VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

OPERATING SYSTEMS

Submitted by

VARUN R(1BM24CS430)

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
Feb-2025 to June-2025

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 Varun R(1BM24CS430), who is Bonafide student of B. M. S. College of Engineering. It is in partial fulfilment for the award of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum during the year Feb 2025-June 2025. 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. Sandhya A Kulkarni
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Kavitha Sooda
Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Sl. No.	Experiment Title	Page No.
1.	Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time. →FCFS → SJF (pre-emptive & Non-preemptive)	1-16
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)	17-36
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.	37-44
4.	Write a C program to simulate Real-Time CPU Scheduling algorithms: a) Rate- Monotonic b) Earliest-deadline First	45-56
5.	Write a C program to simulate producer-consumer problem using semaphores	57-63
6.	Write a C program to simulate the concept of Dining Philosophers problem.	63-70
7.	Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.	71-77
8.	Write a C program to simulate deadlock detection	77-82
9.	Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit b) Best-fit c) First-fit	83-95

10.	Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) Optimal	95-105

Course Outcomes

C01	Apply the different concepts and functionalities of Operating System
C02	Analyse various Operating system strategies and techniques
C03	Demonstrate the different functionalities of Operating System.
C04	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

→ SJF (pre-emptive & Non-preemptive)

→FCFS

```
#include <stdio.h>

int main() {
    int n, i, at[10], bt[10], ct[10], tat[10], wt[10], rt[10];
    float total_tat = 0, total_wt = 0;

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

    for(i = 0; i < n; i++) {
        printf("P%d AT BT: ", i+1);
        scanf("%d %d", &at[i], &bt[i]);
    }

    int time = 0;
    for(i = 0; i < n; i++) {
        if(time < at[i]) time = at[i];
        rt[i] = time - at[i];   // Response time
        ct[i] = time + bt[i];
        tat[i] = ct[i] - at[i];
        wt[i] = tat[i] - bt[i];
        time = ct[i];

        total_tat += tat[i];
        total_wt += wt[i];
```

```
    }

    printf("\nP\tAT\tBT\tCT\tTAT\tWT\tRT\n");
    for(i = 0; i < n; i++)
        printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n", i+1, at[i], bt[i], ct[i], tat[i], wt[i], rt[i]);

    printf("\nAvg TAT = %.2f\nAvg WT = %.2f\n", total_tat / n, total_wt / n);

    return 0;
}
```

LAB

1) Write a c program to simulate the following CPU Scheduling algorithm to find turn-around time and waiting time (Any one).

a) FCFS

```
#include <stdio.h>

struct process {
        int at;
        int bt;
        int ct;
        int tat;
        int wat;
        int rt;

};

Void CalculateFCFS (Struct Process proc[ ], int n){
        int time = 0;
        for (int i = 0; i < n; i++) {
            if (time < proc[i].at) {
                time = proc[i].at);
            }
 Proc[i].ct = time + proc[i].bt;
 proc[i].tat = proc[i].ct - proc[i].at;
 Proc[i].wt = proc[i].tat - proc[i].bt;
 Proc[i].rt = time - proc[i].at;
```

```
time = proc[i].ct;
    }
}
int main() {
    int n;
    Printf(" Enter the no of process:");
    Scanf ("%d", &n);
    Struct Process proc[n];
Printf(" Enter Arrival Time & Burst Time for each
Process:\n");
    for(int i=0; i<n; i++){
        Printf("P%d Arrival Time:", i+1);
        Scanf("%d", &proc[i].at);
        Printf("P%d Burst time: " i+1);
        Scanf("%d", &proc[i].bt);
    }
CalculateFCFS(proc, n);
Printf("\n process\TAT\tBT\tCT\tTAT\tWT\tRT\n");
for(int i=0; i<n; i++){
printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
+1, proc[i].at, proc[i].bt, proc[i].ct, proc[i].tat,
proc[i].wt, proc[i].rt);
    } return0;
```

O/P->

Enter the number of process : 4

Enter Arrival Time and Burst Time for each process:

P_1 Arrival Time : 0

P_1 Burst Time : 7

P_2 Arrival Time : 0

P_2 Burst Time : 3

P_3 Arrival Time : 0

P_3 Burst Time 4

P_4 Arrival Time : 0

P_4 Burst Time : 6

Process	AT	BT	CT	TAT	WT	RT
P_1	0	7	7	7	0	0
P_2	0	3	10	10	7	7
P_3	0	4	14	14	10	10
P_4	0	6	20	20	14	14

6/3/25

Gantt ~~chart~~

→SJF(Non Preemptive)

```
#include <stdio.h>
#include <limits.h>
struct Process {
    int at;
    int bt;
    int ct;
    int tat;
    int wt;
    int rt;
    int pid;
};
void calculateSJF(struct Process proc[], int n) {
    int time = 0;
    int completed = 0;
    int min_index;
    int is_completed[n];
    for (int i = 0; i < n; i++) {
        is_completed[i] = 0;
    }
    while (completed < n) {
        min_index = -1;
        int min_bt = INT_MAX;
        for (int i = 0; i < n; i++) {
            if (proc[i].at <= time && !is_completed[i] && proc[i].bt < min_bt) {
                min_bt = proc[i].bt;
                min_index = i;
            }
```

```
        }
        if (min_index == -1) {
            time++;
        } else {
            proc[min_index].ct = time + proc[min_index].bt;
            proc[min_index].tat = proc[min_index].ct - proc[min_index].at;
            proc[min_index].wt = proc[min_index].tat - proc[min_index].bt;
            proc[min_index].rt = time - proc[min_index].at;
            time = proc[min_index].ct;
            is_completed[min_index] = 1;
            completed++;
        }
    }
}
int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    struct Process proc[n];
    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("P%d Arrival Time: ", i + 1);
        scanf("%d", &proc[i].at);
        printf("P%d Burst Time: ", i + 1);
        scanf("%d", &proc[i].bt);
    }
    calculateSJF(proc, n);
    printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++) {
```

```
        printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n", proc[i].pid, proc[i].at, proc[i].bt, proc[i].ct,
proc[i].tat, proc[i].wt, proc[i].rt);
    }
    return 0;
}
```

```
C:\Users\Admin\Desktop\sjf.exe

Enter number of processes: 4
Enter Arrival Time and Burst Time for each process:
P1 Arrival Time: 0
P1 Burst Time: 7
P2 Arrival Time: 0
P2 Burst Time: 3
P3 Arrival Time: 0
P3 Burst Time: 4
P4 Arrival Time: 0
P4 Burst Time: 6

Process AT      BT      CT      TAT     WT      RT
P1      0       7       20      20      13      13
P2      0       3       3       3       0       0
P3      0       4       7       7       3       3
P4      0       6       13      13      7       7

Average Waiting Time: 5.75
Average Turnaround Time: 10.75

Process returned 0 (0x0)   execution time : 88.594 s
Press any key to continue.
```

b) SJF (Non-Preemptive):-

->

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

struct process {
    int at;
    int bt;
    int ct;
    int tat;
    int wt;
    int rt;
    int pid;
};

Void CalculateSJF (Struct Process proc[], int n){
    int time = 0;
    int completed = 0;
    int min_index;
    int is_completed [n];
    for (int i = 0; i < n; i++){
        is_completed [i] = 0;
    }
    while (completed < n){
        min_index = -1;
        int min_bt = Int_max;
        for (int i = 0; i < n; i++){
        if (proc [i].at <= time && ! is_completed [i]
```

```
if proc[i].bt < min_bt){
        min_bt = proc[i].bt;
        min_index = i;
    }
}
  if (min_index == -1){
      time++;
  }else{
Proc[min_index].ct = time + proc[min_index].bt;
Proc[min_index].tat = proc[min_index].ct - proc[min-
index].at;
Proc[min_index].wt = proc[min_index].tat - proc[
min_index].bt;
  time = proc[min_index].ct;
  is_Completed[min_index] ~~ct~~ = 1;
  ~~Completed++~~;
  }
}
}
int main(){
    int n;
    Printf("Enter no of process");
    Scanf("%d", &n);
    Struct process proc[n];
Printf("Enter Arrival Time & Burst Time for each \n")
    for (int i=0; i<n; i++){
        Proc[i].pid = i+1;
        Printf("P%d Arrival Time:", i+1);
```

```
Scanf("%d", &proc[i].at);
printf("P%d Burst Time:", i+1);
Scanf("%d", proc[i].bt);
}

Calculate SJF (proc, n);
Printf("\n process\tAT\tBT\t CT\tTAT\t WT\tRT,
\n");
for(int i=0; i<n; i++){
printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
Proc[i].pid, proc[i].at, proc[i].bt, proc[i].ct,
proc[i].tat, proc[i].wt, proc[i].rt);
}
return 0;
}
```

O/P →

Enter the number of process : 4

Enter the Arrival Time & Burst Time for each

Process :

P_1 Arrival Time : 0

P_1 Burst Time : 7

P_2 Arrival Time : 0

P_2 Burst Time : 3

P_3 Arrival Time : 0

P_4 Arrival Time : 0

P_4 Burst Time : 6

Process	AT	BT	CT	TAT	WT	RT
P_1	0	7	20	20	13	13
P_2	0	3	3	3	0	0
P_3	0	4	7	7	3	3
P_4	0	6	13	13	7	7

AY
6/3/25

10

→SJF(Pre-emptive)

```
#include <stdio.h>
#define MAX 10
typedef struct {
    int pid, at, bt, rt, wt, tat, completed;
} Process;

void sjf_preemptive(Process p[], int n) {
    int time = 0, completed = 0, shortest = -1, min_bt = 9999;

    while (completed < n) {
        shortest = -1;
        min_bt = 9999;
        for (int i = 0; i < n; i++) {
            if (p[i].at <= time && p[i].rt > 0 && p[i].rt < min_bt) {
                min_bt = p[i].rt;
                shortest = i;
            }
        }
        if (shortest == -1) {
            time++;
            continue;
        }
        p[shortest].rt--;
        time++;
        if (p[shortest].rt == 0) {
            p[shortest].completed = 1;
            completed++;
            p[shortest].tat = time - p[shortest].at
```

