VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

(23CS4PCOPS)

Submitted by

NEELVANI VARSHA VITTAL (1BM23CS412)

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

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Apr-2024 to Aug-2024

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

CERTIFICATE

This is to certify that the Lab work entitled "OPERATING SYSTEMS – 23CS4PCOPS" carried out by **NEELVANI VARSHA VITTAL (1BM23CS412),** who is bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2024. The Lab report has been approved as it satisfies the academic requirements in respect of a **OPERATING SYSTEMS - (23CS4PCOPS)** work prescribed for the said degree.

Prof. Swathi Sridharan	**Dr. Jyothi S Nayak**
Assistant Professor	Professor and Head
Department of CSE	Department of CSE
BMSCE, Bengaluru	BMSCE, Bengaluru

INDEX PAGE

Course Outcome

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

Program - 1

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

→FCFS

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_PROCESS 30
int p[MAX_PROCESS], arrTime[MAX_PROCESS], burstTime[MAX_PROCESS],
compTime[MAX_PROCESS], TAT[MAX_PROCESS], waitTime[MAX_PROCESS];
void sortProcess(int arrTime[], int burstTime[], int n) {
   int temp;
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n - i - 1; j++) {
         if (arrTime[j] > arrTime[j + 1]) {
            temp = arrTime[j];
            arrTime[j] = arrTime[j + 1];
            arrTime[j + 1] = temp;
            temp = burstTime[j];
            burstTime[j] = burstTime[j + 1];
            burstTime[j + 1] = temp;
         }
      }
   }
}
int findTurnAroundTime(int ct, int at) {
   return ct - at;
}
int waitingTime(int tat, int bt) {
   return tat - bt;
}
int main() {
   int n;
   printf("Enter total number of processes: ");
   scanf("%d", &n);
   int total_TAT = 0;
   int total_WT = 0;
   for (int i = 0; i < n; i++) {
      printf("Process [%d]\n", i + 1);
      printf("Arrival time: ");
      scanf("%d", &arrTime[i]);
      printf("Burst time: ");
```

```c
        scanf("%d", &burstTime[i]);
        p[i] = i + 1;
    }
    sortProcess(arrTime, burstTime, n);
    for (int i = 0; i < n; i++) {
        if (i == 0 || arrTime[i] > compTime[i - 1]) {
            compTime[i] = arrTime[i] + burstTime[i];
        } else {
            compTime[i] = compTime[i - 1] + burstTime[i];
        }
        TAT[i] = findTurnAroundTime(compTime[i], arrTime[i]);
        waitTime[i] = waitingTime(TAT[i], burstTime[i]);
        total_TAT += TAT[i];
        total_WT += waitTime[i];
    }
    float avg_TAT = (float)total_TAT / n;
    float avg_WT = (float)total_WT / n;
    printf("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i], arrTime[i], burstTime[i], compTime[i], TAT[i], waitTime[i]);
    }
    printf("\nAverage Turnaround Time: %.2f", avg_TAT);
    printf("\nAverage Waiting Time: %.2f\n", avg_WT);
    return 0;
}
```

```
Enter total number of processes: 4
Process [1]
Arrival time: 0
Burst time: 2
Process [2]
Arrival time: 1
Burst time: 2
Process [3]
Arrival time: 5
Burst time: 3
Process [4]
Arrival time: 6
Burst time: 4

Process Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
1       0               2               2               2               0
2       1               2               4               3               1
3       5               3               8               3               0
4       6               4               12              6               2

Average Turnaround Time: 3.50
Average Waiting Time: 0.75

Process returned 0 (0x0)   execution time : 81.227 s
Press any key to continue.
```

→ SJF (pre-emptive)

```
#include <stdio.h>
#include <limits.h>
int main() {
   int n;
   printf("Enter the number of processes: ");
   scanf("%d", &n);
   int pid[n], arrival[n], burst[n], remaining[n], completion[n], waiting[n], turnaround[n];
   float avg_waiting_time = 0, avg_turnaround_time = 0;
   for (int i = 0; i < n; i++) {
      pid[i] = i + 1;
      printf("Enter arrival time and burst time for process %d: ", i + 1);
      scanf("%d %d", &arrival[i], &burst[i]);
      remaining[i] = burst[i];
   }
   int completed = 0, current_time = 0, shortest = 0;
   int min_remaining_time = INT_MAX;
   int finish_time;
   int check = 0;
   while (completed != n) {
      for (int j = 0; j < n; j++) {
         if ((arrival[j] <= current_time) &&
            (remaining[j] < min_remaining_time) && remaining[j] > 0) {
            min_remaining_time = remaining[j];
            shortest = j;
            check = 1;
         }
      }
      if (check == 0) {
         current_time++;
         continue;
      }
      remaining[shortest]--;
      min_remaining_time = remaining[shortest];
      if (min_remaining_time == 0) {
         min_remaining_time = INT_MAX;
      }
      if (remaining[shortest] == 0) {
         completed++;
         check = 0;
```

```
            finish_time = current_time + 1;
            completion[shortest] = finish_time;
            turnaround[shortest] = finish_time - arrival[shortest];
            waiting[shortest] = turnaround[shortest] - burst[shortest];
            avg_waiting_time += waiting[shortest];
            avg_turnaround_time += turnaround[shortest];
        }
        current_time++;
    }
    avg_waiting_time /= n;
    avg_turnaround_time /= n;
    printf("\nPID\t\tAT\t\tBT\t\tCT\t\tTAT\t\tWT\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", pid[i], arrival[i], burst[i], completion[i],
turnaround[i], waiting[i]);
    }
    printf("\nAverage Waiting Time: %.2f", avg_waiting_time);
    printf("\nAverage Turnaround Time: %.2f", avg_turnaround_time);
    return 0;
}
```

```
Enter the number of processes: 5
Enter arrival time and burst time for process 1: 2 1
Enter arrival time and burst time for process 2: 1 5
Enter arrival time and burst time for process 3: 4 1
Enter arrival time and burst time for process 4: 0 6
Enter arrival time and burst time for process 5: 2 3

