**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

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



**LAB REPORT**

**on**

**Operating Systems**

**(23CS4PCOPS)**

***Submitted by:***

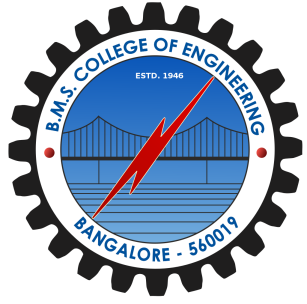
**ASHISH(1BM22CS056)**

***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**

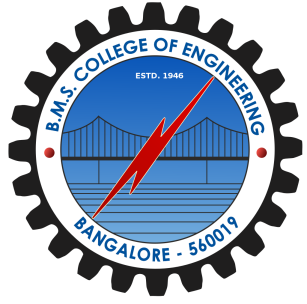
**June 2024 - August 2024**

**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 **Ashish (1BM22CS056),** 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 - (23CS4PCOPS)** work prescribed for the said degree.

**Basavaraj Jakkalli**              **Dr. Jyothi S Nayak**

Associate Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

**Table Of Contents**

| **Lab Program No.** | **Program Details** | **Page No.** |
| --- | --- | --- |
| 1 | FCFS AND SJF | 1-5 |
| 2 | PRIORITY AND ROUND ROBIN | 6-13 |
| 3 | RATE-MONOTONIC AND EARLIEST DEADLINE FIRST | 14-19 |
| 4 | PRODUCER-CONSUMER PROBLEM | 20-21 |
| 5 | DINERS-PHILOSOPHERS PROBLEM | 22-24 |
| 6 | BANKERS ALGORITHM(DEADLOCK AVOIDANCE) | 25-27 |
| 7 | DEADLOCK DETECTION | 28-30 |
| 8 | CONTIGUOUS MEMORY ALLOCATION(FIRST, BEST, WORST FIT) | 31-34 |
| 9 | PAGE REPLACEMENT(FIFO, LRU, OPTIMAL) | 35-40 |
| 10 | DISK SCHEDULING ALGORITHMS(FCFS, SCAN, C-SCAN) | 41-46 |

**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.

**LAB - 1**

**Question 1:**

**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, total = 0;

int Burst\_time[20], Arrival\_time[20], Waiting\_time[20], Turn\_around\_time[20], process[20];

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

void FCFS() {

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

int current\_time = 0;

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

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

if (Arrival\_time[i] > Arrival\_time[j]) {

temp = Arrival\_time[i];

Arrival\_time[i] = Arrival\_time[j];

Arrival\_time[j] = temp;

temp = Burst\_time[i];

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

Burst\_time[j] = temp;

temp = process[i];

process[i] = process[j];

process[j] = temp;

}

}

}

Waiting\_time[0] = 0;

current\_time = Arrival\_time[0] + Burst\_time[0];

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

if (current\_time < Arrival\_time[i]) {

current\_time = Arrival\_time[i];

}

Waiting\_time[i] = current\_time - Arrival\_time[i];

current\_time += Burst\_time[i];

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

}

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

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

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

total\_turnaround\_time += Turn\_around\_time[i];

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

}

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

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

printf("\nAverage Waiting Time: %.2f", avg\_Waiting\_time);

printf("\nAverage Turnaround Time: %.2f\n", avg\_Turn\_around\_time);

}

void SJF() {

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

int completed = 0, current\_time = 0, min\_index;

int is\_completed[20] = {0};

while (completed != n) {

int min\_burst\_time = 9999;

min\_index = -1;

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

if (Arrival\_time[i] <= current\_time && is\_completed[i] == 0) {

if (Burst\_time[i] < min\_burst\_time) {

min\_burst\_time = Burst\_time[i];

min\_index = i;

}

if (Burst\_time[i] == min\_burst\_time) {

if (Arrival\_time[i] < Arrival\_time[min\_index]) {

min\_burst\_time = Burst\_time[i];

min\_index = i;

}

}

}

}

if (min\_index != -1) {

Waiting\_time[min\_index] = current\_time - Arrival\_time[min\_index];

current\_time += Burst\_time[min\_index];

Turn\_around\_time[min\_index] = current\_time - Arrival\_time[min\_index];

total\_waiting\_time += Waiting\_time[min\_index];

total\_turnaround\_time += Turn\_around\_time[min\_index];

is\_completed[min\_index] = 1;

completed++;

