**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

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



**LAB REPORT**

**on**

Operating Systems

**(23CS4PCOPS)**

***Submitted by:***

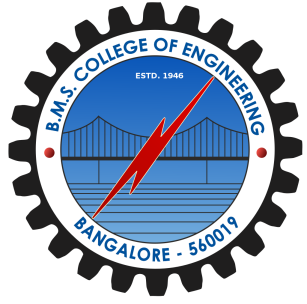
**Sudarshan Komar (1BM22CS291)**

***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**

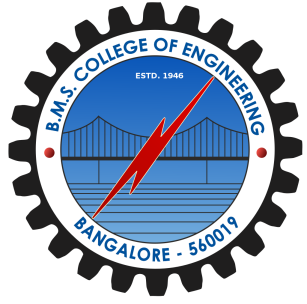
**June 2024 - August 2024**

# 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

**Sudarshan Komar (1BM22CS291),** 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.

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

Assistant Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengalur

**Table Of Contents**

|  |  |  |
| --- | --- | --- |
| **Lab Program No.** | **Program Details** | **Page No.** |
| 1 | FCFS, SJF AND SRTF | 2-7 |
| 2 | PRIORITY AND ROUND ROBIN | 8-17 |
| 3 | .MULTILEVEL QUEUE SCHEDULING | 18-20 |
| 4 | RATE-MONOTONIC AND EARLIEST DEADLINE FIRST | 21-28 |
| 5 | PRODUCER-CONSUMER PROBLEM | 29-30 |
| 6 | DINERS-PHILOSOPHERS PROBLEM | 31-32 |
| 7 | BANKERS ALGORITHM (DEADLOCK AVOIDANCE) | 33-35 |
| 8 | DEADLOCK DETECTION | 36-38 |
| 9 | CONTIGUOUS MEMORY ALLOCATION (FIRST, BEST, WORST FIT) | 39-43 |
| 10 | PAGE REPLACEMENT (FIFO, LRU, OPTIMAL) | 44-48 |

# 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**
3. **SRTF**

**CODE:**

#include <stdio.h>

#include <limits.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);

}

void SRTF()

{

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

int completed = 0, current\_time = 0, min\_index = -1;

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

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

{

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

}

while (completed != n)

{

int min\_burst\_time = INT\_MAX;

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

{

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

{

if (Remaining\_time[i] < min\_burst\_time)

{

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

min\_index = i;

}

if (Remaining\_time[i] == min\_burst\_time)

{

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

{

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

min\_index = i;

}

}

}

}

if (min\_index != -1)

{

Remaining\_time[min\_index]--;

current\_time++;

if (Remaining\_time[min\_index] == 0)

{

is\_completed[min\_index] = 1;

completed++;

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

Waiting\_time[min\_index] = Turn\_around\_time[min\_index] - Burst\_time[min\_index];

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

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

min\_index = -1;

}

}

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\n3. SRTF Scheduling\n");

printf("\nEnter your choice: ");

scanf("%d", &choice);

switch (choice)

{

case 1:

FCFS();

break;

case 2:

SJF();

break;

case 3:

SRTF();

break;

default:

printf("Invalid Input!!!\n");

