VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

OPERATING SYSTEMS

Submitted by

Trishul (1WA23CS023)

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 Trishul (1WA23CS023), who is Bonafide student of B. M. S. College of Engineering. It is inpartial 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 Seema Patil
Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

Index Sheet

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.

Course outcome

Program 1

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

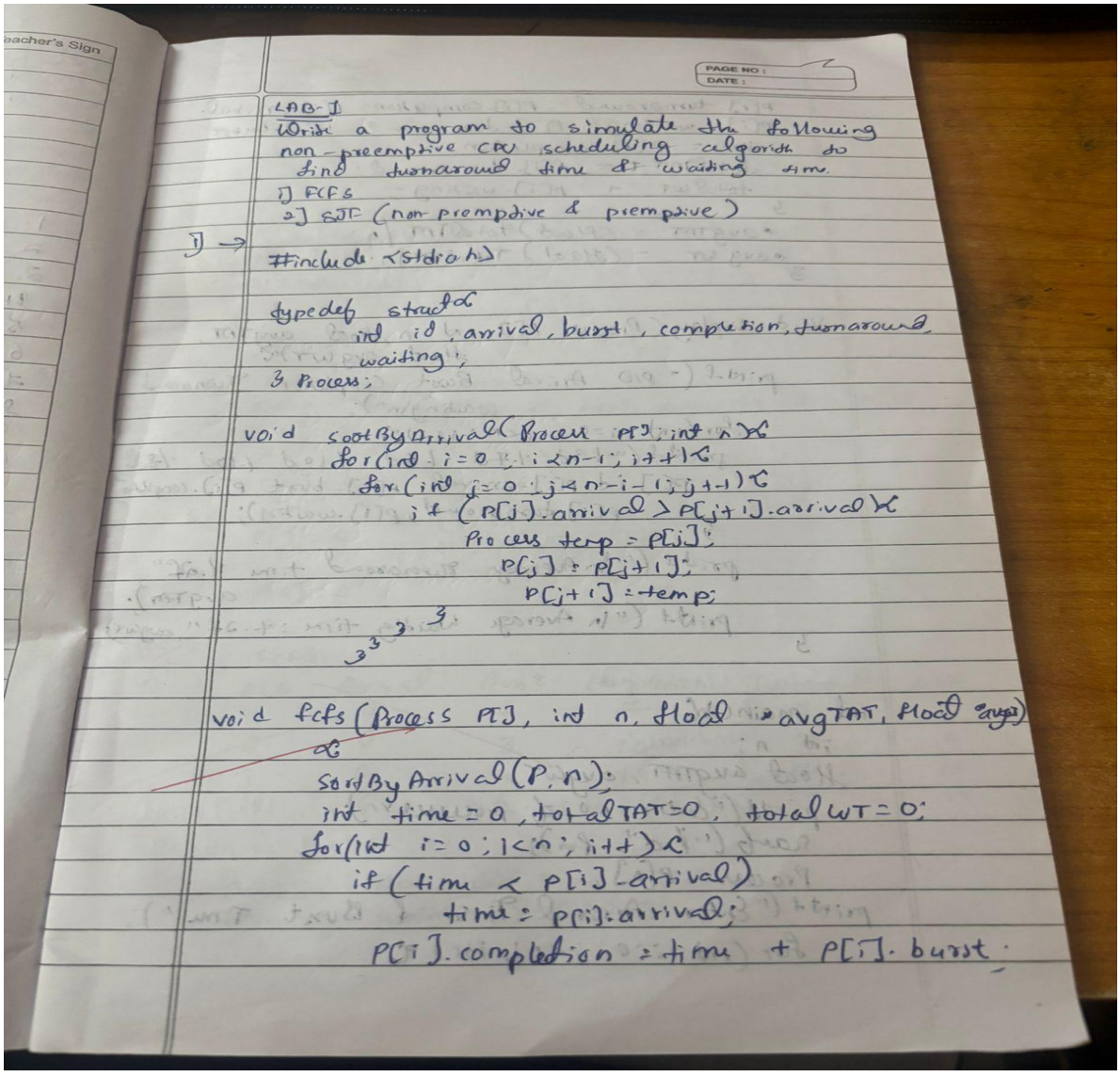
a) First Come First Serve

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

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

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

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) {
                swap(&p[j], &p[j + 1]);
            }
        }
    }
}
```

```
void fcfs(Process p[], int n) {
    sortByArrival(p, n);
    int time = 0;

    for (int i = 0; i < n; i++) {
        if (time < p[i].arrival)
            time = p[i].arrival;

        p[i].response = time - p[i].arrival;
        p[i].completion = time + p[i].burst;
        p[i].turnaround = p[i].completion - p[i].arrival;
        p[i].waiting = p[i].turnaround - p[i].burst;
        time = p[i].completion;
    }
}

void displayResults(Process p[], int n, const char *title) {
    printf("\n--- %s ---\n", title);
    printf("\nPID\tAT\tBT\tCT\tTAT\tWT\tRT\n");

    float totalWT = 0, totalTAT = 0, totalRT = 0;
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n", p[i].id, p[i].arrival, p[i].burst,
p[i].completion, p[i].turnaround, p[i].waiting, p[i].response);
        totalWT += p[i].waiting;
        totalTAT += p[i].turnaround;
        totalRT += p[i].response;
    }
```

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

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

    Process p[n];

    printf("Enter Arrival Time and Burst Time:\n");
    for (int i = 0; i < n; i++) {
        p[i].id = i + 1;
        scanf("%d %d", &p[i].arrival, &p[i].burst);
        p[i].remaining = p[i].burst;
        p[i].waiting = p[i].turnaround = p[i].completion = p[i].response = p[i].started = 0;
    }

    fcfs(p, n);
    displayResults(p, n, "First Come First Serve (FCFS)");

    return 0;
}O/P:
```

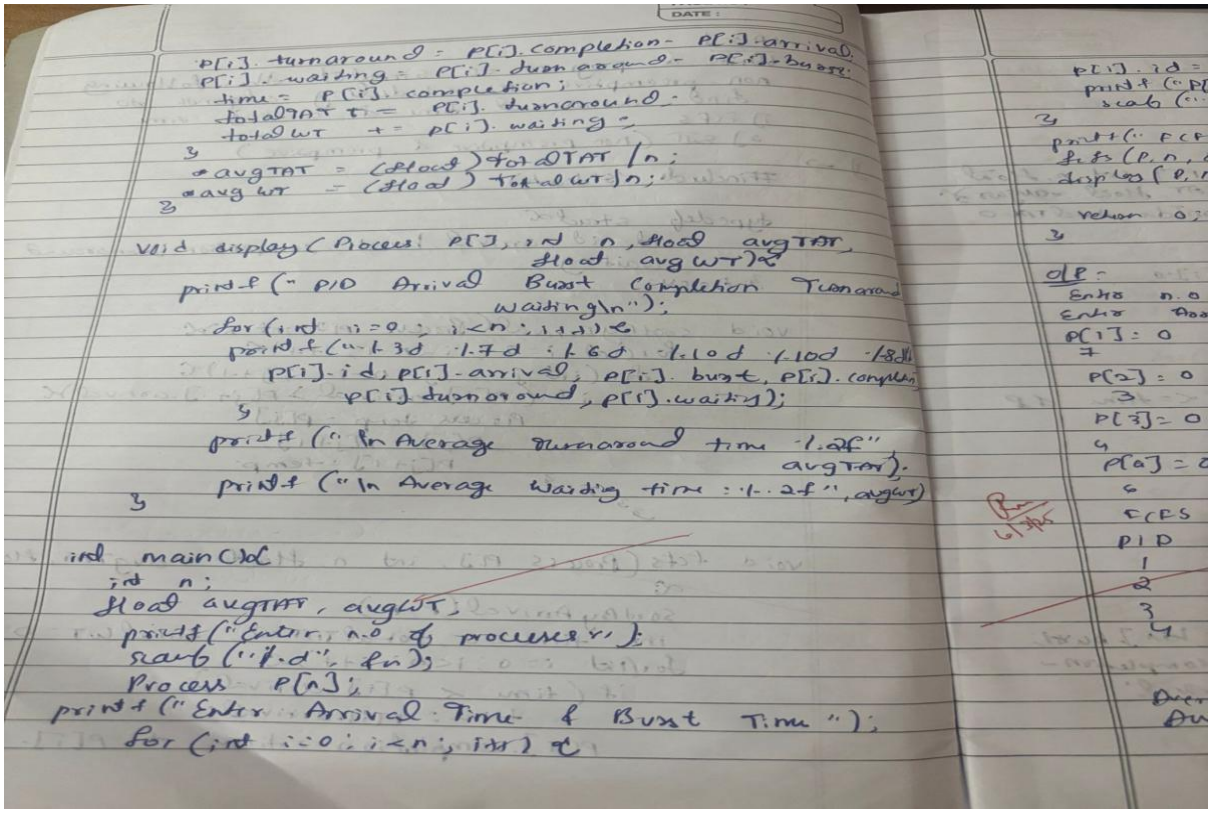
```
Enter number of processes: 4
Enter Arrival Time and Burst Time for each process:
P[1]: 0
7
P[2]: 0
3
P[3]: 0
4
P[4]: 0
6
(FCFS)
PID Arrival Burst Completion Turnaround Waiting
  1       0     7          7          7        0
  2       0     3         10         10        7
  3       0     4         14         14       10
  4       0     6         20         20       14

Average Turnaround Time: 12.75
Average Waiting Time: 7.75
```