} else {

current\_time++;

}

}

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

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

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

}

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

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

printf("\n\nAverage Waiting Time = %.2f", avg\_Waiting\_time);

printf("\nAverage Turnaround Time = %.2f\n", avg\_Turn\_around\_time);

}

int main() {

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

scanf("%d", &n);

printf("\nEnter Arrival Time and Burst Time:\n");

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

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

scanf("%d", &Arrival\_time[i]);

printf("P[%d] Burst Time: ", 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!!!\n");

}

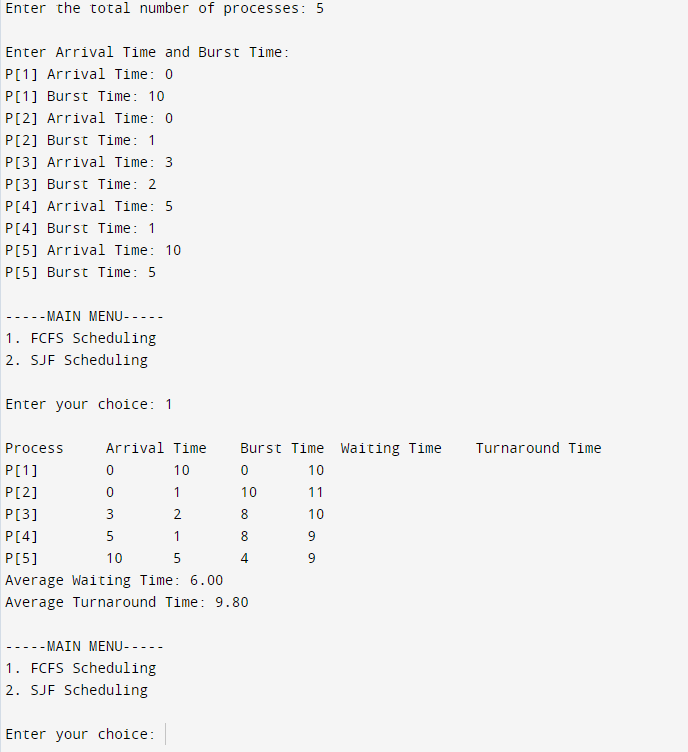
}

return 0;

