

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT

on

Operating System

Submitted by

Vrishank J Vasist (1BM21CS246)

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

May-2023 to July-2023

B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering



CERTIFICATE

This is to certify that the Lab work entitled “**Operating System**” carried out by **Vrishank J Vasist (1BM21CS246)**, who is bonafide student of **B.M.S. College of Engineering**. It is in partial fulfilment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the academic semester May-2023 to July-2023. The Lab report has been approved as it satisfies the academic requirements in respect of an **Operating System(22CS4PCOPS)** work prescribed for the said degree.

Dr K. Panimozhi

Assistant Professor

Department of CSE

BMSCE, Bengaluru

Dr. Jyothi S Nayak

Professor and Head

Department of CSE

BMSCE, Bengaluru

Course outcome:

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

Lab program no.	Program details	Page no.
1	Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time. <ul style="list-style-type: none"> • FCFS • SJF (pre-emptive & Non-pre-emptive) 	1
2	Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time. <ul style="list-style-type: none"> • Priority (pre-emptive & Non-pre-emptive) • Round Robin (Experiment with different quantum sizes for RR algorithm) 	5
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.	17
4	Write a C program to simulate Real-Time CPU Scheduling algorithms: <ul style="list-style-type: none"> a) Rate- Monotonic b) Earliest-deadline First c) Proportional scheduling 	20
5	Write a C program to simulate producer-consumer problem using semaphores.	37
6	Write a C program to simulate the concept of Dining-Philosophers problem.	41
7	Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.	46
8	Write a C program to simulate deadlock detection	50
9	Write a C program to simulate the following contiguous memory allocation techniques <ul style="list-style-type: none"> a) Worst-fit b) Best-fit c) First-fit 	56
10	Write a C program to simulate paging technique of memory management.	
11	Write a C program to simulate page replacement algorithms <ul style="list-style-type: none"> a) FIFO b) LRU c) Optimal 	67
12	Write a C program to simulate disk scheduling algorithms <ul style="list-style-type: none"> a) FCFS b) SCAN c) C-SCAN 	77
13	Write a C program to simulate disk scheduling algorithms <ul style="list-style-type: none"> a) SSTF b) LOOK c) c-LOOK 	85

1. Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.

- FCFS
- SJF (Non-pre-emptive)

FCFS

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

```
void main()
```

```
{
```

```
    int i,pid[10],n,burst[10],wt[10],ta[10];
```

```
    float avgwt=0,avgta=0;
```

```
    printf("enter the number of processes\n");
```

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

```
    printf("enter the process id and it's burst time\n");
```

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

```
    {
```

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

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

```
    }
```

```
    wt[0]=0;
```

```
    printf("according first come first serve schedule\n");
```

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

```
    {
```

```
        printf("processor id %d\t",pid[i]);
```

```
        printf("burst time %d\n",burst[i]);
```

```
    }
```

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

```
    {
```

```
        wt[i]=wt[i-1]+burst[i-1];
```

```
    }
```

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

```
    {
```

```

        ta[i]=wt[i]+burst[i];
    }
    for(i=0;i<n;i++)
    {
        avgwt+=wt[i];
        avgta+=ta[i];
    }
    avgwt=avgwt/n;
    avgta=avgta/n;
    printf("\nAverage waiting time is %f\n",avgwt);
    printf("\nAverage turnaround time is %f\n",avgta);

}

```

Output:

```

enter the number of processes
4
enter the process id and it's burst time
1 5
2 4
3 3
4 1
according first come first serve schedule
processor id 1 burst time 5
processor id 2 burst time 4
processor id 3 burst time 3
processor id 4 burst time 1

Average waiting time is 6.500000

Average turnaround time is 9.750000

```

SJF(non-pre-emptive)

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

```
void swap(int *a,int *b)
```

```

{
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}

```

```

}

void main()
{
    int i,j,temp,pid[10],n,burst[10],wt[10],ta[10];
    float avgwt=0,avgta=0;
    printf("enter the number of processes\n");
    scanf("%d",&n);
    printf("enter the process id and it's burst time\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&pid[i]);
        scanf("%d",&burst[i]);
    }
    //sorting
    for(i=0;i<n-1;i++)
    {
        for(j=0;j<n-i-1;j++)
        {
            if(burst[j]>burst[j+1])
            {
                swap(&burst[j],&burst[j+1]);
                swap(&pid[j],&pid[j+1]);
            }
        }
    }
    printf("according shortest job schedule\n");
    for(i=0;i<n;i++)
    {
        printf("processor id %d\t",pid[i]);
        printf("burst time %d\n",burst[i]);
    }
}

```

```

wt[0]=0;
for(i=1;i<n;i++)
{
    wt[i]=wt[i-1]+burst[i-1];
}
for(i=0;i<n;i++)
{
    ta[i]=wt[i]+burst[i];
}
for(i=0;i<n;i++)
{
    avgwt+=wt[i];
    avgta+=ta[i];
}
avgwt=avgwt/n;
avgta=avgta/n;
printf("\nAverage waiting time is %f\n",avgwt);
printf("\nAverage turnaround time is %f\n",avgta);
}

```

Output:

```

enter the number of processes
4
enter the process id and it's burst time
1 2
2 5
3 1
4 4
according shortest job schedule
processor id 3  burst time 1
processor id 1  burst time 2
processor id 4  burst time 4
processor id 2  burst time 5

Average waiting time is 2.750000
Average turnaround time is 5.750000

```


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

- SJF (pre-emptive)
- Priority (pre-emptive & Non-pre-emptive)
- Round Robin (Experiment with different quantum sizes for RR algorithm)

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

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

```
#define MAX_PROCESSES 10
```

```
struct Process
```

```
{
```

```
    int pid;
```

```
    int arrival_time;
```

```
    int burst_time;
```

```
    int priority;
```

```
    int remaining_time;
```

```
    int turnaround_time;
```

```
    int waiting_time;
```

```
};
```

```
void sjf_preemptive(struct Process processes[], int n)
```

```
{
```

```
    int total_time = 0, i;
```

```
    int completed = 0;
```

```
    while (completed < n)
```

```
    {
```

```
        int shortest_burst = -1;
```

```
        int next_process = -1;
```

```

for (i = 0; i < n; i++)
{
    if (processes[i].arrival_time <= total_time && processes[i].remaining_time > 0)
    {
        if (shortest_burst == -1 || processes[i].remaining_time < shortest_burst)
        {
            shortest_burst = processes[i].remaining_time;
            next_process = i;
        }
    }
}

if (next_process == -1)
{
    total_time++;
    continue;
}

processes[next_process].remaining_time--;
total_time++;

if (processes[next_process].remaining_time == 0)
{
    completed++;

    processes[next_process].turnaround_time = total_time -
processes[next_process].arrival_time;

    processes[next_process].waiting_time = processes[next_process].turnaround_time -
processes[next_process].burst_time;
}
}

```

```

double total_turnaround_time = 0;
double total_waiting_time = 0;

printf("Process\tTurnaround Time\tWaiting Time\n");
for (i = 0; i < n; i++)
{
    printf("%d\t%d\t\t%d\n", processes[i].pid, processes[i].turnaround_time,
processes[i].waiting_time);

    total_turnaround_time += processes[i].turnaround_time;
    total_waiting_time += processes[i].waiting_time;
}

printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
}

void priority_nonpreemptive(struct Process processes[], int n)
{
    int i, j, count = 0, m;
    for (i = 0; i < n; i++)
    {
        if (processes[i].arrival_time == 0)
            count++;
    }
    if (count == n || count == 1)
    {
        if (count == n)
        {
            for (i = 0; i < n - 1; i++)
            {

```

```

    for (j = 0; j < n - i - 1; j++)
    {
        if (processes[j].priority > processes[j + 1].priority)
        {
            struct Process temp = processes[j];
            processes[j] = processes[j + 1];
            processes[j + 1] = temp;
        }
    }
}

else
{
    for (i = 1; i < n - 1; i++)
    {
        for (j = 1; j <= n - i - 1; j++)
        {
            if (processes[j].priority > processes[j + 1].priority)
            {
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}
}

```

```
int total_time = 0;
```

```
double total_turnaround_time = 0;
```

```

double total_waiting_time = 0;

for (i = 0; i < n; i++)
{
    total_time += processes[i].burst_time;
    processes[i].turnaround_time = total_time - processes[i].arrival_time;
    processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;

    total_turnaround_time += processes[i].turnaround_time;
    total_waiting_time += processes[i].waiting_time;
}

printf("Process\tTurnaround Time\tWaiting Time\n");
for (i = 0; i < n; i++)
{
    printf("%d\t%d\t%d\n", processes[i].pid, processes[i].turnaround_time,
processes[i].waiting_time);
}

printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
}

void priority_preemptive(struct Process processes[], int n)
{
    int total_time = 0, i;
    int completed = 0;

    while (completed < n)
    {
        int highest_priority = -1;

```

```

int next_process = -1;

for (i = 0; i < n; i++)
{
    if (processes[i].arrival_time <= total_time && processes[i].remaining_time > 0)
    {
        if (highest_priority == -1 || processes[i].priority < highest_priority)
        {
            highest_priority = processes[i].priority;
            next_process = i;
        }
    }
}

if (next_process == -1)
{
    total_time++;
    continue;
}

processes[next_process].remaining_time--;
total_time++;

if (processes[next_process].remaining_time == 0)
{
    completed++;

    processes[next_process].turnaround_time = total_time -
processes[next_process].arrival_time;

    processes[next_process].waiting_time = processes[next_process].turnaround_time -
processes[next_process].burst_time;
}
}

```

```

double total_turnaround_time = 0;
double total_waiting_time = 0;

printf("Process\tTurnaround Time\tWaiting Time\n");
for (i = 0; i < n; i++)
{
    printf("%d\t%d\t\t%d\n", processes[i].pid, processes[i].turnaround_time,
processes[i].waiting_time);

    total_turnaround_time += processes[i].turnaround_time;
    total_waiting_time += processes[i].waiting_time;
}

printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
}

void round_robin(struct Process processes[], int n, int quantum)
{
    int total_time = 0, i;
    int completed = 0;

    while (completed < n)
    {
        for (i = 0; i < n; i++)
        {
            if (processes[i].arrival_time <= total_time && processes[i].remaining_time > 0)
            {
                if (processes[i].remaining_time <= quantum)
                {

```

```

        total_time += processes[i].remaining_time;
        processes[i].remaining_time = 0;
        processes[i].turnaround_time = total_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;
        completed++;
    }
    else
    {
        total_time += quantum;
        processes[i].remaining_time -= quantum;
    }
}

double total_turnaround_time = 0;
double total_waiting_time = 0;

printf("Process\tTurnaround Time\tWaiting Time\n");
for (i = 0; i < n; i++)
{
    printf("%d\t%d\t\t%d\n", processes[i].pid, processes[i].turnaround_time,
processes[i].waiting_time);

    total_turnaround_time += processes[i].turnaround_time;
    total_waiting_time += processes[i].waiting_time;
}

printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
}

```



```

int main()
{
    int n, quantum, i, choice;
    struct Process processes[MAX_PROCESSES];

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

    for (i = 0; i < n; i++)
    {
        printf("Process %d\n", i + 1);
        printf("Enter arrival time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Enter burst time: ");
        scanf("%d", &processes[i].burst_time);
        printf("Enter priority: ");
        scanf("%d", &processes[i].priority);
        processes[i].pid = i + 1;
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].turnaround_time = 0;
        processes[i].waiting_time = 0;
    }

    printf("Select a scheduling algorithm:\n");
    printf("1. SJF Preemptive\n");
    printf("2. Priority Non-preemptive\n");
    printf("3. Priority Preemptive\n");
    printf("4. Round Robin\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

```

```

switch (choice)
{
case 1:
    printf("\nSJF Preemptive Scheduling:\n");
    sjf_preemptive(processes, n);
    break;
case 2:
    printf("\nPriority Non-preemptive Scheduling:\n");
    priority_nonpreemptive(processes, n);
    break;
case 3:
    printf("\nPriority Preemptive Scheduling:\n");
    priority_preemptive(processes, n);
    break;
case 4:
    printf("\nEnter the quantum size for Round Robin: ");
    scanf("%d", &quantum);
    printf("\nRound Robin Scheduling (Quantum: %d):\n", quantum);
    round_robin(processes, n, quantum);
    break;
default:
    printf("Invalid choice!\n");
    return 1;
}

return 0;
}

```

Output:

```

Enter the number of processes: 3
Process 1
Enter arrival time: 0
Enter burst time: 3
Enter priority: 2
Process 2
Enter arrival time: 1
Enter burst time: 4
Enter priority: 3
Process 3
Enter arrival time: 2
Enter burst time: 5
Enter priority: 3
Select a scheduling algorithm:
1. SJF Preemptive
2. Priority Non-preemptive
3. Priority Preemptive
4. Round Robin
Enter your choice: 1

SJF Preemptive Scheduling:
Process Turnaround Time Waiting Time
1      3      0
2      6      2
3     10      5
Average Turnaround Time: 6.33
Average Waiting Time: 2.33

```

```

Enter the number of processes: 3
Process 1
Enter arrival time: 0
Enter burst time: 2
Enter priority: 2
Process 2
Enter arrival time: 1
Enter burst time: 6
Enter priority: 3
Process 3
Enter arrival time: 2
Enter burst time: 4
Enter priority: 2
Select a scheduling algorithm:
1. SJF Preemptive
2. Priority Non-preemptive
3. Priority Preemptive
4. Round Robin
Enter your choice: 2

Priority Non-preemptive Scheduling:
Process Turnaround Time Waiting Time
1      2      0
3      4      0
2     11      5
Average Turnaround Time: 5.67
Average Waiting Time: 1.67

```

```

Enter the number of processes: 3
Process 1
Enter arrival time: 0
Enter burst time: 2
Enter priority: 0
Process 2
Enter arrival time: 1
Enter burst time: 5
Enter priority: 3
Process 3
Enter arrival time: 2
Enter burst time: 4
Enter priority: 2
Select a scheduling algorithm:
1. SJF Preemptive
2. Priority Non-preemptive
3. Priority Preemptive
4. Round Robin
Enter your choice: 3

Priority Preemptive Scheduling:
Process Turnaround Time Waiting Time
1         2             0
2        10             5
3         4             0
Average Turnaround Time: 5.33
Average Waiting Time: 1.67

```

```

Enter the number of processes: 3
Process 1
Enter arrival time: 0
Enter burst time: 1
Enter priority: 2
Process 2
Enter arrival time: 1
Enter burst time: 5
Enter priority: 3
Process 3
Enter arrival time: 2
Enter burst time: 4
Enter priority: 2
Select a scheduling algorithm:
1. SJF Preemptive
2. Priority Non-preemptive
3. Priority Preemptive
4. Round Robin
Enter your choice: 4

Enter the quantum size for Round Robin: 2

Round Robin Scheduling (Quantum: 2):
Process Turnaround Time Waiting Time
1         1             0
2         9             4
3         7             3
Average Turnaround Time: 5.67
Average Waiting Time: 2.33

```

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>

void swap(int *a,int *b)
{
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}

void main()
{
    int n,pid[10],burst[10],type[10],arr[10],wt[10],ta[10],ct[10],i,j;
    float avgwt=0,avgta=0;
    int sum = 0;
    printf("Enter the total number of processes\n");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter the process id, type of process(user-0 and system-1), arrival time and burst time\n");
        scanf("%d",&pid[i]);
        scanf("%d",&type[i]);
        scanf("%d",&arr[i]);
        scanf("%d",&burst[i]);
    }
    //sorting the processes according to arrival time
    for(i=0;i<n-1;i++)
    {
        for(j=0;j<n-i-1;j++)
```

```

{
    if(arr[j]>arr[j+1])
    {
        swap(&arr[j],&arr[j+1]);
        swap(&pid[j],&pid[j+1]);
        swap(&burst[j],&burst[j+1]);
        swap(&type[j],&type[j+1]);

    }
}

//assuming only two process can have same arrival time and different priority
for(i=0;i<n-1;i++)
{
    for(j=0;j<n-i-1;j++)
    {
        if(arr[j]==arr[j+1] && type[j]<type[j+1])
        {
            swap(&arr[j],&arr[j+1]);
            swap(&pid[j],&pid[j+1]);
            swap(&burst[j],&burst[j+1]);
            swap(&type[j],&type[j+1]);
        }
    }
}

//calculating completion time, arrival time and waiting time
sum = sum + arr[0];
for(i = 0;i<n;i++){
    sum = sum + burst[i];
    ct[i] = sum;
    ta[i] = ct[i] - arr[i];
}

```

```

    wt[i] = ta[i] - burst[i];

    if(sum<arr[i+1]){
        int t = arr[i+1]-sum;
        sum = sum+t;
    }
}

printf("Process id\tType\tarrival time\tburst time\twaiting time\tturnaround time\n");
for(i=0;i<n;i++)
{
    avgta+=ta[i];
    avgwt+=wt[i];
    printf("%d\t%d\t%d\t%d\t%d\t%d\n",pid[i],type[i],arr[i],burst[i],wt[i],ta[i]);
}

printf("average waiting time =%f\n",avgwt/n);
printf("average turnaround time =%f",avgta/n);

}

```

Output:

```

Enter the total number of processes
4
Enter the process id, type of process(user-0 and system-1), arrival time and burst time
1 0 0 5
Enter the process id, type of process(user-0 and system-1), arrival time and burst time
2 1 0 4
Enter the process id, type of process(user-0 and system-1), arrival time and burst time
3 0 2 5
Enter the process id, type of process(user-0 and system-1), arrival time and burst time
4 1 4 3
Process id      Type      arrival time    burst time      waiting time     turnaround time
2              1          0              4              0              4
1              0          0              5              4              9
3              0          2              5              7              12
4              1          4              3              10             13
average waiting time =5.250000
average turnaround time =9.500000

```

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

algorithms:

a) Rate- Monotonic

b) Earliest-deadline First

c) Proportional scheduling

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

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

```
#include <math.h>
```

```
#include <stdbool.h>
```

```
#define MAX_PROCESS 10
```

```
int num_of_process = 3, count, remain, time_quantum;
```

```
int execution_time[MAX_PROCESS], period[MAX_PROCESS],
```

```
    remain_time[MAX_PROCESS], deadline[MAX_PROCESS],
```

```
    remain_deadline[MAX_PROCESS];
```

```
int burst_time[MAX_PROCESS], wait_time[MAX_PROCESS],
```

```
    completion_time[MAX_PROCESS], arrival_time[MAX_PROCESS];
```

```
// collecting details of processes
```

```
void get_process_info(int selected_algo)
```

```
{
```

```
    printf("Enter total number of processes (maximum %d): ",
```

```
        MAX_PROCESS);
```

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

```
    if (num_of_process < 1)
```

```
    {
```

```
        printf("Do you really want to schedule %d processes? -_-",
```

```
            num_of_process);
```

```
        exit(0);
```

```
    }
```

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

```
    {
```

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



```

    printf("==> Execution time: ");
    scanf("%d", &execution_time[i]);
    remain_time[i] = execution_time[i];

    printf("==> Period: ");
    scanf("%d", &period[i]);
}
}

// get maximum of three numbers
int max(int a, int b, int c)
{
    int max;
    if (a >= b && a >= c)
        max = a;
    else if (b >= a && b >= c)
        max = b;
    else if (c >= a && c >= b)
        max = c;
    return max;
}

// calculating the observation time for scheduling timeline
int get_observation_time(int selected_algo)
{
    return max(period[0], period[1], period[2]);
}

// print scheduling sequence
void print_schedule(int process_list[], int cycles)
{
    printf("\nScheduling:\n\n");
    printf("Time: ");

```

```

for (int i = 0; i < cycles; i++)
{
    if (i < 10)
        printf("| 0%d ", i);
    else
        printf("| %d ", i);
}
printf("\n");
for (int i = 0; i < num_of_process; i++)
{
    printf("P[%d]: ", i + 1);
    for (int j = 0; j < cycles; j++)
    {
        if (process_list[j] == i + 1)
            printf("#####");
        else
            printf(" | ");
    }
    printf("\n");
}
}

void rate_monotonic(int time)
{
    int process_list[100] = {0}, min = 999, next_process = 0;
    float utilization = 0;
    for (int i = 0; i < num_of_process; i++)
    {
        utilization += (1.0 * execution_time[i]) / period[i];
    }
    int n = num_of_process;
    if (utilization > n * (pow(2, 1.0 / n) - 1))

```

```

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

    exit(0);
}
for (int i = 0; i < time; i++)
{
    min = 1000;
    for (int j = 0; j < num_of_process; j++)
    {
        if (remain_time[j] > 0)
        {
            if (min > period[j])
            {
                min = period[j];
                next_process = j;
            }
        }
    }
    if (remain_time[next_process] > 0)
    {
        process_list[i] = next_process + 1; // +1 for catering 0 array index.
        remain_time[next_process] -= 1;
    }
    for (int k = 0; k < num_of_process; k++)
    {
        if ((i + 1) % period[k] == 0)
        {
            remain_time[k] = execution_time[k];
            next_process = k;
        }
    }
}

```

```

    }
}
print_schedule(process_list, time);
}
int main(int argc, char *argv[])
{
    int option = 0;
    printf("3. Rate Monotonic Scheduling\n");
    printf("Select > ");
    scanf("%d", &option);
    printf("-----\n");
    get_process_info(option); // collecting processes detail
    int observation_time = get_observation_time(option);
    if (option == 3)
        rate_monotonic(observation_time);
    return 0;
}

```

Output:

```

3. Rate Monotonic Scheduling
Select > 3
-----
Enter total number of processes (maximum 10): 3

Process 1:
==> Execution time: 2
==> Period: 5

Process 2:
==> Execution time: 1
==> Period: 10

Process 3:
==> Execution time: 3
==> Period: 15

Scheduling:
Time: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 |
P[1]: | #### | #### |      |      |      | #### | #### |      |      |      | #### | #### |      |      |
P[2]: |      |      | #### |      |      |      |      |      |      |      |      |      | #### |      |
P[3]: |      |      |      | #### | #### |      |      | #### |      |      |      |      |      |

```

```

#include <stdio.h>

#define arrival            0
#define execution         1
#define deadline          2
#define period            3
#define abs_arrival       4
#define execution_copy    5
#define abs_deadline      6


typedef struct
{
    int T[7],instance,alive;

}task;


#define IDLE_TASK_ID 1023
#define ALL 1
#define CURRENT 0


void get_tasks(task *t1,int n);
int hyperperiod_calc(task *t1,int n);
float cpu_util(task *t1,int n);
int gcd(int a, int b);
int lcm(int *a, int n);
int sp_interrupt(task *t1,int tmr,int n);
int min(task *t1,int n,int p);
void update_abs_arrival(task *t1,int n,int k,int all);
void update_abs_deadline(task *t1,int n,int all);
void copy_execution_time(task *t1,int n,int all);

```

```

int timer = 0;

int main()
{
    task *t;
    int n, hyper_period, active_task_id;
    float cpu_utilization;
    printf("Enter number of tasks\n");
    scanf("%d", &n);
    t = malloc(n * sizeof(task));
    get_tasks(t, n);
    cpu_utilization = cpu_util(t, n);
    printf("CPU Utilization %f\n", cpu_utilization);

    if (cpu_utilization < 1)
        printf("Tasks can be scheduled\n");
    else
        printf("Schedule is not feasible\n");

    hyper_period = hyperperiod_calc(t, n);
    copy_execution_time(t, n, ALL);
    update_abs_arrival(t, n, 0, ALL);
    update_abs_deadline(t, n, ALL);

    while (timer <= hyper_period)
    {

        if (sp_interrupt(t, timer, n))
        {
            active_task_id = min(t, n, abs_deadline);

```

```

    }

    if (active_task_id == IDLE_TASK_ID)
    {
        printf("%d Idle\n", timer);
    }

    if (active_task_id != IDLE_TASK_ID)
    {

        if (t[active_task_id].T[execution_copy] != 0)
        {
            t[active_task_id].T[execution_copy]--;
            printf("%d Task %d\n", timer, active_task_id + 1);
        }

        if (t[active_task_id].T[execution_copy] == 0)
        {
            t[active_task_id].instance++;
            t[active_task_id].alive = 0;
            copy_execution_time(t, active_task_id, CURRENT);
            update_abs_arrival(t, active_task_id, t[active_task_id].instance,
CURRENT);

            update_abs_deadline(t, active_task_id, CURRENT);
            active_task_id = min(t, n, abs_deadline);
        }
    }

    ++timer;
}

free(t);

return 0;

```

```

}

void get_tasks(task *t1, int n)
{
    int i = 0;
    while (i < n)
    {
        printf("Enter Task %d parameters\n", i + 1);
        printf("Arrival time: ");
        scanf("%d", &t1->T[arrival]);
        printf("Execution time: ");
        scanf("%d", &t1->T[execution]);
        printf("Deadline time: ");
        scanf("%d", &t1->T[deadline]);
        printf("Period: ");
        scanf("%d", &t1->T[period]);
        t1->T[abs_arrival] = 0;
        t1->T[execution_copy] = 0;
        t1->T[abs_deadline] = 0;
        t1->instance = 0;
        t1->alive = 0;
        t1++;
        i++;
    }
}

```

```

int hyperperiod_calc(task *t1, int n)
{
    int i = 0, ht, a[10];
    while (i < n)

    {

```



```

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

        t1++;

        i++;
    }

    ht = lcm(a, n);

    return ht;
}

int gcd(int a, int b)
{
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}

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

int sp_interrupt(task *t1, int tmr, int n)
{
    int i = 0, n1 = 0, a = 0;
    task *t1_copy;

```

```

t1_copy = t1;
while (i < n)
{
    if (tmr == t1->T[abs_arrival])
    {
        t1->alive = 1;
        a++;
    }
    t1++;
    i++;
}

t1 = t1_copy;
i = 0;

while (i < n)
{
    if (t1->alive == 0)
        n1++;
    t1++;
    i++;
}

if (n1 == n || a != 0)
{
    return 1;
}

return 0;
}

```

```

void update_abs_deadline(task *t1, int n, int all)
{
    int i = 0;
    if (all)
    {
        while (i < n)
        {
            t1->T[abs_deadline] = t1->T[deadline] + t1->T[abs_arrival];
            t1++;
            i++;
        }
    }
    else
    {
        t1 += n;
        t1->T[abs_deadline] = t1->T[deadline] + t1->T[abs_arrival];
    }
}

```

```

void update_abs_arrival(task *t1, int n, int k, int all)
{
    int i = 0;
    if (all)
    {
        while (i < n)
        {
            t1->T[abs_arrival] = t1->T[arrival] + k * (t1->T[period]);
            t1++;
            i++;
        }
    }
}

```

```

        else
        {
            t1 += n;
            t1->T[abs_arrival] = t1->T[arrival] + k * (t1->T[period]);
        }
    }
}

```

```

void copy_execution_time(task *t1, int n, int all)
{
    int i = 0;
    if (all)
    {
        while (i < n)
        {
            t1->T[execution_copy] = t1->T[execution];
            t1++;
            i++;
        }
    }
    else
    {
        t1 += n;
        t1->T[execution_copy] = t1->T[execution];
    }
}

```

```

int min(task *t1, int n, int p)
{
    int i = 0, min = 0x7FFF, task_id = IDLE_TASK_ID;
    while (i < n)
    {

```

```

        if (min > t1->T[p] && t1->alive == 1)
        {
            min = t1->T[p];
            task_id = i;
        }
        t1++;
        i++;
    }
    return task_id;
}

float cpu_util(task *t1, int n)
{
    int i = 0;
    float cu = 0;
    while (i < n)
    {
        cu = cu + (float)t1->T[execution] / (float)t1->T[deadline];
        t1++;
        i++;
    }
    return cu;
}

```

Output:

```

Enter number of tasks
3
Enter Task 1 parameters
Arrival time: 0
Execution time: 3
Deadline time: 7
Period: 20
Enter Task 2 parameters
Arrival time: 0
Execution time: 2
Deadline time: 4
Period: 5
Enter Task 3 parameters
Arrival time: 0
Execution time: 2
Deadline time: 8
Period: 10
CPU Utilization 1.178571
Schedule is not feasible
0 Task 2
1 Task 2
2 Task 1
3 Task 1
4 Task 1
5 Task 3
6 Task 3
7 Task 2
8 Task 2
9 Idle
10 Task 2
11 Task 2
12 Task 3
13 Task 3
14 Idle
15 Task 2
16 Task 2
17 Idle
18 Idle
19 Idle
20 Task 2

```

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

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

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

```
#define n 3
```

```
int main() {
```

```
    srand(time(0));
```

```
    int numbers[n];
```

```

int i;

for (i = 0; i < n; i++) {
    numbers[i] = rand() % 10 + 1;
}

printf("Initial Numbers: ");
for (i = 0; i < n; i++) {
    printf("%d ", numbers[i]);
}
printf("\n");

while (1) {

    int all_zero = 1;
    for (i = 0; i < n; i++) {
        if (numbers[i] > 0) {
            all_zero = 0;
            break;
        }
    }

    if (all_zero) {
        break;
    }

    int selected_index;
    do {
        selected_index = rand() % n;
    } while (numbers[selected_index] == 0);
}

```

```

    numbers[selected_index]--;

    printf("Decrementing number at index %d: ", selected_index);

    for (i = 0; i < n; i++) {
        printf("%d ", numbers[i]);
    }

    printf("\n");
}

printf("All numbers reached 0.\n");

return 0;
}

```

Output:

```

Initial Numbers: 5 7 10
Decrementing number at index 1: 5 6 10
Decrementing number at index 0: 4 6 10
Decrementing number at index 2: 4 6 9
Decrementing number at index 0: 3 6 9
Decrementing number at index 0: 2 6 9
Decrementing number at index 0: 1 6 9
Decrementing number at index 1: 1 5 9
Decrementing number at index 2: 1 5 8
Decrementing number at index 1: 1 4 8
Decrementing number at index 0: 0 4 8
Decrementing number at index 2: 0 4 7
Decrementing number at index 1: 0 3 7
Decrementing number at index 1: 0 2 7
Decrementing number at index 2: 0 2 6
Decrementing number at index 1: 0 1 6
Decrementing number at index 1: 0 0 6
Decrementing number at index 2: 0 0 5
Decrementing number at index 2: 0 0 4
Decrementing number at index 2: 0 0 3
Decrementing number at index 2: 0 0 2
Decrementing number at index 2: 0 0 1
Decrementing number at index 2: 0 0 0
All numbers reached 0.

```


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

```
#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#define BUFFER_SIZE 10

#define NUM_ITEMS 20

int buffer[BUFFER_SIZE];

int fill = 0; // Index to add data by producer

int use = 0; // Index to consume data by consumer

int count = 0; // Number of items in the buffer

sem_t empty; // Semaphore to track empty slots in the buffer

sem_t full; // Semaphore to track the number of items available for consumption

void put(int value) {
    buffer[fill] = value;
    fill = (fill + 1) % BUFFER_SIZE;
    count++;
}

int get() {
    int tmp = buffer[use];
    use = (use + 1) % BUFFER_SIZE;
    count--;
    return tmp;
}

void *producer(void *arg) {
    int i;
```

```

for (i = 0; i < NUM_ITEMS; i++) {
    sem_wait(&empty); // Wait for an empty slot
    put(i);
    printf("Produced: %d\n", i);
    sem_post(&full); // Signal that an item is produced
}
pthread_exit(NULL);
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < NUM_ITEMS; i++) {
        sem_wait(&full); // Wait for an item to be produced
        int value = get();
        printf("Consumed: %d\n", value);
        sem_post(&empty); // Signal that an empty slot is available
    }
    pthread_exit(NULL);
}

int main() {
    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE); // Set empty slots to BUFFER_SIZE
    sem_init(&full, 0, 0); // No items available initially

    pthread_t producer_thread, consumer_thread;

    // Create threads
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

```

```
// Wait for threads to finish
pthread_join(producer_thread, NULL);
pthread_join(consumer_thread, NULL);

// Destroy semaphores
sem_destroy(&empty);
sem_destroy(&full);

return 0;
}
```

Output:

```
Produced: 0
Produced: 1
Produced: 2
Produced: 3
Produced: 4
Produced: 5
Produced: 6
Consumed: 0
Produced: 7
Produced: 8
Produced: 9
Produced: 10
Consumed: 1
Consumed: 2
Consumed: 3
Produced: 11
Produced: 12
Produced: 13
Consumed: 4
Consumed: 5
Consumed: 6
Consumed: 7
Consumed: 8
Consumed: 9
Consumed: 10
Consumed: 11
Consumed: 12
Consumed: 13
Produced: 14
Produced: 15
Consumed: 14
Produced: 16
Produced: 17
Produced: 18
Produced: 19
Consumed: 15
Consumed: 16
Consumed: 17
Consumed: 18
Consumed: 19
```

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

```
#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#define N 5

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (phnum + 4) % N

#define RIGHT (phnum + 1) % N

int state[N];

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

sem_t mutex;

sem_t S[N];

void test(int phnum)
{
    if (state[phnum] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
    {
        // state that eating
        state[phnum] = EATING;

        sleep(2);

        printf("Philosopher %d takes fork %d and %d\n",
            phnum + 1, LEFT + 1, phnum + 1);

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

```

        // sem_post(&S[phnum]) has no effect
        // during takefork
        // used to wake up hungry philosophers
        // during putfork
        sem_post(&S[phnum]);
    }
}

// take up chopsticks
void take_fork(int phnum)
{

    sem_wait(&mutex);

    // state that hungry
    state[phnum] = HUNGRY;

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

    // eat if neighbours are not eating
    test(phnum);

    sem_post(&mutex);

    // if unable to eat wait to be signalled
    sem_wait(&S[phnum]);

    sleep(1);
}

```

```

// put down chopsticks
void put_fork(int phnum)
{

    sem_wait(&mutex);

    // state that thinking
    state[phnum] = THINKING;

    printf("Philosopher %d putting fork %d and %d down\n",
           phnum + 1, LEFT + 1, phnum + 1);
    printf("Philosopher %d is thinking\n", phnum + 1);

    test(LEFT);
    test(RIGHT);

    sem_post(&mutex);
}

void *philosopher(void *num)
{

    while (1)
    {

        int *i = num;

        sleep(1);

        take_fork(*i);
    }
}

```

```

        sleep(0);

        put_fork(*i);
    }
}

int main()
{

    int i;
    pthread_t thread_id[N];

    // initialize the semaphores
    sem_init(&mutex, 0, 1);

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

        sem_init(&S[i], 0, 0);

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

        // create philosopher processes
        pthread_create(&thread_id[i], NULL,
                      philosopher, &phil[i]);

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

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

```



```
pthread_join(thread_id[i], NULL);  
}
```

Output:

```
Philosopher 1 is thinking  
Philosopher 2 is thinking  
Philosopher 3 is thinking  
Philosopher 4 is thinking  
Philosopher 5 is thinking  
Philosopher 1 is Hungry  
Philosopher 2 is Hungry  
Philosopher 3 is Hungry  
Philosopher 4 is Hungry  
Philosopher 5 is Hungry  
Philosopher 5 takes fork 4 and 5  
Philosopher 5 is Eating  
Philosopher 5 putting fork 4 and 5 down  
Philosopher 5 is thinking  
Philosopher 4 takes fork 3 and 4  
Philosopher 4 is Eating  
Philosopher 1 takes fork 5 and 1  
Philosopher 1 is Eating
```

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

```
#include <stdio.h>

int main()
{
    int n, m, all[10][10], req[10][10], ava[10], need[10][10];
    int i, j, k, flag[10], prev[10], c, count = 0, array[10], z = 0;
    printf("Enter number of processes and number of resources required \n");
    scanf("%d %d", &n, &m);
    printf("Enter the max matrix for all process\n", n);
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            scanf("%d", &req[i][j]);
    printf("Enter number of allocated resources %d for each process\n", n);
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            scanf("%d", &all[i][j]);
    printf("Enter number of available resources \n");
    for (i = 0; i < m; i++)
        scanf("%d", &ava[i]);
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            need[i][j] = req[i][j] - all[i][j];
    for (i = 0; i < n; i++)
        flag[i] = 1;
    k = 1;
    while (k)
    {
        k = 0; // Reset the value of k for each iteration of the loop
        for (i = 0; i < n; i++)
        {
            if (flag[i])
```

```

{
    c = 0;
    for (j = 0; j < m; j++)
    {
        if (need[i][j] <= ava[j])
        {
            c++;
        }
    }
    if (c == m)
    {
        array[z++] = i;
        printf("Resources can be allocated to Process:%d and available resources are: ", (i
+ 1));

        for (j = 0; j < m; j++)
        {
            printf("%d ", ava[j]);
        }
        printf("\n");
        for (j = 0; j < m; j++)
        {
            ava[j] += all[i][j];
            all[i][j] = 0;
        }
        flag[i] = 0;
        count++;
    }
}
}

```

```

// Check if the current state is different from the previous state
for (i = 0; i < n; i++)
{
    if (flag[i] != prev[i])
    {
        k = 1;
        break;
    }
}
for (i = 0; i < n; i++)
{
    prev[i] = flag[i];
}
}
printf("\nNeed Matrix:\n");
for (i = 0; i < n; i++) // printing need matrix
{
    for (j = 0; j < m; j++)
        printf("%d ", need[i][j]);
    printf("\n");
}

if (count == n)
{
    printf("\nSystem is in safe mode \n<");
    for (i = 0; i < n; i++)
        printf("P%d ", (array[i] + 1));
    printf(">\n");
}
else
{

```

```

        printf("\nSystem is not in safe mode deadlock occurred \n");
    }

    return 0;
}

```

Output:

```

Enter number of processes and number of resources required
5 3
Enter the max matrix for all process
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter number of allocated resources 5 for each process
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter number of available resources
3 3 2
Resources can be allocated to Process:2 and available resources are: 3 3 2
Resources can be allocated to Process:4 and available resources are: 5 3 2
Resources can be allocated to Process:5 and available resources are: 7 4 3
Resources can be allocated to Process:1 and available resources are: 7 4 5
Resources can be allocated to Process:3 and available resources are: 7 5 5

Need Matrix:
7 4 3
1 2 2
6 0 0
0 1 1
4 3 1

System is in safe mode
<P2 P4 P5 P1 P3 >

```

8. Write a C program to simulate deadlock detection

```
#include <stdio.h>

#include <conio.h>

int max[100][100];
int alloc[100][100];
int need[100][100];
int avail[100];
int n, r;

void input();
void show();
void cal();

int main()
{
    int i, j;

    printf("***** Deadlock Detection Algo *****\n");

    input();

    show();

    cal();

    getch();

    return 0;
}

void input()
{
    int i, j;

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

    scanf("%d", &n);

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

    scanf("%d", &r);

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

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

```

        for (j = 0; j < r; j++)
        {
            scanf("%d", &max[i][j]);
        }
    }
    printf("Enter the Allocation Matrix\n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < r; j++)
        {
            scanf("%d", &alloc[i][j]);
        }
    }
    printf("Enter the available Resources\n");
    for (j = 0; j < r; j++)
    {
        scanf("%d", &avail[j]);
    }
}

void show()
{
    int i, j;
    printf("Process\t Allocation\t Request\t Available\t");
    for (i = 0; i < n; i++)
    {
        printf("\nP%d\t ", i + 1);
        for (j = 0; j < r; j++)
        {
            printf("%d ", alloc[i][j]);
        }
        printf("\t");
    }
}

```

```

        for (j = 0; j < r; j++)
        {
            printf("%d ", max[i][j]);
        }
        printf("\t");
        if (i == 0)
        {
            for (j = 0; j < r; j++)
                printf("%d ", avail[j]);
        }
    }
}

void cal()
{
    int finish[100], temp, need[100][100], flag = 1, k, c1 = 0;
    int dead[100];
    int safe[100];
    int i, j;
    for (i = 0; i < n; i++)
    {
        finish[i] = 0;
    }
    // find need matrix
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < r; j++)
        {
            need[i][j] = max[i][j] - alloc[i][j];
        }
    }
    while (flag)

```



```

{
    flag = 0;
    for (i = 0; i < n; i++)
    {
        int c = 0;
        for (j = 0; j < r; j++)
        {
            if ((finish[i] == 0) && (need[i][j] <= avail[j]))
            {
                c++;
                if (c == r)
                {
                    for (k = 0; k < r; k++)
                    {
                        avail[k] += alloc[i][j];
                        finish[i] = 1;
                        flag = 1;
                    }
                    // printf("\nP%d",i);
                    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("\n\nSystem is in Deadlock and the Deadlock process are\n");
    for (i = 0; i < n; i++)
    {
        printf("P%d\t", dead[i]);
    }
}
else
{
    printf("\nNo Deadlock Occur");
}
}

```

Output:

```

***** Deadlock Detection Algo *****
Enter the no of Processes      5
Enter the no of resource instances    3
Enter the request Matrix
0 0 0
2 0 2
0 0 0
1 0 0
0 0 2
Enter the Allocation Matrix
0 1 0
2 0 0
3 0 3
2 1 1
0 0 2
Enter the available Resources
0 0 0

```

Process	Allocation	Request	Available
P1	0 1 0	0 0 0	0 0 0
P2	2 0 0	2 0 2	
P3	3 0 3	0 0 0	
P4	2 1 1	1 0 0	
P5	0 0 2	0 0 2	

```

No Deadlock Occur

```

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>
```

```
void print(int processSize[], int allocation[], int n)
```

```
{
    int i;
    printf("\nProcess No.\tProcess Size\tBlock no.\n");
    for (i = 0; i < n; i++)
    {
        printf(" %i\t\t", i + 1);
        printf("%i\t", processSize[i]);
        if (allocation[i] != -1)
            printf("%i", allocation[i] + 1);
        else
            printf("Not Allocated");
        printf("\n");
    }
}
```

```
void firstFit(int blockSize[], int m, int processSize[], int n)
```

```
{
    int i, j;
    // Stores block id of the
    // block allocated to a process
    int allocation[n];

    // Initially no block is assigned to any process
    for (i = 0; i < n; i++)
    {
```

```

        allocation[i] = -1;
    }

    // pick each process and find suitable blocks
    // according to its size and assign to it
    for (i = 0; i < n; i++) // here, n -> number of processes
    {
        for (j = 0; j < m; j++) // here, m -> number of blocks
        {
            if (blockSize[j] >= processSize[i])
            {
                // allocating block j to the ith process
                allocation[i] = j;

                // Reduce available memory in this block.
                blockSize[j] -= processSize[i];

                break; // go to the next process in the queue
            }
        }
    }

    print(processSize, allocation, n);
}

void bestFit(int blockSize[], int m, int processSize[], int n)
{
    // Stores block id of the block allocated to a process
    int allocation[n];
    int i, j, bestIdx;

    // Initially no block is assigned to any process
    for (i = 0; i < n; i++)

```

```

allocation[i] = -1;

// pick each process and find suitable blocks
// according to its size ad assign to it
for (i = 0; i < n; i++)
{
    // Find the best fit block for current process
    bestIdx = -1;
    for (j = 0; j < m; j++)
    {
        if (blockSize[j] >= processSize[i])
        {
            if (bestIdx == -1)
                bestIdx = j;
            else if (blockSize[bestIdx] > blockSize[j])
                bestIdx = j;
        }
    }

    // If we could find a block for current process
    if (bestIdx != -1)
    {
        // allocate block j to p[i] process
        allocation[i] = bestIdx;

        // Reduce available memory in this block.
        blockSize[bestIdx] -= processSize[i];
    }
}

print(processSize, allocation, n);

```

```

}

// Function to allocate memory to blocks as per worst fit
// algorithm
void worstFit(int blockSize[], int m, int processSize[],
              int n)
{
    // Stores block id of the block allocated to a
    // process
    int allocation[n], i, j, wstIdx;

    // Initially no block is assigned to any process
    for (i = 0; i < n; i++)
        allocation[i] = -1;

    // pick each process and find suitable blocks
    // according to its size ad assign to it
    for (i = 0; i < n; i++)
    {
        // Find the best fit block for current process
        wstIdx = -1;
        for (j = 0; j < m; j++)
        {
            if (blockSize[j] >= processSize[i])
            {
                if (wstIdx == -1)
                    wstIdx = j;
                else if (blockSize[wstIdx] < blockSize[j])
                    wstIdx = j;
            }
        }
    }
}

```

```

        // If we could find a block for current process
        if (wstIdx != -1)
        {
            // allocate block j to p[i] process
            allocation[i] = wstIdx;

            // Reduce available memory in this block.
            blockSize[wstIdx] -= processSize[i];
        }
    }

    print(processSize, allocation, n);
}

void main()
{
    int m,i; // number of blocks in the memory
    int n; // number of processes in the input queue
    int blockSize[20];
    int processSize[20];
    int choice;

    printf("Enter the number of blocks\n");
    scanf("%d",&m);

    printf("Enter the number of processes\n");
    scanf("%d",&n);

    printf("Enter the block size\n");
    for(i=0;i<m;i++)
    {
        scanf("%d",&blockSize[i]);
    }

    printf("Enter the process size\n");
    for(i=0;i<n;i++)

```



```

{
    scanf("%d",&processSize[i]);
}
printf("\n1.First-fit\n2.Best-fit\n3.Worst-fit\n");
printf("Enter your choice\n");
scanf("%d",&choice);
switch(choice)
{
    case 1:firstFit(blockSize, m, processSize, n);
        break;
    case 2:bestFit(blockSize,m,processSize,n);
        break;
    case 3:worstFit(blockSize,m,processSize,n);
        break;
    default:printf("invalid choice\n");
}

}

```

Output:

```
Enter the number of blocks
6
Enter the number of processes
4
Enter the block size
200
700
500
300
100
400
Enter the process size
315
427
250
550
```

```
1.First-fit
2.Best-fit
3.Worst-fit
Enter your choice
1
```

Process No.	Process Size	Block no.
1	315	2
2	427	3
3	250	2
4	550	Not Allocated

```
Enter the number of blocks
6
Enter the number of processes
4
Enter the block size
200
700
500
300
100
400
Enter the process size
315
427
250
550
```

```
1.First-fit
2.Best-fit
3.Worst-fit
Enter your choice
2
```

Process No.	Process Size	Block no.
1	315	6
2	427	3
3	250	4
4	550	2

Enter the number of blocks

6

Enter the number of processes

4

Enter the block size

200

700

500

300

100

400

Enter the process size

315

427

250

550

1.First-fit

2.Best-fit

3.Worst-fit

Enter your choice

3

Process No.	Process Size	Block no.
1	315	2
2	427	3
3	250	6
4	550	Not Allocated

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

```
#include <stdio.h>

void main()
{
    int ms, ps, nop, np, rempages, i, j, x, y, pa, offset;
    int s[10], fno[10][20];
    printf("\nEnter the memory size -- ");
    scanf("%d", &ms);

    printf("\nEnter the page size -- ");
    scanf("%d", &ps);

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

    printf("\nEnter number of processes -- ");
    scanf("%d", &np);
    rempages = nop;
    for (i = 1; i <= np; i++)
    {

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

        if (s[i] > rempages)
        {

            printf("\nMemory is Full");
            break;
        }
    }
}
```

```

    rempages = rempages - s[i];

    printf("\nEnter pagetable for p[%d] --- ", i);
    for (j = 0; j < s[i]; j++)
        scanf("%d", &fno[i][j]);
}

printf("\nEnter Logical Address to find Physical Address ");
printf("\nEnter process no. and pagenumber and offset -- ");

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

if (x > np || y >= s[i] || offset >= ps)
    printf("\nInvalid Process or Page Number or offset");

else
{
    pa = fno[x][y] * ps + offset;
    printf("\nThe Physical Address is -- %d", pa);
}
}

```

Output:

```
Enter the memory size -- 1000
Enter the page size -- 100
The no. of pages available in memory are -- 10
Enter number of processes -- 3
Enter no. of pages required for p[1]-- 4
Enter pagetable for p[1] --- 8 6 9 5
Enter no. of pages required for p[2]-- 5
Enter pagetable for p[2] --- 1 4 5 7 3
Enter no. of pages required for p[3]-- 5
Memory is Full
Enter Logical Address to find Physical Address
Enter process no. and pagenumber and offset -- 2 3 60
The Physical Address is -- 760
```

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

```
#include<stdio.h>

int n,nf;
int in[100];
int p[50];
int hit=0;
int i,j,k;
int pgfaultcnt=0;

void getData()
{
    printf("\nEnter length of page reference sequence:");
    scanf("%d",&n);
    printf("\nEnter the page reference sequence:");
    for(i=0; i<n; i++)
        scanf("%d",&in[i]);
    printf("\nEnter no of frames:");
    scanf("%d",&nf);
}

void initialize()
{
    pgfaultcnt=0;
    for(i=0; i<nf; i++)
        p[i]=9999;
}

int isHit(int data)
{
    hit=0;
    for(j=0; j<nf; j++)
```

```

{
    if(p[j]==data)
    {
        hit=1;
        break;
    }

}

return hit;
}

```

```

int getHitIndex(int data)
{
    int hitind;
    for(k=0; k<nf; k++)
    {
        if(p[k]==data)
        {
            hitind=k;
            break;
        }
    }
    return hitind;
}

```

```

void dispPages()
{
    for (k=0; k<nf; k++)
    {
        if(p[k]!=9999)

```



```

        printf(" %d",p[k]);
    }

}

void dispPgFaultCnt()
{
    printf("\nTotal no of page faults:%d",pgfaultcnt);
}

void fifo()
{
    initialize();
    for(i=0; i<n; i++)
    {
        printf("\nFor %d :",in[i]);

        if(isHit(in[i])==0)
        {

            for(k=0; k<nf-1; k++)
                p[k]=p[k+1];

            p[k]=in[i];
            pgfaultcnt++;
            dispPages();
        }
        else
            printf("No page fault");
    }
    dispPgFaultCnt();
}

```

```
}
```

```
void optimal()
```

```
{
```

```
    initialize();
```

```
    int near[50];
```

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

```
    {
```

```
        printf("\nFor %d :",in[i]);
```

```
        if(isHit(in[i])==0)
```

```
        {
```

```
            for(j=0; j<nf; j++)
```

```
            {
```

```
                int pg=p[j];
```

```
                int found=0;
```

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

```
                {
```

```
                    if(pg==in[k])
```

```
                    {
```

```
                        near[j]=k;
```

```
                        found=1;
```

```
                        break;
```

```
                    }
```

```
                else
```

```
                    found=0;
```

```
                }
```

```
            if(!found)
```

```

        near[j]=9999;
    }
    int max=-9999;
    int repindex;
    for(j=0; j<nf; j++)
    {
        if(near[j]>max)
        {
            max=near[j];
            repindex=j;
        }
    }
    p[repindex]=in[i];
    pgfaultcnt++;

    dispPages();
}
else
    printf("No page fault");
}
dispPgFaultCnt();
}

void lru()
{
    initialize();

    int least[50];
    for(i=0; i<n; i++)
    {

```

```
printf("\nFor %d :",in[i]);
```

```
if(isHit(in[i])==0)
```

```
{
```

```
    for(j=0; j<nf; j++)
```

```
    {
```

```
        int pg=p[j];
```

```
        int found=0;
```

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

```
        {
```

```
            if(pg==in[k])
```

```
            {
```

```
                least[j]=k;
```

```
                found=1;
```

```
                break;
```

```
            }
```

```
            else
```

```
                found=0;
```

```
        }
```

```
        if(!found)
```

```
            least[j]=-9999;
```

```
    }
```

```
    int min=9999;
```

```
    int repindex;
```

```
    for(j=0; j<nf; j++)
```

```
    {
```

```
        if(least[j]<min)
```

```
        {
```

```
            min=least[j];
```

```
            repindex=j;
```

```

        }
    }
    p[repindex]=in[i];
    pgfaultcnt++;

    dispPages();
}
else
    printf("No page fault!");
}
dispPgFaultCnt();
}

int main()
{
    int choice;
    while(1)
    {
        printf("\nPage Replacement Algorithms\n1.Enter
data\n2.FIFO\n3.Optimal\n4.LRU\n5.Exit\nEnter your choice:");

        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                getData();
                break;
            case 2:
                fifo();
                break;
            case 3:
                optimal();

```

```
        break;
    case 4:
        lru();
        break;
    default:
        return 0;
        break;
    }
}
}
```

Output:

Page Replacement Algorithms

1.Enter data

2.FIFO

3.Optimal

4.LRU

5.Exit

Enter your choice:1

Enter length of page reference sequence:6

Enter the page reference sequence:1 2 5 3 1 2

Enter no of frames:3

Page Replacement Algorithms

1.Enter data

2.FIFO

3.Optimal

4.LRU

5.Exit

Enter your choice:2

For 1 : 1

For 2 : 1 2

For 5 : 1 2 5

For 3 : 2 5 3

For 1 : 5 3 1

For 2 : 3 1 2

Total no of page faults:6

Page Replacement Algorithms

1.Enter data

2.FIFO

3.Optimal

4.LRU

5.Exit

Enter your choice:3

For 1 : 1

For 2 : 1 2

For 5 : 1 2 5

For 3 : 1 2 3

For 1 :No page fault

For 2 :No page fault

Total no of page faults:4

Page Replacement Algorithms

1.Enter data

2.FIFO

3.Optimal

4.LRU

5.Exit

Enter your choice:4

For 1 : 1

For 2 : 1 2

For 5 : 1 2 5

For 3 : 3 2 5

For 1 : 3 1 5

For 2 : 3 1 2

Total no of page faults:6

Page Replacement Algorithms

1.Enter data

2.FIFO

3.Optimal

4.LRU

5.Exit

Enter your choice:5

12. Write a C program to simulate disk scheduling algorithms a) FCFS b) SCAN c) C-SCAN

a)FCFS

```
/*FCFS*/
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int RQ[100], i, n, TotalHeadMoment = 0, initial;
    printf("Enter the number of Requests\n");
    scanf("%d", &n);
    printf("Enter the Requests sequence\n");
    for (i = 0; i < n; i++)
        scanf("%d", &RQ[i]);
    printf("Enter initial head position\n");
    scanf("%d", &initial);

    // logic for FCFS disk scheduling

    for (i = 0; i < n; i++)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }

    printf("Total head moment is %d", TotalHeadMoment);
    return 0;
}
```

Output:

```
Enter the number of Requests
8
Enter the Requests sequence
98 183 37 122 14 134 65 67
Enter initial head position
53
Total head moment is 660
```

b)SCAN

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

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

```
int main()
```

```
{
```

```
    int RQ[100], i, j, n, TotalHeadMoment = 0, initial, size, move;
```

```
    printf("Enter the number of Requests\n");
```

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

```
    printf("Enter the Requests sequence\n");
```

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

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

```
    printf("Enter initial head position\n");
```

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

```
    printf("Enter total disk size\n");
```

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

```
    printf("Enter the head movement direction for high 1 and for low 0\n");
```

