# VISVESVARAYA TECHNOLOGICAL UNIVERSITY
**"JnanaSangama", Belgaum -590014, Karnataka.**

**LAB REPORT**
**on**

# OPERATING SYSTEMS

*Submitted by*

**DHANUSH S (1BM21CS053)**

*in partial fulfillment for the award of the degree of*
**BACHELOR OF ENGINEERING**
*in*
**COMPUTER SCIENCE AND ENGINEERING**

**B.M.S. COLLEGE OF ENGINEERING**
**(Autonomous Institution under VTU)**
**BENGALURU-560019**
**June-2023 to September-2023**

# B. M. S. College of Engineering,

**Bull Temple Road, Bangalore 560019**

(Affiliated To Visvesvaraya Technological University, Belgaum)

## Department of Computer Science and Engineering



## <u>CERTIFICATE</u>

This is to certify that the Lab work entitled "OPERATING SYSTEMS" carried out by **DHANUSH S(1BM21CS053),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the academic semester June-2023 to September-2023.  The Lab report has been approved as it satisfies the academic requirements in respect of a OPERATING SYSTEMS **(22CS4PCOPS)** work prescribed for the said degree.

Dr. Nandhini  Vineeth                                        Dr. Jyothi S Nayak

Associate Professor                                          Professor and Head

Department of CSE                                            Department of CSE

BMSCE, Bengaluru                                             BMSCE, Bengaluru

# Content

| Lab Exp No. | Program Details | Page No. |
|---|---|---|
| exp 1 | Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.<br>● FCFS<br>● SJF (pre-emptive & Non-pre-emptive) | 5-13 |
| exp 2 | Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.<br>● Priority (pre-emptive & Non-pre-emptive)<br>● Round Robin (Experiment with different quantum sizes for RR algorithm) | 14-25 |
| exp 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. | 26-30 |
| exp 4 | Write a C program to simulate Real-Time CPU Scheduling algorithms:<br>● Rate- Monotonic<br>● Earliest-deadline First<br>● Proportional scheduling | 31-43 |
| exp 5 | Write a C program to simulate producer-consumer problem using semaphores. | 44-46 |
| exp 6 | Write a C program to simulate the concept of Dining-Philosophers problem. | 47-51 |
| exp 7 | Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance. | 52-58 |
| exp 8 | Write a C program to simulate deadlock detection. | 59-62 |

| | | |
|---|---|---|
| exp 9 | Write a C program to simulate the following contiguous memory allocation techniques<br>● Worst-fit<br>● Best-fit<br>● First-fit | 63-70 |
| exp 10 | Write a C program to simulate page replacement algorithms<br>● FIFO<br>● LRU<br>● Optimal | 71-80 |
| exp 11 | Write a C program to simulate disk scheduling algorithms<br>● FCFS<br>● SCAN<br>● C-SCAN | 81-89 |

## 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. |

# EXPERIMENT – 1

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

- FCFS
- SJF (pre-emptive & Non-pre-emptive)

**FCFS:**

```c
#include<stdio.h>
int main()
{
    int bt[20], at[20], wt[20], tat[20],ct[20], i, n;
    float wtavg, tatavg;
    printf("\nEnter the number of processes: ");
    scanf("%d", &n);
    for(i=1;i<=n;i++)
    {
        printf("\nEnter Arrival Time and Burst Time for Process %d: ", i);
        scanf("%d %d", &at[i], &bt[i]);
    }
    for(i=1; i<=n; i++)
    {
        ct[i]=bt[i]+ct[i-1];
    }
    for(i=1; i<=n; i++)
```

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

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

    tatavg=tatavg+tat[i];

    wtavg=wtavg+wt[i];

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

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

{

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

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

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

}
```

**OUTPUT:**

```
Enter the number of processes: 4

Enter Arrival Time and Burst Time for Process 1: 0 3

Enter Arrival Time and Burst Time for Process 2: 1 6

Enter Arrival Time and Burst Time for Process 3: 4 4

Enter Arrival Time and Burst Time for Process 4: 6 2
        PROCESS          BURST TIME        WAITING TIME      TURNAROUND TIME

        P1                  3                0                3
        P2                  6                2                8
        P3                  4                5                9
        P4                  2                7                9
Average Turnaround Time: 7.250000
Average Waiting Time: 3.500000
```

```
Enter the number of processes: 5

Enter Arrival Time and Burst Time for Process 1: 0 1

Enter Arrival Time and Burst Time for Process 2: 0 10

Enter Arrival Time and Burst Time for Process 3: 3 2

Enter Arrival Time and Burst Time for Process 4: 5 1

Enter Arrival Time and Burst Time for Process 5: 10 5
        PROCESS          BURST TIME        WAITING TIME      TURNAROUND TIME

        P1               1                 0                 1
        P2               10                1                 11
        P3               2                 8                 10
        P4               1                 8                 9
        P5               5                 4                 9
Average Turnaround Time: 8.000000
Average Waiting Time: 4.200000
```

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

#include<stdio.h>

int at[20], cput[20];

void sjf(int n)

{

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

  float awt=0, atat=0,sum_burst_time=0;

  int sum=0,i,j, smallest;

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

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

  {

     cput1[i]=cput[i];

     sum_burst_time += cput[i];

  }

```c
    cput1[9]=9999;
    while(sum < sum_burst_time)
    {
        smallest = 9;
        for (i = 0; i < n; i++)
        {
            if (at[i] <= sum && cput1[i] > 0 && cput1[i] < cput1[smallest])
            smallest = i;
        }
        printf("\t P[%d] \t\t %d \t\t %d\n", smallest , sum + cput1[smallest]-
        at[smallest], sum - at[smallest]);
        awt += sum + cput1[smallest] - at[smallest];
        atat += sum - at[smallest];
        sum += cput1[smallest];
        cput1[smallest] = 0;
    }
    awt = awt/n;
    atat =atat/n;
    printf("\nAverage Waiting Time -- %f", awt);
    printf("\nAverage Turnaround Time -- %f\n", atat);
}
void main()
{
    int n,i;
    printf("Enter the number of processes\n");
    scanf("%d",&n);
```

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

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

        {

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

        }

        sjf(n);

}
```

**OUTPUT:**

```
Enter arrival time and cpu time for each process respectively
0 3
1 6
4 4
6 2
        PROCESS             TURNAROUND TIME            WAITING TIME
        P[0]            3                          0
        P[1]            8                          2
        P[3]            5                          3
        P[2]            11                         7

Average Waiting Time -- 6.750000
Average Turnaround Time -- 3.000000
```

```
Enter the number of processes: 5
Enter arrival time and cpu time for each process respectively
0 10
0 1
3 2
5 2
2 5
        PROCESS             TURNAROUND TIME            WAITING TIME
        P[1]            1                          0
        P[0]            11                         1
        P[2]            10                         8
        P[3]            10                         8
        P[4]            18                         13

Average Waiting Time -- 10.000000
Average Turnaround Time -- 6.000000
```

