**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

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

****

**LAB REPORT**

**on**

**OPERATING SYSTEMS**

***Submitted by***

**Aditya Kumar (1BM22CS018)**

***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**

**Apr-2024 to Aug-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 – 23CS4PCOPS” carried out by **Aditya Kumar (1BM22CS018),** 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 2024. The Lab report has been approved as it satisfies the academic requirements in respect of a **OPERATING SYSTEMS - (23CS4PCOPS)** work prescribed for the said degree.

Name of the Lab-Incharge: Sneha S Bagalkot               **Dr. Jyothi S Nayak**

Designation: Assistant Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

**Index Sheet**

|  |  |  |
| --- | --- | --- |
| **Sl. No.** | **Experiment Title** | **Page No.** |
| 1. | Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.  →FCFS  → SJF (pre-emptive & non-preemptive) | 1-7 |
| 2. | Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.  → Priority (pre-emptive & non-pre-emptive)  →Round Robin | 8-14 |
| 3. | Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are  to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue. | 15-17 |
| 4. | Write a C program to simulate Real-Time CPU Scheduling algorithms:  a) Rate- Monotonic  b) Earliest-deadline First  c) Proportional scheduling. | 18-26 |
| 5. | Write a C program to simulate producer-consumer problem using semaphores.. | 27-39 |
| 6. | Write a C program to simulate the concept of Dining-Philosophers problem.. | 30-32 |
| 7. | Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance | 33-36 |
| 8. | Write a C program to simulate deadlock detection | 37-40 |
| 9. | Write a C program to simulate the following contiguous memory allocation techniques:  a) Worst-fit  b) Best-fit  c) First-fit | 41-47 |
| 10. | Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) Optimal. | 47-51 |
| 11. | Write a C program to simulate disk scheduling algorithms  a) FCFS  b) SCAN  c)c-SCAN | 52-58 |

**Program - 1**

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

algorithm to find turnaround time and waiting time.

→FCFS

→ SJF (pre-emptive & Non-preemptive)

**FCFS Code:**

#include <stdio.h>

#include <stdlib.h>

int main(){

int n;

int process\_id[n],at[n],bt[n],ct[n],tat[n],wt[n];

printf("\nEnter number of processes: ");

scanf("%d",&n);

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

process\_id[i] = i+1;

printf("\nArrival Time for %d: ",(i+1));

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

printf("\nBurst Time for %d: ",(i+1));

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

}

int temp = 0;

int temp2 = 0;

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

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

if(at[i] > at[j]){

temp = at[i];

at[i] = at[j];

at[j] = temp;

temp2 = bt[i];

bt[i] = bt[j];

bt[j] = temp2;

}

}

}

int timePassed = 0;

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

if(at[i] > timePassed){

timePassed = timePassed + (at[i] - ct[i-1]);

}

timePassed += bt[i];

ct[i] = timePassed;

}

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

tat[i] = ct[i] = at[i];

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

}

printf("\nPID\tAT\tBT\tCT\tTAT\tWT");

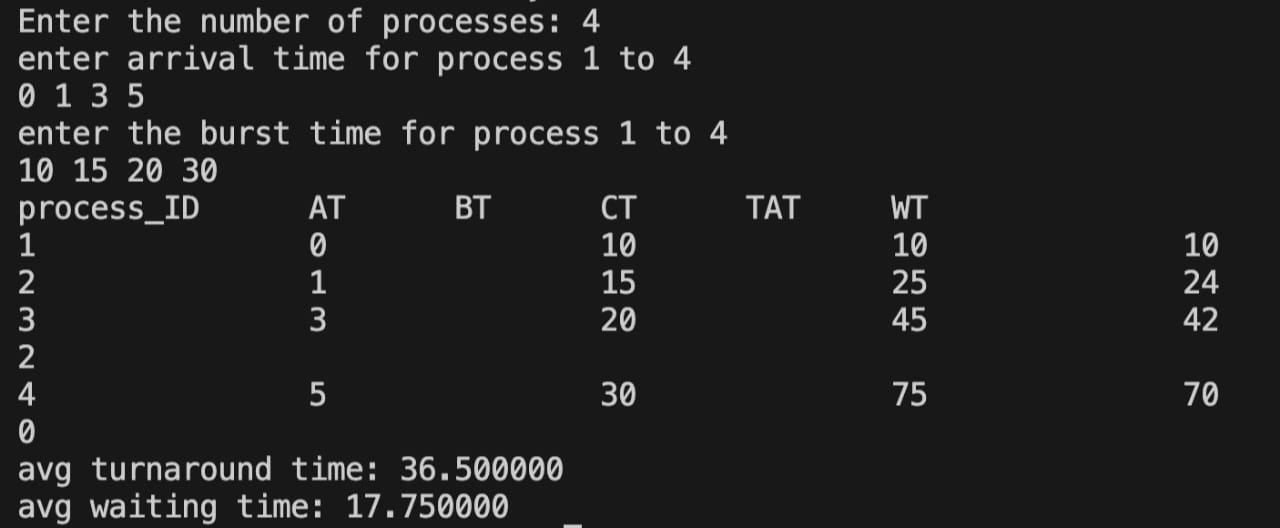
for(int i = 0; i<n;i++){

printf("\n%d\t%d\t%d\t%d\t%d\t%d",process\_id[i],at[i],bt[i],ct[i],tat[i],wt[i]);

}

}

**Result:**

****

**SJF Non Pre-emptive Code:**

#include <stdio.h>

#include <stdlib.h>

typedef struct {

char process\_name;

int arrival\_time;

int burst\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

} Process;

void sort\_by\_burst\_time(Process \*processes, int n) {

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

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

if (processes[j].burst\_time > processes[j + 1].burst\_time) {

Process temp = processes[j];

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

processes[j + 1] = temp;

}

}

}

}

void compute\_completion\_time(Process \*processes, int n) {

int current\_time = 0;

int index = 0;

while (index < n) {

int next\_process = -1;

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

if (processes[i].arrival\_time <= current\_time && processes[i].completion\_time == 0) {

if (next\_process == -1 || processes[i].burst\_time < processes[next\_process].burst\_time) {

next\_process = i;

}

}

}

if (next\_process == -1) {

current\_time = processes[index].arrival\_time;

} else {

processes[next\_process].completion\_time = current\_time + processes[next\_process].burst\_time;

current\_time = processes[next\_process].completion\_time;

}

index++;

}

}

void compute\_turnaround\_waiting\_time(Process \*processes, int n) {

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

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

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

}

}

void display\_table(Process \*processes, int n) {

printf("Process Arrival Time Burst Time Completion Time Turnaround Time Waiting Time\n");

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

printf(" %c\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].process\_name,

processes[i].arrival\_time,

processes[i].burst\_time,

processes[i].completion\_time,

processes[i].turnaround\_time,

processes[i].waiting\_time);

}

}

int main() {

int n;

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

scanf("%d", &n);

Process \*processes = (Process \*)malloc(n \* sizeof(Process));

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

printf("Enter details for process %d (Name Arrival Burst): ", i + 1);

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

processes[i].completion\_time = 0;

processes[i].turnaround\_time = 0;

processes[i].waiting\_time = 0;

}

sort\_by\_burst\_time(processes, n);

compute\_completion\_time(processes, n);

compute\_turnaround\_waiting\_time(processes, n);

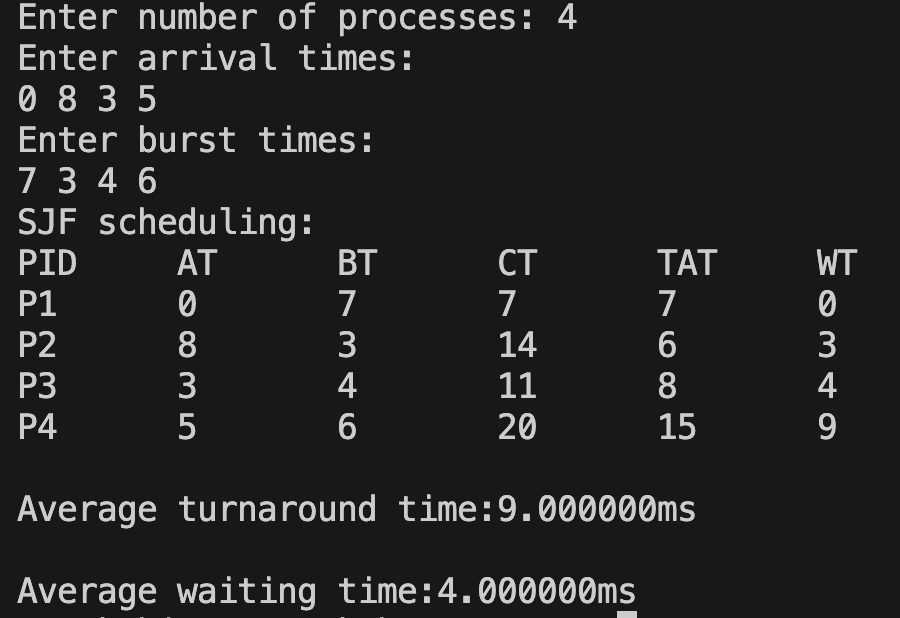
display\_table(processes, n);

free(processes);

return 0;