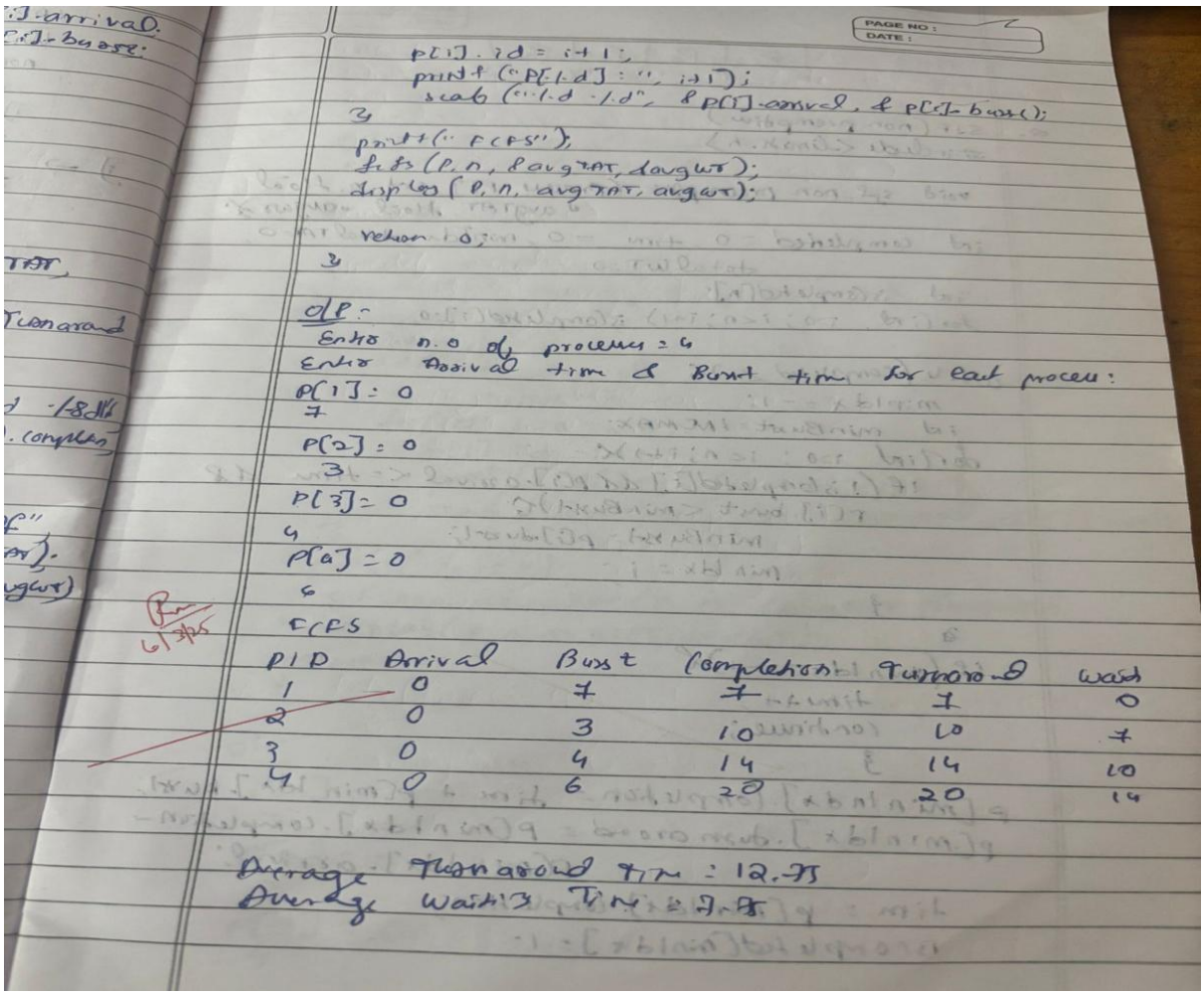
LAB-1
Write a program to simulate the following non-preemptive CPU scheduling algorithm to find turnaround time & waiting time.
1] FCFS
2] SJF (non prempdive & prempsive)

1] →

```
#include <stdio.h>

typedef struct{
    int id, arrival, burst, 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 fcfs(Process P[], int n, float *avgTAT, float *avgWT)
{
    sortByArrival(P, n);
    int time = 0, totalTAT=0, totalWT=0;
    for(int i=0; i<n; i++){
        if (time < P[i].arrival)
            time = P[i].arrival;
        P[i].completion = time + P[i].burst;
```



```
p[i].turnaround = p[i].completion - p[i].arrival;
p[i].waiting = p[i].turnaround - p[i].burst;
time = p[i].completion;
totalTAT += p[i].turnaround;
totalWT += p[i].waiting;
}
avgTAT = (float) totalTAT / n;
avgWT = (float) totalWT / n;
}

void display(Process p[], int n, float avgTAT,
             float avgWT){
printf("PID Arrival Burst Completion Turnaround
        Waiting\n");
for(int i=0; i<n; i++){
printf("%3d %7d %7d %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", avgWT);
}

int main(){
int n;
float avgTAT, avgWT;
printf("Enter no. of processes:");
scanf("%d", &n);
Process p[n];
printf("Enter Arrival Time & Burst Time");
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("FCFS");
findTs(p, n, &avgTAT, &avgWT);
display(p, n, avgTAT, avgWT);

return 0;
}
```

O/P:

```
Enter no. of process: 4
Enter Arrival time & Burst time for each process:
P[1]: 0
7
P[2]: 0
3
P[3]: 0
4
P[4]: 0
6
FCFS
```

PID	Arrival	Burst	Completion	Turnaround	Wait
1	0	7	7	7	0
2	0	3	10	10	7
3	0	4	14	14	10
4	0	6	20	20	14

Average Turnaround Time: 12.75
Average Waiting Time: 7.75

Program 1

b) Shortest Job First (non preemptive)

Programme:

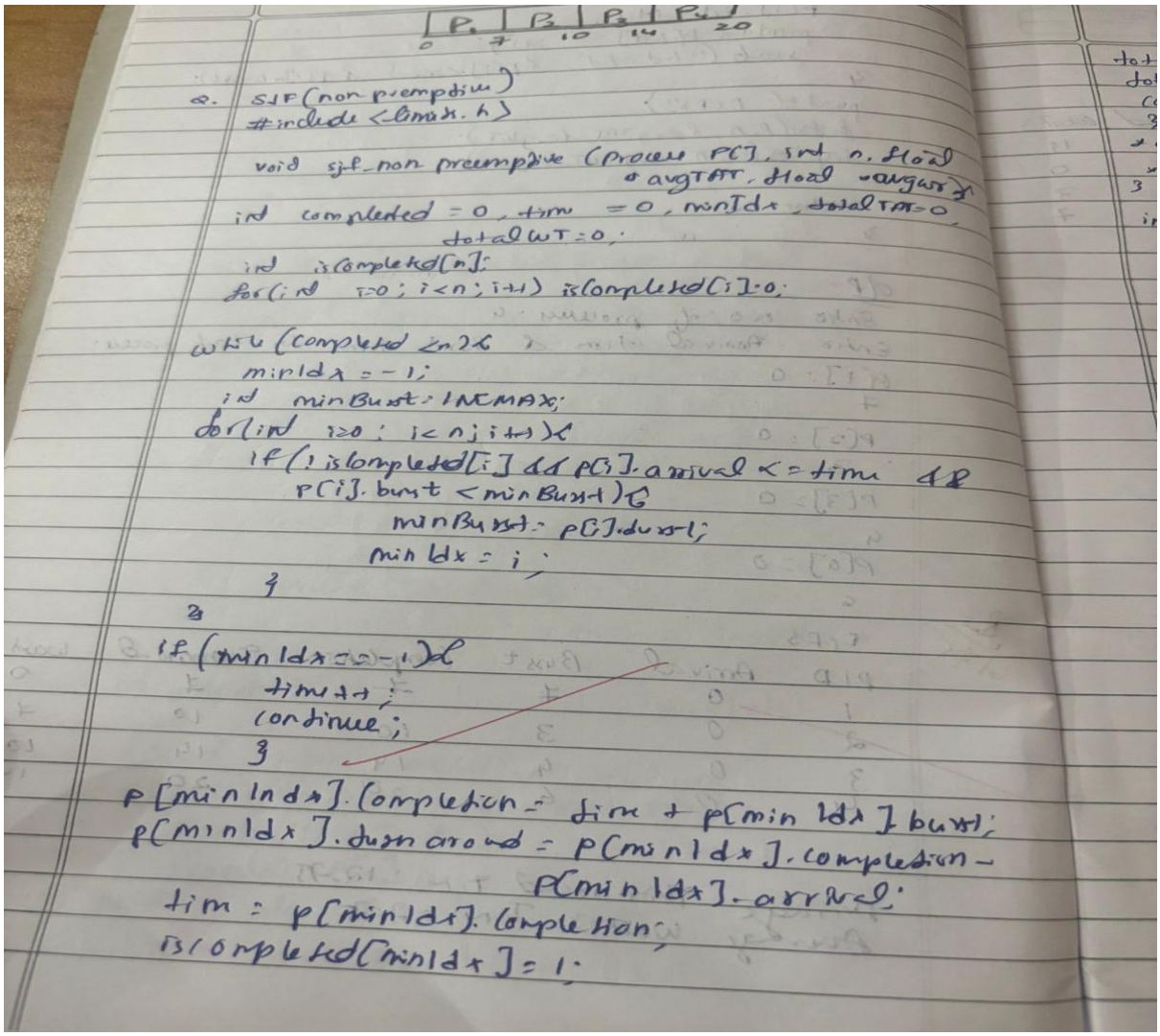
```
#include <stdio.h>
#include <limits.h>
typedef struct {
    int id, arrival, burst, completion, turnaround, waiting;
} Process;

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;
            }
        }
    }
}

// SJF Scheduling (Non-Preemptive)
void sjf_non_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;
```

```
    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].burst < minBurst) {
                minBurst = p[i].burst;
                minIdx = i;
            }
        }
        if (minIdx == -1) { time++; continue; }

        p[minIdx].completion = time + p[minIdx].burst;
        p[minIdx].turnaround = p[minIdx].completion - p[minIdx].arrival;
        p[minIdx].waiting = p[minIdx].turnaround - p[minIdx].burst;
        time = p[minIdx].completion;
        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 (Non-Preemptive) Scheduling\n");
    sjf_non_preemptive(p, n, &avgTAT, &avgWT);
    display(p, n, avgTAT, avgWT);

    return 0;
}
```