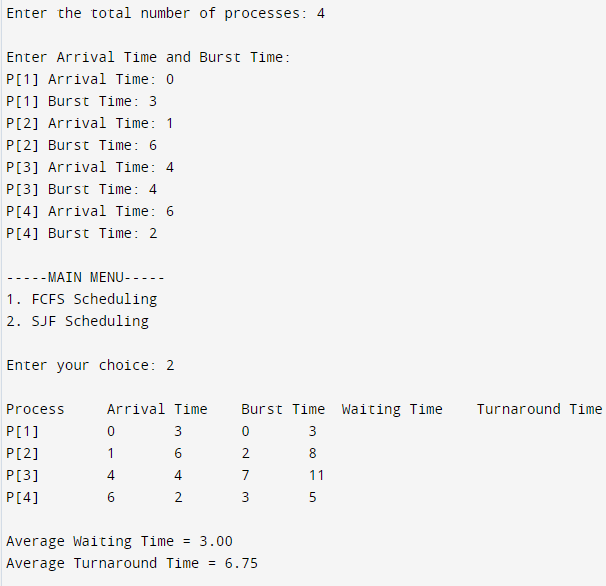
}

**OUTPUTS:**

**a.**



**b.**



**LAB - 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];

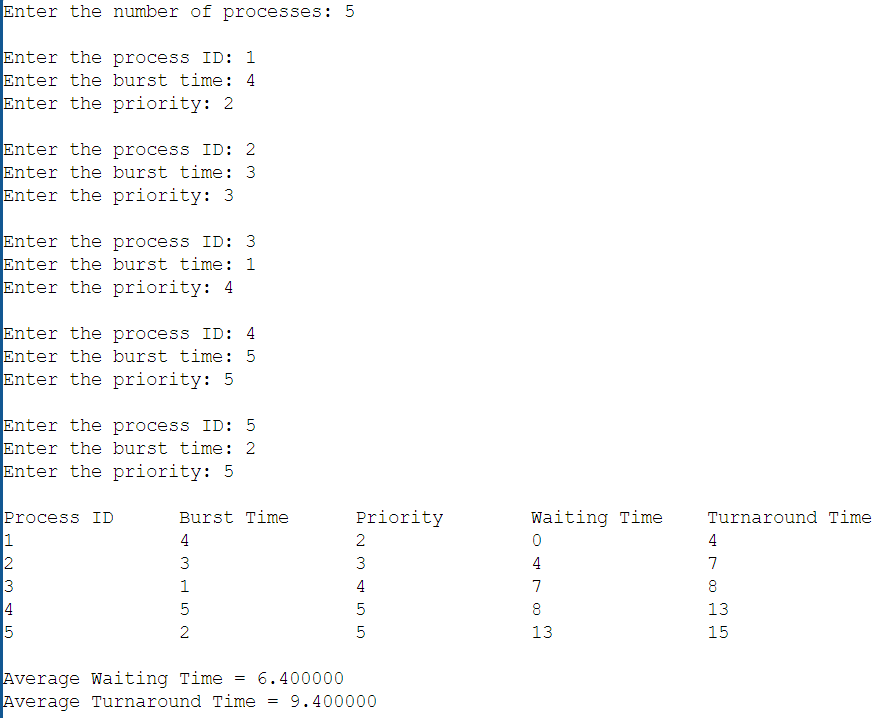
proc[pos] = temp;

}

find\_average\_time(proc, n);

}

**OUTPUT:**



**Priority (Pre-emptive):**

**CODE:**

#include<stdio.h>

#include<stdlib.h>

struct process {

int process\_id;

int burst\_time;

int priority;

int arrival\_time;

int remaining\_time;

int waiting\_time;

int turnaround\_time;

int is\_completed;

};

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 arrival time: ");

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

printf("Enter the priority: ");

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

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

proc[i].is\_completed = 0;

}

priority\_scheduling(proc, n);

return 0;

}

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

int time = 0, completed = 0, min\_priority, shortest = 0;

while (completed != n) {

min\_priority = 10000;

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

if ((proc[i].arrival\_time <= time) && (!proc[i].is\_completed) && (proc[i].priority < min\_priority)) {

min\_priority = proc[i].priority;

shortest = i;

}

}

proc[shortest].remaining\_time--;

time++;

if (proc[shortest].remaining\_time == 0) {

proc[shortest].waiting\_time = time - proc[shortest].arrival\_time - proc[shortest].burst\_time;

proc[shortest].turnaround\_time = time - proc[shortest].arrival\_time;

proc[shortest].is\_completed = 1;

completed++;

}

}

}

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

// Turnaround time is calculated during the find\_waiting\_time function

}

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

int total\_wt = 0, total\_tat = 0;

find\_waiting\_time(proc, n);

find\_turnaround\_time(proc, n);

printf("\nProcess ID\tBurst Time\tArrival Time\tPriority\tWaiting Time\tTurnaround Time");

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

total\_wt += proc[i].waiting\_time;

total\_tat += proc[i].turnaround\_time;