PID             AT              BT              CT              TAT             WT
1               2               1               3               1               0
2               1               5               16              15              10
3               4               1               5               1               0
4               0               6               11              11              5
5               2               3               7               5               2

Average Waiting Time: 3.40
Average Turnaround Time: 6.60
Process returned 0 (0x0)   execution time : 25.947 s
Press any key to continue.
```

→ SJF (Non - preemptive)

```
#include <stdio.h>
struct Process {
    int id;
    int at;
```

```
    int bt;
    int ct;
    int tt;
    int wt;
};
void sort(struct Process p[], int n);
void sjf(struct Process p[], int n);
int main() {
    int n;
    int total_tat = 0;
    int total_wt = 0;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process p[n];
    printf("Enter the arrival time and burst time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        p[i].id = i + 1;
        printf("Arrival Time: ");
        scanf("%d", &p[i].at);
        printf("Burst Time: ");
        scanf("%d", &p[i].bt);
    }
    sort(p, n);
    sjf(p, n);
    printf("\nProcess Schedule:\n");
    printf("Process ID\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting
Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i].id, p[i].at, p[i].bt, p[i].ct, p[i].tt, p[i].wt);
        total_tat += p[i].tt;
        total_wt += p[i].wt;
    }
    printf("\nAvg TAT: %.2f", (float)total_tat / n);
    printf("\nAvg WT: %.2f", (float)total_wt / n);
    return 0;
}
void sort(struct Process p[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (p[j].at > p[j + 1].at || (p[j].at == p[j + 1].at && p[j].bt > p[j + 1].bt)) {
```

```
            struct Process temp = p[j];
            p[j] = p[j + 1];
            p[j + 1] = temp;  }
        }
    }
}
void sjf(struct Process p[], int n) {
    int current_time = 0;
    for (int i = 0; i < n; i++) {
        int sj_index = i;
        for (int j = i + 1; j < n && p[j].at <= current_time; j++) {
            if (p[j].bt < p[sj_index].bt) {
                sj_index = j;
            }
        }
        p[sj_index].ct = current_time + p[sj_index].bt;
        p[sj_index].tt = p[sj_index].ct - p[sj_index].at;
        p[sj_index].wt = p[sj_index].tt - p[sj_index].bt;
        current_time = p[sj_index].ct;
        struct Process temp = p[i];
        p[i] = p[sj_index];
        p[sj_index] = temp;
    }
}
```

```
Enter the arrival time and burst time for each process:
Process 1:
Arrival Time: 2
Burst Time: 1
Process 2:
Arrival Time: 1
Burst Time: 5
Process 3:
Arrival Time: 4
Burst Time: 1
Process 4:
Arrival Time: 0
Burst Time: 6
Process 5:
Arrival Time: 2
Burst Time: 3

Process Schedule:
Process ID      Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
4               0               6               6               6               0
1               2               1               7               5               4
3               4               1               8               4               3
5               2               3               11              9               6
2               1               5               16              15              10

Avg TAT: 7.80
Avg WT: 4.60
Process returned 0 (0x0)   execution time : 37.106 s
Press any key to continue.
```

Program - 2

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

```
#include <stdio.h>
typedef struct {
    int id;
    int at;
    int bt;
    int priority;
    int rt;
} Proc;
void swap(Proc *a, Proc *b) {
    Proc temp = *a;
    *a = *b;
    *b = temp;
}
void sort_by_priority(Proc p[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (p[j].priority > p[j + 1].priority) {
                swap(&p[j], &p[j + 1]);
            }
        }
    }
}
void priority_preemptive(Proc p[], int n) {
    int wt[n], tat[n];
    int total_time = 0;
    int completed = 0;
    while (completed < n) {
        int highest_priority_index = -1;
        int highest_priority = -1;
        for (int i = 0; i < n; i++) {
            if (p[i].at <= total_time && p[i].priority > highest_priority && p[i].rt > 0) {
                highest_priority_index = i;
                highest_priority = p[i].priority;
            }
        }
```

```
        if (highest_priority_index == -1) {
            total_time++;
            continue;
        }
        p[highest_priority_index].rt--;
        if (p[highest_priority_index].rt == 0) {
            completed++;
            wt[highest_priority_index] = total_time + 1 - p[highest_priority_index].at -
p[highest_priority_index].bt;
            if (wt[highest_priority_index] < 0) {
                wt[highest_priority_index] = 0;
            }
            tat[highest_priority_index] = wt[highest_priority_index] + p[highest_priority_index].bt;
        }
        total_time++;
    }
    float avg_wt = 0, avg_tat = 0;
    for (int i = 0; i < n; i++) {
        avg_wt += wt[i];
        avg_tat += tat[i];
    }
    avg_wt /= n;
    avg_tat /= n;
    printf("\nPreemptive Priority Scheduling:\n");
    printf("ID\tArrival Time\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i].id, p[i].at, p[i].bt, p[i].priority, wt[i],
tat[i]);
    }
    printf("Avg Waiting Time: %.2f\n", avg_wt);
    printf("Avg Turnaround Time: %.2f\n", avg_tat);
}
void priority_non_preemptive(Proc p[], int n) {
    int wt[n], tat[n];
    sort_by_priority(p, n);
    int total_time = p[0].at;
    for (int i = 0; i < n; i++) {
        wt[i] = total_time - p[i].at;
        if (wt[i] < 0) {
            wt[i] = 0;
            total_time = p[i].at;
```