O/p:

```
Enter number of processes: 4
Enter Arrival Time and Burst Time for each process:
P[1]: 0
7
P[2]: 0
3
P[3]: 0
4
P[4]: 0
6

Shortest Job First (Non-Preemptive) Scheduling

PID Arrival Burst Completion Turnaround Waiting
  1       0     7         20         20       13
  2       0     3          3          3        0
  3       0     4          7          7        3
  4       0     6         13         13        7

Average Turnaround Time: 10.75
Average Waiting Time: 5.75
```



```
totalTAT += p[minIdx].turnaround;
totalWT += p[minIdx].waiting;
completed++;
}
avgTAT = (float) totalTAT / n;
avgWT = (float) totalWT / n;
}
int main() {
    int n;
    float avgTAT, avgWT;
    printf("Enter number of process : ");
    scanf("%d", &n);
    Process p[n];
    printf("Enter Arrival & Burst time for each
            process");
    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("SJF");
    sjf_non_preemptive(p, n, &avgTAT, &avgWT);
    display(p, n, avgTAT, avgWT);
    return 0;
}
```

o/p

```
p[1] : 0
7
p[2] : 0
3
p[3] : 0
4
```

```
p[4] : 0
6
```

PID	Arrival	Burst	Completion	Turnaround	Waiting
1	0	7	20	20	13
2	0	3	3	3	0
3	0	4	7	7	3
4	0	6	13	13	7

Average Turnaround Time : 10.75
Average Waiting Time : 5.75

P_2	P_3	P_4	P_1

0 3 7 13 20

Program 1

c) Shortest Job First (preemptive)

Programme:

```
#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;
```

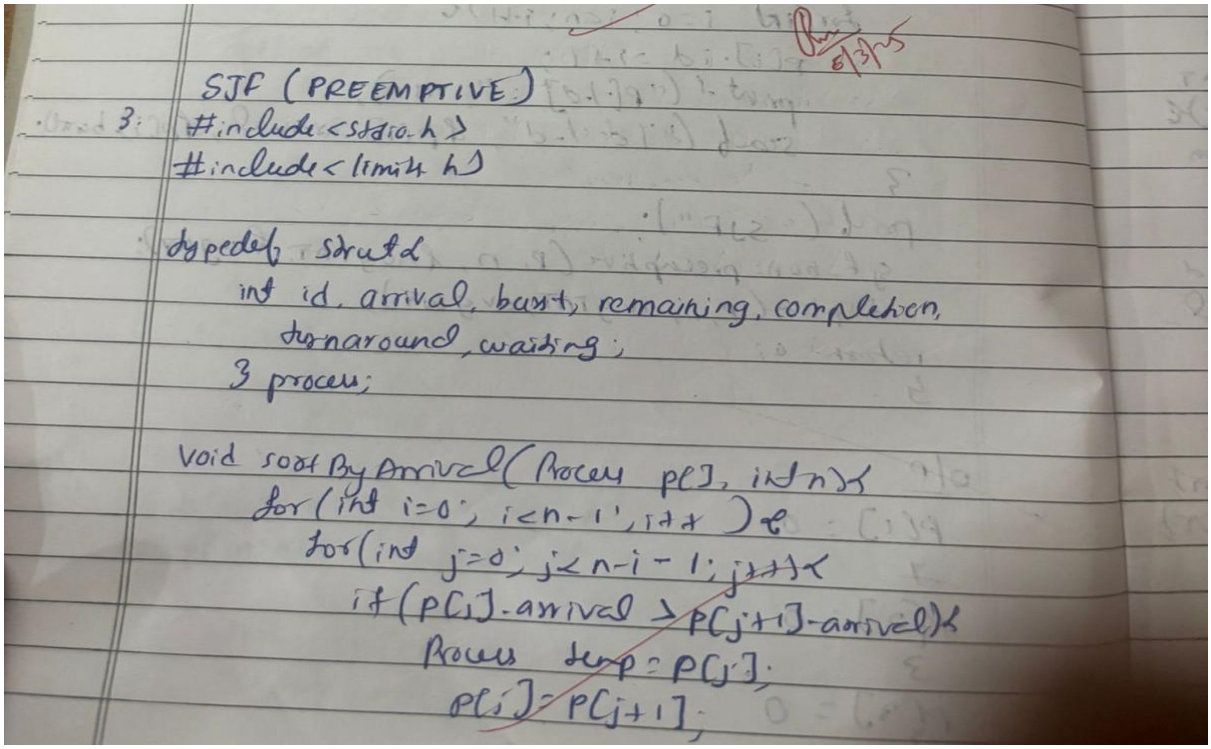
```
    }

    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;
}
```

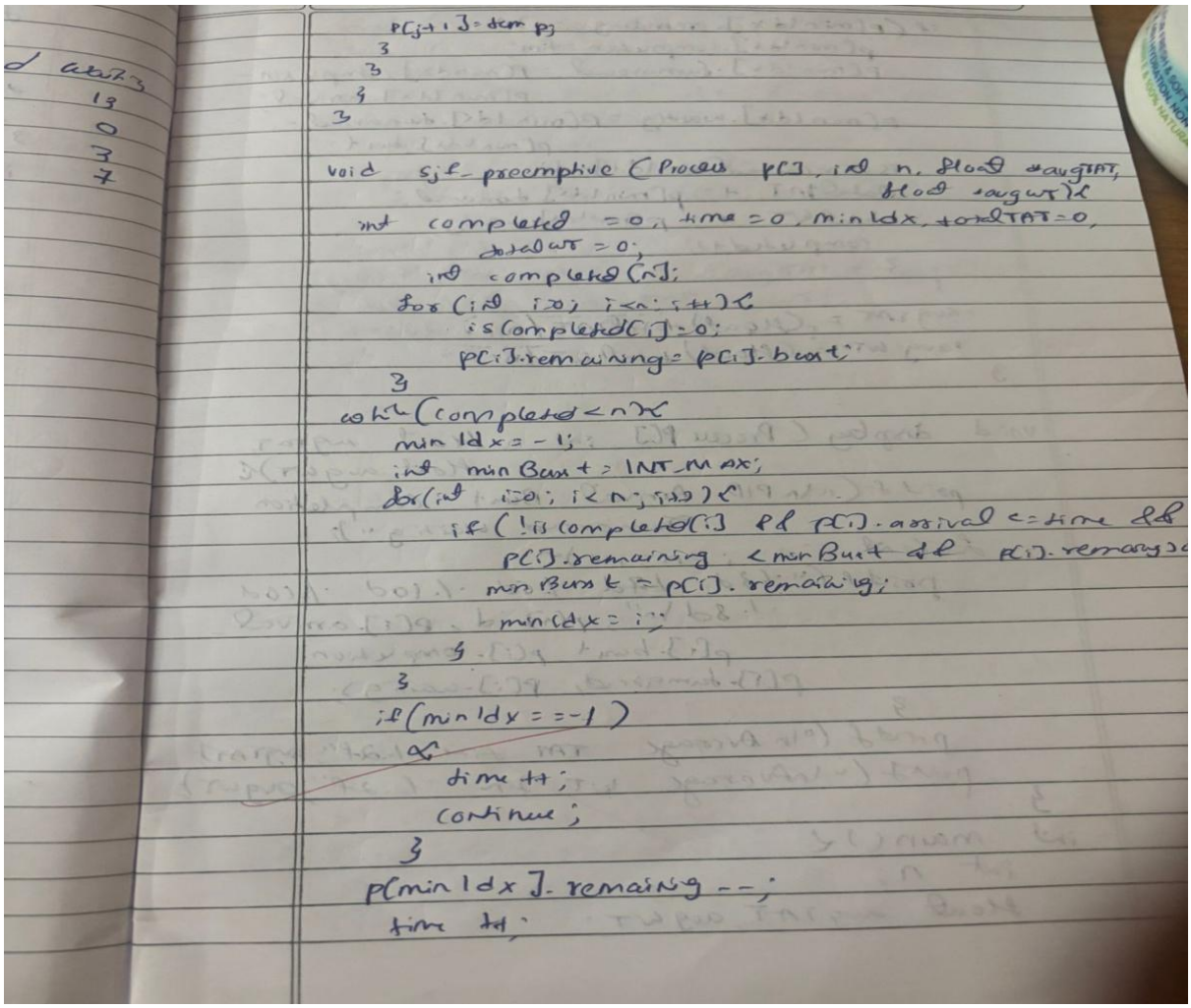
O/P:

```
PS C:\Users\trish\OneDrive\Desktop\OS_LAB> gcc -o sjf2 sjf2.c
PS C:\Users\trish\OneDrive\Desktop\OS_LAB> ./sjf2.exe
Enter number of processes: 4
Enter Arrival Time and Burst Time for each process:
P[1]: 0
2
P[2]: 1
6
P[3]: 2
8
P[4]: 3
4

Shortest Job First (Preemptive) Scheduling

PID Arrival Burst Completion Turnaround Waiting
  1       0     2          2          2        0
  2       1     6         12         11        5
  3       2     8         20         18       10
  4       3     4          7          4        0

Average Turnaround Time: 8.75
Average Waiting Time: 3.75
```

