**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

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

****

**LAB REPORT**

**on**

**OPERATING SYSTEMS**

***Submitted by***

**DHIKSHA RATHIS (1BM21CS055)**

***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

**DHIKSHA RATHIS (1BM21CS055),** who is a 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.

Dr. Nandini Vineeth         Dr. Jyothi S Nayak

Assistant 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 (preemptive & Non-pre-emptive) | 5-13 |
| 2 | Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.   * Priority (preemptive & Non-pre-emptive) * Round Robin (Experiment with different quantum sizes for RR algorithm) | 14-25 |
| 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. | 26-30 |
| 4 | Write a C program to simulate Real-Time CPU Scheduling algorithms:   * Rate- Monotonic * Earliest-deadline First | 31-43 |
| 5 | Write a C program to simulate producer-consumer problem using semaphores. | 44-46 |
| 6 | Write a C program to simulate the concept of Dining-Philosophers problem. | 47-51 |
| 7 | Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance. | 52-58 |
| 8 | Write a C program to simulate deadlock detection. | 59-63 |
| 9 | Write a C program to simulate the following contiguous memory allocation techniques   * Worst-fit * Best-fit * First-fit | 64-71 |
| 10 | Write a C program to simulate page replacement algorithms   * FIFO * LRU * Optimal | 72-81 |
| 11 | Write a C program to simulate disk scheduling algorithms   * FCFS * SCAN * C-SCAN | 82-90 |

**Course Outcome**

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

**LAB 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 (preemptive & Non-pre-emptive)

**FCFS:**

#include<stdio.h>

int main()

{

int bt[20], at[20], wt[20], tat[20],ct[20], i, n;

float wtavg, tatavg;

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

scanf("%d", &n);

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

{

printf("\nEnter Arrival Time and Burst Time for Process %d: ", i);

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

}

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

{

ct[i]=bt[i]+ct[i-1];

}

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

{

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

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

tatavg=tatavg+tat[i];

wtavg=wtavg+wt[i];

}

printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");

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

{

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

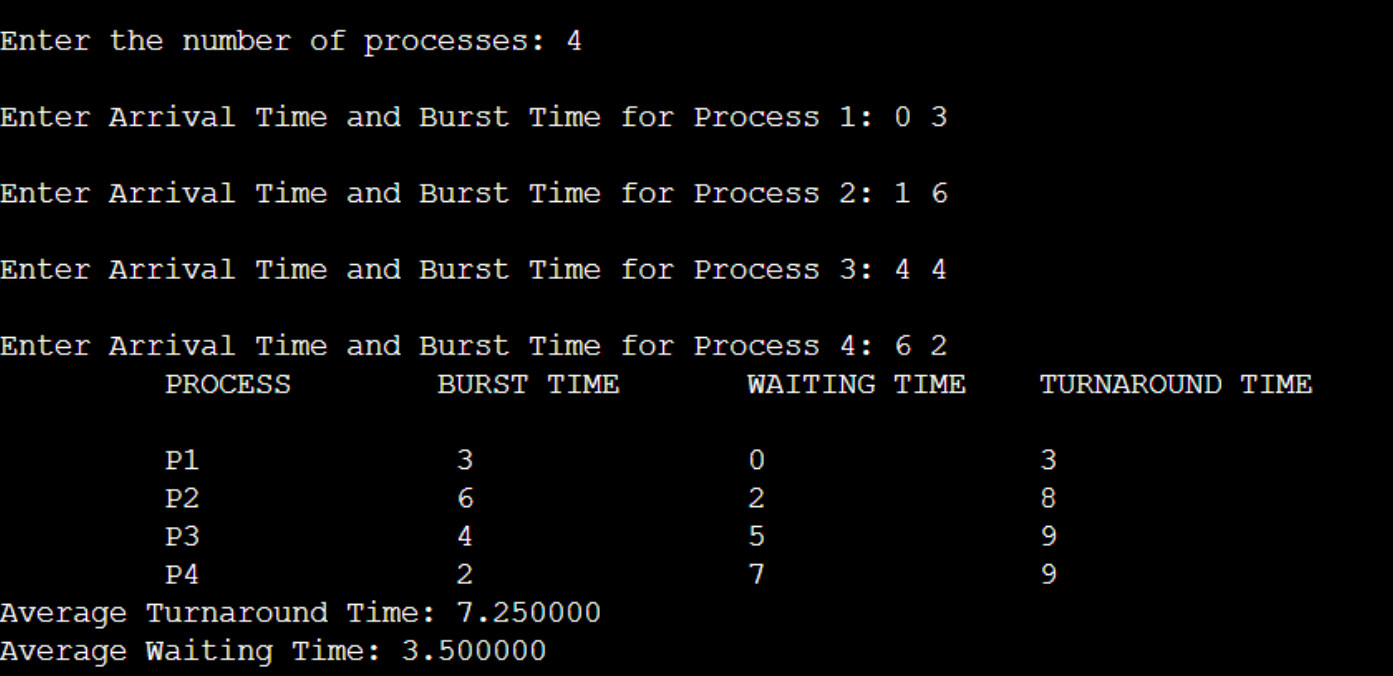
}

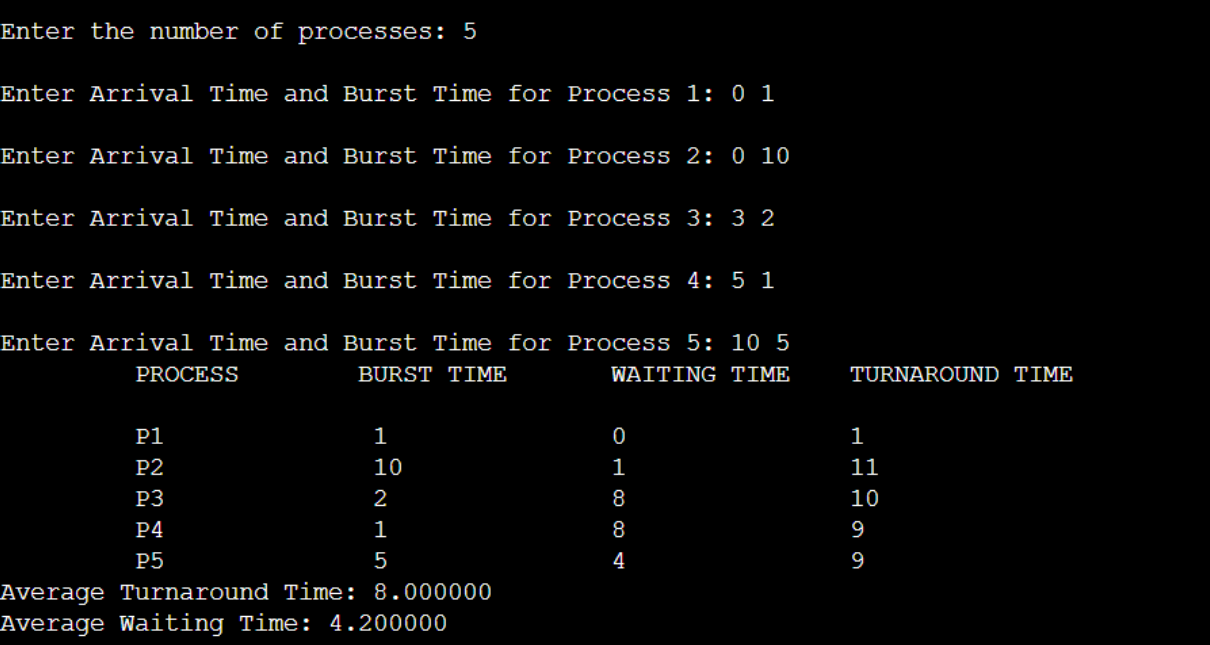
printf("\nAverage Turnaround Time: %f", tatavg/n);

printf("\nAverage Waiting Time: %f", wtavg/n);

}

**OUTPUT:**

****

****

**SJF (Non-Pre-emptive):**

#include<stdio.h>

int at[20], cput[20];

void sjf(int n)

{

int cmpt[20],tat[20],wt[20],cput1[20];

float awt=0, atat=0,sum\_burst\_time=0;

int sum=0,i,j, smallest;

printf("\t PROCESS \t TURNAROUND TIME\t WAITING TIME\n");

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

{

cput1[i]=cput[i];

sum\_burst\_time += cput[i];

}

cput1[9]=9999;

while(sum < sum\_burst\_time)

{

smallest = 9;