```
        }
        tat[i] = wt[i] + p[i].bt;
        total_time += p[i].bt;
    }
    float avg_wt = 0, avg_tat = 0;
    for (int i = 0; i < n; i++) {
        avg_wt += wt[i];
        avg_tat += tat[i];
    }
    avg_wt /= n;
    avg_tat /= n;
    printf("\nNon-preemptive Priority Scheduling:\n");
    printf("ID\tArrival Time\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i].id, p[i].at, p[i].bt, p[i].priority, wt[i],
tat[i]);
    }
    printf("Avg Waiting Time: %.2f\n", avg_wt);
    printf("Avg Turnaround Time: %.2f\n", avg_tat);
}
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    Proc p[n];
    printf("Enter arrival time, burst time, and priority for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d:\n", i + 1);
        p[i].id = i + 1;
        printf("Arrival Time: ");
        scanf("%d", &p[i].at);
        printf("Burst Time: ");
        scanf("%d", &p[i].bt);
        printf("Priority: ");
        scanf("%d", &p[i].priority);
        p[i].rt = p[i].bt;
    }
    priority_non_preemptive(p, n);
    priority_preemptive(p, n);
    return 0;
}
```

```
Enter the number of processes: 4
Enter arrival time, burst time, and priority for each process:
Process 1:
Arrival Time: 0
Burst Time: 5
Priority: 10
Process 2:
Arrival Time: 1
Burst Time: 4
Priority: 20
Process 3:
Arrival Time: 2
Burst Time: 2
Priority: 30
Process 4:
Arrival Time: 4
Burst Time: 1
Priority: 40

Non-preemptive Priority Scheduling:
ID      Arrival Time    Burst Time      Priority        Waiting Time    Turnaround Time
1       0               5               10              0               5
2       1               4               20              4               8
3       2               2               30              7               9
4       4               1               40              7               8
Avg Waiting Time: 4.50
Avg Turnaround Time: 7.50

Preemptive Priority Scheduling:
ID      Arrival Time    Burst Time      Priority        Waiting Time    Turnaround Time
1       0               5               10              7               12
2       1               4               20              3               7
3       2               2               30              0               2
4       4               1               40              0               1
Avg Waiting Time: 2.50
Avg Turnaround Time: 5.50

Process returned 0 (0x0)   execution time : 79.280 s
Press any key to continue.
```

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

```
#include<stdio.h>
#include<stdlib.h>
struct queue{
   int pid;
   struct queue* next;
};
struct queue* rq = NULL;
struct queue* create(int p){
   struct queue* nn = malloc(sizeof(struct queue));
   nn->pid = p;
   nn->next = NULL;
   return nn;
}
void enqueue(int p){
   struct queue * nn = create(p);
   if(rq==NULL)
      rq=nn;
   else{
```

```
        struct queue* temp = rq;
        while(temp->next!=NULL)
            temp = temp->next;
        temp->next = nn;
    }
}
int dequeue(){
    int x=0;
    if (rq == NULL)
        return x;
    else{
        struct queue* temp = rq;
        x = temp->pid;
        rq = rq->next;
        free(temp);
    }
    return x;
}
void printq(){
    struct queue* temp = rq;
    while(temp!=NULL){
        printf("%d\t",temp->pid);
        temp = temp->next;
    }
    printf("\n");
}
void swap(int *a,int *b){
    *a = *a + *b;
    *b = *a - *b;
    *a = *a - *b;
}
void sort(int *pid, int *at, int *bt, int n){
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            if(at[i]<at[j]){
                swap(&at[i],&at[j]);
                swap(&bt[i],&bt[j]);
                swap(&pid[i],&pid[j]);
            }
        }
    }
```

```
}
int main(){
   int n,t,x=1;
   printf("Enter the number of processes:");
   scanf("%d",&n);
   printf("Enter the time quantum : ");
   scanf("%d",&t);
   int pid[n],at[n],bt1[n],ct[n],tat[n],wt[n],bt2[n],rt[n];
   for(int i=0;i<n;i++){
      printf("Enter arrival time and burst time : ");
      scanf("%d%d",&at[i],&bt1[i]);
      pid[i]=i+1;
   }
   sort(pid,at,bt1,n);
   enqueue(pid[0]);
   for(int i=0;i<n;i++){
      bt2[i]=bt1[i];
      rt[i] = -1;
   }

   int count = 0;
   int ctvar = at[0];
   while (count != n){
      int curp = rq->pid;
      int curi = 0;
      for(int i = 0;i<n;i++){
         if(pid[i] == curp){
            curi = i;
            break;
         }
      }
      if(rt[curi]==-1){
         rt[curi] = ctvar - at[curi];
      }
      if(bt2[curi]<=t){
         ctvar += bt2[curi];
         bt2[curi] = 0;
      }
      else{
         ctvar += t;
         bt2[curi] -=t;
```

```
        }
        while(at[x]<=ctvar && x<n){
            enqueue(pid[x]);
            x +=1;
        }
        if(bt2[curi]>0)
            enqueue(pid[curi]);
        if(bt2[curi] == 0){
            count +=1;
            ct[curi] = ctvar;
        }
        dequeue();
    }
    for(int i=0;i<n;i++){
        tat[i]=ct[i]-at[i];
        wt[i]=tat[i]-bt1[i];
    }
    float avg_tat=0;
    float avg_wt=0;
    for(int i=0;i<n;i++){
        avg_tat+=tat[i];
        avg_wt+=wt[i];
    }
    printf("pid\tat\tbt\tct\ttat\twt\trt\n\n");
    for(int i=0;i<n;i++){
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t",pid[i],at[i],bt1[i],ct[i],tat[i],wt[i],rt[i]);
        printf("\n");
    }
    printf("\nAverage Turn around time : %f",avg_tat/n);
    printf("\nAverage waiting time : %f",avg_wt/n);
    return 0;
}
```

```
Enter the number of processes:5
Enter the time quantum : 2
Enter arrival time and burst time : 0 5
Enter arrival time and burst time : 1 3
Enter arrival time and burst time : 2 1
Enter arrival time and burst time : 3 2
Enter arrival time and burst time : 4 3
pid     at      bt      ct      tat     wt      rt

1       0       5       13      13      8       0
2       1       3       12      11      8       1
3       2       1       5       3       2       2
4       3       2       9       6       4       4
5       4       3       14      10      7       5