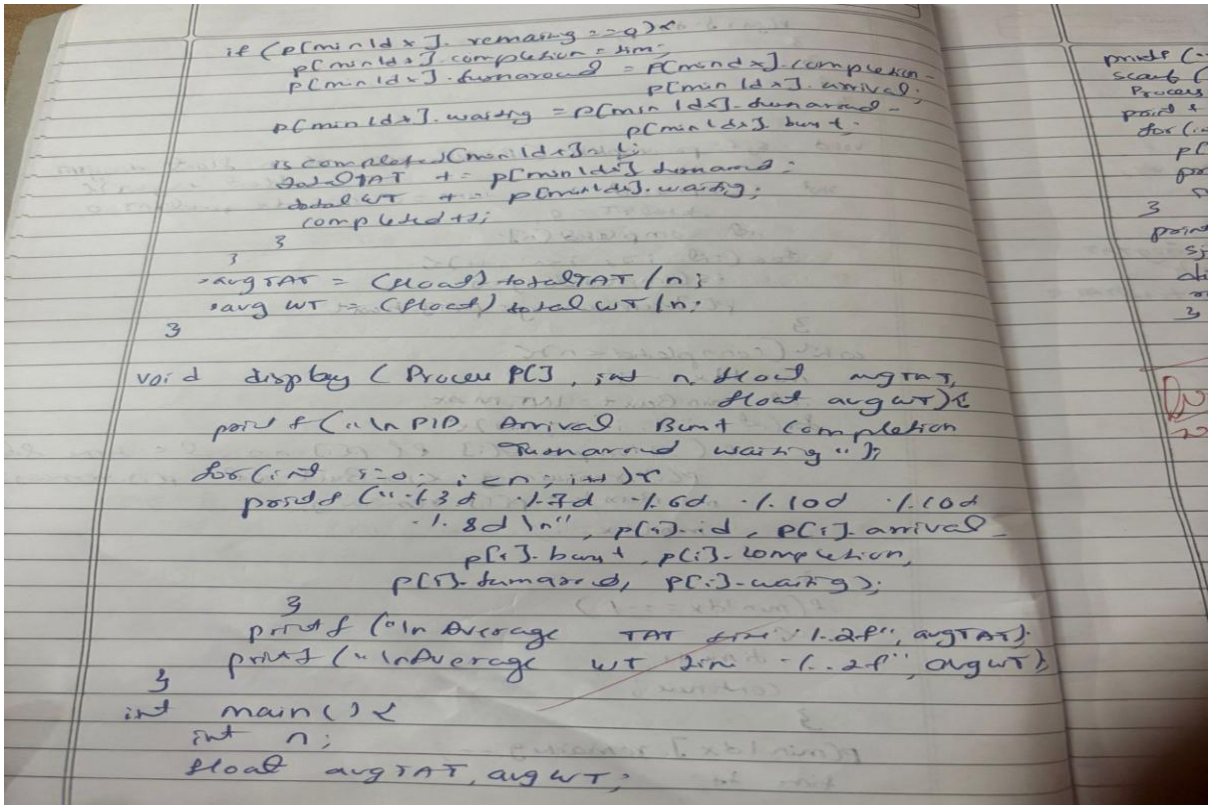


SJF (PREEMPTIVE)

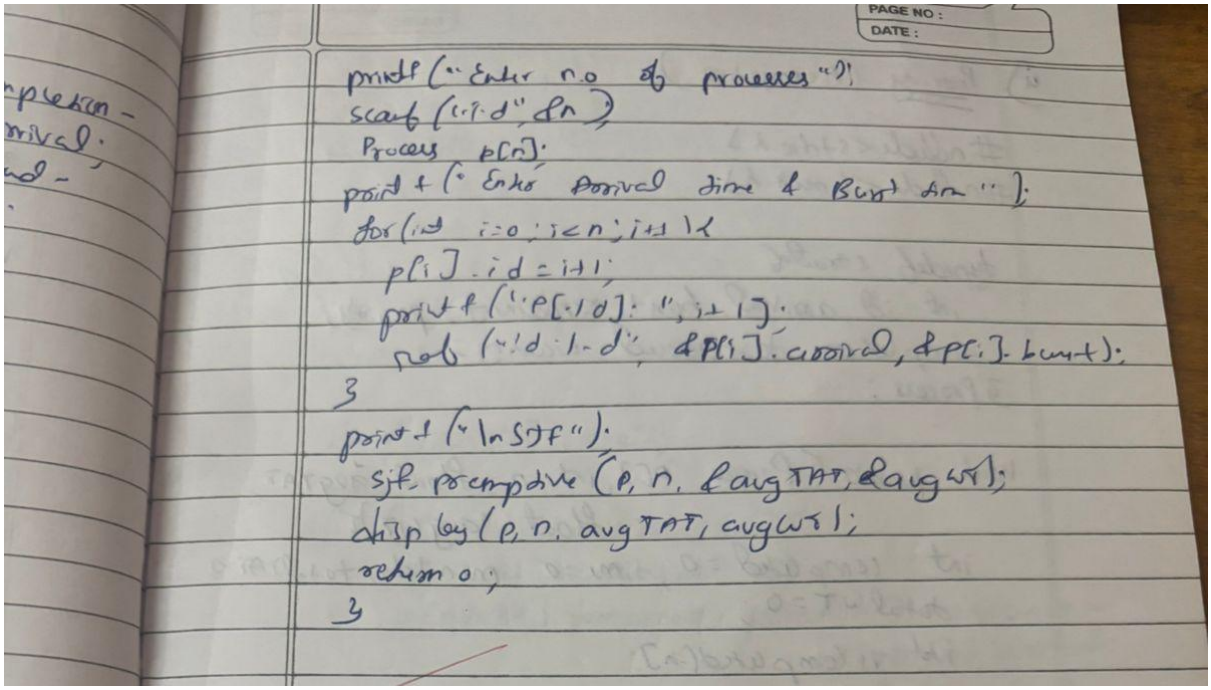
```
3:  #include <stdio.h>
    #include <limits.h>

    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[minId].remaining == 0){
    p[minId].completion = time;
    p[minId].turnaround = p[minId].completion -
                          p[minId].arrival;
    p[minId].waiting = p[minId].turnaround -
                       p[minId].burst;
    isCompleted[minId] = 1;
    totalTAT += p[minId].turnaround;
    totalWT += p[minId].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");
  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 TAT time: %.2f", avgTAT);
  printf("\nAverage WT time: %.2f", avgWT);
}
int main(){
  int n;
  float avgTAT, avgWT;
```



```
printf("Enter no of processes");
scanf("%d", &n);
Process p[n];
printf("Enter Arrival time & Burst time");
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("\nSJF");
sjf_preemptive(p, n, &avgTAT, &avgWT);
display(p, n, avgTAT, avgWT);
return 0;
}
```

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)

Programme:

```
#include <stdio.h>

#define MAX 100

void priorityPreemptive(int n, int at[], int bt[], int pr[]) {
    int ct[n], tat[n], wt[n], rem_bt[n], is_completed[n];
    int time = 0, completed = 0, min_priority, index;

    for (int i = 0; i < n; i++) {
        rem_bt[i] = bt[i];
        is_completed[i] = 0;
    }

    while (completed < n) {
        min_priority = 9999;
        index = -1;

        for (int i = 0; i < n; i++) {
            if (at[i] <= time && is_completed[i] == 0 && pr[i] < min_priority && rem_bt[i] > 0)
{
                min_priority = pr[i];
                index = i;
            }
        }
```

```
        if (index == -1) {
            time++;
        } else {
            rem_bt[index]--;
            time++;

            if (rem_bt[index] == 0) {
                ct[index] = time;
                is_completed[index] = 1;
                completed++;
            }
        }
    }

    float total_tat = 0, total_wt = 0;
    printf("\nP#\tAT\tBT\tPR\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\t%d\n", i + 1, at[i], bt[i], pr[i], ct[i], tat[i], wt[i]);
    }

    printf("Average TAT: %.2f\n", total_tat / n);
    printf("Average WT: %.2f\n", total_wt / n);
}
```