}

**Result:**



**SJF Pre-emptive Code:**

#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

struct Process {

int pid;

int arrival\_time;

int burst\_time;

int remaining\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

};

int findShortestJob(struct Process processes[], int n, int current\_time) {

int shortest\_job\_index = -1;

int shortest\_job = INT\_MAX;

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

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

shortest\_job\_index = i;

shortest\_job = processes[i].remaining\_time;

}

}

return shortest\_job\_index;

}

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

int current\_time = 0;

int completed = 0;

while (completed < n) {

int shortest\_job\_index = findShortestJob(processes, n, current\_time);

if (shortest\_job\_index == -1) {

current\_time++;

} else {

processes[shortest\_job\_index].remaining\_time--;

current\_time++;

if (processes[shortest\_job\_index].remaining\_time == 0) {

processes[shortest\_job\_index].completion\_time = current\_time;

processes[shortest\_job\_index].turnaround\_time = processes[shortest\_job\_index].completion\_time - processes[shortest\_job\_index].arrival\_time;

processes[shortest\_job\_index].waiting\_time = processes[shortest\_job\_index].turnaround\_time - processes[shortest\_job\_index].burst\_time;

completed++;

}

}

}

}

int main() {

int n;

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

scanf("%d", &n);

struct Process processes[n];

printf("Enter Arrival Time and Burst Time for each process:\n");

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

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

printf("Arrival Time: ");

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

printf("Burst Time: ");

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

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

processes[i].pid = i + 1;

}

SJF(processes, n);

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

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

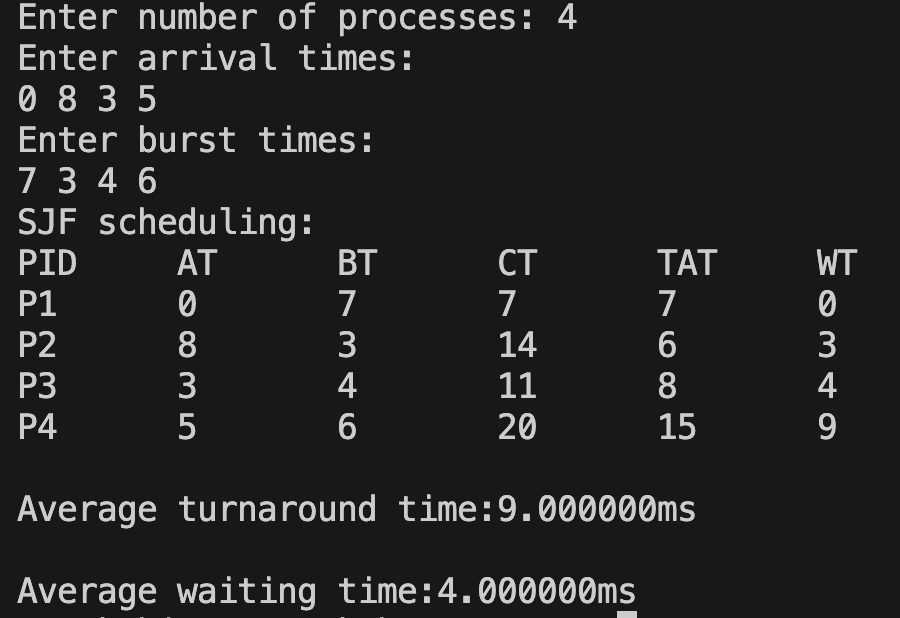
printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrival\_time, processes[i].burst\_time, processes[i].completion\_time, processes[i].waiting\_time, processes[i].turnaround\_time);

}

return 0;

}

**Result:**

****

**Program - 2**

Write a C program to simulate the following CPU scheduling algorithm to find

turnaround time and waiting time.

→ Priority

→Round Robin

**Priority Scheduling Code:**

#include<stdio.h>

void sort (int proc\_id[], int p[], int at[], int bt[], int n){

int min = p[0], temp = 0;

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

{

min = p[i];

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

{

if (p[j] < min)

{

temp = at[i];

at[i] = at[j];

at[j] = temp;

temp = bt[j];

bt[j] = bt[i];

bt[i] = temp;

temp = p[j];

p[j] = p[i];

p[i] = temp;

temp = proc\_id[i];

proc\_id[i] = proc\_id[j];

proc\_id[j] = temp;

}

}

}

}

int main (){

int n, c = 0;

printf ("Enter number of processes: ");

scanf ("%d", &n);

int proc\_id[n], at[n], bt[n], ct[n], tat[n], wt[n], m[n], rt[n], p[n];

double avg\_tat = 0.0, ttat = 0.0, avg\_wt = 0.0, twt = 0.0;

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

{

proc\_id[i] = i + 1;

m[i] = 0;

}

printf ("Enter priorities:\n");

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

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

printf ("Enter arrival times:\n");

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

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

printf ("Enter burst times:\n");

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

{

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

m[i] = -1;

rt[i] = -1;

}

sort (proc\_id, p, at, bt, n);

int count = 0, pro = 0, priority = p[0];

int x = 0;

c = 0;

while (count < n)

{

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

{

if (at[i] <= c && p[i] >= priority && m[i] != 1)

{

x = i;

priority = p[i];

}

}

if (rt[x] == -1)

rt[x] = c - at[x];

if (at[x] <= c)

c += bt[x];

else

c += at[x] - c + bt[x];

count++;

ct[x] = c;

m[x] = 1;

while (x >= 1 && m[--x] != 1)

{

priority = p[x];

break;

}

x++;

if (count == n)

break;

}

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

tat[i] = ct[i] - at[i];

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

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

printf ("\nPriority scheduling:\n");

printf ("PID\tPrior\tAT\tBT\tCT\tTAT\tWT\tRT\n");

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

printf ("P%d\t %d\t\t%d\t%d\t%d\t%d\t%d\t%d\n", proc\_id[i], p[i], at[i],

bt[i], ct[i], tat[i], wt[i], rt[i]);

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

{

ttat += tat[i];

twt += wt[i];

}

avg\_tat = ttat / (double) n;

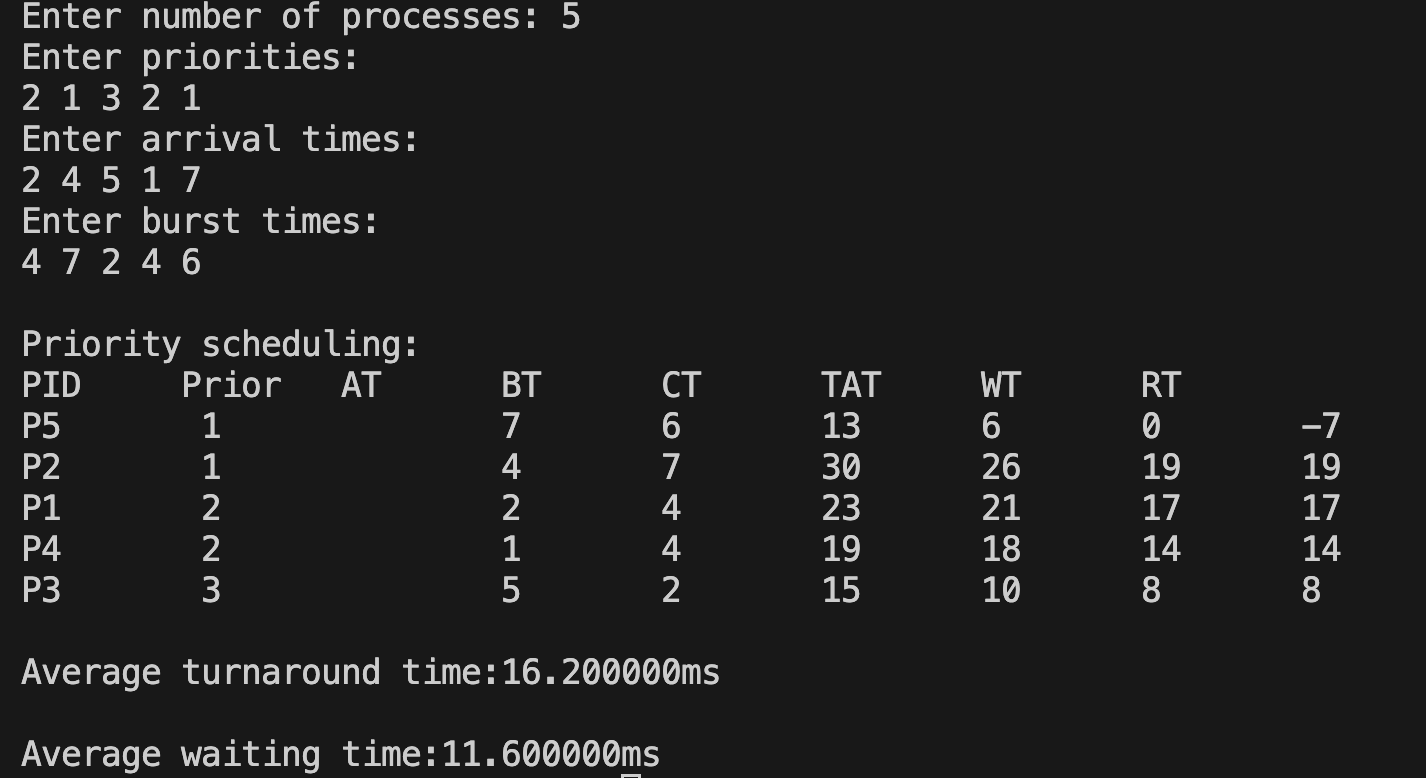
avg\_wt = twt / (double) n;

printf ("\nAverage turnaround time:%lfms\n", avg\_tat);

printf ("\nAverage waiting time:%lfms\n", avg\_wt);