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

}

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

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

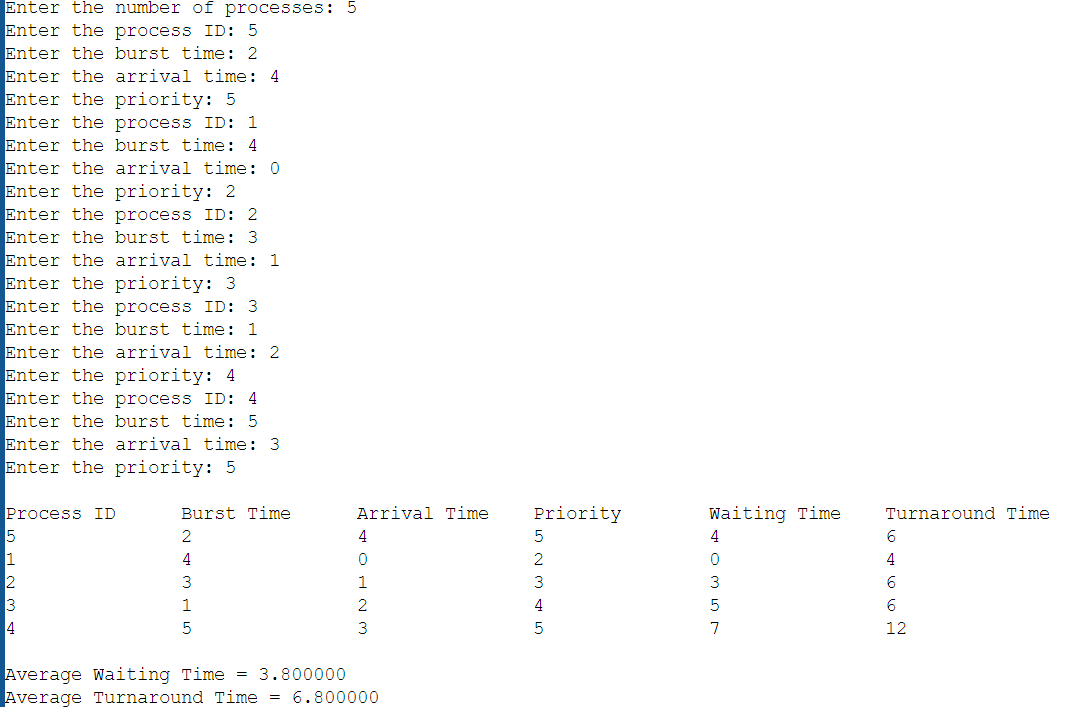
}

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

find\_average\_time(proc, n);

}

**OUTPUT:**

****

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

#include <stdio.h>

#include <stdbool.h>

void findTurnaroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {

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

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

}

}

void findWaitingTime(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 += rem\_bt[i];

wt[i] = t - bt[i];

rem\_bt[i] = 0;

}

}

}

if (done == true)

break;

}

}

void findAvgTime(int processes[], int n, int bt[], int quantum) {

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

findWaitingTime(processes, n, bt, wt, quantum);

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

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

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

total\_wt += wt[i];

total\_tat += tat[i];

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

}

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

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

}

int main() {

int n, quantum;

printf("Enter the Number of Processes: ");

scanf("%d", &n);

int processes[n], burst\_time[n];

printf("\nEnter the quantum time: ");

scanf("%d", &quantum);

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

printf("\nEnter the process ID: ");

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

printf("Enter the Burst Time: ");

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

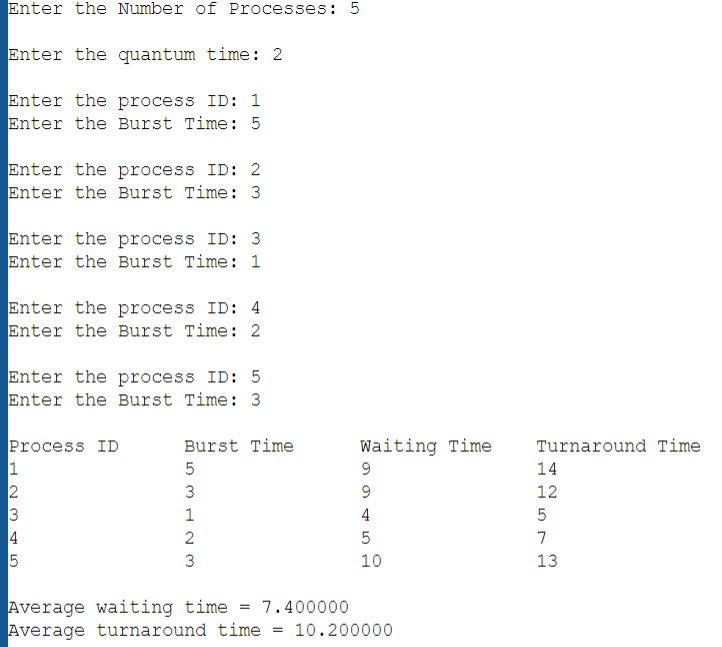
}

findAvgTime(processes, n, burst\_time, quantum);

return 0;

}

**OUTPUT:**



**LAB - 3**

**Question 1:**

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

1. **Rate- Monotonic**
2. **Earliest-deadline First**

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);

}

int main()

{

int option;

int observation\_time;

while (1)

{

printf("\n1. Rate Monotonic\n2. Earliest Deadline first\\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: exit (0);

default: printf("\nInvalid Statement");