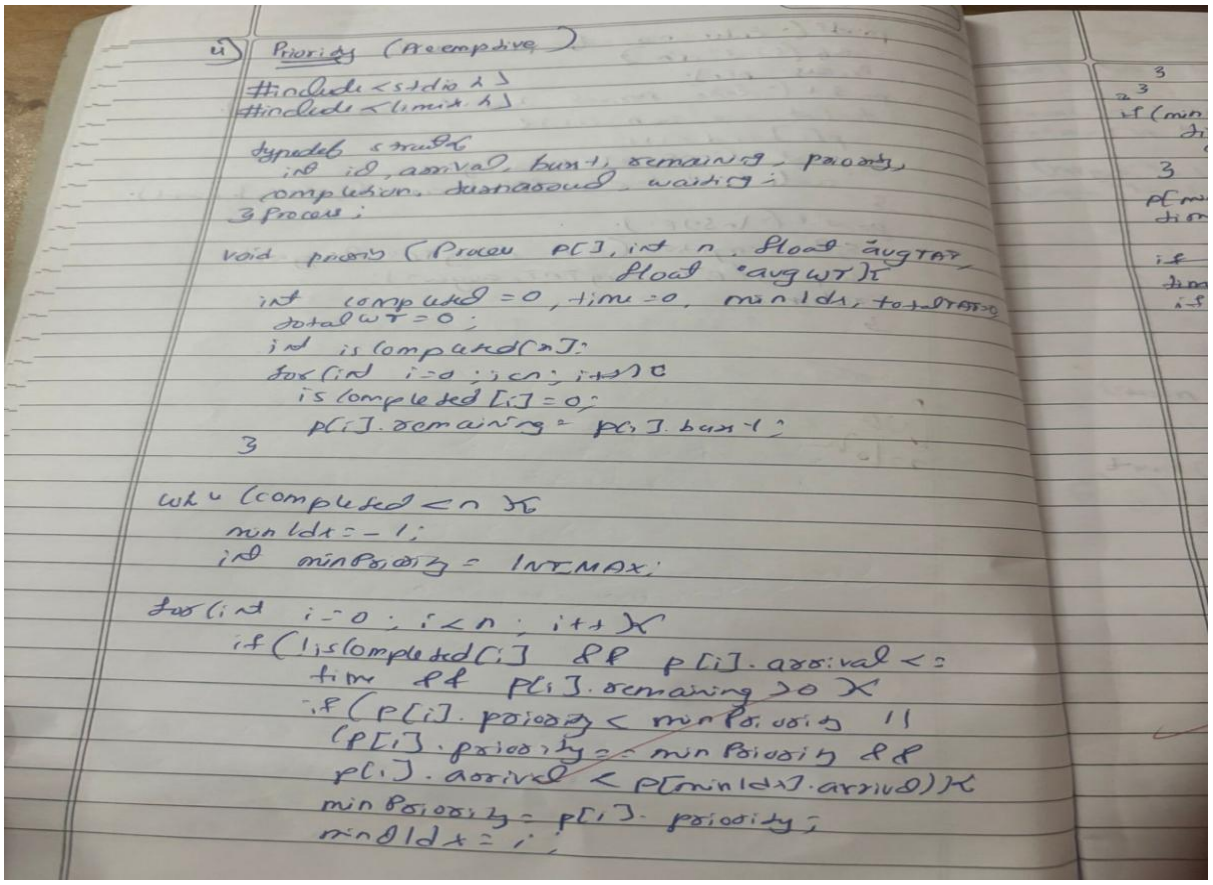
```
void priorityNonPreemptive(int n, int at[], int bt[], int pr[]) {
    int ct[n], tat[n], wt[n], is_completed[n], rem_bt[n];
    int time = 0, completed = 0;

    for (int i = 0; i < n; i++) {
        is_completed[i] = 0;
        rem_bt[i] = bt[i];
    }

    while (completed < n) {
        int min_priority = 9999, index = -1;

        for (int i = 0; i < n; i++) {
            if (at[i] <= time && is_completed[i] == 0 && pr[i] < min_priority) {
                min_priority = pr[i];
                index = i;
            }
        }

        if (index == -1) {
            time++;
        } else {
            time += bt[index];
            ct[index] = time;
            is_completed[index] = 1;
            completed++;
        }
    }
```

```
    float total_tat = 0, total_wt = 0;
    printf("\nP#\tAT\tBT\tPR\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\t%d\n", i + 1, at[i], bt[i], pr[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, choice;
    printf("Enter number of processes: ");
    scanf("%d", &n);

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

    printf("\nChoose Scheduling Algorithm:\n");
    printf("1. Preemptive Priority Scheduling\n");
    printf("2. Non-Preemptive Priority Scheduling\n");
    printf("Enter choice: ");
```

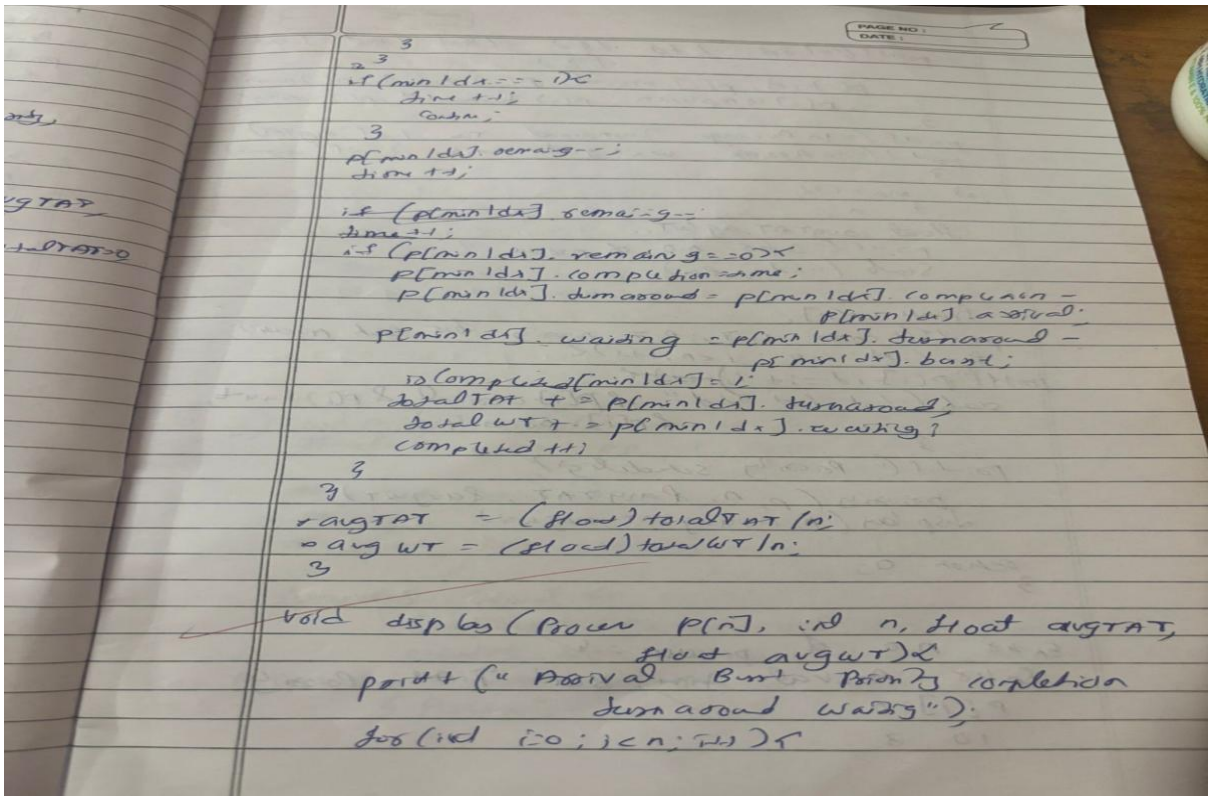
```
    scanf("%d", &choice);

    if (choice == 1) {
        priorityPreemptive(n, at, bt, pr);
    } else if (choice == 2) {
        priorityNonPreemptive(n, at, bt, pr);
    } else {
        printf("Invalid choice!\n");
    }

    return 0;
}
```

O/P:

```
PS C:\Users\trish\OneDrive\Desktop\OS_LAB> gcc priority.c -o priority
PS C:\Users\trish\OneDrive\Desktop\OS_LAB> ./priority
Enter number of processes: 5
Enter AT, BT, and Priority P1: 0
10
3
Enter AT, BT, and Priority P2: 0
1
1
Enter AT, BT, and Priority P3: 3
2
3
Enter AT, BT, and Priority P4: 5
1
4
Enter AT, BT, and Priority P5: 10
5
2

Choose Scheduling Algorithm:
1. Preemptive Priority Scheduling
2. Non-Preemptive Priority Scheduling
Enter choice: 2

P#      AT      BT      PR      CT      TAT     WT
1       0       10      3       11      11      1
2       0       1       1       1       1       0
3       3       2       3       18      15      13
4       5       1       4       19      14      13
5       10      5       2       16      6       1
Average TAT: 9.40
Average WT: 5.60
```



```
4) Priority (Preemptive)

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

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

void priority (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 minPriority = INT_MAX;

for(int i=0 ; i<n ; i++){
    if(!isCompleted[i] && p[i].arrival <=
        time && p[i].remaining >0 ){
      if(p[i].priority < minPriority ||
      (p[i].priority == minPriority &&
      p[i].arrival < p[minIdx].arrival)){
      minPriority = p[i].priority;
      minIdx = i;
```



```
  }
}
if(minIdx == -1){
    time ++;
    continue;
}
p[minIdx].remaining--;
time ++;

if (p[minIdx].remaining==0){
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("\n Process Arrival Burst Priority Completion
          Turnaround Waiting\n");
  for(int i=0;i<n;i++){
```

```
printf("%3d %7d %6d %8d %10d %[illegible]
%8d",
    p[i].id, p[i].arrival, p[i].burst, p[i].priority,
    p[i].completion, p[i].turnaround, p[i].waiting);
}
printf("\n Average Turnaround Time : %.2f", avgTAT);
printf("\n Average waiting time time : %.2f", avgWT);
}
int main(){
  int n;
  float avgTAT, avgWT;
  printf("Enter no of process : ");
  scanf("%d", &n);

  Process p[n];
  printf("Enter A.T, B.T, Priority for each process");
  for(int i=0 ; i<n; i++){
printf("P[i].id = i+ 1);
  scanf("%d %d %d", &p[i].arrival, &p[i].burst,
        &p[i].priority);
  }
  printf("Priority scheduling");
    priority(p, n, &avgTAT, &avgWT);
  display(p, n, avgTAT, avgWT);

  return 0;
}
```

Enter no of process : 4
Enter Arrival time, Burst Time, Priority:
P[1]: 0
10 3
P[2]: 0 1 1
P[3]: 3 2 3
P[4]: 5 1 4

Preemptive Priority scheduling

PID	Arrival	Burst	Priority	Completion	TAT	WT
1	0	10	3	11	11	1
2	0	1	1	1	1	0
3	3	2	3	13	10	8
4	5	1	4	14	9	8