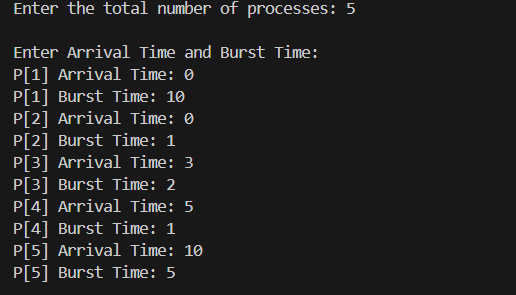
}

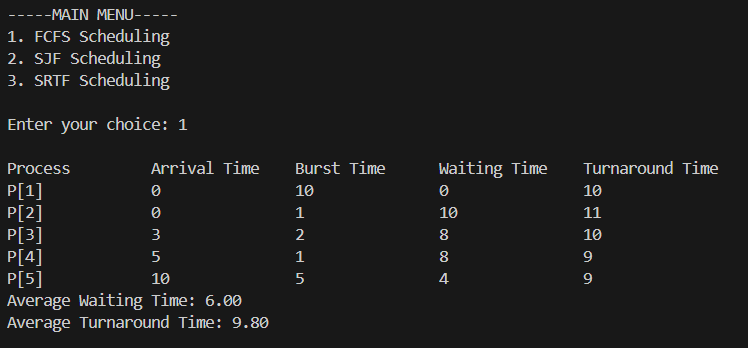
}

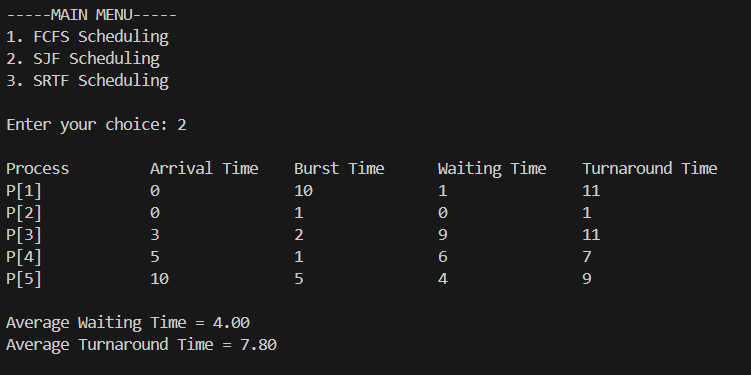
return 0;

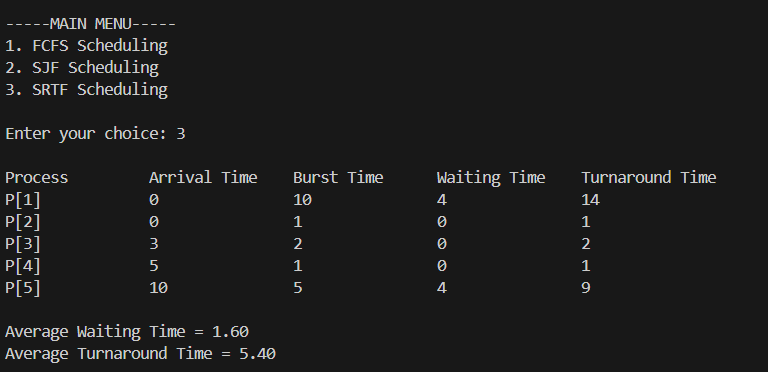
}

**OUTPUTS:**









**LAB-2**

**Question:**

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

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

## CODE:

**Priority (Non-pre-emptive) #lower value higher priority**

#include <stdio.h>

#include <stdlib.h>

struct process

{

int process\_id;

int burst\_time;

int priority;

int arrival\_time;

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);

printf("Enter the arrival time: ");

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

}

priority\_scheduling(proc, n);

return 0;

}

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

{

int i;

int current\_time = 0;

wt[0] = 0;

current\_time = proc[0].arrival\_time + proc[0].burst\_time;

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

{

if (current\_time < proc[i].arrival\_time)

{

current\_time = proc[i].arrival\_time;

}

wt[i] = current\_time - proc[i].arrival\_time;

current\_time += proc[i].burst\_time;

}

}

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\tArrival Time\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\t\t%d", proc[i].process\_id, proc[i].arrival\_time,

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;

// Sort based on arrival time

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

{

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

{

if (proc[i].arrival\_time > proc[j].arrival\_time)

{

temp = proc[i];

proc[i] = proc[j];

proc[j] = temp;

}

}

}

// Sort based on priority (for processes with the same arrival time)

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

{

pos = i;

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

{

if (proc[j].arrival\_time <= proc[i].arrival\_time && proc[j].priority < proc[pos].priority)

{

pos = j;

}

}

if (pos != i)

{

temp = proc[i];

proc[i] = proc[pos];

proc[pos] = temp;