**SJF (Pre-emptive):**

```c
#include<stdio.h>

int at[20], cput[20];

void srtf(int n)

{

   int remaining_time[20], tat[20], wt[20], completion_time[20], smallest, time, i,
count = 0;

   float awt=0,atat=0;

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

     remaining_time[i] = cput[i];

   time = 0;

   while (count != n)

   {

     smallest = -1;

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

     {

       if (at[i] <= time && remaining_time[i] > 0)

       {

         if (smallest == -1 || remaining_time[i] < remaining_time[smallest])

           smallest = i;

       }

     }


     if (smallest == -1) {

       time++;

       continue;
```

```c
        }

    remaining_time[smallest]--;

    if (remaining_time[smallest] == 0) {
        count++;
        completion_time[smallest] = time + 1;
        wt[smallest] = completion_time[smallest] - at[smallest] - cput[smallest];
        tat[smallest] = completion_time[smallest] - at[smallest];
    }

    time++;
    }
    for(i=0;i<n;i++)
    {
        awt+=wt[i];
        atat += tat[i];
    }
    awt = awt/n;
    atat =atat/n;
    printf("\nProcess\tArrival Time\tCPU Time\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < n; i++)
    {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", i, at[i], cput[i], wt[i], tat[i]);
    }
```

```c
    printf("\nAverage Waiting Time -- %f", awt);
    printf("\nAverage Turnaround Time -- %f\n", atat);


}
void main()
{
    int n,i,choice;
    printf("Enter the number of processes\n");
    scanf("%d",&n);
    printf("Enter arrival time and cpu time for each process respectively\n");
    for(i =0;i<n;i++)
    {
        scanf("%d %d",&at[i],&cput[i]);
    }
    srtf(n);
}
```

**OUTPUT:**

```
Enter the number of processes
4
Enter arrival time and cpu time for each process respectively
0 3
1 6
4 4
6 2

Process Arrival Time     CPU Time          Waiting Time      Turnaround Time
0        0               3                 0                 3
1        1               6                 8                 14
2        4               4                 0                 4
3        6               2                 2                 4

Average Waiting Time -- 2.500000
Average Turnaround Time -- 6.250000
```

```
Enter the number of processes
03
Enter arrival time and cpu time for each process respectively
0 8
0 4
1 1

Process Arrival Time     CPU Time          Waiting Time      Turnaround Time
0        0               8                 5                 13
1        0               4                 1                 5
2        1               1                 0                 1

Average Waiting Time -- 2.000000
Average Turnaround Time -- 6.333333
```

# EXPERIMENT – 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)

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

```c
#include <stdio.h>
#include <stdbool.h>
#include<stdlib.h>
#define MAX_PROCESSES 10
struct Process
{
  int pid;
  int arrival_time;
  int burst_time;
  int priority;
  int remaining_time;
  int turnaround_time;
  int waiting_time;
};
void priority_nonpreemptive(struct Process processes[], int n) {
  int i,j,count=0,m;
  for(i=0;i<n;i++)
  {
    if(processes[i].arrival_time==0)
    count++;
  }
  if(count==n||count==1)
  {
```

```c
    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;
```

```c
        double total_turnaround_time = 0;
        double total_waiting_time = 0;
        for (i = 0; i < n; i++)
        {
            total_time += processes[i].burst_time;
            processes[i].turnaround_time = total_time - processes[i].arrival_time;
            processes[i].waiting_time = processes[i].turnaround_time –
            processes[i].burst_time;
            total_turnaround_time += processes[i].turnaround_time;
            total_waiting_time += processes[i].waiting_time;
        }
        printf("Process\tTurnaround Time\tWaiting Time\n");
        for (i = 0; i < n; i++)
        {
            printf("%d\t%d\t\t%d\n", processes[i].pid, processes[i].turnaround_time,
            processes[i].waiting_time);
        }
        printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
        printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
    }
    void priority_preemptive(struct Process processes[], int n) {
        int total_time = 0,i;
        int completed = 0;
        while (completed < n) {
            int highest_priority = -1;
            int next_process = -1;
            for (i = 0; i < n; i++) {
                if (processes[i].arrival_time <= total_time && processes[i].remaining_time
                > 0) {
                    if (highest_priority == -1 || processes[i].priority < highest_priority) {
                        highest_priority = processes[i].priority;
                        next_process = i;
                    }
```

```c
            }
        }
        if (next_process == -1) {
        total_time++;
        continue;
        }
        processes[next_process].remaining_time--;
        total_time++;
        if (processes[next_process].remaining_time == 0) {
            completed++;
            processes[next_process].turnaround_time = total_time -
processes[next_process].arrival_time;
            processes[next_process].waiting_time =
processes[next_process].turnaround_time - processes[next_process].burst_time;
        }
}
    double total_turnaround_time = 0;
    double total_waiting_time = 0;
    printf("Process\tTurnaround Time\tWaiting Time\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\n", processes[i].pid, processes[i].turnaround_time,
processes[i].waiting_time);
        total_turnaround_time += processes[i].turnaround_time;
        total_waiting_time += processes[i].waiting_time;
    }
    printf("Average Turnaround Time: %.2f\n", total_turnaround_time / n);
    printf("Average Waiting Time: %.2f\n", total_waiting_time / n);
}

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

```c
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++)
    {
        printf("Process %d\n", i + 1);
        printf("Enter arrival time, burst time, priority: ");
        scanf("%d %d %d", &processes[i].arrival_time, &processes[i].burst_time,
&processes[i].priority);
        processes[i].pid = i + 1;
        processes[i].remaining_time = processes[i].burst_time;
        processes[i].turnaround_time = 0;
        processes[i].waiting_time = 0;
    }
    printf("1. Priority Non-preemptive\n");
    printf("2. Priority Preemptive\n");
    printf("3. Exit\n");
    while(1)
    {
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("\nPriority Non-preemptive Scheduling:\n");
                priority_nonpreemptive(processes, n);
                break;

            case 2:
                printf("\nPriority Preemptive Scheduling:\n");
                priority_preemptive(processes, n);
                break;
            case 3: exit(0); break;
            default: printf("Invalid choice!\n");
        }
```

```
    }
return 0;
}
```

**OUTPUT:**

```
Enter the number of processes: 4
Process 1
Enter arrival time, burst time, priority: 0 4 3
Process 2
Enter arrival time, burst time, priority: 1 3 4
Process 3
Enter arrival time, burst time, priority: 2 3 6
Process 4
Enter arrival time, burst time, priority: 3 5 5
1. Priority Non-preemptive
2. Priority Preemptive
3. Exit
Enter your choice: 1

