**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

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

****

**LAB REPORT**

**on**

**OPERATING SYSTEMS**

(23CS4PCOPS)

***Submitted by***

**HRUTHVIK MALLIKARJUNAPPA(1BM22CS114)**

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

**BACHELOR OF ENGINEERING**

***in***

**COMPUTER SCIENCE AND ENGINEERING**



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

**(Autonomous Institution under VTU)**

**BENGALURU-560019**

**Apr-2024 to Aug-2024**

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

**Bull Temple Road, Bangalore 560019**

(Affiliated To Visvesvaraya Technological University, Belgaum)

**Department of Computer Science and Engineering**



**CERTIFICATE**

This is to certify that the Lab work entitled “OPERATING SYSTEMS – 23CS4PCOPS” carried out by **HRUTHVIK MALLIKARJUNAPPA(1BM22CS114),** who is bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2024. The Lab report has been approved as it satisfies the academic requirements in respect of **OPERATING SYSTEMS - (23CS4PCOPS)** work prescribed for the said degree.

Radhika A D               **Dr. Jyothi S Nayak**

Assistant Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

**Index Sheet**

|  |  |  |
| --- | --- | --- |
| **Sl. No.** | **Experiment Title** | **Page No.** |
|  | Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time. →FCFS → SJF (pre-emptive & Non-preemptive) | 4 |
|  | 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) | 14 |
|  | 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. | 24 |
|  | Write a C program to simulate Real-Time CPU Scheduling algorithms: a) Rate- Monotonic b) Earliest-deadline First c) Proportional scheduling | 28 |
|  | Write a C program to simulate producer-consumer problem using semaphores. | 41 |
|  | Write a C program to simulate the concept of Dining-Philosophers problem. | 44 |
|  | Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance. | 49 |
|  | Write a C program to simulate deadlock detection | 54 |
|  | Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit b) Best-fit c) First-fit | 59 |
|  | Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) Optimal | 64 |

**Course Outcome**

|  |  |
| --- | --- |
| CO1 | Apply the different concepts and functionalities of Operating System |
| CO2 | Analyze 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 |

**Program -1**

**Question:** Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time. →FCFS → SJF (pre-emptive & non-preemptive)

**Code:**

**1.FCFS**

#include<stdio.h>

void main()

{

int n;

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

scanf("%d",&n);

int pr[n], at[n], bt[n], ct[n], tat[n], wt[n];

printf("Enter Process number:\n");

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

{

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

}

printf("Enter Arrival Time:\n");

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

{

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

}

printf("Enter Burst Time:\n");

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

{

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

}

int temp1, temp2, temp3;

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

{

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

{

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

{

temp1 = at[j];

at[j] = at[i];

at[i] = temp1;

temp2 = bt[j];

bt[j] = bt[i];

bt[i] = temp2;

temp3 = pr[j];

pr[j] = pr[i];

pr[i] = temp3;

}

}

}

int x=at[0];

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

{

if (x<at[i])

{

x = at[i];

}

ct[i] = bt[i] + x;

x = ct[i];

}

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

{

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

}

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

{

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

}

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

{

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

}

float avg\_tat = 0, avg\_wt = 0;

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

{

avg\_tat = avg\_tat + tat[i];

avg\_wt = avg\_wt + wt[i];

}

avg\_tat = avg\_tat/n;

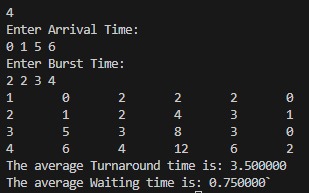
avg\_wt = avg\_wt/n;

printf("The average Turnaround time is: %f", avg\_tat);

printf("\nThe average Waiting time is: %f`", avg\_wt);

}

**Result:**



**2.SJF(Pre-emptive)**

#include <stdio.h>

void findCompletionTime(int processes[], int n, int bt[], int at[], int wt[], int tat[], int ct[])

{

int remaining[n];

int currentTime = 0;

int completed = 0;

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

remaining[i] = bt[i];

while (completed < n)

{

int shortest = -1;

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

{

if (at[i] <= currentTime && remaining[i] > 0)

{

if (shortest == -1 || remaining[i] <= remaining[shortest])

shortest = i;

}

}

if (shortest == -1)

{

currentTime++;

continue;

}

remaining[shortest]--;

if (remaining[shortest] == 0)

{

completed++;

ct[shortest] = currentTime + 1;

wt[shortest] = ct[shortest] - bt[shortest] - at[shortest];

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

}

currentTime++;

}

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

{

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

}

float avg\_tat = 0, avg\_wt = 0;

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

{

avg\_tat += tat[i];

avg\_wt += wt[i];

}

avg\_tat /= n;

avg\_wt /= n;

printf("The average Turnaround time is %f\n", avg\_tat);

printf("The average Waiting time is %f\n", avg\_wt);

}

void main()

{

int n;

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

scanf("%d", &n);

int processes[n];

int burst\_time[n];

int arrival\_time[n];

printf("Enter Process Number:\n");

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

{

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

}

printf("Enter Arrival Time:\n");

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

{

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

}

printf("Enter Burst Time:\n");

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

{

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

}

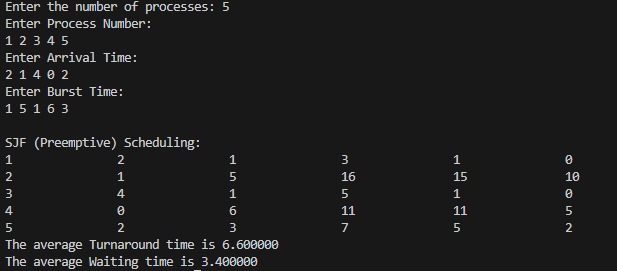
int wt[n], tat[n], ct[n];

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

findCompletionTime(processes, n, burst\_time, arrival\_time, wt, tat, ct);

}

**Result:**



**3.SJF(NON-pre-emptive)**

#include<stdio.h>

void findCompletionTime(int processes[], int n, int bt[], int at[], int wt[], int tat[], int rt[], int ct[])

