+10,0,0);

else

fillellipse(root->x,root->y,20,20);

outtextxy(root->x,root->y,root->name); for(i=0;i<root->nc;i++)

{

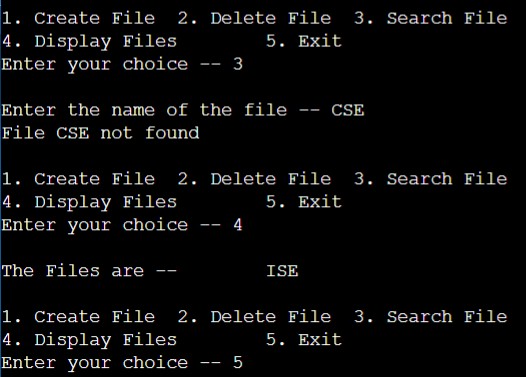
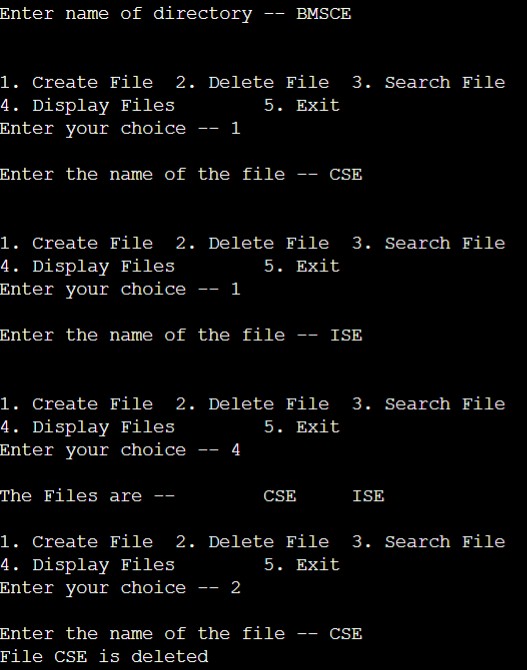
}

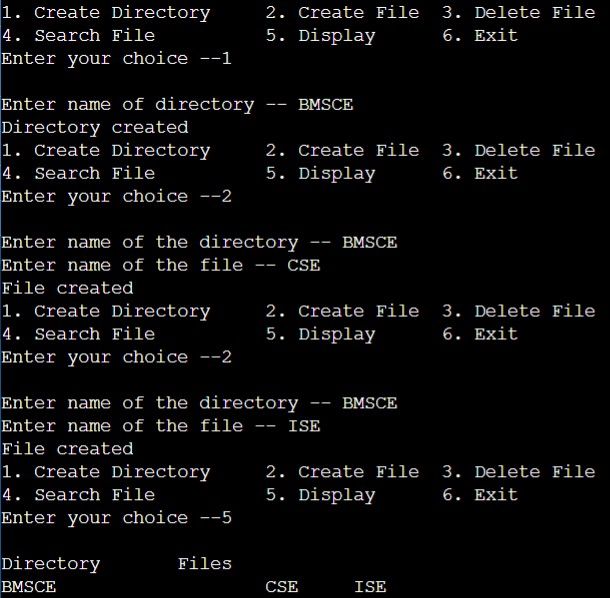
}\*/

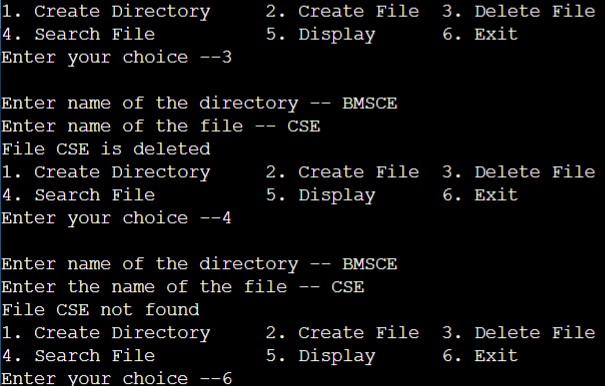
display(root->link[i]);

}

## Output:

1. **Single Level Directory:**
2. **Two Level Directory:**





# Experiment - 14

## Question:

Write a C program to simulate disk scheduling algorithms:

1. FCFS
2. SCAN
3. c-SCAN

## Code:

1. **FCFS:** #include<stdio.h> #include<stdlib.h> int main()