}

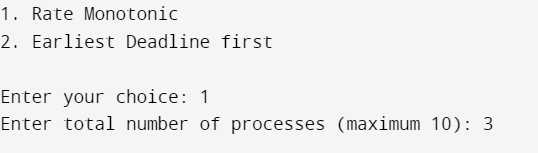
}

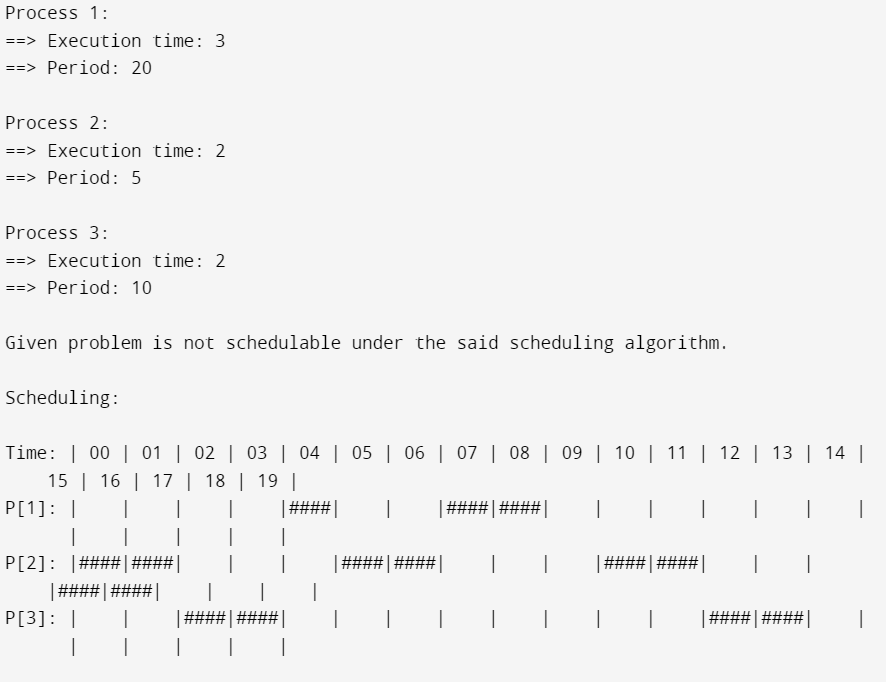
return 0;

}

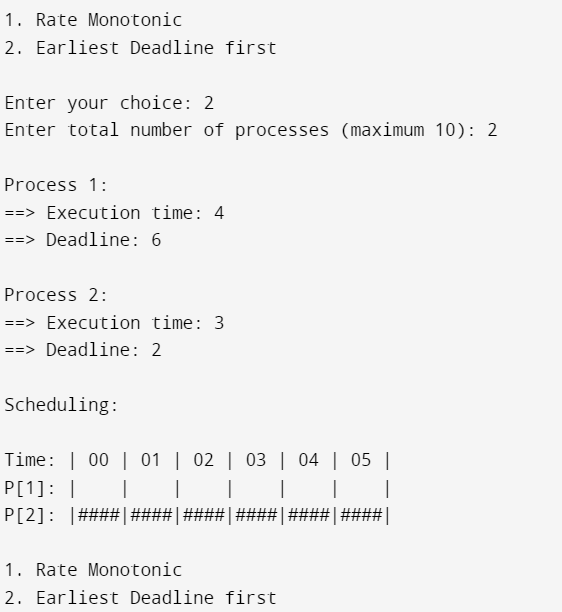
**Output:**

1. **Rate Monotonic:**

****

****

1. **Earliest Deadline First:**

****

**LAB - 4**

**Question 1:**

**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\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);

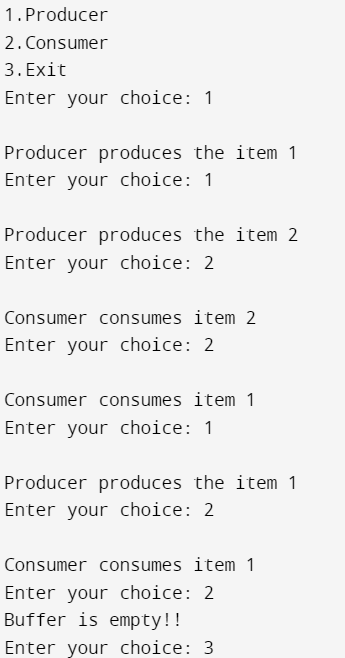
printf("\nConsumer consumes item %d",x);

x--;

mutex=signal(mutex);