```
 p[shortest].wt = p[shortest].tat - p[shortest].bt;
        }
    }
}

int main() {
    Process p[MAX];
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("Enter arrival time and burst time for process %d: ", p[i].pid);
        scanf("%d %d", &p[i].at, &p[i].bt);
        p[i].rt = p[i].bt;
        p[i].completed = 0;
    }
    sjf_preemptive(p, n);
    printf("\nPID\tAT\tBT\tWT\tTAT\n");
    for (int i = 0; i < n; i++)
        printf("%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].wt, p[i].tat);
    return 0;
}
```

```
"C:\Users\aDMIN\Documents\vicky042\SJF preemptive.exe"
Enter number of processes: 4
Enter arrival time and burst time for process 1: 0
8
Enter arrival time and burst time for process 2: 1
4
Enter arrival time and burst time for process 3: 2
9
Enter arrival time and burst time for process 4: 3
5

PID     AT      BT      WT      TAT
1       0       8       9       17
2       1       4       0       4
3       2       9       15      24
4       3       5       2       7

Process returned 0 (0x0)   execution time : 37.778 s
Press any key to continue.
```

b) SJF (Preemptive):-

```
#include <stdio.h>
# define MAX 100

typedef struct {
        int at, bt, rt, ct, tat, wt;
  } Process;

int main(){
    int n, completed = 0; time = 0;
    min_rt, shortest;
    Process p[max];
    Printf ("Enter the no of process: ");
    Scanf ("%d", &n)

for (int i = 0; i < n ; i++){
    Printf ("AT & BT of P%d: ", i+1);
    Scanf ("%d %d", &p[i].at , &p[i].bt);
    p[i].rt = p[i].bt;
  }

while (completed < n> {
    Shortest = -1, min_rt = 1000;

    for (int i = 0; i < n ; i++)
        if (p[i].at <= time && p[i].rt > 0 &&
```

```
p[i].rt < min_rt)
 min_rt = p[i].rt, shortest = i;

if (shortest == -1)
        { time++;
          continue;
        }

 p[shortest].rt--;

 if (p[shortest].rt == 0) {
   p[shortest].ct = time+1;
   p[shortest].tat = p[shortest].ct - p[shortest].at;
   p[shortest].wt = p[shortest].tat - p[shortest].bt;
   Complited++;
 }
   time++;
 }

Printf("\nAt\t BT\t TAT\t WT\n");
 float total TAT = 0; totalWT = 0;
   for (int i=0; i<n; i++) {

      Printf("%d\t%d\t%d\t%d\t%d\n",
p[i].at, p[i].bt, p[i].ct, p[i].tat, p[i].wt);
   TotalTAT += p[i].tat;
   TotalWT += p[i].wt;
   }
   Printf("\nAvg TAT: %.2f, Avg wt: %.2f\n",
 TotalTAT/n, Totalwt/n);
```

```
return 0;
}
```

O/P ->

Enter number of Processes: 4

AT & BT of P_1 : 0 8

AT & BT of P_2 : 1 4

AT & BT of P_3 : 2 9

AT & BT of P_4 : 3 5

AT	BT	CT	TAT	WT
0	8	17	17	9
1	4	5	4	0
2	9	26	24	15
3	5	10	7	2

AvgTAT: 13.00 , AvgWT: 6.50

Program 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)

→Priority(Non-pre-emptive)

```
#include <stdio.h>
struct Process {
    int pid, at, bt, pr, ct, wt, tat, rt;
    int isCompleted; // Flag to check if process is completed
}
void sortByArrival(struct Process p[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (p[i].at > p[j].at) {
                struct Process temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }
}
void findPriorityScheduling(struct Process p[], int n) {
    sortByArrival(p, n);
    int time = 0, completed = 0;
    float totalWT = 0, totalTAT = 0;
```

```
while (completed < n) {
    int idx = -1, highestPriority = 9999;
    for (int i = 0; i < n; i++) {
        if (p[i].at <= time && p[i].isCompleted == 0) {
            if (p[i].pr < highestPriority) {
                highestPriority = p[i].pr;
                idx = i;
            }
        }
    }

    if (idx == -1) {
        time++; // CPU idle
    } else {
        p[idx].rt = time - p[idx].at;
        time += p[idx].bt;
        p[idx].ct = time;
        p[idx].tat = p[idx].ct - p[idx].at;
        p[idx].wt = p[idx].tat - p[idx].bt;
        p[idx].isCompleted = 1;

        totalWT += p[idx].wt;
        totalTAT += p[idx].tat;
        completed++;
    }
}

printf("PID\tAT\tBT\tPR\tCT\tTAT\tWT\tRT\n");
for (int i = 0; i < n; i++) {
    printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
```

```
            p[i].pid, p[i].at, p[i].bt, p[i].pr, p[i].ct, p[i].tat, p[i].wt, p[i].rt);
    }

    printf("\nAverage Turnaround Time: %.2f\n", totalTAT / n);
    printf("Average Waiting Time: %.2f\n", totalWT / n);
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process p[n];

    printf("Enter Arrival Time, Burst Time, and Priority for each process:\n");
    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("Process %d: ", i + 1);
        scanf("%d %d %d", &p[i].at, &p[i].bt, &p[i].pr);
        p[i].isCompleted = 0;
    }

    findPriorityScheduling(p, n);
    return 0;
}
```

```
Enter the number of processes: 5
Enter Arrival Time, Burst Time, and Priority for each process:
Process 1: 0
10
3
Process 2: 0
1
1
Process 3: 0
2
4
Process 4: 0
1
5
Process 5: 0
5
2
PID AT  BT  PR  CT  TAT WT  RT
1   0   10  3   16  16  6   6
2   0   1   1   1   1   0   0
3   0   2   4   18  18  16  16
4   0   1   5   19  19  18  18
5   0   5   2   6   6   1   1

Average Turnaround Time: 12.00
Average Waiting Time: 8.20
```

Priority - Non Preemptive:

```
#include <stdio.h>
struct process {
    int pid, at, bt, pr, ct, wt, tat, rt;
    int isCompleted
};
void sortByArrival(struct process p[], int n) {
    for (int i=0; i<n-1; i++) {
        for (int j = i+1, j<n; j++) {
            if (p[i].at > p[j].at) {
                struct process temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }}
    }}}

void findPriorityScheduling(struct process p[], int n) {
    sortByArrival(p, n);
    int time=0; Completed = 0;
    float totalWT=0, TotalTAT=0;
    while (completed <n)
        int idx = -1, highestPriority = 9999;

    for (int i=0; i<n; i++) {
        if (p[i].at<= time && p[i].isCompleted
    ==0) {
```

```
if(p[i].pr < highestPriority){
    highestPriority = p[i].pr;
    idx = i;
}}
}
if (idx == -1){
    time++;
}else{
p[idx].st = time - p[idx].at;
time += p[idx].bt;
p[idx].ct = time;
p[idx].tat = p[idx].ct - p[idx].at;
p[idx].wt = p[idx].tat - p[idx].bt;
p[idx].isCompleted = 1;
}
}
Printf("Pid\tAT\tBt\tPR\tCT\tTAT\t WT\tRT\n");
for(int i=0; i<n; i++){
Printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
p[i].pid, p[i].at, p[i].bt, p[i].pr, p[i].ct, p[i].tat,
p[i].wt, p[i].st);
}
Printf("\Average Turnaround Time : %.2f\n",
Total TAT/n);
Printf("\Average Waiting Time : %.2f\n", TotalWT/n);
}
```

```
int main(){
  int n;
  Printf("enter the no of process\n");
  Scanf("%d,&n);
Struct Process p[n];
printf(" Enter the Arrival Time, Burst Time, priority
for each process:\n");
  for (int i=0; i<n; i++){
      p[i].pid=i+1;
  Printf("process%d ",i+1);
Scanf("%d %d %d ",&p[i].at,&p[i].bt,&p[i].pri);
  p[i].isCompleted=0;
}
findPriorityScheduling(p,n);
  return 0;
}
```

O/P =>

Enter the no of process: 5

Enter Arrival Time, Burst Time and Priority for each process:

Process1 : 0 10 3

Process 2 : 0 1 1

Process 3: 0 2 4

Process 4: 0 1 5

Process5 : 0 5 2

PID	AT	BT	PR	CT	TAT	WT	RT
P_1	0	10	3	16	16	6	6
P_2	0	1	1	1	1	0	0
P_3	0	2	4	18	18	16	16
P_4	0	1	5	19	19	18	18
P_5	0	5	2	6	6	1	1

Ganttchart:

P_2	P_5	P_1	P_3	P_4

0 1 6 16 18 19

lower the no, highest the priority

→Priority(Pre-emptive)

```
#include <stdio.h>
#include <limits.h>
struct Process {
    int pid, at, bt, pr, ct, wt, tat, rt, remaining;
};
void findPreemptivePriorityScheduling(struct Process p[], int n) {
    int completed = 0, time = 0, min_idx = -1;
    float totalWT = 0, totalTAT = 0;
    for (int i = 0; i < n; i++) {
        p[i].remaining = p[i].bt;
        p[i].rt = -1;
    }
    while (completed != n) {
        int min_priority = INT_MAX;
        min_idx = -1;
        for (int i = 0; i < n; i++) {
            if (p[i].at <= time && p[i].remaining > 0 && p[i].pr < min_priority) {
                min_priority = p[i].pr;
                min_idx = i;
            }
        }
        if (min_idx == -1) {
            time++;
            continue;
        }
        if (p[min_idx].rt == -1) {
            p[min_idx].rt = time - p[min_idx].at;
        }
```