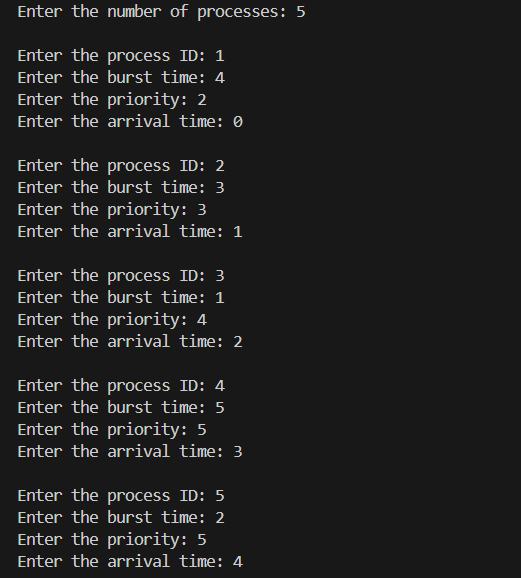
}

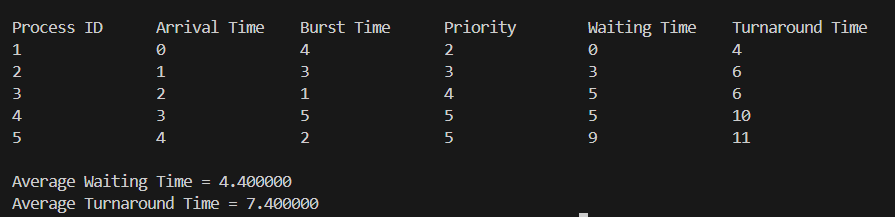
}

find\_average\_time(proc, n);

}

**OUTPUTS:**





**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);

}

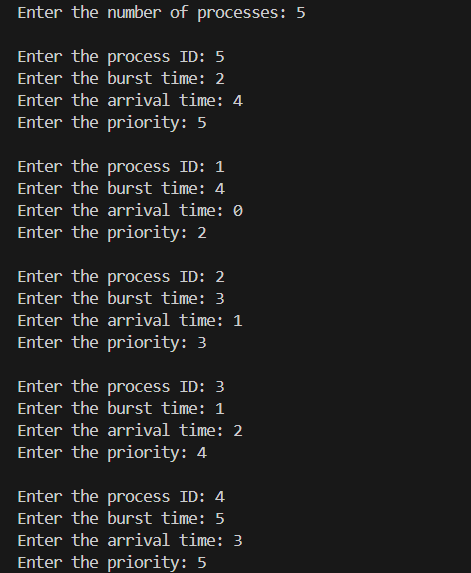
void priority\_scheduling(struct process proc[], int n)

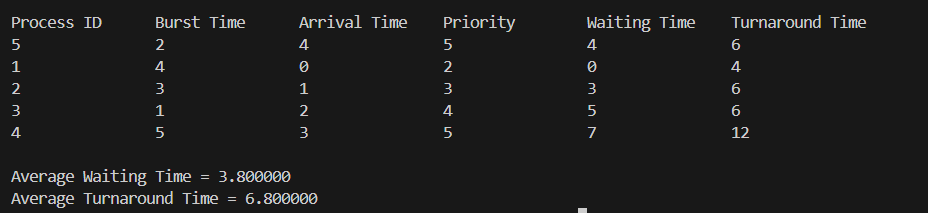
{

find\_average\_time(proc, n);

}

**OUTPUTS:**





**(b) Round Robin (Non-pre-emptive)**

**Code:**

#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]);

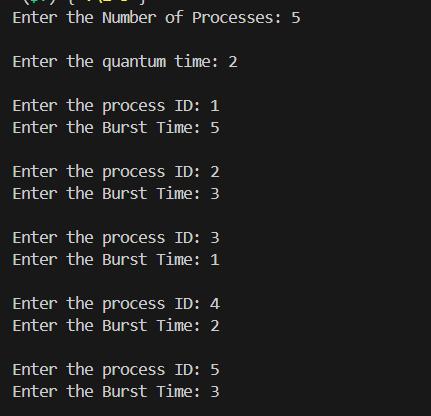
}

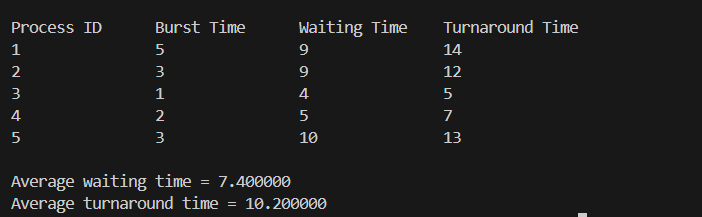
findAvgTime(processes, n, burst\_time, quantum);

return 0;

}

**OUTPUT:**





# LAB - 3

**Question 1:**

**Write a C program to simulate a 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>

#define MAX\_PROCESSES 50

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

{

int temp;

