VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

OPERATING SYSTEMS

Submitted by

VIKAS SHASHI(1WA23CS043)

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 VIKAS SHASHI(1WA23CS043), 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.

Ms. Sheetal V
Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

INDEX

Name : VIKAS SHASHI Class :

Section : G1 Roll No. : 1WA23CS043 Subject :

Sl. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1.	06-02-25	Lab Program 1	10	
2.		Lab Program 2		20-3-25
3.		Lab Program 3		
4.	16-04-25	Lab Program 4		17-4-25
5		Lab Program 5		
6.	08-05-25	Lab Program 6		
7.	15-05-25	Lab Program 7		11-6-2025

Index Sheet

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

Question:

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)

Code:

FCFS:

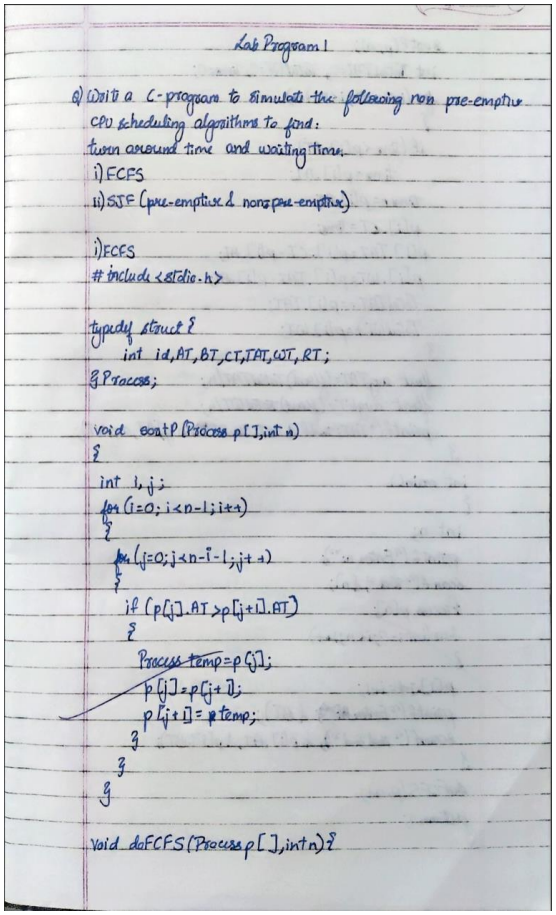
```
#include<stdio.h>

void sort(int proc_id[],int at[],int bt[],int n)
{
   int min=at[0],temp=0;
   for(int i=0;i<n;i++)
   {
      min=at[i];
      for(int j=i;j<n;j++)
      {
         if(at[j]<min)
         {
            temp=at[i];
            at[i]=at[j];
            at[j]=temp;
            temp=bt[j];
            bt[j]=bt[i];
            bt[i]=temp;
            temp=proc_id[i];
            proc_id[i]=proc_id[j];
            proc_id[j]=temp;
         }
      }
   }
}

void main()
{
```

```
int n,c=0;
printf("Enter number of processes: ");
scanf("%d",&n);
int proc_id[n],at[n],bt[n],ct[n],tat[n],wt[n];
double avg_tat=0.0,ttat=0.0,avg_wt=0.0,twt=0.0;
for(int i=0;i<n;i++)
   proc_id[i]=i+1;
printf("Enter arrival times:\n");
for(int i=0;i<n;i++)
   scanf("%d",&at[i]);
printf("Enter burst times:\n");
for(int i=0;i<n;i++)
   scanf("%d",&bt[i]);

sort(proc_id,at,bt,n);
//completion time
for(int i=0;i<n;i++)
{
   if(c>=at[i])
      c+=bt[i];
   else
      c+=at[i]-ct[i-1]+bt[i];
   ct[i]=c;
}
//turnaround time
for(int i=0;i<n;i++)
   tat[i]=ct[i]-at[i];
//waiting time
for(int i=0;i<n;i++)
   wt[i]=tat[i]-bt[i];

printf("FCFS scheduling:\n");
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]);

for(int i=0;i<n;i++)
{
   ttat+=tat[i];twt+=wt[i];
}
avg_tat=ttat/(double)n;
avg_wt=twt/(double)n;
printf("\nAverage turnaround time:%lfms\n",avg_tat);
printf("\nAverage waiting time:%lfms\n",avg_wt);
```

}

Result:

```
Process  Burst Time  Arrival Time  Waiting Time  Turn Around Time
0        5          0             0             5
1        3          1             4             7
2        8          2             6             14
3        6          3             13            19
Average Waiting Time: 5.75
Average Turnaround Time: 11.25

Process returned 0 (0x0)   execution time : 0.320 s
Press any key to continue.
```

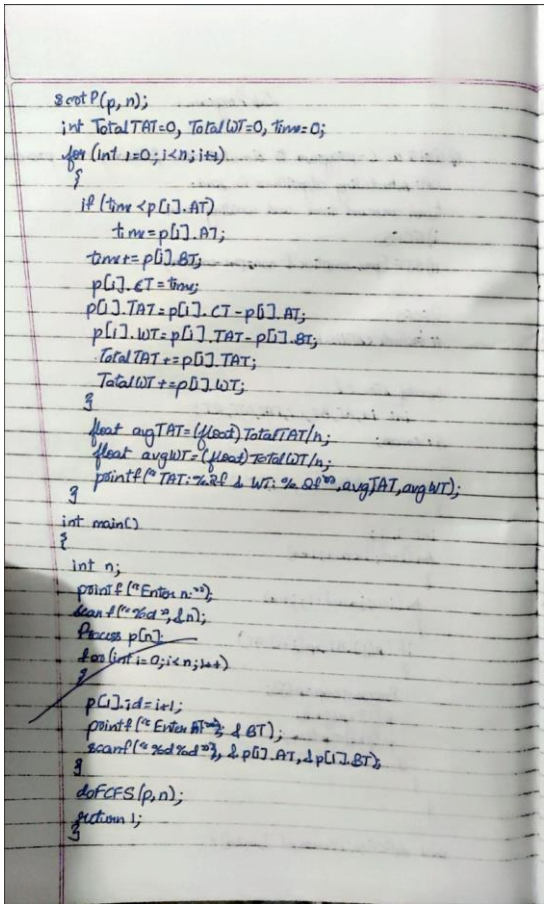


Lab Program 1

Q) Write a C-program to simulate the following non pre-emptive CPU scheduling algorithms to find: turn around time and waiting time.
i) FCFS
ii) SJF (pre-emptive & non-pre-emptive)

i) FCFS

```
#include<stdio.h>

typedef struct {
    int id,AT,BT,CT,TAT,WT,RT;
}Process;

void sortP(Process p[],int n)
{
 int i,j;
 for(i=0;i<n-1;i++)
 {
  for(j=0;j<n-i-1;j++)
  {
   if (p[j].AT>p[j+1].AT)
   {
    Process temp=p[j];
    p[j]=p[j+1];
    p[j+1]=temp;
   }
  }
 }

void doFCFS(Process p[],int n){
```



```
sortP(p,n);
int TotalTAT=0, TotalWT=0, time=0;
for (int i=0;i<n;i++)
 {
  if (time<p[i].AT)
     time=p[i].AT;
  time+=p[i].BT;
  p[i].CT=time;
  p[i].TAT=p[i].CT-p[i].AT;
  p[i].WT=p[i].TAT-p[i].BT;
  TotalTAT+=p[i].TAT;
  TotalWT+=p[i].WT;
 }
 float avgTAT=(float)TotalTAT/n;
 float avgWT=(float)TotalWT/n;
 printf("TAT:%.2f & WT: %.2f",avgTAT,avgWT);
}
int main()
{
 int n;
 printf("Enter n:");
 scanf("%d",&n);
 Process p[n];
 for(int i=0;i<n;i++)
 {
  p[i].id=i+1;
  printf("Enter AT & BT");
  scanf("%d %d",&p[i].AT,&p[i].BT);
 }
 doFCFS(p,n);
 return 1;
}
```

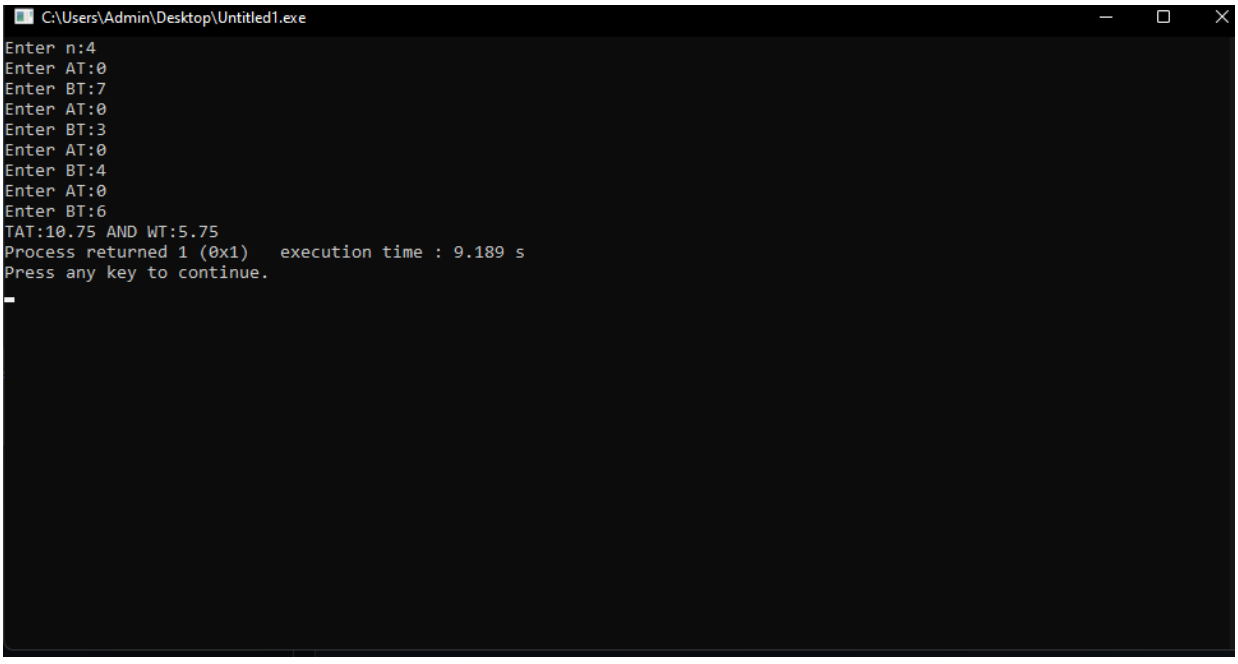
SJF(Non-preemptive):