```
        p[min_idx].remaining--;
        time++;
        if (p[min_idx].remaining == 0) {
            completed++;
            p[min_idx].ct = time;
            p[min_idx].tat = p[min_idx].ct - p[min_idx].at;
            p[min_idx].wt = p[min_idx].tat - p[min_idx].bt;
            totalWT += p[min_idx].wt;
            totalTAT += p[min_idx].tat;
        }
    }
    printf("PID\tAT\tBT\tPR\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
                p[i].pid, p[i].at, p[i].bt, p[i].pr, p[i].ct, p[i].tat, p[i].wt, p[i].rt);
    }
    printf("\nAverage Turnaround Time: %.2f\n", totalTAT / n);
    printf("Average Waiting Time: %.2f\n", totalWT / n);
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process p[n];

    printf("Enter Arrival Time, Burst Time, and Priority for each process:\n");
    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
```

```
        printf("Process %d: ", i + 1);
        scanf("%d %d %d", &p[i].at, &p[i].bt, &p[i].pr);
    }

    findPreemptivePriorityScheduling(p, n);
    return 0;
}
```

Output

Clear

```
Enter the number of processes: 7
Enter Arrival Time, Burst Time, and Priority for each process:
Process 1: 0
8
3
Process 2: 1
2
4
Process 3: 3
4
4
Process 4: 4
1
5
Process 5: 5
6
2
Process 6: 6
5
6
Process 7: 7
1
1
PID AT  BT  PR  CT  TAT WT  RT
1   0   8   3   15  15  7   0
2   1   2   4   17  16  14  14
3   3   4   4   21  18  14  14
4   4   1   5   22  18  17  17
5   5   6   2   12  7   1   0
6   6   5   6   27  21  16  16
7   7   1   1   8   1   0   0

Average Turnaround Time: 13.71
Average Waiting Time: 9.86
```

Priority - Preemptive

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

    struct process {
    int pid, at, bt, pr, ct, wt, tat, rt, remaing;
};

void fendPreemptivePriorityScheduling (StructProcess p[],
int n){
  int completed =0; time =0;
  min_idx = -1;
  float totalWT = 0; totalTAT = 0;

    for(int i=0; i<n; i++){
        p[i].remaining = p[i].bt;
        p[i].rt = -1;
        ~~for (int i=0; i<n; i++){~~
        while (completed != n){
            int min_priority = Int_max;
            min_idx = -1;

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

  if(p[i].at <= time && p[i].remaining > && p[i].pr <
min_priority){
        min_priority = p[i].pr;
        min_idx = i;
    }}
```

```
if(min_idx == -1){
    time++;
    continue;
  }

if(p[min_idx].rt == -1){
  p[min_idx].rt = time - p[min_idx].at;
 }
p[min_idx].rt == -1){
 p[min_idx].rt = time - p[min_idx].at;
  }
 p[min_idx].remaing--;
  time++;
 if(p[min_idx].remaing == 0){
     Completed++;
   p[min_idx].ct = time;
   p[min_idx].tat = p[min_idx].ct - p[min_
   idx].at;
   p[min_idx].wt = p[min_idx].tat - p[min_idx]
   bt;
   totalWT += p[min_idx].wt;
   totalTAT += p[min_idx].tat;
   }}
Printf("PID\tAT\tBT\tPR\tCT\tTAT\tWT\t\n")
    for(int i = 0; i<n; i++){
      Printf("%d\t%d\t %d\t %d\t %d\t %d\t
```

```
%d\t%.\n)");
  p[i].Pid, p[i].at, p[i].bt, p[i].pr, p[i].ct
  p[i].tat, p[i].wt, p[i].rt);
}
Printf("\n Average Turnaround Time: %.2f\n", total
TAT/n);
Printf("\n Average Waiting Time: %.2f\n", totalWT/n);
}
```

O/P ->

Enter the no of Processor:- 7

Enter AT, BT and priority for each processor:-

PID	AT	BT	PR	CT	TAT	WT	RT
1	6	8	3	15	15	7	0
2	1	2	4	17	16	14	14
3	3	4	4	21	18	14	14
4	4	6	5	22	18	17	17
5	5	5	2	12	7	1	0
6	6	5	6	27	21	16	16
7	7	1	1	8	1	0	0

P_1	P_1	P_1	P_1	P_1	P_5	P_5	P_7	P_5	P_1	P_2	P_3	P_4	P_6

0 1 2 3 4 5 6 7 8 12 15 17 21 22 27

→Round Robin

```
#include <stdio.h>
#define MAX 100
void roundRobin(int n, int at[], int bt[], int quant) {
    int ct[n], tat[n], wt[n], rem_bt[n];
    int queue[MAX], front = 0, rear = 0;
    int time = 0, completed = 0, visited[n];
    for (int i = 0; i < n; i++) {
        rem_bt[i] = bt[i];
        visited[i] = 0;
    }
    queue[rear++] = 0;
    visited[0] = 1;
    while (completed < n) {
        int index = queue[front++];
        if (rem_bt[index] > quant) {
            time += quant;
            rem_bt[index] -= quant;
        } else {
            time += rem_bt[index];
            rem_bt[index] = 0;
            ct[index] = time;
            completed++;
        }

        for (int i = 0; i < n; i++) {
            if (at[i] <= time && rem_bt[i] > 0 && !visited[i]) {
                queue[rear++] = i;
```

```
            visited[i] = 1;
        }
    }
    if (rem_bt[index] > 0) {
        queue[rear++] = index;
    }
    if (front == rear) {
        for (int i = 0; i < n; i++) {
            if (rem_bt[i] > 0) {
                queue[rear++] = i;
                visited[i] = 1;
                break;
            }
        }
    }
}
float total_tat = 0, total_wt = 0;
printf("P#\tAT\tBT\tCT\tTAT\tWT\n");
for (int i = 0; i < n; i++) {
    tat[i] = ct[i] - at[i];
    wt[i] = tat[i] - bt[i];
    total_tat += tat[i];
    total_wt += wt[i];
    printf("%d\t%d\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i], ct[i], tat[i], wt[i]);
}
printf("Average TAT: %.2f\n", total_tat / n);
printf("Average WT: %.2f\n", total_wt / n);
}
int main() {
    int n, quant;
```

```
    printf("Enter number of processes: ");
    scanf("%d", &n);
    int at[n], bt[n];
    for (int i = 0; i < n; i++) {
        printf("Enter AT and BT for process %d: ", i + 1);
        scanf("%d %d", &at[i], &bt[i]);
    }
    printf("Enter time quantum: ");
    scanf("%d", &quant);
    roundRobin(n, at, bt, quant);
    return 0;
}
```

```
Enter number of processes: 5
Enter AT and BT for process 1: 0 8
Enter AT and BT for process 2: 5 2
Enter AT and BT for process 3: 1 7
Enter AT and BT for process 4: 6 3
Enter AT and BT for process 5: 8 5
Enter time quantum: 3
P#      AT      BT      CT      TAT     WT
1       0       8       22      22      14
2       5       2       11      6       4
3       1       7       23      22      15
4       6       3       14      8       5
5       8       5       25      17      12
Average TAT: 15.00
Average WT: 10.00
```

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>

#define MAX_PROCESSES 10

#define TIME_QUANTUM 2

typedef struct {

   int burst_time, arrival_time, queue_type, waiting_time, turnaround_time, response_time,
remaining_time;

} Process;

void round_robin(Process processes[], int n, int time_quantum, int *time) {

   int done, i;

   do {

      done = 1;

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

         if (processes[i].remaining_time > 0) {

            done = 0;

            if (processes[i].remaining_time > time_quantum) {

               *time += time_quantum;

               processes[i].remaining_time -= time_quantum;

            } else {

               *time += processes[i].remaining_time;

               processes[i].waiting_time = *time - processes[i].arrival_time -
processes[i].burst_time;

               processes[i].turnaround_time = *time - processes[i].arrival_time;

               processes[i].response_time = processes[i].waiting_time;
```

```
                processes[i].remaining_time = 0;
            }
        }
    }
} while (!done);
}
void fcfs(Process processes[], int n, int *time) {
    for (int i = 0; i < n; i++) {
        if (*time < processes[i].arrival_time) {
            *time = processes[i].arrival_time;
        }
        processes[i].waiting_time = *time - processes[i].arrival_time;
        processes[i].turnaround_time = processes[i].waiting_time + processes[i].burst_time;
        processes[i].response_time = processes[i].waiting_time;
        *time += processes[i].burst_time;
    }
}
int main() {
    Process processes[MAX_PROCESSES], system_queue[MAX_PROCESSES],
user_queue[MAX_PROCESSES];
    int n, sys_count = 0, user_count = 0, time = 0;
    float avg_waiting = 0, avg_turnaround = 0, avg_response = 0, throughput;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        printf("Enter Burst Time, Arrival Time and Queue of P%d: ", i + 1);
        scanf("%d %d %d", &processes[i].burst_time, &processes[i].arrival_time,
&processes[i].queue_type);
        processes[i].remaining_time = processes[i].burst_time;
```

```
        if (processes[i].queue_type == 1) {
            system_queue[sys_count++] = processes[i];
        } else {
            user_queue[user_count++] = processes[i];
        }
    }
    // Sort user processes by arrival time for FCFS
    for (int i = 0; i < user_count - 1; i++) {
        for (int j = 0; j < user_count - i - 1; j++) {
            if (user_queue[j].arrival_time > user_queue[j + 1].arrival_time) {
                Process temp = user_queue[j];
                user_queue[j] = user_queue[j + 1];
                user_queue[j + 1] = temp;
            }
        }
    }
    printf("\nQueue 1 is System Process\nQueue 2 is User Process\n");
    round_robin(system_queue, sys_count, TIME_QUANTUM, &time);
    fcfs(user_queue, user_count, &time);
    printf("\nProcess  Waiting Time  Turn Around Time  Response Time\n");

    for (int i = 0; i < sys_count; i++) {
        avg_waiting += system_queue[i].waiting_time;
        avg_turnaround += system_queue[i].turnaround_time;
        avg_response += system_queue[i].response_time;
        printf("%d        %d            %d              %d\n", i + 1, system_queue[i].waiting_time,
system_queue[i].turnaround_time, system_queue[i].response_time);
    }
```