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

{

if (at[i] <= sum && cput1[i] > 0 && cput1[i] < cput1[smallest])

smallest = i;

}

printf("\t P[%d] \t\t %d \t\t %d\n", smallest , sum + cput1[smallest]-

at[smallest], sum - at[smallest]);

awt += sum + cput1[smallest] - at[smallest];

atat += sum - at[smallest];

sum += cput1[smallest];

cput1[smallest] = 0;

}

awt = awt/n;

atat =atat/n;

printf("\nAverage Waiting Time -- %f", awt);

printf("\nAverage Turnaround Time -- %f\n", atat);

}

void main()

{

int n,i;

printf("Enter the number of processes\n");

scanf("%d",&n);

printf("Enter arrival time and cpu time for each process respectively\n");

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

{

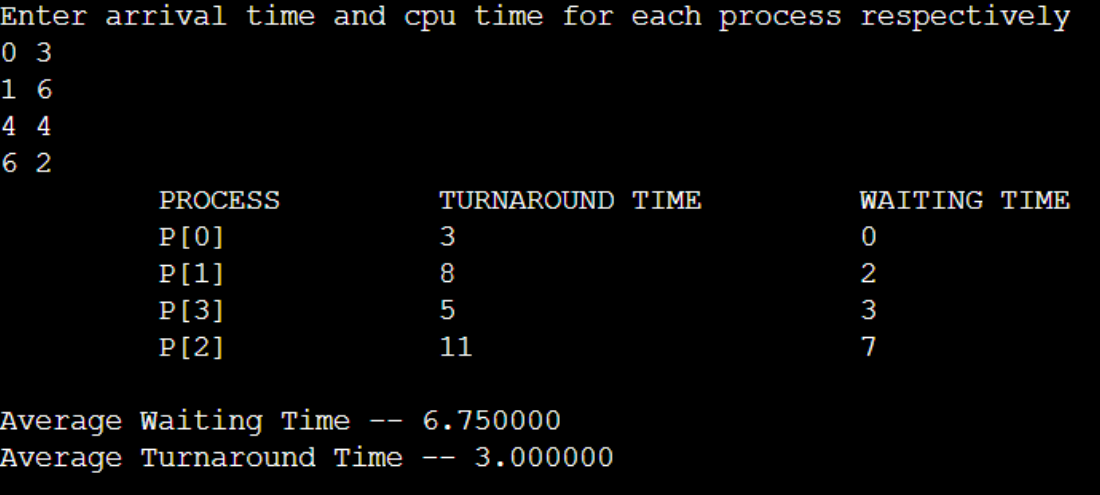
scanf("%d %d",&at[i],&cput[i]);

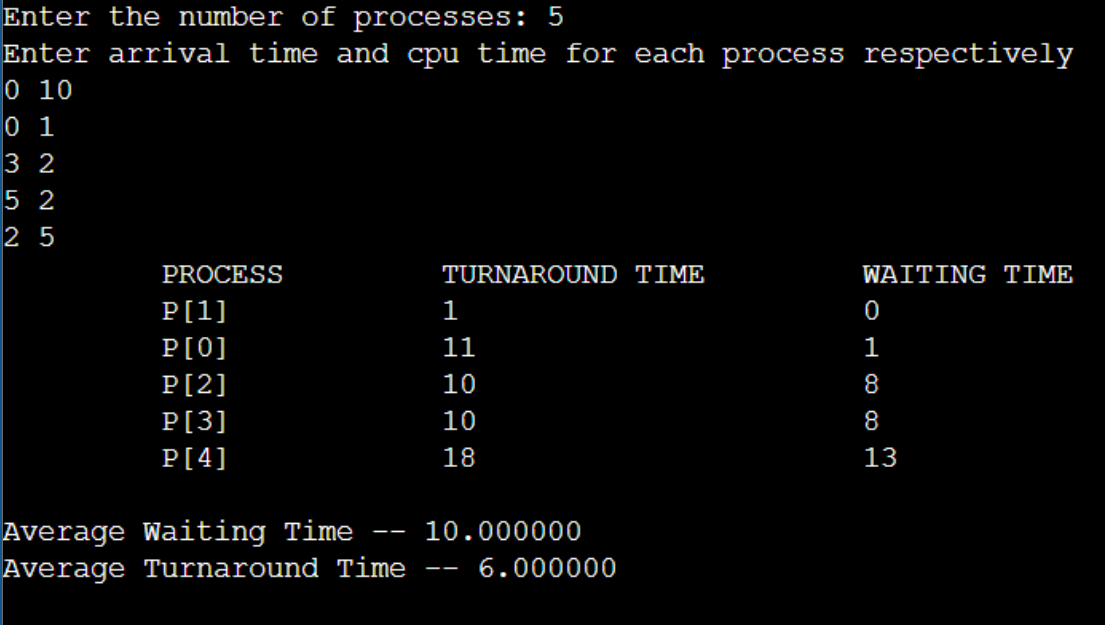
}

sjf(n);

}

**OUTPUT:**

****

****

**SJF (Pre-emptive):**

#include<stdio.h>

int at[20], cput[20];

void srtf(int n)

{

int remaining\_time[20], tat[20], wt[20], completion\_time[20], smallest, time, i, count = 0;

float awt=0,atat=0;

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

remaining\_time[i] = cput[i];

time = 0;

while (count != n)

{

smallest = -1;

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

{

if (at[i] <= time && remaining\_time[i] > 0)

{

if (smallest == -1 || remaining\_time[i] < remaining\_time[smallest])

smallest = i;

}

}

if (smallest == -1) {

time++;

continue;

}

remaining\_time[smallest]--;

if (remaining\_time[smallest] == 0) {

count++;

completion\_time[smallest] = time + 1;

wt[smallest] = completion\_time[smallest] - at[smallest] - cput[smallest];

tat[smallest] = completion\_time[smallest] - at[smallest];

}

time++;

}

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

{

awt+=wt[i];

atat += tat[i];

}

awt = awt/n;

atat =atat/n;

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

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

{

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

}

printf("\nAverage Waiting Time -- %f", awt);

printf("\nAverage Turnaround Time -- %f\n", atat);

}

void main()

{

int n,i,choice;

printf("Enter the number of processes\n");

scanf("%d",&n);

printf("Enter arrival time and cpu time for each process respectively\n");

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

{

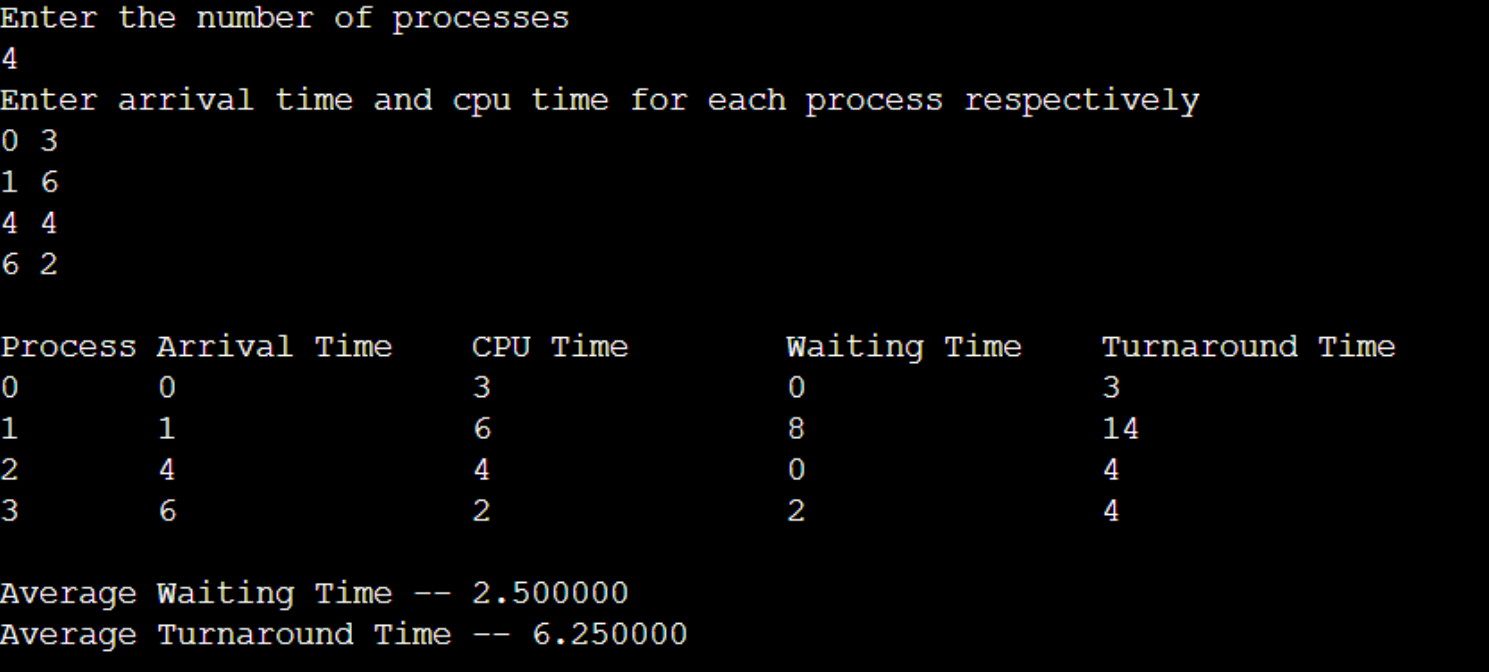
scanf("%d %d",&at[i],&cput[i]);