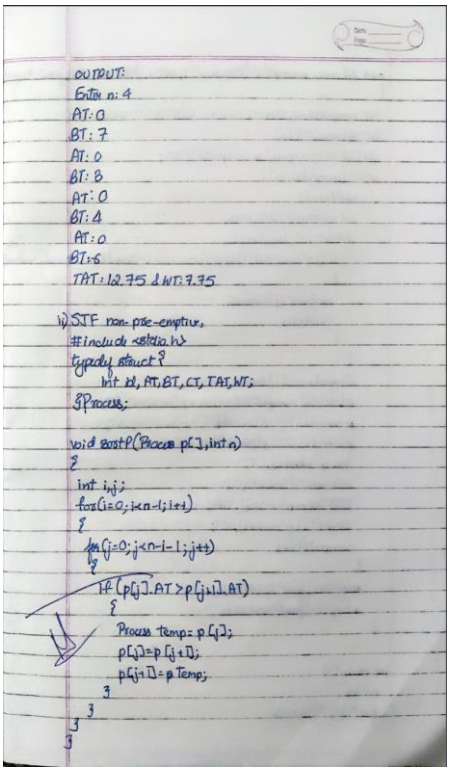
```
#include<stdio.h>

typedef struct {
   int id,AT,BT,CT,TAT,WT,RT;
}Process;

void sortP(Process p[],int n)
{
   int i,j;
   for(i=0;i<n-1;i++)
   {
      for(j=0;j<n-i-1;j++)
      {
         if(p[j].AT>p[j+1].AT)
         {
         Process temp=p[j];
         p[j]=p[j+1];
         p[j+1]=temp;
         }
      }
   }
}

void sjfNP(Process p[], int n) {
   int completed = 0, time = 0, minIdx, totalTAT = 0, totalWT = 0;
   int isCompleted[n];
   for (int i = 0; i < n; i++)
      isCompleted[i] = 0;
```

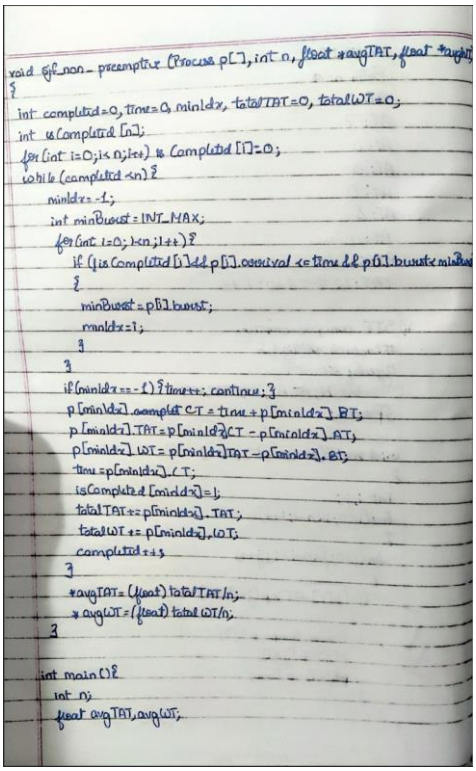
```
    while (completed < n) {
        minIdx = -1;
        int minBurst = 100;
        for (int i = 0; i < n; i++) {
            if (!isCompleted[i] && p[i].AT <= time && p[i].BT < minBurst) {
                minBurst = p[i].BT;
                minIdx = i;
            }
        }
        if (minIdx == -1)
        {
                time++;
                continue;
        }

        p[minIdx].CT = time + p[minIdx].BT;
        p[minIdx].TAT = p[minIdx].CT - p[minIdx].AT;
        p[minIdx].WT = p[minIdx].TAT - p[minIdx].BT;
        time = p[minIdx].CT;
        isCompleted[minIdx] = 1;
        totalTAT += p[minIdx].TAT;
        totalWT += p[minIdx].WT;
        completed++;
    }
    float avgTAT = (float)totalTAT / n;
    float avgWT = (float)totalWT / n;
    printf("TAT:%.2f AND WT:%.2f",avgTAT,avgWT);
}

int main()
{
    int n;
```

```
    printf("Enter n:");
    scanf("%d",&n);
    Process p[n];
    for(int i=0;i<n;i++)
    {
        p[i].id=i+1;
        printf("Enter AT:");
        scanf("%d",&p[i].AT);
        printf("Enter BT:");
        scanf("%d",&p[i].BT);
    }
    sjfNP(p,n);
    return 1;
}
```

OUTPUT:







SJF(Pre-Emptive):

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

typedef struct {
    int id, arrival, burst, remaining, completion, turnaround, waiting;
} Process;

void sortByArrival(Process p[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (p[j].arrival > p[j + 1].arrival) {
                Process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
```

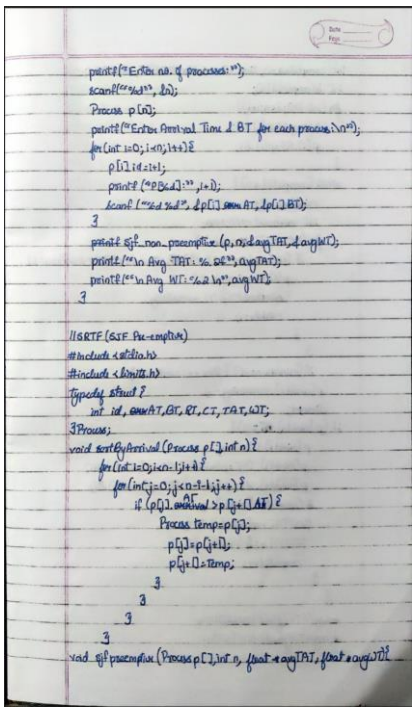
```
        }
      }
   }
}

void sjf_preemptive(Process p[], int n, float *avgTAT, float *avgWT) {
   int completed = 0, time = 0, minIdx, totalTAT = 0, totalWT = 0;
   int isCompleted[n];
   for (int i = 0; i < n; i++) {
      isCompleted[i] = 0;
      p[i].remaining = p[i].burst;
   }

   while (completed < n) {
      minIdx = -1;
      int minBurst = INT_MAX;
      for (int i = 0; i < n; i++) {
         if (!isCompleted[i] && p[i].arrival <= time && p[i].remaining < minBurst &&
p[i].remaining > 0) {
            minBurst = p[i].remaining;
            minIdx = i;
         }
      }
      if (minIdx == -1) { time++; continue; }

      p[minIdx].remaining--;
      time++;
      if (p[minIdx].remaining == 0) {
         p[minIdx].completion = time;
         p[minIdx].turnaround = p[minIdx].completion - p[minIdx].arrival;
         p[minIdx].waiting = p[minIdx].turnaround - p[minIdx].burst;
         isCompleted[minIdx] = 1;
```

```
            totalTAT += p[minIdx].turnaround;
            totalWT += p[minIdx].waiting;
            completed++;
        }
    }
    *avgTAT = (float)totalTAT / n;
    *avgWT = (float)totalWT / n;
}

void display(Process p[], int n, float avgTAT, float avgWT) {
    printf("\nPID  Arrival  Burst  Completion  Turnaround  Waiting\n");
    for (int i = 0; i < n; i++) {
        printf("%3d %7d %6d %10d %10d %8d\n", p[i].id, p[i].arrival, p[i].burst, p[i].completion,
p[i].turnaround, p[i].waiting);
    }
    printf("\nAverage Turnaround Time: %.2f", avgTAT);
    printf("\nAverage Waiting Time: %.2f\n", avgWT);
}

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

    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1;
        printf("P[%d]: ", i + 1);
        scanf("%d %d", &p[i].arrival, &p[i].burst);
    }
```

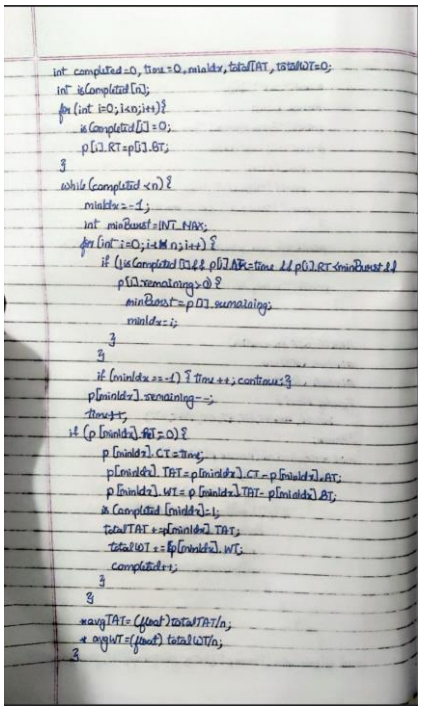
```
    printf("\nShortest Job First (Preemptive) Scheduling\n");
    sjf_preemptive(p, n, &avgTAT, &avgWT);
    display(p, n, avgTAT, avgWT);

    return 0;
}
```

OUTPUT:

```
Enter number of processes: 4
Enter Arrival Time and Burst Time for each process:
P[1]: 0 8
P[2]: 1 4
P[3]: 2 9
P[4]: 3 5