```
    for (int i = 0; i < user_count; i++) {
        avg_waiting += user_queue[i].waiting_time;
        avg_turnaround += user_queue[i].turnaround_time;
        avg_response += user_queue[i].response_time;
        printf("%d        %d        %d        %d\n", i + 1 + sys_count,
user_queue[i].waiting_time, user_queue[i].turnaround_time, user_queue[i].response_time);
    }
    avg_waiting /= n;
    avg_turnaround /= n;
    avg_response /= n;
    throughput = (float)n / time;
    printf("\nAverage Waiting Time: %.2f", avg_waiting);
    printf("\nAverage Turn Around Time: %.2f", avg_turnaround);
    printf("\nAverage Response Time: %.2f", avg_response);
    printf("\nThroughput: %.2f", throughput);
    printf("\nProcess returned %d (0x%d) execution time: %.3f s\n", time, time, (float)time);
    return 0;
}
```

```
C:\Users\Admin\Desktop\Multilevel-queue-Scheduling.exe
Enter number of processes: 4
Enter Burst Time, Arrival Time and Queue of P1: 2 0 1
Enter Burst Time, Arrival Time and Queue of P2: 1 0 2
Enter Burst Time, Arrival Time and Queue of P3: 5 0 1
Enter Burst Time, Arrival Time and Queue of P4: 3 0 2

Queue 1 is System Process
Queue 2 is User Process

Process  Waiting Time  Turn Around Time  Response Time
1         0           2                 0
2         2           7                 2
3         7           8                 7
4         8           11                 8

Average Waiting Time: 4.25
Average Turn Around Time: 7.00
Average Response Time: 4.25
Throughput: 0.36
Process returned 11 (0x11) execution time: 11.000 s

Process returned 0 (0x0)   execution time : 41.000 s
Press any key to continue.
```

Multilevel-Queue Scheduling:-

```
#include <stdio.h>
#define max-process 10
#define Time-Quantum 2

typedef struct {
int bt, at, queue-type, WT, TAT, RT, RemainTime;
} Process;

void round-robin(Process process[], int n,
int time-quantum, int* time){
    int done, i;
  do {
      done=1;
      for (i=0; i<n; i++){
  if(processes[i].Remaining-time>0){
      done=0;
  if (processes[i].remaining-time > time-quantum)
  {
      *time += time-quantum
  Processes[i].remaing-time -= time-quantum;
  }
    else {
    *time += process[i].remaining-time;
```

```
Procus[i].WT = *time - procus[i].AT - procus[i].
BT;
Procus[i].TAT = *time - procus[i].AT;
Processes[i].RT = processes[i].WT;
Processes[i].remaining_time = 0;
}
}
}
}while(!done);
}
}
void fcfs(Procus processes[], int n, int *time){
for(int i=0; i<n; i++){
if(*time < processes[i].AT){
*time = processes[i].AT;
}
Processes[i].WT = *time - processes[i].AT;
Processes[i].TAT = processes[i].WT + processes[i].bT;
Processes[i].RT = Processes[i].Waiting_time;
*time += processes[i].bT;
}
}
int main(){
Procus processes[max_processes], System_queue[max_
Processes], user_queue[max_processes];
int n, sys_count=0, usr_count=0; time=0;
float avg_waiting=0; avg_turnaround=0, avg_response=
```

```
printf("\n Throughput : %.2f", throughput);
printf("\n process returned %d (0x%d) execution
time : %.3f s\n", time, time, (float)(time));

    return 0;
}
```

O/P ->

Enter the no of process : 4

Enter the BT, AT and Quan of P1 : 2 0 1

Enter the BT, AT and Quan of P2 : 1 0 2

Enter the BT, AT and Quan of P3 : 5 0 1

Enter the BT, AT and Quan of P4 : 3 0 2

Quan 1 is System process

Quan 2 is Our process

Process	Waiting Time	Turnaround Time	Response Time
1	0	2	0
2	2	7	2
3	7	8	7
4	8	11	8

Average waiting time : 4.25

Average Turnaround Time : 7.00

Average Response Time : 4.25

Throughput : 0.36

Process returned 11 (0x11) execution time : 11.0005

Program 4:

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

→Rate- Monotonic

```
#include <stdio.h>
#include <math.h>
typedef struct {
   int id, burst, period;
} Task;
int gcd(int a, int b) {
   return (b == 0) ? a : gcd(b, a % b);
}
int lcm(int a, int b) {
   return (a * b) / gcd(a, b);
}
int findLCM(Task tasks[], int n) {
   int result = tasks[0].period;
   for (int i = 1; i < n; i++)
      result = lcm(result, tasks[i].period);
   return result;
}
void rateMonotonic(Task tasks[], int n) {
   float utilization = 0;
   printf("\nRate Monotonic Scheduling:\nPID\tBurst\tPeriod\n");
   for (int i = 0; i < n; i++) {
      printf("%d\t%d\t%d\n", tasks[i].id, tasks[i].burst, tasks[i].period);
      utilization += (float)tasks[i].burst / tasks[i].period;
   }
```

```
    float bound = n * (pow(2, (1.0 / n)) - 1);
    printf("\nUtilization: %.6f, Bound: %.6f\n", utilization, bound);
    if (utilization <= bound)
        printf("Tasks are Schedulable\n");
    else
        printf("Tasks are NOT Schedulable\n");
}
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    Task tasks[n];
    printf("Enter the CPU burst times: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &tasks[i].burst);
    printf("Enter the time periods: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &tasks[i].period);
        tasks[i].id = i + 1;
    }
    rateMonotonic(tasks, n);
    return 0;
}
```

Output

```
Enter the number of processes: 3
Enter the CPU burst times: 3 6 8
Enter the time periods: 3 4 5

Rate Monotonic Scheduling:
PID Burst   Period
1   3   3
2   6   4
3   8   5

Utilization: 4.100000, Bound: 0.779763
Tasks are NOT Schedulable
```

Rate-Monotonic

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

typedef struct{
    int id, burst, period;
} Task;

int gcd(int a, int b){
    return (b==0)? a: gcd(b, a%b);
}
int lcm(int a, int b){
    return (a*b)/gcd(a,b);
}
int findLCM(Task tasks[], int n){
    int result = tasks[0].period;
    for(int i=1; i<n; i++)
        result = lcm(result, tasks[i].period);
    return result;
}
void rateMonotonic(task tasks[], int n){
    float utilization = 0;
    Printf("\n Rate Monotonic Scheduling:\nPID\t Burst
\t period\n");
    for(int i=0; i<n; i++){
        Printf("%d\t%d\t%d\n", tasks[i].id,
tasks[i].burst, tasks[i].period);
```

```
utilization += (float) tasks[i].burst / tasks[i].period;
}
float bound = n * (pow(2, (1.0/n)) - 1);
Printf("\n utilization: %.6f, Bound: %.6f \n", utilization,
bound);
    if (utilization <= bound)
        Printf("tasks are Schedulable \n");
    else
        Printf("Tasks are NOT Schedulable \n");
}

int main(){
    int n;
    Printf("Enter the no of processes: ");
    Scanf("%d", &n);

    Task tasks[n];
    Printf("Enter the CPU Burst times: ");
    for (int i=0; i<n; i++)
        Scanf("%d", &tasks[i].burst);

    Printf("Enter the time periods: ");
    for (int i=0; i<n; i++){
        Scanf("%d", &tasks[i].period);
        tasks[i].id = i+1;
    }
```

```
rateMonotonic(tasks, n);
return 0;
}
```

O/P →

Enter the number of Processes: 3

Enter the CPU burst times: 3 6 8

Enter the time periods: 3 4 5

Rate monotonic Scheduling:

PID	Burst	Period
1	3	3
2	6	4
3	8	5

utilization: 4.100000, Bound: 0.779763

Tasks are not Schedulable

→Earliast-Deadline First

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    int pid;
    int burst_time;
    int deadline;
    int period;
    int remaining_time;
} Process;
// Function to compare processes based on their deadlines
int compare_deadlines(const void *a, const void *b) {
    return ((Process *)a)->deadline - ((Process *)b)->deadline;
}
int main() {
    int num_processes;
    printf("Enter the number of processes: ");
    scanf("%d", &num_processes);
    Process processes[num_processes];
    printf("Enter the CPU burst times:\n");
    for (int i = 0; i < num_processes; i++) {
        scanf("%d", &processes[i].burst_time);
        processes[i].pid = i + 1;
        processes[i].remaining_time = processes[i].burst_time;
    }
    printf("Enter the deadlines:\n");
    for (int i = 0; i < num_processes; i++) {
        scanf("%d", &processes[i].deadline);
```

```
    }
    printf("Enter the time periods:\n");
    for (int i = 0; i < num_processes; i++) {
        scanf("%d", &processes[i].period);
    }
    // Sort processes based on deadlines
    qsort(processes, num_processes, sizeof(Process), compare_deadlines);
    printf("\nEarliest Deadline Scheduling:\n");
    printf("PID\tBurst\tDeadline\tPeriod\n");
    for (int i = 0; i < num_processes; i++) {
        printf("%d\t%d\t%d\t\t%d\n", processes[i].pid, processes[i].burst_time,
processes[i].deadline, processes[i].period);
    }
    int current_time = 0;
    int completed_processes = 0;
    printf("\nScheduling occurs for %d ms\n", processes[0].deadline); // Assuming the first
process has the earliest deadline
    while (completed_processes < num_processes) {
        for (int i = 0; i < num_processes; i++) {
            if (processes[i].remaining_time > 0) {
                printf("%dms : Task %d is running.\n", current_time, processes[i].pid);
                processes[i].remaining_time--;
                current_time++;
                if (processes[i].remaining_time == 0) {
                    completed_processes++;
                } } } }
 printf("\nProcess returned %d (0x%X)\texecution time : %.3f s\n", current_time, current_time,
(float)current_time / 1000.0);
    return 0;
}
```

```
C:\Users\Admin\Downloads\early.exe
Enter the number of processes: 3
Enter the CPU burst times:
2 3 4
Enter the deadlines:
1 2 3
Enter the time periods:
1 2 3

Earliest Deadline Scheduling:
PID     Burst   Deadline        Period
1       2       1               1
2       3       2               2
3       4       3               3

Scheduling occurs for 1 ms
0ms : Task 1 is running.
1ms : Task 2 is running.
2ms : Task 3 is running.
3ms : Task 1 is running.
4ms : Task 2 is running.
5ms : Task 3 is running.
6ms : Task 2 is running.
7ms : Task 3 is running.
8ms : Task 3 is running.