}

**Result:**

****

**Round Robin Code:**

#include<stdio.h>

void sort (int proc\_id[], int at[], int bt[], int b[], int n){

int min = at[0], temp = 0;

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

min = at[i];

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

{

if (at[j] < min)

{

temp = at[i];

at[i] = at[j];

at[j] = temp;

temp = bt[j];

bt[j] = bt[i];

bt[i] = temp;

temp = b[j];

b[j] = b[i];

b[i] = temp;

temp = proc\_id[i];

proc\_id[i] = proc\_id[j];

proc\_id[j] = temp;

}

}

}

}

int main (){

int n, c = 0, t = 0;

printf ("Enter number of processes: ");

scanf ("%d", &n);

printf ("Enter Time Quantum: ");

scanf ("%d", &t);

int proc\_id[n], at[n], bt[n], ct[n], tat[n], wt[n], b[n], rt[n], m[n];

int f = -1, r = -1;

int q[100];

int count = 0;

double avg\_tat = 0.0, ttat = 0.0, avg\_wt = 0.0, twt = 0.0;

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

proc\_id[i] = i + 1;

printf ("Enter arrival times:\n");

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

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

printf ("Enter burst times:\n");

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

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

b[i] = bt[i];

m[i] = 0;

rt[i] = -1;

}

sort (proc\_id, at, bt, b, n);

f = r = 0;

q[0] = proc\_id[0];

int p = 0, i = 0;

while (f >= 0){

p = q[f++];

i = 0;

while (p != proc\_id[i])

i++;

if (b[i] >= t)

{

if (rt[i] == -1)

rt[i] = c;

b[i] -= t;

c += t;

m[i] = 1;

}

else

{

if (rt[i] == -1)

rt[i] = c;

c += b[i];

b[i] = 0;

m[i] = 1;

}

m[0] = 1;

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

if (at[j] <= c && proc\_id[j] != p && m[j] != 1)

{

q[++r] = proc\_id[j];

m[j] = 1;

}

}

if (b[i] == 0){

count++;

ct[i] = c;

}

else

q[++r] = proc\_id[i];

if (f > r)

f = -1;

}

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

tat[i] = ct[i] - at[i];

rt[i] = rt[i] - at[i];

}

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

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

printf ("\nRRS scheduling:\n");

printf ("PID\tAT\tBT\tCT\tTAT\tWT\tRT\n");

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

printf ("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", proc\_id[i], at[i], bt[i], ct[i],

tat[i], wt[i], rt[i]);

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

{

ttat += tat[i];

twt += wt[i];

}

avg\_tat = ttat / (double) n;

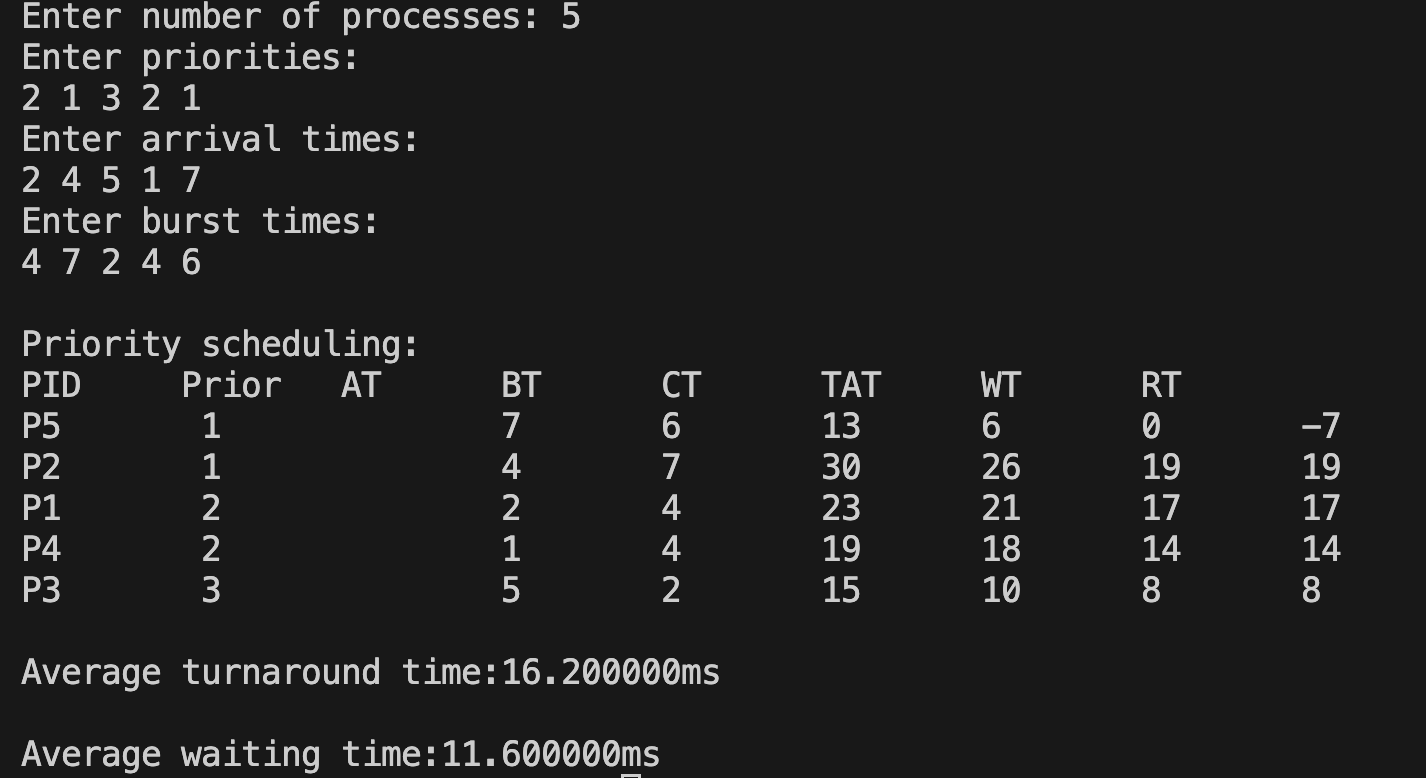
avg\_wt = twt / (double) n;

printf ("\nAverage turnaround time:%lfms\n", avg\_tat);

printf ("\nAverage waiting time:%lfms\n", avg\_wt);

}

**Result:**

****

**Program - 3**

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

**Code:**

#include <stdio.h>

void sort(int proc\_id[], int at[], int bt[], int n) {

int min, temp;

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

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

if(at[j] < at[i]) {

temp = at[i];

at[i] = at[j];

at[j] = temp;

temp = bt[i];

bt[i] = bt[j];

bt[j] = temp;

temp = proc\_id[i];

proc\_id[i] = proc\_id[j];

proc\_id[j] = temp;

}

}

}

}

void simulateFCFS(int proc\_id[], int at[], int bt[], int n, int start\_time) {

int c = start\_time, ct[n], tat[n], wt[n];

double ttat = 0.0, twt = 0.0;

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

if(c >= at[i])

c += bt[i];

else

c = at[i] + bt[i];

ct[i] = c;

}

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

tat[i] = ct[i] - at[i];

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

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

printf("PID\tAT\tBT\tCT\tTAT\tWT\n");

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

printf("%d\t%d\t%d\t%d\t%d\t%d\n", proc\_id[i], at[i], bt[i], ct[i], tat[i], wt[i]);

ttat += tat[i];

twt += wt[i];

}

printf("Average Turnaround Time: %.2lf ms\n", ttat/n);

printf("Average Waiting Time: %.2lf ms\n", twt/n);

}

void main() {

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

int proc\_id[n], at[n], bt[n], type[n];

int sys\_proc\_id[n], sys\_at[n], sys\_bt[n], user\_proc\_id[n], user\_at[n], user\_bt[n];

int sys\_count = 0, user\_count = 0;