{

int completion[n]; // Array to store completion times of processes

int remaining[n]; // Array to store remaining burst time of processes

// Initialize remaining array with burst times

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

remaining[i] = bt[i];

int currentTime = 0; // Current time

// Find process with shortest burst time

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

{

int shortest = -1;

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

{

if (at[j] <= currentTime && remaining[j] > 0)

{

if (shortest == -1 || remaining[j] < remaining[shortest])

shortest = j;

}

}

if (shortest == -1)

{

currentTime++;

continue;

}

completion[shortest] = currentTime + remaining[shortest];

currentTime = completion[shortest];

wt[shortest] = currentTime - bt[shortest] - at[shortest];

tat[shortest] = currentTime - at[shortest];

rt[shortest] = wt[shortest]; // Response time for non-preemptive SJF is the same as waiting time

remaining[shortest] = 0;

}

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

{

ct[i] = completion[i];

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

}

float avg\_tat = 0, avg\_wt = 0;

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

{

avg\_tat += tat[i];

avg\_wt += wt[i];

}

avg\_tat /= n;

avg\_wt /= n;

printf("The average Turnaround time is %f\n", avg\_tat);

printf("The average Waiting time is %f\n", avg\_wt);

}

void main()

{

int n;

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

scanf("%d", &n);

int processes[n];

int burst\_time[n];

int arrival\_time[n];

printf("Enter Process Number:\n");

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

{

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

}

printf("Enter Arrival Time:\n");

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

{

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

}

printf("Enter Burst Time:\n");

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

{

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

}

int wt[n], tat[n], rt[n], ct[n];

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

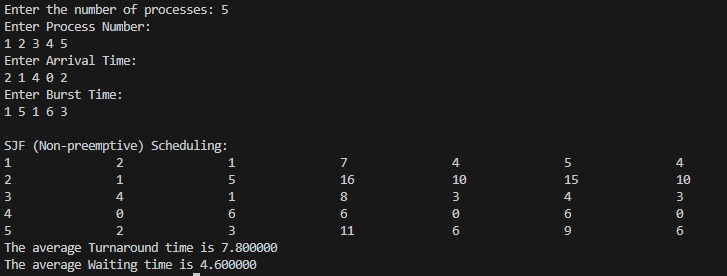
rt[i] = -1;

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

findCompletionTime(processes, n, burst\_time, arrival\_time, wt, tat, rt, ct);

}

**Result:**



**Program -2**

**Question:** 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)

**Code:**

**1.Priority(Pre-emptive)**

#include <stdio.h>

#include <stdbool.h>

// Function to find the waiting time, turnaround time, and completion time for all processes using Priority Scheduling (Preemptive)

void findCompletionTime(int processes[], int n, int bt[], int at[], int wt[], int tat[], int ct[], int rt[], int priority[], bool isLowerPriorityHigher)

{

int remaining[n]; // Array to store remaining burst time of processes

int currentTime = 0; // Current time

int completed = 0; // Counter for completed processes

bool isFinished[n]; // Array to indicate if the process is finished

// Initialize remaining array with burst times and set response times

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

remaining[i] = bt[i];

isFinished[i] = false;

rt[i] = -1; // Response time is initially unset

}

while (completed < n) {

int highestPriorityIndex = -1;

int highestPriority = isLowerPriorityHigher ? 1000000 : -1; // Adjust initial value based on priority type

// Find the process with the highest priority that has arrived and is not finished

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

if (at[i] <= currentTime && !isFinished[i] &&

((isLowerPriorityHigher && priority[i] < highestPriority) ||

(!isLowerPriorityHigher && priority[i] > highestPriority))) {

highestPriority = priority[i];

highestPriorityIndex = i;

}

}

// If no process is found, move to the next time

if (highestPriorityIndex == -1) {

currentTime++;

continue;

}

int currentProcess = highestPriorityIndex;

// Set response time if it's the first time the process is executed

if (rt[currentProcess] == -1) {

rt[currentProcess] = currentTime - at[currentProcess];

}

// Execute the process for 1 unit of time

remaining[currentProcess]--;

currentTime++;

// If the process is completed

if (remaining[currentProcess] == 0) {

isFinished[currentProcess] = true;

completed++;

ct[currentProcess] = currentTime; // Set completion time for the process

tat[currentProcess] = ct[currentProcess] - at[currentProcess]; // Calculate turnaround time

wt[currentProcess] = tat[currentProcess] - bt[currentProcess]; // Calculate waiting time

}

}

// Print the table

printf("Process\tArrival Time\tBurst Time\tPriority\tCompletion Time\tTurnaround Time\tWaiting Time\tResponse Time\n");

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

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",

processes[i], at[i], bt[i], priority[i], ct[i], tat[i], wt[i], rt[i]);

}

}

void main()

{

// Number of processes

int n;

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

scanf("%d", &n);

// Process id's

int processes[n];

// Burst time of all processes

int burst\_time[n];

// Arrival time of all processes

int arrival\_time[n];

// Priority of all processes

int priority[n];

// Priority type (true for lower number = higher priority, false for higher number = higher priority)

int priorityType;

bool isLowerPriorityHigher;

printf("Enter 1 if lower number indicates higher priority, 0 if higher number indicates higher priority: ");

scanf("%d", &priorityType);

isLowerPriorityHigher = (priorityType == 1);

printf("Enter Process Number:\n");

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

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

}

printf("Enter Priority:\n");

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

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

}

printf("Enter Arrival Time:\n");

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

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

}

printf("Enter Burst Time:\n");

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

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

}

// Arrays to store waiting time, turnaround time, completion time, and response time

