

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT
on

OPERATING SYSTEMS **(23CS4PCOPS)**

Submitted by

KAVANA PATIL(1BM22CS124)

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 **KAVANA PATIL(1BM22CS124)**, who is bonafide student of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2024. The Lab report has been approved as it satisfies the academic requirements in respect of a **OPERATING SYSTEMS - (23CS4PCOPS)** work prescribed for the said degree.

Dr. Radhika A D
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Sl. No.	Experiment Title	Page No.
1.	Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time. →FCFS → SJF (pre-emptive & Non-preemptive)	4
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)	13
3.	Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue.	19
4.	Write a C program to simulate Real-Time CPU Scheduling algorithms: a) Rate-Monotonic b) Earliest-deadline First c) Proportional scheduling	22
5.	Write a C program to simulate producer-consumer problem using semaphores.	31
6.	Write a C program to simulate the concept of Dining-Philosophers problem.	33
7.	Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.	37
8.	Write a C program to simulate deadlock detection	41
9.	Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit b) Best-fit c) First-fit	44
10.	Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) Optimal	47

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

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

a.FCFS scheduling using array

```
#include<stdio.h>

int main(){

    int n,i;

    float atat=0,awt=0;

    printf("enter the number of process");

    scanf("%d",&n);

    int atime1[n],btime2[n],ctime3[n],tattime4[n],wtime5[n];

    printf("enter arrival time of process");

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

        scanf("%d",&atime1[i]);

    }

    printf("enter burst time of process");

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

        scanf("%d",&btime2[i]);

    }

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

        if(i==0){

            ctime3[i]=atime1[i]+btime2[i];

        }

        else{

            if(ctime3[i-1]<atime1[i]){

                ctime3[i]=(atime1[i]-ctime3[i-1])+ctime3[i-1]+btime2[i];

            }

            else{

                ctime3[i]=ctime3[i-1]+btime2[i];

            }

        }

    }

}
```

```

    }
}
for(i=0;i<n;i++){
    tatetime4[i]=ctime3[i]-atime1[i];
}
for(i=0;i<n;i++){
    wtime5[i]=tatetime4[i]-btime2[i];
}
for(i=0;i<n;i++){
    atat=atat+tatetime4[i];
}
atat=(atat/n);
for(i=0;i<n;i++){
    awt=awt+vertime5[i];
}
awt=(awt/n);
for(i=0;i<n;i++){
    printf("process id %d arrival time %d burst time %d complete time %d turn around time %d\n",i+1,atime1[i],btime2[i],ctime3[i],tatetime4[i],vertime5[i]);
}
printf("average turn around time is %f",atat);
printf("average working time is %f",awt);
}

```

output:

```
C:\Users\saisr\OneDrive\Desk x + v
enter the number of process4
enter arrival time of process0
1
5
6
enter burst time of process2
2
3
4
process id 1 arrival time 0 burst time 2 complete time 2 turn around time 2 waiting time 0
process id 2 arrival time 1 burst time 2 complete time 4 turn around time 3 waiting time 1
process id 3 arrival time 5 burst time 3 complete time 8 turn around time 3 waiting time 0
process id 4 arrival time 6 burst time 4 complete time 12 turn around time 6 waiting time 2
average turn around time is 3.500000average working time is 0.750000
Process returned 0 (0x0) execution time : 141.043 s
Press any key to continue.
```

b.SJF(non-preemptive)scheduling using array

```
#include<stdio.h>
```

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

```
{
```

```
int completion[n];
```

```
int remaining[n];
```

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

```
remaining[i] = bt[i];
```

```
int currentTime = 0;
```

```
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];
remaining[shortest] = 0;
}

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

for (int i = 0; i < n; i++)
{
ct[i] = completion[i];
printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n", processes[i], at[i], bt[i], wt[i], tat[i], rt[i], ct[i]);
}

float avg_tat=tat[0];
for(int i=1;i<n;i++)
{
avg_tat+=tat[i];
}

printf("\n Average TAT=%f ms",avg_tat/n);

float avg_wt=wt[0];
for(int i=1;i<n;i++)
{
avg_wt+=wt[i];
}

printf("\n Average WT= %f ms",avg_wt/n);

```

```

}

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);
}

```

output:


```
C:\Users\saisr\OneDrive\Desk x + v
Enter the number of processes: 4
Enter Process Number:
1
2
3
4
Enter Arrival Time:
0
0
0
0
Enter Burst Time:
6
8
7
3

SJF (Non-preemptive) Scheduling:
Process Arrival Time Burst Time Waiting Time Turnaround Time ResponseTime Completion Time
1 0 6 3 9 3 9
2 0 8 16 24 16 24
3 0 7 9 16 9 16
4 0 3 0 3 0 3

Average TAT=13.000000 ms
Average WT= 7.000000 ms
Process returned 25 (0x19) execution time : 57.452 s
Press any key to continue.
|
```

c.SJF(preemptive)scheduling using array

```
#include <stdio.h>
```

```
#define MAX 10
```

```
int find_min(int arr[], int n) {
```

```
    int min = arr[0];
```

```
    int index = 0;
```

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

```
        if (arr[i] < min) {
```

```
            min = arr[i];
```

```
            index = i;
```

```
        }
```

```
    }
```

```
    return index;
```

```
}
```

```
void sjf_preemptive(int n, int at[], int bt[]) {
```

```
    int ct[MAX] = {0};
```

```
    int tat[MAX] = {0};
```

```
    int wt[MAX] = {0};
```

```
    int rt[MAX];
```

```
    int total_wt = 0;
```

```

int total_tat = 0;
for (int i = 0; i < n; i++) {
    rt[i] = bt[i];
}

int current_time = 0;
int completed_processes = 0;
while (completed_processes < n) {
    int available_processes[MAX];
    int available_count = 0;
    for (int i = 0; i < n; i++) {
        if (at[i] <= current_time && rt[i] > 0) {
            available_processes[available_count] = i;
            available_count++;
        }
    }
    if (available_count == 0) {
        current_time++;
        continue;
    }
    int shortest_job_index = available_processes[find_min(rt, available_count)];
    rt[shortest_job_index]--;
    current_time++;
    if (rt[shortest_job_index] == 0) {
        completed_processes++;
        ct[shortest_job_index] = current_time;
        tat[shortest_job_index] = ct[shortest_job_index] - at[shortest_job_index];
        wt[shortest_job_index] = tat[shortest_job_index] - bt[shortest_job_index];
        total_wt += wt[shortest_job_index];
        total_tat += tat[shortest_job_index];
    }
}