Average Turn around time : 8.600000
Average waiting time : 5.800000
Process returned 0 (0x0)   execution time : 17.149 s
Press any key to continue.
```

Program - 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 sort(int proc_id[], int at[], int bt[], int n) {
   int min, temp;
   for(int i=0; i<n-1; i++) {
      for(int j=i+1; j<n; j++) {
         if(at[j] < at[i]) {
            temp = at[i];
            at[i] = at[j];
            at[j] = temp;
            temp = bt[i];
            bt[i] = bt[j];
            bt[j] = temp;
            temp = proc_id[i];
            proc_id[i] = proc_id[j];
            proc_id[j] = temp;
         }
      }
   }
}
void simulateFCFS(int proc_id[], int at[], int bt[], int n, int start_time) {
   int c = start_time, ct[n], tat[n], wt[n];
   double ttat = 0.0, twt = 0.0;
   for(int i=0; i<n; i++) {
      if(c >= at[i])
         c += bt[i];
      else
         c = at[i] + bt[i];
      ct[i] = c;
   }
   for(int i=0; i<n; i++)
      tat[i] = ct[i] - at[i];
   for(int i=0; i<n; i++)
      wt[i] = tat[i] - bt[i];
```

```
    printf("PID\tAT\tBT\tCT\tTAT\tWT\n");
    for(int i=0; i<n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", proc_id[i], at[i], bt[i], ct[i], tat[i], wt[i]);
        ttat += tat[i];
        twt += wt[i];
    }
    printf("Average Turnaround Time: %.2lf ms\n", ttat/n);
    printf("Average Waiting Time: %.2lf ms\n", twt/n);
}
void main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    int proc_id[n], at[n], bt[n], type[n];
    int sys_proc_id[n], sys_at[n], sys_bt[n], user_proc_id[n], user_at[n], user_bt[n];
    int sys_count = 0, user_count = 0;
    for(int i=0; i<n; i++) {
        proc_id[i] = i + 1;
        printf("Enter arrival time, burst time and type (0 for system, 1 for user) for process %d: ",
i+1);
        scanf("%d %d %d", &at[i], &bt[i], &type[i]);
        if(type[i] == 0) {
            sys_proc_id[sys_count] = proc_id[i];
            sys_at[sys_count] = at[i];
            sys_bt[sys_count] = bt[i];
            sys_count++;
        } else {
            user_proc_id[user_count] = proc_id[i];
            user_at[user_count] = at[i];
            user_bt[user_count] = bt[i];
            user_count++;
        }
    }
    sort(sys_proc_id, sys_at, sys_bt, sys_count);
    sort(user_proc_id, user_at, user_bt, user_count);
    printf("System Processes Scheduling:\n");
    simulateFCFS(sys_proc_id, sys_at, sys_bt, sys_count, 0);
    int system_end_time = 0;
    if (sys_count > 0) {
        system_end_time = sys_at[sys_count - 1] + sys_bt[sys_count - 1];
```

```
        for (int i = 0; i < sys_count - 1; i++) {
            if (sys_at[i + 1] > system_end_time) {
                system_end_time = sys_at[i + 1];
            }
            system_end_time += sys_bt[i];
        }
    }
    printf("\nUser Processes Scheduling:\n");
    simulateFCFS(user_proc_id, user_at, user_bt, user_count, system_end_time);
}
```

```
Enter number of processes: 4
Enter arrival time, burst time and type (0 for system, 1 for user) for process 1: 0 4 0
Enter arrival time, burst time and type (0 for system, 1 for user) for process 2: 0 3 0
Enter arrival time, burst time and type (0 for system, 1 for user) for process 3: 0 8 1
Enter arrival time, burst time and type (0 for system, 1 for user) for process 4: 10 5 0
System Processes Scheduling:
PID     AT      BT      CT      TAT     WT
1       0       4       4       4       0
2       0       3       7       7       4
4       10      5       15      5       0
Average Turnaround Time: 5.33 ms
Average Waiting Time: 1.33 ms

User Processes Scheduling:
PID     AT      BT      CT      TAT     WT
3       0       8       30      30      22
Average Turnaround Time: 30.00 ms
Average Waiting Time: 22.00 ms

Process returned 31 (0x1F)   execution time : 23.760 s
Press any key to continue.
```

Program - 4

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

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_TASKS 10
typedef struct{
   int Ti;
   int Ci;
   int deadline;
   int RT; //Remaining_Time
   int id;
} Task;
void Input(Task tasks[], int *n_tasks){
   printf("Enter number of tasks: ");
   scanf("%d", n_tasks);
   for (int i = 0; i < *n_tasks; i++){
      tasks[i].id = i + 1;
      printf("Enter Ti of task %d: ", i + 1);
      scanf("%d", &tasks[i].Ti);
      printf("Enter execution time of task %d: ", i + 1);
      scanf("%d", &tasks[i].Ci);
      tasks[i].deadline = tasks[i].Ti; // In RM, deadline is equal to Ti
      tasks[i].RT = tasks[i].Ci;
   }
}
int compare_by_period(const void *a, const void *b){
   return ((Task*)a)->Ti - ((Task*)b)->Ti;
}
void RMS(Task tasks[], int n_tasks, int time_frame){
   qsort(tasks, n_tasks, sizeof(Task), compare_by_period);
   printf("\nRate-Monotonic Scheduling:\n");
   for (int time = 0; time < time_frame; time++) {
      int s_task = -1;
      for (int i = 0; i < n_tasks; i++) {
         if (time % tasks[i].Ti == 0) {
            tasks[i].RT = tasks[i].Ci;
         }
         if (tasks[i].RT > 0 && (s_task == -1 || tasks[i].Ti < tasks[s_task].Ti)){
            s_task = i;
         } }
```