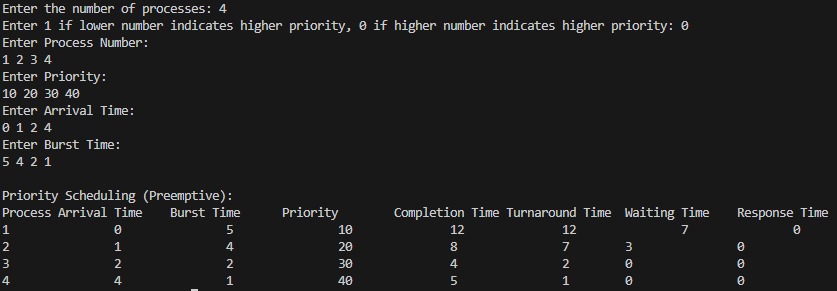
int wt[n], tat[n], ct[n], rt[n];

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

findCompletionTime(processes, n, burst\_time, arrival\_time, wt, tat, ct, rt, priority, isLowerPriorityHigher);

}

**Result:**



**2.Priority(Non-pre-emptive)**

#include<stdio.h>

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

{

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

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

{

min = p[i];

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

{

if (p[j] < min)

{

temp = at[i];

at[i] = at[j];

at[j] = temp;

temp = bt[j];

bt[j] = bt[i];

bt[i] = temp;

temp = p[j];

p[j] = p[i];

p[i] = temp;

temp = proc\_id[i];

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

proc\_id[j] = temp;

}

}

}

}

void main ()

{

int n, c = 0;

printf ("Enter number of processes: ");

scanf ("%d", &n);

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

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

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

{

proc\_id[i] = i + 1;

m[i] = 0;

}

printf ("Enter priorities:\n");

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

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

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

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

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

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

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

{

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

m[i] = -1;

rt[i] = -1;

}

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

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

int x = 0;

c = 0;

while (count < n)

{

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

{

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

{

x = i;

priority = p[i];

}

}

if (rt[x] == -1)

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

if (at[x] <= c)

c += bt[x];

else

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

count++;

ct[x] = c;

m[x] = 1;

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

{

priority = p[x];

break;

}

x++;

if (count == n)

break;

}

//turnaround time and RT

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

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

//waiting time

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

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

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

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

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

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

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

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

{

ttat += tat[i];

twt += wt[i];

}

avg\_tat = ttat / (double) n;

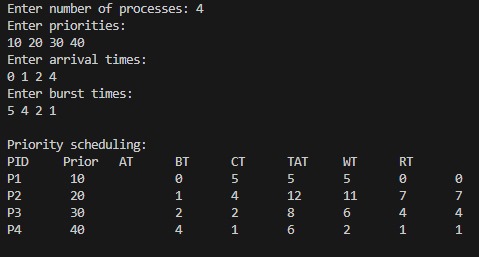
avg\_wt = twt / (double) n;

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

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

}

**Result:**



**3.Round Robin**

#include <stdio.h>

#include <stdbool.h>

void findCompletionTime(int processes[], int n, int bt[], int at[], int wt[], int tat[], int ct[], int rt[], int quantum)

{

int remaining[n]; // Array to store remaining burst time of processes

bool firstResponse[n]; // Array to track if response time has been set

int currentTime = 0; // Current time

int completed = 0; // Counter for completed processes

// Initialize remaining array with burst times and first response array

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

remaining[i] = bt[i];

firstResponse[i] = true;

}

// Queue to hold the indices of the processes

int queue[n];

int front = -1, rear = -1;

// Function to add process to the queue

void enqueue(int process) {

if (rear == n - 1) rear = -1;

queue[++rear] = process;

if (front == -1)

front = 0;

}

// Function to remove process from the queue

int dequeue() {

int process = queue[front];

if (front == rear)

front = rear = -1;

else {

front++;

if (front == n)

front = 0;

}

return process;

}

// To track which processes have been added to the queue

bool inQueue[n];

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

inQueue[i] = false;

while (completed < n) {

// Add all processes to the queue that have arrived by the current time

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

if (at[i] <= currentTime && !inQueue[i]) {

enqueue(i);

inQueue[i] = true;

}

}

// If no process is ready, increment the current time

if (front == -1) {

currentTime++;

continue;

}

int currentProcess = dequeue();

// Set response time if it's the first time the process is executed

if (firstResponse[currentProcess]) {

rt[currentProcess] = currentTime - at[currentProcess];

firstResponse[currentProcess] = false;

}

// Execute the process for the time quantum or until completion

if (remaining[currentProcess] > quantum) {

remaining[currentProcess] -= quantum;

currentTime += quantum;

} else {

currentTime += remaining[currentProcess];

remaining[currentProcess] = 0;

completed++;

// Set completion time for the process

ct[currentProcess] = currentTime;

// Calculate waiting time and turnaround time for the process

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

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

}

// Add all processes to the queue that have arrived by the current time

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

if (at[i] <= currentTime && !inQueue[i]) {

enqueue(i);

inQueue[i] = true;

}

}

// Re-enqueue the current process if it is not finished

if (remaining[currentProcess] > 0) {

enqueue(currentProcess);

}

}

// Print the table

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

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

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",

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

}

}

void main()

{

// Number of processes

int n;

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

scanf("%d", &n);

// Process id's

int processes[n];

// Burst time of all processes

int burst\_time[n];

// Arrival time of all processes

int arrival\_time[n];

printf("Enter Process Number:\n");

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

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

}

printf("Enter Arrival Time:\n");

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

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

}

printf("Enter Burst Time:\n");

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

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

}

// Time quantum for Round Robin

int quantum;

printf("Enter the time quantum: ");

scanf("%d", &quantum);

// Arrays to store waiting time, turnaround time, completion time, and response time

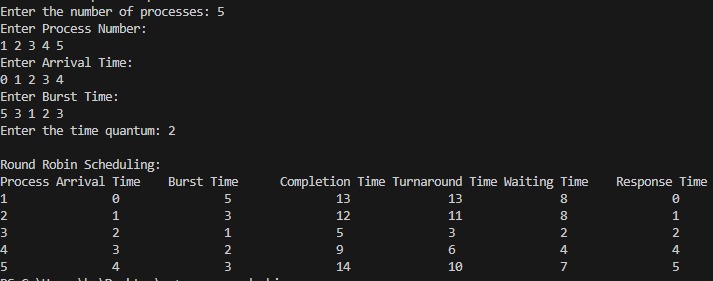
int wt[n], tat[n], ct[n], rt[n];