```

```

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

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

        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", i+1, at[i], bt[i], ct[i], tat[i], wt[i]);

    }

    printf("\nAverage waiting time: %.2f", (float)total_wt / n);

    printf("\nAverage turnaround time: %.2f", (float)total_tat / n);

}

int main() {

    int n;

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

    scanf("%d", &n);

    int at[MAX], bt[MAX];

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

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

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

    }

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

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

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

    }

    sjf_preemptive(n, at, bt);

    return 0;

}

```

output:

```
C:\Users\saisr\OneDrive\Desk  x + v
Enter the number of processes: 5
Enter the arrival time:
2
1
4
0
2
Enter the burst time:
1
5
1
6
3

Process Arrival Time    Burst Time    Completion Time Turnaround Time Waiting Time
1      2              1              3              1              0
2      1              5              7              6              1
3      4              1              8              4              3
4      0              6              13             13              7
5      2              3              16             14             11

Average waiting time: 4.40
Average turnaround time: 7.60
Process returned 0 (0x0)  execution time : 162.144 s
Press any key to continue.
```

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)

a.priority(non preemptive)scheduling

```
#include<stdio.h>
```

```
#define MAX 10
```

```
void priority_non_preemptive(int n, int at[], int bt[], int p[]) {
```

```
    int ct[MAX] = {0};
```

```
    int tat[MAX] = {0};
```

```
    int wt[MAX] = {0};
```

```
    int total_wt = 0;
```

```
    int total_tat = 0;
```

```
    int bt_copy[MAX];
```

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

```
        bt_copy[i] = bt[i];
```

```
    }
```

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

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

```
            if (p[i] < p[j]) {
```

```
                int temp = at[i];
```

```
                at[i] = at[j];
```

```
                at[j] = temp;
```

```
                temp = bt[i];
```

```
                bt[i] = bt[j];
```

```
                bt[j] = temp;
```

```
                temp = p[i];
```

```
                p[i] = p[j];
```

```
                p[j] = temp;
```

```
            }
```

```
        }
```

```
    }
```

```
    ct[0] = at[0] + bt[0];
```

```

    tat[0] = ct[0] - at[0];
    wt[0] = tat[0] - bt_copy[0];
    total_wt += wt[0];
    total_tat += tat[0];
    for (int i = 1; i < n; i++) {
        ct[i] = ct[i - 1] + bt[i];
        tat[i] = ct[i] - at[i];
        wt[i] = tat[i] - bt_copy[i];
        total_wt += wt[i];
        total_tat += tat[i];
    }

    printf("\nProcess\tArrival Time\tBurst Time\tPriority\tCompletion Time\tTurnaround
Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n", i+1, at[i], bt_copy[i], p[i], ct[i], tat[i], wt[i]);
    }

    printf("\nAverage waiting time: %.2f", (float)total_wt / n);
    printf("\nAverage turnaround time: %.2f", (float)total_tat / n);
}

int main() {
    int n;

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

    scanf("%d", &n);

    int at[MAX], bt[MAX], p[MAX];

    printf("Enter the arrival time:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &at[i]);
    }

    printf("Enter the burst time:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &bt[i]);
    }

```

```

}

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

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

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

}

priority_non_preemptive(n, at, bt, p);

return 0;

}

```

output:

```

C:\Users\saisr\OneDrive\Desk x + v
Enter the number of processes: 4
Enter the arrival time:
0
1
2
4
Enter the burst time:
5
4
2
1
Enter the priority:
10
20
30
40

Process Arrival Time    Burst Time    Priority    Completion Time    Turnaround Time    Waiting Time
1         4             5             40          5                  1                 -4
2         2             4             30          7                  5                  1
3         1             2             20         11                 10                 8
4         0             1             10         16                 16                 15

Average waiting time: 5.00
Average turnaround time: 8.00
Process returned 0 (0x0)   execution time : 108.063 s
Press any key to continue.

```

b.Round robin scheduling

```

#include <stdio.h>

struct Process {

    int pid;

    int burst_time;

    int arrival_time;

    int remaining_time;

};

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

    int remaining_processes = n;

    int current_time = 0;

```

```

int completed[n];

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

for (int i = 0; i < n; i++) {
    completed[i] = 0;
}

while (remaining_processes > 0) {
    for (int i = 0; i < n; i++) {
        if (completed[i] == 0 && processes[i].arrival_time <= current_time) {
            if (processes[i].remaining_time > 0) {
                if (processes[i].remaining_time <= time_quantum) {
                    current_time += processes[i].remaining_time;
                    processes[i].remaining_time = 0;
                    completed[i] = 1;
                    remaining_processes--;
                    ct[i] = current_time;
                    tat[i] = ct[i] - processes[i].arrival_time;
                } else {
                    current_time += time_quantum;
                    processes[i].remaining_time -= time_quantum;
                }
            }
            wt[i] = ct[i] - processes[i].arrival_time - processes[i].burst_time;
            rt[i] = wt[i];
        }
    }
}

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

float avg_tat = 0, avg_wt = 0;

for (int i = 0; i < n; i++) {
    printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", processes[i].pid, processes[i].arrival_time,
        processes[i].burst_time, ct[i], wt[i], tat[i], rt[i]);
}