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

proc\_id[i] = i + 1;

printf("Enter arrival time, burst time and type (0 for system, 1 for user) for process %d: ", i+1);

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

if(type[i] == 0) {

sys\_proc\_id[sys\_count] = proc\_id[i];

sys\_at[sys\_count] = at[i];

sys\_bt[sys\_count] = bt[i];

sys\_count++;

} else {

user\_proc\_id[user\_count] = proc\_id[i];

user\_at[user\_count] = at[i];

user\_bt[user\_count] = bt[i];

user\_count++;

}

}

sort(sys\_proc\_id, sys\_at, sys\_bt, sys\_count);

sort(user\_proc\_id, user\_at, user\_bt, user\_count); //arrival time sort

printf("System Processes Scheduling:\n");

simulateFCFS(sys\_proc\_id, sys\_at, sys\_bt, sys\_count, 0);

int system\_end\_time = 0;

if (sys\_count > 0) {

system\_end\_time = sys\_at[sys\_count - 1] + sys\_bt[sys\_count - 1];

for (int i = 0; i < sys\_count - 1; i++) {

if (sys\_at[i + 1] > system\_end\_time) {

system\_end\_time = sys\_at[i + 1];

}

system\_end\_time += sys\_bt[i];

}

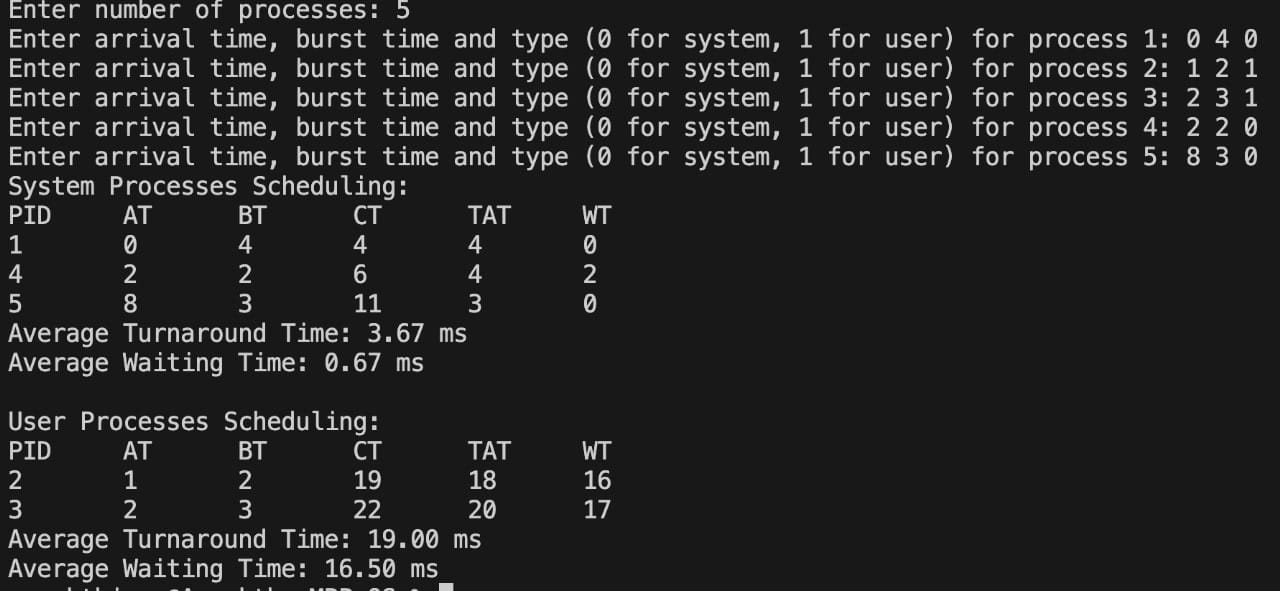
}

printf("\nUser Processes Scheduling:\n");

simulateFCFS(user\_proc\_id, user\_at, user\_bt, user\_count, system\_end\_time);

}

**Result:**

****

**Program - 4**

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

a) Rate- Monotonic

b) Earliest-deadline First

c) Proportional scheduling

**a) Rate-Monotonic Code:**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

void sort (int proc[], int b[], int pt[], int n){

int temp = 0;

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

{

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

{

if (pt[j] < pt[i])

{

temp = pt[i];

pt[i] = pt[j];

pt[j] = temp;

temp = b[j];

b[j] = b[i];

b[i] = temp;

temp = proc[i];

proc[i] = proc[j];

proc[j] = temp;

}

}

}

}

int gcd (int a, int b){

int r;

while (b > 0)

{

r = a % b;

a = b;

b = r;

}

return a;

}

int lcmul (int p[], int n){

int lcm = p[0];

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

lcm = (lcm \* p[i]) / gcd (lcm, p[i]);

}

return lcm;

}

int main(){

int n;

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

scanf ("%d", &n);

int proc[n], b[n], pt[n], rem[n];

printf ("Enter the CPU burst times:\n");

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

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

rem[i] = b[i];

}

printf ("Enter the time periods:\n");

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

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

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

proc[i] = i + 1;

sort (proc, b, pt, n);

int l = lcmul (pt, n);

printf ("LCM=%d\n", l);

printf ("\nRate Monotone Scheduling:\n");

printf ("PID\t Burst\tPeriod\n");

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

printf ("%d\t\t%d\t\t%d\n", proc[i], b[i], pt[i]);

double sum = 0.0;

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

sum += (double) b[i] / pt[i];

}

double rhs = n \* (pow (2.0, (1.0 / n)) - 1.0);

printf ("\n%lf <= %lf =>%s\n", sum, rhs, (sum <= rhs) ? "true" : "false");

if (sum > rhs)

exit (0);

printf ("Scheduling occurs for %d ms\n\n", l);

int time = 0, prev = 0, x = 0;

while (time < l){

int f = 0;

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

{

if (time % pt[i] == 0)

rem[i] = b[i];

if (rem[i] > 0)

{

if (prev != proc[i])

{

printf ("%dms onwards: Process %d running\n", time,

proc[i]);

prev = proc[i];

}

rem[i]--;

f = 1;

break;

x = 0;

}

}

if (!f)

{

if (x != 1)

{

printf ("%dms onwards: CPU is idle\n", time);

x = 1;

}

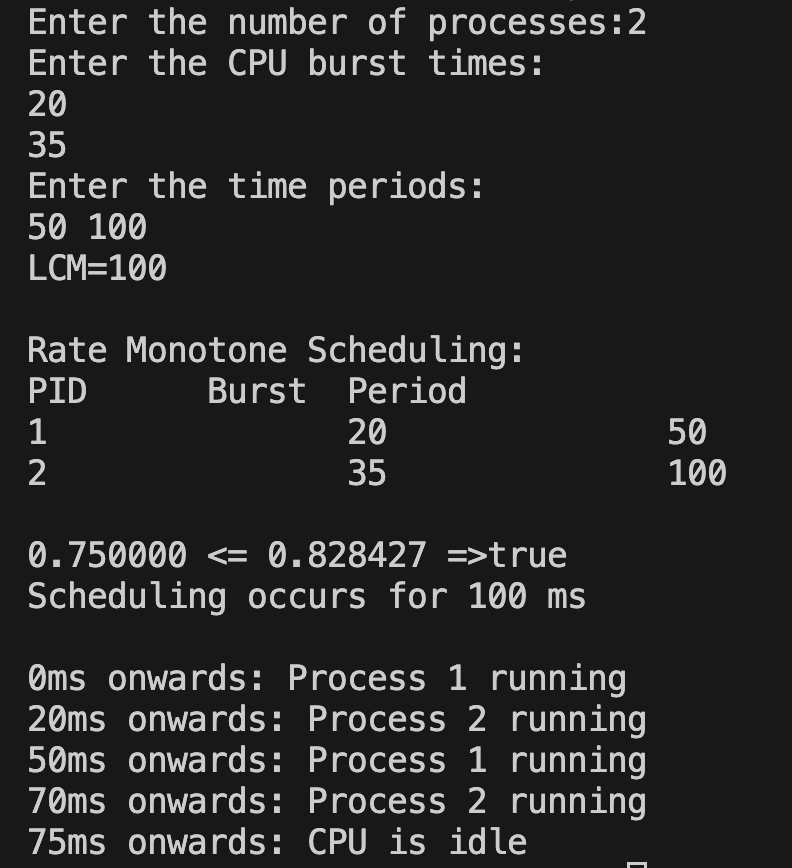
}

time++;

}

}

**Result:**

****

**b) Earliest Deadline First Code:**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

void sort (int proc[], int d[], int b[], int pt[], int n){

int temp = 0;

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

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

if (d[j] < d[i]){

temp = d[j];

d[j] = d[i];

d[i] = temp;

temp = pt[i];

pt[i] = pt[j];

pt[j] = temp;

temp = b[j];

b[j] = b[i];

b[i] = temp;

temp = proc[i];

proc[i] = proc[j];

proc[j] = temp;