Avg TAT time : 7.75
Avg WT : 4.25

Program 2

Round Robin

Program:

```
#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;
```

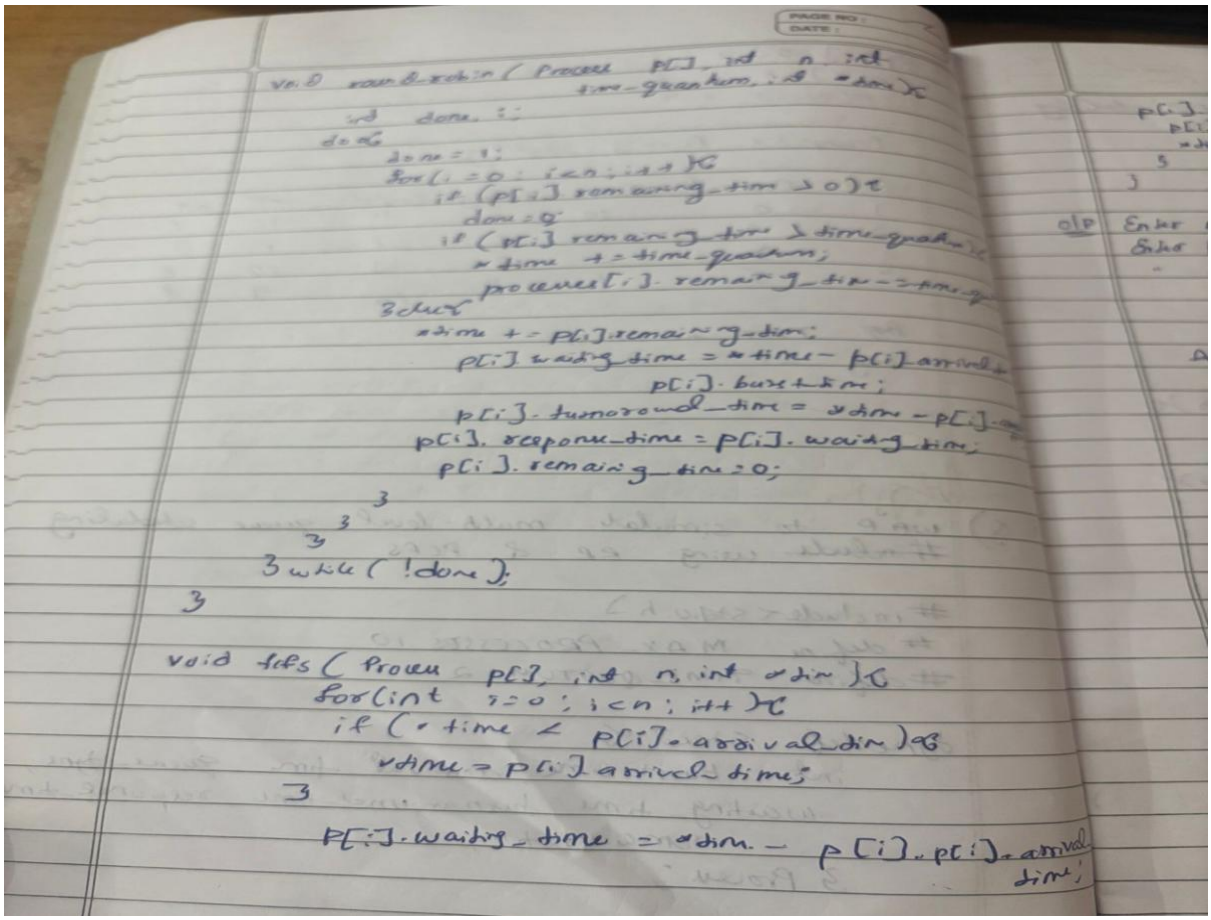
}

O/P:

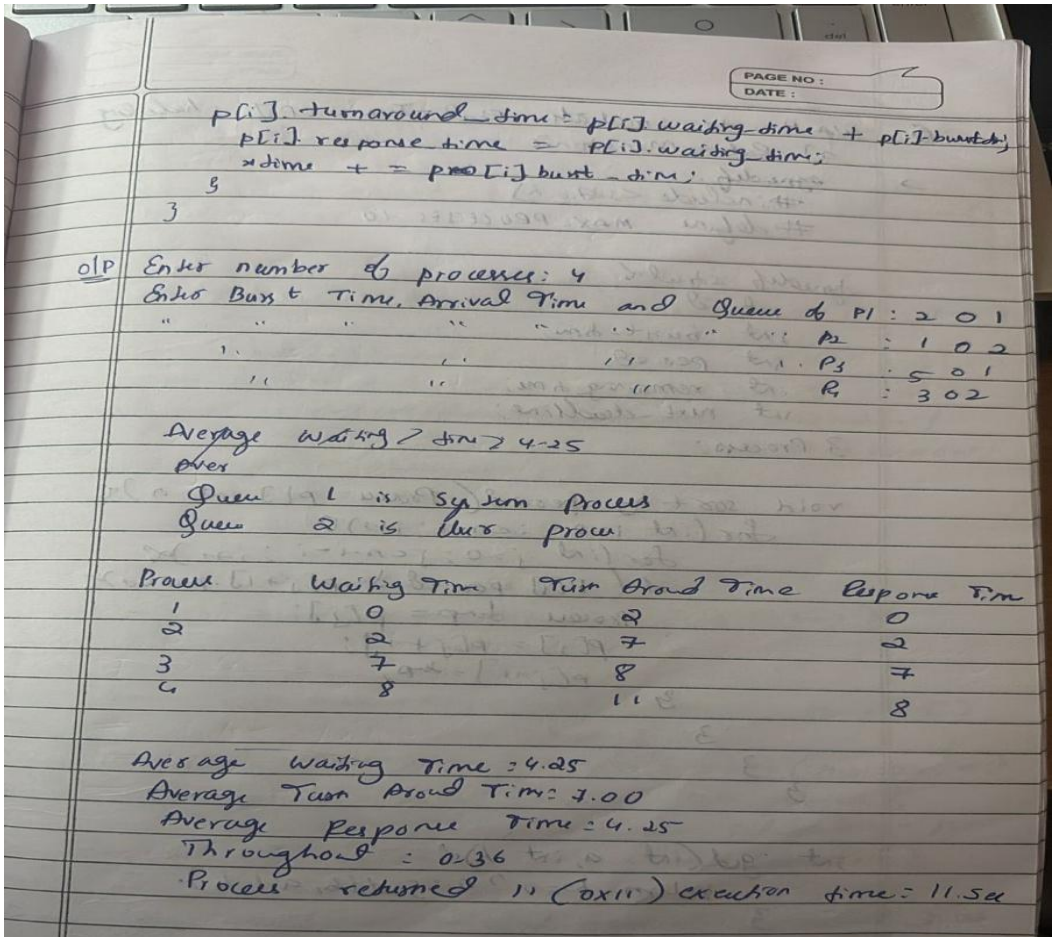
```
PS C:\Users\trish\OneDrive\Desktop\OS_LAB> gcc Round.c -o Round
PS C:\Users\trish\OneDrive\Desktop\OS_LAB> ./Round
Enter number of processes: 5
Enter AT and BT for process 1: 0
8
Enter AT and BT for process 2: 5
2
Enter AT and BT for process 3: 1
7
Enter AT and BT for process 4: 6
3
Enter AT and 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
PS C:\Users\trish\OneDrive\Desktop\OS LAB>
```

5 Q) WAP to simulate multi-level queue scheduling using RR & FCFS

```
#include<stdio.h>
#define MAX_PROCESSES 10
#define TIME_QUATUM 2

typedef struct {
    int burst_time, arrival_time, queue_type,
    waiting_time, turnaround_time, response_time,
    remaining_time;
} Process;
```



```
void rr(Process p[], int n, int *time, int time_quantum, int *done) {
    int done;
    do {
        done = 1;
        for(i=0; i<n; i++) {
            if (p[i].remaining_time > 0) {
                done = 0;
                if (p[i].remaining_time > time_quantum) {
                    *time += time_quantum;
                    process[i].remaining_time -= time_quantum;
                } else {
                    *time += p[i].remaining_time;
                    p[i].waiting_time = *time - p[i].arrival_time
                                        - p[i].burst_time;
                    p[i].turnaround_time = *time - p[i].arrival_time;
                    p[i].response_time = p[i].waiting_time;
                    p[i].remaining_time = 0;
                }
            }
        }
    } while (!done);
}

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



p[i].turnaround_time = p[i].waiting_time + p[i].burst_time;
p[i].response_time = p[i].waiting_time;
*time += p[i].burst_time;
}
}

O/P Enter number of processes: 4
Enter Burst Time, Arrival Time and Queue of P1 : 2 0 1
" " " " " " P2 : 1 0 2
" " " " " " P3 : 5 0 1
" " " " " " P4 : 3 0 2

Average Waiting Time = 4.25
Over
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 : 11.5s

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 RR and FCFS scheduling for the processes in each queue.

Program:

```
#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;
```

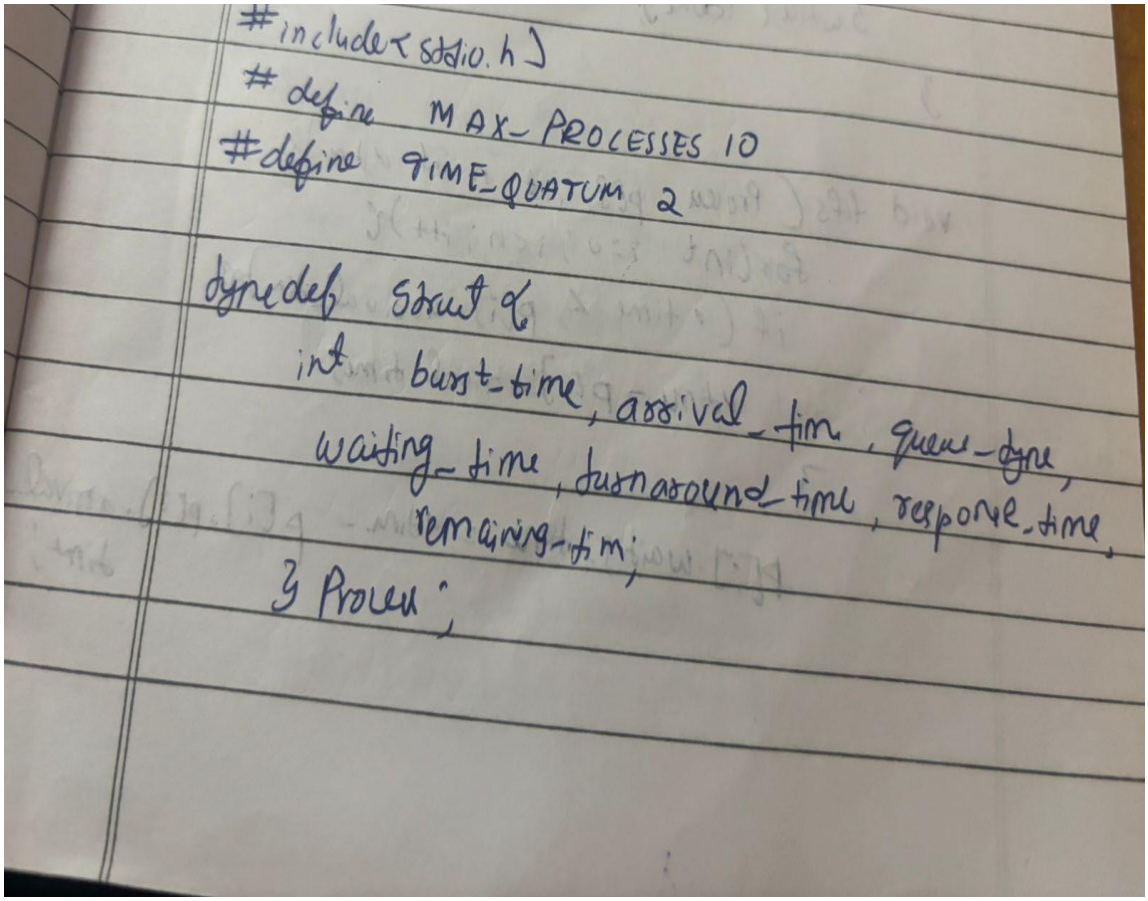
```
                processes[i].response_time = processes[i].waiting_time;
                processes[i].remaining_time = 0;
            }
        }
    }
  } while (!done);
}