```



```

    avg_tat += tat[i];
    avg_wt += wt[i];
}

avg_tat /= n;
avg_wt /= n;

printf("\nAverage Turnaround Time: %.2f\n", avg_tat);
printf("Average Waiting Time: %.2f\n", avg_wt);
}

int main() {
    int n, time_quantum;

    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the time quantum: ");
    scanf("%d", &time_quantum);

    struct Process processes[n];
    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Enter Arrival Time for process %d: ", i+1);
        scanf("%d", &processes[i].arrival_time);
        printf("Enter Burst Time for process %d: ", i+1);
        scanf("%d", &processes[i].burst_time);
        processes[i].pid = i+1;
        processes[i].remaining_time = processes[i].burst_time;
    }

    roundRobin(processes, n, time_quantum);
}

```

output:

```
C:\Users\saisr\OneDrive\Desk x + v
Enter the number of processes: 5
Enter the time quantum: 2
Enter Arrival Time and Burst Time for each process:
Enter Arrival Time for process 1: 0
Enter Burst Time for process 1: 5
Enter Arrival Time for process 2: 1
Enter Burst Time for process 2: 3
Enter Arrival Time for process 3: 2
Enter Burst Time for process 3: 1
Enter Arrival Time for process 4: 3
Enter Burst Time for process 4: 2
Enter Arrival Time for process 5: 4
Enter Burst Time for process 5: 3
PID    AT    BT    CT    WT    TAT    RT
1       0     5    14     9    14     9
2       1     3    12     8    11     8
3       2     1     5     2     3     2
4       3     2     7     2     4     2
5       4     3    13     6     9     6

Average Turnaround Time: 8.20
Average Waiting Time: 5.40

Process returned 0 (0x0)   execution time : 108.783 s
Press any key to continue.
```

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

```
#include <stdio.h>

// Function to find waiting time for FCFS
void findWaitingTime(int processes[], int n, int bt[], int at[], int wt[]) {
    wt[0] = 0;
    for (int i = 1; i < n; i++) {
        wt[i] = bt[i-1] + wt[i-1] - at[i-1];
        if (wt[i] < 0)
            wt[i] = 0;
    }
}

// Function to find turnaround time
void findTurnaroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i];
    }
}

// Function to implement Round Robin scheduling
void roundRobin(int processes[], int n, int bt[], int at[], int quantum) {
    int wt[n], tat[n], ct[n], total_wt = 0, total_tat = 0;
    int remaining_bt[n];
    int completed = 0;
    int time = 0;
    for (int i = 0; i < n; i++) {
        remaining_bt[i] = bt[i];
    }
    while (completed < n) {
        for (int i = 0; i < n; i++) {
            if (remaining_bt[i] > 0 && at[i] <= time) {

```

```

        if (remaining_bt[i] <= quantum) {
            time += remaining_bt[i];
            remaining_bt[i] = 0;
            ct[i] = time;
            completed++;
        } else {
            time += quantum;
            remaining_bt[i] -= quantum;
        }
    }
}

findWaitingTime(processes, n, bt, at, wt);
findTurnaroundTime(processes, n, bt, wt, tat);

printf("Processes  Burst Time  Arrival Time  Waiting Time  Turnaround Time  Completion Time\n");
for (int i = 0; i < n; i++) {
    printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], at[i], wt[i], tat[i], ct[i]);
    total_wt += wt[i];
    total_tat += tat[i];
}

printf("Average Waiting Time (Round Robin) = %f\n", (float)total_wt / n);
printf("Average Turnaround Time (Round Robin) = %f\n", (float)total_tat / n);
}

// Function to implement FCFS scheduling
void fcfs(int processes[], int n, int bt[], int at[]) {
    int wt[n], tat[n], ct[n], total_wt = 0, total_tat = 0;

    findWaitingTime(processes, n, bt, at, wt);
    findTurnaroundTime(processes, n, bt, wt, tat);

    printf("Processes  Burst Time  Arrival Time  Waiting Time  Turnaround Time  Completion Time\n");
    for (int i = 0; i < n; i++) {
        ct[i] = at[i] + bt[i];
    }
}

```

```

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

        total_wt += wt[i];

        total_tat += tat[i];
    }

    printf("Average Waiting Time (FCFS) = %f\n", (float)total_wt / n);

    printf("Average Turnaround Time (FCFS) = %f\n", (float)total_tat / n);
}

int main() {

    int processes[] = {1, 2, 3, 4, 5};

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

    int bt[] = {10, 5, 8, 12, 15};

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

    int quantum = 2;

    roundRobin(processes, n, bt, at, quantum);

    fcfs(processes, n, bt, at);

    return 0;
}

```

output:

```

C:\Users\saisr\OneDrive\Desktop
Processes Burst Time Arrival Time Waiting Time Turnaround Time Completion Time
P1      10          0          0          10          39
P2       5          1         10          15          23
P3       8          2         14          22          33
P4      12          3         20          32          45
P5      15          4         29          44          50
Average Waiting Time (Round Robin) = 14.600000
Average Turnaround Time (Round Robin) = 24.600000
Processes Burst Time Arrival Time Waiting Time Turnaround Time Completion Time
P1      10          0          0          10          10
P2       5          1         10          15          6
P3       8          2         14          22          10
P4      12          3         20          32          15
P5      15          4         29          44          19
Average Waiting Time (FCFS) = 14.600000
Average Turnaround Time (FCFS) = 24.600000
Process returned 0 (0x0)   execution time : 0.063 s
Press any key to continue.