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

{

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

{

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

{

// Swap arrival times

temp = at[i];

at[i] = at[j];

at[j] = temp;

// Swap burst times

temp = bt[i];

bt[i] = bt[j];

bt[j] = temp;

// Swap process IDs

temp = proc\_id[i];

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

proc\_id[j] = temp;

}

}

}

}

void fcfs(int at[], int bt[], int ct[], int tat[], int wt[], int n, int \*c)

{

double ttat = 0.0, twt = 0.0;

// Completion time

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

{

if (\*c >= at[i])

{

\*c += bt[i];

}

else

{

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

}

ct[i] = \*c;

}

// Turnaround time

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

{

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

}

// Waiting time

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

{

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

}

}

int main()

{

int sn, un, c = 0;

int n = 0;

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

scanf("%d", &sn);

n = sn;

int sproc\_id[MAX\_PROCESSES], sat[MAX\_PROCESSES], sbt[MAX\_PROCESSES];

int sct[MAX\_PROCESSES], stat[MAX\_PROCESSES], swt[MAX\_PROCESSES];

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

{

sproc\_id[i] = i + 1;

}

printf("Enter arrival times of the system processes:\n");

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

{

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

}

printf("Enter burst times of the system processes:\n");

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

{

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

}

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

scanf("%d", &un);

n = un;

int uproc\_id[MAX\_PROCESSES], uat[MAX\_PROCESSES], ubt[MAX\_PROCESSES];

int uct[MAX\_PROCESSES], utat[MAX\_PROCESSES], uwt[MAX\_PROCESSES];

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

{

uproc\_id[i] = i + 1;

}

printf("Enter arrival times of the user processes:\n");

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

{

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

}

printf("Enter burst times of the user processes:\n");

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

{

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

}

sort(sproc\_id, sat, sbt, sn);

sort(uproc\_id, uat, ubt, un);

fcfs(sat, sbt, sct, stat, swt, sn, &c);

fcfs(uat, ubt, uct, utat, uwt, un, &c);

printf("\nScheduling:\n");

printf("System processes:\n");

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

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

{

printf("%d\t%d\t%d\t%d\t%d\t%d\n", sproc\_id[i], sat[i], sbt[i], sct[i], stat[i], swt[i]);

}

printf("User processes:\n");

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

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

{

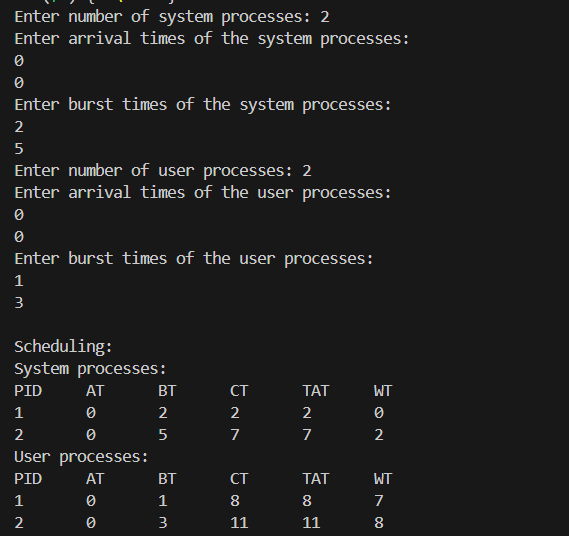
printf("%d\t%d\t%d\t%d\t%d\t%d\n", uproc\_id[i], uat[i], ubt[i], uct[i], utat[i], uwt[i]);

}

return 0;

}

**OUTPUT:**



**LAB - 4**

**Question 1:**

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

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

**CODE:**

1. **Rate monotonic**

#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;

}

void 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);

// LCM

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]);

// feasibility

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);

// RMS

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;

}

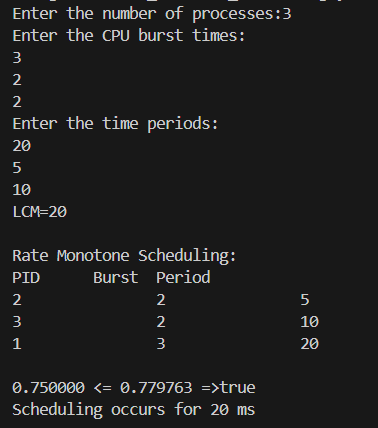
}

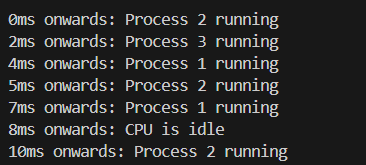
time++;