}

**OUTPUT:**

****

**Question 2:**

**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 (i + 4) % N

#define RIGHT (i + 1) % N

int state[N];

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

sem\_t mutex;

sem\_t S[N];

void test(int i)

{

if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)

{

state[i] = EATING;

sleep(2);

printf("Philosopher %d takes fork %d and %d\n", i +1, LEFT +1, i +1);

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

sem\_post(&S[i]);

}

}

void take\_fork(int i)

{

sem\_wait(&mutex);

state[i] = HUNGRY;

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

test(i);

sem\_post(&mutex);

sem\_wait(&S[i]);

sleep(1);

}

void put\_fork(int i)

{

sem\_wait(&mutex);

state[i] = THINKING;

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

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

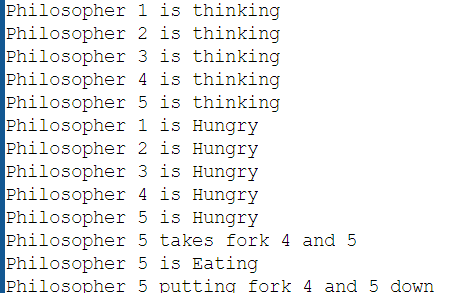
{

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

}

}

**OUTPUT:**



**LAB 5**

**Question 1:**

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

**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]);

}

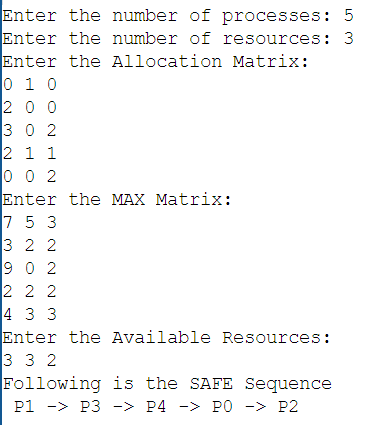
printf(" P%d\n", ans[n - 1]);

}

return 0;

}

**OUTPUT:**



**Question 2:**

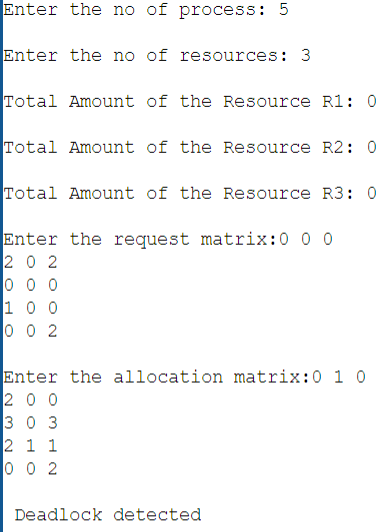
**Write a C program to simulate deadlock detection.**

**CODE**:

#include<stdio.h>  
static int mark[20];  
int i,j,np,nr;  
  
int main()  
{  
int alloc[10][10],request[10][10],avail[10],r[10],w[10];  
  
printf("\nEnter the no of process: ");  
scanf("%d",&np);  
printf("\nEnter the no of resources: ");  
scanf("%d",&nr);  
for(i=0;i<nr;i++)  
{  
printf("\nTotal Amount of the Resource R%d: ",i+1);  
scanf("%d",&r[i]);  
}  
  
printf("\nEnter the request matrix:");  
  
for(i=0;i<np;i++)  
for(j=0;j<nr;j++)  
scanf("%d",&request[i][j]);  
  
printf("\nEnter the allocation matrix:");  
for(i=0;i<np;i++)  
for(j=0;j<nr;j++)  
scanf("%d",&alloc[i][j]);  
  
for(j=0;j<nr;j++)  
{  
avail[j]=r[j];  
for(i=0;i<np;i++)  
{  
avail[j]-=alloc[i][j];  
  