```
        if (s_task != -1){
            printf("Time %d: Task %d\n", time, tasks[s_task].id);
            tasks[s_task].RT--;
        } else {
            printf("Time %d: Idle\n", time);
        }
    }
}
int main(){
    Task tasks[MAX_TASKS];
    int n_tasks;
    int time_frame;
    Input(tasks, &n_tasks);
    printf("Enter time frame for simulation: ");
    scanf("%d", &time_frame);
    RMS(tasks, n_tasks, time_frame);
    return 0;
}
```

```
Enter execution time of task 2: 2
Enter Ti of task 3: 10
Enter execution time of task 3: 2
Enter time frame for simulation: 20

Rate-Monotonic Scheduling:
Time 0: Task 2
Time 1: Task 2
Time 2: Task 3
Time 3: Task 3
Time 4: Task 1
Time 5: Task 2
Time 6: Task 2
Time 7: Task 1
Time 8: Task 1
Time 9: Idle
Time 10: Task 2
Time 11: Task 2
Time 12: Task 3
Time 13: Task 3
Time 14: Idle
Time 15: Task 2
Time 16: Task 2
Time 17: Idle
Time 18: Idle
Time 19: Idle

Process returned 0 (0x0)   execution time : 13.966 s
Press any key to continue.
```

b) Earliest-deadline First

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_TASKS 10
typedef struct{
```

```
    int Ti;
    int Ci;
    int deadline;
    int RT; // remaining time
    int n_deadline; // next_deadline
    int id;
} Task;
void Input(Task tasks[], int *n_tasks){
    printf("Enter number of tasks: ");
    scanf("%d", n_tasks);
    for (int i = 0; i < *n_tasks; i++){
        tasks[i].id = i + 1;
        printf("Enter Ti of task %d: ", i + 1);
        scanf("%d", &tasks[i].Ti);
        printf("Enter execution time of task %d: ", i + 1);
        scanf("%d", &tasks[i].Ci);
        printf("Enter deadline of task %d: ", i + 1);
        scanf("%d", &tasks[i].deadline);
        tasks[i].RT = tasks[i].Ci;
        tasks[i].n_deadline = tasks[i].deadline; // Initialize the next deadline
    }
}
void EDF(Task tasks[], int n_tasks, int time_frame){
    printf("\nEarliest-Deadline First Scheduling:\n");
    for (int time = 0; time < time_frame; time++) {
        int s_task = -1;
        for (int i = 0; i < n_tasks; i++){
            if (time % tasks[i].Ti == 0){
                tasks[i].RT = tasks[i].Ci;
                tasks[i].n_deadline = time + tasks[i].deadline;
            }
        }
        for (int i = 0; i < n_tasks; i++){
            if (tasks[i].RT > 0 && (s_task == -1 || tasks[i].n_deadline < tasks[s_task].n_deadline)) {
                s_task = i;
            }
        }
        if (s_task != -1){
            printf("Time %d: Task %d\n", time, tasks[s_task].id);
            tasks[s_task].RT--;
        } else{
            printf("Time %d: Idle\n", time);
        }
    }
```

```
}
int main(){
   Task tasks[MAX_TASKS];
   int n_tasks;
   int time_frame;
   Input(tasks, &n_tasks);
   printf("Enter time frame for simulation: ");
   scanf("%d", &time_frame);
   EDF(tasks, n_tasks, time_frame);
   return 0;
}
```

```
Enter Ti of task 3: 10
Enter execution time of task 3: 2
Enter deadline of task 3: 8
Enter time frame for simulation: 20

Earliest-Deadline First Scheduling:
Time 0: Task 2
Time 1: Task 2
Time 2: Task 1
Time 3: Task 1
Time 4: Task 1
Time 5: Task 3
Time 6: Task 3
Time 7: Task 2
Time 8: Task 2
Time 9: Idle
Time 10: Task 2
Time 11: Task 2
Time 12: Task 3
Time 13: Task 3
Time 14: Idle
Time 15: Task 2
Time 16: Task 2
Time 17: Idle
Time 18: Idle
Time 19: Idle

Process returned 0 (0x0)   execution time : 263.812 s
Press any key to continue.
```

c) Proportional scheduling

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX_TASKS 10
#define MAX_TICKETS 100
#define TIME_UNIT_DURATION_MS 100
struct Task {
```

```
    int tid;
    int tickets;
};
void schedule(struct Task tasks[], int num_tasks, int *time_span_ms) {
    int total_tickets = 0;
    for (int i = 0; i < num_tasks; i++) {
        total_tickets += tasks[i].tickets;
    }
    srand(time(NULL));
    int current_time = 0;
    int completed_tasks = 0;
        printf("Process Scheduling:\n");
    while (completed_tasks < num_tasks) {
        int winning_ticket = rand() % total_tickets;
        int cumulative_tickets = 0;
        for (int i = 0; i < num_tasks; i++) {
            cumulative_tickets += tasks[i].tickets;
            if (winning_ticket < cumulative_tickets) {
                printf("Time %d-%d: Task %d is running\n", current_time, current_time + 1,
tasks[i].tid);
                current_time++;
                break;
            }
        }
        completed_tasks++;
    }
    *time_span_ms = current_time * TIME_UNIT_DURATION_MS;
}
int main() {
    struct Task tasks[MAX_TASKS];
    int num_tasks;
    int time_span_ms;
    printf("Enter the number of tasks: ");
    scanf("%d", &num_tasks);

    if (num_tasks <= 0 || num_tasks > MAX_TASKS) {
        printf("Invalid number of tasks. Please enter a number between 1 and %d.\n",
MAX_TASKS);
        return 1;
    }
    printf("Enter number of tickets for each task:\n");
    for (int i = 0; i < num_tasks; i++) {
        tasks[i].tid = i + 1;
        printf("Task %d tickets: ", tasks[i].tid);
```

```
    scanf("%d", &tasks[i].tickets);
  }
  printf("\nRunning tasks:\n");
  schedule(tasks, num_tasks, &time_span_ms);
  printf("\nTime span of the Gantt chart: %d milliseconds\n", time_span_ms);
  return 0;
}
```