}

}

**OUTPUT:**





1. **Earliest-deadline First**

#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;

}

void 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);

// LCM

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);

// EDF

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);

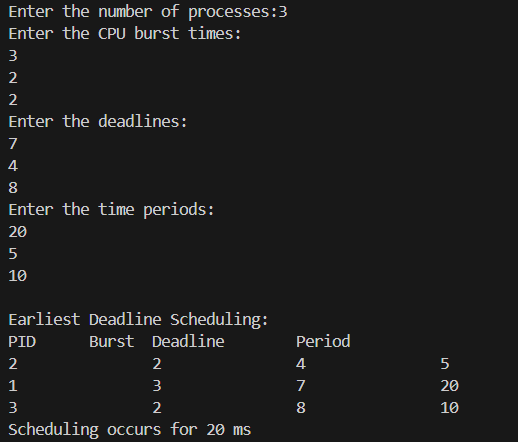
}

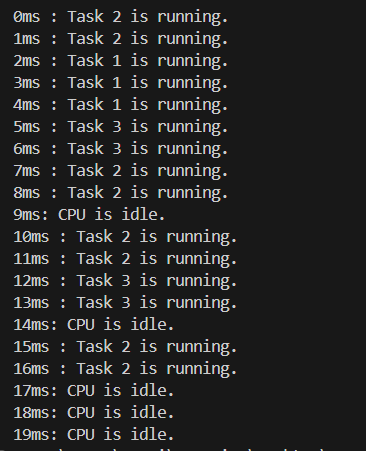
time++;

}

}

**OUTPUT:**





**LAB - 5**

**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);

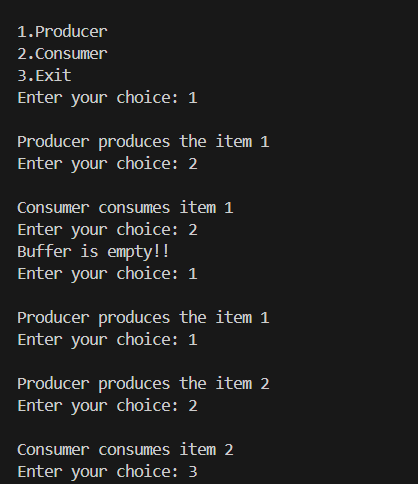
printf("\nConsumer consumes item %d", x);

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++) {

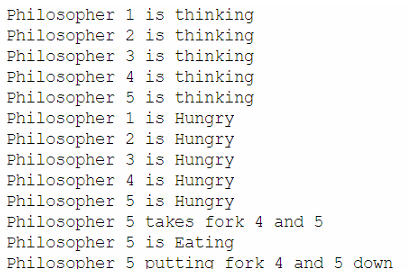
pthread\_join(thread\_id[i], NULL);

}

return 0;

}

**OUTPUT:**



**LAB - 6**

**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]);

}

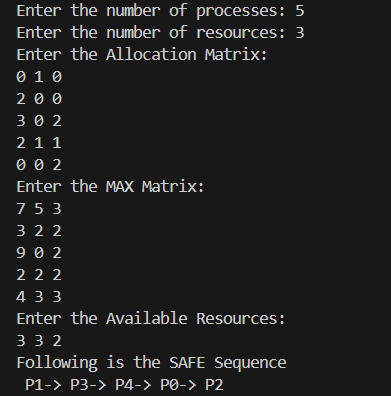
printf(" P%d\n", ans[n - 1]);

}

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)