}

}

}

}

int gcd (int a, int b){

int r;

while (b > 0)

{

r = a % b;

a = b;

b = r;

}

return a;

}

int lcmul (int p[], int n){

int lcm = p[0];

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

{

lcm = (lcm \* p[i]) / gcd (lcm, p[i]);

}

return lcm;

}

int main (){

int n;

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

scanf ("%d", &n);

int proc[n], b[n], pt[n], d[n], rem[n];

printf ("Enter the CPU burst times:\n");

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

{

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

rem[i] = b[i];

}

printf ("Enter the deadlines:\n");

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

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

printf ("Enter the time periods:\n");

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

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

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

proc[i] = i + 1;

sort (proc, d, b, pt, n);

int l = lcmul (pt, n);

printf ("\nEarliest Deadline Scheduling:\n");

printf ("PID\t Burst\tDeadline\tPeriod\n");

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

printf ("%d\t\t%d\t\t%d\t\t%d\n", proc[i], b[i], d[i], pt[i]);

printf ("Scheduling occurs for %d ms\n\n", l);

int time = 0, prev = 0, x = 0;

int nextDeadlines[n];

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

{

nextDeadlines[i] = d[i];

rem[i] = b[i];

}

while (time < l)

{

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

{

if (time % pt[i] == 0 && time != 0)

{

nextDeadlines[i] = time + d[i];

rem[i] = b[i];

}

}

int minDeadline = l + 1;

int taskToExecute = -1;

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

if (rem[i] > 0 && nextDeadlines[i] < minDeadline){

minDeadline = nextDeadlines[i];

taskToExecute = i;

}

}

if (taskToExecute != -1){

printf ("%dms : Task %d is running.\n", time, proc[taskToExecute]);

rem[taskToExecute]--;

}

else{

printf ("%dms: CPU is idle.\n", time);

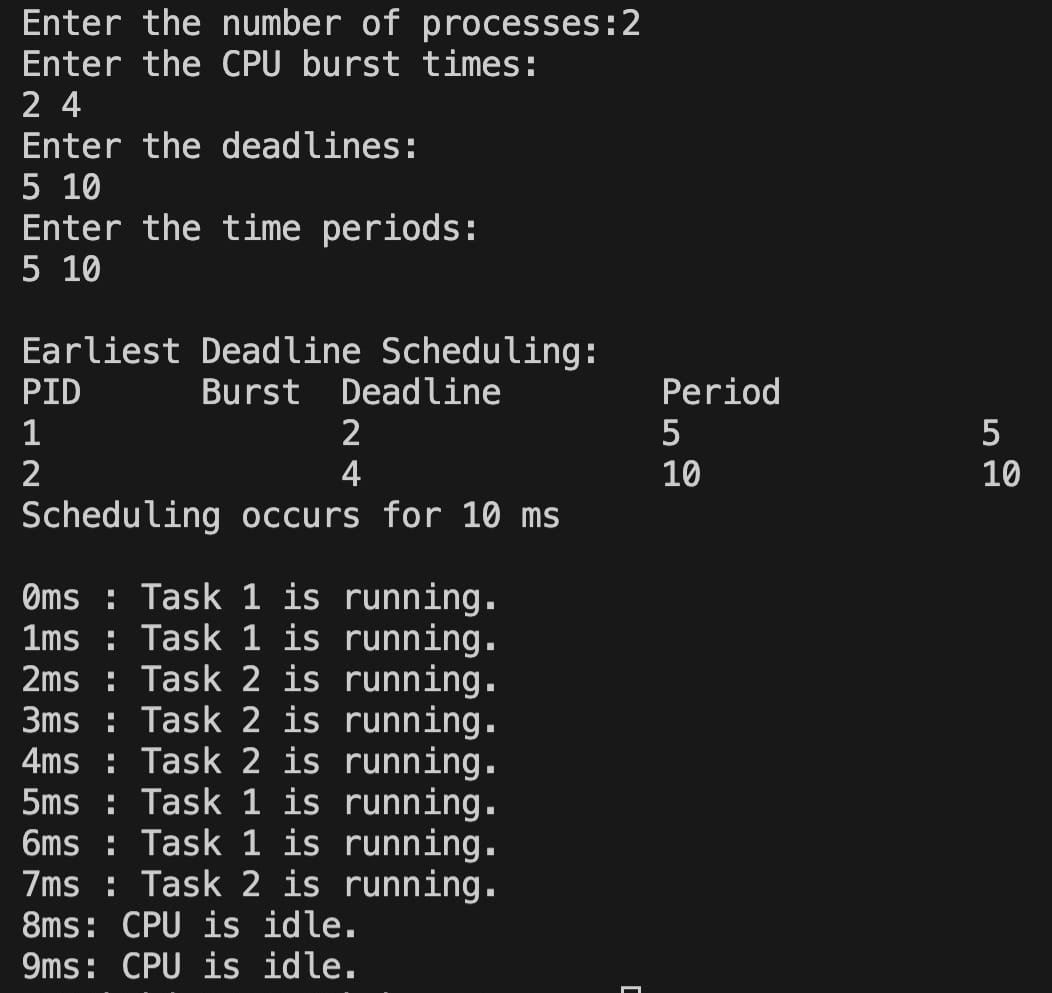
}

time++;

}

}

**Result:**



**c) Proportional Scheduling Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int main() {

srand(time(NULL));

int n;

printf("Enter number of processes:");

scanf("%d",&n);

int p[n],t[n],cum[n],m[n];int c=0;int total = 0,count=0;

printf("Enter tickets of the processes:\n");

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

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

c+=t[i];

cum[i]=c;

p[i]=i+1;

m[i]=0;

total+= t[i];

}

while(count<n){

int wt=rand()%total;

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

{

if (wt<cum[i] && m[i]==0)

{

printf("The winning number is %d and winning participant is: %d\n",wt,p[i]);

m[i]=1;count++;

}

}

}

printf("\nProbabilities:\n");

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

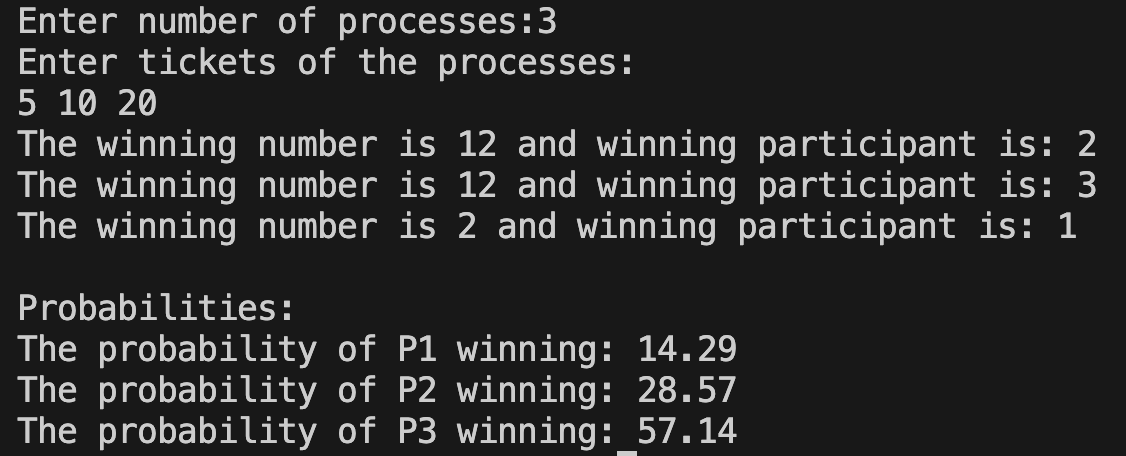
{

printf("The probability of P%d winning: %.2f\n",p[i],((double)t[i]/total\*100));

}

}

**Result:**



**Program - 5**

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

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 5

int buffer[MAX];

int empty = MAX;

int full = 0;

int mutex = 1;

int x = 0;

void custom\_wait(int\* s) {

while (\*s <= 0);

--(\*s);

}

void custom\_signal(int\* s) {

++(\*s);

}

void producer() {

custom\_wait(&mutex);

custom\_wait(&empty);

x++;

buffer[full] = x;

custom\_signal(&full);

custom\_signal(&mutex);

printf("Producer produced %d.\n", x);

printf("Empty = %d\n", empty);

printf("Buffer:\n");

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

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

}

printf("%d\n", buffer[full - 1]); /

}

void consumer() {

custom\_wait(&full);

custom\_wait(&mutex);

printf("Consumer consumed %d.\n", buffer[full - 1]);

full--;

custom\_signal(&empty);

custom\_signal(&mutex);

printf("Empty = %d\n", empty);

printf("Buffer:\n");

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

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

}

printf("\n");