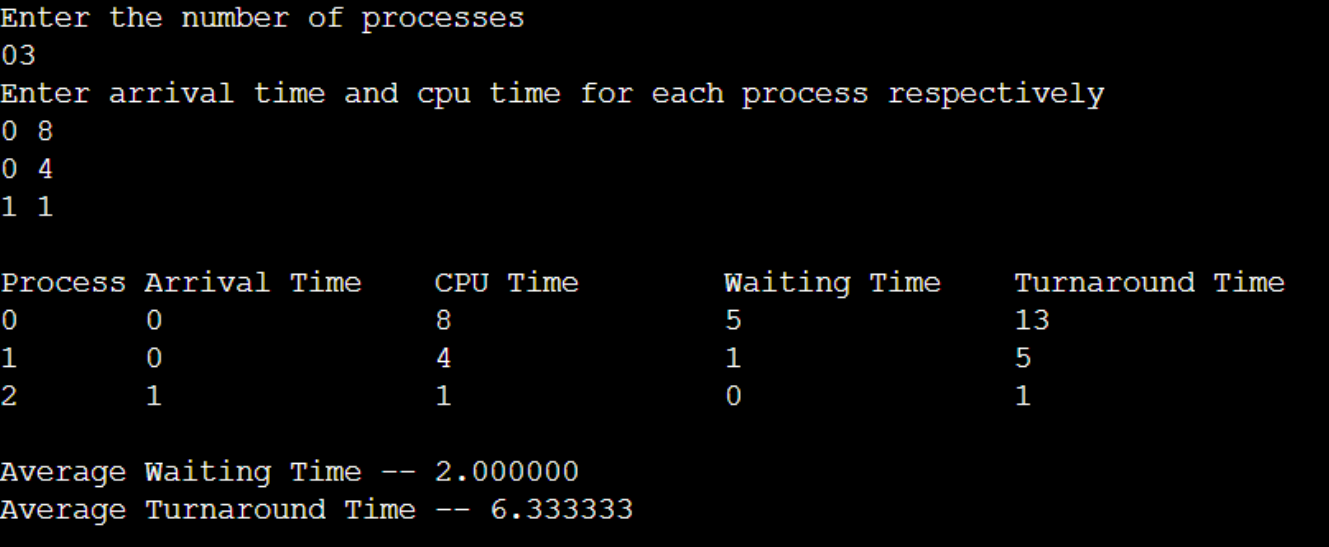
}

srtf(n);

}

**OUTPUT:**





**LAB PROGRAM 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)

**Priority (Pre-emptive & Non-pre-emptive):**

#include <stdio.h>

#include <stdbool.h>

#include<stdlib.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, burst time, priority: ");

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

}

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

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

printf("3. Exit\n");

while(1)

{

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");

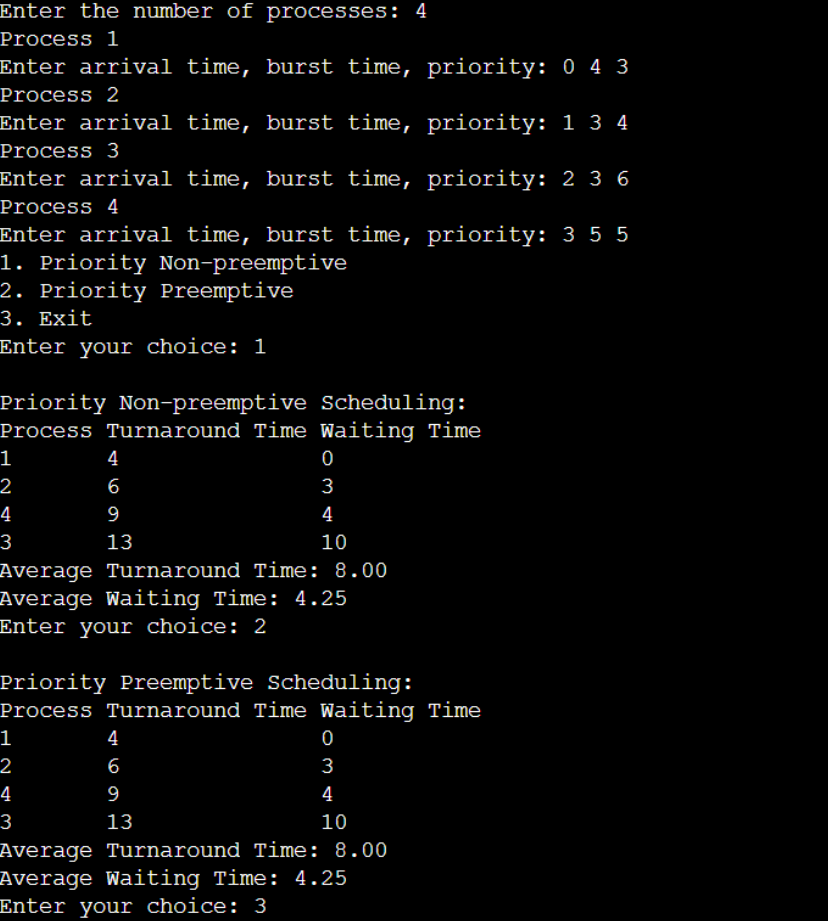
}

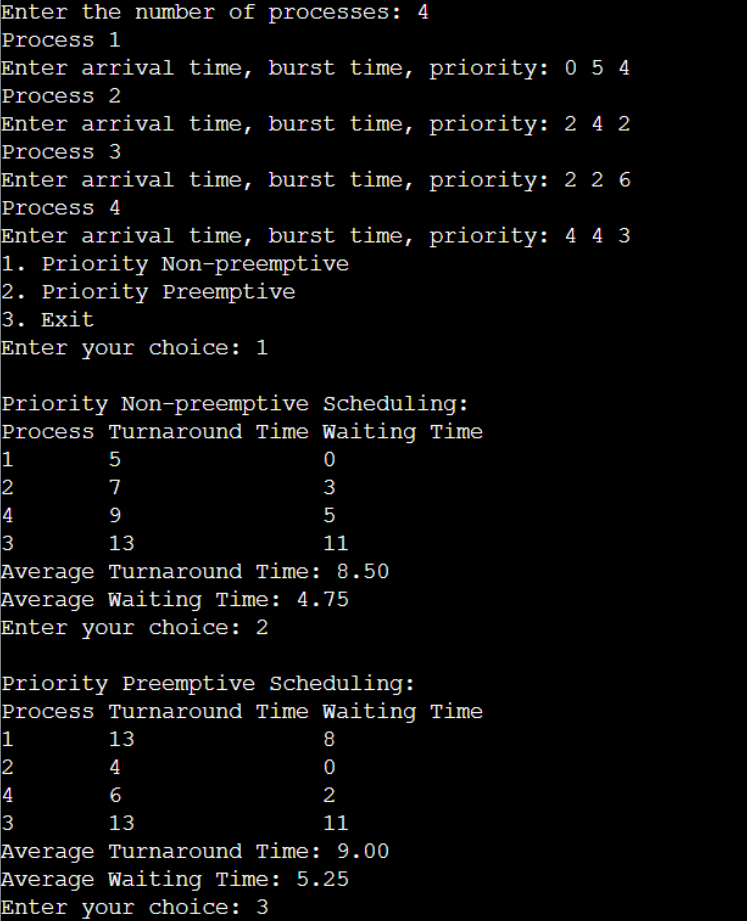
}

return 0;