Process returned 9 (0x9)        execution time : 0.009 s

Process returned 0 (0x0)   execution time : 15.281 s
Press any key to continue.
```

Earliest - deadline first :-

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

    typedef struct {
        int pid;
        int burst_Time;
        int deadline;
        int period;
        int remaining time;
    } Process;

int Compare_deadlines (const void *a, const void *b) {
    return ((Process *)a) -> deadline - ((process *)b) -> deadline;
}
    int main () {
        int num_processes;
        Printf("Enter the no of processes : ");
        Scanf ("%d", &num_processes);

    Process Processes [num_processes];

    Printf("Enter the CPU burst times : \n");
    for (int i=0; i<num_processes; i++) {
        Scanf("%d", &processes[i].burst_Time);
```

```
Processes[i].pid = i+1;
Processes[i].remaining_time = processes[i].burst_time;
}
Printf("Enter the deadline:\n");
for (int i=0; i<num_processes; i++){
        Scanf("%d", &processes[i].deadline);
}
Printf("Enter the time period:\n");
for (int i=0; i<num_processes; i++){
        Scanf("%d", &processes[i].period);

qSort(processes, num_processes, sizeof(Process), compare_
deadlines);
Printf("\n Earliest Deadline Scheduling:\n");
Printf("PID\t Burst\t Deadline\t period\n");
for (int i=0; i<num_processes; i++){
Printf("%d\t%d\t%d\t%d\n", Processes[i].pid,
Processes[i].burst_Time; Processes[i].deadline, processes[i]
.period);
}
int current_time = 0;
int completed_processes = 0;
Printf("\n Scheduling occurs for %d ms\n",
Processes[0].deadline);
while (completed_processes < num_processes){
for (int i=0; i<num_processes; i++){
```

```
if(process[i].remaining_time>0){
printf("%d ms: Task %d is running.\n",
current_time, process[i].pid);
Process[i].remaining_time--;
current_time++;
    if (process[i].remaining_time==0) {
        Completed_process++;
    }
  }
 }
}
Printf("\n process returned %d (0x%x) Execution time
3f s\n", current_time, current_time, (float) current_time/
1000.0);
return 0;
}
```

O/P ->

Enter the no of Processes : 3

Enter the CPU burst times : 2 3 4

Enter the deadlines : 1 2 3

Enter the time periods : 1 2 3

Earliest Deadline scheduling

PID	Burst	Deadline	Period
1	2	1	1
2	3	2	2
3	4	3	3

Scheduling occurs for 1ms:

0ms : Task 1 is running.
1ms : Task 2 is running.
2ms : Task 3 is running.
3ms : Task 1 is running.
4ms : Task 2 is running.
5ms : Task 3 is running.
6ms : Task 2 is running.
7ms : Task 3 is running.
8ms : Task 3 is running.

Process returned 9(0 x 9) execution time : 0.0095 s

Program 5:

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

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
int mutex = 1, full = 0, empty, x = 0;
int *buffer, buffer_size;
int in = 0, out = 0;
int wait(int s) {
   return (--s);
}
int signal(int s) {
   return (++s);
}
void producer(int id) {
   if ((mutex == 1) && (empty != 0)) {
      mutex = wait(mutex);
      full = signal(full);
      empty = wait(empty);
      int item = rand() % 50;
      buffer[in] = item;
      x++;
      printf("Producer %d produced %d\n", id, item);
      printf("Buffer:%d\n", item);
      in = (in + 1) % buffer_size;
      mutex = signal(mutex);
   } else {
      printf("Buffer is full\n");
```

```
    }
}
void consumer(int id) {
    if ((mutex == 1) && (full != 0)) {
        mutex = wait(mutex);
        full = wait(full);
        empty = signal(empty);
        int item = buffer[out];
        printf("Consumer %d consumed %d\n", id, item);
        x--;
        printf("Current buffer len: %d\n", x);
        out = (out + 1) % buffer_size;
        mutex = signal(mutex);
    } else {
        printf("Buffer is empty\n");
    }
}
int main() {
    int producers, consumers;
    printf("Enter the number of Producers:");
    scanf("%d", &producers);
    printf("Enter the number of Consumers:");
    scanf("%d", &consumers);
    printf("Enter buffer capacity:");
    scanf("%d", &buffer_size);
    buffer = (int *)malloc(sizeof(int) * buffer_size);
    empty = buffer_size;
    for (int i = 1; i <= producers; i++)
        printf("Successfully created producer %d\n", i);
    for (int i = 1; i <= consumers; i++)
```

```
        printf("Successfully created consumer %d\n", i);
    srand(time(NULL));
    int iterations = 10;
    for (int i = 0; i < iterations; i++) {
        producer(1);
        sleep(1);
        consumer(2);
        sleep(1);
    }
    free(buffer);
    return 0;
}
```

```
C:\Users\aDMIN\Documents\vicky042\Poducer-Cnsumer.exe
Enter the number of Producers:1
Enter the number of Consumers:1
Enter buffer capacity:1
Successfully created producer 1
Successfully created consumer 1
Producer 1 produced 28
Buffer:28
Consumer 2 consumed 28
Current buffer len: 0
Producer 1 produced 42
Buffer:42
Consumer 2 consumed 42
Current buffer len: 0
Producer 1 produced 7
Buffer:7
Consumer 2 consumed 7
Current buffer len: 0
Producer 1 produced 8
Buffer:8
Consumer 2 consumed 8
Current buffer len: 0
Producer 1 produced 26
Buffer:26
Consumer 2 consumed 26
Current buffer len: 0
Producer 1 produced 32
Buffer:32
Consumer 2 consumed 32
Current buffer len: 0
Producer 1 produced 4
Buffer:4
Consumer 2 consumed 4
Current buffer len: 0
Producer 1 produced 46
Buffer:46
Consumer 2 consumed 46
Current buffer len: 0
Producer 1 produced 10
Buffer:10
Consumer 2 consumed 10
Current buffer len: 0
Producer 1 produced 37
Buffer:37
Consumer 2 consumed 37
Current buffer len: 0

Process returned 0 (0x0)   execution time : 25.678 s
Press any key to continue.
```

Producer - Consumer Problem using Semaphores;

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

int mutex=1, full=0, empty, x=0;
int *buffer, buffer_size;
int in=0; out=0;

int wait(int s){
        return(--s);
    }

int signal(int s){
    return(++s);
    }

void producer(int id){
if((mutex==1) && (empty!=0)){
        mutex = wait(mutex);
        full = signal(full);
        empty = wait(empty);
        int item = rand()%50;
        buffer[in] = item;
        x++;
    printf("Producer %d produced %d\n", id, item);
```

```
Printf ("Buffer : %d\n", item);
in = (in+1) % buffersize;
mutex = Signal (mutex);
} else {
    Printf("Buffer is full\n");
  }
}

void consumer (int id) {
    if ((mutex == -1) && (ffull != 0)) {
      mutex = wait (mutex);
      full = wait (full);
      empty = Signal (empty);
      int item = buffer [out];
printf(" Consumer %d Consumed %d\n", id, item);
    x--;
  Printf (" current buffer len: %d\n", x);
    out = (out+1) % buffer_size;
    mutex = Signal (mutex);
  } else {
      Printf("Buffer is empty \n");
    }
  }

int main () {
    int producers, consumers;
  Printf("Enter the no of Producers");
  Scanf ("%d", & producers);
```

```
Printf("Enter the no of Consumers:");
Scanf("%d", Consumers);
Printf("Enter buffer Capacity:");
Scanf("%d", &buffer_size);

buffer=(int*) malloc(sizeof(int)*buffer_size);
empty=buffer_size;
for(int i=1; i<=producers; i++)
    Printf("~~Suff~~ Successfully Created producers %d\n",
    i);
for (int i=1; i<=Consumers; i++)
    Printf("Successfully Created Consumers %d\n", i);
Srand(time(Null));

    int iterations=10;
    for (int i=0; i<iterations; i++){
        Producer(i);
        Sleep(1);
        Consumer(2);
        Sleep(1);
    }
    free(buffer);
    return 0;
}
```

O/P →

Enter the number of Producers: 1

Enter the number of Consumers: 1

Enter buffer capacity: 1

Successfully created producer 1

Successfully created consumer 1

Producer 1 produced 30

Buffer: 30

Consumer 2 consumed 30

Current buffer size: 0

Program 6:

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

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.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) {
      // Eating state
      state[phnum] = EATING;
      printf("Philosopher %d takes chopstick %d and %d\n", phnum + 1, LEFT + 1, phnum +
1);
      printf("Philosopher %d is Eating\n", phnum + 1);
      sem_post(&S[phnum]);
   }
}
```