```

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

a) Rate- Monotonic

```
#include <stdio.h>

// Structure to represent a process
struct Process {
    int execution_time;
    int time_period;
};

// Function to calculate the least common multiple (LCM)
int lcm(int a, int b) {
    int max = (a > b) ? a : b;
    while (1) {
        if (max % a == 0 && max % b == 0)
            return max;
        max++;
    }
}

// Function to check if the set of processes is schedulable
int is_schedulable(struct Process processes[], int n) {
    float utilization = 0.0;
    for (int i = 0; i < n; i++) {
        utilization += (float)processes[i].execution_time / processes[i].time_period;
    }
    return utilization <= 1.0;
}

int main() {
    struct Process processes[] = {
        {3, 20}, // P1
        {2, 5},  // P2
        {2, 10}  // P3
    };
}
```

```

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

// Check if the processes are schedulable
if (!is_schedulable(processes, n)) {
    printf("The given set of processes is not schedulable.\n");
    return 0;
}

// Calculate the scheduling time (LCM of time periods)
int scheduling_time = lcm(processes[0].time_period, processes[1].time_period);
scheduling_time = lcm(scheduling_time, processes[2].time_period);

// Display the execution order
printf("Execution order:\n");
for (int t = 0; t < scheduling_time; t++) {
    if (t % processes[1].time_period == 0)
        printf("P2 ");
    if (t % processes[2].time_period == 0)
        printf("P3 ");
    if (t % processes[0].time_period == 0)
        printf("P1 ");
}
printf("\n");
return 0;
}

```

output:

```
C:\Users\saisr\OneDrive\Desk x + v
Execution order:
P2 P3 P1 P2 P2 P3 P2
Process returned 0 (0x0)   execution time : 0.053 s
Press any key to continue.
```

b) Earliest-deadline First

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

void
sort (int proc[], int d[], int b[], int pt[], int n)
{
    int temp = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = i; j < n; j++)
        {
            if (d[j] < d[i])
            {
                temp = d[j];
                d[j] = d[i];
                d[i] = temp;
                temp = pt[i];
                pt[i] = pt[j];
                pt[j] = temp;
            }
        }
    }
}
```



```

        temp = b[j];
        b[j] = b[i];
        b[i] = temp;
        temp = proc[i];
        proc[i] = proc[j];
        proc[j] = temp;
    }

}

}

int gcd (int a, int b)
{
    int r;
    while (b > 0)
    {
        r = a % b;
        a = b;
        b = r;
    }

    return a;
}

int lcmul (int p[], int n)
{
    int lcm = p[0];
    for (int i = 1; i < n; i++)
    {
        lcm = (lcm * p[i]) / gcd (lcm, p[i]);
    }

    return lcm;
}

```

```

void main ()
{
    int n;

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

    scanf ("%d", &n);

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

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

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

        rem[i] = b[i];
    }

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

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

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

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

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

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

    //LCM
    int l = lcmul (pt, n);

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

    printf ("PID\tBurst\tDeadline\tPeriod\n");

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

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

    //EDF

```

```

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

int nextDeadlines[n];

for (int i = 0; i < n; i++)
{
    nextDeadlines[i] = d[i];
    rem[i] = b[i];
}

while (time < l)
{
    for (int i = 0; i < n; i++)
    {
        if (time % pt[i] == 0 && time != 0)
        {
            nextDeadlines[i] = time + d[i];
            rem[i] = b[i];
        }
    }

    int minDeadline = l + 1;
    int taskToExecute = -1;
    for (int i = 0; i < n; i++)
    {
        if (rem[i] > 0 && nextDeadlines[i] < minDeadline)
        {
            minDeadline = nextDeadlines[i];
            taskToExecute = i;
        }
    }

    if (taskToExecute != -1)
    {
        printf ("%dms : Task %d is running.\n", time, proc[taskToExecute]);
        rem[taskToExecute]--;
    }
}

```

```

    }

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

time++;
}
}

```

output:

```

C:\Users\saisr\OneDrive\Desk x + v
Enter the number of processes:3
Enter the CPU burst times:
3
2
2
Enter the deadlines:
7
4
8
Enter the time periods:
20
5
10

Earliest Deadline Scheduling:
PID    Burst  Deadline  Period
2       2      4         5
1       3      7         20
3       2      8         10
Scheduling occurs for 20 ms

0ms : Task 2 is running.
1ms : Task 2 is running.
2ms : Task 1 is running.
3ms : Task 1 is running.
4ms : Task 1 is running.
5ms : Task 3 is running.
6ms : Task 3 is running.
7ms : Task 2 is running.
8ms : Task 2 is running.
9ms : CPU is idle.
10ms : Task 2 is running.
11ms : Task 2 is running.

```

```

C:\Users\saisr\OneDrive\Desk x + v
10

Earliest Deadline Scheduling:
PID    Burst  Deadline  Period
2       2      4         5
1       3      7         20
3       2      8         10
Scheduling occurs for 20 ms

0ms : Task 2 is running.
1ms : Task 2 is running.
2ms : Task 1 is running.
3ms : Task 1 is running.
4ms : Task 1 is running.
5ms : Task 3 is running.
6ms : Task 3 is running.
7ms : Task 2 is running.
8ms : Task 2 is running.
9ms : CPU is idle.
10ms : Task 2 is running.
11ms : Task 2 is running.
12ms : Task 3 is running.
13ms : Task 3 is running.
14ms : CPU is idle.
15ms : Task 2 is running.
16ms : Task 2 is running.
17ms : CPU is idle.
18ms : CPU is idle.
19ms : CPU is idle.

Process returned 20 (0x14)    execution time : 63.708 s
Press any key to continue.

```

c) Proportional scheduling

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_TASKS 10
#define MAX_TICKETS 100
#define TIME_UNIT_DURATION_MS 100 // Duration of each time unit in milliseconds

struct Task {
    int tid;
    int tickets;
};

void schedule(struct Task tasks[], int num_tasks, int *time_span_ms) {
    int total_tickets = 0;

    for (int i = 0; i < num_tasks; i++) {
        total_tickets += tasks[i].tickets;
    }

    srand(time(NULL));

    int current_time = 0;
    int completed_tasks = 0;

    printf("Process Scheduling:\n");

    while (completed_tasks < num_tasks) {
        int winning_ticket = rand() % total_tickets;
        int cumulative_tickets = 0;

        for (int i = 0; i < num_tasks; i++) {
            cumulative_tickets += tasks[i].tickets;

            if (winning_ticket < cumulative_tickets) {
                printf("Time %d-%d: Task %d is running\n", current_time, current_time + 1, tasks[i].tid);
                current_time++;
                break;
            }
        }
        completed_tasks++;
    }

    // Calculate time span in milliseconds
    *time_span_ms = current_time * TIME_UNIT_DURATION_MS;
}

int main() {
    struct Task tasks[MAX_TASKS];
    int num_tasks;
    int time_span_ms;

    printf("Enter the number of tasks: ");
```

```

scanf("%d", &num_tasks);

if (num_tasks <= 0 || num_tasks > MAX_TASKS) {
    printf("Invalid number of tasks. Please enter a number between 1 and %d.\n", MAX_TASKS);
    return 1;
}

printf("Enter number of tickets for each task:\n");
for (int i = 0; i < num_tasks; i++) {
    tasks[i].tid = i + 1;
    printf("Task %d tickets: ", tasks[i].tid);
    scanf("%d", &tasks[i].tickets);
}

printf("\nRunning tasks:\n");
schedule(tasks, num_tasks, &time_span_ms);

printf("\nTime span of the Gantt chart: %d milliseconds\n", time_span_ms);

return 0;
}

```

output:

```

C:\Users\saisr\OneDrive\Desktop
Enter the number of tasks: 3
Enter number of tickets for each task:
Task 1 tickets: 10
Task 2 tickets: 20
Task 3 tickets: 30

Running tasks:
Process Scheduling:
Time 0-1: Task 1 is running
Time 1-2: Task 3 is running
Time 2-3: Task 2 is running

Time span of the Gantt chart: 300 milliseconds

Process returned 0 (0x0) execution time : 47.068 s
Press any key to continue.

```

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);
```

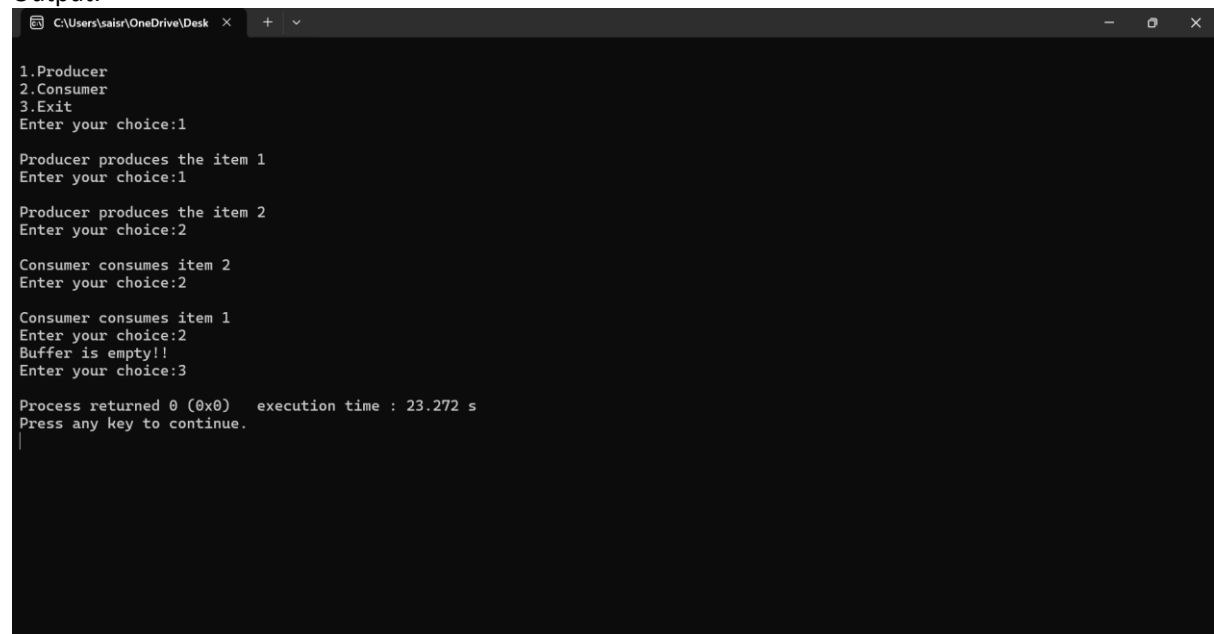
```
pthread_create(&producerThread, NULL, producer, NULL);
pthread_create(&consumerThread, NULL, consumer, NULL);

pthread_join(producerThread, NULL);
pthread_join(consumerThread, NULL);

sem_destroy(&mutex);
sem_destroy(&full);
sem_destroy(&empty);

return 0;
}
```

Output:



```
C:\Users\saisr\OneDrive\Desktop x + v
1.Producer
2.Consumer
3.Exit
Enter your choice:1

Producer produces the item 1
Enter your choice:1

Producer produces the item 2
Enter your choice:2

Consumer consumes item 2
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!!
Enter your choice:3

Process returned 0 (0x0) execution time : 23.272 s
Press any key to continue.
|
```


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

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#define MAX_PHILOSOPHERS 5
```

```
void allow_one_to_eat(int hungry[], int n) {
```

```
    int isWaiting[MAX_PHILOSOPHERS];
```

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

```
        isWaiting[i] = 1;
```

```
    }
```

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

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

```
        isWaiting[hungry[i]] = 0;
```

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

```
            if (isWaiting[hungry[j]]) {
```

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

```
            }
```

```
        }
```

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

```
            isWaiting[k] = 1;
```

```
        }
```

```
        isWaiting[hungry[i]] = 0;
```

```
    }
```

```
}
```

```
void allow_two_to_eat(int hungry[], int n) {
```

```
    if (n < 2 || n > MAX_PHILOSOPHERS) {
```

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

```
        return;
```

```
    }
```

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

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

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

```

        for (int k = 0; k < n; k++) {
            if (k != i && k != j) {
                printf("P %d is waiting\n", hungry[k]);
            }
        }
    }
}

int main() {
    int total_philosophers, hungry_count;
    int hungry_positions[MAX_PHILOSOPHERS];
    printf("DINING PHILOSOPHER PROBLEM\n");
    printf("Enter the total no. of philosophers: ");
    scanf("%d", &total_philosophers);
    if (total_philosophers > MAX_PHILOSOPHERS || total_philosophers < 2) {
        printf("Invalid number of philosophers.\n");
        return 1;
    }
    printf("How many are hungry: ");
    scanf("%d", &hungry_count);
    if (hungry_count < 1 || hungry_count > total_philosophers) {
        printf("Invalid number of hungry philosophers.\n");
        return 1;
    }
    for (int i = 0; i < hungry_count; i++) {
        printf("Enter philosopher %d position: ", i + 1);
        scanf("%d", &hungry_positions[i]);
        if (hungry_positions[i] < 0 || hungry_positions[i] >= total_philosophers) {
            printf("Invalid philosopher position.\n");
            return 1;
        }
    }
}

```

```

}

int choice;

while (1) {
    printf("\n1. One can eat at a time\n");
    printf("2. Two can eat at a time\n");
    printf("3. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
    switch (choice) {
        case 1:
            allow_one_to_eat(hungry_positions, hungry_count);
            break;
        case 2:
            allow_two_to_eat(hungry_positions, hungry_count);
            break;
        case 3:
            exit(0);
        default:
            printf("Invalid choice\n");
    }
}

return 0;
}

```

output:

```
C:\Users\saisr\OneDrive\Desk x + v
DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 5
How many are hungry: 2
Enter philosopher 1 position: 1
Enter philosopher 2 position: 4

1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 1
P 1 is granted to eat
P 4 is waiting
P 4 is granted to eat

1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 2
P 1 and P 4 are granted to eat

1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 3

Process returned 0 (0x0)   execution time : 59.936 s
Press any key to continue.
```

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

```
#include <stdio.h>
```

```
int main() {
```

```
    int n, m;
```

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

```
    scanf("%d", &n);
```

```
    printf("Enter the number of resources: ");
```

```
    scanf("%d", &m);
```

```
    int available[m];
```

```
    printf("Enter the available resources: ");
```

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

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

```
    }
```

```
    int maximum[n][m];
```

```
    printf("Enter the maximum resources for each process:\n");
```

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

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

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

```
        }
```

```
    }
```

```
    int allocation[n][m];
```

```
    printf("Enter the allocated resources for each process:\n");
```

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

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

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

```
        }
```

```
    }
```

```
    int need[n][m];
```

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

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

```
            need[i][j] = maximum[i][j] - allocation[i][j];
```

```

    }
}
printf(" Process Allocation Max Need \n");
for (int i = 0; i < n; i++) {
    printf(" | P%d | ", i + 1);
    for (int j = 0; j < m; j++) {
        printf("%d ", allocation[i][j]);
    }
    printf(" | ");
    for (int j = 0; j < m; j++) {
        printf("%d ", maximum[i][j]);
    }
    printf(" | ");
    for (int j = 0; j < m; j++) {
        printf("%d ", need[i][j]);
    }
    printf(" |\n");
}

int work[m];
for (int i = 0; i < m; i++) {
    work[i] = available[i];
}

int finish[n];
for (int i = 0; i < n; i++) {
    finish[i] = 0;
}

int safeSequence[n];
int count = 0;
int safe = 1;
while (count < n) {
    int found = 0;

```

```

for (int i = 0; i < n; i++) {
    if (finish[i] == 0) {
        int j;
        for (j = 0; j < m; j++) {
            if (need[i][j] > work[j]) {
                break;
            }
        }
        if (j == m) {
            for (j = 0; j < m; j++) {
                work[j] += allocation[i][j];
            }
            finish[i] = 1;
            safeSequence[count++] = i;
            found = 1;
        }
    }
}

if (!found) {
    safe = 0;
    break;
}

if (safe) {
    printf("The system is in a safe state.\n");
    printf("Safety sequence: ");
    for (int i = 0; i < n; i++) {
        printf("P%d ", safeSequence[i] + 1);
    }
    printf("\n");
} else {

```

```

        printf("The system is in an unsafe state and might lead to deadlock.\n");
    }

    return 0;
}

```

output:

```

C:\Users\saisr\OneDrive\Desktop > .\program.exe
Enter the number of processes: 5
Enter the number of resources: 3
Enter the available resources: 3 3 2
Enter the maximum resources for each process:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter the allocated resources for each process:
0 1 0
3 0 2
3 0 2
2 1 1
0 0 2

```

Process	Allocation	Max	Need
P1	0 1 0	7 5 3	7 4 3
P2	3 0 2	3 2 2	0 2 0
P3	3 0 2	9 0 2	6 0 0
P4	2 1 1	2 2 2	0 1 1
P5	0 0 2	4 3 3	4 3 1

```

The system is in a safe state.
Safety sequence: P2 P3 P4 P5 P1

Process returned 0 (0x0)   execution time : 109.005 s
Press any key to continue.

```


8. Write a C program to simulate deadlock detection

```
#include<stdio.h>
void main()
{
    int n,m,i,j;
    printf("Enter the number of processes and number of types of resources:\n");
    scanf("%d %d",&n,&m);
    int max[n][m],need[n][m],all[n][m],ava[m],flag=1,finish[n],dead[n],c=0;
    printf("Enter the maximum number of each type of resource needed by each process:\n");
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
        {
            scanf("%d",&max[i][j]);
        }
    }
    printf("Enter the allocated number of each type of resource needed by each process:\n");
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
        {
            scanf("%d",&all[i][j]);
        }
    }
    printf("Enter the available number of each type of resource:\n");
    for(j=0;j<m;j++)
    {
        scanf("%d",&ava[j]);
    }
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
        {
            need[i][j]=max[i][j]-all[i][j];
        }
    }
    for(i=0;i<n;i++)
    {
        finish[i]=0;
    }
    while(flag)
    {
        flag=0;
        for(i=0;i<n;i++)
        {
            c=0;
            for(j=0;j<m;j++)
            {
                if(finish[i]==0 && need[i][j]<=ava[j])
                {
                    c++;
                    if(c==m)
                    {
                        for(j=0;j<m;j++)
                        {
                            ava[j]+=all[i][j];
                        }
                    }
                }
            }
        }
    }
}
```

```

        finish[i]=1;
        flag=1;
    }
    if(finish[i]==1)
    {
        i=n;
    }
}
}
}
j=0;
flag=0;
for(i=0;i<n;i++)
{
    if(finish[i]==0)
    {
        dead[j]=i;
        j++;
        flag=1;
    }
}
if(flag==1)
{
    printf("Deadlock has occurred:\n");
    printf("The deadlock processes are:\n");
    for(i=0;i<n;i++)
    {
        printf("P%d ",dead[i]);
    }
}
else
    printf("No deadlock has occurred!\n");
}

```

output:

```

C:\Users\saisr\OneDrive\Desktop >
Enter the number of processes and number of types of resources:
5 3
Enter the maximum number of each type of resource needed by each process:
7 5 3
3 2 2
9 0 2
2 2 4
4 3 3
Enter the allocated number of each type of resource needed by each process:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the available number of each type of resource:
3 3 2
No deadlock has occurred!

Process returned 0 (0x0)   execution time : 120.785 s
Press any key to continue.

```

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

a) Worst-fit

b) Best-fit

c) First-fit

```
#include <stdio.h>
```

```
struct Block {  
    int block_no;  
    int block_size;  
    int is_free; // 1 for free, 0 for allocated  
};
```

```
struct File {  
    int file_no;  
    int file_size;  
};
```

```

void firstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf("Memory Management Scheme - First Fit\n");
    printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");

    for (int i = 0; i < n_files; i++) {
        for (int j = 0; j < n_blocks; j++) {
            if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
                blocks[j].is_free = 0;
                printf("%d\t%d\t%d\t%d\t%d\n", files[i].file_no, files[i].file_size, blocks[j].block_no,
                    blocks[j].block_size, blocks[j].block_size - files[i].file_size);
                break;
            }
        }
    }
}

```

```

void worstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf("Memory Management Scheme - Worst Fit\n");
    printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");

    for (int i = 0; i < n_files; i++) {
        int worst_fit_block = -1;
        int max_fragment = -1; // Initialize with a negative value
        for (int j = 0; j < n_blocks; j++) {
            if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
                int fragment = blocks[j].block_size - files[i].file_size;
                if (fragment > max_fragment) {
                    max_fragment = fragment;
                    worst_fit_block = j;
                }
            }
        }

        if (worst_fit_block != -1) {
            blocks[worst_fit_block].is_free = 0;
            printf("%d\t%d\t%d\t%d\t%d\n", files[i].file_no, files[i].file_size,
                blocks[worst_fit_block].block_no, blocks[worst_fit_block].block_size, max_fragment);
        }
    }
}

```

```

void bestFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf("Memory Management Scheme - Best Fit\n");
    printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");

    for (int i = 0; i < n_files; i++) {
        int best_fit_block = -1;
        int min_fragment = 10000; // Initialize with a large value
        for (int j = 0; j < n_blocks; j++) {
            if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
                int fragment = blocks[j].block_size - files[i].file_size;
                if (fragment < min_fragment) {
                    min_fragment = fragment;
                }
            }
        }
    }
}

```

```

        best_fit_block = j;
    }
}

if (best_fit_block != -1) {
    blocks[best_fit_block].is_free = 0;
    printf("%d\t%d\t%d\t%d\t%d\t%d\n", files[i].file_no, files[i].file_size,
blocks[best_fit_block].block_no, blocks[best_fit_block].block_size, min_fragment);
}
}
}

```

```

int main() {
    int n_blocks, n_files;
    printf("Enter the number of blocks: ");
    scanf("%d", &n_blocks);
    printf("Enter the number of files: ");
    scanf("%d", &n_files);

    struct Block blocks[n_blocks];
    for (int i = 0; i < n_blocks; i++) {
        blocks[i].block_no = i + 1;
        printf("Enter the size of block %d: ", i + 1);
        scanf("%d", &blocks[i].block_size);
        blocks[i].is_free = 1;
    }

    struct File files[n_files];
    for (int i = 0; i < n_files; i++) {
        files[i].file_no = i + 1;
        printf("Enter the size of file %d: ", i + 1);
        scanf("%d", &files[i].file_size);
    }

    firstFit(blocks, n_blocks, files, n_files);
    printf("\n");

    // Reset blocks for worst fit
    for (int i = 0; i < n_blocks; i++) {
        blocks[i].is_free = 1;
    }

    worstFit(blocks, n_blocks, files, n_files);
    printf("\n");

    // Reset blocks for best fit
    for (int i = 0; i < n_blocks; i++) {
        blocks[i].is_free = 1;
    }

    bestFit(blocks, n_blocks, files, n_files);

    return 0;
}

```

```
}
```

output:

```
C:\Users\saisr\OneDrive\Desktop >
Enter the number of blocks: 3
Enter the number of files: 2
Enter the size of block 1: 5
Enter the size of block 2: 2
Enter the size of block 3: 7
Enter the size of file 1: 1
Enter the size of file 2: 4
Memory Management Scheme - First Fit
File_no:      File_size:      Block_no:      Block_size:      Fragment
1             1             1             5             4
2             4             3             7             3

Memory Management Scheme - Worst Fit
File_no:      File_size:      Block_no:      Block_size:      Fragment
1             1             3             7             6
2             4             1             5             1

Memory Management Scheme - Best Fit
File_no:      File_size:      Block_no:      Block_size:      Fragment
1             1             2             2             1
2             4             1             5             1

Process returned 0 (0x0)   execution time : 54.559 s
Press any key to continue.
```