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);
```

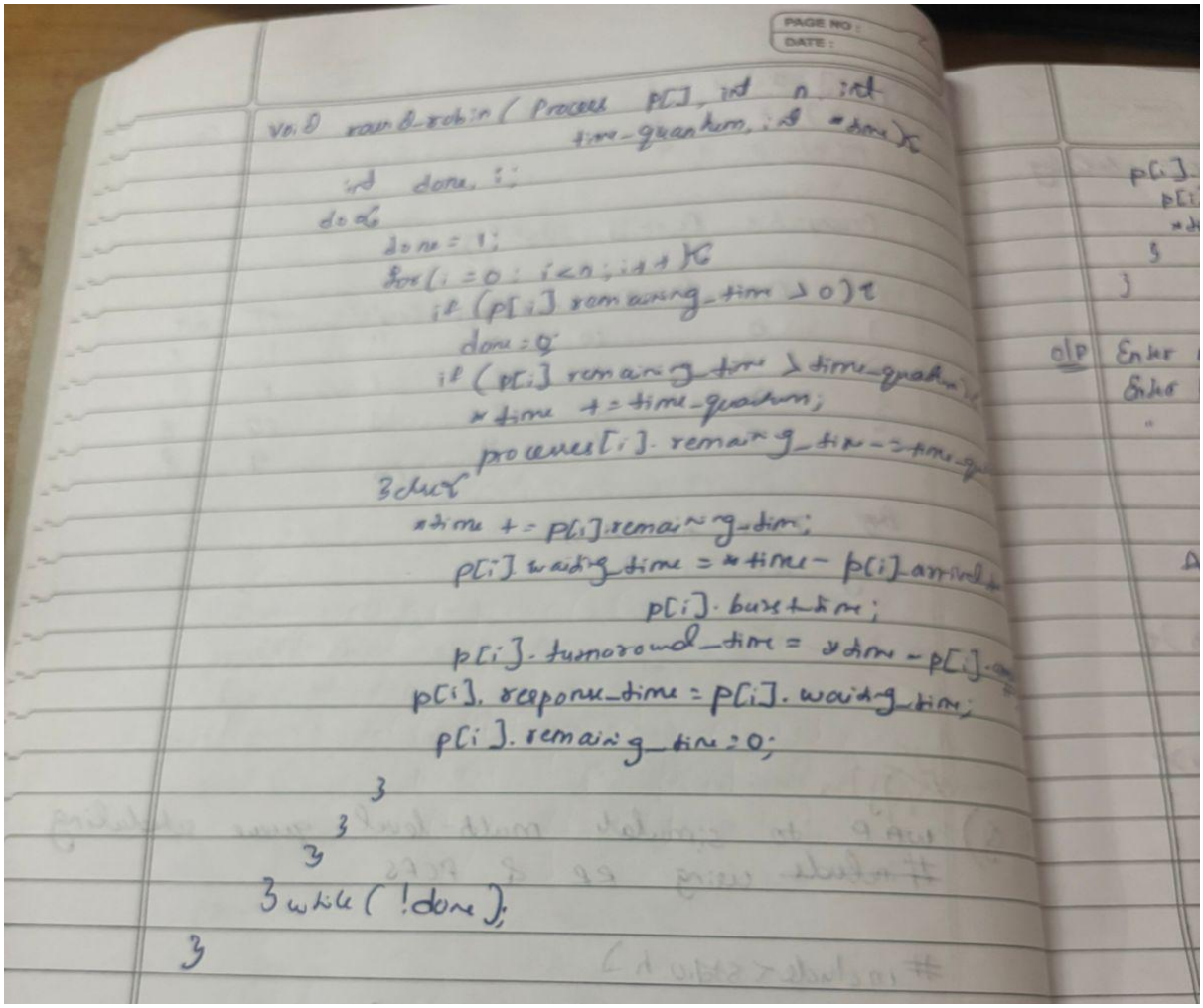
```
    for (int i = 0; i < n; i++) {
        printf("Enter Burst Time, Arrival Time and Queue of P%d: ", i + 1);
        scanf("%d %d %d", &processes[i].burst_time, &processes[i].arrival_time,
&processes[i].queue_type);
        processes[i].remaining_time = processes[i].burst_time;

        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;
}
```

O/P:

```
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
PS C:\Users\Admin\Desktop\1wa23cs023>
```





```
void round_robin(Process p[], int n, int
                 time_quantum, int *time)
{
    int done, i;
    do {
        done = 1;
        for(i=0; i<n; i++){
            if (p[i].remaining_time > 0){
                done = 0;
                if (p[i].remaining_time > time_quantum){
                    *time += time_quantum;
                    process[i].remaining_time -= time_q
                }else{
                    *time += p[i].remaining_time;
                    p[i].waiting_time = *time - p[i].arrival_t
                                        p[i].burst_time;
                    p[i].turnaround_time = *time - p[i].ar
                    p[i].response_time = p[i].waiting_time;
                    p[i].remaining_time = 0;
                }
            }
        }
    } while(!done);
}
```

Program 4

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

a)Earliest-deadline First

Program:

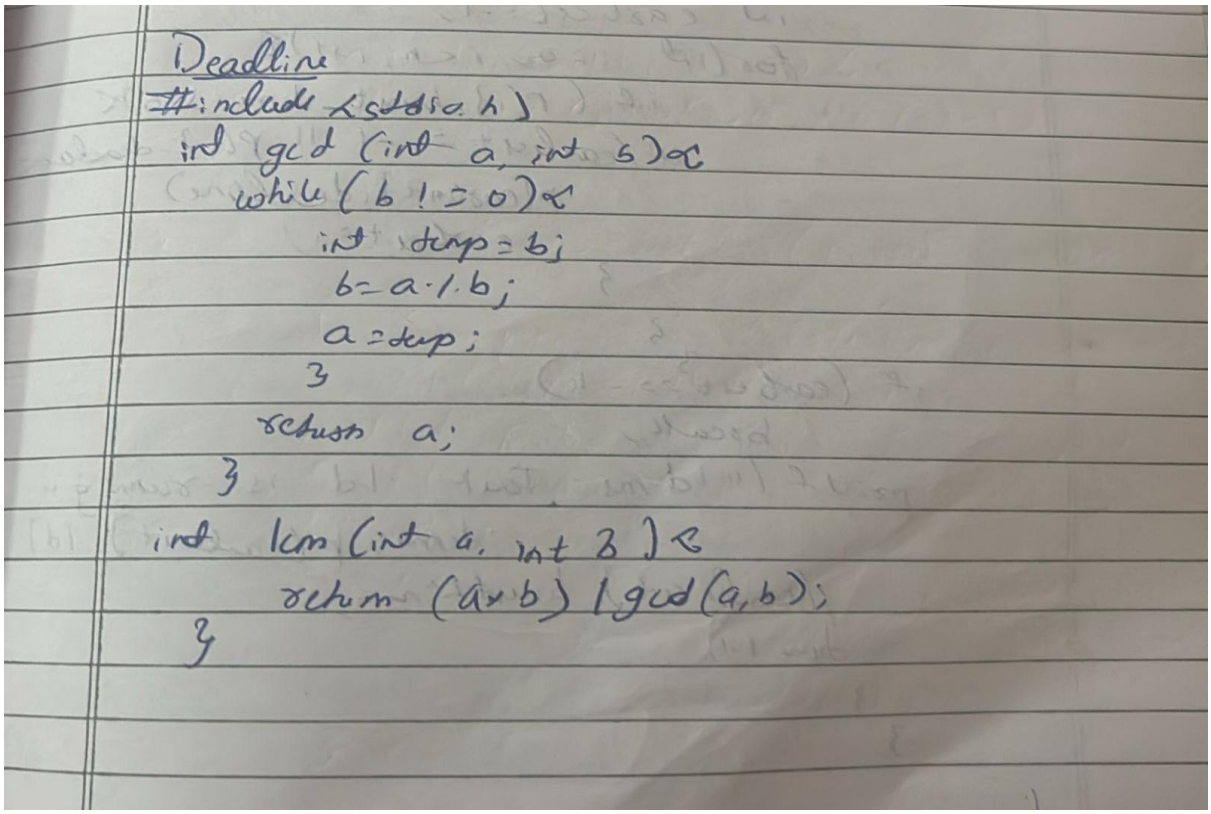
```
#include <stdio.h>

int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

int lcm(int a, int b) {
    return (a * b) / gcd(a, b);
}

struct Process {
    int id, burst_time, deadline, period;
};

void earliest_deadline_first(struct Process p[], int n, int time_limit) {
    int time = 0;
    printf("Earliest Deadline Scheduling:\n");
    printf("PID\tBurst\tDeadline\tPeriod\n");
    for (int i = 0; i < n; i++) {
```