```
void take_fork(int phnum)
{
   sem_wait(&mutex);
   state[phnum] = HUNGRY;
   printf("Philosopher %d is Hungry\n", phnum + 1);
   test(phnum);
   sem_post(&mutex);
   sem_wait(&S[phnum]);
   sleep(1);
}
void put_fork(int phnum)
{
   sem_wait(&mutex);
   state[phnum] = THINKING;
   printf("Philosopher %d putting chopstick %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(1);
      put_fork(*i);
```

```
    }
}
int main()
{
    int i;
    pthread_t thread_id[N];
    sem_init(&mutex, 0, 1);
    for (i = 0; i < N; i++)
        sem_init(&S[i], 0, 0);
    for (i = 0; i < N; i++) {
        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);

    return 0;
}
```

```
C:\Users\aDMIN\Documents\vicky042\Dining-philosophers-problem.exe
Philosopher 5 is thinking
Philosopher 1 takes chopstick 5 and 1
Philosopher 1 is Eating
Philosopher 3 putting chopstick 2 and 3 down
Philosopher 3 is thinking
Philosopher 4 takes chopstick 3 and 4
Philosopher 4 is Eating
Philosopher 3 is Hungry
Philosopher 5 is Hungry
Philosopher 1 putting chopstick 5 and 1 down
Philosopher 1 is thinking
Philosopher 2 takes chopstick 1 and 2
Philosopher 2 is Eating
Philosopher 4 putting chopstick 3 and 4 down
Philosopher 4 is thinking
Philosopher 5 takes chopstick 4 and 5
Philosopher 5 is Eating
Philosopher 1 is Hungry
Philosopher 4 is Hungry
Philosopher 5 putting chopstick 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes chopstick 3 and 4
Philosopher 4 is Eating
Philosopher 2 putting chopstick 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes chopstick 5 and 1
Philosopher 1 is Eating
Philosopher 2 is Hungry
Philosopher 5 is Hungry
Philosopher 1 putting chopstick 5 and 1 down
Philosopher 1 is thinking
Philosopher 2 takes chopstick 1 and 2
Philosopher 2 is Eating
Philosopher 4 putting chopstick 3 and 4 down
Philosopher 4 is thinking
Philosopher 5 takes chopstick 4 and 5
Philosopher 5 is Eating
Philosopher 1 is Hungry
Philosopher 4 is Hungry
Philosopher 5 putting chopstick 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes chopstick 3 and 4
Philosopher 4 is Eating
Philosopher 2 putting chopstick 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes chopstick 5 and 1
Philosopher 1 is Eating
Philosopher 2 is Hungry
Philosopher 5 is Hungry
Philosopher 1 putting chopstick 5 and 1 down
Philosopher 1 is thinking
Philosopher 2 takes chopstick 1 and 2
Philosopher 2 is Eating
Philosopher 4 putting chopstick 3 and 4 down
Philosopher 4 is thinking
Philosopher 5 takes chopstick 4 and 5
Philosopher 5 is Eating
Philosopher 1 is Hungry
Philosopher 4 is Hungry
```

Dining - Philosophers

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>

#define N5
#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[phnum]= Eating;

Printf("Philosophers %d takes chopstick %d and
```

```
%d and %d \n", phnum+1, left+1, phnum+1);
Printf(" philosophas %d is eating \n", phnum+1);

  Sem-post(&S[phnum]);
  }
}

  Void take-fork(int phnum)
{
   Sem_wait(&mutex);
   State[phnum] = Hungry;
   Printf("philosophas %d is hungry \n", phnum+1);
   test(phnum);
   Sem_post(&mutex);
   Sem-wait(&S[phnum]);
   Sleep(1);
  }
  Void put_fork(int phnum){
    Sem_wait(&mutex);
    State[phnum] = Thinking;
Printf(" philosopher %d putting chopstick %d and %d
down \n", phnum+1, left+1, phnum+1);
Printf(" Philosophr %d is thinking, phnum+1);
   test(left);
   test(Right);
     Sem-post(&mutex);
  }
```

```
Void * philosopher (void* num)
{
    while (1) {
        int * i = num;

        Sleep(1);
        take_fork (*i);
        Sleep(1);
        Put_fork (*i);
    }
}

int main ()
{
    int i;
    pthread_t thread_id [N];
    Sem_init(&mutex,0,1);

        for (i=0; i<N; i++)
            Sem_init(&S[i],0,0);

        for (i=0; i<N; i++){
        Pthread_Create(&thread_id[i], Null, Philosopher,
&Phil[i]);
    }
    for (i=0; i<N; i++)
        Pthread_join (thread_id[i], Null);
        return0;
    }
```

O/P ->

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 5 is Hungry
Philosopher 1 is Hungry
Philosopher 3 is Hungry
Philosopher 2 is Hungry
Philosopher 2 takes chopstick 1 and 2
Philosopher 2 is Eating
Philosophr 2 is thinking
Philosopher 2 ~~takes~~ Putting chopstick 1 and 2
Philosopher 1 takes chopstick 5 and 1
Philosopher 1 is Eating
Philosopher 3 takes chopstick 2 and 3
Philosopher 3 is Eating.

Program 7:

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

```
#include <stdio.h>
int main() {
    int n, m, i, j, k;
    printf("Enter number of processes and resources:\n");
    scanf("%d %d", &n, &m);
    int alloc[n][m], max[n][m], avail[m];
    int need[n][m], f[n], ans[n], ind = 0;
    printf("Enter allocation matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            scanf("%d", &alloc[i][j]);
    printf("Enter max matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            scanf("%d", &max[i][j]);
    printf("Enter available matrix:\n");
    for (i = 0; i < m; i++)
        scanf("%d", &avail[i]);
    for (i = 0; i < n; i++)
        f[i] = 0;
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];
    for (k = 0; k < n; k++) {
        for (i = 0; i < n; i++) {
            if (f[i] == 0) {
                int flag = 0;
```

```
            for (j = 0; j < m; j++) {
                if (need[i][j] > avail[j]) {
                    flag = 1;
                    break;
                }
            }
            if (flag == 0) {
                ans[ind++] = i;
                for (j = 0; j < m; j++)
                    avail[j] += alloc[i][j];
                f[i] = 1;
            }
        }
    }
}

int safe = 1;
for (i = 0; i < n; i++)
    if (f[i] == 0)
        safe = 0;
if (safe) {
    printf("System is in safe state.\nSafe sequence is: ");
    for (i = 0; i < n - 1; i++)
        printf("P%d -> ", ans[i]);
    printf("P%d\n", ans[n - 1]);
} else {
    printf("System is not in safe state\n");
} return 0;
}
```

```
Enter number of processes and resources:
5 3
Enter allocation matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter max matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter available matrix:
3 3 2
System is in safe state.
Safe sequence is: P1 -> P3 -> P4 -> P0 -> P2

Process returned 0 (0x0)   execution time : 47.859 s
Press any key to continue.
```

Banker's Algorithm for the purpose of deadlock avoidance.

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

int Condition (int *need, int *work, int i, int m)
{
    for (int j = 0; j < m; j++)
    {
        if (need[i][j] > work[j])
            return 0;
    }
    return 1;
}

int Safety (int m, int n, int **allocated, int **max,
int *available, int *sequence)
{
    int **need = (int**) malloc(n* sizeof (int *));
    for (int i = 0; i < n; i++)
    {
    need[i] = (int **) malloc( m * sizeof (int));
    for (int j = 0; j < m; j++)
    {
    need[i][j] = max[i][j] - allocated [i][j];
    }
}
```

```
int *work = (int*) malloc (m * sizeof(int));
 for(int i=0; i<m; i++)
 {
    work[i] = available[i];
 }
int *finish = (int*) malloc (n * sizeof(int));
 for (int i=0; i<n; i++)
  { finish[i] = 0;
 }
 int SafeIndex = 0;
 int changed;
 do {
    changed = 0;
    for(int i = 0; i<n; i++)
    {
    if (!finish[i] && condition(need, work, i, m))
    {
      for(int j = 0; j<m; j++)
      {
        work[j] += allocated[i][j];
      }
    finish[i] = 1;
    Sequence[SafeIndex++] = i;
    changed = 1;
    }
 }
} while (changed);
```

```
for (int i=0; i<n; i++)
{
  if (! finished[i])
  {
     return0;
     }
  }
  return 1;
}

int main()
{
  int n=5, m=3;
  Printf ("Enter Allocation matrix:\n");
int ** allocated = (int**) malloc (n * sizeof (int *));
  for (int i=0; i<n; i++)
  {
allocated[i] = (int *) malloc(m * sizeof (int));
     for (int j=0; j<m; j++)
       {
       Scanf ("%d", &allocated[i][j]);
       }
  }
  Printf ("Enter max matrix:\n");
int ** max = (int * *) malloc (n * sizeof(int *));
  for ( int i=0; i < n; i++)
{  max [i] = (int *) malloc(m * sizeof (int));
     for ( j=0; j<m; j++)
```

```
Scanf ("%d", &max[i][j]);
}
}
Printf (" Enter Available matrix:\n");
int *available = (int *) malloc (m * sizeof (int));
for (int i=0; i<m; i++)
{
Scanf ("%d", &available [i]);
}
int *Sequence= (int *) malloc (n * sizeof (int));
int safe= safety (m, n, allocated, max, available, Sequence);
if (safe)
{
Printf("System is in a safe state. In safe sequence");
for (int i=0; i< n; i++)
{
Printf(" P%d \t", Sequence[i]);
}
Printf("\n");
}
else
{
Printf(" System is not in a safe state.\n")
}
return 0;
}
```

O/P ->

Enter Allocation matrix:-

0	1	0
2	0	0
3	0	2
2	1	1
0	0	2

Enter max matrix:

7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

Enter Available matrix:

3 3 2

System is in safe state.

Safe Sequence: $P_1 \rightarrow P_3 \rightarrow P_4 \rightarrow P_0 \rightarrow P_2$

Program 8:

Write a C program to simulate deadlock detection