}

**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;

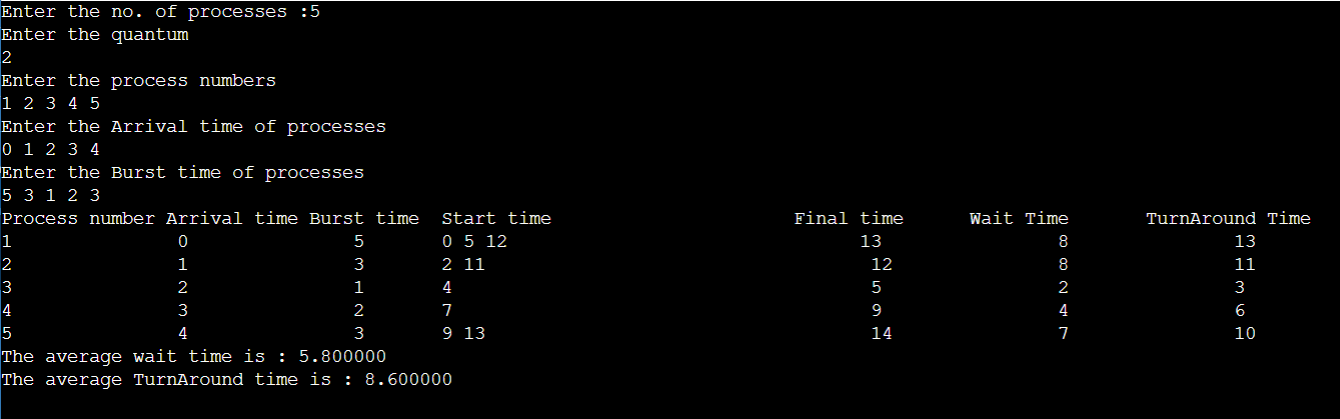
printf("The average wait time is : %lf\n",avg\_wt);

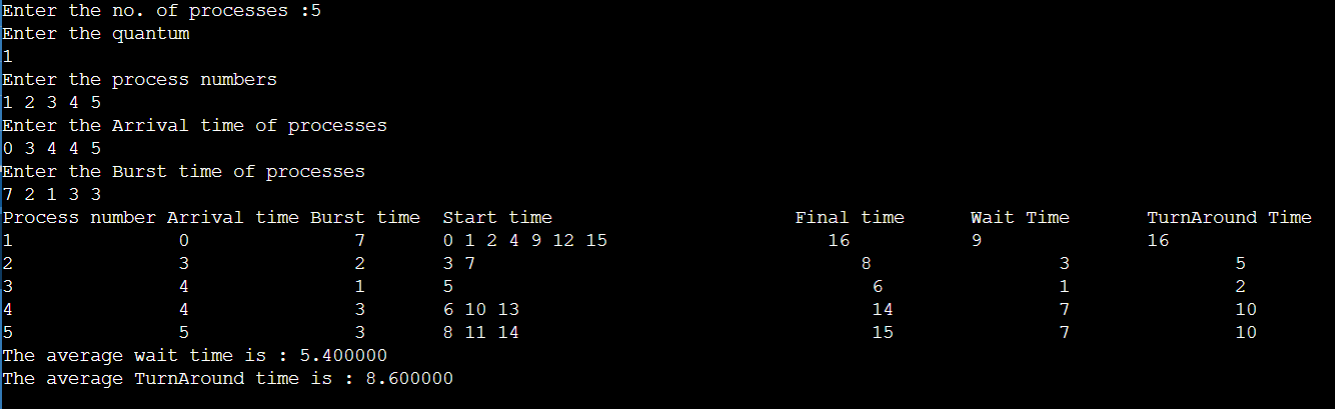
printf("The average TurnAround time is : %lf\n",avg\_tat);

return 0;

}

**OUTPUT:**

****

****

**LAB PROGRAM 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.

#include <stdio.h>

int spat[10], upat[10], i, n1, n2, p1[10], p2[10];

int sppt[10], uppt[10], time = 0, op = 0, y, z, pt;

int sptat[10], uptat[10];

int spwt[10], upwt[10];

float spatat = 0, spawt = 0;

float upatat = 0, upawt = 0;

void process(int x, int isSystem) {

if (isSystem) {

op += sppt[x];

sptat[x] = op - spat[x];

sppt[x] = 0;

spwt[x] = sptat[x] - p1[x];

spatat += sptat[x];

spawt += spwt[x];

} else {

op += uppt[x];

uptat[x] = op - upat[x];

uppt[x] = 0;

upwt[x] = uptat[x] - p2[x];

upatat += uptat[x];

upawt += upwt[x];

}

}

int main() {

printf("Enter the number of System Processes: ");

scanf("%d", &n1);

printf("Enter the number of User Processes: ");

scanf("%d", &n2);

printf("Enter the arrival times for System Processes:\n");

for (i = 0; i < n1; i++)

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

printf("Enter the burst times for System Processes:\n");

for (i = 0; i < n1; i++)

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

printf("Enter the arrival times for User Processes:\n");

for (i = 0; i < n2; i++)

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

printf("Enter the burst times for User Processes:\n");

for (i = 0; i < n2; i++)

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

for (i = 0; i < n1; i++)

time += sppt[i];

for (i = 0; i < n2; i++)

time += uppt[i];

for (i = 0; i < n1; i++)

p1[i] = sppt[i];

for (i = 0; i < n2; i++)

p2[i] = uppt[i];

printf("\n");

while (op < time) {

y = -1;

z = -1;

for (i = 0; i < n1; i++) {

if (op >= spat[i] && sppt[i] != 0) {

y = i;

break;

}

}

for (i = 0; i < n2; i++) {

if (op >= upat[i] && uppt[i] != 0) {

z = i;

break;

}

}

if (y != -1) {

printf("%d SP%d ", op, y + 1);

process(y, 1);

} else if (z != -1) {

printf("%d UP%d ", op, z + 1);

process(z, 0);

} else {

op++;

}

}

printf("%d ",op);

printf("\n");

printf("System Processes:\n");

for (i = 0; i < n1; i++)

printf("SP%d %d 0\n", i + 1, sptat[i]);

printf("Average Turnaround Time (System Processes): %.2f\n", spatat / n1);

printf("Average Waiting Time (System Processes): 0\n");

printf("\n");

printf("User Processes:\n");

for (i = 0; i < n2; i++)

printf("UP%d %d %d\n", i + 1, uptat[i], upwt[i]);

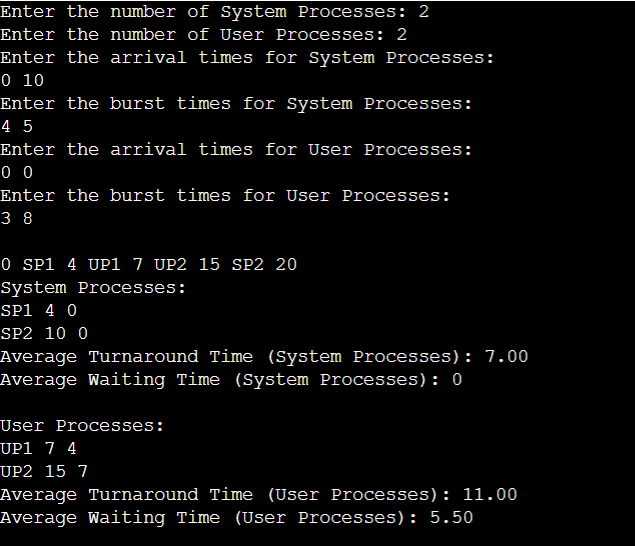
printf("Average Turnaround Time (User Processes): %.2f\n", upatat / n2);

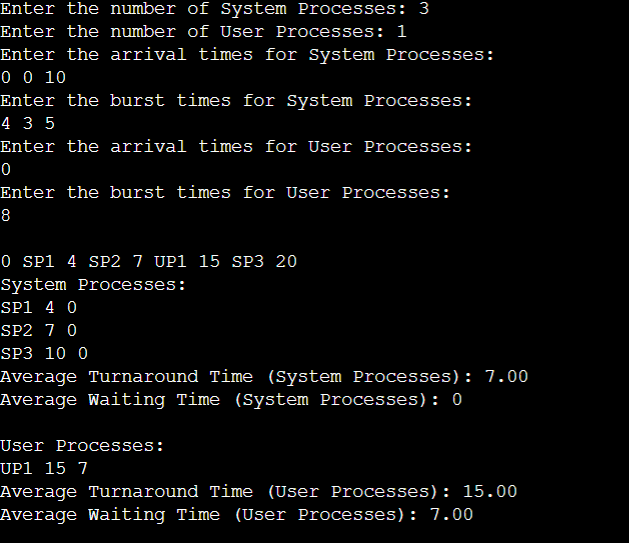
printf("Average Waiting Time (User Processes): %.2f\n", upawt / n2);

return 0;