}

int main() {

int ch;

while (1) {

printf("1.Produce\t2.Consume\t3.Exit\n");

scanf("%d", &ch);

switch (ch) {

case 1:

if (mutex == 1 && empty != 0) {

producer();

} else {

printf("Buffer is full\n");

}

break;

case 2:

if (mutex == 1 && full != 0) {

consumer();

} else {

printf("Buffer is empty\n");

}

break;

case 3:

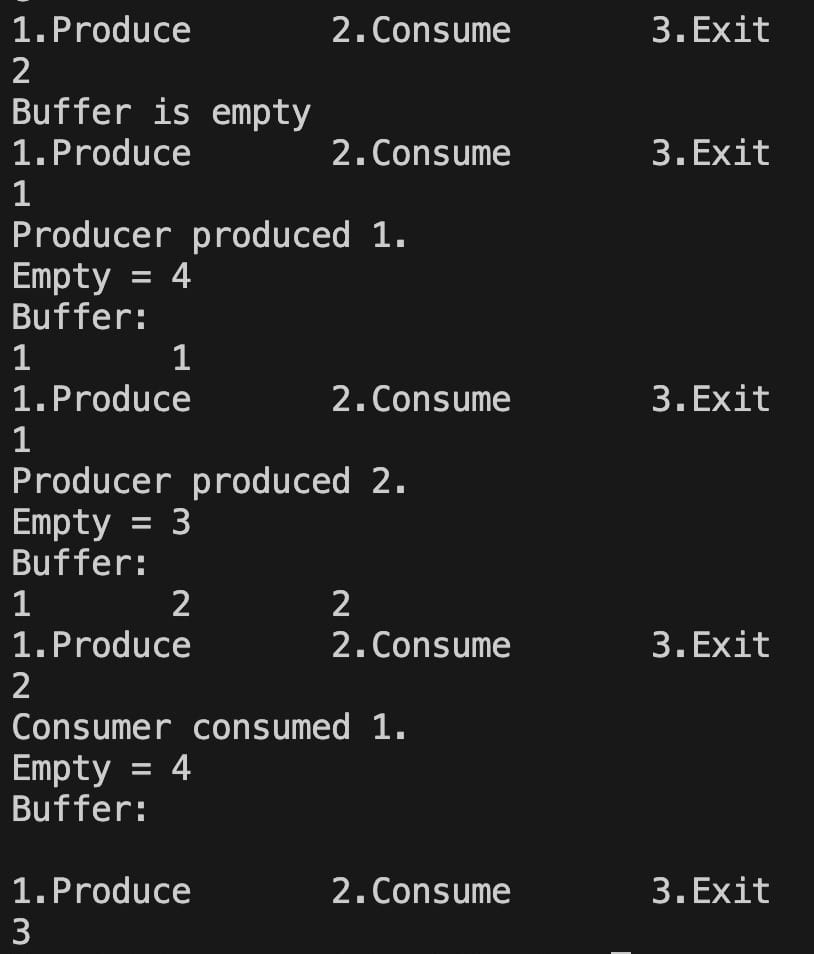
exit(0);

}

}

}

**Result:**

****

**Program - 6**

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

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_PHILOSOPHERS 5

void allow\_one\_to\_eat(int hungry[], int n) {

int isWaiting[MAX\_PHILOSOPHERS];

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

isWaiting[i] = 1;

}

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

printf("P %d is granted to eat\n", hungry[i]);

isWaiting[hungry[i]] = 0;

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

if (isWaiting[hungry[j]]) {

printf("P %d is waiting\n", hungry[j]);

}

}

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

isWaiting[k] = 1;

}

isWaiting[hungry[i]] = 0;

}

}

void allow\_two\_to\_eat(int hungry[], int n) {

if (n < 2 || n > MAX\_PHILOSOPHERS) {

printf("Invalid number of philosophers.\n");

return;

}

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

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

printf("P %d and P %d are granted to eat\n", hungry[i], hungry[j]);

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

if (k != i && k != j) {

printf("P %d is waiting\n", hungry[k]);

}

}

}

}

}

int main() {

int total\_philosophers, hungry\_count;

int hungry\_positions[MAX\_PHILOSOPHERS];

printf("DINING PHILOSOPHER PROBLEM\n");

printf("Enter the total no. of philosophers: ");

scanf("%d", &total\_philosophers);

if (total\_philosophers > MAX\_PHILOSOPHERS || total\_philosophers < 2) {

printf("Invalid number of philosophers.\n");

return 1;

}

printf("How many are hungry: ");

scanf("%d", &hungry\_count);

if (hungry\_count < 1 || hungry\_count > total\_philosophers) {

printf("Invalid number of hungry philosophers.\n");

return 1;

}

for (int i = 0; i < hungry\_count; i++) {

printf("Enter philosopher %d position: ", i + 1);

scanf("%d", &hungry\_positions[i]);

if (hungry\_positions[i] < 0 || hungry\_positions[i] >= total\_philosophers) {

printf("Invalid philosopher position.\n");

return 1;

}

}

int choice;

while (1) {

printf("\n1. One can eat at a time\n");

printf("2. Two can eat at a time\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

allow\_one\_to\_eat(hungry\_positions, hungry\_count);

break;

case 2:

allow\_two\_to\_eat(hungry\_positions, hungry\_count);

break;

case 3:

exit(0);

default:

printf("Invalid choice\n");

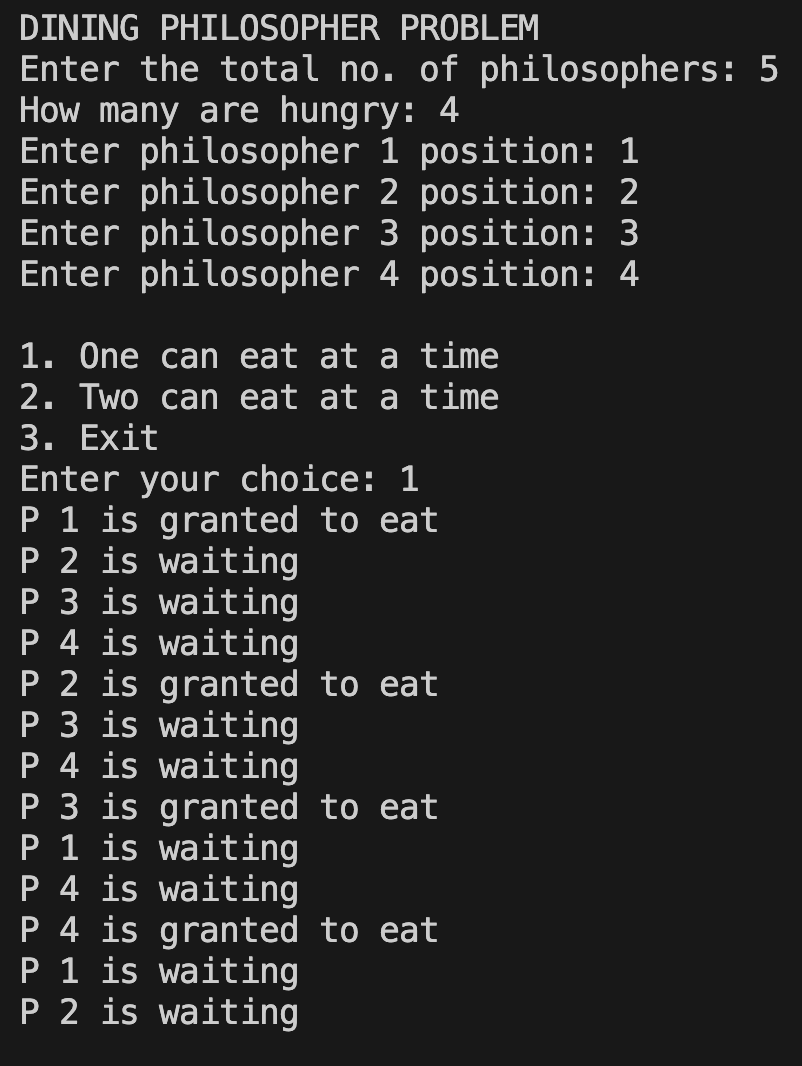
}

}

return 0;

}

**Result:**

****

**Program - 7**

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

**Code:**

#include <stdio.h>

#include <stdbool.h>

void calculateNeed(int P, int R, int need[P][R], int max[P][R], int allot[P][R]) {

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

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

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

}

bool isSafe(int P, int R, int processes[], int avail[], int max[][R], int allot[][R]) {

int need[P][R];

calculateNeed(P, R, need, max, allot);

bool finish[P];

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

finish[i] = 0;

}

int safeSeq[P];

int work[R];

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

work[i] = avail[i];

}

int count = 0;

while (count < P) {

bool found = false;

for (int p = 0; p < P; p++) {

if (finish[p] == 0) {

int j;

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

if (need[p][j] > work[j])

break;

if (j == R) {

printf("P%d is visited (", p);

for (int k = 0; k < R; k++) {

work[k] += allot[p][k];