```
#include <stdio.h>
int main() {
    int n, m, i, j, k;
    printf("Enter number of processes and resources:\n");
    scanf("%d %d", &n, &m);
    int alloc[n][m], req[n][m], avail[m], finish[n];
    printf("Enter allocation matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            scanf("%d", &alloc[i][j]);
    printf("Enter request matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            scanf("%d", &req[i][j]);

    printf("Enter available matrix:\n");
    for (i = 0; i < m; i++)
        scanf("%d", &avail[i]);

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

    int done;
    do {
        done = 0;
        for (i = 0; i < n; i++) {
            if (finish[i] == 0) {
```

```
            int canFinish = 1;
            for (j = 0; j < m; j++) {
                if (req[i][j] > avail[j]) {
                    canFinish = 0;
                    break;
                }
            }
            if (canFinish) {
                for (j = 0; j < m; j++)
                    avail[j] += alloc[i][j];
                finish[i] = 1;
                done = 1;
                printf("Process %d can finish.\n", i);
            }
        }
    }
} while (done);

int deadlock = 0;
for (i = 0; i < n; i++)
    if (finish[i] == 0)
        deadlock = 1;

if (deadlock)
    printf("System is in a deadlock state.\n");
else
    printf("System is not in a deadlock state.\n");

return 0;
}
```

```
Enter number of processes and resources:
5 3
Enter allocation matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter request matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter available matrix:
3 3 2
Process 1 can finish.
Process 3 can finish.
Process 4 can finish.
System is in a deadlock state.

Process returned 0 (0x0)   execution time : 47.372 s
Press any key to continue.
```

Deadlock Detection:-

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

#define P 5
#define R 3

void deadlockDetection (int allocation[P][R], int request
[R][R], int available[R]){
    int work[R];
    bool finish[P] = {false};
    bool deadlock = false;

    for (int i=0; i<R; i++){
        ~~work~~ work[i] = available[i];
    }
    int count = 0;
    while (count<P){
        bool found = false;
        for (int i=0; i<P; i++){
            if (!finish[i]){
                bool canProceed = true;
                for (int j=0; j<R; j++){
                    if (request[i][j] > work[j]){
                        canProceed = false;
                        break;
                    }
                }
```

```
if (canProceed){
    for (int j = 0; j<R; j++){
        work[j] += allocation[i][j];
    }
    finish[i] = true;
    Printf("Process %d is finished.\n", i);
    found = true;
    count++;
}
}

if (!found){
    break;
}

for (int i = 0; i<P; i++){
    if (!finished[i]){
        deadlock = true;
    Printf("Process %d is in deadlock.\n", i);
}
}
if (!deadlock){
    Printf("No deadlock detected. All processes can
complete.\n");
} else {
    Printf("Deadlock detected.\n");

}
}
```

```
int main () {
    int allocation[R][R] = {
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 2},
        {2, 1, 1},
        {0, 0, 2}
    };
    int request[R][R] = {
        {0, 0, 0},
        {2, 0, 2},
        {0, 0, 0},
        {1, 0, 0},
        {0, 0, 2},
    };
    int available[R] = {0, 0, 0};
    deadlockDetection (allocation, request, available);
    return 0;
}
```

O/P →
Process P_0 is finished.
Process P_2 is finished.
Process P_3 is finished.
Process P_4 is finished.
Process P_1 is finished.

No deadlock detected, All processes can complete.

17/4

Program 9:

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

a) Worst Fit
b) Best Fit
c) First Fit

→Worst Fit

```
#include <stdio.h>
struct Block {
   int block_no;
   int block_size;
   int is_free;
};
struct File {
   int file_no;
   int file_size;
};

void worstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
   printf("\nMemory 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%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);
    } else {
        printf("%d\t%d\t\tNot Allocated\n", files[i].file_no, files[i].file_size);
    }
  }
}

int main() {
  int n_blocks, n_files;

  printf("Enter the number of blocks: ");
  scanf("%d", &n_blocks);

  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;
  }
```

```
    printf("Enter the number of files: ");
    scanf("%d", &n_files);

    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);
    }

    worstFit(blocks, n_blocks, files, n_files);

    return 0;
}
```

```
Enter the number of blocks: 5
Enter the size of block 1: 100
Enter the size of block 2: 500
Enter the size of block 3: 200
Enter the size of block 4: 300
Enter the size of block 5: 600
Enter the number of files: 4
Enter the size of file 1: 212
Enter the size of file 2: 417
Enter the size of file 3: 112
Enter the size of file 4: 426

Memory Management Scheme - Worst Fit
File_no File_size      Block_no       Block_size     Fragment
1       212            5              600            388
2       417            2              500            83
3       112            4              300            188
4       426            Not Allocated
```

→Best Fit

```
#include <stdio.h>
struct Block {
   int block_no;
   int block_size;
   int is_free;
};
struct File {
   int file_no;
   int file_size;
};
void bestFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
   printf("\nMemory 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; // Large initial value

      for (int j = 0; j < n_blocks; j++) {
         if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
            int fragment = blocks[j].block_size - files[i].file_size;
            if (fragment < min_fragment) {
               min_fragment = fragment;
               best_fit_block = j;
            }
         }
      }

      if (best_fit_block != -1) {
         blocks[best_fit_block].is_free = 0;
```

```
            printf("%d\t%d\t\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);
        } else {
            printf("%d\t%d\t\tNot Allocated\n", files[i].file_no, files[i].file_size);
        }
    }
}

int main() {
    int n_blocks, n_files;

    printf("Enter the number of blocks: ");
    scanf("%d", &n_blocks);

    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;
    }

    printf("Enter the number of files: ");
    scanf("%d", &n_files);

    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);
    }

    bestFit(blocks, n_blocks, files, n_files);

    return 0;
}
```

```
Enter the number of blocks: 5
Enter the size of block 1: 100
Enter the size of block 2: 500
Enter the size of block 3: 200
Enter the size of block 4: 300
Enter the size of block 5: 600
Enter the number of files: 4
Enter the size of file 1: 212
Enter the size of file 2: 417
Enter the size of file 3: 113
Enter the size of file 4: 426

Memory Management Scheme - Best Fit
File_no File_size       Block_no        Block_size      Fragment
1       212             4               300             88
2       417             2               500             83
3       113             3               200             87
4       426             5               600             174
```

→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("\nMemory Management Scheme - First Fit\n");
   printf("File_no\tFile_size\tBlock_no\tBlock_size\tFragment\n");
   for (int i = 0; i < n_files; i++) {
      int allocated = 0;
      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;
            blocks[j].is_free = 0;
            printf("%d\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,
                  fragment);
            allocated = 1;
            break;
         }
```

```
        }

        if (!allocated) {
            printf("%d\t%d\t\tNot Allocated\n", files[i].file_no, files[i].file_size);
        }
    }
}

int main() {
    int n_blocks, n_files;

    printf("Enter the number of blocks: ");
    scanf("%d", &n_blocks);

    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;
    }

    printf("Enter the number of files: ");
    scanf("%d", &n_files);

    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);

    return 0;
}
```

```
Enter the number of blocks: 5
Enter the size of block 1: 100
Enter the size of block 2: 500
Enter the size of block 3: 200
Enter the size of block 4: 300
Enter the size of block 5: 600
Enter the number of files: 4
Enter the size of file 1: 212
Enter the size of file 2: 417
Enter the size of file 3: 112
Enter the size of file 4: 426

Memory Management Scheme - First Fit
File_no File_size       Block_no        Block_size      Fragment
1       212             2               500             288
2       417             5               600             183
3       112             3               200             88
4       426             Not Allocated
```

Contigous Memory Allocation:-

```
#include <stdio.h>

struct Block {
        int block_no;
        int block_size;
        int is_free;
    };

struct file {
        int file_no;
        int file_size;
    };

void resultBlocks(struct Block blocks[], int n) {
        for (int i=0; i<n; i++) {
            blocks[i].is_free=1;
        }
    }

void firstfit (struct Block blocks[], int n_blocks,
struct File files[], int n_files) {
        Printf ("\n Memory managint scheme - firstfit");
    Printf ("file no :\t File_size : \t Block no :\t
Block_size :\n");