Shortest Job First (Preemptive) Scheduling

PID  Arrival  Burst  Completion  Turnaround  Waiting
  1        0      8          17          17        9
  2        1      4           5           4        0
  3        2      9          26          24       15
  4        3      5          10           7        2

Average Turnaround Time: 13.00
Average Waiting Time: 6.50
```



Lab 2



Program-

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)

Code:

Priority:

```
#include <stdio.h>

#define MAX 10

typedef struct {
    int pid, at, bt, pt, remaining_bt, ct, tat, wt, rt, is_completed, st;
} Process;

// Function for Non-Preemptive Priority Scheduling
void nonPreemptivePriority(Process p[], int n) {
    int time = 0, completed = 0;

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

        for (int i = 0; i < n; i++) {
            if (p[i].at <= time && !p[i].is_completed && p[i].pt > highest_priority) {
                highest_priority = p[i].pt;
                selected = i;
            }
        }
```

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

        // If RT is not yet calculated, calculate it
        if (p[selected].rt == -1) {
            p[selected].st = time;  // Start time
            p[selected].rt = time - p[selected].at;  // Response Time = Start Time - Arrival Time
        }

        time += p[selected].bt;
        p[selected].ct = time;
        p[selected].tat = p[selected].ct - p[selected].at;
        p[selected].wt = p[selected].tat - p[selected].bt;
        p[selected].is_completed = 1;
        completed++;
    }
}

// Function for Preemptive Priority Scheduling
void preemptivePriority(Process p[], int n) {
    int time = 0, completed = 0;

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

        for (int i = 0; i < n; i++) {
            if (p[i].at <= time && p[i].remaining_bt > 0 && p[i].pt > highest_priority) {
                highest_priority = p[i].pt;
                selected = i;
```

```
            }
        }

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

        // If RT is not yet calculated, calculate it
        if (p[selected].rt == -1) {
            p[selected].st = time;  // Start time
            p[selected].rt = time - p[selected].at;  // Response Time = Start Time - Arrival Time
        }

        p[selected].remaining_bt--;
        time++;

        if (p[selected].remaining_bt == 0) {
            p[selected].ct = time;
            p[selected].tat = p[selected].ct - p[selected].at;
            p[selected].wt = p[selected].tat - p[selected].bt;
            completed++;
        }
    }
}

// Function to display the results of processes
void displayProcesses(Process p[], int n) {
    float avg_tat = 0, avg_wt = 0, avg_rt = 0;

    printf("\nPID\tAT\tBT\tPriority\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++) {
```

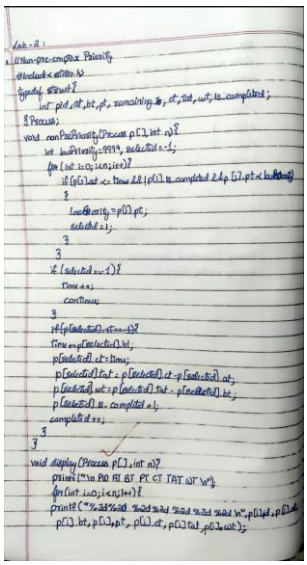
```
        printf("%d\t%d\t%d\t%d\t\t%d\t%d\t%d\t%d\n",
               p[i].pid, p[i].at, p[i].bt, p[i].pt, p[i].ct, p[i].tat, p[i].wt, p[i].rt);
        avg_tat += p[i].tat;
        avg_wt += p[i].wt;
        avg_rt += p[i].rt;
    }

    printf("\nAverage TAT: %.2f", avg_tat / n);
    printf("\nAverage WT: %.2f", avg_wt / n);
    printf("\nAverage RT: %.2f\n", avg_rt / n);
}

int main() {
    Process p[MAX];
    int n, choice;

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

    // Getting arrival times, burst times, and priorities for each process
    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1; // Process ID starts from 1
        printf("\nEnter Arrival Time, Burst Time, and Priority for Process %d:\n", p[i].pid);
        printf("Arrival Time: ");
        scanf("%d", &p[i].at);
        printf("Burst Time: ");
        scanf("%d", &p[i].bt);
        printf("Priority (higher number means higher priority): ");
        scanf("%d", &p[i].pt);
        p[i].remaining_bt = p[i].bt;  // Initialize remaining burst time
        p[i].is_completed = 0;        // Mark process as incomplete
```

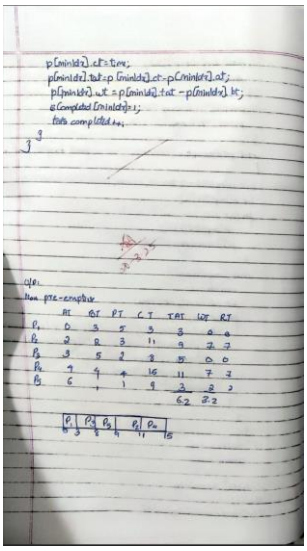
```
    p[i].rt = -1;               // Response time will be calculated later
}

// Menu to choose the scheduling method
while (1) {
    printf("\nPriority Scheduling Menu:\n");
    printf("1. Non-Preemptive Priority Scheduling\n");
    printf("2. Preemptive Priority Scheduling\n");
    printf("3. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            nonPreemptivePriority(p, n);
            printf("Non-Preemptive Scheduling Completed!\n");
            displayProcesses(p, n);
            break;
        case 2:
            preemptivePriority(p, n);
            printf("Preemptive Scheduling Completed!\n");
            displayProcesses(p, n);
            break;
        case 3:
            printf("Exiting...\n");
            return 0;
        default:
            printf("Invalid choice! Try again.\n");
    }
}

return 0;
```

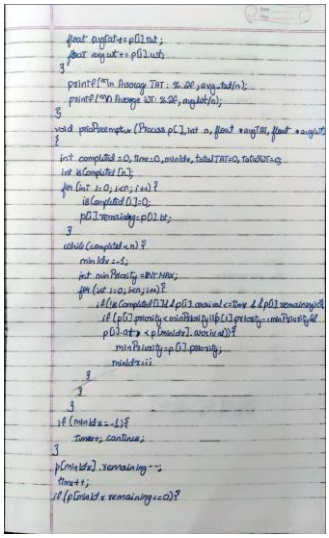
}

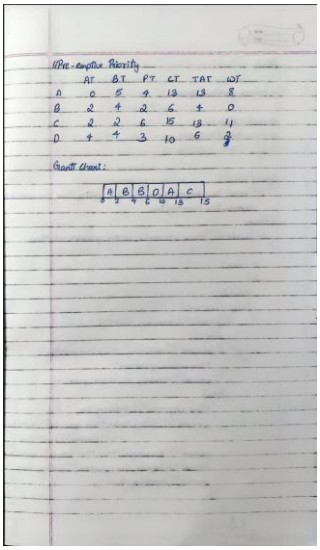
OUTPUT:

```
Priority Scheduling Menu:
1. Non-Preemptive Priority Scheduling
2. Preemptive Priority Scheduling
3. Exit
Enter your choice: 1
Non-Preemptive Scheduling Completed!

PID     AT      BT      Priority        CT      TAT     WT      RT
1       0       4       2               4       4       0       0
2       1       3       3               15      14      11      11
3       2       1       4               12      10      9       9
4       3       5       5               9       6       1       1
5       4       2       5               11      7       5       5

Average TAT: 8.20
Average WT: 5.20
Average RT: 5.20

Priority Scheduling Menu:
1. Non-Preemptive Priority Scheduling
2. Preemptive Priority Scheduling
3. Exit
Enter your choice: 2
Preemptive Scheduling Completed!

PID     AT      BT      Priority        CT      TAT     WT      RT
1       0       4       2               15      15      11      0
2       1       3       3               12      11      8       11
3       2       1       4               3       1       0       9
4       3       5       5               8       5       0       1
5       4       2       5               10      6       4       5

Average TAT: 7.60
Average WT: 4.60
Average RT: 5.20
```









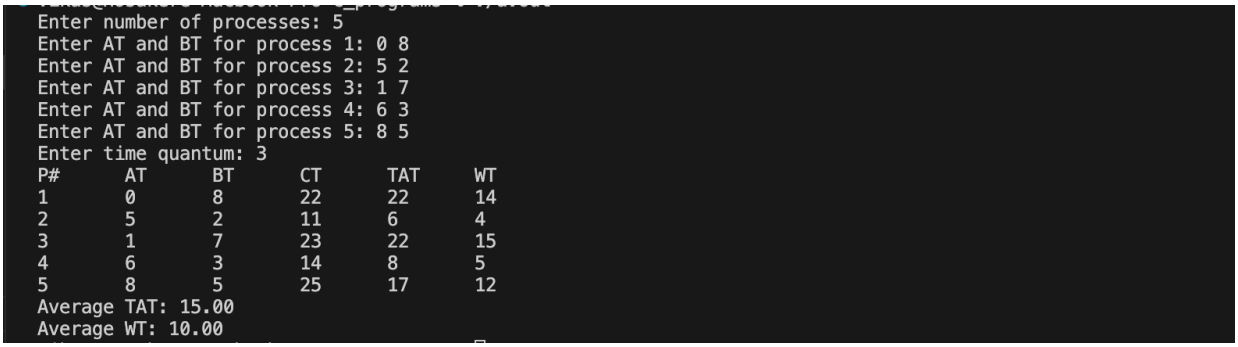
Round Robin(CODE)