Priority Non-preemptive Scheduling:
Process Turnaround Time Waiting Time
1       4                0
2       6                3
4       9                4
3       13               10
Average Turnaround Time: 8.00
Average Waiting Time: 4.25
Enter your choice: 2

Priority Preemptive Scheduling:
Process Turnaround Time Waiting Time
1       4                0
2       6                3
4       9                4
3       13               10
Average Turnaround Time: 8.00
Average Waiting Time: 4.25
Enter your choice: 3
```

```
Enter the number of processes: 4
Process 1
Enter arrival time, burst time, priority: 0 5 4
Process 2
Enter arrival time, burst time, priority: 2 4 2
Process 3
Enter arrival time, burst time, priority: 2 2 6
Process 4
Enter arrival time, burst time, priority: 4 4 3
1. Priority Non-preemptive
2. Priority Preemptive
3. Exit
Enter your choice: 1

Priority Non-preemptive Scheduling:
Process Turnaround Time Waiting Time
1        5                 0
2        7                 3
4        9                 5
3        13                11
Average Turnaround Time: 8.50
Average Waiting Time: 4.75
Enter your choice: 2

Priority Preemptive Scheduling:
Process Turnaround Time Waiting Time
1        13                8
2        4                 0
4        6                 2
3        13                11
Average Turnaround Time: 9.00
Average Waiting Time: 5.25
Enter your choice: 3
```

**Round Robin (Experiment with different quantum sizes for RR algorithm):**

#include<stdio.h>
#include<limits.h>

```c
#include<stdbool.h>
struct P
{
    int AT,BT,ST[20],WT,FT,TAT,pos;
};
int quant;
int main()
{
    int n,i,j;
    printf("Enter the no. of processes :");
    scanf("%d",&n);
    struct P p[n];
    printf("Enter the quantum \n");
    scanf("%d",&quant);
    printf("Enter the process numbers \n");
    for(i=0;i<n;i++)
        scanf("%d",&(p[i].pos));
    printf("Enter the Arrival time of processes \n");
    for(i=0;i<n;i++)
        scanf("%d",&(p[i].AT));
    printf("Enter the Burst time of processes \n");
    for(i=0;i<n;i++)
        scanf("%d",&(p[i].BT));
    int c=n,s[n][20];
    float time=0, mini=INT_MAX, b[n], a[n];
    int index=-1;
    for(i=0;i<n;i++)
    {
        b[i]=p[i].BT;
        a[i]=p[i].AT;
        for(j=0;j<20;j++)
        {
            s[i][j]=-1;
```

```
        }
    }
    int tot_wt,tot_tat;
    tot_wt=0;
    tot_tat=0;
    bool flag=false;
    while(c!=0)
    {
        mini=INT_MAX;
        flag=false;
        for(i=0;i<n;i++)
        {
            float p=time+0.1;
            if(a[i]<=p && mini>a[i] && b[i]>0)
            {
                index=i;
                mini=a[i];
                flag=true;

            }
        }
        if(!flag)
        {
            time++;
            continue;
        }
        j=0;
        while(s[index][j]!=-1)
        {
            j++;
        }
        if(s[index][j]==-1)
        {
```

```c
        s[index][j]=time;
        p[index].ST[j]=time;
      }
    if(b[index]<=quant)
    {
       time+=b[index];
       b[index]=0;
    }
    else
    {
       time+=quant;
       b[index]-=quant;
    }
    if(b[index]>0)
    {
       a[index]=time+0.1;
    }
    if(b[index]==0)
    {
       c--;
       p[index].FT=time;
       p[index].WT=p[index].FT-p[index].AT-p[index].BT;
       tot_wt+=p[index].WT;
       p[index].TAT=p[index].BT+p[index].WT;
       tot_tat+=p[index].TAT;
    }

}
printf("Process number ");
printf("Arrival time ");
printf("Burst time ");
printf("\tStart time");
j=0;
```

```c
while(j!=10)
{
   j+=1;
   printf(" ");
}
printf("\t\tFinal time");
printf("\tWait Time ");
printf("\tTurnAround Time \n");
for(i=0;i<n;i++)
{
   printf("%d \t\t",p[i].pos);
   printf("%d \t\t",p[i].AT);
   printf("%d \t",p[i].BT);
   j=0;
   int v=0;
   while(s[i][j]!=-1)
   {
      printf("%d ",p[i].ST[j]);
      j++;
      v+=3;
   }
   while(v!=40)
   {
      printf(" ");
      v+=1;
   }
   printf("%d \t\t",p[i].FT);
   printf("%d \t\t",p[i].WT);
   printf("%d \n",p[i].TAT);
}
double avg_wt,avg_tat;
avg_wt=tot_wt/(float)n;
avg_tat=tot_tat/(float)n;
```

```
    printf("The average wait time is : %lf\n",avg_wt);
    printf("The average TurnAround time is : %lf\n",avg_tat);
    return 0;
}
```

**OUTPUT:**

```
Enter the no. of processes :5
Enter the quantum
2
Enter the process numbers
1 2 3 4 5
Enter the Arrival time of processes
0 1 2 3 4
Enter the Burst time of processes
5 3 1 2 3
Process number Arrival time Burst time  Start time                        Final time    Wait Time    TurnAround Time
1               0           5           0 5 12                            13            8            13
2               1           3           2 11                              12            8            11
3               2           1           4                                 5             2            3
4               3           2           7                                 9             4            6
5               4           3           9 13                              14            7            10
The average wait time is : 5.800000
The average TurnAround time is : 8.600000
```

```
Enter the no. of processes :5
Enter the quantum
1
Enter the process numbers
1 2 3 4 5
Enter the Arrival time of processes
0 3 4 4 5
Enter the Burst time of processes
7 2 1 3 3
Process number Arrival time Burst time  Start time                        Final time    Wait Time    TurnAround Time
1               0           7           0 1 2 4 9 12 15                   16            9            16
2               3           2           3 7                               8             3            5
3               4           1           5                                 6             1            2
4               4           3           6 10 13                           14            7            10
5               5           3           8 11 14                           15            7            10
The average wait time is : 5.400000
The average TurnAround time is : 8.600000
```

# EXPERIMENT - 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.