}  
}  
  
  
  
for(i=0;i<np;i++)  
{  
int count=0;  
 for(j=0;j<nr;j++)  
   {  
      if(alloc[i][j]==0)  
        count++;  
      else  
        break;  
    }  
 if(count==nr)  
 mark[i]=1;  
}  
  
  
for(j=0;j<nr;j++)  
    w[j]=avail[j];

for(i=0;i<np;i++)  
{  
int canbeprocessed=0;  
 if(mark[i]!=1)  
{  
   for(j=0;j<nr;j++)  
    {  
      if(request[i][j]<=w[j])  
        canbeprocessed=1;  
      else  
         {  
         canbeprocessed=0;  
         break;  
          }  
     }  
if(canbeprocessed)  
{  
mark[i]=1;  
  
for(j=0;j<nr;j++)  
w[j]+=alloc[i][j];  
}  
}  
}  
  
  
int deadlock=0;  
for(i=0;i<np;i++)  
if(mark[i]!=1)  
deadlock=1;  
  
  
if(deadlock)  
printf("\n Deadlock detected");  
else  
printf("\n No Deadlock possible");  
}

**OUTPUT:**

****

**LAB 6**

**Question 1:**

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

1. **Worst-fit**
2. **Best-fit**

**(c) 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++)

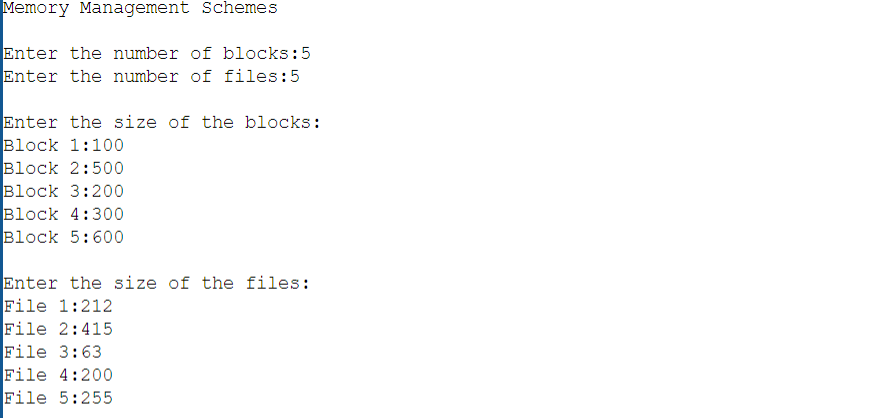
{

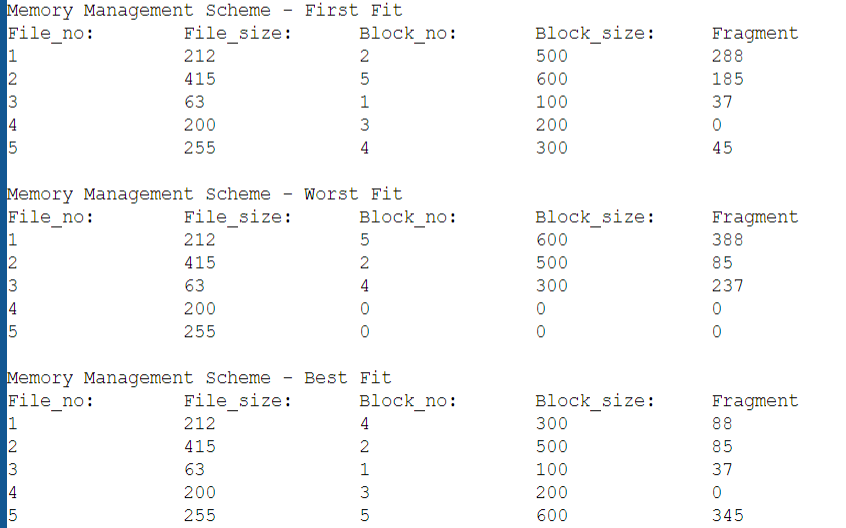
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);

}

}

**OUTPUT:**

****

****

**Question 2:**

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

1. **FIFO**
2. **LRU**
3. **Optimal**

**CODE**:

#include<stdio.h>

int n, f, 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",&f);

}

void initialize()

{

pgfaultcnt=0;

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

p[i]=9999;

}

int isHit(int data)

{

hit=0;

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

{

if(p[j]==data)

{

hit=1;

break;

}

}

return hit;

}

int getHitIndex(int data)

{

int hitind;

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

{

if(p[k]==data)

{

hitind=k;

break;

}

}

return hitind;