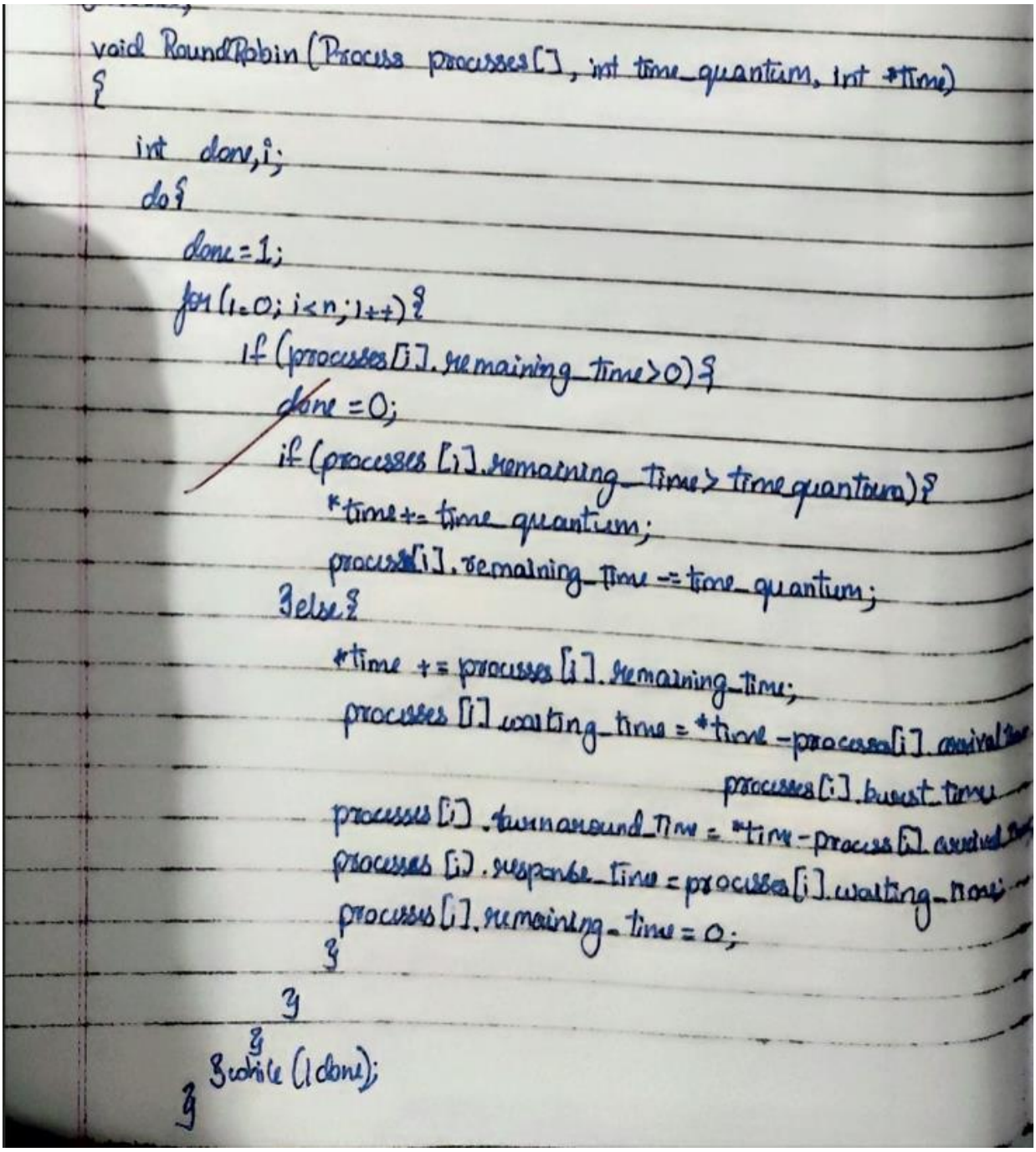
```
#include <stdio.h>

#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;
}
```

OUTPUT:


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



```
void RoundRobin(Process processes[], 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;
                    process[i].remaining_time -= time_quantum;
                }else{
                    *time += processes[i].remaining_time;
                    processes[i].waiting_time = *time - process[i].arrival
                                                processes[i].burst_time
                    processes[i].turnaround_time = *time - process[i].arrival
                    processes[i].response_time = processes[i].waiting_time
                    process[i].remaining_time = 0;
                }
            }
        }
    }while(!done);
}
```

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.

CODE:

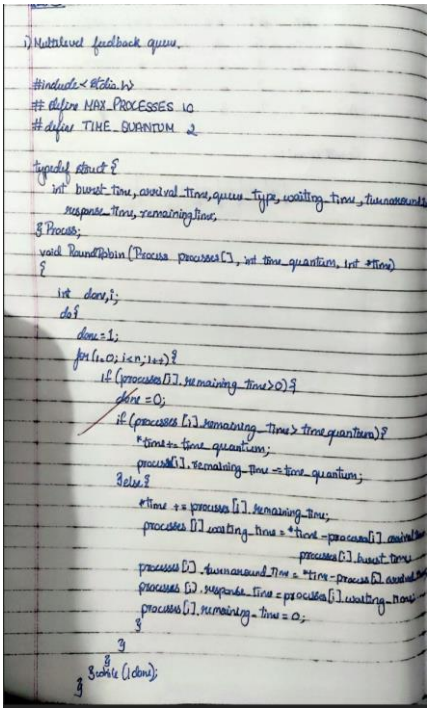
```
#include <stdio.h>

#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);
}
```

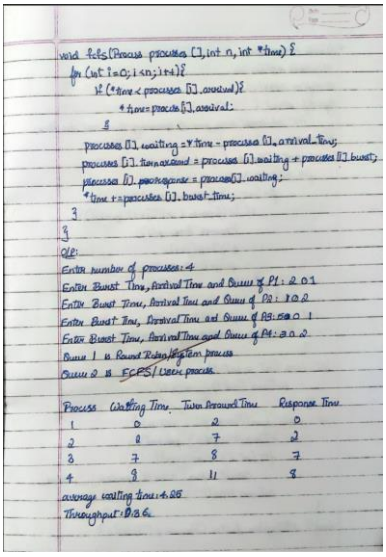
```
void fcfs(Process processes[], int n, int *time) {
    for (int i = 0; i < n; i++) {
        if (*time < processes[i].arrival_time) {
            *time = processes[i].arrival_time;
        }
        processes[i].waiting_time = *time - processes[i].arrival_time;
        processes[i].turnaround_time = 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;
            }
        }
```

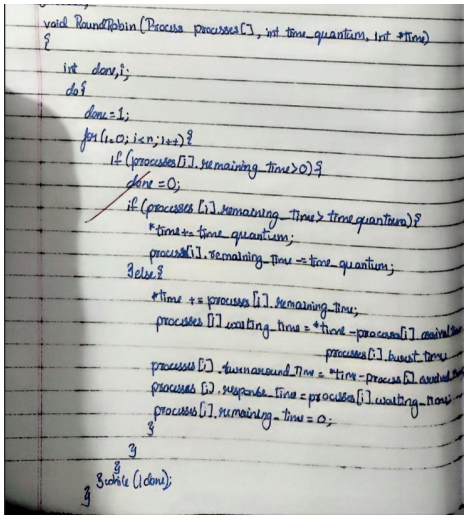
```
    }

    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;
}
```

OUTPUT:

```
Enter number of processes: 4
Enter Burst Time, Arrival Time and Queue of P1: 5 3 1
Enter Burst Time, Arrival Time and Queue of P2: 9 4 2
Enter Burst Time, Arrival Time and Queue of P3: 8 3 1
Enter Burst Time, Arrival Time and Queue of P4: 2 4 3

Queue 1 is System Process
Queue 2 is User Process

Process  Waiting Time  Turn Around Time  Response Time
1         1            6                 1
2         2            10                 2
3         9            18                 9
4         18            20                  18

Average Waiting Time: 7.50
Average Turn Around Time: 13.50
Average Response Time: 7.50
Throughput: 0.17
Process returned 24 (0x24) execution time: 24.000 s
```







Program 4

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

A. **Rate- Monotonic**
B. **Earliest-deadline First**
C. **Proportional scheduling**

A)Rate monotonic:

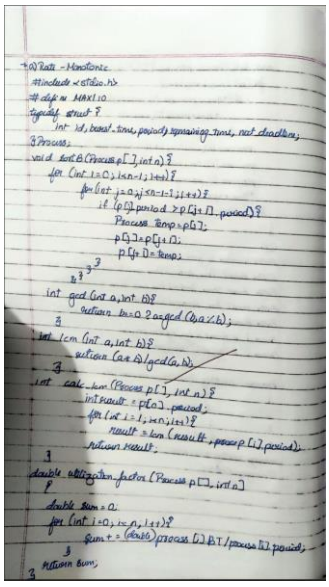
```
#include <stdio.h>

#define MAX_TASKS 10

typedef struct {
    int id;
    int exec_time;
    int period;
    int remaining;
} Task;

// Function to sort tasks by period (ascending order)
void sortTasksByPeriod(Task tasks[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (tasks[j].period > tasks[j+1].period) {
                // Swap tasks
```

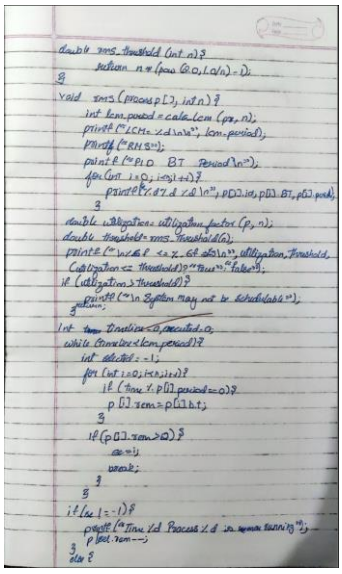
```
            Task temp = tasks[j];
            tasks[j] = tasks[j+1];
            tasks[j+1] = temp;
        }
    }
  }
}

void schedule(Task tasks[], int n, int hyperperiod) {
  printf("\nTime\tTask\n");

  for (int t = 0; t < hyperperiod; t++) {
    int task_to_run = -1;

    // Find first task with remaining work (already in priority order)
    for (int i = 0; i < n; i++) {
      if (tasks[i].remaining > 0) {
        task_to_run = i;
        break;
      }
    }

    if (task_to_run != -1) {
      tasks[task_to_run].remaining--;
      printf("%d\tT%d\n", t, tasks[task_to_run].id);
    } else {
      printf("%d\tIDLE\n", t);
    }

    // Reset tasks at start of their periods
    for (int i = 0; i < n; i++) {
      if ((t + 1) % tasks[i].period == 0) {
        tasks[i].remaining = tasks[i].exec_time;
      }
    }
  }
}

int main() {
  Task tasks[MAX_TASKS];
  int n, hyperperiod;
```