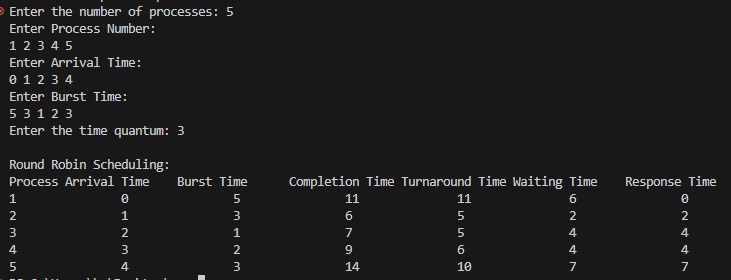
printf("\nRound Robin Scheduling:\n");

findCompletionTime(processes, n, burst\_time, arrival\_time, wt, tat, ct, rt, quantum);

}

**Result:**





**Program -3**

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

**Code:**

#include<stdio.h>

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

{

int temp=0;

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

{

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

{

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

{

temp=at[i];at[i]=at[j];at[j]=temp;

temp=bt[j];bt[j]=bt[i];bt[i]=temp;

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

}

}

}

}

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

{

double ttat=0.0,twt=0.0;

//completion time

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

{

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

\*c+=bt[i];

else

\*c+=at[i]-ct[i-1]+bt[i];

ct[i]=\*c;

}

//turnaround time

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

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

//waiting time

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

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

}

void main()

{

int sn,un,c=0;int n=0;

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

scanf("%d",&sn);n=sn;

int sproc\_id[n],sat[n],sbt[n],sct[n],stat[n],swt[n];

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

sproc\_id[i]=i+1;

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

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

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

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

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

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

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

scanf("%d",&un);n=un;

int uproc\_id[n],uat[n],ubt[n],uct[n],utat[n],uwt[n];

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

uproc\_id[i]=i+1;

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

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

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

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

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

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

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

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

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

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

printf("\nScheduling:\n");

printf("System processes:\n");

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

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

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

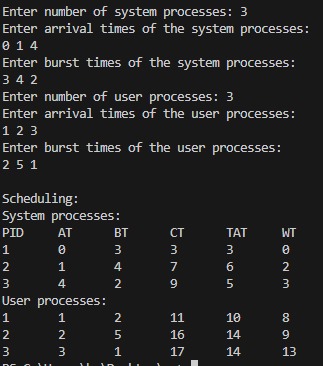
printf("User processes:\n");

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

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

}

**Result:**



**Program -4**

**Question:** Write a C program to simulate Real-Time CPU Scheduling algorithms: a) Rate- Monotonic b) Earliest-deadline First c) Proportional scheduling

**Code:**

**1.Rate-Monotonic**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

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

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

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

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

int temp = proc[i];

proc[i] = proc[j];

proc[j] = temp;

temp = b[i];

b[i] = b[j];

b[j] = temp;

temp = pt[i];

pt[i] = pt[j];

pt[j] = temp;

}}}}

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

int lcm = p[0];

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

int a = lcm, b = p[i];

while (b != 0) {

int r = a % b;

a = b;

b = r;}

lcm = (lcm \* p[i]) / a;}

return lcm;}

int main() {

int n;

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

scanf("%d", &n);

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

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

for (int i = 0; i < n; i++) scanf("%d", &b[i]);

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

for (int i = 0; i < n; i++) scanf("%d", &pt[i]);

for (int i = 0; i < n; i++) proc[i] = i + 1;

sort(proc, b, pt, n);

int l = lcmul(pt, n);

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

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

for (int i = 0; i < n; i++) printf("%d\t\t%d\t\t%d\n", proc[i], b[i], pt[i]);

double sum = 0.0;

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

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

}

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

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

if (sum > rhs) {

printf("The given set of processes is not schedulable.\n");

exit(0);

}

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

int time = 0, prev = -1;

for (int i = 0; i < n; i++) rem[i] = b[i];

int nextRelease[n];

for (int i = 0; i < n; i++) nextRelease[i] = 0;

while (time < l) {

int taskToExecute = -1;

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

if (time == nextRelease[i]) {

rem[i] = b[i]; // Reset remaining time at the start of the period

nextRelease[i] += pt[i]; // Schedule next release

}

if (rem[i] > 0 && (taskToExecute == -1 || pt[i] < pt[taskToExecute])) {

taskToExecute = i;}}

if (taskToExecute != -1) {

if (prev != taskToExecute) {

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

prev = taskToExecute; }

rem[taskToExecute]--;

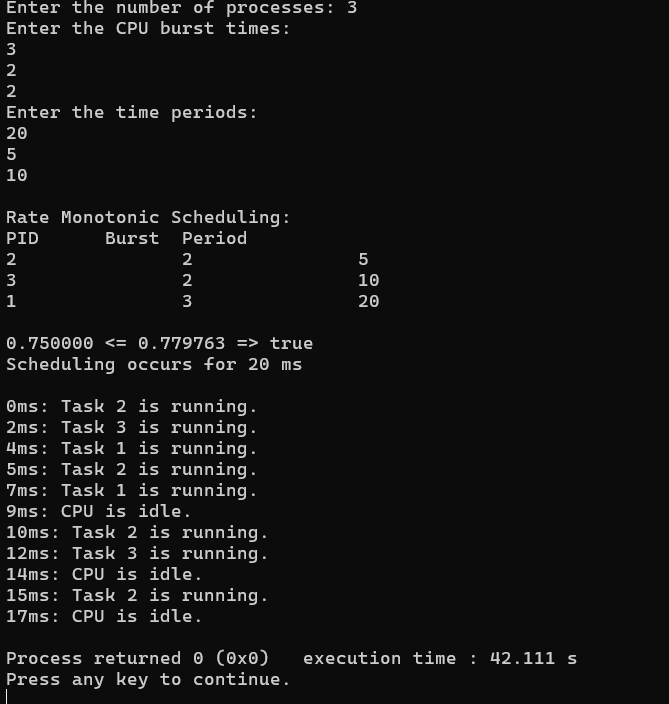
} else if (prev != -1) {

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

prev = -1; }

time++; } return 0;}

**Result:**



**2.Earliest Deadline First**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

void

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

{

int temp = 0;