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

}

printf(")\n");

safeSeq[count++] = p;

finish[p] = 1;

found = true;

}

}

}

if (found == false) {

printf("System is not in safe state\n");

return false;

}

}

printf("SYSTEM IS IN SAFE STATE\nThe Safe Sequence is -- (");

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

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

}

printf(")\n");

return true;

}

int main() {

int P, R;

printf("Enter number of processes: ");

scanf("%d", &P);

printf("Enter number of resources: ");

scanf("%d", &R);

int processes[P];

int avail[R];

int max[P][R];

int allot[P][R];

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

processes[i] = i;

}

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

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

printf("Enter allocation -- ");

for (int j = 0; j < R; j++) {

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

}

printf("Enter Max -- ");

for (int j = 0; j < R; j++) {

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

}

}

printf("Enter Available Resources -- ");

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

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

}

isSafe(P, R, processes, avail, max, allot);

printf("\nProcess\tAllocation\tMax\tNeed\n");

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

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

for (int j = 0; j < R; j++) {

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

}

printf("\t");

for (int j = 0; j < R; j++) {

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

}

printf("\t");

for (int j = 0; j < R; j++) {

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

}

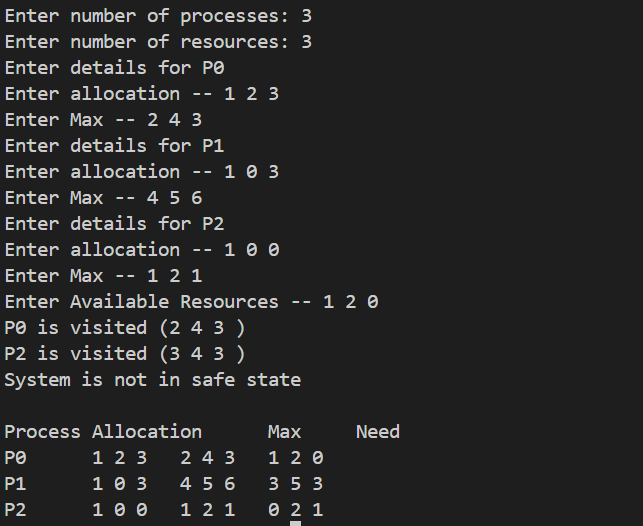
printf("\n");

}

return 0;

}

**Result:**

****

**Program -8**

Write a C program to simulate deadlock detection.

Code:

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

// Function prototypes

bool isCycle(int process, bool visited[], bool \*recStack, int allocation[][MAX\_RESOURCES], int n, int m);

bool isDeadlocked(int allocation[][MAX\_RESOURCES], int n, int m);

int main() {

int n, m; // n -> number of processes, m -> number of resources types

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter number of resource types: ");

scanf("%d", &m);

int allocation[MAX\_PROCESSES][MAX\_RESOURCES]; // Allocation matrix

// Input allocation matrix

printf("Enter allocation matrix:\n");

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

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

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

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

}

}

// Check if there is a deadlock

if (isDeadlocked(allocation, n, m)) {

printf("\nDeadlock detected.\n");

} else {

printf("\nNo deadlock detected.\n");

}

return 0;

}

// Function to detect if there is a cycle in the resource allocation graph using DFS

bool isCycle(int process, bool visited[], bool \*recStack, int allocation[][MAX\_RESOURCES], int n, int m) {

if (!visited[process]) {

visited[process] = true;

recStack[process] = true;

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

int resource = allocation[process][i];

if (resource > 0) {

if (!visited[resource] && isCycle(resource, visited, recStack, allocation, n, m)) {

return true;

} else if (recStack[resource]) {

return true;

}

}

}

}

recStack[process] = false;

return false;

}

// Function to check if there is a deadlock in the system

bool isDeadlocked(int allocation[][MAX\_RESOURCES], int n, int m) {

bool visited[MAX\_PROCESSES] = {false};

bool recStack[MAX\_PROCESSES] = {false};

// Check for deadlock using DFS traversal

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

if (!visited[i]) {

if (isCycle(i, visited, recStack, allocation, n, m)) {

return true;

}

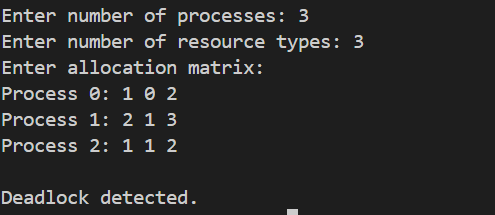
}

}

return false;

}

Result: --



**Program 9: -**

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

a) Worst-fit

b) Best-fit

c) First-fit

Code: --

#include <stdio.h>

#include <stdbool.h>

#define MAX\_BLOCKS 10

#define MAX\_PROCESSES 10

void firstFit(int blockSize[], int m, int processSize[], int n) {

int allocation[MAX\_PROCESSES];

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

allocation[i] = -1; // Initialize allocation as -1

}

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

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

if (blockSize[j] >= processSize[i]) {

allocation[i] = j;

blockSize[j] -= processSize[i];

break;

}

}

}

printf("\nFirst Fit Allocation:\n");

printf("Process No.\tProcess Size\tBlock No.\n");

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

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

if (allocation[i] != -1)

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

else

printf("Not Allocated\n");

}

}

void bestFit(int blockSize[], int m, int processSize[], int n) {

int allocation[MAX\_PROCESSES];

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

allocation[i] = -1; // Initialize allocation as -1

}

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

int bestIdx = -1;

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

if (blockSize[j] >= processSize[i]) {

if (bestIdx == -1 || blockSize[j] < blockSize[bestIdx])

bestIdx = j;

}

}

if (bestIdx != -1) {

allocation[i] = bestIdx;

blockSize[bestIdx] -= processSize[i];

}

}

printf("\nBest Fit Allocation:\n");

printf("Process No.\tProcess Size\tBlock No.\n");

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

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

if (allocation[i] != -1)

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

else

printf("Not Allocated\n");

}

}

void worstFit(int blockSize[], int m, int processSize[], int n) {

int allocation[MAX\_PROCESSES];

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

allocation[i] = -1; // Initialize allocation as -1

}

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

int worstIdx = -1;

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

if (blockSize[j] >= processSize[i]) {

if (worstIdx == -1 || blockSize[j] > blockSize[worstIdx])

worstIdx = j;

}

}

if (worstIdx != -1) {

allocation[i] = worstIdx;

blockSize[worstIdx] -= processSize[i];

}

}

printf("\nWorst Fit Allocation:\n");

printf("Process No.\tProcess Size\tBlock No.\n");

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

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

if (allocation[i] != -1)

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

else

printf("Not Allocated\n");

}

}

int main() {

int blockSize[MAX\_BLOCKS], processSize[MAX\_PROCESSES];

int m, n;

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

scanf("%d", &m);

printf("Enter the size of each memory block:\n");

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

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

}

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

scanf("%d", &n);

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

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

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

}

// Make copies of block sizes to reuse for different allocation methods

int blockSize1[MAX\_BLOCKS], blockSize2[MAX\_BLOCKS], blockSize3[MAX\_BLOCKS];

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

blockSize1[i] = blockSize2[i] = blockSize3[i] = blockSize[i];

}

firstFit(blockSize1, m, processSize, n);

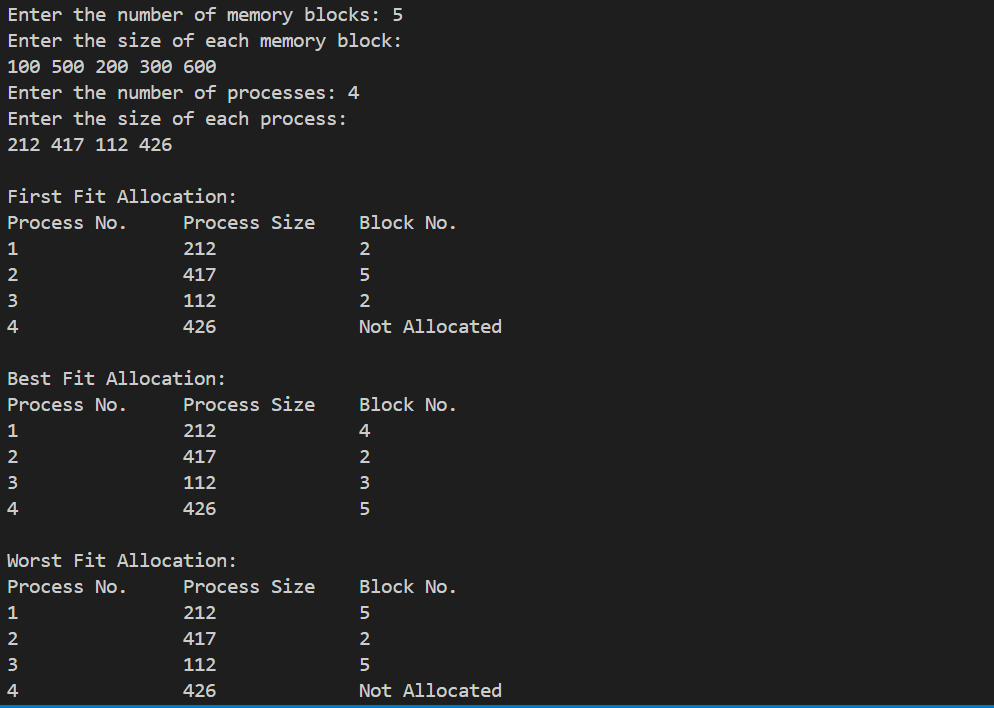
bestFit(blockSize2, m, processSize, n);

worstFit(blockSize3, m, processSize, n);

return 0;