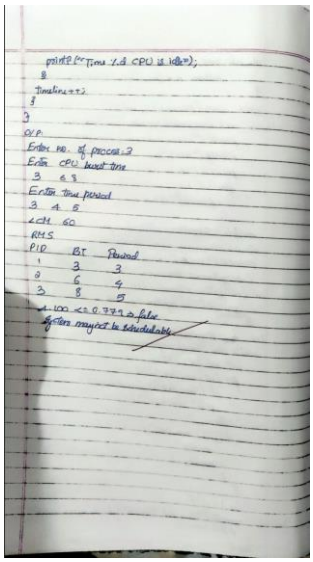
```
    printf("Number of tasks (max %d): ", MAX_TASKS);
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        tasks[i].id = i + 1;

        printf("T%d execution time: ", i + 1);
        scanf("%d", &tasks[i].exec_time);

        printf("T%d period: ", i + 1);
        scanf("%d", &tasks[i].period);

        tasks[i].remaining = tasks[i].exec_time;
    }

    // Sort tasks by period (Rate Monotonic priority order)
    sortTasksByPeriod(tasks, n);

    printf("Hyperperiod (LCM of all periods): ");
    scanf("%d", &hyperperiod);

    schedule(tasks, n, hyperperiod);

    return 0;
}
```

OUTPUT:

```
Enter the number of processes: 3
Enter the CPU burst times:
3 6 8
Enter the time periods:
2 4 5
LCM=20

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

4.600000 <= 0.779763 => false

System may not be schedulable!
```







EARLIEST DEADLINE:

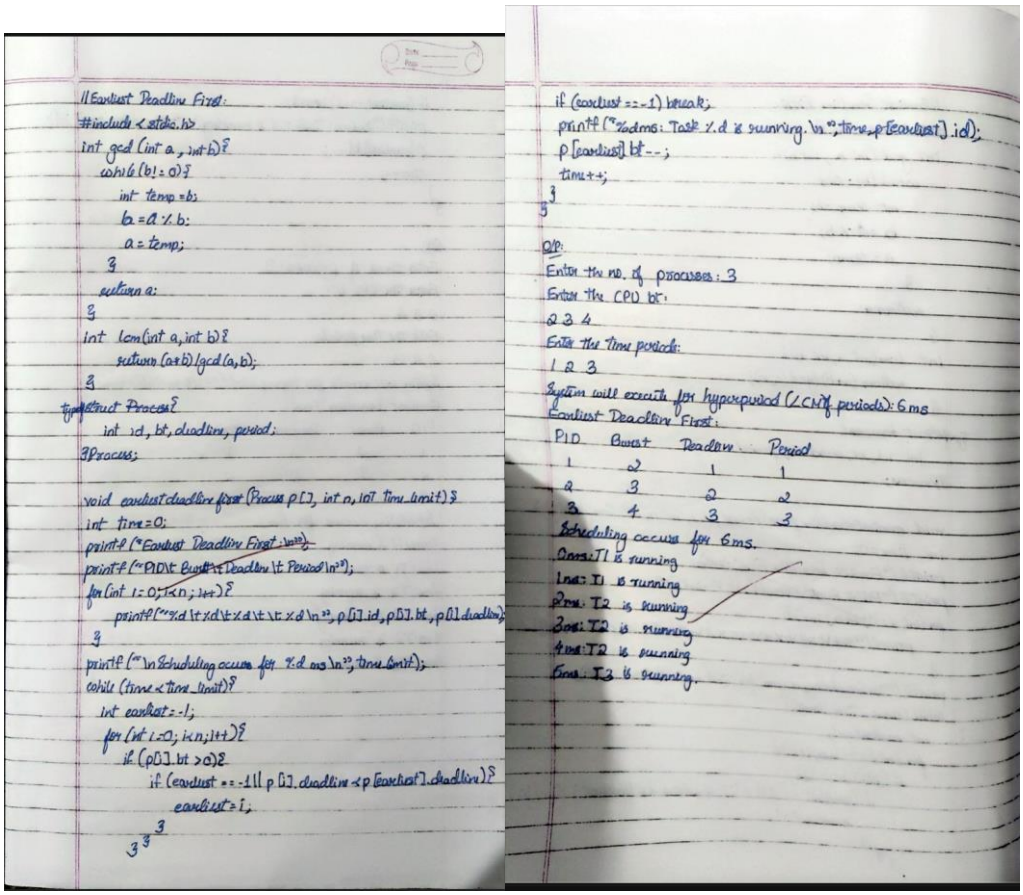
```
#include <stdio.h>

#define MAX_TASKS 10
#define MAX_INSTANCES 100

typedef struct {
    int pid;            // Task ID
```

```
    int arrival;        // Arrival time
    int deadline;       // Relative deadline
    int abs_deadline;   // Absolute deadline (arrival + deadline)
    int burst;          // Total execution time
    int remaining;      // Remaining execution time
} Task;

// Function to perform Earliest Deadline First scheduling
void earliestDeadlineFirst(Task instances[], int inst_count, int sim_time) {
    printf("\nEarliest Deadline First Scheduling:\n");
    printf("Time\tTask\n");

    for (int time = 0; time < sim_time; time++) {
        int selected = -1;
        int min_deadline = 1e9;

        // Find the task with the earliest absolute deadline that is ready
        for (int i = 0; i < inst_count; i++) {
            if (instances[i].arrival <= time && instances[i].remaining > 0 &&
                instances[i].abs_deadline < min_deadline) {
                min_deadline = instances[i].abs_deadline;
                selected = i;
            }
        }

        if (selected != -1) {
            instances[selected].remaining--;
            printf("%d\tT%d\n", time, instances[selected].pid);
        } else {
            printf("%d\tIdle\n", time);
        }
    }
}

int main() {
    int n, sim_time;
    Task instances[MAX_INSTANCES];
    int inst_count = 0;

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

```
    int period[MAX_TASKS], deadline[MAX_TASKS], burst[MAX_TASKS];

    // Input task details
    for (int i = 0; i < n; i++) {
        printf("Enter Burst, Deadline, Period for Task T%d: ", i + 1);
        scanf("%d %d %d", &burst[i], &deadline[i], &period[i]);
    }

    printf("Enter total simulation time: ");
    scanf("%d", &sim_time);

    // Generate task instances
    for (int i = 0; i < n; i++) {
        for (int t = 0; t < sim_time; t += period[i]) {
            instances[inst_count++] = (Task){
                .pid = i + 1,
                .arrival = t,
                .burst = burst[i],
                .remaining = burst[i],
                .deadline = deadline[i],
                .abs_deadline = t + deadline[i]
            };
        }
    }

    // Call the EDF scheduler
    earliestDeadlineFirst(instances, inst_count, sim_time);

    return 0;
}
```

OUTPUT:

```
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

System will execute for hyperperiod (LCM of periods): 6 ms
Earliest Deadline Scheduling:
PID     Burst   Deadline        Period
1       2               1               1
2       3               2               2
3       4               3               3

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

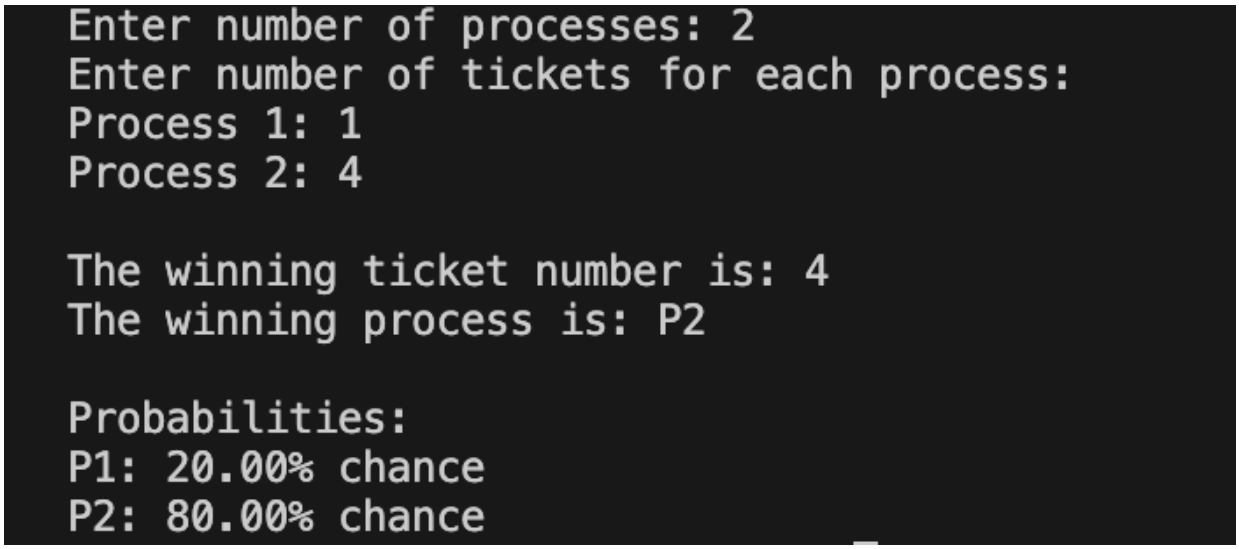


PROPORTIONAL DEADLINE :

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

int main() {
   srand(time(NULL)); // Seed random number generator

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

   int tickets[n], cumulative[n], process[n];
   int totalTickets = 0;

   printf("Enter number of tickets for each process:\n");
   for (int i = 0; i < n; i++) {
```

```
        printf("Process %d: ", i + 1);
        scanf("%d", &tickets[i]);
        totalTickets += tickets[i];
        process[i] = i + 1;
    }