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

{

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

{

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

{

temp = d[j];

d[j] = d[i];

d[i] = temp;

temp = pt[i];

pt[i] = pt[j];

pt[j] = temp;

temp = b[j];

b[j] = b[i];

b[i] = temp;

temp = proc[i];

proc[i] = proc[j];

proc[j] = temp;

}

}

}

}

int

gcd (int a, int b)

{

int r;

while (b > 0)

{

r = a % b;

a = b;

b = r;

}

return a;

}

int

lcmul (int p[], int n)

{

int lcm = p[0];

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

{

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

}

return lcm;

}

Void main ()

{

int n;

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

scanf ("%d", &n);

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

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

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

{

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

rem[i] = b[i];

}

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

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

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

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

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

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

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

proc[i] = i + 1;

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

int l = lcmul (pt, n);

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

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

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

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

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

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

int nextDeadlines[n];

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

{

nextDeadlines[i] = d[i];

rem[i] = b[i];

}

while (time < l)

{

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

{

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

{

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

rem[i] = b[i];

}

}

int minDeadline = l + 1;

int taskToExecute = -1;

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

{

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

{

minDeadline = nextDeadlines[i];

taskToExecute = i;

}

}

if (taskToExecute != -1)

{

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

rem[taskToExecute]--;

}

else

{

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

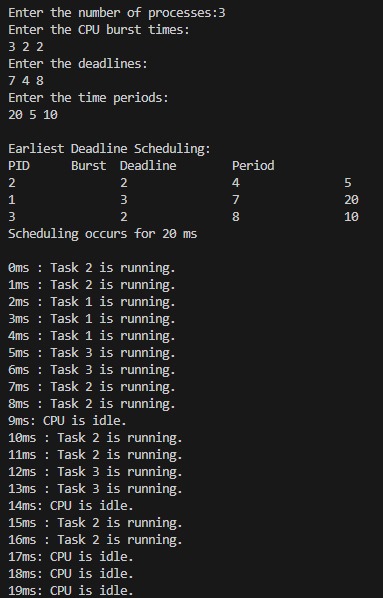
}

time++;

}

}

**Result:**



**3.Proportional**

#include<stdio.h>

#include<stdlib.h>

void sort\_by\_burst(int proc[], int burst[], int tickets[], int remaining[], int n) {

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

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

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

// Swap process IDs

int temp = proc[j];

proc[j] = proc[j + 1];

proc[j + 1] = temp;

// Swap burst times

temp = burst[j];

burst[j] = burst[j + 1];

burst[j + 1] = temp;

// Swap tickets

temp = tickets[j];

tickets[j] = tickets[j + 1];

tickets[j + 1] = temp;

// Swap remaining burst times

temp = remaining[j];

remaining[j] = remaining[j + 1];

remaining[j + 1] = temp;

}

}

}

}

int main() {

int n;

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

scanf("%d", &n);

int proc[n], burst[n], tickets[n], remaining[n];

int total\_tickets = 0;

printf("Enter the CPU burst times and number of tickets for each process:\n");

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

proc[i] = i + 1; // Process IDs

printf("Process %d burst time: ", i + 1);

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

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

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

remaining[i] = burst[i];

total\_tickets += tickets[i];

}

sort\_by\_burst(proc, burst, tickets, remaining, n);

printf("\nProportional Share Scheduling (Lottery Scheduling):\n");

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

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

printf("%d\t %d\t %d\n", proc[i], burst[i], tickets[i]);

}

int time = 0;

while (1) {

int active\_processes = 0;

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

if (remaining[i] > 0) {

active\_processes++;

}

}

if (active\_processes == 0) {

break;

}

int winning\_ticket = rand() % total\_tickets;

int cumulative\_tickets = 0;

int selected\_process = -1;

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

if (remaining[i] > 0) {

cumulative\_tickets += tickets[i];

if (winning\_ticket < cumulative\_tickets) {

selected\_process = i;

break;

}

}

}

if (selected\_process != -1) {

printf("%dms: Process %d is running.\n", time, proc[selected\_process]);

remaining[selected\_process]--;

} else {

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

}

time++;

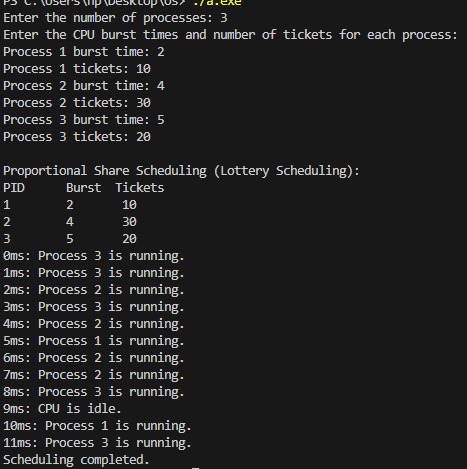
}

printf("Scheduling completed.\n");

return 0;

}

**Result:**



**Program -5**

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

**Code:**

#include<stdio.h>

#include<stdlib.h>

int mutex = 1, full = 0, empty = 7, x = 0;

int main() {

int n;

void producer();

void consumer();

int wait(int);

int signal(int);

printf("\n1.Producer\n2.Consumer\n3.Exit");

while (1) {

printf("\nEnter your choice:");

scanf("%d", &n);

switch (n) {

case 1:

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

producer();

else

printf("Buffer is full!!");

break;

case 2:

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

consumer();

else

printf("Buffer is empty!!");

break;

case 3:

printf("\nNumber of Items remaining in buffer: %d\n", x);

exit(0);

break;

}

}

return 0;