}

void dispPages()

{

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

{

if(p[k]!=9999)

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

}

}

void dispPgFaultCnt()

{

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

}

void fifo()

{

getdata();

initialize();

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

{

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

//not a hit

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

{

for(k=0; k<f-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<f; 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;

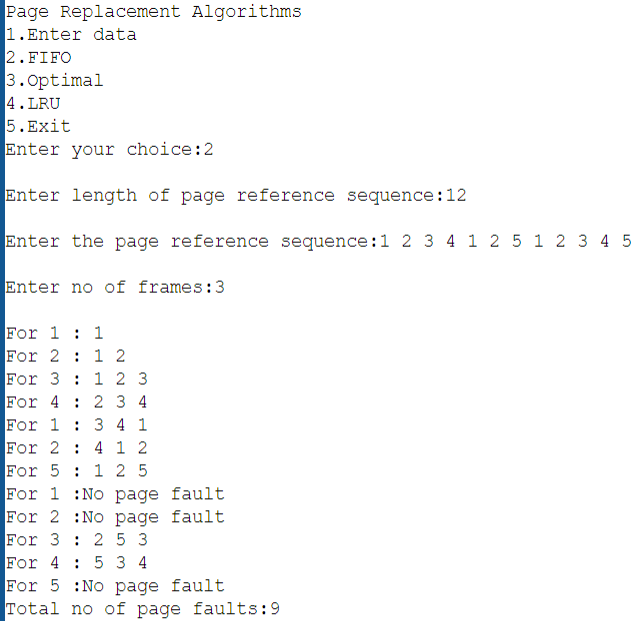
break;

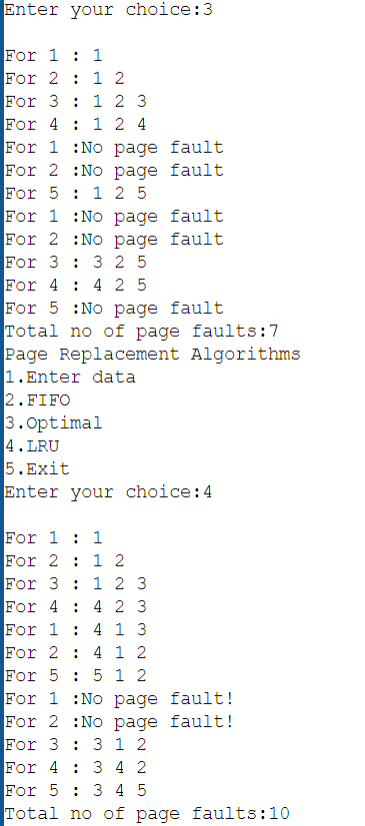
}

}

}

**OUTPUT:**





**LAB 7**

**Question 1:**

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

**(a)FCFS**

**(b)SCAN**

**(c)C-SCAN**

1. FCFS:

**CODE**:#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);

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

{

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

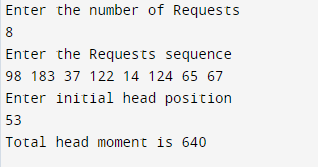
initial=RQ[i];

}

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

return 0;

}

**OUTPUT**:  


1. SCAN: **CODE**:

#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);

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(move==1)

{

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

{

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

initial=RQ[i];

}

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];

}

}

else

{

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

{

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

initial=RQ[i];

}

TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);

initial =0;

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

{

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

initial=RQ[i];

}

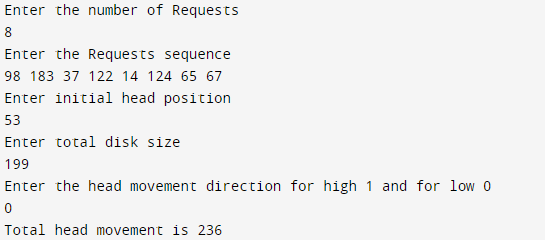
}

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

return 0;

}

**OUTPUT:**

****

1. C-SCAN:CODE:

#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);

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(move==1)

{

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

{

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

initial=RQ[i];

}

TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);

TotalHeadMoment=TotalHeadMoment+abs(size-1-0);

initial=0;

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

{

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

initial=RQ[i];

}

}

else

{

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

{

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

initial=RQ[i];

}

TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);

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:**