    // Create cumulative ticket ranges
    cumulative[0] = tickets[0];
    for (int i = 1; i < n; i++) {
        cumulative[i] = cumulative[i - 1] + tickets[i];
    }

    // Pick a random ticket
    int winningTicket = rand() % totalTickets;
    printf("\nThe winning ticket number is: %d\n", winningTicket);

    // Find which process owns the winning ticket
    for (int i = 0; i < n; i++) {
        if (winningTicket < cumulative[i]) {
            printf("The winning process is: P%d\n", process[i]);
            break;
        }
    }

    // Display probabilities
    printf("\nProbabilities:\n");
    for (int i = 0; i < n; i++) {
        double probability = (double)tickets[i] / totalTickets * 100;
        printf("P%d: %.2f%% chance\n", process[i], probability);
    }

    return 0;
}
```

OUTPUT:


Enter number of processes: 2
Enter number of tickets for each process:
Process 1: 1
Process 2: 4
The winning ticket number is: 4
The winning process is: P2
Probabilities:
P1: 20.00% chance
P2: 80.00% chance

Lab Program-04

Write a C program to simulate producer-consumer problem using semaphores and concept of Dining Philosophers problem.

Producer consumer problem

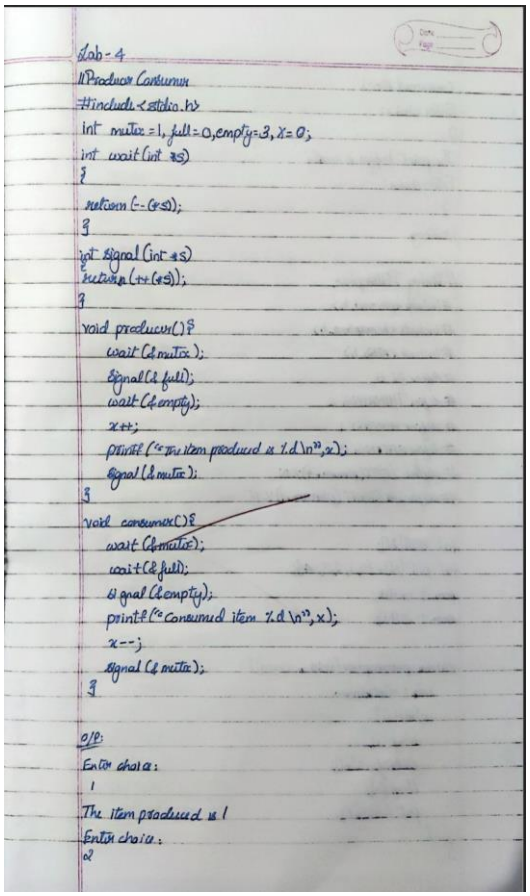
```
#include <stdio.h>
#include <stdlib.h>

int mutex = 1;
int full = 0;
int empty = 5;
int item = 0;

int wait(int s);
int signal(int s);
void producer();
void consumer();

int main() {
   int choice;

   printf("Producer-Consumer Problem Simulation\n");

   while (1) {
      printf("\n1. Produce\n2. Consume\n3. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            if ((mutex == 1) && (empty != 0)) {
               producer();
            } else {
               printf("Buffer is full or mutex is locked. Cannot produce.\n");
            }
            break;
         case 2:
            if ((mutex == 1) && (full != 0)) {
```

```
                consumer();
            } else {
                printf("Buffer is empty or mutex is locked. Cannot consume.\n");
            }
            break;
        case 3:
            exit(0);
        default:
            printf("Invalid choice. Try again.\n");
      }
   }

   return 0;
}

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

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

void producer() {
   mutex = wait(mutex);
   empty = wait(empty);
   full = signal(full);

   item++;
   printf("Produced item %d\n", item);

   mutex = signal(mutex);
}

void consumer() {
   mutex = wait(mutex);
   full = wait(full);
   empty = signal(empty);

   printf("Consumed item %d\n", item);
   item--;
```

```
    mutex = signal(mutex);
}
```

OUTPUT:

```
Enter choice:
1
The item produced is 1
Enter choice:
2
Consumed item 1
Enter choice:
2
The print buffer is empty
Enter choice:
3
Exiting.
```



Dining philosopher problem:

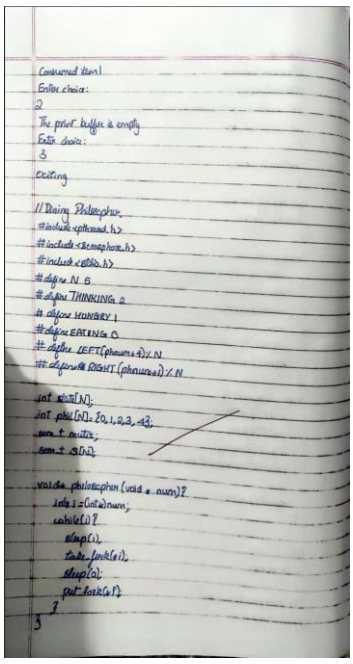
```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <stdlib.h>

#define N 5 // Number of philosophers
#define MEALS 3 // Number of meals per philosopher

sem_t chopsticks[N];

void* philosopher(void* num) {
    int id = *(int*)num;
    int left = id;
    int right = (id + 1) % N;
    int meals_eaten = 0;
```

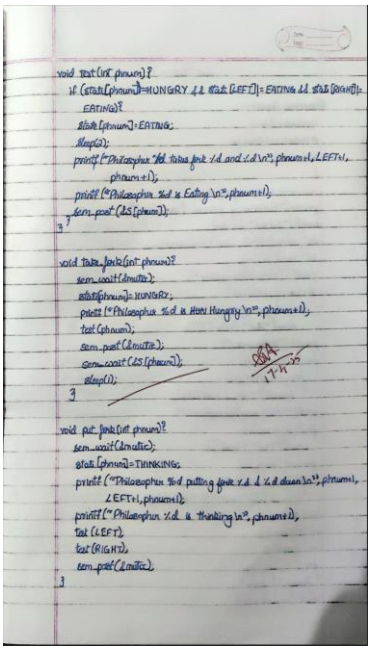
```
    while (meals_eaten < MEALS) {
        // Thinking
        printf("Philosopher %d is thinking...\n", id);
        sleep(1 + rand() % 3);

        // Hungry - try to get chopsticks
        printf("Philosopher %d is hungry\n", id);

        if (id % 2 == 0) {
            // Even philosopher: right then left
            sem_wait(&chopsticks[right]);
            printf("\tPhilosopher %d picked up right chopstick %d\n", id, right);
            usleep(100000); // Small delay
            sem_wait(&chopsticks[left]);
            printf("\tPhilosopher %d picked up left chopstick %d\n", id, left);
        } else {
            // Odd philosopher: left then right
            sem_wait(&chopsticks[left]);
            printf("\tPhilosopher %d picked up left chopstick %d\n", id, left);
            usleep(100000); // Small delay
            sem_wait(&chopsticks[right]);
            printf("\tPhilosopher %d picked up right chopstick %d\n", id, right);
        }

        // Eating
        printf("Philosopher %d is eating (meal %d/%d)\n", id, meals_eaten+1, MEALS);
        sleep(1 + rand() % 2);
        meals_eaten++;

        // Release chopsticks
        sem_post(&chopsticks[left]);
        sem_post(&chopsticks[right]);
        printf("\tPhilosopher %d put down chopsticks %d and %d\n", id, left, right);
    }

    printf("Philosopher %d finished all meals and left\n", id);
    return NULL;
}

int main() {
```

```
    pthread_t philosophers[N];
    int ids[N];
    srand(time(NULL));

    // Initialize semaphores
    for (int i = 0; i < N; i++) {
        sem_init(&chopsticks[i], 0, 1);
    }

    // Create philosopher threads
    for (int i = 0; i < N; i++) {
        ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &ids[i]);
    }

    // Wait for all threads to finish
    for (int i = 0; i < N; i++) {
        pthread_join(philosophers[i], NULL);
    }

    // Cleanup
    for (int i = 0; i < N; i++) {
        sem_destroy(&chopsticks[i]);
    }

    printf("All philosophers finished eating. No deadlock occurred.\n");
    return 0;
}
```

OUTPUT:

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





Lab Program-05

Write a C program to simulate Bankers algorithm for the

- **Purpose of deadlock avoidance.**
- **Purpose of deadlock detection.**

Banker's Algorithm:Deadlock Avoidance :

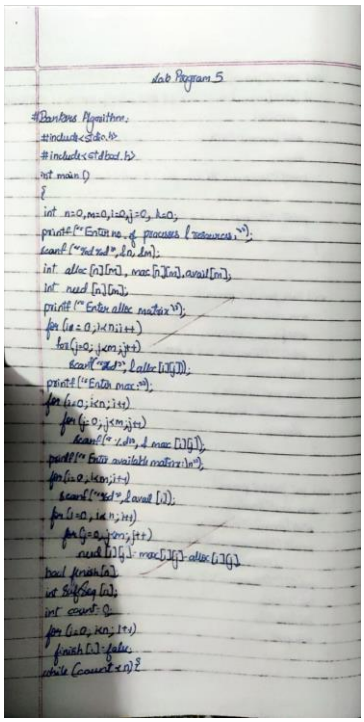
```
#include <stdio.h>
#include <stdbool.h>

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

int main() {
   int n, m; // Number of processes and resources
   int alloc[MAX_PROCESSES][MAX_RESOURCES];
   int max[MAX_PROCESSES][MAX_RESOURCES];
   int avail[MAX_RESOURCES];
   int need[MAX_PROCESSES][MAX_RESOURCES];
   bool finish[MAX_PROCESSES] = {false};
   int safeSeq[MAX_PROCESSES];
   int count = 0;

   // Input number of processes and resources
   printf("Enter number of processes (max %d): ", MAX_PROCESSES);
   scanf("%d", &n);
   printf("Enter number of resources (max %d): ", MAX_RESOURCES);
   scanf("%d", &m);