}

int wait(int s) {

return (--s);

}

int signal(int s) {

return (++s);

}

void producer() {

mutex = wait(mutex);

full = signal(full);

empty = wait(empty);

x++;

printf("\nProducer produces the item %d", x);

mutex = signal(mutex);

}

void consumer() {

mutex = wait(mutex);

full = wait(full);

empty = signal(empty);

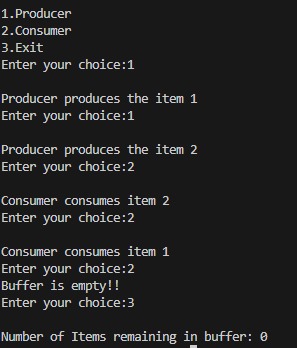
printf("\nConsumer consumes item %d", x);

x--;

mutex = signal(mutex);

}

**Result:**



**Program -6**

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

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <math.h>

#define THINKING 0

#define HUNGRY 1

#define EATING 2

#define N 5

#define MAX\_PHILOSOPHERS 5

bool spoon[N] = {true, true, true, true, true}; // true means the spoon is available

int state[N] = {THINKING, THINKING, THINKING, THINKING, THINKING};

bool wait(int i)

{

return !spoon[i]; // if spoon is not available, return true

}

void signal(int i)

{

spoon[i] = true; // make the spoon available

}

void take\_spoon(int i)

{

spoon[i] = false; // take the spoon

}

void test(int i)

{

if (state[i] == HUNGRY && !wait(i) && !wait((i + 1) % N))

{

state[i] = EATING;

take\_spoon(i);

take\_spoon((i + 1) % N);

printf("Philosopher %d is Granted to Eat\n", i + 1);

}

}

void put\_spoon(int i)

{

signal(i);

signal((i + 1) % N);

state[i] = THINKING;

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

}

int completed()

{

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

{

if (state[i] != THINKING)

return 0;

}

printf("Dinner completed\n");

return 1;

}

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

{

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

{

state[hungry[i]-1]=HUNGRY;

}

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

{

int pos = hungry[i] - 1;

test(pos);

if (state[pos] == EATING)

{

put\_spoon(pos);

}

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

{

if (state[hungry[j] - 1] == HUNGRY)

{

printf("Philosopher %d is Waiting\n", hungry[j]);

}

}

}

}

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

{

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

{

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

return;

}

else if(n==2)

{

if(abs(hungry[0]-hungry[1])==1)

{

printf("Not possible");

exit(0);

}

printf("P %d and P %d are granted to eat\n", hungry[0], hungry[1]);

}

else{

int combination\_count = 1,a[n-2];

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

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

int p=0;

if(abs(hungry[i]-hungry[j])==1)

{

continue;

}

printf("\n\n\ncombination %d\n\n", combination\_count);

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

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

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

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

a[p]=k;

p++;

}

}

printf("\n");

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

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

int com=1;

if(abs(a[0]-a[1])!=1 && n>3)

{

if(n==5)

{

printf("Combination %d.%d\n",combination\_count,com);

}

printf("P %d and P %d are granted to eat\n", hungry[a[0]], hungry[a[1]]);

if(n==5)

{

printf("P %d is waiting\n", hungry[a[2]]);

}

printf("P %d is waiting\n", hungry[a[0]]);

printf("P %d is waiting\n", hungry[a[1]]);

com++;

}

if(abs(a[2]-a[1])!=1 && n>4)

{

printf("Combination %d.%d\n",combination\_count,com);

printf("P %d and P %d are granted to eat\n", hungry[a[2]], hungry[a[1]]);

printf("P %d is waiting\n", hungry[a[0]]);

printf("P %d is waiting\n", hungry[a[1]]);

printf("P %d is waiting\n", hungry[a[2]]);

com++;

}

if(abs(a[0]-a[2])!=1 && n>4)

{

printf("Combination %d.%d\n",combination\_count,com);

printf("P %d and P %d are granted to eat\n", hungry[a[0]], hungry[a[2]]);

printf("P %d is waiting\n", hungry[a[1]]);

printf("P %d is waiting\n", hungry[a[0]]);

printf("P %d is waiting\n", hungry[a[2]]);

}

combination\_count++;

}

}

}

}

int main()

{

while (1)

{

int total\_philosophers, hungry\_count;

int hungry\_positions[MAX\_PHILOSOPHERS];

printf("DINING PHILOSOPHER PROBLEM\n");

printf("Enter the total no. of philosophers: %d\n", N);

total\_philosophers = N;

printf("How many are hungry: ");

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

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

{

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

return 1;

}

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

{

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

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

if (hungry\_positions[i] < 1 || hungry\_positions[i] > total\_philosophers)

{

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

return 1;

}

}

int choice;

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

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

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice)

{

case 1:

printf("Allow one philosopher to eat at any time\n");

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

break;

case 2:

printf("Allow two philosophers to eat at the same time\n");

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

break;

case 3:

exit(0);

default:

printf("Invalid choice\n");

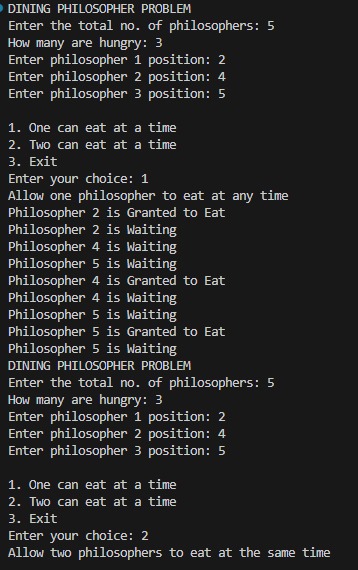
}

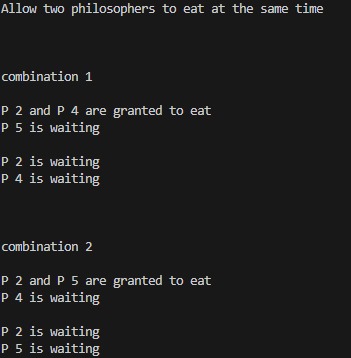
}

return 0;