```
Enter the number of tasks: 3
Enter number of tickets for each task:
Task 1 tickets: 10
Task 2 tickets: 20
Task 3 tickets: 30

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

Time span of the Gantt chart: 300 milliseconds

Process returned 0 (0x0)   execution time : 8.244 s
Press any key to continue.
```

Program – 5

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

```
#include<stdio.h>
#include<stdlib.h>
int mutex = 1, full = 0, empty = 5, x = 0;
int wait(int s){
      return (--s);
}
int signal(int s){
      return (++s);
}
void producer(){
     mutex = wait(mutex);
     full = signal(full);
     empty = wait(empty);
     x++;
     printf("Producer produces the item %d\n", x);
     mutex = signal(mutex);
}
void consumer(){
    mutex = wait(mutex);
    full = wait(full);
    empty = signal(empty);
    printf("Consumer consumes item %d\n", x);
    x--;
    mutex = signal(mutex);
 }
int main(){
int n;
void producer();
void consumer();
int wait(int);
int signal(int);
printf("\n1.Producer\n2.Consumer\n3.Exit");
while (1){
     printf("\nEnter your choice:");
     scanf("%d", &n);
     switch (n){
          case 1:
              if ((mutex == 1) && (empty != 0))
              producer();
              else
```

```
            printf("Buffer is full!!\n");
            break;
        case 2:
            if ((mutex == 1) && (full != 0))
                consumer();
            else
                printf("Buffer is empty!!\n");
                break;
        case 3:
            exit(0);
            break;
      }
  }
  return 0;
}
```

```
1.Producer
2.Consumer
3.Exit
Enter your choice:2
Buffer is empty!!

Enter your choice:1
Producer produces the item 1

Enter your choice:1
Producer produces the item 2

Enter your choice:2
Consumer consumes item 2

Enter your choice:1
Producer produces the item 2

Enter your choice:1
Producer produces the item 3

Enter your choice:2
Consumer consumes item 3

Enter your choice:2
Consumer consumes item 2

Enter your choice:2
Consumer consumes item 1

Enter your choice:1
Producer produces the item 1

Enter your choice:2
Consumer consumes item 1

Enter your choice:3

Process returned 0 (0x0)   execution time : 33.293 s
Press any key to continue.
```

Program – 6

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

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_PHILOSOPHERS 10
typedef enum { THINKING, HUNGRY, EATING } state_t;
state_t states[MAX_PHILOSOPHERS];
int num_philosophers;
int num_hungry;
int hungry_philosophers[MAX_PHILOSOPHERS];
int forks[MAX_PHILOSOPHERS];
void print_state() {
   printf("\n");
   for (int i = 0; i < num_philosophers; ++i) {
      if (states[i] == THINKING) printf("P %d is thinking\n", i + 1);
      else if (states[i] == HUNGRY) printf("P %d is waiting\n", i + 1);
      else if (states[i] == EATING) printf("P %d is eating\n", i + 1);
   }
}
int can_eat(int philosopher_id) {
   int left_fork = philosopher_id;
   int right_fork = (philosopher_id + 1) % num_philosophers;
   if (forks[left_fork] == 0 && forks[right_fork] == 0) {
      forks[left_fork] = forks[right_fork] = 1;
      return 1;
   }
   return 0;
}
void simulate(int allow_two) {
   int eating_count = 0;
   for (int i = 0; i < num_hungry; ++i) {
      int philosopher_id = hungry_philosophers[i];
      if (states[philosopher_id] == HUNGRY) {
         if (can_eat(philosopher_id)) {
            states[philosopher_id] = EATING;
            eating_count++;
            printf("P %d is granted to eat\n", philosopher_id + 1);
            if (!allow_two && eating_count == 1) break;
            if (allow_two && eating_count == 2) break;
         }
      }
   }
```

```
    for (int i = 0; i < num_hungry; ++i) {
        int philosopher_id = hungry_philosophers[i];
        if (states[philosopher_id] == EATING) {
            int left_fork = philosopher_id;
            int right_fork = (philosopher_id + 1) % num_philosophers;
            forks[left_fork] = forks[right_fork] = 0;
            states[philosopher_id] = THINKING;
        }
    }
}
int main() {
    printf("Enter the total number of philosophers (max %d): ", MAX_PHILOSOPHERS);
    scanf("%d", &num_philosophers);
    if (num_philosophers < 2 || num_philosophers > MAX_PHILOSOPHERS) {
        printf("Invalid number of philosophers. Exiting.\n");
        return 1;
    }
    printf("How many are hungry: ");
    scanf("%d", &num_hungry);
    for (int i = 0; i < num_hungry; ++i) {
        printf("Enter philosopher %d position: ", i + 1);
        int position;
        scanf("%d", &position);
        hungry_philosophers[i] = position - 1;
        states[hungry_philosophers[i]] = HUNGRY;
    }
    for (int i = 0; i < num_philosophers; ++i) {
        forks[i] = 0;
    }
    int choice;
    do {
        print_state();
        printf("\n1. One can eat at a time\n");
        printf("2. Two can eat at a time\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                simulate(0);
                break;
            case 2:
                simulate(1);
                break;
```

```
        case 3:
          printf("Exiting.\n");
          break;
        default:
          printf("Invalid choice. Please try again.\n");
          break;
      }
    } while (choice != 3);
    return 0;
}
```

```
Enter the total number of philosophers (max 10): 5
How many are hungry: 3
Enter philosopher 1 position: 1
Enter philosopher 2 position: 3
Enter philosopher 3 position: 5

P 1 is waiting
P 2 is thinking
P 3 is waiting
P 4 is thinking
P 5 is waiting

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

P 1 is thinking
P 2 is thinking
P 3 is waiting
P 4 is thinking
P 5 is waiting

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

P 1 is thinking
P 2 is thinking
P 3 is thinking
P 4 is thinking
P 5 is waiting
```

```
1. One can eat at a time
2. Two can eat at a time
3. Exit
Enter your choice: 2
P 5 is granted to eat

P 1 is thinking
P 2 is thinking
P 3 is thinking
P 4 is thinking
P 5 is thinking

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

Process returned 0 (0x0)   execution time : 45.076 s
Press any key to continue.
```