}

**OUTPUT:**

****

****

**LAB PROGRAM 4**

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

* Rate- Monotonic
* Earliest-deadline First

**Rate- Monotonic:**

#include<stdio.h>

#include<math.h>

void main()

{

int n;

float e[20],p[20];

int i;

float ut,u,x,y;

printf("Enter Number of Processes: ");

scanf("%d",&n);

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

{

printf("\nEnter Execution Time for P%d: ",(i+1));

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

printf("\nEnter Period for P%d: ",(i+1));

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

}

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

{

x=e[i]/p[i];

ut+=x;

}

y=(float)n;

y=y\*((pow(2.0,1/y))-1);

u=y;

if(ut<u)

{

printf("\nAs %f < %f ,",ut,u);

printf("\nThe System is Schedulable");

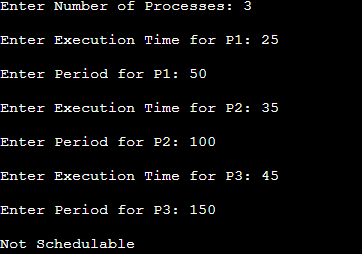
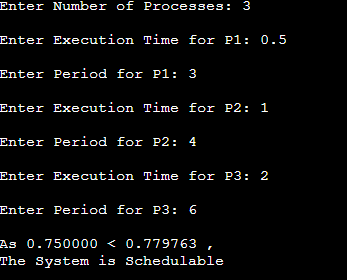
}

else

printf("\nNot Schedulable");

}

**OUTPUT:**

** **

**Earliest-deadline First:**

#include <stdio.h>

#include <stdlib.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)

{ if (b == 0)

return a;

else

return gcd(b, a % 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)

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];

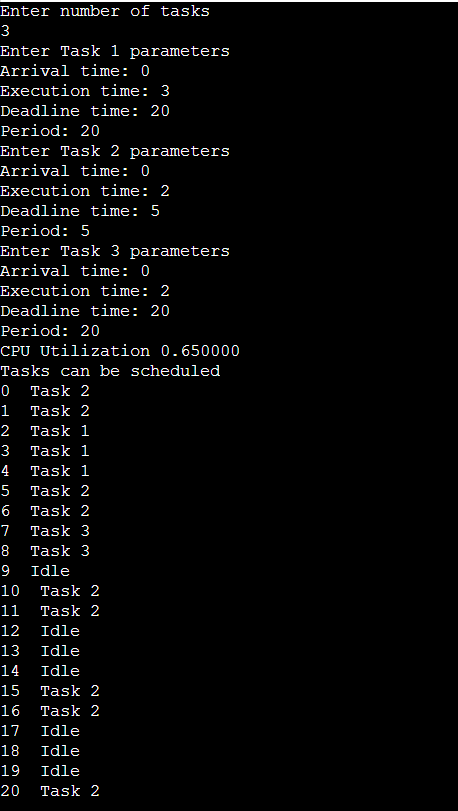
t1++;

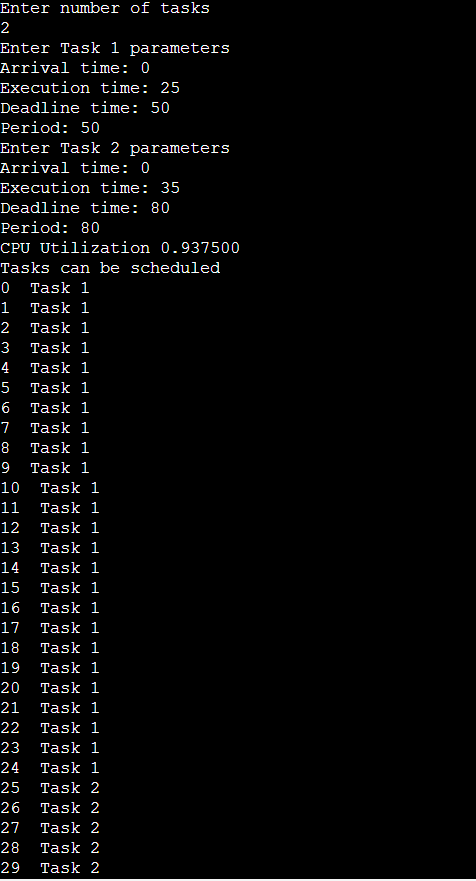
i++;

}return cu;

}

**OUTPUT:**

****

****

**LAB PROGRAM 5**

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

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define BUFFER\_SIZE 5

#define MAX\_ITEMS 20

int buffer[BUFFER\_SIZE];

int in = 0;

int out = 0;

int produced\_count = 0;

int consumed\_count = 0;

sem\_t mutex;

sem\_t full;

sem\_t empty;

void\* producer(void\* arg) {

int item = 1;

while (produced\_count < MAX\_ITEMS) {

sem\_wait(&empty);

sem\_wait(&mutex);

buffer[in] = item;

printf("Produced: %d\n", item);

item++;

in = (in + 1) % BUFFER\_SIZE;

produced\_count++;

sem\_post(&mutex);

sem\_post(&full);

} pthread\_exit(NULL);

}

void\* consumer(void\* arg) {

while (consumed\_count < MAX\_ITEMS) {

sem\_wait(&full);

sem\_wait(&mutex);

int item = buffer[out];

printf("Consumed: %d\n", item);

out = (out + 1) % BUFFER\_SIZE;

consumed\_count++;

sem\_post(&mutex);

sem\_post(&empty);

} pthread\_exit(NULL);

}

int main() {

pthread\_t producerThread, consumerThread;

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

sem\_init(&full, 0, 0);

sem\_init(&empty, 0, BUFFER\_SIZE);

pthread\_create(&producerThread, NULL, producer, NULL);

pthread\_create(&consumerThread, NULL, consumer, NULL);

pthread\_join(producerThread, NULL);

pthread\_join(consumerThread, NULL);

sem\_destroy(&mutex);

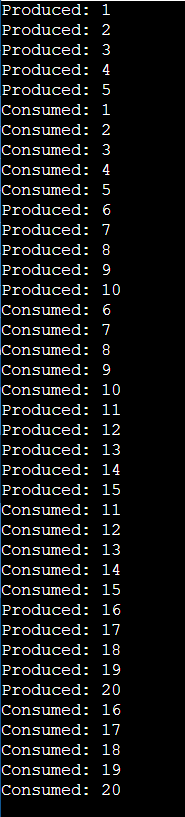
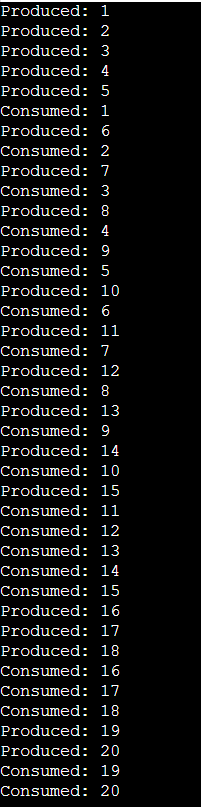
sem\_destroy(&full);

sem\_destroy(&empty);

return 0;