```c
#include <stdio.h>
int spat[10], upat[10], i, n1, n2, p1[10], p2[10];
int sppt[10], uppt[10], time = 0, op = 0, y, z, pt;
int sptat[10], uptat[10];
int spwt[10], upwt[10];
float spatat = 0, spawt = 0;
float upatat = 0, upawt = 0;
void process(int x, int isSystem) {
    if (isSystem) {
        op += sppt[x];
        sptat[x] = op - spat[x];
        sppt[x] = 0;
        spwt[x] = sptat[x] - p1[x];
        spatat += sptat[x];
        spawt += spwt[x];
    } else {
        op += uppt[x];
        uptat[x] = op - upat[x];
        uppt[x] = 0;
```

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

        upatat += uptat[x];

        upawt += upwt[x];

    }

}

int main() {

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

    scanf("%d", &n1);

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

    scanf("%d", &n2);

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

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

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

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

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

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

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

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

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

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

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

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

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

        time += sppt[i];

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

        time += uppt[i];
```

```c
for (i = 0; i < n1; i++)
    p1[i] = sppt[i];
for (i = 0; i < n2; i++)
    p2[i] = uppt[i];
printf("\n");
while (op < time) {
    y = -1;
    z = -1;
    for (i = 0; i < n1; i++) {
        if (op >= spat[i] && sppt[i] != 0) {
            y = i;
            break;
        }
    }
    for (i = 0; i < n2; i++) {
        if (op >= upat[i] && uppt[i] != 0) {
            z = i;
            break;
        }
    }
    if (y != -1) {
        printf("%d SP%d ", op, y + 1);
        process(y, 1);
    } else if (z != -1) {
        printf("%d UP%d ", op, z + 1);
        process(z, 0);
```

```c
        } else {
            op++;
        }
    }
    printf("%d ",op);
    printf("\n");
    printf("System Processes:\n");
    for (i = 0; i < n1; i++)
        printf("SP%d %d 0\n", i + 1, sptat[i]);
    printf("Average Turnaround Time (System Processes): %.2f\n", spatat / n1);
    printf("Average Waiting Time (System Processes): 0\n");
    printf("\n");
    printf("User Processes:\n");
    for (i = 0; i < n2; i++)
        printf("UP%d %d %d\n", i + 1, uptat[i], upwt[i]);
    printf("Average Turnaround Time (User Processes): %.2f\n", upatat / n2);
    printf("Average Waiting Time (User Processes): %.2f\n", upawt / n2);
    return 0;
}
```

**OUTPUT:**

```
Enter the number of System Processes: 2
Enter the number of User Processes: 2
Enter the arrival times for System Processes:
0 10
Enter the burst times for System Processes:
4 5
Enter the arrival times for User Processes:
0 0
Enter the burst times for User Processes:
3 8

0 SP1 4 UP1 7 UP2 15 SP2 20
System Processes:
SP1 4 0
SP2 10 0
Average Turnaround Time (System Processes): 7.00
Average Waiting Time (System Processes): 0

User Processes:
UP1 7 4
UP2 15 7
Average Turnaround Time (User Processes): 11.00
Average Waiting Time (User Processes): 5.50
```

```
Enter the number of System Processes: 3
Enter the number of User Processes: 1
Enter the arrival times for System Processes:
0 0 10
Enter the burst times for System Processes:
4 3 5
Enter the arrival times for User Processes:
0
Enter the burst times for User Processes:
8

0 SP1 4 SP2 7 UP1 15 SP3 20
System Processes:
SP1 4 0
SP2 7 0
SP3 10 0
Average Turnaround Time (System Processes): 7.00
Average Waiting Time (System Processes): 0

User Processes:
UP1 15 7
Average Turnaround Time (User Processes): 15.00
Average Waiting Time (User Processes): 7.00
```

# EXPERIMENT – 4

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

- Rate- Monotonic
- Earliest-deadline First

**Rate- Monotonic:**

```c
#include<stdio.h>
#include<math.h>
void main()
{
 int n;
 float e[20],p[20];
 int i;
 float ut,u,x,y;
 printf("Enter Number of Processes: ");
 scanf("%d",&n);
 for(i=0;i<n;i++)
 {
 printf("\nEnter Execution Time for P%d: ",(i+1));
 scanf("%f",&e[i]);
 printf("\nEnter Period for P%d: ",(i+1));
 scanf("%f",&p[i]);
 }
 for(i=0;i<n;i++)
 {
 x=e[i]/p[i];
```

```c
 ut+=x;
 }
 y=(float)n;
 y=y*((pow(2.0,1/y))-1);
 u=y;
 if(ut<u)
 {
printf("\nAs %f < %f ,",ut,u);
 printf("\nThe System is Schedulable");
 }
 else
 printf("\nNot Schedulable");

}
```

**OUTPUT:**

```
Enter Number of Processes: 3

Enter Execution Time for P1: 25

Enter Period for P1: 50

Enter Execution Time for P2: 35

Enter Period for P2: 100

Enter Execution Time for P3: 45

Enter Period for P3: 150

Not Schedulable
Enter Number of Processes: 3

Enter Execution Time for P1: 0.5

Enter Period for P1: 3

Enter Execution Time for P2: 1

Enter Period for P2: 4

Enter Execution Time for P3: 2

Enter Period for P3: 6

As 0.750000 < 0.779763 ,
The System is Schedulable
```

**Earliest-deadline First:**

#include <stdio.h>

#include <stdlib.h>

#define arrival 0

#define execution 1

#define deadline 2

#define period 3

#define abs_arrival 4

#define execution_copy  5

#define abs_deadline 6

typedef struct

```c
{ int T[7],instance,alive;
}task;
#define IDLE_TASK_ID 1023
#define ALL 1
#define CURRENT 0
void get_tasks(task *t1,int n);
int hyperperiod_calc(task *t1,int n);
float cpu_util(task *t1,int n);
int gcd(int a, int b)
{ if (b == 0)
return a;
else
return gcd(b, a % b);
}
int lcm(int *a, int n);
int sp_interrupt(task *t1,int tmr,int n);
int min(task *t1,int n,int p);
void update_abs_arrival(task *t1,int n,int k,int all);
void update_abs_deadline(task *t1,int n,int all);
void copy_execution_time(task *t1,int n,int all);
int timer = 0;
int main()
{ task *t;
int n, hyper_period, active_task_id;
float cpu_utilization;
printf("Enter number of tasks\n");
```

```c
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++;
```

```c
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;
```

```c
t1->instance = 0;

t1->alive = 0;

t1++;

i++;

}

}

int hyperperiod_calc(task *t1, int n)

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

while (i < n)

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

t1++;

i++;

}

ht = lcm(a, n);


return ht;

}

int gcd(int a, int b)

int lcm(int *a, int n)

{

int res = 1, i;

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

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

}

return res;

}
```

```c
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: 20
Period: 20
Enter Task 2 parameters
Arrival time: 0
Execution time: 2
Deadline time: 5
Period: 5
Enter Task 3 parameters
Arrival time: 0
Execution time: 2
Deadline time: 20
Period: 20
CPU Utilization 0.650000
Tasks can be scheduled
0   Task 2
1   Task 2
2   Task 1
3   Task 1
4   Task 1
5   Task 2
6   Task 2
7   Task 3
8   Task 3
9   Idle
10  Task 2
11  Task 2
12  Idle
13  Idle
14  Idle
15  Task 2
16  Task 2
17  Idle
18  Idle
19  Idle
20  Task 2
```

```
Enter number of tasks
2
Enter Task 1 parameters
Arrival time: 0
Execution time: 25
Deadline time: 50
Period: 50
Enter Task 2 parameters
Arrival time: 0
Execution time: 35
Deadline time: 80
Period: 80
CPU Utilization 0.937500
Tasks can be scheduled
0   Task 1
1   Task 1
2   Task 1
3   Task 1
4   Task 1
5   Task 1
6   Task 1
7   Task 1
8   Task 1
9   Task 1
10  Task 1
11  Task 1
12  Task 1
13  Task 1
14  Task 1
15  Task 1
16  Task 1
17  Task 1
18  Task 1
19  Task 1
20  Task 1
21  Task 1
22  Task 1
23  Task 1
24  Task 1
25  Task 2
26  Task 2
27  Task 2
28  Task 2
29  Task 2
```

# EXPERIMENT – 5

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

```c
#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);
```

```c
      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:**

```
Produced: 1
Produced: 2
Produced: 3
Produced: 4
Produced: 5
Consumed: 1
Consumed: 2
Consumed: 3
Consumed: 4
Consumed: 5
Produced: 6
Produced: 7
Produced: 8
Produced: 9
Produced: 10
Consumed: 6
Consumed: 7
Consumed: 8
Consumed: 9
Consumed: 10
Produced: 11
Produced: 12
Produced: 13
Produced: 14
Produced: 15
Consumed: 11
Consumed: 12
Consumed: 13
Consumed: 14
Consumed: 15
Produced: 16
Produced: 17
Produced: 18
Produced: 19
Produced: 20
Consumed: 16
Consumed: 17
Consumed: 18
Consumed: 19
Consumed: 20
```

```
Produced: 1
Produced: 2
Produced: 3
Produced: 4
Produced: 5
Consumed: 1
Produced: 6
Consumed: 2
Produced: 7
Consumed: 3
Produced: 8
Consumed: 4
Produced: 9
Consumed: 5
Produced: 10
Consumed: 6
Produced: 11
Consumed: 7
Produced: 12
Consumed: 8
Produced: 13
Consumed: 9
Produced: 14
Consumed: 10
Produced: 15
Consumed: 11
Consumed: 12
Consumed: 13
Consumed: 14
Consumed: 15
Produced: 16
Produced: 17
Produced: 18
Consumed: 16
Consumed: 17
Consumed: 18
Produced: 19
Produced: 20
Consumed: 19
Consumed: 20
```

# EXPERIMENT – 6

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

```c
#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);
```

```c
            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)

{
```

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

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

# EXPERIMENT - 7

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

```c
#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("********** Baner's Algo ***********\n");
 input();
 show();
 cal();
 getch();
 return 0;
}
void input()
{
 int i,j;
```

```c
printf("Enter the no of Processes\t");
scanf("%d",&n);
printf("Enter the no of resources instances\t");
scanf("%d",&r);
printf("Enter the Max Matrix\n");
for(i=0;i<n;i++)
{
 for(j=0;j<r;j++)
 {
  scanf("%d",&max[i][j]);
 }
}
printf("Enter the Allocation Matrix\n");
for(i=0;i<n;i++)
{
 for(j=0;j<r;j++)
 {
  scanf("%d",&alloc[i][j]);
 }
}
printf("Enter the available Resources\n");
for(j=0;j<r;j++)
{
 scanf("%d",&avail[j]);
}
}
```

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

```c
int finish[100],temp,need[100][100],flag=1,k,c1=0;
int safe[100];
int i,j;
for(i=0;i<n;i++)
{
 finish[i]=0;
}
//find need matrix
for(i=0;i<n;i++)
{
 for(j=0;j<r;j++)
 {
  need[i][j]=max[i][j]-alloc[i][j];
 }
}
printf("\n");
while(flag)
{
 flag=0;
  for(i=0;i<n;i++)
 {
  int c=0;
  for(j=0;j<r;j++)
  {
   if((finish[i]==0)&&(need[i][j]<=avail[j]))
   {
```

```c
    c++;
   if(c==r)
  {
   for(k=0;k<r;k++)
  {
   avail[k]+=alloc[i][j];
   finish[i]=1;
   flag=1;
     }
   printf("P%d->",i);
   if(finish[i]==1)
  {
   i=n;
  }
     }
 }
 }
 }
}
for(i=0;i<n;i++)
{
if(finish[i]==1)
{
 c1++;
}
else
```

```
    {

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

    }

   }

   if(c1==n)

   {

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

   }

   else

   {

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

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

   }

   }
```

**OUTPUT:**

```
********** Baner's Algo ************
Enter the no of Processes        5
Enter the no of resources instances     3
Enter the Max Matrix
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter the Allocation Matrix
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the available Resources
3 2 2
Process  Allocation      Max      Available
P1          0 1 0        7 5 3    3 2 2
P2          2 0 0        3 2 2
P3          3 0 2        9 0 2
P4          2 1 1        2 2 2
P5          0 0 2        4 3 3
P1->P3->P4->P2->P0->
 The system is in safe state
```

```
********** Baner's Algo ************
Enter the no of Processes        3
Enter the no of resources instances     3
Enter the Max Matrix
3 6 8
4 3 3
3 4 4
Enter the Allocation Matrix
3 3 3
2 0 3
1 2 4
Enter the available Resources
1 2 0
Process   Allocation      Max      Available
P1          3 3 3       3 6 8   1 2 0
P2          2 0 3       4 3 3
P3          1 2 4       3 4 4
P0->P1->P2->
 Process are in dead lock
 System is in unsafe state
```

# EXPERIMENT - 8

Write a C program to simulate deadlock detection.

```c
#include<stdio.h>
static int mark[20];
int i, j, np, nr;
int main()
{
int alloc[10][10],request[10][10],avail[10],r[10],w[10];
printf ("\nEnter the no of the process: ");
scanf("%d",&np);
printf ("\nEnter the no of resources: ");
scanf("%d",&nr);
for(i=0;i<nr; i++)
{
printf("\nTotal Amount of the Resource R % d: ",i+1);
scanf("%d", &r[i]);
}
printf("\nEnter the request matrix:");
for(i=0;i<np;i++)
for(j=0;j<nr;j++)
scanf("%d",&request[i][j]);
printf("\nEnter the allocation matrix:");
for(i=0;i<np;i++)
for(j=0;j<nr;j++)
scanf("%d",&alloc[i][j]);
```

```c
/*Available Resource calculation*/
for(j=0;j<nr;j++)
{
avail[j]=r[j];
for(i=0;i<np;i++)
{
avail[j]-=alloc[i][j];
}
}
//marking processes with zero allocation
for(i=0;i<np;i++)
{
int count=0;
 for(j=0;j<nr;j++)
  {
    if(alloc[i][j]==0)
     count++;
    else
     break;
  }
 if(count==nr)
 mark[i]=1;
}
// initialize W with avail
for(j=0;j<nr; j++)
   w[j]=avail[j];
```

```
//mark processes with request less than or equal to W
for(i=0;i<np; i++)
{
int canbeprocessed= 0;
 if(mark[i]!=1)
{
  for(j=0;j<nr;j++)
   {
    if(request[i][j]<=w[j])
     canbeprocessed=1;
    else
      {
      canbeprocessed=0;
      break;
      }
   }
if(canbeprocessed)
{
mark[i]=1;
for(j=0;j<nr;j++)
w[j]+=alloc[i][j];
}
}
}
//checking for unmarked processes
int deadlock=0;
```

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

if(mark[i]!=1)

deadlock=1;

if(deadlock)

printf("\n Deadlock detected");

else

printf("\n No Deadlock possible");

}
```

**OUTPUT:**

```
Enter the no of the process: 4

Enter the no of resources: 5

Total Amount of the Resource R  1: 2

Total Amount of the Resource R  2: 1

Total Amount of the Resource R  3: 1

Total Amount of the Resource R  4: 2

Total Amount of the Resource R  5: 1

Enter the request matrix:0 1 0 0 1
0 0 1 0 1
0 0 0 0 1
1 0 1 0 1

Enter the allocation matrix:1 0 1 1 0
1 1 0 0 0
0 0 0 1 0
0 0 0 0 0

 Deadlock detected
```

```
Enter the no of the process: 3

Enter the no of resources: 3

Total Amount of the Resource R  1: 1

Total Amount of the Resource R  2: 2

Total Amount of the Resource R  3: 4

Enter the request matrix:1 0 2
2 0 9
1 1 0

Enter the allocation matrix:0 0 1
1 3 6
9 5 1

 Deadlock detected
```

# EXPERIMENT – 9

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

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

**Worst-fit:**

```c
#include <stdio.h>
#include <string.h>
// Function to allocate memory to blocks as per worst fit
// algorithm
void worstFit(int blockSize[], int m, int processSize[], int n)
{
   // Stores block id of the block allocated to a
   // process
   int allocation[n];
   // Initially no block is assigned to any process
   memset(allocation, -1, sizeof(allocation));
   // pick each process and find suitable blocks
   // according to its size and assign to it
   for (int i = 0; i < n; i++)
   {
      // Find the best fit block for the current process
      int wstIdx = -1;
      for (int j = 0; j < m; j++)
```

```c
        {
            if (blockSize[j] >= processSize[i])
            {
                if (wstIdx == -1)
                    wstIdx = j;
                else if (blockSize[wstIdx] < blockSize[j])
                    wstIdx = j;
            }
        }
        // If we could find a block for the current process
        if (wstIdx != -1)
        {
            // allocate block j to p[i] process
            allocation[i] = wstIdx;
            // Reduce available memory in this block.
            blockSize[wstIdx] -= processSize[i];
        }
    }
    printf("\nProcess No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < n; i++)
    {
        printf(" %d\t\t%d\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1)
            printf("%d", allocation[i] + 1);
        else
            printf("Not Allocated");
```

```c
        printf("\n");
    }
}
// Driver code
int main()
{
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);
    worstFit(blockSize, m, processSize, n);
    return 0;
}
```

**OUTPUT:**

```
Process No.      Process Size      Block no.
1                212               5
2                417               2
3                112               5
4                426               Not Allocated
```

```
Process No.      Process Size      Block no.
1                1                 3
2                4                 3
```

**Best-fit:**

```c
#include <stdio.h>
void implimentBestFit(int blockSize[], int blocks, int processSize[], int processes)
{
    // This will store the block id of the allocated block to a process
```

```c
int allocation[processes];
// initially assigning -1 to all allocation indexes
// means nothing is allocated currently
for(int i = 0; i < processes; i++){
    allocation[i] = -1;
}
 // pick each process and find suitable blocks
// according to its size ad assign to it
for (int i=0; i < processes; i++)
{
    int indexPlaced = -1;
    for (int j=0; j < blocks; j++)
    {
        if (blockSize[j] >= processSize[i])
        {
            // place it at the first block fit to accomodate process
            if (indexPlaced == -1)
                indexPlaced = j;
            // if any future block is better that is
            // any future block with smaller size encountered
            // that can accomodate the given process
            else if (blockSize[j] < blockSize[indexPlaced])
                indexPlaced = j;
        }
    }
```

```c
    // If we were successfully able to find block for the process
    if (indexPlaced != -1)
    {
      // allocate this block j to process p[i]
      allocation[i] = indexPlaced;


      // Reduce available memory for the block
      blockSize[indexPlaced] -= processSize[i];
    }
  }
  printf("\nProcess No.\tProcess Size\tBlock no.\n");
  for (int i = 0; i < processes; i++)
  {
    printf("%d \t\t\t %d \t\t\t", i+1, processSize[i]);
    if (allocation[i] != -1)
      printf("%d\n",allocation[i] + 1);
    else
      printf("Not Allocated\n");
  }
}
// Driver code
int main()
{
  int blockSize[] = {50, 20, 100, 90};
  int processSize[] = {10, 30, 60, 80};
  int blocks = sizeof(blockSize)/sizeof(blockSize[0]);
```

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

    implimentBestFit(blockSize, blocks, processSize, processes);

    return 0 ;

}
```

**OUTPUT:**

```
Process No.       Process Size     Block no.
1                        10                         2
2                        30                         1
3                        60                         4
4                        80                         3
```

```
Process No.       Process Size     Block no.
1                        1                          2
2                        4                          1
```

**First-fit:**

```
#include <stdio.h>

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

{

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

    int allocate[processes];

    // initially assigning -1 to all allocation indexes

    // means nothing is allocated currently

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

    {

            allocate[i] = -1;

    }
```

```c
    // take each process one by one and find
    // first block that can accomodate it
    for (int i = 0; i < processes; i++)
    {
      for (int j = 0; j < blocks; j++) {
        if (blockSize[j] >= processSize[i])
        {
          // allocate block j to p[i] process
          allocate[i] = j;
          // Reduce size of block j as it has accomodated p[i]
          blockSize[j] -= processSize[i];
          break;
        }
      }
    }
    printf("\nProcess No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < processes; i++)
    {
      printf("%d \t\t\t %d \t\t\t", i+1, processSize[i]);
      if (allocate[i] != -1)
        printf("%d\n",allocate[i] + 1);
      else
        printf("Not Allocated\n");
    }
}
```