{

int RQ[100],i,n,TotalHeadMoment=0,initial; printf("Enter the number of Requests\n"); scanf("%d",&n);

printf("Enter the Requests sequence\n"); for(i=0;i<n;i++)

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

printf("Enter initial head position\n"); scanf("%d",&initial);

// logic for FCFS disk scheduling for(i=0;i<n;i++)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial); initial=RQ[i];

}

printf("Total head moment is %d",TotalHeadMoment); return 0;

}

1. **SCAN:** #include<stdio.h> #include<stdlib.h> int main()

{

int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move; printf("Enter the number of Requests\n"); scanf("%d",&n);

printf("Enter the Requests sequence\n"); for(i=0;i<n;i++)

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

printf("Enter initial head position\n"); scanf("%d",&initial);

printf("Enter total disk size\n"); scanf("%d",&size);

printf("Enter the head movement direction for high 1 and for low 0\n"); scanf("%d",&move);

// logic for Scan disk scheduling

/\*logic for sort the request array \*/ for(i=0;i<n;i++)

{

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

{

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

{

int temp; temp=RQ[j]; RQ[j]=RQ[j+1];

RQ[j+1]=temp;

}

}

}

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

{

if(initial<RQ[i])

{

index=i; break;

}

}

// if movement is towards high value if(move==1)

{

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial); initial=RQ[i];

}

// last movement for max size TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);

initial = size-1; for(i=index-1;i>=0;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial); initial=RQ[i];

}

}

// if movement is towards low value else

{

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial); initial=RQ[i];

}

// last movement for min size TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0); initial =0;

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial); initial=RQ[i];

}

}

printf("Total head movement is %d",TotalHeadMoment); return 0;

}

1. **c-SCAN:** #include<stdio.h> #include<stdlib.h> int main()

{

int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move; printf("Enter the number of Requests\n"); scanf("%d",&n);

printf("Enter the Requests sequence\n"); for(i=0;i<n;i++)

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

printf("Enter initial head position\n"); scanf("%d",&initial);

printf("Enter total disk size\n"); scanf("%d",&size);

printf("Enter the head movement direction for high 1 and for low 0\n");

scanf("%d",&move);

// logic for C-Scan disk scheduling

/\*logic for sort the request array \*/ for(i=0;i<n;i++)

{

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

{

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

{

int temp; temp=RQ[j]; RQ[j]=RQ[j+1];

RQ[j+1]=temp;

}

}

}

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

{

if(initial<RQ[i])

{

index=i; break;

}

}

// if movement is towards high value if(move==1)

{

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial); initial=RQ[i];

}

// last movement for max size TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);

/\*movement max to min disk \*/ TotalHeadMoment=TotalHeadMoment+abs(size-1-0); initial=0;

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial); initial=RQ[i];

}

}

// if movement is towards low value else

{

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial); initial=RQ[i];

}

// last movement for min size TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);

/\*movement min to max disk \*/ TotalHeadMoment=TotalHeadMoment+abs(size-1-0); initial =size-1;

for(i=n-1;i>=index;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial); initial=RQ[i];

}

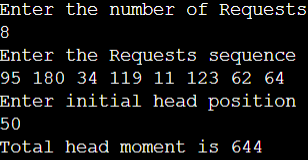
}

printf("Total head movement is %d",TotalHeadMoment); return 0;

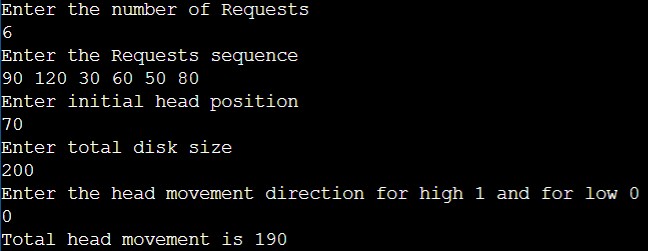
}

## Output:

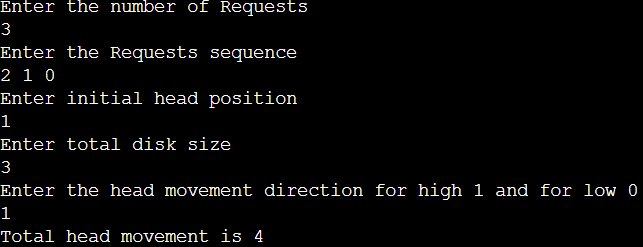
1. **FCFS:**



## SCAN:



1. **C-SCAN:**



# Experiment - 15

## Question:

Write a C program to simulate disk scheduling algorithms:

1. SSTF
2. LOOK
3. C-LOOK

## Code:

1. **SSTF:** #include<stdio.h> #include<stdlib.h> int main()

{

int RQ[100],i,n,TotalHeadMoment=0,initial,count=0; printf("Enter the number of Requests\n"); scanf("%d",&n);

printf("Enter the Requests sequence\n"); for(i=0;i<n;i++)

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

printf("Enter initial head position\n"); scanf("%d",&initial);

// logic for sstf disk scheduling

/\* loop will execute until all process is completed\*/ while(count!=n)

{

int min=1000,d,index; for(i=0;i<n;i++)

{

d=abs(RQ[i]-initial); if(min>d)

{

min=d; index=i;

}

}

TotalHeadMoment=TotalHeadMoment+min; initial=RQ[index];

// 1000 is for max

// you can use any number RQ[index]=1000; count++;

}

printf("Total head movement is %d",TotalHeadMoment); return 0;

}

1. **LOOK:** #include<stdio.h> #include<stdlib.h> int main()

{

int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move; printf("Enter the number of Requests\n"); scanf("%d",&n);

printf("Enter the Requests sequence\n"); for(i=0;i<n;i++)

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

printf("Enter initial head position\n"); scanf("%d",&initial);

printf("Enter total disk size\n"); scanf("%d",&size);

printf("Enter the head movement direction for high 1 and for low 0\n"); scanf("%d",&move);

// logic for look disk scheduling

/\*logic for sort the request array \*/ for(i=0;i<n;i++)

{

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

{

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

{

int temp; temp=RQ[j]; RQ[j]=RQ[j+1];

RQ[j+1]=temp;

}

}

}

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

{

if(initial<RQ[i])

{

index=i; break;

}

}

// if movement is towards high value if(move==1)

{

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial); initial=RQ[i];

}

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial); initial=RQ[i];

}

}

// if movement is towards low value else

{

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial); initial=RQ[i];

}

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial); initial=RQ[i];

}

}

printf("Total head movement is %d",TotalHeadMoment); return 0;

}

1. **c-LOOK:** #include<stdio.h> #include<stdlib.h> int main()

{

int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move; printf("Enter the number of Requests\n"); scanf("%d",&n);

printf("Enter the Requests sequence\n"); for(i=0;i<n;i++)

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

printf("Enter initial head position\n"); scanf("%d",&initial);

printf("Enter total disk size\n"); scanf("%d",&size);

printf("Enter the head movement direction for high 1 and for low 0\n"); scanf("%d",&move);

// logic for C-look disk scheduling

/\*logic for sort the request array \*/ for(i=0;i<n;i++)

{

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

{

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

{

int temp; temp=RQ[j]; RQ[j]=RQ[j+1];

RQ[j+1]=temp;

}

}

}

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

{

if(initial<RQ[i])

{

index=i; break;

}

}

// if movement is towards high value if(move==1)

{

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial); initial=RQ[i];

}

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial); initial=RQ[i];

}

}

// if movement is towards low value else

{

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial); initial=RQ[i];

}

for(i=n-1;i>=index;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial); initial=RQ[i];

}

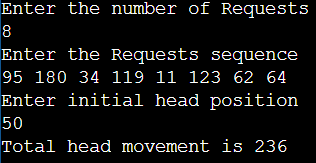
}

printf("Total head movement is %d",TotalHeadMoment); return 0;

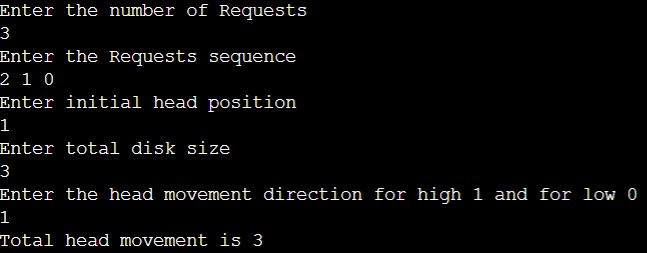
}

## Output:

1. **SSTF:**



## LOOK:



1. **c-LOOK:**