}

**OUTPUT:**

** **

**LAB PROGRAM 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);

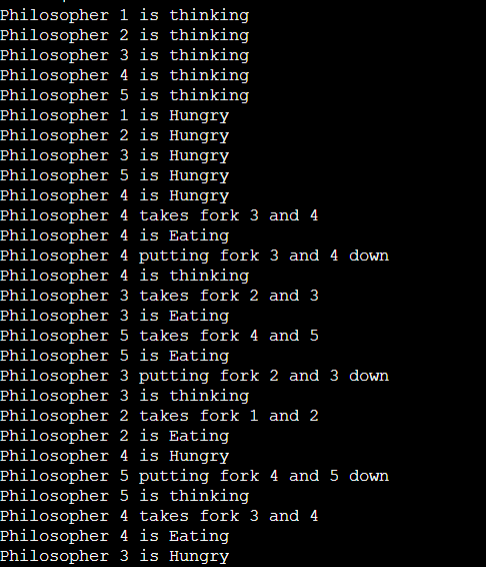
}

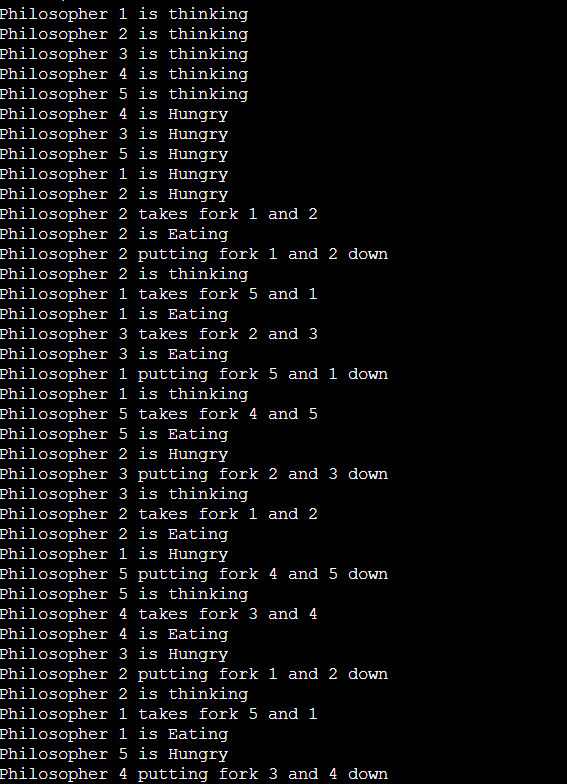
for (i = 0; i < N; i++)

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

}

**OUTPUT:**

****

****

**LAB PROGRAM 7**

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

#include<stdio.h>

#include<conio.h>

int max[100][100];

int alloc[100][100];

int need[100][100];

int avail[100];

int n,r;

void input();

void show();

void cal();

int main()

{

int i,j;

printf("\*\*\*\*\*\*\*\*\*\* Banker's Algo \*\*\*\*\*\*\*\*\*\*\*\*\n");

input();

show();

cal();

getch();

return 0;

}

void input()

{

int i,j;

printf("Enter the no of Processes\t");

scanf("%d",&n);

printf("Enter the no of resources instances\t");

scanf("%d",&r);

printf("Enter the Max Matrix\n");

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

{

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

{

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

}

}

printf("Enter the Allocation Matrix\n");

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

{

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

{

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

}

}

printf("Enter the available Resources\n");

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

{

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

}

}

void show()

{

int i,j;

printf("Process\t Allocation\t Max\t Available\t");

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

{

printf("\nP%d\t ",i+1);

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

{

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

}

printf("\t");

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

{

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

}

printf("\t");

if(i==0)

{

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

printf("%d ",avail[j]);

}

}

}

void cal()

{

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

int safe[100];

int i,j;

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

{

finish[i]=0;

}

//find need matrix

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

{

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

{

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

}

}

printf("\n");

while(flag)

{

flag=0;

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

{

int c=0;

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

{

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

{

c++;

if(c==r)

{

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

{

avail[k]+=alloc[i][j];

finish[i]=1;

flag=1;

}

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

if(finish[i]==1)

{

i=n;

}

}

}

}

}

}

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

{

if(finish[i]==1)

{

c1++;

}

else

{

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

}

}

if(c1==n)

{

printf("\n The system is in safe state");

}

else

{

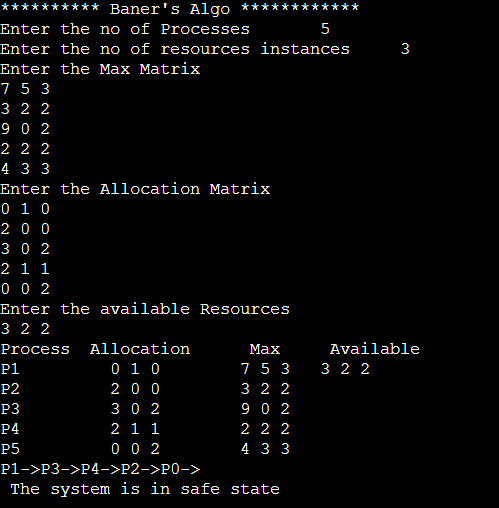
printf("\n Process are in dead lock");

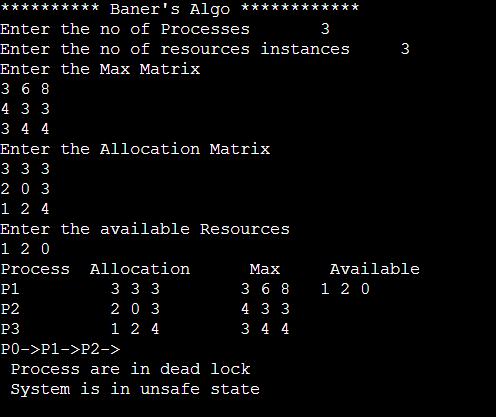
printf("\n System is in unsafe state");

}

}

**OUTPUT:**

****

****

**LAB PROGRAM 8**

Write a C program to simulate deadlock detection.

#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 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(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];

}

}

}

//checking for unmarked processes

int deadlock=0;

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

if(mark[i]!=1)

deadlock=1;

if(deadlock)

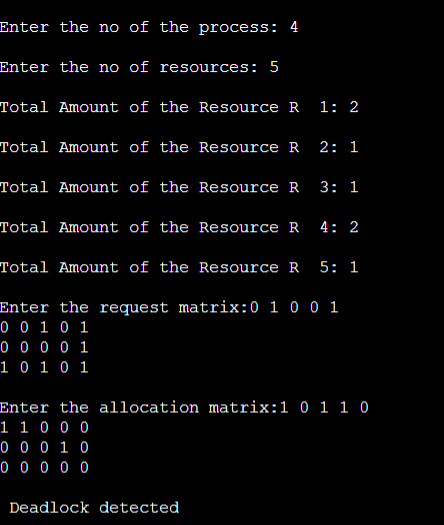
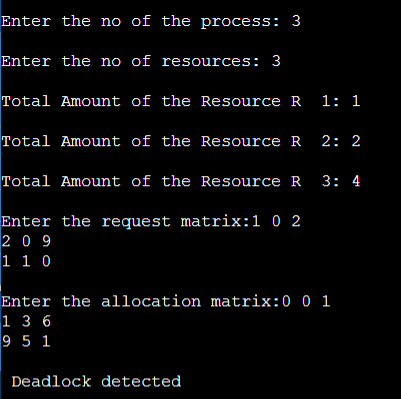
printf("\n Deadlock detected");

else

printf("\n No Deadlock possible");

}

**OUTPUT:**

** **

**LAB PROGRAM 9**

Write a C program to simulate the following contiguous memory

allocation techniques.

* Worst-fit
* Best-fit
* First-fit

**Worst-fit:**

#include <stdio.h>

#include <string.h>

// Function to allocate memory to blocks as per worst fit

// algorithm

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

{

// Stores block id of the block allocated to a

// process

int allocation[n];

// Initially no block is assigned to any process

memset(allocation, -1, sizeof(allocation));

// pick each process and find suitable blocks

// according to its size and assign to it

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

{

// Find the best fit block for the current process

int wstIdx = -1;

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

{

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

{

if (wstIdx == -1)

wstIdx = j;

else if (blockSize[wstIdx] < blockSize[j])

wstIdx = j;

}

}

// If we could find a block for the current process

if (wstIdx != -1)

{

// allocate block j to p[i] process

allocation[i] = wstIdx;

// Reduce available memory in this block.

blockSize[wstIdx] -= processSize[i];

}

}

printf("\nProcess 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", allocation[i] + 1);

else

printf("Not Allocated");

printf("\n");

}

}

// Driver code

int main()

{

int blockSize[] = {100, 500, 200, 300, 600};

int processSize[] = {212, 417, 112, 426};

int m = sizeof(blockSize) / sizeof(blockSize[0]);

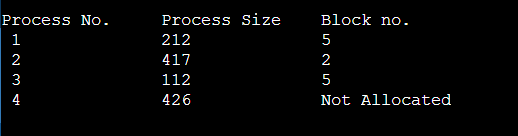
int n = sizeof(processSize) / sizeof(processSize[0]);