```
void main()

{

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

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

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

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

    implimentFirstFit(blockSize, m, processSize, n);

}
```

```
Process No.      Process Size     Block no.
1                      10                    1
2                       6                    1
3                       9                    1
```

```
Process No.      Process Size     Block no.
1                       1                    1
2                       4                    1
```

# EXPERIMENT – 10

Write a C program to simulate page replacement algorithms

- FIFO
- LRU
- Optimal

**FIFO:**

```c
#include<stdio.h>
int main()
{
   int i,j,n,a[50],frame[10],no,k,avail,count=0;
   printf("ENTER THE NUMBER OF PAGES: ");
   scanf("%d",&n);
   printf("\nENTER THE PAGE REFERENCE : ");
   for(i=1;i<=n;i++)
      scanf("%d",&a[i]);
   printf("\nENTER THE NUMBER OF FRAMES : ");
   scanf("%d",&no);
   for(i=0;i<no;i++)
      frame[i]= -1;
   j=0;
   printf("\tref string\t page frames\n");
   for(i=1;i<=n;i++)
   {
      printf("%d\t\t",a[i]);
      avail=0;
```

```c
    for(k=0;k<no;k++)
        if(frame[k]==a[i])
            avail=1;
    if (avail==0)
    {
        frame[j]=a[i];
        j=(j+1)%no;
        count++;
        for(k=0;k<no;k++)
            printf("%d\t",frame[k]);
    }
    printf("\n");
    }
    printf("Page Fault Is %d",count);
    return 0;
}
```

**OUTPUT:**

```
ENTER THE NUMBER OF PAGES: 7

ENTER THE PAGE REFERENCE : 1 3 0 3 5 6 3

ENTER THE NUMBER OF FRAMES : 3
        ref string         page frames
1               1          -1        -1
3               1          3         -1
0               1          3         0
3
5               5          3         0
6               5          6         0
3               5          6         3
Page Fault Is 6
```

```
ENTER THE NUMBER OF PAGES: 14

ENTER THE PAGE REFERENCE : 7 0 1 2 0 3 0 4 2 3 0 3 2 3

ENTER THE NUMBER OF FRAMES : 4
        ref string          page frames
7               7       -1          -1          -1
0               7        0          -1          -1
1               7        0           1          -1
2               7        0           1           2
0
3               3        0           1           2
0
4               3        4           1           2
2
3
0               3        4           0           2
3
2
3
Page Fault Is 7
```

**LRU:**

```c
#include<stdio.h>

void main()

{

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

   printf("Enter no of pages: ");

   scanf("%d",&n);

   printf("Enter the reference string: ");

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

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

   printf("Enter no of frames: ");

   scanf("%d",&f);

   q[k]=p[k];

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

   c++;
```

```c
k++;
for(i=1;i<n;i++)
{
    c1=0;
    for(j=0;j<f;j++)
    {
        if(p[i]!=q[j])
            c1++;
    }
    if(c1==f)
    {
        c++;
        if(k<f)
        {
            q[k]=p[i];
            k++;
            for(j=0;j<k;j++)
                printf("\t%d",q[j]);
            printf("\n");
        }
        else
        {
            for(r=0;r<f;r++)
            {
                c2[r]=0;
                for(j=i-1;j<n;j--)
```

```c
        {
            if(q[r]!=p[j])
                c2[r]++;
            else
                break;
        }
    }
    for(r=0;r<f;r++)
        b[r]=c2[r];
    for(r=0;r<f;r++)
    {
        for(j=r;j<f;j++)
        {
            if(b[r]<b[j])
            {
                t=b[r];
                b[r]=b[j];
                b[j]=t;
            }
        }
    }
    for(r=0;r<f;r++)
    {
        if(c2[r]==b[0])
            q[r]=p[i];
        printf("\t%d",q[r]);
```

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

**OUTPUT:**

```
Enter no of pages: 10
Enter the reference string: 7 5 9 4 3 7 9 6 2 1
Enter no of frames: 3

        7
        7       5
        7       5       9
        4       5       9
        4       3       9
        4       3       7
        9       3       7
        9       6       7
        9       6       2
        1       6       2

The no of page faults is 10
```

```
Enter no of pages: 14
Enter the reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 3
Enter no of frames: 4

        7
        7       1
        7       1       2
        7       1       2       3
        0       1       2       3
        0       4       2       3

The no of page faults is 6
```

**Optimal:**

```c
#include<stdio.h>
int main()
{
    int n,pg[30],fr[10];
    int count[10],i,j,k,fault,f,flag,temp,current,c,dist,max,m,cnt,p,x;
    fault=0;
    dist=0;
    k=0;
    printf("Enter the total no pages: ");
    scanf("%d",&n);
    printf("\nEnter the sequence: ");
    for(i=0;i<n;i++)
        scanf("%d",&pg[i]);
    printf("\nEnter frame size: ");
    scanf("%d",&f);
    for(i=0;i<f;i++)
    {
        count[i]=0;
        fr[i]=-1;
    }
    for(i=0;i<n;i++)
    {
        flag=0;
        temp=pg[i];
        for(j=0;j<f;j++)
```

```c
{
   if(temp==fr[j])
   {
      flag=1;
      break;
   }
}
if((flag==0)&&(k<f))
{
   fault++;
   fr[k]=temp;
   k++;
}
else if((flag==0)&&(k==f))
{
   fault++;
   for(cnt=0;cnt<f;cnt++)
   {
      current=fr[cnt];
      for(c=i;c<n;c++)
      {
         if(current!=pg[c])
            count[cnt]++;
         else
            break;
      }
```

```c
        }
        max=0;
        for(m=0;m<f;m++)
        {
           if(count[m]>max)
           {
              max=count[m];
              p=m;
           }
        }
        fr[p]=temp;
     }
     printf("\npage  %d  frame\t",pg[i]);
     for(x=0;x<f;x++)
     {
        printf("%d\t",fr[x]);
     }
   }
   printf("\nTotal number of faults= %d",fault);
   return 0;
}
```

**OUTPUT:**

```
Enter the total no pages: 10

Enter the sequence: 0 1 2 3 0 1 2 3 0 1

Enter frame size: 3

page  0   frame   0        -1        -1
page  1   frame   0         1        -1
page  2   frame   0         1         2
page  3   frame   0         1         3
page  0   frame   0         1         3
page  1   frame   0         1         3
page  2   frame   0         2         3
page  3   frame   0         2         3
page  0   frame   0         2         3
page  1   frame   0         1         3
Total number of faults= 6
```