}

Result :-



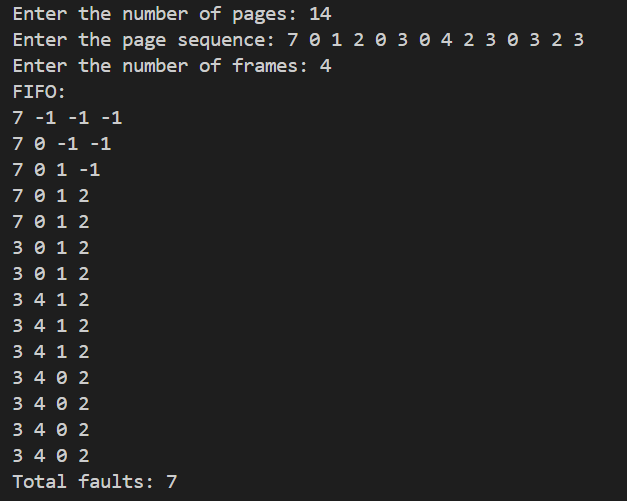
**Program 10:-**

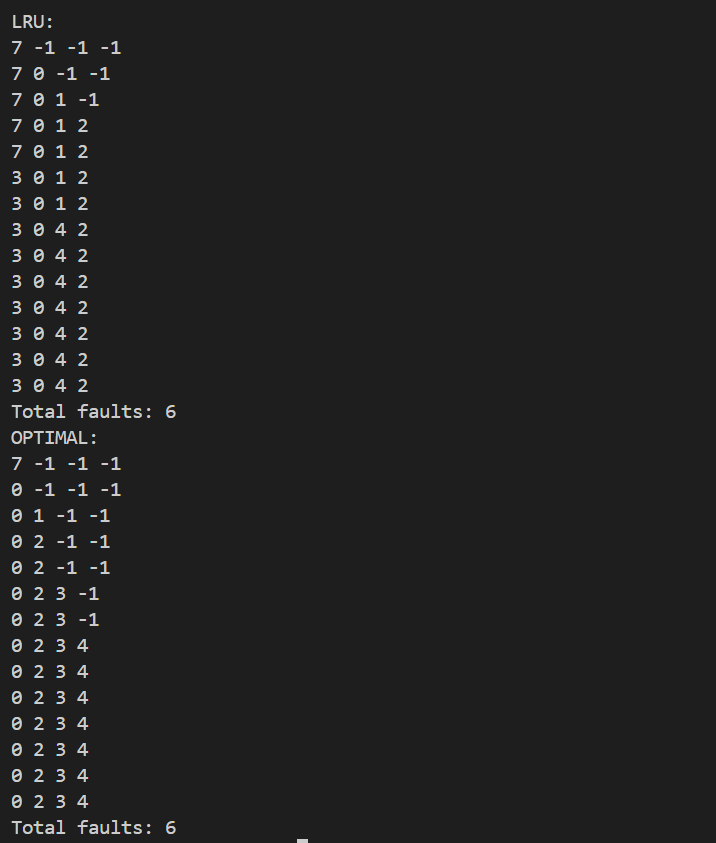
Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) Optimal

Code: -

#include <stdio.h>  
#include <stdlib.h>  
  
void printFrames(int frames[], int n) {  
    for (int i = 0; i < n; i++)  
        printf("%d ", frames[i]);  
    printf("\n");  
}  
  
void FIFO(int pages[], int n, int frame\_size) {  
    int frames[frame\_size];  
    for (int i = 0; i < frame\_size; i++) frames[i] = -1;  
    int index = 0, faults = 0;  
  
    printf("FIFO: \n");  
    for (int i = 0; i < n; i++) {  
        int found = 0;  
        for (int j = 0; j < frame\_size; j++) {  
            if (frames[j] == pages[i]) {  
                found = 1;  
                break;  
            }  
        }  
        if (!found) {  
            frames[index] = pages[i];  
            index = (index + 1) % frame\_size;  
            faults++;  
        }  
        printFrames(frames, frame\_size);  
    }  
    printf("Total faults: %d\n", faults);  
}  
  
void LRU(int pages[], int n, int frame\_size) {  
    int frames[frame\_size], last\_used[frame\_size];  
    for (int i = 0; i < frame\_size; i++) frames[i] = -1;  
    for (int i = 0; i < frame\_size; i++) last\_used[i] = 0;  
    int time = 0, faults = 0;  
  
    printf("LRU: \n");  
    for (int i = 0; i < n; i++) {  
        int found = 0;  
        for (int j = 0; j < frame\_size; j++) {  
            if (frames[j] == pages[i]) {  
                found = 1;  
                last\_used[j] = ++time;  
                break;  
            }  
        }  
        if (!found) {  
            int lru\_index = 0;  
            for (int j = 1; j < frame\_size; j++) {  
                if (last\_used[j] < last\_used[lru\_index])  
                    lru\_index = j;  
            }  
            frames[lru\_index] = pages[i];  
            last\_used[lru\_index] = ++time;  
            faults++;  
        }  
        printFrames(frames, frame\_size);  
    }  
    printf("Total faults: %d\n", faults);  
}  
  
void OPTIMAL(int pages[], int n, int frame\_size) {  
    int frames[frame\_size];  
    for (int i = 0; i < frame\_size; i++) frames[i] = -1;  
    int faults = 0;  
  
    printf("OPTIMAL: \n");  
    for (int i = 0; i < n; i++) {  
        int found = 0;  
        for (int j = 0; j < frame\_size; j++) {  
            if (frames[j] == pages[i]) {  
                found = 1;  
                break;  
            }  
        }  
        if (!found) {  
            int replace\_index = 0, farthest = -1;  
            for (int j = 0; j < frame\_size; j++) {  
                int k;  
                for (k = i + 1; k < n; k++) {  
                    if (frames[j] == pages[k]) {  
                        if (k > farthest) {  
                            farthest = k;  
                            replace\_index = j;  
                        }  
                        break;  
                    }  
                }  
                if (k == n) {  
                    replace\_index = j;  
                    break;  
                }  
            }  
            frames[replace\_index] = pages[i];  
            faults++;  
        }  
        printFrames(frames, frame\_size);  
    }  
    printf("Total faults: %d\n", faults);  
}  
  
int main() {  
    int n, frame\_size;  
  
    printf("Enter the number of pages: ");  
    scanf("%d", &n);  
  
    int \*pages = (int \*)malloc(n \* sizeof(int));  
    printf("Enter the page sequence: ");  
    for (int i = 0; i < n; i++) {  
        scanf("%d", &pages[i]);  
    }  
  
    printf("Enter the number of frames: ");  
    scanf("%d", &frame\_size);  
  
    FIFO(pages, n, frame\_size);  
    LRU(pages, n, frame\_size);  
    OPTIMAL(pages, n, frame\_size);  
  
    free(pages);  
    return 0;  
}

Result:-





Program 11

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

1. **FCFS**
2. **SCAN**
3. **c-SCAN**
4. **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];

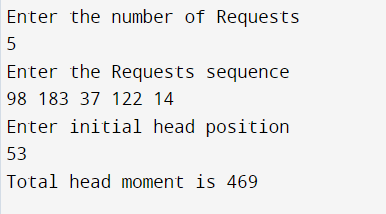
}

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

return 0;

}

Result:



**b)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];

}

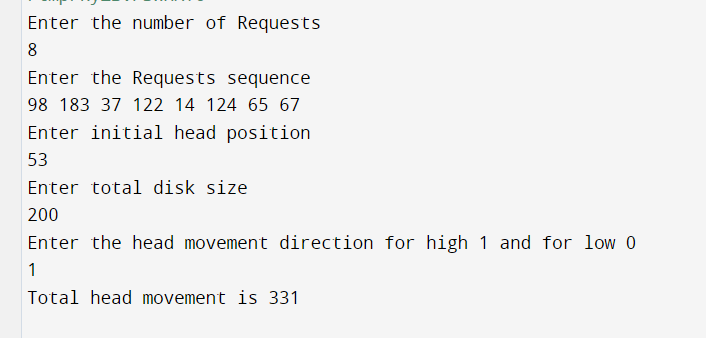
}

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

return 0;

}

Result:



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;

}

Result:

