**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

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

****

**LAB REPORT**

**on**

**OPERATING SYSTEMS**

***Submitted by***

**UDAY S (1BM22CS420)**

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

**BACHELOR OF ENGINEERING**

***in***

**COMPUTER SCIENCE AND ENGINEERING**



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

**(Autonomous Institution under VTU)**

**BENGALURU-560019**

**June-2023 to September-2023**

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

**Bull Temple Road, Bangalore 560019**

(Affiliated To Visvesvaraya Technological University, Belgaum)

**Department of Computer Science and Engineering**



**CERTIFICATE**

This is to certify that the Lab work entitled “OPERATING SYSTEMS” carried out by **UDAY S (1BM22CS420),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the academic semester June-2023 to September-2023. The Lab report has been approved as it satisfies the academic requirements in respect of a OPERATING SYSTEMS **(22CS4PCOPS)** work prescribed for the said degree.

Name of the Lab-In charge:               Dr. Jyothi S Nayak

Basavaraj Jakkali

Associate Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

**Index Sheet**

|  |  |  |
| --- | --- | --- |
| **Lab Program No.** | **Program Details** | **Page No.** |
| 1 | Write a C program to simulate the following non-pre-emptive CPU  scheduling algorithm to find turnaround time and waiting time.  🡪FCFS  🡪 SJF (pre-emptive & Non-pre-emptive) | 6 |
| 2 | Write a C program to simulate the following CPU scheduling  algorithm to find turnaround time and waiting time.  🡪 Priority (pre-emptive & Non-pre-emptive)  🡪Round Robin (Experiment with different quantum sizes for RR  algorithm) | 15 |
| 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. | 27 |
| 4 | Write a C program to simulate Real-Time CPU Scheduling  algorithms:  a) Rate- Monotonic  b) Earliest-deadline First | 33 |
| 5 | Write a C program to simulate producer-consumer problem using  semaphores. | 51 |
| 6 | Write a C program to simulate the concept of Dining-Philosophers  problem. | 55 |
| 7 | Write a C program to simulate Bankers algorithm for the purpose of  deadlock avoidance. | 61 |
| 8 | Write a program to simulate deadlock detection. | 65 |
| 9 | Write a program to simulate the following contiguous memory allocation techniques.  a)Worst-fit  b)Best-fit  c)First-fit | 68 |
| 10 | Write a C program to simulate paging technique of memory management. | 75 |
| 11 | Write a C program to simulate disk scheduling algorithms  a)FCFS  b)SCAN  c)C-SCAN | 82 |
| 12 | Write a C program to simulate disk scheduling algorithms  a)SSTF  b)LOOK  c)C-LOOK | 90 |
| 13 | Write a C program to simulate page replacement algorithms  a)FIFO  b)LRU  c)OPTIMAL | 96 |

**Course Outcome**

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

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

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

**🡪FCFS**

#include<stdio.h>

#include<conio.h>

void main()

{

int n,art[20],burst[20],wait[20],i,s=0,sum=0,tt[20],sum1=0;

float avg,avg1;

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

scanf("%d",&n);

printf("\nEnter the arrival time for %d processes\n",n);

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

{

printf("\nArrival time of %d process=",i);

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

}

printf("\nEnter the Burst Time for %d processes\n",n);

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

{

printf("\nBurst Time of %d process=",i);

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

}

printf("\Gmatt Chart is\n");

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

{

tt[i]=s+burst[i]-art[i];

wait[i]=tt[i]-burst[i];

printf("\nProcess %d starts at %d and ends at %d",i,s,burst[i]+s);

printf("\nTurn Around Time for %d process is:%d",i,tt[i]);

printf("\nWaiting Time for %d process is:%d",i,wait[i]);

s=s+burst[i];

sum=sum+tt[i];

sum1=sum1+wait[i];

}

avg=(float)sum/n;

avg1=(float)sum1/n;

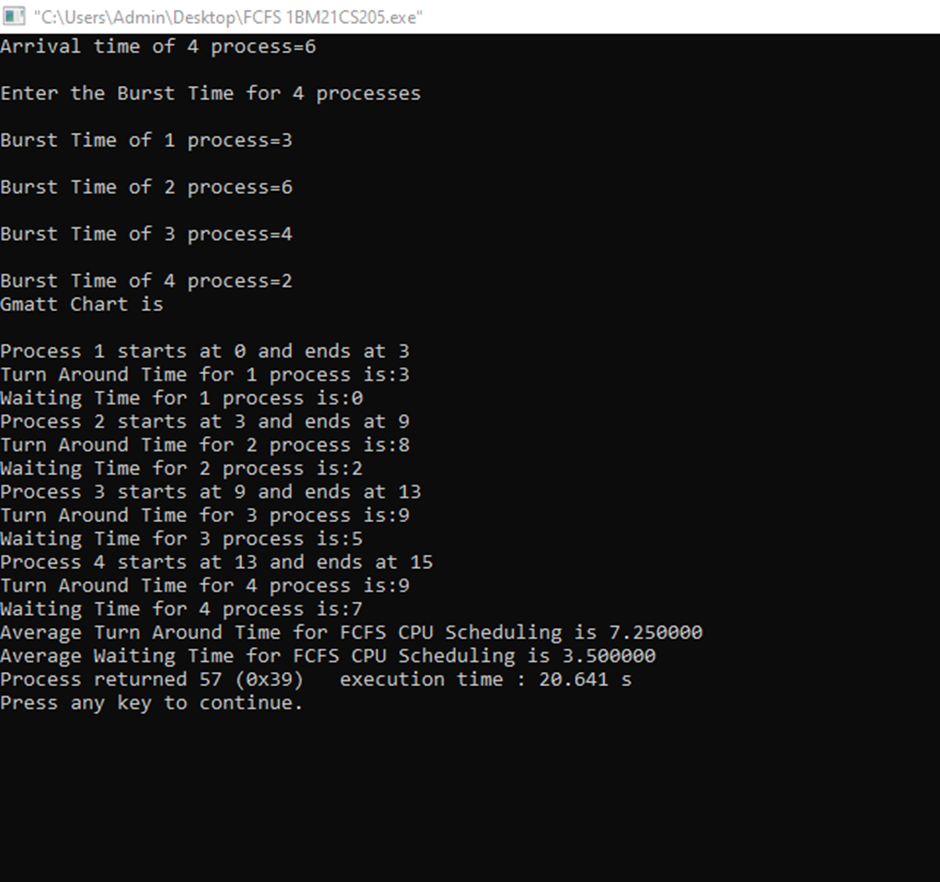
printf("\nAverage Turn Around Time for FCFS CPU Scheduling is %f",avg);

printf("\nAverage Waiting Time for FCFS CPU Scheduling is %f",avg1);

getch();

}

**SAMPLE OUTPUT**



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

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

struct Process {

int pid;

int arrival\_time;

int burst\_time;

int remaining\_time;

int turnaround\_time;

int waiting\_time;

};

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

int i,j,count=0,m;

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

{

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

count++;

}

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

{

if(count==n)

{

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

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

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

struct Process temp = processes[j];

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

processes[j + 1] = temp;

}

}

}

}

else

{

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

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

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

struct Process temp = processes[j];

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

processes[j + 1] = temp;

}

}

}

}

}

int total\_time = 0;

double total\_turnaround\_time = 0;

double total\_waiting\_time = 0;

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

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

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

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

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

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

}

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

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

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

}

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

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

}

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

int total\_time = 0,i;

int completed = 0;

while (completed < n) {

int shortest\_burst = -1;

int next\_process = -1;

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

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

if (shortest\_burst == -1 || processes[i].remaining\_time < shortest\_burst) {

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

next\_process = i;

}

}

}

if (next\_process == -1) {

total\_time++;

continue;

}

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

total\_time++;

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

completed++;

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

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

}

}

double total\_turnaround\_time = 0;

double total\_waiting\_time = 0;

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

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

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

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

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

}

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

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

}

int main() {

int n, quantum,i,choice;

struct Process processes[MAX\_PROCESSES];

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

scanf("%d", &n);

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

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

printf("Enter arrival time: ");

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

printf("Enter burst time: ");

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

processes[i].pid = i + 1;

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

processes[i].turnaround\_time = 0;

processes[i].waiting\_time = 0;

}

while(1)

{

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

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

printf("2. SRTF\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

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

sjf\_nonpreemptive(processes, n);

break;

case 2:

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

sjf\_preemptive(processes, n);

break;

case 3:exit(0);

break;

default:

printf("Invalid choice!\n");

return 1;

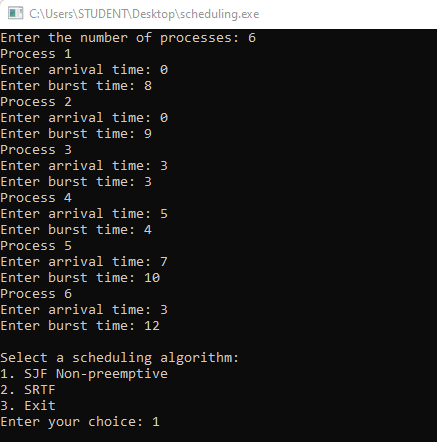
}

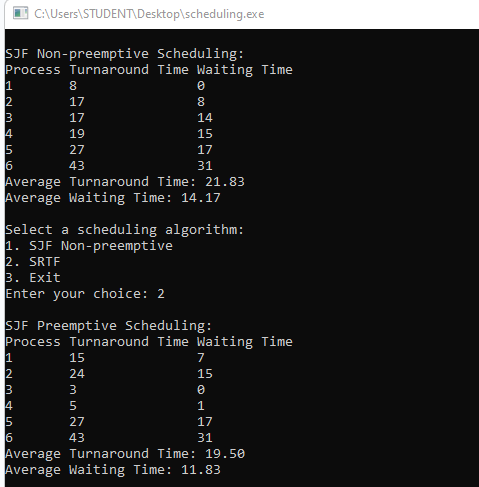
}

return 0;

}

**SAMPLE OUTPUT**

****

****

**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)**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

struct Process {

int pid;

int arrival\_time;

int burst\_time;

int priority;

int remaining\_time;

int turnaround\_time;

int waiting\_time;

};

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

int i,j,count=0,m;

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

{

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

count++;

}

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

{

if(count==n)

{

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

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

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

struct Process temp = processes[j];

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

processes[j + 1] = temp;

}

}

}

}

else

{

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

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

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

struct Process temp = processes[j];

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

processes[j + 1] = temp;

}

}

}

}

}

int total\_time = 0;

double total\_turnaround\_time = 0;

double total\_waiting\_time = 0;

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

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

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

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

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

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

}

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

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

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

}

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

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

}

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

int total\_time = 0,i;

int completed = 0;

while (completed < n) {

int highest\_priority = -1;

int next\_process = -1;

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

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

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

highest\_priority = processes[i].priority;

next\_process = i;

}

}

}

if (next\_process == -1) {

total\_time++;

continue;

}

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

total\_time++;

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

completed++;

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

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

}

}

double total\_turnaround\_time = 0;

double total\_waiting\_time = 0;

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

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

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

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

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

}

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

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

}

int main() {

int n, quantum,i,choice;

struct Process processes[MAX\_PROCESSES];

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

scanf("%d", &n);

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

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

printf("Enter arrival time: ");

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

printf("Enter burst time: ");

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

printf("Enter priority: ");

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

processes[i].pid = i + 1;

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

processes[i].turnaround\_time = 0;

processes[i].waiting\_time = 0;

}

while(1)

{

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

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

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

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

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

priority\_nonpreemptive(processes, n);

break;

case 2:

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

priority\_preemptive(processes, n);

break;

case 3:exit(0);

break;

default:

printf("Invalid choice!\n");

return 1;

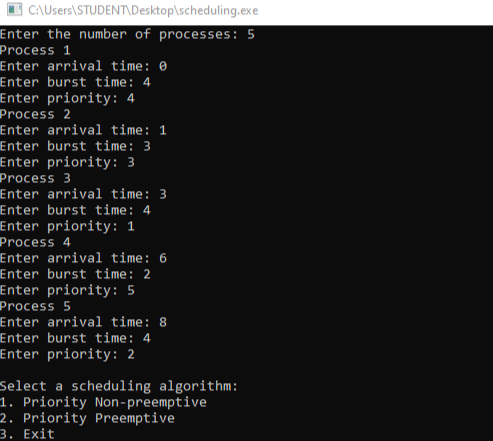
}

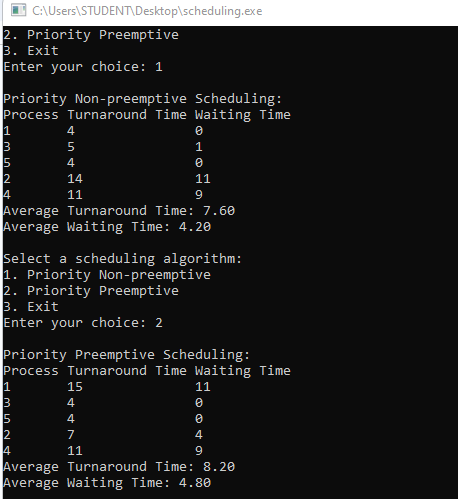
}

return 0;

}

**SAMPLE OUTPUT**

****

****

**🡪Round Robin (Experiment with different quantum sizes for RR**

algorithm)

#include<stdio.h>

#include<limits.h>

#include<stdbool.h>

struct P{

int AT,BT,ST[20],WT,FT,TAT,pos;

};

int quant;

int main(){

int n,i,j;

printf("Enter the no. of processes :");

scanf("%d",&n);

struct P p[n];

printf("Enter the quantum \n");

scanf("%d",&quant);

printf("Enter the process numbers \n");

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

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

printf("Enter the Arrival time of processes \n");

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

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

printf("Enter the Burst time of processes \n");

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

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

int c=n,s[n][20];

float time=0,mini=INT\_MAX,b[n],a[n];

int index=-1;

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

b[i]=p[i].BT;

a[i]=p[i].AT;

for(j=0;j<20;j++){

s[i][j]=-1;

}

}

int tot\_wt,tot\_tat;

tot\_wt=0;

tot\_tat=0;

bool flag=false;

while(c!=0){

mini=INT\_MAX;

flag=false;

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

float p=time+0.1;

if(a[i]<=p && mini>a[i] && b[i]>0){

index=i;

mini=a[i];

flag=true;

}

}

if(!flag){

time++;

continue;

}

j=0;

while(s[index][j]!=-1){

j++;

}

if(s[index][j]==-1){

s[index][j]=time;

p[index].ST[j]=time;

}

if(b[index]<=quant){

time+=b[index];

b[index]=0;

}

else{

time+=quant;

b[index]-=quant;

}

if(b[index]>0){

a[index]=time+0.1;

}

if(b[index]==0){

c--;

p[index].FT=time;

p[index].WT=p[index].FT-p[index].AT-p[index].BT;

tot\_wt+=p[index].WT;

p[index].TAT=p[index].BT+p[index].WT;

tot\_tat+=p[index].TAT;

}

}

printf("Process number ");

printf("Arrival time ");

printf("Burst time ");

printf("\tStart time");

j=0;

while(j!=10){

j+=1;

printf(" ");

}

printf("\t\tFinal time");

printf("\tWait Time ");

printf("\tTurnAround Time \n");

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

printf("%d \t\t",p[i].pos);

printf("%d \t\t",p[i].AT);

printf("%d \t",p[i].BT);

j=0;

int v=0;

while(s[i][j]!=-1){

printf("%d ",p[i].ST[j]);

j++;

v+=3;

}

while(v!=40){

printf(" ");

v+=1;

}

printf("%d \t\t",p[i].FT);

printf("%d \t\t",p[i].WT);

printf("%d \n",p[i].TAT);

}

double avg\_wt,avg\_tat;

avg\_wt=tot\_wt/(float)n;

avg\_tat=tot\_tat/(float)n;

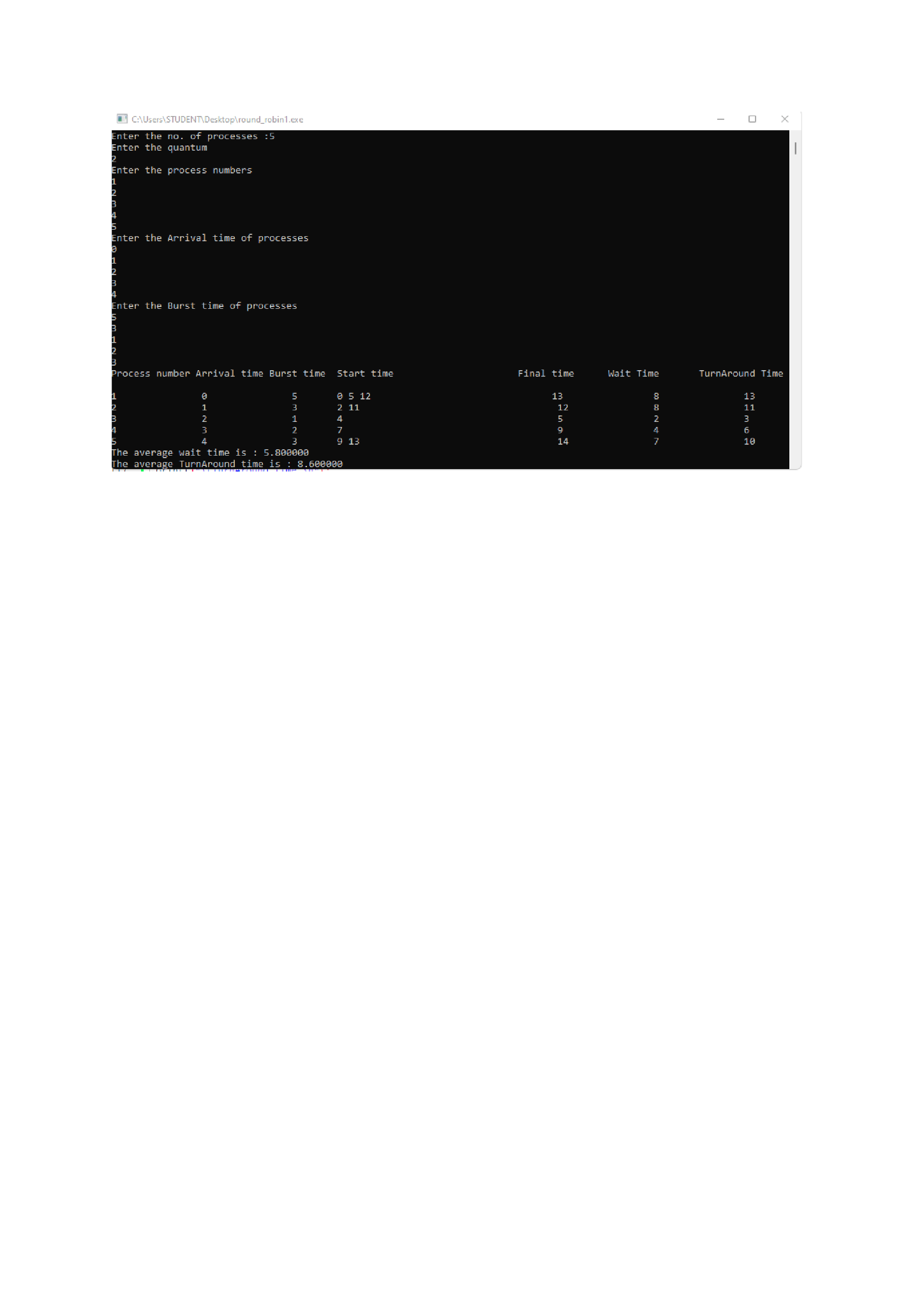
printf("The average wait time is : %lf\n",avg\_wt);

printf("The average TurnAround time is : %lf\n",avg\_tat);

return 0;

}

**SAMPLE OUTPUT**

****

**3.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.**

|  |  |  |  |
| --- | --- | --- | --- |
| Process | Arrival Time | Burst Time | System(0)/User(1) |
| P1 | 0 | 3 | 0 |
| P2 | 2 | 2 | 0 |
| P3 | 4 | 4 | 1 |
| P4 | 4 | 2 | 1 |
| P5 | 8 | 2 | 0 |
| P6 | 10 | 3 | 1 |

#include <stdio.h>

#include<stdlib.h>

#include <stdbool.h>

#define MAX\_QUEUE\_SIZE 100

int totalTime=0;

int userProcess=0,systemProcess=0;

// Structure to represent a process

typedef struct {

int processID;

int arrivalTime;

int burstTime;

int remainingTime;

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

} Process;

// Function to execute a process

void executeProcess(Process process) {

int i;

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

// Simulating the execution time of the process

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

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

}

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

}

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

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

int i,j;

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

{

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

{

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

{

Process temp=system[i];

system[i]=system[j];

system[j]=temp;

}

}

}

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

{

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

{

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

{

Process temp=user[i];

user[i]=user[j];

user[j]=temp;

}

}

}

int completed=0;

int currentProcess=-1;

bool isUserProcess=false;

int size=userProcess+systemProcess;

while(1)

{

int count=0;

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

{

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

{

count++;

}

}

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

{

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

{

count++;

}

}

if(count==size)

{

printf("\n end of processess");

exit(0);

}

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

{

if(totalTime>=system[i].arrivalTime && system[i].remainingTime>0)

{

currentProcess=i;

isUserProcess=false;

break;

}

}

if(currentProcess==-1)

{

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

{

if(totalTime>=user[j].arrivalTime && user[j].remainingTime>0)

{

currentProcess=j;

isUserProcess=true;

break;

}

}

}

if(currentProcess==-1)

{

totalTime++;

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

if(totalTime==1000)

{

exit(0);

}

continue;

}

if(isUserProcess==true)

{

user[currentProcess].remainingTime--;

printf("\n User process %d will excecute at %d ",user[currentProcess].processID,(totalTime));

totalTime++;

isUserProcess=false;

currentProcess=-1;

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

{

completed++;

}

}else{

int temp=totalTime;

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

totalTime++;

}

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

{

completed++;

}

printf("\n System process %d will excecute from %d to %d ",system[currentProcess].processID,temp,(totalTime));

isUserProcess=false;

currentProcess=-1;

}

}

}

int main() {

int numProcesses,i;

Process processes[MAX\_QUEUE\_SIZE];

// Reading the number of processes

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

scanf("%d", &numProcesses);

// Reading process details

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

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

printf("Arrival Time: ");

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

printf("Burst Time: ");

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

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

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

processes[i].processID = i + 1;

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

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

{

userProcess++;

}else{

systemProcess++;

}

}

Process systemQueue[MAX\_QUEUE\_SIZE];

int systemQueueSize = 0;

Process userQueue[MAX\_QUEUE\_SIZE];

int userQueueSize = 0;

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

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

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

} else {

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

}

}

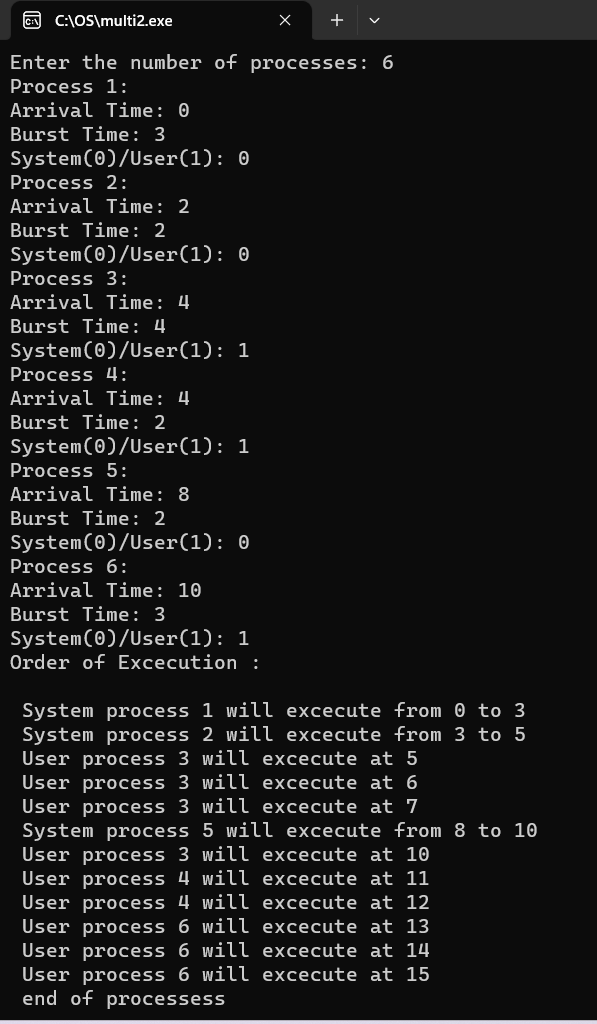
printf("Order of Excecution :\n");

scheduleFCFS(systemQueue,userQueue);

return 0;

}

**SAMPLE OUTPUT**



**4.a.Simulate Rate Monotonic Scheduling for the following and show the order of execution of processes in CPU timeline:**



#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <stdbool.h>

#define MAX\_PROCESS 10

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

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

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

// collecting details of processes

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

{

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

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

if (num\_of\_process < 1)

{

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

exit(0);

}

if (selected\_algo == 2)

{

printf("\nEnter Time Quantum: ");

scanf("%d", &time\_quantum);

if (time\_quantum < 1)

{

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

exit(0);

}

}

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

{

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

if (selected\_algo == 1)

{

printf("==> Burst time: ");

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

}

else if (selected\_algo == 2)

{

printf("=> Arrival Time: ");

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

printf("=> Burst Time: ");

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

remain\_time[i] = burst\_time[i];

}

else if (selected\_algo > 2)

{

printf("==> Execution time: ");

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

remain\_time[i] = execution\_time[i];

if (selected\_algo == 4)

{

printf("==> Deadline: ");

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

}

else

{

printf("==> Period: ");

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

}

}

}

}

// get maximum of three numbers

int max(int a, int b, int c)

{

int max;

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

max = a;

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

max = b;

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

max = c;

return max;

}

// calculating the observation time for scheduling timeline

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

{

if (selected\_algo < 3)

{

int sum = 0;

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

{

sum += burst\_time[i];

}

return sum;

}

else if (selected\_algo == 3)

{

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

}

else if (selected\_algo == 4)

{

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

}

}

// print scheduling sequence

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

{

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

printf("Time: ");

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

{

if (i < 10)

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

else

printf("| %d ", i);

}

printf("|\n");

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

{

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

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

{

if (process\_list[j] == i + 1)

printf("|####");

else

printf("| ");

}

printf("|\n");

}

}

void rate\_monotonic(int time)

{

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

float utilization = 0;

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

{

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

}

int n = num\_of\_process;

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

{

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

exit(0);

}

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

{

min = 1000;

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

{

if (remain\_time[j] > 0)

{

if (min > period[j])

{

min = period[j];

next\_process = j;

}

}

}

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

{

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

remain\_time[next\_process] -= 1;

}

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

{

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

{

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

next\_process = k;

}

}

}

print\_schedule(process\_list, time);

}

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

{

int option = 0;

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

printf("Select > ");

scanf("%d", &option);

printf("-----------------------------\n");

get\_process\_info(option); // collecting processes detail

int observation\_time = get\_observation\_time(option);

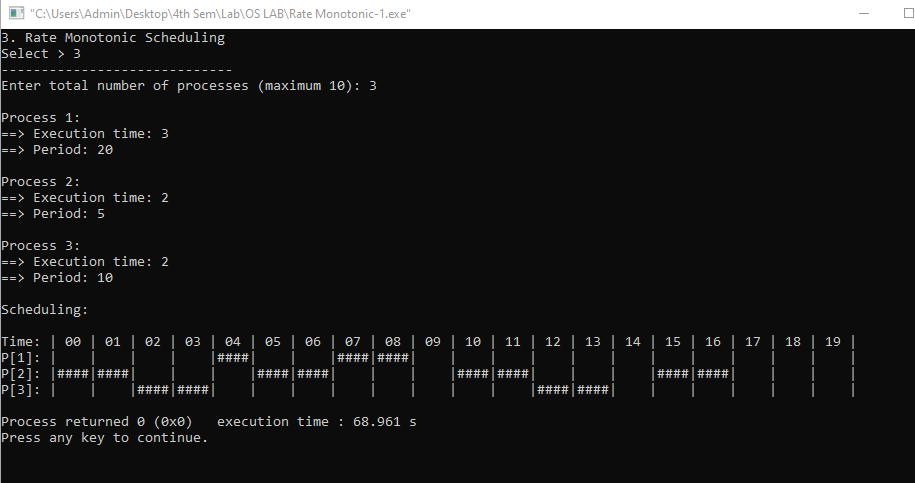
if (option == 3)

rate\_monotonic(observation\_time);

return 0;

}

**SAMPLE OUTPUT**



**4.b.Simulate Earliest Deadline First for the following and show the order of execution of processes in CPU timeline:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Process** | **Execution Time** | **Deadline** | **Period** |
| P1 | 3 | 7 | 20 |
| P2 | 2 | 4 | 5 |
| P3 | 2 | 8 | 10 |

#include <stdio.h>

#define arrival 0

#define execution 1

#define deadline 2

#define period 3

#define abs\_arrival 4

#define execution\_copy 5

#define abs\_deadline 6

typedef struct

{

int T[7],instance,alive;

}task;

#define IDLE\_TASK\_ID 1023

#define ALL 1

#define CURRENT 0

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

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

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

int gcd(int a, int b);

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

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

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

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

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

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

int timer = 0;

int main()

{

task \*t;

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

float cpu\_utilization;

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

scanf("%d", &n);

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

get\_tasks(t, n);

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

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

if (cpu\_utilization < 1)

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

else

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

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

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

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

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

while (timer <= hyper\_period)

{

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

{

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

}

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

{

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

}

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

{

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

{

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

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

}

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

{

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

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

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

update\_abs\_arrival(t, active\_task\_id, t[active\_task\_id].instance, CURRENT);

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

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

}

}

++timer;

}

free(t);

return 0;

}

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

{

int i = 0;

while (i < n)

{

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

printf("Arrival time: ");

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

printf("Execution time: ");

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

printf("Deadline time: ");

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

printf("Period: ");

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

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

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

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

t1->instance = 0;

t1->alive = 0;

t1++;

i++;

}

}

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

{

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

while (i < n)

{

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

t1++;

i++;

}

ht = lcm(a, n);

return ht;

}

int gcd(int a, int b)

{

if (b == 0)

return a;

else

return gcd(b, a % b);

}

int lcm(int \*a, int n)

{

int res = 1, i;

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

{

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

}

return res;

}

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

{

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

task \*t1\_copy;

t1\_copy = t1;

while (i < n)

{

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

{

t1->alive = 1;

a++;

}

t1++;

i++;

}

t1 = t1\_copy;

i = 0;

while (i < n)

{

if (t1->alive == 0)

n1++;

t1++;

i++;

}

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

{

return 1;

}

return 0;

}

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

{

int i = 0;

if (all)

{

while (i < n)

{

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

t1++;

i++;

}

}

else

{

t1 += n;

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

}

}

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

{

int i = 0;

if (all)

{

while (i < n)

{

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

t1++;

i++;

}

}

else

{

t1 += n;

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

}

}

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

{

int i = 0;

if (all)

{

while (i < n)

{

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

t1++;

i++;

}

}

else

{

t1 += n;

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

}

}

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

{

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

while (i < n)

{

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

{

min = t1->T[p];

task\_id = i;

}

t1++;

i++;

}

return task\_id;

}

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

{

int i = 0;

float cu = 0;

while (i < n)

{

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

t1++;

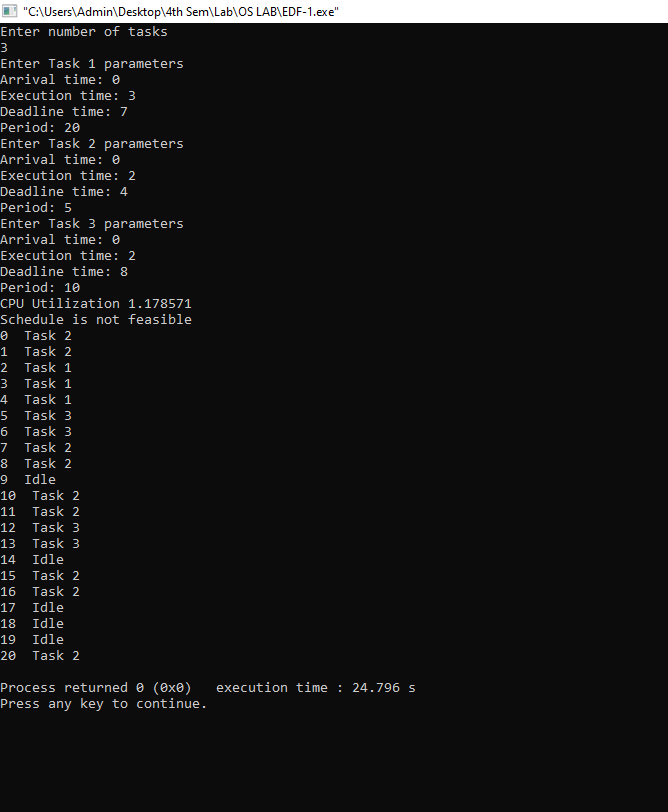
i++;

}

return cu;

}

**SAMPLE OUTPUT**



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

**semaphores.**

#include<stdio.h>

#include<conio.h>

int mutex=1;

int full=0;

int empty=10;

int cnt=0;

int wait(int s)

{

while(s<=0);

s--;

return s;

}

int signal(int s)

{

s++;

return s;

}

void producer()

{

empty=wait(empty);

mutex=wait(mutex);

cnt++;

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

mutex=signal(mutex);

full=signal(full);

}

void consumer()

{

full=wait(full);

mutex=wait(mutex);

printf("Consumer consumes an item %d\n",cnt);

cnt--;

mutex=signal(mutex);

empty=signal(empty);

}

void main()

{

int choice;

printf("1.Produce\n2.Consume\n3.Exit\n");

while(1)

{

printf("Enter your choice:\n");

scanf("%d",&choice);

switch(choice)

{

case 1:if(empty==0)

{

printf("Buffer is full\n");

}

else{

producer();

}

break;

case 2:if(full==0)

{

printf("Buffer is empty\n");

}

else{

consumer();

}

break;

case 3:exit(0);

break;

default:printf("Invalid choice\n");

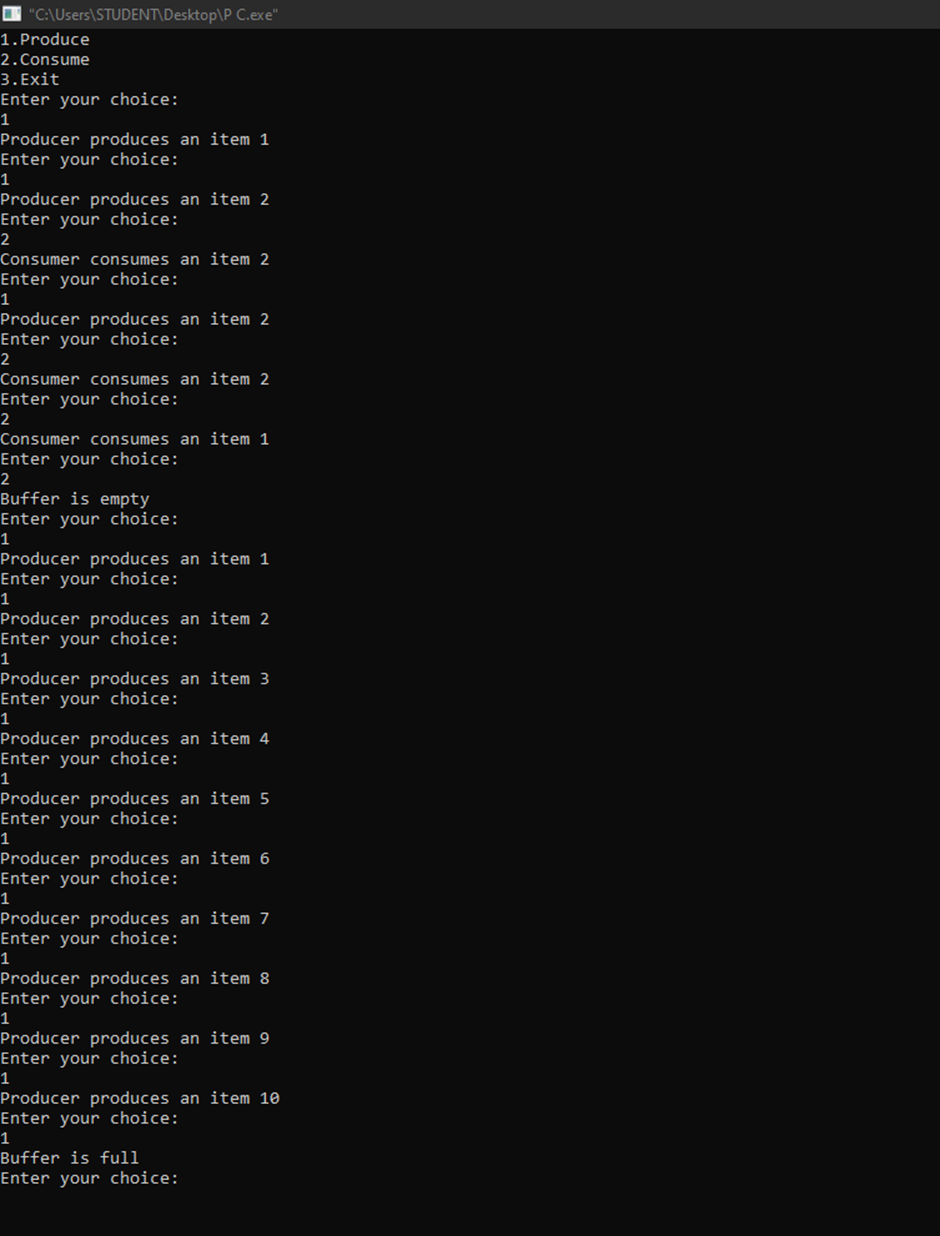
}

}

getch();

}

**SAMPLE OUTPUT**



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

**problem.**

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#define N 5

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (phnum + 4) % N

#define RIGHT (phnum + 1) % N

int state[N];

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

sem\_t mutex;

sem\_t S[N];

void test(int phnum)

{

if (state[phnum] == HUNGRY

&& state[LEFT] != EATING

&& state[RIGHT] != EATING) {

// state that eating

state[phnum] = EATING;

sleep(2);

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

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

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

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

// during takefork

// used to wake up hungry philosophers

// during putfork

sem\_post(&S[phnum]);

}

}

// take up chopsticks

void take\_fork(int phnum)

{

sem\_wait(&mutex);

// state that hungry

state[phnum] = HUNGRY;

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

// eat if neighbours are not eating

test(phnum);

sem\_post(&mutex);

// if unable to eat wait to be signalled

sem\_wait(&S[phnum]);

sleep(1);

}

// put down chopsticks

void put\_fork(int phnum)

{

sem\_wait(&mutex);

// state that thinking

state[phnum] = THINKING;

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

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

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

test(LEFT);

test(RIGHT);

sem\_post(&mutex);

}

void\* philosopher(void\* num)

{

while (1) {

int\* i = num;

sleep(1);

take\_fork(\*i);

sleep(0);

put\_fork(\*i);

}

}

int main()

{

int i;

pthread\_t thread\_id[N];

// initialize the semaphores

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

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

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

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

// create philosopher processes

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

philosopher, &phil[i]);

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

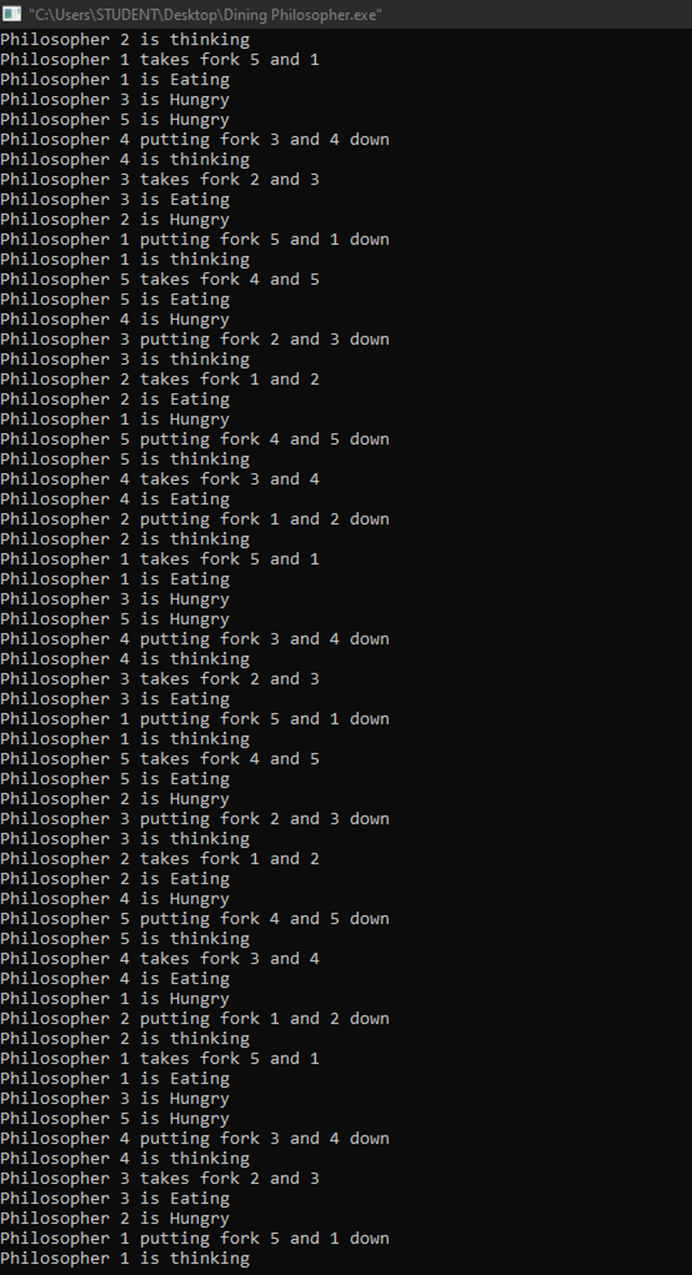
}

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

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

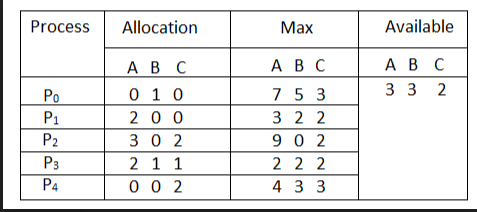
}

**SAMPLE OUTPUT**



**7.BANKERS ALGORITHM**

**Use bankers algorithm to check if the following state is safe/unsafe:**



Is the system in a safe state? If Yes, then what is the safe sequence? What will happen if process P1 requests one additional instance of resource type A and two instances of resource type C?

#include <stdio.h>

int n, m, i, j, k,alloc[10][10],max[10][10],avail[10],ch,t,add[10];

void main()

{

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

scanf("%d",&n);

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

scanf("%d",&m);

printf("\nEnter the allocation array");

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

{

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

{

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

}

}

printf("\nEnter the maximum available array");

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

{

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

{

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

}

}

printf("\nEnter the total available number of resources:");

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

{

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

}

printf("Is there any request from the process, if yes (1),no (0)");

scanf("%d",&ch);

if(ch==1)

{

printf("Enter the process number for which there is an additional request");

scanf("%d",&t);

printf("Enter the number of instances required for each resource");

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

{

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

}

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

{

alloc[t][i]+=add[i];

}

if(max[t][0]<alloc[t][0]||max[t][1]<alloc[t][1]||max[t][2]<alloc[t][2])

printf("It is not a valid request");

else

{

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

{

avail[i]-=add[i];

}

bankers();

}

}

else

bankers();

}

void bankers()

{

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] - alloc[i][j];

}

int y = 0;

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

{

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

{

if (f[i] == 0)

{

int flag = 0;

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

{

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

{

flag = 1;

break;

}

}

if (flag == 0)

{

ans[ind++] = i;

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

avail[y] += alloc[i][y];

f[i] = 1;

break;

}

}

}

}

int flag = 1;

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

{

if (f[i] == 0)

{

flag = 0;

printf("The following system is not safe");

break;

}

}

if (flag == 1)

{

printf("Following is the SAFE Sequence\n");

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

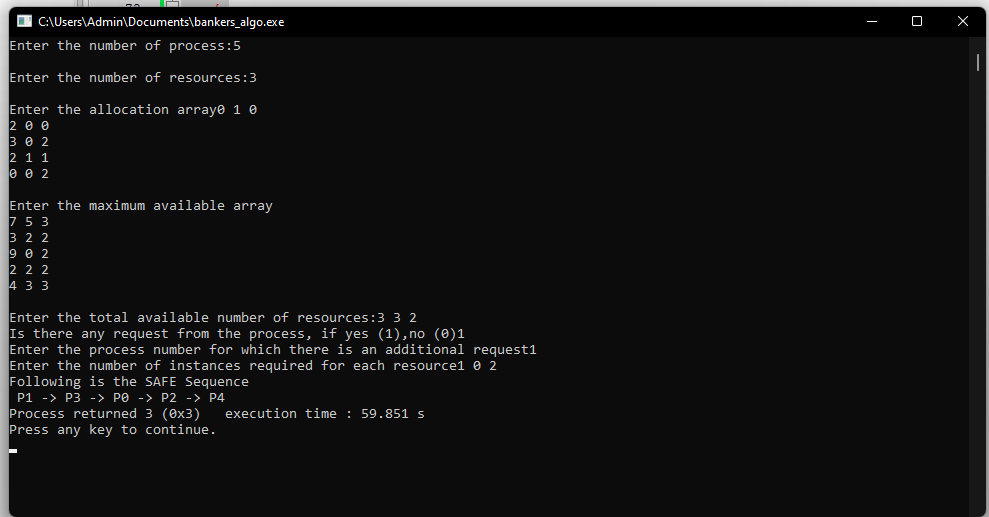
printf(" P%d ->", ans[i]);

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

}

}

**SAMPLE OUTPUT**



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

#include<stdio.h>

static int mark[20];

int i, j, np, nr,k;

int main()

{

int alloc[10][10],request[10][10],avail[10],r[10],w[10];

printf ("\nEnter the no of the 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]);

/\*Available Resource calculation\*/

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

{

avail[j]=r[j];

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

{

avail[j]-=alloc[i][j];

}

}

//marking processes with zero allocation

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;

}

// initialize W with avail

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

w[j]=avail[j];

//mark processes with request less than or equal to W

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

{

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];

break;

}

}

}

}

//checking for unmarked processes

int deadlock=0;

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

{

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

if(mark[i]!=1)

deadlock=1;

}

if(deadlock==1)

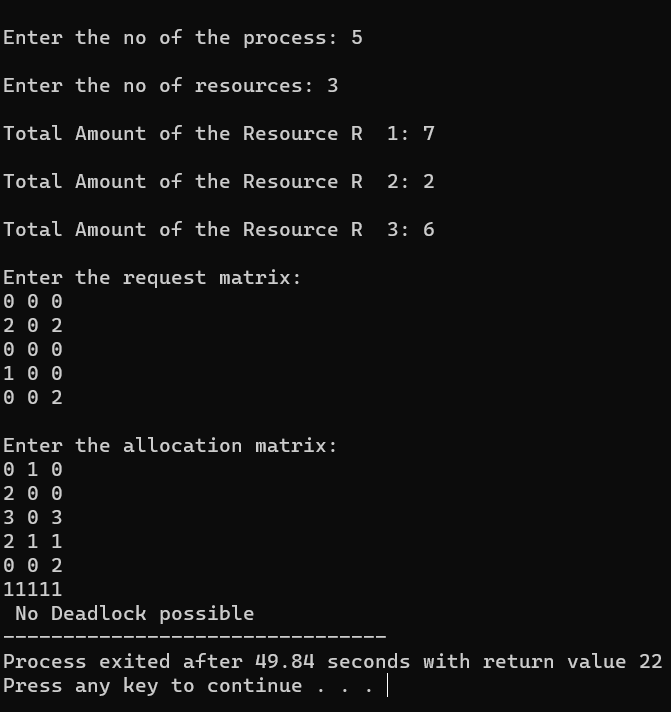
printf("\n Deadlock detected");

else

printf("\n No Deadlock possible");

}

**SAMPLE OUTPUT**



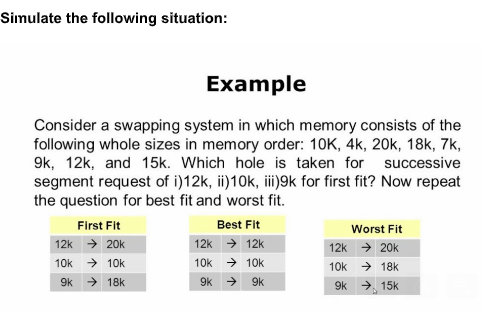
**9.Write a C program to simulate the following contiguous memory**

**allocation techniques**

a) Worst-fit

b) Best-fit

c) First-fit



#include <stdio.h>

#include<stdlib.h>

#define max 25

void readInput(int \*nb, int \*nf, int b[], int f[]);

void bestFit(int nb, int nf, int b[], int f[], int bf[], int ff[], int frag[]);

void worstFit(int nb, int nf, int b[], int f[], int bf[], int ff[], int frag[]);

void firstFit(int nb, int nf, int b[], int f[], int bf[], int ff[], int frag[]);

void displayResults(int nf, int f[], int ff[], int b[], int frag[]);

int main()

{

int nb, nf, ch;

int b[max], f[max], bf[max] ={0}, ff[max]={0}, frag[max]={0};

readInput(&nb, &nf, b, f);

printf("1.Best Fit 2.Worst Fit 3.First Fit 4. Exit\n");

scanf("%d",&ch);

switch(ch)

{

case 1: bestFit(nb, nf, b, f, bf, ff, frag);

break;

case 2: worstFit(nb, nf, b, f, bf, ff, frag);

break;

case 3: firstFit(nb, nf, b, f, bf, ff, frag);

break;

case 4: exit(0);

break;

default: printf("Inavlid choice\n");

break;

}

displayResults(nf, f, ff, b, frag);

return 0;

}

void readInput(int \*nb, int \*nf, int b[], int f[])

{

int i;

printf("Enter 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 (i = 1; i <= \*nb; i++)

{

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

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

}

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

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

{

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

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

}

}

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

{

int i, j, temp, lowest = 999;

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

{

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

{

if (bf[j] != 1) //if bf[j] is not allocated

{

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

if (temp >= 0)

{

if (lowest>temp)

{

ff[i] = j;

lowest = temp;

}

}

}

}

frag[i] = lowest;

bf[ff[i]] = 1;

lowest = 999;

}

}

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

{

int 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)

{

if (lowest == 10000 || temp > lowest)

{

ff[i] = j;

lowest = temp;

}

}

}

}

frag[i] = lowest;

bf[ff[i]] = 1;

lowest = 10000;

}

}

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

{

int i, j, temp;

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

{

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

{

if (bf[j] != 1)

{

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

if (temp >= 0)

{

ff[i] = j;

break;

}

}

}

frag[i] = temp;

bf[ff[i]] = 1;

}

}

void displayResults(int nf, int f[], int ff[], int b[], int frag[])

{

int i;

printf("\nFile\_no\t\tFile\_size\tBlock\_no\tBlock\_size\tFragment");

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

{

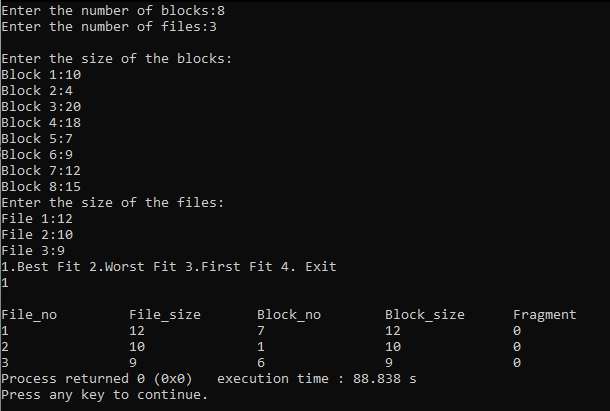
printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);

}

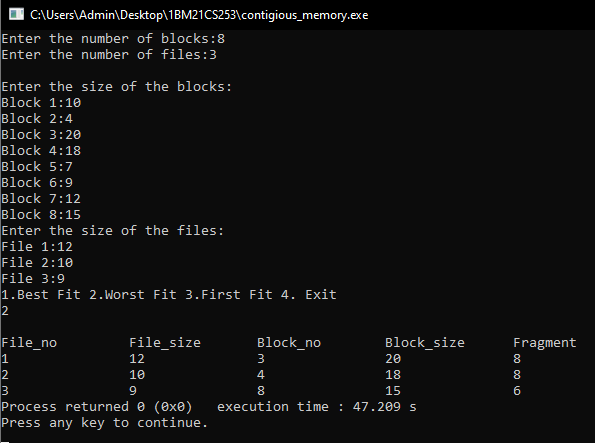
}

SAMPLE OUTPUT:

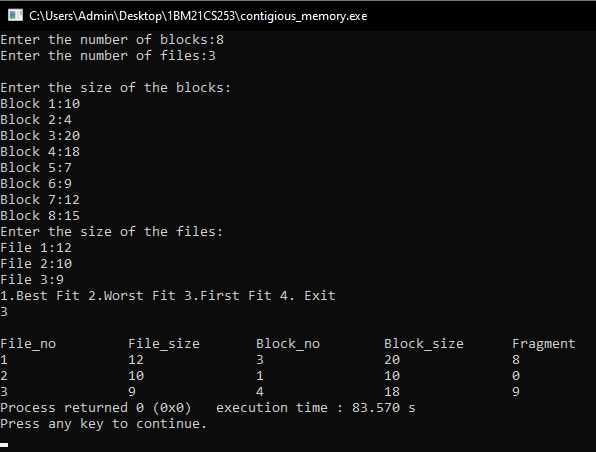
FOR BEST FIT



FOR WORST FIT



FOR FIRST FIT



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

**a) FCFS**

**b) SCAN**

**c) C-SCAN**

#include<stdio.h>

#include<conio.h>

int head,a[20],range,n;

void fcfs()

{

int headm=0,temp,i;

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

{

if(i==0)

{

if(a[i]<head)

headm=headm+(head-a[i]);

else

headm=headm+(a[i]-head);

}

else

{

if(a[i-1]<a[i])

headm=headm+(a[i]-a[i-1]);

else

headm=headm+(a[i-1]-a[i]);

}

}

printf("\nFCFS-Total head movement=%d\n",headm);

}

void scan()

{

int headm=0,i,dir,temp,cnt=0;

printf("\nEnter the direction, upward/right=1, downward/left=-1:");

scanf("%d",&dir);

if(dir==1)

{

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

{

if(a[i]<head)

{

cnt++;

continue;

}

else if(i==cnt)

headm=headm+(a[i]-head);

else

headm=headm+(a[i]-a[i-1]);

}

headm=headm+(range-a[i-1]);

headm+=(range-a[cnt-1]);

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

{

headm+=(a[i]-a[i-1]);

}

}

else

{

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

{

if(a[i]>head)

break;

else

cnt++;

}

headm+=(head-a[cnt-1]);

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

{

headm+=(a[i]-a[i-1]);

}

headm+=(a[0]-0);

headm+=(a[cnt]-0);

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

{

headm+=(a[i+1]-a[i]);

}

}

printf("\nSCAN-Total head movement=%d\n",headm);

}

void cscan()

{

int headm=0,i,dir,temp,cnt=0;

printf("\nEnter the direction, upward/right=1, downward/left=-1:");

scanf("%d",&dir);

if(dir==1)

{

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

{

if(a[i]<head)

{

cnt++;

continue;

}

else if(i==cnt)

headm=headm+(a[i]-head);

else

headm=headm+(a[i]-a[i-1]);

}

headm=headm+(range-a[i-1]);

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

{

headm+=(a[i]-a[i-1]);

}

headm+=(a[i]-0);

}

else

{

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

{

if(a[i]>head)

break;

else

cnt++;

}

headm+=(head-a[cnt-1]);

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

{

headm+=(a[i]-a[i-1]);

}

headm+=(a[0]-0);

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

{

headm+=(a[i+1]-a[i]);

}

headm=headm+(range-a[i]);

}

printf("\nCSCAN-Total head movement=%d\n",headm);

}

void main()

{

int i,j,temp;

printf("\nEnter the total range of cylinders:");

scanf("%d",&range);

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

scanf("%d",&n);

printf("\nEnter the cylinder numbers:");

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

{

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

}

printf("\nEnter the header:");

scanf("%d",&head);

fcfs();

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

{

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

{

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

{

temp=a[j];

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

a[j+1]=temp;

}

}

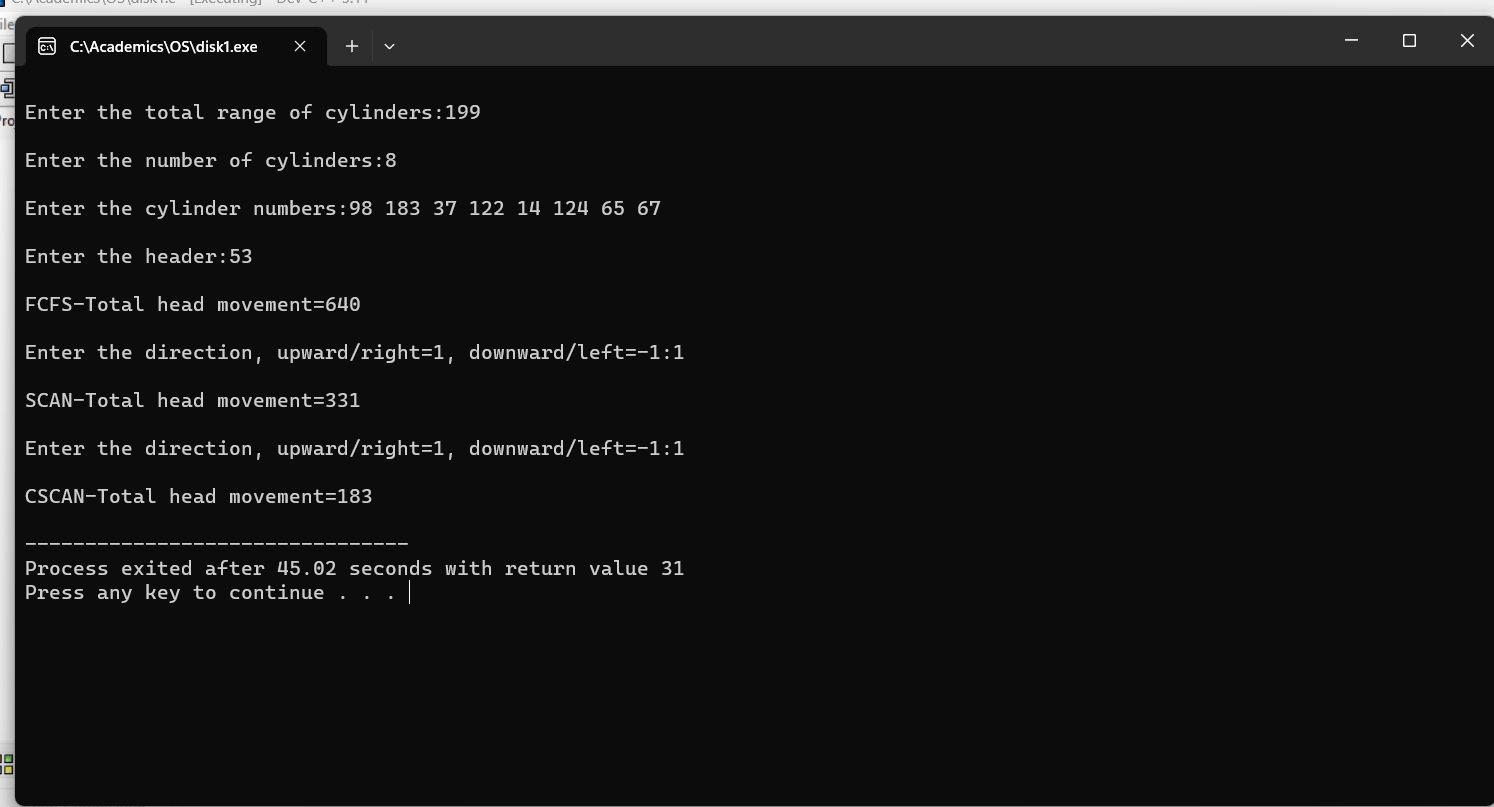
}

scan();

cscan();

}

**SAMPLE OUTPUT**



**11.Write a C program to simulate disk scheduling algorithms**  
  
**a)    SSTF  
b)  LOOK  
c)  c-LOOK**

#include<stdio.h>

#include<conio.h>

int head,a[20],range,n;

void sstf()

{

int c=0,i,j,headm=0,k,t,temp,b[20];

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

{

b[i]=a[i];

}

b[n]=head;

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

{

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

{

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

{

temp=b[j];

b[j]=b[j+1];

b[j+1]=temp;

}

}

}

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

{

if(b[i]==head)

break;

else

c++;

}

j=c;

k=c;

t=j;

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

{

if((b[k+1]-b[t])<(b[t]-b[j-1]) && j>0)

{

headm+=(b[k+1]-b[t]);

k++;

t=k;

}

else if(j==0)

{

headm+=(b[k+1]-b[t]);

k++;

t=k;

}

else

{

headm+=(b[t]-b[j-1]);

j--;

t=j;

}

}

printf("SSTF-Total head movement=%d\n",headm);

}

void look()

{

int headm=0,i,dir,temp,cnt=0;

printf("Enter the direction, upward/right=1, downward/left=-1:\n");

scanf("%d",&dir);

if(dir==1)

{

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

{

if(a[i]<head)

{

cnt++;

continue;

}

else if(i==cnt)

headm=headm+(a[i]-head);

else

headm=headm+(a[i]-a[i-1]);

}

headm+=a[n-1]-a[cnt-1];

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

{

headm+=(a[i]-a[i-1]);

}

}

else

{

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

{

if(a[i]>head)

break;

else

cnt++;

}

headm+=(head-a[cnt-1]);

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

{

headm+=(a[i]-a[i-1]);

}

headm+=(a[cnt]-a[0]);

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

{

headm+=(a[i+1]-a[i]);

}

}

printf("LOOK-Total head movement=%d\n",headm);

}

void clook()

{

int headm=0,i,dir,temp,cnt=0;

printf("Enter the direction, upward/right=1, downward/left=-1:\n");

scanf("%d",&dir);

if(dir==1)

{

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

{

if(a[i]<head)

{

cnt++;

continue;

}

else if(i==cnt)

headm=headm+(a[i]-head);

else

headm=headm+(a[i]-a[i-1]);

}

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

{

headm+=(a[i]-a[i-1]);

}

}

else

{

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

{

if(a[i]>head)

break;

else

cnt++;

}

headm+=(head-a[cnt-1]);

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

{

headm+=(a[i]-a[i-1]);

}

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

{

headm+=(a[i+1]-a[i]);

}

}

printf("\nCLOOK-Total head movement=%d\n",headm);

}

void main()

{

int i,j,temp;

printf("\nEnter the total range of cylinders:");

scanf("%d",&range);

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

scanf("%d",&n);

printf("\nEnter the header:");

scanf("%d",&head);

printf("\nEnter the cylinder numbers:");

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

{

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

}

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

{

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

{

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

{

temp=a[j];

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

a[j+1]=temp;

}

}

}

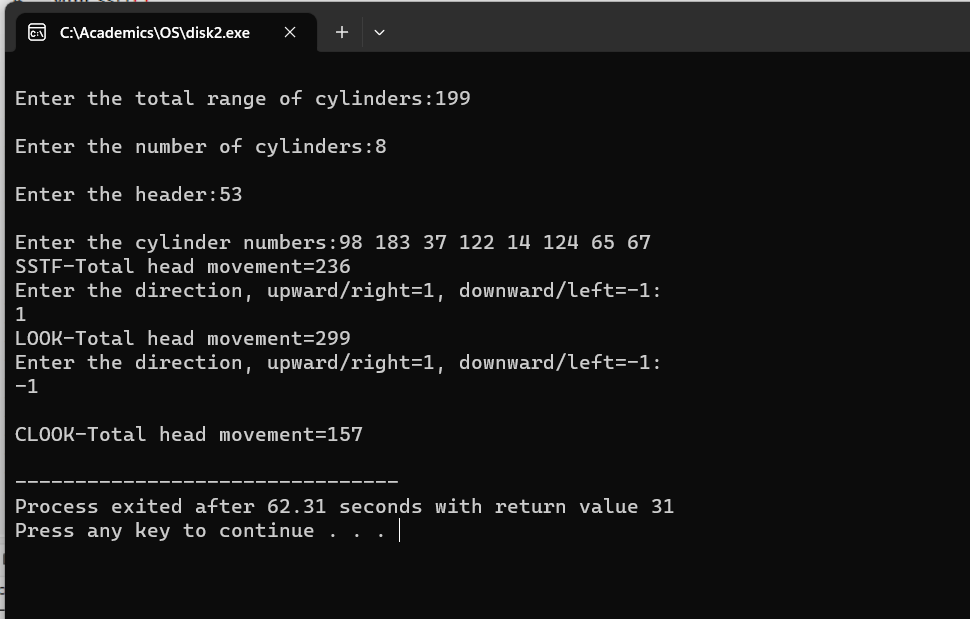
sstf();

look();

clook();

}

**SAMPLE OUTPUT**



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

#include<stdio.h>

#include<conio.h>

int n,m,a[20],p[10];

void fifo()

{

int i,j,flag,cnt=0,k=0;

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

{

flag=1;

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

{

if(a[i]==p[j])

{

flag=0;

break;

}

}

if(flag==1)

{

cnt++;

p[k]=a[i];

k=(k+1)%m;

}

}

printf("\nFIFO-Page faults=%d",cnt);

}

void optimal()

{

int i,j,flag,cnt=0,k=0,t,temp,f,help[10],ct;

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

{

flag=1,f=1,ct=0;

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

{

help[j]=0;

if(a[i]==p[j])

{

flag=0;

break;

}

}

if(flag==1)

{

cnt++;

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

{

if(p[j]==-1)

{

p[j]=a[i];

f=0;

break;

}

}

if(f==1)

{

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

{

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

{

if(p[j]==a[k]&&help[j]==0)

{

temp=j;

help[j]=1;

}

}

}

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

{

if(help[j]==0)

temp=j;

}

p[temp]=a[i];

}

}

}

printf("\nOPTIMAL-Page faults=%d",cnt);

}

void lru()

{

int flag,f,k,cnt=0,i,j,temp,ct,help[10];

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

{

flag=1,f=1,ct=0;

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

{

help[j]=0;

if(p[j]==a[i])

{

flag=0;

break;

}

}

if(flag==1)

{

cnt++;

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

{

if(p[j]==-1)

{

p[j]=a[i];

{

f=0;

break;

}

}

}

if(f==1)

{

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

{

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

{

if(p[j]==a[k]&& help[j]==0)

{

temp=j;

help[j]=1;

}

}

}

p[temp]=a[i];

}

}

}

printf("\nLRU-Page faults=%d",cnt);

}

void main()

{

int i;

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

scanf("%d",&n);

printf("\nEnter the page numbers:");

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

{

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

}

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

scanf("%d",&m);

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

p[i]=-1;

fifo();

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

p[i]=-1;

optimal();

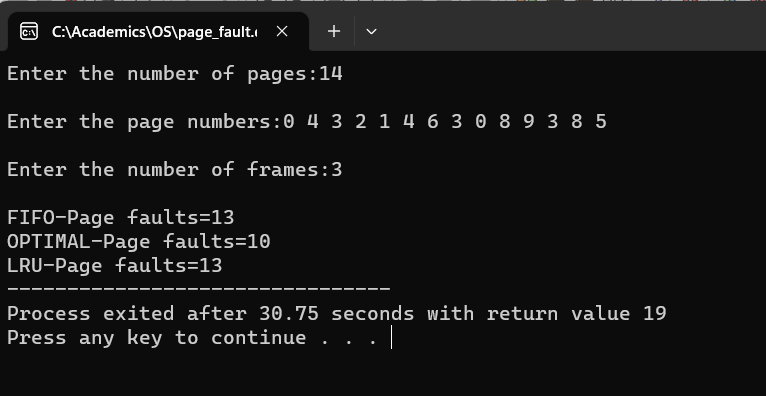
for(i=0;i<m;i++)

p[i]=-1;

lru();

}

**SAMPLE OUTPUT**



**13.Write a C program to simulate paging technique of memory management. (create a logical memory space, physical memory space and page table, you should show the address translation entirely)**

#include<stdio.h>

#include<conio.h>

main()

{

int ms, ps, nop, np, rempages, i, j, x, y, pa, offset;

int s[10], fno[10][20];

printf("\nEnter the memory size -- ");

scanf("%d",&ms);

printf("\nEnter the page size -- ");

scanf("%d",&ps);

nop = ms/ps;

printf("\nThe no. of pages available in memory are -- %d ",nop);

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

scanf("%d",&np);

rempages = nop;

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

{

printf("\nEnter no. of pages required for p[%d]-- ",i);

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

if(s[i] >rempages)

{

printf("\nMemory is Full");

break;

}

rempages = rempages - s[i];

printf("\nEnter pagetable for p[%d] --- ",i);

for(j=0;j<s[i];j++)

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

}

printf("\nEnter Logical Address to find Physical Address ");

printf("\nEnter process no. and pagenumber and offset -- ");

scanf("%d %d %d",&x,&y, &offset);

if(x>np || y>=s[i] || offset>=ps)

printf("\nInvalid Process or Page Number or offset");

else

{ pa=fno[x][y]\*ps+offset;

printf("\nThe Physical Address is -- %d",pa);

}

getch();

}

OUTPUT