```
Enter the total no pages: 14

Enter the sequence: 7 0 1 2 0 3 0 4 3 2 0 3 2 3

Enter frame size: 4

page  7   frame   7        -1        -1        -1
page  0   frame   7         0        -1        -1
page  1   frame   7         0         1        -1
page  2   frame   7         0         1         2
page  0   frame   7         0         1         2
page  3   frame   3         0         1         2
page  0   frame   3         0         1         2
page  4   frame   3         0         4         2
page  3   frame   3         0         4         2
page  2   frame   3         0         4         2
page  0   frame   3         0         4         2
page  3   frame   3         0         4         2
page  2   frame   3         0         4         2
page  3   frame   3         0         4         2
Total number of faults= 6
```

# EXPERIMENT - 11

Write a C program to simulate disk scheduling algorithms

- FCFS
- SCAN
- C-SCAN

**FCFS:**

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int RQ[100],i,n,TotalHeadMoment=0,initial;
   printf("Enter the number of Requests: ");
   scanf("%d",&n);
   printf("\nEnter the Requests sequence: ");
   for(i=0;i<n;i++)
    scanf("%d",&RQ[i]);
   printf("\nEnter initial head position: ");
   scanf("%d",&initial);
   // logic for FCFS disk scheduling
   for(i=0;i<n;i++)
   {   TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
     initial=RQ[i];
   }
   printf("Total head moment is %d",TotalHeadMoment);
   return 0;
}
```

**OUTPUT:**

```
Enter the number of Requests: 8

Enter the Requests sequence: 95 180 34 119 11 123 62 64

Enter initial head position: 50
Total head moment is 644
```

```
Enter the number of Requests: 7

Enter the Requests sequence: 82
170
43
140
24
16
190

Enter initial head position: 50
Total head moment is 642
```

**SCAN:**

```c
#include <stdio.h>

int request[50];

int SIZE;

int pre;

int head;

int uptrack;

int downtrack;

struct max{

    int up;

    int down;

} kate[50];

int dist(int a, int b){

    if (a > b)
```

```
        return a - b;
     return b - a;
  }
  void sort(int n){
     int i, j;
     for (i = 0; i < n - 1; i++){
        for (j = 0; j < n - i - 1; j++){
           if (request[j] > request[j + 1]){
              int temp = request[j];
              request[j] = request[j + 1];
              request[j + 1] = temp;
           }
        }
     }
     j = 0;
     i = 0;
     while (request[i] != head){
        kate[j].down = request[i];
        j++;
        i++;
     }
     downtrack = j;
     i++;
     j = 0;
     while (i < n){
        kate[j].up = request[i];
```

```c
        j++;
        i++;
    }
    uptrack = j;
}
void scan(int n){
    int i;
    int seekcount = 0;
    printf("SEEK SEQUENCE = ");
    sort(n);
    if (pre < head){
        for (i = 0; i < uptrack; i++){
            printf("%d ", head);
            seekcount = seekcount + dist(head, kate[i].up);
            head = kate[i].up;
        }
        for (i = downtrack - 1; i > 0; i--){
            printf("%d ", head);
            seekcount = seekcount + dist(head, kate[i].down);
            head = kate[i].down;
        }
    }
    else{
        for (i = downtrack - 1; i >= 0; i--){
            printf("%d ", head);
            seekcount = seekcount + dist(head, kate[i].down);
```

```c
            head = kate[i].down;
        }
        for (i = 0; i < uptrack - 1; i++){
            printf("%d ", head);
            seekcount = seekcount + dist(head, kate[i].up);
            head = kate[i].up;
        }
    }
    printf(" %d\nTOTAL DISTANCE :%d", head, seekcount);
}
int main(){
    int n, i;
    printf("ENTER THE DISK SIZE : ");
    scanf("%d", &SIZE);
    printf("\nENTER THE NO OF REQUEST SEQUENCE : ");
    scanf("%d", &n);
    printf("\nENTER THE REQUEST SEQUENCE : ");
    for (i = 0; i < n; i++)
        scanf("%d", &request[i]);
    printf("\nENTER THE CURRENT HEAD : ");
    scanf("%d", &head);
    request[n] = head;
    request[n + 1] = SIZE - 1;
    request[n + 2] = 0;
    printf("\nENTER THE PRE REQUEST : ");
    scanf("%d", &pre);
```

```
    scan(n + 3);

}
```

**OUTPUT:**

```
ENTER THE DISK SIZE : 4

ENTER THE NO OF REQUEST SEQUENCE : 2

ENTER THE REQUEST SEQUENCE : 1 2

ENTER THE CURRENT HEAD : 1

ENTER THE PRE REQUEST : 2
SEEK SEQUENCE = 1 0 1  2
TOTAL DISTANCE :3
```

```
ENTER THE DISK SIZE : 200

ENTER THE NO OF REQUEST SEQUENCE : 7

ENTER THE REQUEST SEQUENCE : 82
170
43
140
24
16
190

ENTER THE CURRENT HEAD : 50

ENTER THE PRE REQUEST : 1
SEEK SEQUENCE = 50 82 140 170 190 199 43 24   16
TOTAL DISTANCE :332
```

**C-SCAN:**

#include <stdio.h>

#include <stdlib.h>

int main(){

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

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

   scanf("%d", &n);

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

```c
for (i = 0; i < n; i++)
    scanf("%d", &RQ[i]);
printf("\nEnter initial head position: ");
scanf("%d", &initial);
printf("\nEnter total disk size: ");
scanf("%d", &size);
printf("\nEnter the head movement direction for high 1 and for low 0: ");
scanf("%d", &move);
for (i = 0; i < n; i++){
    for (j = 0; j < n - i - 1; j++){
        if (RQ[j] > RQ[j + 1]){
            int temp;
            temp = RQ[j];
            RQ[j] = RQ[j + 1];
            RQ[j + 1] = temp;
        }
    }
}
int index;
for (i = 0; i < n; i++){
    if (initial < RQ[i]){
        index = i;
        break;
    }
}
if (move == 1){
```

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

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

        initial = RQ[i];

    }

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

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

    initial = 0;

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

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

        initial = RQ[i];

    }

}

else{

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

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

        initial = RQ[i];

    }

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

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

    initial = size - 1;

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

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

        initial = RQ[i];

    }

}

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

```
    return 0;

}
```

**OUTPUT:**

```
Enter the number of Requests: 7

Enter the Requests sequence: 82
170
43
140
24
16
190

Enter initial head position: 50

Enter total disk size: 200

Enter the head movement direction for high 1 and for low 0: 0
Total head movement is 366
```

```
Enter the number of Requests: 3

Enter the Requests sequence: 2 1 0

Enter initial head position: 1

Enter total disk size: 3

Enter the head movement direction for high 1 and for low 0: 1
Total head movement is 4
```