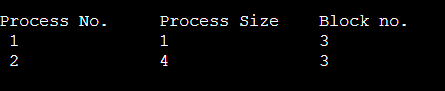
worstFit(blockSize, m, processSize, n);

return 0;

}

**OUTPUT:**

****

****

**Best-fit:**

#include <stdio.h>

void implimentBestFit(int blockSize[], int blocks, int processSize[], int processes)

{

// This will store the block id of the allocated block to a process

int allocation[processes];

// initially assigning -1 to all allocation indexes

// means nothing is allocated currently

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

allocation[i] = -1;

}

// pick each process and find suitable blocks

// according to its size ad assign to it

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

{

int indexPlaced = -1;

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

{

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

{

// place it at the first block fit to accomodate process

if (indexPlaced == -1)

indexPlaced = j;

// if any future block is better that is

// any future block with smaller size encountered

// that can accomodate the given process

else if (blockSize[j] < blockSize[indexPlaced])

indexPlaced = j;

}

}

// If we were successfully able to find block for the process

if (indexPlaced != -1)

{

// allocate this block j to process p[i]

allocation[i] = indexPlaced;

// Reduce available memory for the block

blockSize[indexPlaced] -= processSize[i];

}

}

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

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

{

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

if (allocation[i] != -1)

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

else

printf("Not Allocated\n");

}

}

// Driver code

int main()

{

int blockSize[] = {50, 20, 100, 90};

int processSize[] = {10, 30, 60, 80};

int blocks = sizeof(blockSize)/sizeof(blockSize[0]);

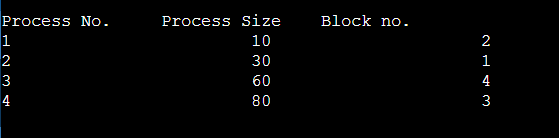
int processes = sizeof(processSize)/sizeof(processSize[0]);

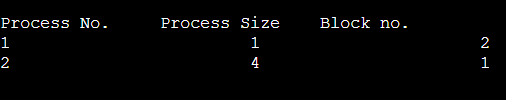
implimentBestFit(blockSize, blocks, processSize, processes);

return 0 ;

}

**OUTPUT:**

****

****

**First-fit:**

#include <stdio.h>

void implimentFirstFit(int blockSize[], int blocks, int processSize[], int processes)

{

// This will store the block id of the allocated block to a process

int allocate[processes];

// initially assigning -1 to all allocation indexes

// means nothing is allocated currently

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

{

allocate[i] = -1;

}

// take each process one by one and find

// first block that can accomodate it

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

{

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

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

{

// allocate block j to p[i] process

allocate[i] = j;

// Reduce size of block j as it has accomodated p[i]

blockSize[j] -= processSize[i];

break;

}

}

}

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

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

{

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

if (allocate[i] != -1)

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

else

printf("Not Allocated\n");

}

}

void main()

{

int blockSize[] = {30, 5, 10};

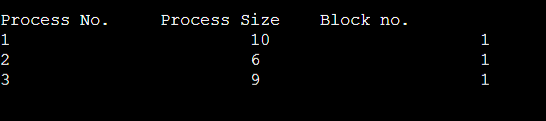
int processSize[] = {10, 6, 9};

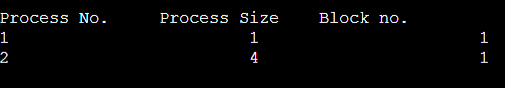
int m = sizeof(blockSize)/sizeof(blockSize[0]);

int n = sizeof(processSize)/sizeof(processSize[0]);

implimentFirstFit(blockSize, m, processSize, n);

}





**LAB PROGRAM 10**

Write a C program to simulate page replacement algorithms

* FIFO
* LRU
* Optimal

**FIFO:**

#include<stdio.h>

int main()

{

int i,j,n,a[50],frame[10],no,k,avail,count=0;

printf("ENTER THE NUMBER OF PAGES: ");

scanf("%d",&n);

printf("\nENTER THE PAGE REFERENCE : ");

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

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

printf("\nENTER THE NUMBER OF FRAMES : ");

scanf("%d",&no);

for(i=0;i<no;i++)

frame[i]= -1;

j=0;

printf("\tref string\t page frames\n");

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

{

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

avail=0;

for(k=0;k<no;k++)

if(frame[k]==a[i])

avail=1;

if (avail==0)

{

frame[j]=a[i];

j=(j+1)%no;

count++;

for(k=0;k<no;k++)

printf("%d\t",frame[k]);

}

printf("\n");

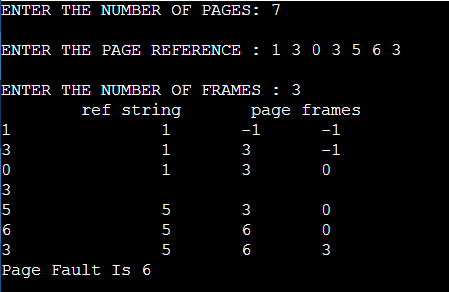
}

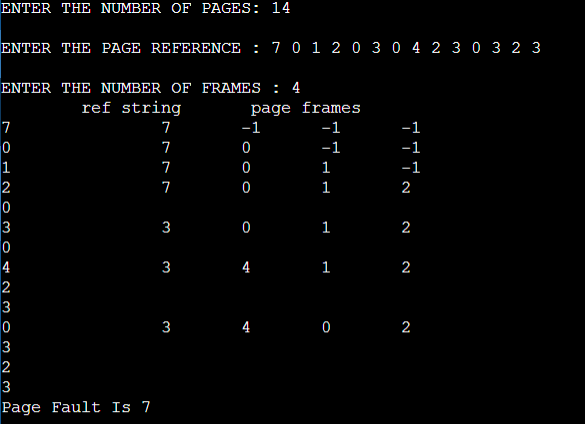
printf("Page Fault Is %d",count);

return 0;

}

**OUTPUT:**

****

****

**LRU:**

#include<stdio.h>

void main()

{

int q[20],p[50],c=0,c1,d,f,i,j,k=0,n,r,t,b[20],c2[20];

printf("Enter no of pages: ");

scanf("%d",&n);

printf("Enter the reference string: ");

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

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

printf("Enter no of frames: ");

scanf("%d",&f);

q[k]=p[k];

printf("\n\t%d\n",q[k]);

c++;

k++;

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

{

c1=0;

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

{

if(p[i]!=q[j])

c1++;

}

if(c1==f)

{

c++;

if(k<f)

{

q[k]=p[i];

k++;

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

printf("\t%d",q[j]);

printf("\n");

}

else

{

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

{

c2[r]=0;

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

{

if(q[r]!=p[j])

c2[r]++;

else

break;

}

}

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

b[r]=c2[r];

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

{

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

{

if(b[r]<b[j])

{

t=b[r];

b[r]=b[j];

b[j]=t;

}

}

}

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

{

if(c2[r]==b[0])

q[r]=p[i];

printf("\t%d",q[r]);

}

printf("\n");

}

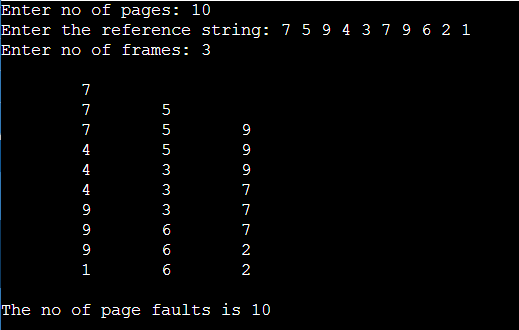
}

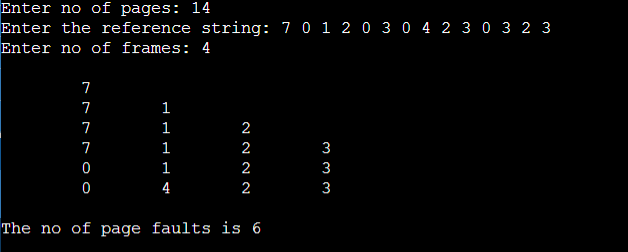
}

printf("\nThe no of page faults is %d",c);

}

**OUTPUT:**

****

****

**Optimal:**

#include<stdio.h>

int main()

{

int n,pg[30],fr[10];

int count[10],i,j,k,fault,f,flag,temp,current,c,dist,max,m,cnt,p,x;

fault=0;

dist=0;

k=0;

printf("Enter the total no pages: ");

scanf("%d",&n);

printf("\nEnter the sequence: ");