}

**Result:**





**Program -7**

**Question:** Write a C program to simulate Banker’s algorithm for the purpose of deadlock avoidance.

**Code:**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

void calculateNeed(int need[MAX\_PROCESSES][MAX\_RESOURCES], int max[MAX\_PROCESSES][MAX\_RESOURCES], int allot[MAX\_PROCESSES][MAX\_RESOURCES], int np, int nr) {

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

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

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

}

bool isSafe(int processes[], int avail[], int max[][MAX\_RESOURCES], int allot[][MAX\_RESOURCES], int np, int nr) {

int need[MAX\_PROCESSES][MAX\_RESOURCES];

calculateNeed(need, max, allot, np, nr);

bool finish[MAX\_PROCESSES] = {0};

int safeSeq[MAX\_PROCESSES];

int work[MAX\_RESOURCES];

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

work[i] = avail[i];

int count = 0;

while (count < np) {

bool found = false;

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

if (finish[p] == 0) {

int j;

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

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

break;

if (j == nr) {

for (int k = 0; k < nr; k++)

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

safeSeq[count++] = p;

finish[p] = 1;

found = true;

}

}

}

if (found == false) {

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

return false;

}

}

printf("System is in safe state.\nSafe sequence is: ");

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

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

printf("\n");

return true;

}

void printResourceAllocationDetails(int np, int nr, int processes[], int max[][MAX\_RESOURCES], int allot[][MAX\_RESOURCES], int avail[]) {

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

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

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

// Print Allocation

printf("\t");

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

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

// Print Max

printf("\t\t");

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

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

// Print Need

printf("\t");

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

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

// Print Available

if (i == 0) {

printf("\t");

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

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

}

printf("\n");

}

}

int main() {

int np, nr;

printf("Enter number of processes: ");

scanf("%d", &np);

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

scanf("%d", &nr);

int processes[MAX\_PROCESSES];

for (int i = 0; i < np; i++) processes[i] = i;

int avail[MAX\_RESOURCES];

printf("Enter available resources: ");

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

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

int max[MAX\_PROCESSES][MAX\_RESOURCES];

printf("Enter maximum resource matrix:\n");

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

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

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

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

}

int allot[MAX\_PROCESSES][MAX\_RESOURCES];

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

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

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

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

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

}

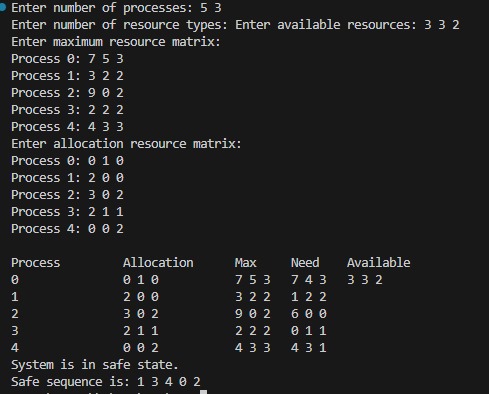
printResourceAllocationDetails(np, nr, processes, max, allot, avail);

isSafe(processes, avail, max, allot, np, nr);

return 0;

}

**Result:**



**Program -8**

**Question:** Write a C program to simulate deadlock detection

**Code:**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

bool isSafe(int processes[], int avail[], int req[][MAX\_RESOURCES], int allot[][MAX\_RESOURCES], int np, int nr) {

int need[MAX\_PROCESSES][MAX\_RESOURCES];

bool finish[MAX\_PROCESSES] = {0};

int safeSeq[MAX\_PROCESSES];

int work[MAX\_RESOURCES];

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

work[i] = avail[i];

int count = 0,d;

while (count < np) {

bool found = false;

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

if (finish[p] == 0) {

int j;

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

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

{

d=p;

break;

}

if (j == nr) {

for (int k = 0; k < nr; k++)

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

safeSeq[count++] = p;

finish[p] = 1;

found = true;

}

}

}

if (found == false) {

printf("Deadlock Detected at processes <",d);

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

if (finish[i] == 0)

printf("%d ", i);

printf(">");

return false;

}

}

printf("System is in safe state.\nSafe sequence is: ");

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

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

printf("\n");

return true;

}

void printResourceAllocationDetails(int np, int nr, int processes[], int req[][MAX\_RESOURCES], int allot[][MAX\_RESOURCES], int avail[]) {

printf("\nProcess\t\tAllocation\tRequest\t\tAvailable\n");

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

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

// Print Allocation

printf("\t");

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

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

// Print Max

printf("\t\t");

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

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

// Print Available

if (i == 0) {

printf("\t\t");

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

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

}

printf("\n");

}

}

int main() {

int np, nr;

printf("Enter number of processes: ");

scanf("%d", &np);

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

scanf("%d", &nr);

int processes[MAX\_PROCESSES];

for (int i = 0; i < np; i++) processes[i] = i;

int avail[MAX\_RESOURCES];

printf("Enter available resources: ");

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

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

int req[MAX\_PROCESSES][MAX\_RESOURCES];

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

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

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

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

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

}

int allot[MAX\_PROCESSES][MAX\_RESOURCES];

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

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

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

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

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

}

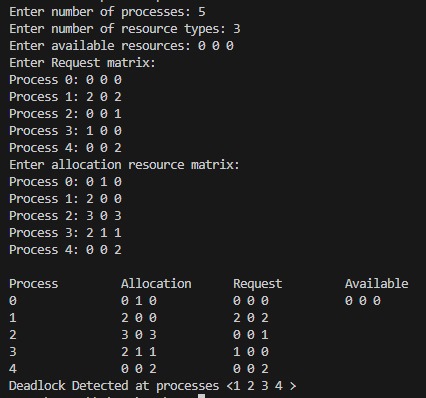
printResourceAllocationDetails(np, nr, processes, req, allot, avail);

isSafe(processes, avail, req, allot, np, nr);

return 0;