Program – 7

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

```
#include <stdio.h>
int p, r;
void avai(int all[p][r], int tot[r], int avail[r]) {
   for (int j = 0; j < r; j++) {
      avail[j] = 0;
      for (int k = 0; k < p; k++) {
         avail[j] += all[k][j];
      }
   }
   for (int j = 0; j < r; j++) {
      avail[j] = tot[j] - avail[j];
   }
}
void need1(int all[p][r], int max[p][r], int needmat[p][r]) {
   for (int i = 0; i < p; i++) {
      for (int j = 0; j < r; j++) {
         needmat[i][j] = max[i][j] - all[i][j];
      }
   }
}
void safety(int p, int r, int all[p][r], int avail[r], int needmat[p][r], int seq[p]) {
   int f[p], c, count = 0, h = 0;
   for (int i = 0; i < p; i++) {
      f[i] = 0;
   }
   while (count < p && h < p) {
      for (int i = 0; i < p; i++) {
         if (f[i] == 0) {
            c = 0;
            for (int j = 0; j < r; j++) {
               if (needmat[i][j] <= avail[j]) {
                  c++;
               }
            }
            if (c == r) {
               printf("P%d is visited(\t",i);
               for (int k = 0; k < r; k++) {
                  avail[k] += all[i][k];
```

```
                printf("%d",avail[k]);
                printf("\t");
              }
              printf(")\n");
              f[i] = 1;
              count++;
              seq[h] = i;
              h++;
            }
          }
        }
      }
    }
    int main() {
      printf("Enter the number of processes: ");
      scanf("%d", &p);
      printf("Enter the number of resources: ");
      scanf("%d", &r);
      int tot[r], needmat[p][r], avail[r], seq[p];
      printf("Enter the total instances of each resource: ");
      for (int i = 0; i < r; i++) {
        scanf("%d", &tot[i]);
      }
      int all[p][r], max[p][r];
      printf("Enter the details of each process (allocation matrix):\n");
      for (int i = 0; i < p; i++) {
        for (int j = 0; j < r; j++) {
          scanf("%d", &all[i][j]);
        }
      }
      printf("Enter the maximum matrix:\n");
      for (int i = 0; i < p; i++) {
        for (int j = 0; j < r; j++) {
          scanf("%d", &max[i][j]);
        }
      }
      avai(all, tot, avail);
      need1(all, max, needmat);
      printf("Need matrix:\n");
      for (int i = 0; i < p; i++) {
        for (int j = 0; j < r; j++) {
          printf("%d\t", needmat[i][j]);
        }
        printf("\n");
```

```
    }
    safety(p, r, all, avail, needmat, seq);
    printf("Safe sequence is: ");
    for (int i = 0; i < p; i++) {
        printf("P%d\t", seq[i]);
    }
    return 0;
}
```

```
Enter the number of processes: 5
Enter the number of resources: 3
Enter the total instances of each resource: 10 5 7
Enter the details of each process (allocation matrix):
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the maximum matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Need matrix:
7       4       3
1       2       2
6       0       0
0       1       1
4       3       1
P1 is visited(  5       3       2       )
P3 is visited(  7       4       3       )
P4 is visited(  7       4       5       )
P0 is visited(  7       5       5       )
P2 is visited(  10      5       7       )
Safe sequence is: P1    P3      P4      P0      P2
Process returned 0 (0x0)   execution time : 147.308 s
Press any key to continue.
```

Program – 8

Write a C program to simulate deadlock detection

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

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

```
Enter the number of processes and number of types of resources:
4 3
Enter the allocated number of each type of resource needed by each process:
1 0 2
2 1 1
1 0 3
1 2 2
Enter the available number of each type of resource:
0 0 0
Enter the request number of each type of resource needed by each process:
0 0 1
1 0 2
0 0 0
3 3 0
Deadlock has occured:
The deadlock processes are:
P3
Process returned 1 (0x1)   execution time : 42.808 s
Press any key to continue.
```