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

```
    // logic for Scan disk scheduling
```

```
    /*logic for sort the request array */
```

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

```
    {
```

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

```
        {
```

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

```
            {
```

```

        int temp;

        temp = RQ[j];

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

        RQ[j + 1] = temp;
    }
}

int index;

for (i = 0; i < n; i++)
{
    if (initial < RQ[i])
    {
        index = i;

        break;
    }
}

// if movement is towards high value
if (move == 1)
{
    for (i = index; i < n; i++)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);

        initial = RQ[i];
    }

    // last movement for max size
    TotalHeadMoment = TotalHeadMoment + abs(size - RQ[i - 1] - 1);

    initial = size - 1;

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

```

```

        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
}
// if movement is towards low value
else
{
    for (i = index - 1; i >= 0; i--)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
    // last movement for min size
    TotalHeadMoment = TotalHeadMoment + abs(RQ[i + 1] - 0);
    initial = 0;
    for (i = index; i < n; i++)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
}

printf("Total head movement is %d", TotalHeadMoment);
return 0;
}

```

Output:

```

Enter the number of Requests
8
Enter the Requests sequence
98 183 37 122 14 124 65 67
Enter initial head position
53
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
0
Total head movement is 236

```

c)C-SCAN

```

#include <stdio.h>

#include <stdlib.h>

int main()
{
    int RQ[100], i, j, n, TotalHeadMoment = 0, initial, size, move;

    printf("Enter the number of Requests\n");
    scanf("%d", &n);
    printf("Enter the Requests sequence\n");
    for (i = 0; i < n; i++)
        scanf("%d", &RQ[i]);
    printf("Enter initial head position\n");
    scanf("%d", &initial);
    printf("Enter total disk size\n");
    scanf("%d", &size);
    printf("Enter the head movement direction for high 1 and for low 0\n");
    scanf("%d", &move);

    // logic for C-Scan disk scheduling

    /*logic for sort the request array */
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n - i - 1; j++)
        {

```

```

        if (RQ[j] > RQ[j + 1])
        {
            int temp;

            temp = RQ[j];

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

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

int index;
for (i = 0; i < n; i++)
{
    if (initial < RQ[i])
    {
        index = i;

        break;
    }
}

// if movement is towards high value
if (move == 1)
{
    for (i = index; i < n; i++)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);

        initial = RQ[i];
    }

    // last movement for max size
    TotalHeadMoment = TotalHeadMoment + abs(size - RQ[i - 1] - 1);
}

```

```

/*movement max to min disk */
TotalHeadMoment = TotalHeadMoment + abs(size - 1 - 0);
initial = 0;
for (i = 0; i < index; i++)
{
    TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
    initial = RQ[i];
}
}
// if movement is towards low value
else
{
    for (i = index - 1; i >= 0; i--)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
    // last movement for min size
    TotalHeadMoment = TotalHeadMoment + abs(RQ[i + 1] - 0);
    /*movement min to max disk */
    TotalHeadMoment = TotalHeadMoment + abs(size - 1 - 0);
    initial = size - 1;
    for (i = n - 1; i >= index; i--)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
}

printf("Total head movement is %d", TotalHeadMoment);
return 0;

```

}

Output:

```
Enter the number of Requests
8
Enter the Requests sequence
98 183 37 122 14 124 65 67
Enter initial head position
53
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
0
Total head movement is 386
```


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

a)SSTF

```
#include <stdio.h>

#include <stdlib.h>

int main()
{
    int RQ[100], i, n, TotalHeadMoment = 0, initial, count = 0;

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

    scanf("%d", &n);

    printf("Enter the Requests sequence\n");

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

    printf("Enter initial head position\n");

    scanf("%d", &initial);

    // logic for sstf disk scheduling

    /* loop will execute until all process is completed*/
    while (count != n)
    {
        int min = 1000, d, index;

        for (i = 0; i < n; i++)
        {
            d = abs(RQ[i] - initial);

            if (min > d)
            {
                min = d;

                index = i;
            }
        }

        TotalHeadMoment = TotalHeadMoment + min;
```

```

        initial = RQ[index];

        // 1000 is for max

        // you can use any number

        RQ[index] = 1000;

        count++;

    }

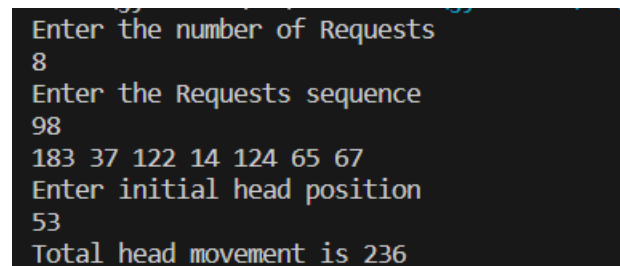
    printf("Total head movement is %d", TotalHeadMoment);

    return 0;

}

```

Output:



```

Enter the number of Requests
8
Enter the Requests sequence
98
183 37 122 14 124 65 67
Enter initial head position
53
Total head movement is 236

```

b)LOOK

```

#include <stdio.h>

#include <stdlib.h>

int main()

{

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

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

    scanf("%d", &n);

    printf("Enter the Requests sequence\n");

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

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

    printf("Enter initial head position\n");

    scanf("%d", &initial);

    printf("Enter total disk size\n");

    scanf("%d", &size);

```

```
printf("Enter the head movement direction for high 1 and for low 0\n");  
scanf("%d", &move);
```

```
// logic for look disk scheduling
```

```
/*logic for sort the request array */
```

```
for (i = 0; i < n; i++)  
{  
    for (j = 0; j < n - i - 1; j++)  
    {  
        if (RQ[j] > RQ[j + 1])  
        {  
            int temp;  
            temp = RQ[j];  
            RQ[j] = RQ[j + 1];  
            RQ[j + 1] = temp;  
        }  
    }  
}
```

```
int index;  
for (i = 0; i < n; i++)  
{  
    if (initial < RQ[i])  
    {  
        index = i;  
        break;  
    }  
}
```

```
// if movement is towards high value
```

```

if (move == 1)
{
    for (i = index; i < n; i++)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }

    for (i = index - 1; i >= 0; i--)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
}

// if movement is towards low value
else
{
    for (i = index - 1; i >= 0; i--)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }

    for (i = index; i < n; i++)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
}

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

```

```
    return 0;
}
```

Output:

```
Enter the number of Requests
8
Enter the Requests sequence
98 183 37 122 14 124 65 67
Enter initial head position
53
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
0
Total head movement is 208
```

c)C-LOOK

```
#include <stdio.h>

#include <stdlib.h>

int main()
{
    int RQ[100], i, j, n, TotalHeadMoment = 0, initial, size, move;

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

    scanf("%d", &n);

    printf("Enter the Requests sequence\n");

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

    printf("Enter initial head position\n");

    scanf("%d", &initial);

    printf("Enter total disk size\n");

    scanf("%d", &size);

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

    scanf("%d", &move);


    // logic for C-look disk scheduling


    /*logic for sort the request array */

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

```

{
    for (j = 0; j < n - i - 1; j++)
    {
        if (RQ[j] > RQ[j + 1])
        {
            int temp;
            temp = RQ[j];
            RQ[j] = RQ[j + 1];
            RQ[j + 1] = temp;
        }
    }
}

int index;
for (i = 0; i < n; i++)
{
    if (initial < RQ[i])
    {
        index = i;
        break;
    }
}

// if movement is towards high value
if (move == 1)
{
    for (i = index; i < n; i++)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
}

```

```

    for (i = 0; i < index; i++)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
}
// if movement is towards low value
else
{
    for (i = index - 1; i >= 0; i--)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }

    for (i = n - 1; i >= index; i--)
    {
        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
        initial = RQ[i];
    }
}

printf("Total head movement is %d", TotalHeadMoment);
return 0;
}

```

Output:

```
Enter the number of Requests
8
Enter the Requests sequence
98 183 37 122 14 124 65 67
Enter initial head position
53
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
0
Total head movement is 326
D:\justhika\oc\oc>
```