}

**Result:**



**Program -9**

**Question:** Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit b) Best-fit c) First-fit

**Code:**

#include <stdio.h>

#define MAX 25

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

int frag[MAX], bf[MAX] = {0}, ff[MAX] = {0};

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;

frag[i] = temp;

bf[j] = 1;

break;

}

}

}

}

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

printf("File\_no:\tFile\_size :\tBlock\_no:\tBlock\_size:\tFragment\n");

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

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

if (ff[i] != 0) {

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

} else {

printf("Not Allocated\n");

}

}

}

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

int frag[MAX], bf[MAX] = {0}, ff[MAX] = {0};

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 && lowest > temp) {

ff[i] = j;

lowest = temp;

}

}

}

frag[i] = lowest;

bf[ff[i]] = 1;

lowest = 10000;

}

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

printf("File No\tFile Size \tBlock No\tBlock Size\tFragment\n");

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

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

if (ff[i] != 0) {

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

} else {

printf("Not Allocated\n");

}

}

}

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

int frag[MAX], bf[MAX] = {0}, ff[MAX] = {0};

int i, j, temp, highest = 0;

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

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

if (bf[j] != 1) {

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

if (temp >= 0 && highest < temp) {

ff[i] = j;

highest = temp;

}

}

}

frag[i] = highest;

bf[ff[i]] = 1;

highest = 0;

}

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

printf("File\_no:\tFile\_size :\tBlock\_no:\tBlock\_size:\tFragment\n");

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

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

if (ff[i] != 0) {

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

} else {

printf("Not Allocated\n");

}

}

}

int main() {

int b[MAX], f[MAX], nb, nf;

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

scanf("%d", &nb);

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

scanf("%d", &nf);

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

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

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

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

}

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

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

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

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

}

int b1[MAX], b2[MAX], b3[MAX];

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

b1[i] = b[i];

b2[i] = b[i];

b3[i] = b[i];

}

firstFit(nb, nf, b1, f);

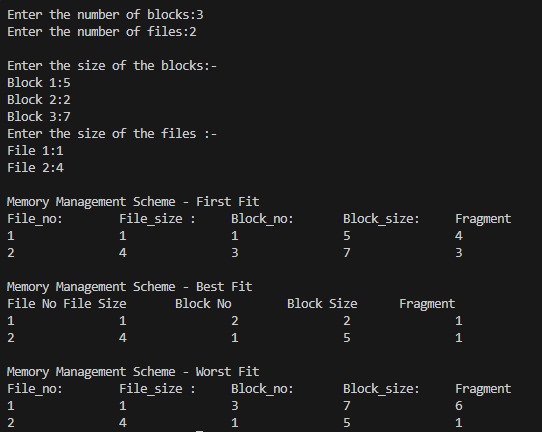
bestFit(nb, nf, b2, f);

worstFit(nb, nf, b3, f);

return 0;

}

**Result:**



**Program -10**

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

**Code:**

#include <stdio.h>

#include <stdlib.h>

void printFrames(int frames[], int n, int faults, int x) {

printf("[");

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

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

if (x==1)

printf("] Fault %d \n", faults);

else

printf("] \n");

}

void FIFO(int pages[], int n, int frame\_size) {

int frames[frame\_size];

for (int i = 0; i < frame\_size; i++) frames[i] = -1;

int index = 0, faults = 0;

printf("FIFO: \n");

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

{

int found=0;

int x=0;

for (int j = 0; j < frame\_size; j++) {

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

found = 1;

break;

}

}

if (!found) {

frames[index] = pages[i];

index = (index + 1) % frame\_size;

faults++;

x=1;

}

printFrames(frames, frame\_size, faults, x);

}

printf("Total faults: %d\n", faults);

}

void LRU(int pages[], int n, int frame\_size) {

int frames[frame\_size], last\_used[frame\_size];

for (int i = 0; i < frame\_size; i++) frames[i] = -1;

for (int i = 0; i < frame\_size; i++) last\_used[i] = 0;

int time = 0, faults = 0;

printf("LRU: \n");

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

{

int x=0;

int found = 0;

for (int j = 0; j < frame\_size; j++) {

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

found = 1;

last\_used[j] = ++time;

break;

}

}

if (!found) {

int lru\_index = 0;

for (int j = 1; j < frame\_size; j++) {

if (last\_used[j] < last\_used[lru\_index])

lru\_index = j;

}

frames[lru\_index] = pages[i];

last\_used[lru\_index] = ++time;

faults++;

x=1;

}

printFrames(frames, frame\_size, faults, x);

}

printf("Total faults: %d\n", faults);

}

void OPTIMAL(int pages[], int n, int frame\_size) {

int frames[frame\_size];

for (int i = 0; i < frame\_size; i++)

frames[i] = -1;

int faults = 0;

printf("OPTIMAL: \n");

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

{

int x=0;

int found = 0;

for (int j = 0; j < frame\_size; j++) {

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

found = 1;

break;

}

}

for(int p = 0; p < frame\_size; p++)

{

if(frames[p]==-1 && found==0)

{

found=1;

frames[p] = pages[i];

faults++;

x=1;

break;

}

}

if (!found) {

int replace\_index = 0, farthest = -1;

for (int j = 0; j < frame\_size; j++)

{

int k;

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

{

if (frames[j] == pages[k])

{

if (k > farthest)

{

farthest = k;

replace\_index = j;

}

break;

}

}

if (k == n) {

replace\_index = j;

break;

}

}

frames[replace\_index] = pages[i];

faults++;

x=1;

}

printFrames(frames, frame\_size, faults, x);

}

printf("Total faults: %d\n", faults);

}

int main() {

int n, frame\_size;

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

scanf("%d", &n);

int \*pages = (int \*)malloc(n \* sizeof(int));

printf("Enter the page sequence: ");

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

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

}

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

scanf("%d", &frame\_size);

FIFO(pages, n, frame\_size);

LRU(pages, n, frame\_size);

OPTIMAL(pages, n, frame\_size);

free(pages);

return 0;

}

**Result:**