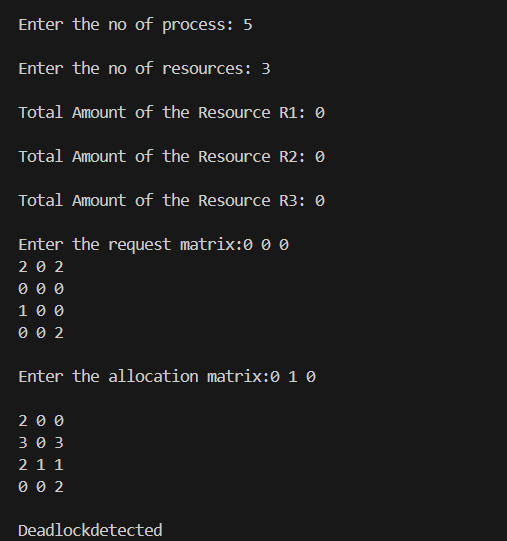
printf("\nDeadlock detected");

else

printf("\nNoDeadlock possible");

}

**OUTPUT:**



**LAB-7**

**Question 1:**

**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 <stdlib.h>

#define MAX 25

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

int ff[MAX] = {0};

int allocated[MAX] = {0};

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

ff[i] = -1;

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

if (allocated[j] == 0 && b[j] >= f[i]) {

ff[i] = j;

allocated[j] = 1;

break;

}

}

}

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

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

if (ff[i] != -1)

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

else

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

}

}

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

int ff[MAX] = {0};

int allocated[MAX] = {0};

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

int best = -1;

ff[i] = -1;

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

if (allocated[j] == 0 && b[j] >= f[i]) {

if (best == -1 || b[j] < b[best])

best = j;

}

}

if (best != -1) {

ff[i] = best;

allocated[best] = 1;

}

}

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

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

if (ff[i] != -1)

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

else

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

}

}

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

int ff[MAX] = {0};

int allocated[MAX] = {0};

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

int worst = -1;

ff[i] = -1;

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

if (allocated[j] == 0 && b[j] >= f[i]) {

if (worst == -1 || b[j] > b[worst])

worst = j;

}

}

if (worst != -1) {

ff[i] = worst;

allocated[worst] = 1;

}

}

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

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

if (ff[i] != -1)

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

else

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

}

}

int main() {

int nb, nf, choice;

printf("Memory Management Scheme");

printf("\nEnter the number of blocks: ");

scanf("%d", &nb);

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

scanf("%d", &nf);

int b[nb], f[nf];

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

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

printf("Block %d: ", i + 1);

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

}

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

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

printf("File %d: ", i + 1);

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

}

while (1) {

printf("\n1. First Fit\n2. Best Fit\n3. Worst Fit\n4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("\n\tMemory Management Scheme - First Fit\n");

firstFit(nb, nf, b, f);

break;

case 2:

printf("\n\tMemory Management Scheme - Best Fit\n");

bestFit(nb, nf, b, f);

break;

case 3:

printf("\n\tMemory Management Scheme - Worst Fit\n");

worstFit(nb, nf, b, f);

break;

case 4:

printf("\nExiting...\n");

exit(0);

break;

default:

printf("\nInvalid choice.\n");

break;

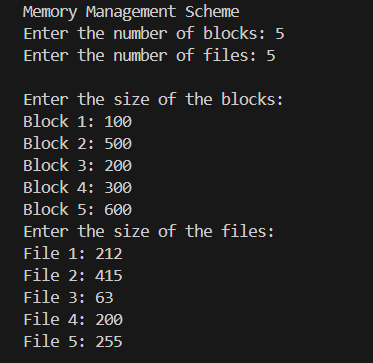
}

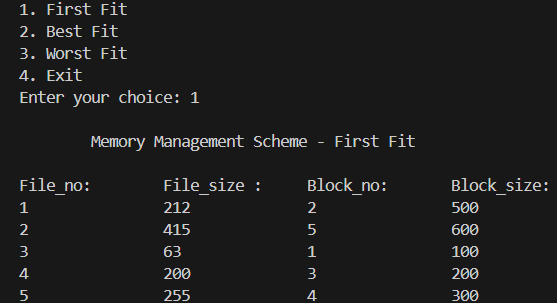
}

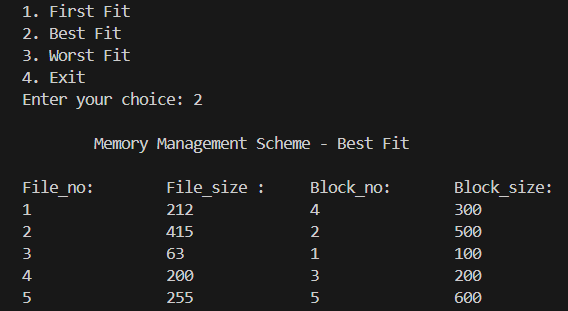
return 0;