   // Input allocation matrix
   printf("\nEnter allocation matrix (%d processes x %d resources):\n", n, m);
   for (int i = 0; i < n; i++) {
      printf("Process P%d allocations: ", i);
      for (int j = 0; j < m; j++) {
         scanf("%d", &alloc[i][j]);
      }
   }

   // Input maximum matrix
```

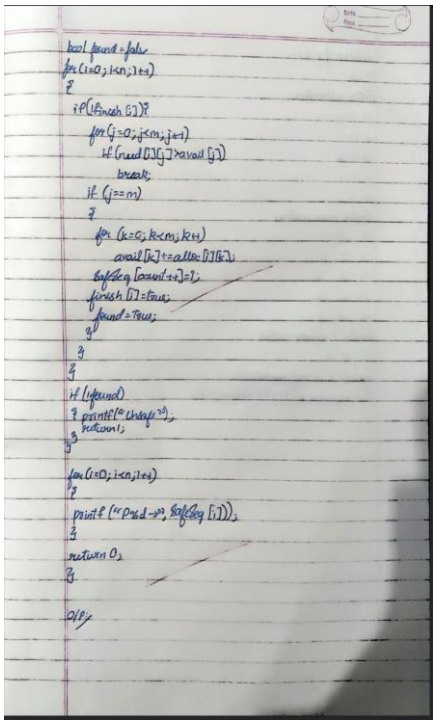
```
printf("\nEnter maximum requirement matrix (%d processes x %d resources):\n", n, m);
for (int i = 0; i < n; i++) {
    printf("Process P%d maximum needs: ", i);
    for (int j = 0; j < m; j++) {
        scanf("%d", &max[i][j]);
    }
}
// Input available resources
printf("\nEnter available resources (%d values): ", m);
for (int i = 0; i < m; i++) {
    scanf("%d", &avail[i]);
}
// Calculate need matrix
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        need[i][j] = max[i][j] - alloc[i][j];
    }
}
// Find safe sequence
while (count < n) {
    bool found = false;
    for (int i = 0; i < n; i++) {
        if (!finish[i]) {
            bool canExecute = true;
            // Check if current process can execute
            for (int j = 0; j < m; j++) {
                if (need[i][j] > avail[j]) {
                    canExecute = false;
                    break;
                }
            }
            if (canExecute) {
                // Release resources after execution
                for (int j = 0; j < m; j++) {
                    avail[j] += alloc[i][j];
                }

                safeSeq[count++] = i;
                finish[i] = true;
                found = true;
```

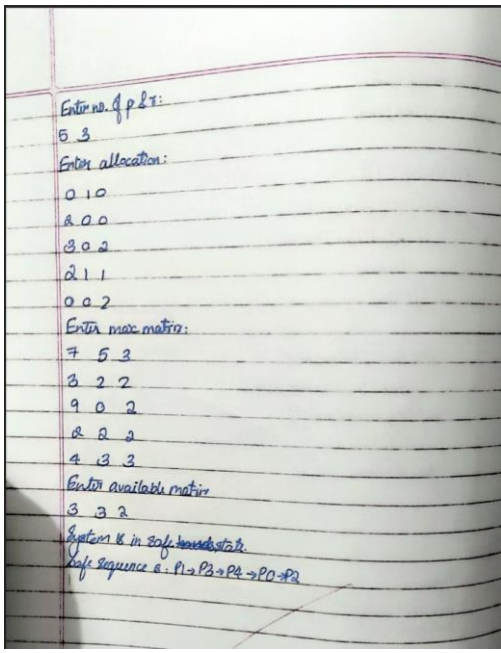
```
                printf("Process P%d can execute. Available resources now: ", i);
                for (int j = 0; j < m; j++) {
                    printf("%d ", avail[j]);
                }
                printf("\n");
            }
        }
    }

    if (!found) {
        printf("\nSystem is NOT in safe state! Deadlock possible.\n");
        return 1;
    }
  }

  // Print safe sequence
  printf("\nSystem is in safe state.\nSafe sequence: ");
  for (int i = 0; i < n; i++) {
    printf("P%d", safeSeq[i]);
    if (i != n-1) printf(" → ");
  }
  printf("\n");

  return 0;
}
```

OUTPUT:

```
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
```







Deadlock detection :

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

int main() {
   int n, m; // Number of processes and resources
   // Get input sizes
   printf("Enter number of processes and resources: ");
   if(scanf("%d %d", &n, &m) != 2 || n <= 0 || m <= 0) {
      printf("Invalid input\n");
      return 1;
   }
   // Arrays for allocation, request, available and finish status
   int alloc[n][m], request[n][m], avail[m];
   bool finish[n];
   // Initialize finish array to false
   for(int i = 0; i < n; i++) {
      finish[i] = false;
```

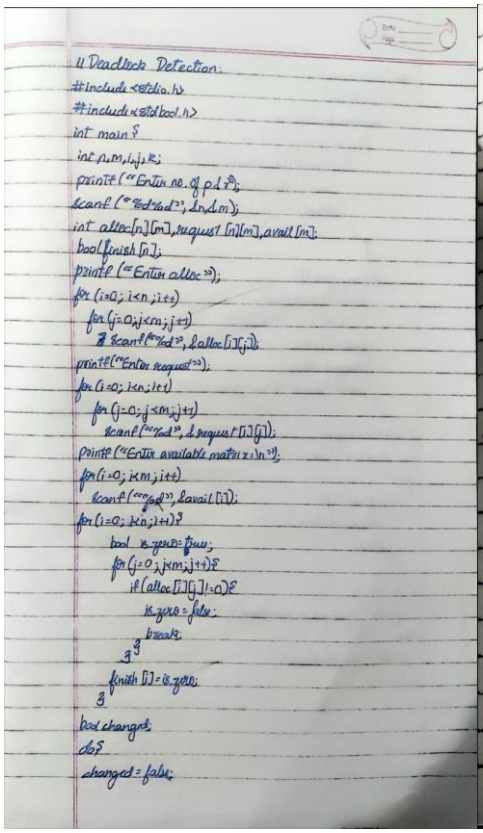
```
}
// Input allocation matrix
printf("Enter allocation matrix:\n");
for(int i = 0; i < n; i++) {
    for(int j = 0; j < m; j++) {
        if(scanf("%d", &alloc[i][j]) != 1) {
            printf("Invalid input\n");
            return 1;
        }
    }
}
// Input request matrix
printf("Enter request matrix:\n");
for(int i = 0; i < n; i++) {
    for(int j = 0; j < m; j++) {
        if(scanf("%d", &request[i][j]) != 1) {
            printf("Invalid input\n");
            return 1;
        }
    }
}
// Input available resources
printf("Enter available resources:\n");
for(int i = 0; i < m; i++) {
    if(scanf("%d", &avail[i]) != 1) {
        printf("Invalid input\n");
        return 1;
    }
}
// CORRECTED: Mark processes with ALL zero allocations as finished
for(int i = 0; i < n; i++) {
    bool all_zero = true;
    for(int j = 0; j < m; j++) {
        if(alloc[i][j] != 0) {
            all_zero = false;
            break;
        }
    }
    finish[i] = all_zero;
}
// Deadlock detection algorithm
```

```
bool changed = true;
int iterations = 0;
const int max_iterations = n; // Prevent infinite loops
while(changed && iterations < max_iterations) {
    changed = false;
    for(int i = 0; i < n; i++) {
        if(!finish[i]) {
            bool can_finish = true;
            // Check if request can be satisfied
            for(int j = 0; j < m; j++) {
                if(request[i][j] > avail[j]) {
                    can_finish = false;
                    break;
                }
            }
            if(can_finish) {
                // Release resources
                for(int j = 0; j < m; j++) {
                    avail[j] += alloc[i][j];
                }
                finish[i] = true;
                changed = true;
                printf("Process P%d can finish\n", i);
            }
        }
    }
    iterations++;
}
// Check for deadlock
bool deadlock = false;
for(int i = 0; i < n; i++) {
    if(!finish[i]) {
        deadlock = true;
        printf("Process P%d is deadlocked\n", i);
    }
}
 if(deadlock) {
    printf("System is in deadlock state\n");
} else {
    printf("System is not in deadlock state\n");
return 0;
```

}

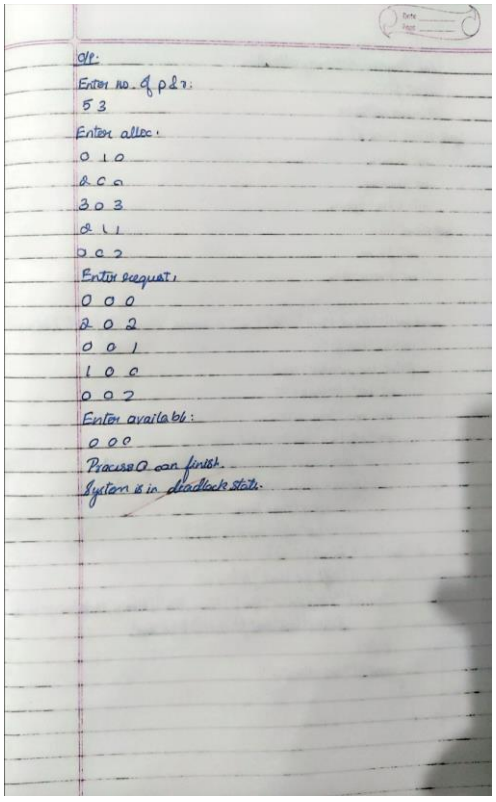
OUTPUT:

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



```
// Deadlock Detection:
#include <stdio.h>
#include <stdbool.h>
int main()
int n,m,i,j,k;
printf("Enter no. of p&r");
scanf("%d%d", &n,&m);
int alloc[n][m], request[n][m], avail[m];
bool finish[n];
printf("Enter alloc");
for(i=0; i<n; i++)
  for(j=0;j<m;j++)
    // scanf("%d", &alloc[i][j]);
printf("Enter request");
for(i=0; i<n; i++)
  for(j=0;j<m;j++)
    scanf("%d", &request[i][j]);
printf("Enter available matrix\n");
for(i=0; i<m; i++)
  scanf("%d", &avail[i]);
for(i=0; i<n; i++){
    bool is_zero=true;
    for(j=0;j<m;j++){
      if(alloc[i][j]!=0){
        is_zero=false;
        break;
      }
    }
    finish[i]=is_zero;
  }
bool changed;
do{
changed=false;
```

```
    for(k=0;k<m;k++)
      avail[k]+=alloc[i][k];
    finish[i]=true;
    changed=true;
    printf("process %d can finish.\n",i);
  }
}
} while(changed);
bool deadlock=false;
for(i=0;i<n;i++){
  if(!finish[i]){
    deadlock=true;
    break;
  }
}
if(deadlock)
  printf("System is in deadlock");
else
  printf("No deadlock");
return 0;
}
```



```
o/p:
Enter no. of p&r:
5 3
Enter alloc:
0 1 0
2 0 0
3 0 3
2 1 1
0 0 2
Enter request:
0 0 0
2 0 2
0 0 1
1 0 0
0 0 2
Enter available:
0 0 0
Process 0 can finish.
System is in deadlock state.
```

Lab Program-06:

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

a) 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;
}
```

OUTPUT:

```
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
```

b) 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;
}
```

OUTPUT:

```
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
```

c) First-fit

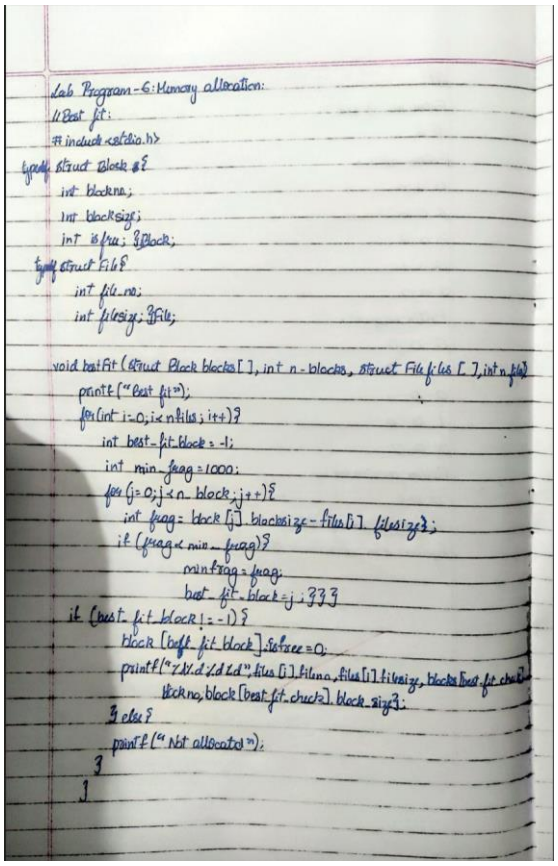
```
#include <stdio.h>

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("\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;
}
```

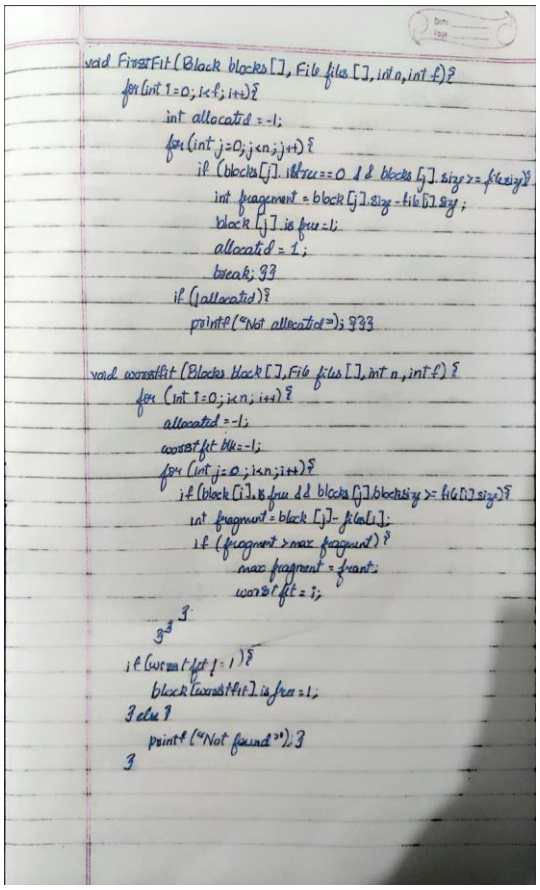
OUTPUT:

```
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
```

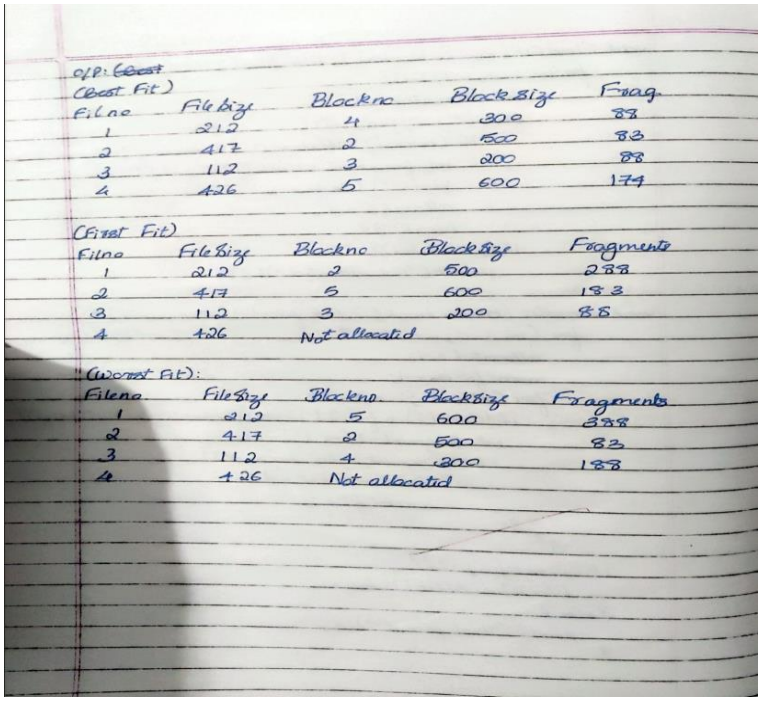


Lab Program-6: Memory allocation:
// Best fit:

```
#include<stdio.h>
typedef struct Block{
    int blockno;
    int blocksize;
    int is_free;}Block;
typedef struct File{
    int file_no;
    int filesize;}File;

void bestFit(struct Block blocks[], int n_blocks, struct File files[], int n_files){
    printf("Best fit:\n");
    for(int i=0;i<n_files;i++){
        int best_fit_block = -1;
        int min_frag=1000;
        for(j=0;j<n_block;j++){
            int frag= block[j].blocksize - files[i].filesize;
            if(frag< min_frag){
                    minfrag=frag;
                    best_fit_block=j;}}
    if(best_fit_block!=-1){
        block[best_fit_block].isfree=0;
        printf("%d %d %d %d",files[i].fileno,files[i].filesize,blocks[best_fit_block].
            blockno,block[best_fit_block].block_size);
    }else{
        printf("Not allocated\n");
    }
}
```



```
void FirstFit(Block blocks[], File files[], int n, int f){
    for(int i=0;i<f;i++){
        int allocated=-1;
        for(int j=0;j<n;j++){
            if(blocks[j].isfree==0 && blocks[j].size>=files[i].size){
                int fragment = block[j].size - file[i].sz;
                block[j].is_free=1;
                allocated=1;
                break;}}
            if(!allocated){
                printf("Not allocated\n");}}

void worstfit(Blocks block[], File files[], int n, int f){
    for(int i=0;i<n;i++){
        allocated=-1;
        worstfit_blk=-1;
        for(int j=0;j<n;j++){
            if(block[i].is_free && blocks[j].blocksize>=file[i].size){
                int fragment=block[j]-file[i];
                if(fragment>max_fragment){
                    max_fragment=frag;
                    worstfit=i;
                }
            }
        }
    if(worstfit!=-1){
        block[worstfit].is_free=1;
    }else{
        printf("Not found");}
}
```



O/P: ~~Best~~

(Best Fit)

File no	File Size	Block no	Block Size	Frag.
1	212	4	300	88
2	417	2	500	83
3	112	3	200	88
4	426	5	600	174

(First Fit)

File no	File Size	Block no	Block Size	Fragments
1	212	2	500	288
2	417	5	600	183
3	112	3	200	88
4	426	Not allocated		

(Worst Fit):

File no.	File Size	Block no.	Block Size	Fragments
1	212	5	600	388
2	417	2	500	83
3	112	4	300	188
4	426	Not allocated		

Lab Program-07:

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

a)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;
}
```

OUTPUT:

```
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
```

b) 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;
}
```

OUTPUT:

```
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
```

c) 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;
}
```

OUTPUT:

```
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
```

Lab-7:

a) FIFO

```
#include<stdio.h>
#
int main(){
  int n,frames,i,j,k, found,indx=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){
```

```
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("\n Page Faults: %d, Page Hits: %d\n",page_faults,hits);
```

```
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;
}
```

O/P:

Enter the size of the pages:

7

Enter the page strings:

1203563

Enter the no of page frames:

3

Optimal Page Faults: 6, Page Hits: 1

O/P:

Enter the size of the pages:

7

Enter the page strings:

1203563

Enter the no. of page frames:

3

FIFO PageFaults: 6, PageHits: 1

b) 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 p:");
  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++){
```

c) Optimal

```
#include<stdio.h>
int main(){
  int n,frames,i,j,k,page_faults=0,hits=0;
  printf("Enter size of p:");
  scanf("%d",&n);
  char page[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;
```