Program – 9

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

```
#include <stdio.h>
struct Block {
   int block_no;
   int block_size;
   int is_free;
};
struct File {
   int file_no;
   int file_size;
};
void firstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
   printf("Memory Management Scheme - First Fit\n");
   printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");
   for (int i = 0; i < n_files; i++) {
      for (int j = 0; j < n_blocks; j++) {
         if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
            blocks[j].is_free = 0;
            printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", files[i].file_no, files[i].file_size,
blocks[j].block_no, blocks[j].block_size, blocks[j].block_size - files[i].file_size);
            break;
         }
      }
   }
}
void worstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
   printf("Memory Management Scheme - Worst Fit\n");
   printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");
   for (int i = 0; i < n_files; i++) {
      int worst_fit_block = -1;
      int max_fragment = -1;
      for (int j = 0; j < n_blocks; j++) {
         if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
            int fragment = blocks[j].block_size - files[i].file_size;
            if (fragment > max_fragment) {
               max_fragment = fragment;
```

```
                worst_fit_block = j;
            }
        }
    }
    if (worst_fit_block!= -1) {
        blocks[worst_fit_block].is_free = 0;
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", files[i].file_no, files[i].file_size,
blocks[worst_fit_block].block_no, blocks[worst_fit_block].block_size, max_fragment);
    }
  }
}
void bestFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
  printf("Memory Management Scheme - Best Fit\n");
  printf("File_no:\tFile_size:\tBlock_no:\tBlock_size:\tFragment\n");
  for (int i = 0; i < n_files; i++) {
    int best_fit_block = -1;
    int min_fragment = 10000;
    for (int j = 0; j < n_blocks; j++) {
      if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
        int fragment = blocks[j].block_size - files[i].file_size;
        if (fragment < min_fragment) {
          min_fragment = fragment;
          best_fit_block = j;
        }
      }
    }
    if (best_fit_block!= -1) {
      blocks[best_fit_block].is_free = 0;
      printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", files[i].file_no, files[i].file_size,
blocks[best_fit_block].block_no, blocks[best_fit_block].block_size, min_fragment);
    }
  }
}
int main() {
  int n_blocks, n_files;
  printf("Enter the number of blocks: ");
  scanf("%d", &n_blocks);
  printf("Enter the number of files: ");
  scanf("%d", &n_files);
  struct Block blocks[n_blocks];
  for (int i = 0; i < n_blocks; i++) {
    blocks[i].block_no = i + 1;
    printf("Enter the size of block %d: ", i + 1);
    scanf("%d", &blocks[i].block_size);
```

```
        blocks[i].is_free = 1;
    }
    struct File files[n_files];
    for (int i = 0; i < n_files; i++) {
        files[i].file_no = i + 1;
        printf("Enter the size of file %d: ", i + 1);
        scanf("%d", &files[i].file_size);
    }
    firstFit(blocks, n_blocks, files, n_files);
    printf("\n");
    for (int i = 0; i < n_blocks; i++) {
        blocks[i].is_free = 1;
    }
    worstFit(blocks, n_blocks, files, n_files);
    printf("\n");
    for (int i = 0; i < n_blocks; i++) {
        blocks[i].is_free = 1;
    }
    bestFit(blocks, n_blocks, files, n_files);
    return 0;
}
```

```
Enter the number of blocks: 5
Enter the number of files: 4
Enter the size of block 1: 400
Enter the size of block 2: 700
Enter the size of block 3: 200
Enter the size of block 4: 300
Enter the size of block 5: 600
Enter the size of file 1: 212
Enter the size of file 2: 517
Enter the size of file 3: 312
Enter the size of file 4: 526
Memory Management Scheme - First Fit
File_no:        File_size:      Block_no:       Block_size:     Fragment
1               212             1               400             188
2               517             2               700             183
3               312             5               600             288

Memory Management Scheme - Worst Fit
File_no:        File_size:      Block_no:       Block_size:     Fragment
1               212             2               700             488
2               517             5               600             83
3               312             1               400             88

Memory Management Scheme - Best Fit
File_no:        File_size:      Block_no:       Block_size:     Fragment
1               212             4               300             88
2               517             5               600             83
3               312             1               400             88
4               526             2               700             174

Process returned 0 (0x0)   execution time : 36.762 s
Press any key to continue.
```

Program – 10

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

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
void fifo(int pages[], int n, int capacity) {
  int frame[capacity], index = 0, page_faults = 0;
  for (int i = 0; i < capacity; i++)
    frame[i] = -1;
  for (int i = 0; i < n; i++) {
    int found = 0;
    for (int j = 0; j < capacity; j++) {
      if (frame[j] == pages[i]) {
        found = 1;
        break;
      }
    }
    if (!found) {
      frame[index] = pages[i];
      index = (index + 1) % capacity;
      page_faults++;
    }
  }
  printf("FIFO Page Faults: %d\n", page_faults);
}
void lru(int pages[], int n, int capacity) {
  int frame[capacity], counter[capacity], time = 0, page_faults = 0;
  for (int i = 0; i < capacity; i++) {
    frame[i] = -1;
    counter[i] = 0;
  }
  for (int i = 0; i < n; i++) {
    int found = 0;
    for (int j = 0; j < capacity; j++) {
      if (frame[j] == pages[i]) {
        found = 1;
        counter[j] = time++;
        break;
      }
    }
    if (!found) {
      int min = INT_MAX, min_index = -1;
```

```
            for (int j = 0; j < capacity; j++) {
                if (counter[j] < min) {
                    min = counter[j];
                    min_index = j;
                }
            }
            frame[min_index] = pages[i];
            counter[min_index] = time++;
            page_faults++;
        }
    }
    printf("LRU Page Faults: %d\n", page_faults);
}
void optimal(int pages[], int n, int capacity) {
    int frame[capacity], page_faults = 0;
    for (int i = 0; i < capacity; i++)
        frame[i] = -1;
    for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = 0; j < capacity; j++) {
            if (frame[j] == pages[i]) {
                found = 1;
                break;
            }
        }
        if (!found) {
            int farthest = i + 1, index = -1;
            for (int j = 0; j < capacity; j++) {
                int k;
                for (k = i + 1; k < n; k++) {
                    if (frame[j] == pages[k])
                        break;
                }
                if (k > farthest) {
                    farthest = k;
                    index = j;
                }
            }
            if (index == -1) {
                for (int j = 0; j < capacity; j++) {
                    if (frame[j] == -1) {
                        index = j;
                        break;
                    }
```

```
                }
            }
            frame[index] = pages[i];
            page_faults++;
        }
    }
    printf("Optimal Page Faults: %d\n", page_faults);
}
int main() {
    int n, capacity;
    printf("Enter the number of pages: ");
    scanf("%d", &n);
    int *pages = (int*)malloc(n * sizeof(int));
    printf("Enter the pages: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &pages[i]);
    printf("Enter the frame capacity: ");
    scanf("%d", &capacity);
    printf("\nPages: ");
    for (int i = 0; i < n; i++)
        printf("%d ", pages[i]);
    printf("\n\n");
    fifo(pages, n, capacity);
    lru(pages, n, capacity);
    optimal(pages, n, capacity);
    free(pages);
    return 0;
}
```

```
Enter the number of pages: 20
Enter the pages: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter the frame capacity: 3

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

FIFO Page Faults: 15
LRU Page Faults: 12
Optimal Page Faults: 9

Process returned 0 (0x0)   execution time : 33.534 s
Press any key to continue.
```