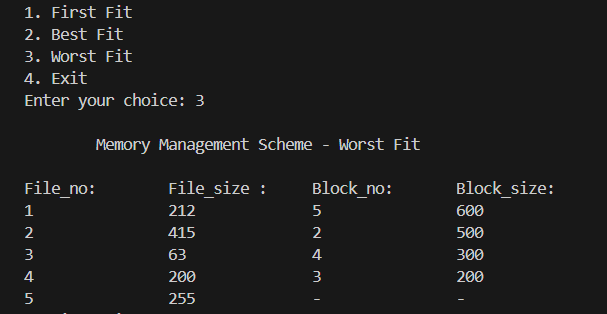
}

**OUTPUT:**









**LAB-8**

**Question 1:**

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

**(a) FIFO**

**(b) LRU**

**(c) Optimal**

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_FRAMES 10

#define MAX\_PAGES 100

int x=0;

void printFrames(int frames[], int framesCount, bool fault) {

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

if (frames[i] == -1)

printf(" ");

else{

printf("%d ", frames[i]);

}

}

if (fault) printf(" - page fault %d",++x);

else printf(" ");

printf("\n");

}

int isPageInFrames(int page, int frames[], int framesCount) {

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

if (frames[i] == page) {

return 1;

}

}

return 0;

}

int getOptimalReplacementIndex(int pages[], int currentIndex, int frames[], int framesCount, int pagesCount) {

int farthest = currentIndex;

int index = -1;

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

int j;

for (j = currentIndex; j < pagesCount; j++) {

if (frames[i] == pages[j]) {

if (j > farthest) {

farthest = j;

index = i;

}

break;

}

}

if (j == pagesCount) {

return i;

}

}

return index == -1 ? 0 : index;

}

void fifo(int pages[], int pagesCount, int framesCount) {

x=0;

printf("FIFO Page Replacement Algorithm\n");

int frames[MAX\_FRAMES];

int currentFrame = 0;

int pageFaults = 0;

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

frames[i] = -1;

}

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

bool fault = false;

if (!isPageInFrames(pages[i], frames, framesCount)) {

frames[currentFrame] = pages[i];

currentFrame = (currentFrame + 1) % framesCount;

fault = true;

pageFaults++;

}

printFrames(frames, framesCount, fault);

}

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

}

void optimal(int pages[], int pagesCount, int framesCount) {

x=0;

printf("Optimal Page Replacement Algorithm\n");

int frames[MAX\_FRAMES];

int pageFaults = 0;

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

frames[i] = -1;

}

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

bool fault = false;

if (!isPageInFrames(pages[i], frames, framesCount)) {

if (frames[i % framesCount] == -1) {

frames[i % framesCount] = pages[i];

} else {

int index = getOptimalReplacementIndex(pages, i + 1, frames, framesCount, pagesCount);

frames[index] = pages[i];

}

fault = true;

pageFaults++;

}

printFrames(frames, framesCount, fault);

}

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

}

void lru(int pages[], int pagesCount, int framesCount) {

x=0;

printf("LRU Page Replacement Algorithm\n");

int frames[MAX\_FRAMES];

int pageFaults = 0;

int recent[MAX\_FRAMES];

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

frames[i] = -1;

recent[i] = -1;

}

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

bool fault = false;

if (!isPageInFrames(pages[i], frames, framesCount)) {

int lruIndex = 0;

for (int j = 1; j < framesCount; j++) {

if (recent[j] < recent[lruIndex]) {

lruIndex = j;

}

}

frames[lruIndex] = pages[i];

fault = true;

pageFaults++;

}

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

if (frames[j] == pages[i]) {

recent[j] = i;

}

}

printFrames(frames, framesCount, fault);

}

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

}

int main() {

int pages[MAX\_PAGES];

int pagesCount;

int framesCount;

printf("Enter number of frames: ");

scanf("%d", &framesCount);

printf("Enter number of pages: ");

scanf("%d", &pagesCount);

printf("Enter the page reference string: ");

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

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

}

fifo(pages, pagesCount, framesCount);

optimal(pages, pagesCount, framesCount);

lru(pages, pagesCount, framesCount);

return 0;

}

**OUTPUT:**