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

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

printf("\nEnter frame size: ");

scanf("%d",&f);

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

{

count[i]=0;

fr[i]=-1;

}

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

{

flag=0;

temp=pg[i];

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

{

if(temp==fr[j])

{

flag=1;

break;

}

}

if((flag==0)&&(k<f))

{

fault++;

fr[k]=temp;

k++;

}

else if((flag==0)&&(k==f))

{

fault++;

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

{

current=fr[cnt];

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

{

if(current!=pg[c])

count[cnt]++;

else

break;

}

}

max=0;

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

{

if(count[m]>max)

{

max=count[m];

p=m;

}

}

fr[p]=temp;

}

printf("\npage %d frame\t",pg[i]);

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

{

printf("%d\t",fr[x]);

}

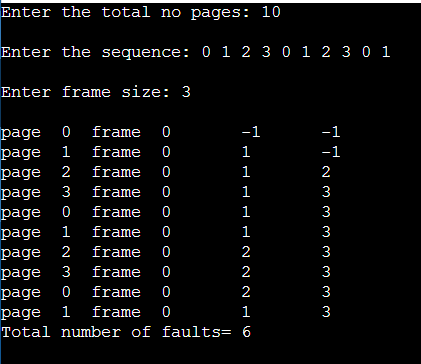
}

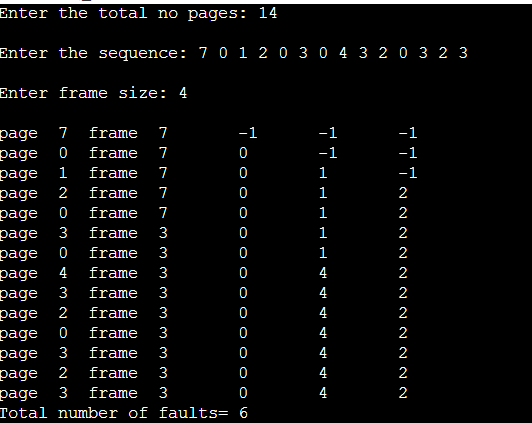
printf("\nTotal number of faults= %d",fault);

return 0;

}

**OUTPUT:**

****

****

**LAB PROGRAM 11**

Write a C program to simulate disk scheduling algorithms

* FCFS
* SCAN
* C-SCAN

**FCFS:**

#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,n,TotalHeadMoment=0,initial;

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

scanf("%d",&n);

printf("\nEnter the Requests sequence: ");

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

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

printf("\nEnter initial head position: ");

scanf("%d",&initial);

// logic for FCFS disk scheduling

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

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

initial=RQ[i];

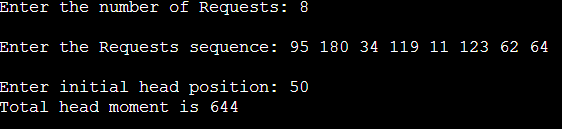
}

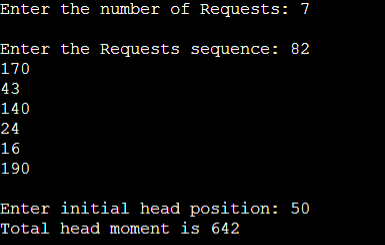
printf("Total head moment is %d",TotalHeadMoment);

return 0;

}

**OUTPUT:**





**SCAN:**

#include <stdio.h>

int request[50];

int SIZE;

int pre;

int head;

int uptrack;

int downtrack;

struct max{

int up;

int down;

} kate[50];

int dist(int a, int b){

if (a > b)

return a - b;

return b - a;

}

void sort(int n){

int i, j;

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

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

if (request[j] > request[j + 1]){

int temp = request[j];

request[j] = request[j + 1];

request[j + 1] = temp;

}

}

}

j = 0;

i = 0;

while (request[i] != head){

kate[j].down = request[i];

j++;

i++;

}

downtrack = j;

i++;

j = 0;

while (i < n){

kate[j].up = request[i];

j++;

i++;

}

uptrack = j;

}

void scan(int n){

int i;

int seekcount = 0;

printf("SEEK SEQUENCE = ");

sort(n);

if (pre < head){

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

printf("%d ", head);

seekcount = seekcount + dist(head, kate[i].up);

head = kate[i].up;

}

for (i = downtrack - 1; i > 0; i--){

printf("%d ", head);

seekcount = seekcount + dist(head, kate[i].down);

head = kate[i].down;

}

}

else{

for (i = downtrack - 1; i >= 0; i--){

printf("%d ", head);

seekcount = seekcount + dist(head, kate[i].down);

head = kate[i].down;

}

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

printf("%d ", head);

seekcount = seekcount + dist(head, kate[i].up);

head = kate[i].up;

}

}

printf(" %d\nTOTAL DISTANCE :%d", head, seekcount);

}

int main(){

int n, i;

printf("ENTER THE DISK SIZE : ");

scanf("%d", &SIZE);

printf("\nENTER THE NO OF REQUEST SEQUENCE : ");

scanf("%d", &n);

printf("\nENTER THE REQUEST SEQUENCE : ");

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

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

printf("\nENTER THE CURRENT HEAD : ");

scanf("%d", &head);

request[n] = head;

request[n + 1] = SIZE - 1;

request[n + 2] = 0;

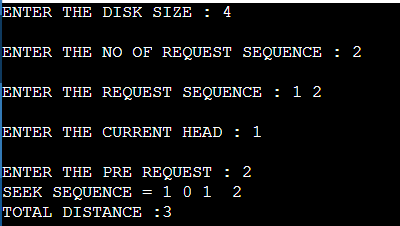
printf("\nENTER THE PRE REQUEST : ");

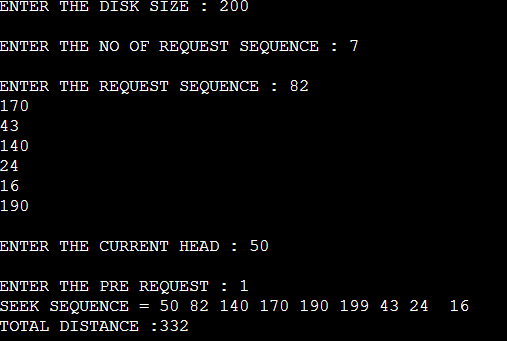
scanf("%d", &pre);

scan(n + 3);

}

**OUTPUT:**





**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: ");

scanf("%d", &n);

printf("\nEnter the Requests sequence: ");

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

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

printf("\nEnter initial head position: ");

scanf("%d", &initial);

printf("\nEnter total disk size: ");

scanf("%d", &size);

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

scanf("%d", &move);

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

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

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

int temp;

temp = RQ[j];

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

RQ[j + 1] = temp;

}

}

}

int index;

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

if (initial < RQ[i]){

index = i;

break;

}

}

if (move == 1){

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

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

initial = RQ[i];

}

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

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

initial = 0;

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

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

initial = RQ[i];

}

}

else{

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

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

initial = RQ[i];

}

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

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

initial = size - 1;

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

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

initial = RQ[i];

}

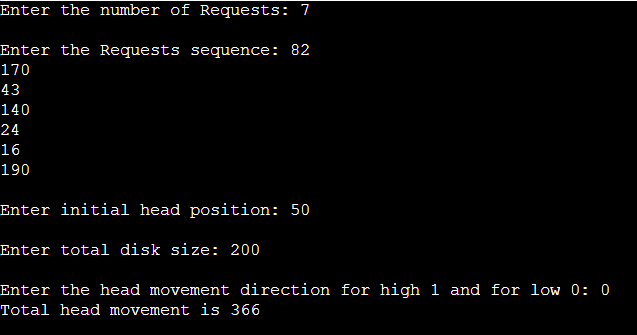
}

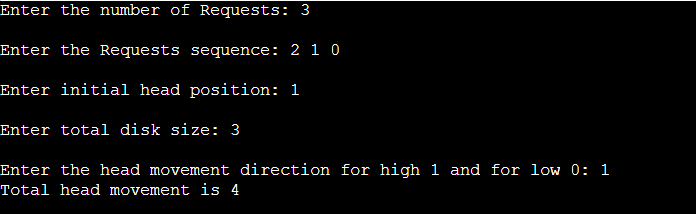
printf("Total head movement is %d", TotalHeadMoment);

return 0;

}

**OUTPUT:**

****

****