10. Write a C program to simulate paging technique of memory management.

```
#include <stdio.h>
```

```
#include <limits.h>
```

```
#include <stdlib.h>
```

```
void print_frames(int frame[], int capacity, int page_faults) {
    for (int i = 0; i < capacity; i++) {
        if (frame[i] == -1)
            printf("- ");
        else
            printf("%d ", frame[i]);
    }
    if (page_faults > 0)
        printf("PF No. %d", page_faults);
    printf("\n");
}
```

```
void fifo(int pages[], int n, int capacity) {
    int frame[capacity], index = 0, page_faults = 0;
```

```

for (int i = 0; i < capacity; i++)
    frame[i] = -1;

printf("FIFO Page Replacement Process:\n");
for (int i = 0; i < n; i++) {
    int found = 0;
    for (int j = 0; j < capacity; j++) {
        if (frame[j] == pages[i]) {
            found = 1;
            break;
        }
    }
    if (!found) {
        frame[index] = pages[i];
        index = (index + 1) % capacity;
        page_faults++;
    }
    print_frames(frame, capacity, found ? 0 : page_faults);
}
printf("Total Page Faults using FIFO: %d\n\n", page_faults);
}

void lru(int pages[], int n, int capacity) {
    int frame[capacity], counter[capacity], time = 0, page_faults = 0;
    for (int i = 0; i < capacity; i++) {
        frame[i] = -1;
        counter[i] = 0;
    }

    printf("LRU Page Replacement Process:\n");
    for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = 0; j < capacity; j++) {
            if (frame[j] == pages[i]) {
                found = 1;
                counter[j] = time++;
                break;
            }
        }
        if (!found) {
            int min = INT_MAX, min_index = -1;
            for (int j = 0; j < capacity; j++) {
                if (counter[j] < min) {
                    min = counter[j];
                    min_index = j;
                }
            }
            frame[min_index] = pages[i];
            counter[min_index] = time++;
            page_faults++;
        }
        print_frames(frame, capacity, found ? 0 : page_faults);
    }
    printf("Total Page Faults using LRU: %d\n\n", page_faults);
}

```

```

}

void optimal(int pages[], int n, int capacity) {
    int frame[capacity], page_faults = 0;
    for (int i = 0; i < capacity; i++)
        frame[i] = -1;

    printf("Optimal Page Replacement Process:\n");
    for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = 0; j < capacity; j++) {
            if (frame[j] == pages[i]) {
                found = 1;
                break;
            }
        }
        if (!found) {
            int farthest = i + 1, index = -1;
            for (int j = 0; j < capacity; j++) {
                int k;
                for (k = i + 1; k < n; k++) {
                    if (frame[j] == pages[k])
                        break;
                }
                if (k > farthest) {
                    farthest = k;
                    index = j;
                }
            }
            if (index == -1) {
                for (int j = 0; j < capacity; j++) {
                    if (frame[j] == -1) {
                        index = j;
                        break;
                    }
                }
            }
            frame[index] = pages[i];
            page_faults++;
        }
        print_frames(frame, capacity, found ? 0 : page_faults);
    }
    printf("Total Page Faults using Optimal: %d\n\n", page_faults);
}

int main() {
    int n, capacity;
    printf("Enter the number of pages: ");
    scanf("%d", &n);
    int *pages = (int*)malloc(n * sizeof(int));
    printf("Enter the pages: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &pages[i]);
    printf("Enter the frame capacity: ");

```



```

scanf("%d", &capacity);

printf("\nPages: ");
for (int i = 0; i < n; i++)
    printf("%d ", pages[i]);
printf("\n\n");

fifo(pages, n, capacity);
lru(pages, n, capacity);
optimal(pages, n, capacity);

free(pages);
return 0;
}

```

output:

```

C:\Users\saisr\OneDrive\Desk x + v
Enter the number of pages: 20
Enter the pages: 7
0
1
2
0
3
0
4
2
3
0
3
2
1
2
0
1
7
0
1
Enter the frame capacity: 3

Pages: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

FIFO Page Replacement Process:
7 - - PF No. 1
7 0 - PF No. 2
7 0 1 PF No. 3
2 0 1 PF No. 4
2 0 1
2 3 1 PF No. 5
2 3 0 PF No. 6
2 3 0 PF No. 6
4 3 0 PF No. 7
4 2 0 PF No. 8
4 2 3 PF No. 9
0 2 3 PF No. 10
0 2 3
0 2 3
0 1 3 PF No. 11
0 1 2 PF No. 12
0 1 2
0 1 2
7 1 2 PF No. 13
7 0 2 PF No. 14
7 0 1 PF No. 15
Total Page Faults using FIFO: 15

LRU Page Replacement Process:
7 - - PF No. 1
0 - - PF No. 2
0 1 - PF No. 3
0 1 2 PF No. 4
0 1 2
0 3 2 PF No. 5
0 3 2
0 3 4 PF No. 6
0 2 4 PF No. 7
3 2 4 PF No. 8
3 2 0 PF No. 9
3 2 0
3 2 0
3 2 1 PF No. 10
3 2 1
0 2 1 PF No. 11

```

```
C:\Users\saisr\OneDrive\Desk x + v
0 2 1 PF No. 11
0 2 1
0 7 1 PF No. 12
0 7 1
0 7 1
Total Page Faults using LRU: 12

Optimal Page Replacement Process:
7 - - PF No. 1
7 0 - PF No. 2
7 0 1 PF No. 3
2 0 1 PF No. 4
2 0 1
2 0 3 PF No. 5
2 0 3
2 4 3 PF No. 6
2 4 3
2 4 3
2 0 3 PF No. 7
2 0 3
2 0 3
2 0 1 PF No. 8
2 0 1
2 0 1
2 0 1
7 0 1 PF No. 9
7 0 1
7 0 1
Total Page Faults using Optimal: 9

Process returned 0 (0x0) execution time : 73.003 s
Press any key to continue.
```