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

```
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\n", files[i].
file_no, files[i].file_size, block[j].block_no, blocks[j].
block_size);
  allocated = 1;
  break;
}
  if(!allocated)
    Printf("%d\t\t%d\t\t%d\t\t%d\n", files[i].
file_no, files[i].file_size);
}

Void bestfit(Struct Block block[ ], int n_blocks,
Struct file files[ ], int n_files) {
    Printf("\nMemory management Scheme - Bestfit\n");
Printf("file_no:\t file_size:\tBlock_no:\tBlock_size:\n");
  for (int i = 0; i < n_files; i++) {
    int best = -1;
  for (int j = 0; j < n_blocks; j++) {
    if (blocks[j].is_free && blocks[j].block_size >=
files[i].file_size) {
```

```
if (best == -1 || blocks[j].block_size < block[best].
block_size)
        best = j;
    }
    if (best != -1) {
        blocks[best].is_free = 0;
Printf("%d\t\t%d\t\t%d\t\t%d\n", files[i].file_no,
files[i].file_size, block[best].block_no, blocks[best].
block_size);
    }
    else {
Printf("%d\t\t%d\t\t-\t\t-n", files[i].file_no,
files[i].file_size);
    }
}

Void worstfit(Struct Block blocks[], int n_blocks, Struct
File files[], int n_files) {
Printf("\nmemory managment Scheme - worst fit\n");
Printf("File_no:\t File_size:\t Block_no:\t Block_size:\n");

    for (int i=0; i<n_files; i++) {
        int worst = -1;
        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\n", files[i].file_no
files[i].file_size, blocks[j].block_no, blocks[j].block_size)
  allocated=1;
  break;
}
}
 if(!allocated)
     Printf("%d\t\t%d\t\t%d\t\t%d\n",
files[i].file_no, file_size[i].file_size, blocks[j].
block_no, blocks[j].block_size);
} else {
 Printf("%d\t\t%d\t\t-1\t\t-1\n", files[i].file_no,
file[i].file_size);
}
}
int main(){
    int n_blocks, n_files, choice;
Printf("memory managmnt scheme");
Printf("Enter the no of blocks:");
Scanf("%d", &n_block);
 Struct Blocks blocks[n_blocks];
    Printf("Enter the size of the blocks:\n");
```

```
for (int i=0; i<n_blocks; i++) {
    blocks[i].blocks_no = i+1);
    Printf("Block %d:", i+1);
    Scanf("%d", &blocks[i].block_size);
    block[i].is_free = 1;
}
Printf("Enter the no of files");
Scanf("%d, &n_files);

Struct file files[n_files];
Printf("Enter the size of the file:");
Scanf("%d", &n_files);

Struct ifile files[n_files];
Printf("Enter the size of the file\n");
  for (int i=0; i<n_files; i++) {
      files[i].file_no = i+1;
      Printf("file %d", i+1);
      Scanf("%d", &files[i].file_size);
  }

do {
Printf("\n1. Best fit \n2. Worst fit. First fit \n4.
Exiting .. \n Enter the choice:");
  Scanf("%d", &choice);
```

```
resetBlocks(blocks, n_blocks);

Switch(choice){

case 1: bestFit( blocks, n_blocks, files, n_files ); break;
case 2: worstFit (blocks, n_blocks, files, n_files); break;
case 3: firstFit (blocks, n_blocks, files, n_files); break;
case 4: printf("Exiting...\n"); break;

default: printf("Invalid choice, Please try again\n");
}

}while(choice != 4);

    return0;

}
```

O/P-> Memory managment scheme

Enter the no of blocks: 5

Enter the size of the block:

Block 1 : 200
Block 2 : 300
Block 3 : 455
Block 4 : 600
Block 5 : 750

Enter the no of files: 3

Enter the size of the file:

File 1: 200
File 2: 300
File 3: 656

1. Best fit
2. Worst fit
3. First fit
4. Exiting

Enter your choice: 1

Memory managmnt Scheme- Best fit

File no:	File-Size:	Block-no:	Block-Size
1	200	1	200
2	300	2	300
3	656	5	750

Enter your choice: 2

Memory managmnt Scheme- Worst fit

File-no:	File-Size:	Block-no:	Block-Size:
1	200	5	750
2	300	4	600
3	656	-	-

Enter your choice: 3

Memory managmnt Scheme - First fit

File-no:	File-Size:	Block-no:	Block-Size:
1	200	1	200
2	300	2	300
3	656	5	750

Enter your choice: 4

Exiting.

Program 10:

Write a C program to simulate page replacement algorithms

a) FIFO
b) LRU
c) Optimal

→FIFO

```
#include <stdio.h>
int main() {
    int n, frames, i, j, k, found, index = 0, page_faults = 0, hits = 0;
    char pages[100];

    printf("Enter the size of the pages:\n");
    scanf("%d", &n);
    printf("Enter the page strings:\n");
    scanf("%s", pages);
    printf("Enter the no of page frames:\n");
    scanf("%d", &frames);

    int mem[frames];
    for (i = 0; i < frames; i++) mem[i] = -1;

    for (i = 0; i < n; i++) {
        found = 0;
        for (j = 0; j < frames; j++) {
            if (mem[j] == pages[i] - '0') {
                hits++;
                found = 1;
                break;
            }
        }
        if (!found) {
            mem[index] = pages[i] - '0';
```

```
            index = (index + 1) % frames;
            page_faults++;
        }
    }

    printf("FIFO Page Faults: %d, Page Hits: %d\n", page_faults, hits);
    return 0;
}
```

```
Enter the size of the pages:
7
Enter the page strings:
103563
Enter the no of page frames:
3
FIFO Page Faults: 6, Page Hits: 1
```

→LRU

```
#include <stdio.h>

int main() {
    int n, frames, i, j, k, page_faults = 0, hits = 0;
    char pages[100];
    int mem[10], used[10];

    printf("Enter the size of the pages:\n");
    scanf("%d", &n);
    printf("Enter the page strings:\n");
    scanf("%s", pages);
    printf("Enter the no of page frames:\n");
    scanf("%d", &frames);
```

```
    for (i = 0; i < frames; i++) {
        mem[i] = -1;
        used[i] = -1;
    }

    for (i = 0; i < n; i++) {
        int page = pages[i] - '0';
        int found = 0;

        for (j = 0; j < frames; j++) {
            if (mem[j] == page) {
                hits++;
                used[j] = i;
                found = 1;
                break;
            }
        }

        if (!found) {
            int lru = 0;
            for (j = 1; j < frames; j++) {
                if (used[j] < used[lru]) lru = j;
            }
            mem[lru] = page;
            used[lru] = i;
            page_faults++;
        }
    }

    printf("LU Page Faults: %d, Page Hits: %d\n", page_faults, hits);
    return 0;
}
```

```
Enter the size of the pages:
7
Enter the page strings:
1303563
Enter the no of page frames:
3
LU Page Faults: 5, Page Hits: 2
```

→Optimal

```
#include <stdio.h>
int main() {
    int n, frames, i, j, k, page_faults = 0, hits = 0;

    printf("Enter the size of the pages:\n");
    scanf("%d", &n);

    char pages[n + 1];
    printf("Enter the page strings:\n");
    scanf("%s", pages);

    printf("Enter the no of page frames:\n");
    scanf("%d", &frames);

    int mem[frames], next_use[frames];
    for (i = 0; i < frames; i++) {
        mem[i] = -1;
    }

    for (i = 0; i < n; i++) {
        int page = pages[i] - '0';
        int found = 0;

        for (j = 0; j < frames; j++) {
```

```
    if (mem[j] == page) {
        hits++;
        found = 1;
        break;
    }
}

if (!found) {
    if (page_faults < frames) {
        mem[page_faults++] = page;
    } else {
        for (j = 0; j < frames; j++) {
            next_use[j] = -1;
            for (k = i + 1; k < n; k++) {
                if (mem[j] == pages[k] - '0') {
                    next_use[j] = k;
                    break;
                }
            }
        }

        int farthest = 0;
        for (j = 1; j < frames; j++) {
            if (next_use[j] > next_use[farthest]) {
                farthest = j;
            }
        }

        mem[farthest] = page;
        page_faults++;
    }
}
```

```
    }

    printf("Optimal Page Faults: %d, Page Hits: %d\n", page_faults, hits);
    return 0;
}
```

```
Enter the size of the pages:
7
Enter the page strings:
13035631
Enter the no of page frames:
3
Optimal Page Faults: 6, Page Hits: 1
```

FIFO, LRU & Optimal :-

```
# include <stdio.h>
int n, m;
int p[10000], f[10000];
void fifo () {
    for (int i=0; i<m; ++i) f[i]=-1;
    int h=0;
    int s=0;
    for (int i=0; i<n; ++i) {
        int k=0;
        for (int r=0; r<m; ++r)
        if (p[i]==f[r]) {++h; k=1; break;}
        if (k==1) continue;
        for (int r=0; r<m; ++r)
        if (f[r]==-1) {
            f[r]=p[i]; k=2; break; }
        if (k==2) continue;
        f[s]=p[i];
        s=(s+1)%m;
    }
    Printf("FIFO:\t\t page-Hit: %d\t\t page-Fault:
%d\n", h, n-h);
}
```

```
Void_LRU() {
  for (int i=0;i<m;++i) f[i]=-1;
  int h=0;
  int t[m];
  for (int i=0;i<m;++i) t[i]=0;
  for (int i=0; i<n;++i) {
    int K=0;
    for (int r=0;r<m;++r)
    if (f[r]==p[i]) {t[r]=1;K=1; break;}
  if (K==1) {++h; continue;}
  for (int r=0;r<m;++r)
  if (f[r]==-1) { f[r]=p[i]; t[r]=1;
      K=2; break;}
  if (K==2) continue;
  int x=0;
  for (int r=1;r<m;++r)
  if ( t[r]<t[x])x=r;
  t[x]=1;f[x]=p[i];
  }
  Printf("LRU:\t\t page-Hit: %d\t\t page-fault:
%d\n", h, n-h);
}
Void Optimal() {
  for (int i=0;i<m;++i) f[i]=-1;
  int h=0;
  for (int i=0; i<n;++i) {
      int K=0;
```

```
for (int r=0; r<m ;++r)
 if (f[r]==p[i]) {K=1;
   ++h; break; }

if (K==1) Continue;

for (int r=0; r<m; ++r)
 if [f[r]==2 -1) {
K=2; f[r]=p[i]; break; }

if(K=2) Continue;
~~int~~ int t=-1; x x=-1, y=1;
   for (int j=0; j<m; ++j) {
      y=1;
      for (K=i+1; K<n; ++K)
      if [f[j]==p[K]] {
         y=0;
         if (K>t) {xx=j; t=K;} break;
      }
      if (y) continue;
      f[xx]=p[i];
   }
   Printf("optimal :\t\t page-hit :%d \t\t page-fault:
%d\n", h, n-h);
}
int main () {
   Printf("Enter the no. of page string & page frame:");
   Scanf("%d%d", &n, &m);
   Printf("Enter the page string:");
   for (int i=0; i<n; ++i)
      Scanf("%d", &p[i]);
```

```
fifo();LRU();Optimal();
}
```

o/p->

Enter the No. of page string & page frame: 7 3

Enter the Page-string: 1 3 0 3 5 6 3

FIFO : Page-Hit=1 Page-Fault=6

LRU : Page-Hit=2 Page-Fault=5

OPTIMAL: Page-Hit=2 Page-Fault=5