```
        printf("%d\t%d\t\t%d\t\t%d\n", p[i].id, p[i].burst_time, p[i].deadline, p[i].period);
    }

    printf("\nScheduling occurs for %d ms\n", time_limit);
    while (time < time_limit) {
        int earliest = -1;
        for (int i = 0; i < n; i++) {
            if (p[i].burst_time > 0) {
                if (earliest == -1 || p[i].deadline < p[earliest].deadline) {
                    earliest = i;
                }
            }
        }

        if (earliest == -1) break;

        printf("%dms: Task %d is running.\n", time, p[earliest].id);
        p[earliest].burst_time--;
        time++;
    }
}

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

    struct Process processes[n];
    printf("Enter the CPU burst times:\n");
```

```
    for (int i = 0; i < n; i++) {
        scanf("%d", &processes[i].burst_time);
        processes[i].id = i + 1;
    }

    printf("Enter the deadlines:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processes[i].deadline);
    }

    printf("Enter the time periods:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &processes[i].period);
    }

    int hyperperiod = processes[0].period;
    for (int i = 1; i < n; i++) {
        hyperperiod = lcm(hyperperiod, processes[i].period);
    }

    printf("\nSystem will execute for hyperperiod (LCM of periods): %d ms\n", hyperperiod);

    earliest_deadline_first(processes, n, hyperperiod);

    return 0;
}
```

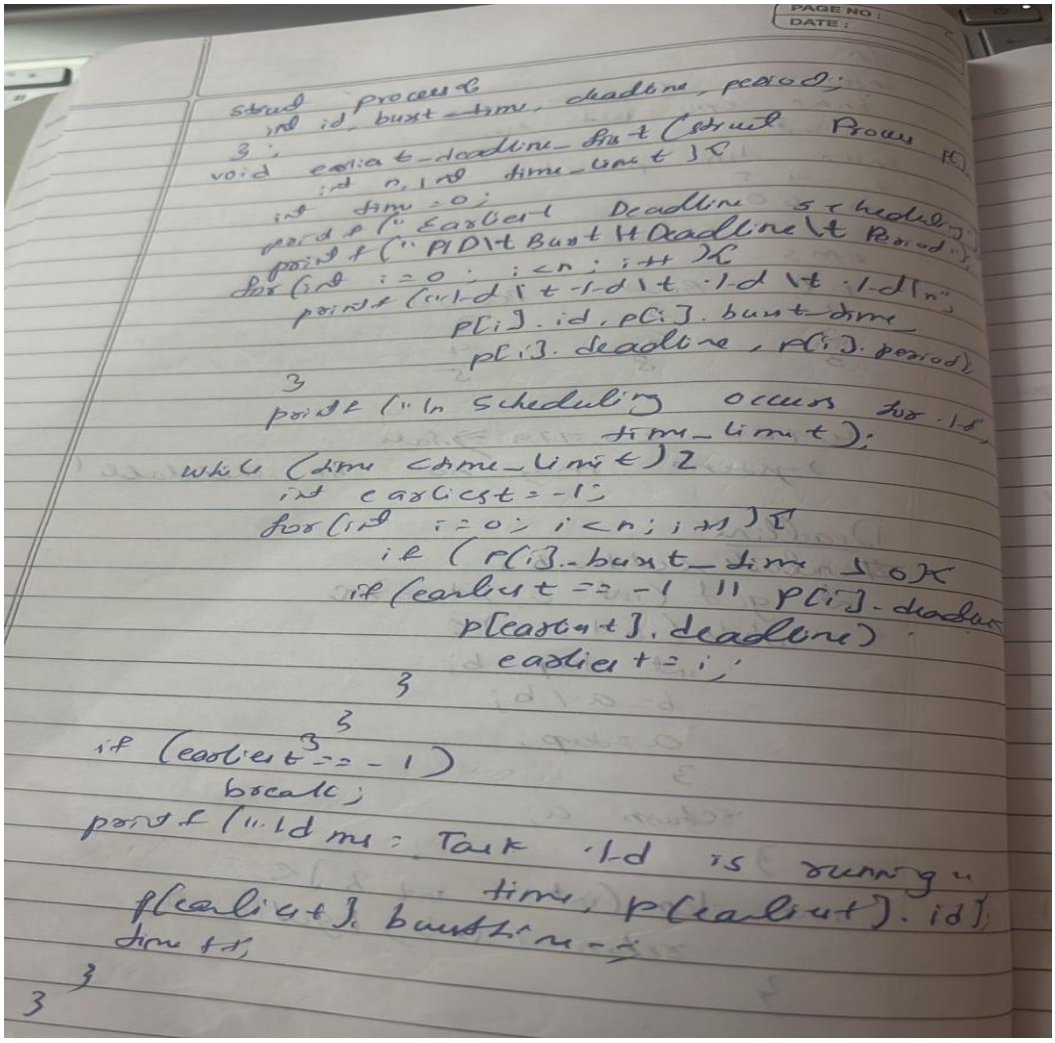
O/P:

```
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.
```

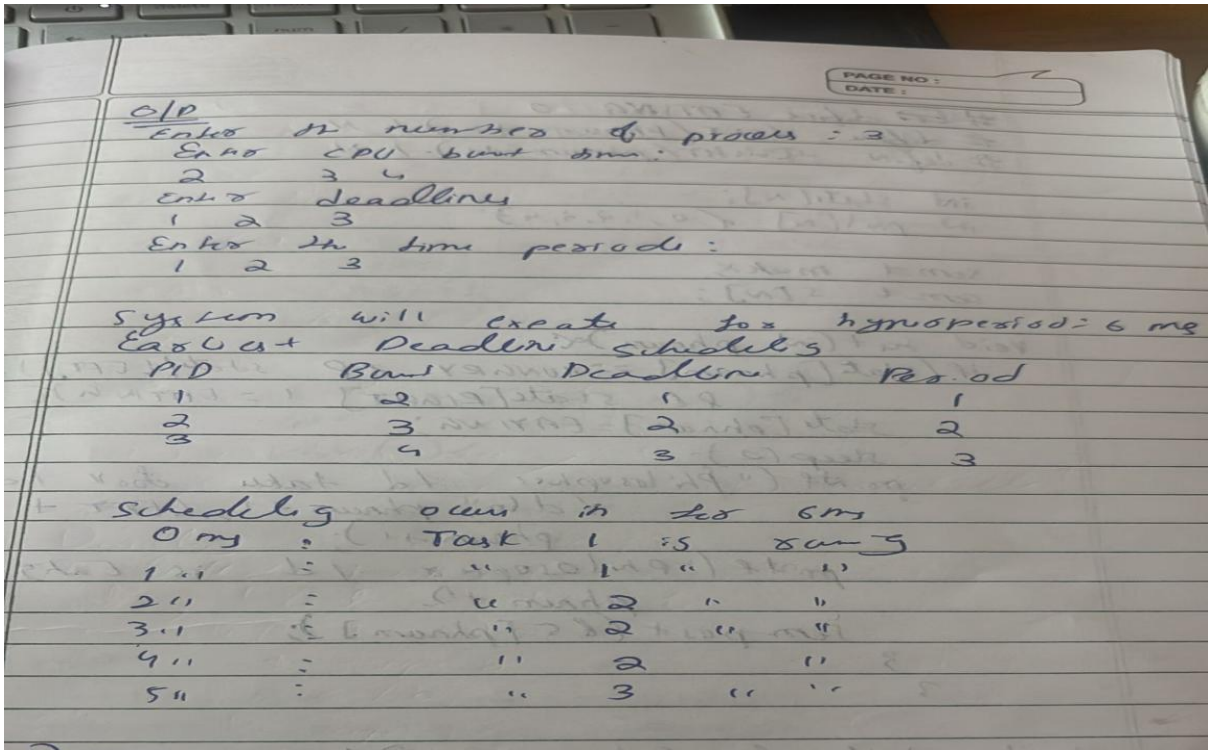


Deadline

```
#include <stdio.h>
int gcd(int a, int b){
    while(b != 0){
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
int lcm(int a, int b){
    return (a*b) / gcd(a,b);
}
```



```
struct process {
    int id, burst_time, deadline, period;
};
void earliest_deadline_first(struct Process p[],
    int n, int time_limit) {
    int time = 0;
    printf("Earliest Deadline Scheduling\n");
    printf("PID\tBurst\tDeadline\tPeriod\n");
    for(int i=0; i<n; i++)
        printf("%d\t%d\t%d\t%d\n",
            p[i].id, p[i].burst_time,
            p[i].deadline, p[i].period);
    printf("\nScheduling occurs for %d",
        time_limit);
    while(time < time_limit) {
        int earliest = -1;
        for(int i=0; i<n; i++) {
            if(p[i].burst_time > 0 &&
            (earliest == -1 || p[i].deadline <
            p[earliest].deadline)
                earliest = i;
        }
    }
    if(earliest == -1)
        break;
    printf("%d ms: Task %d is running\n",
        time, p[earliest].id);
    p[earliest].burst_time--;
    time++;
}
}
```



```
O/P
Enter the number of process : 3
Enter CPU burst time:
2 3 4
Enter deadlines
1 2 3
Enter the time periods:
1 2 3

System will execute for hyperperiod: 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
0 ms : Task 1 is running
1 ms : " 1 " "
2 ms : " 2 " "
3 ms : " 2 " "
4 ms : " 2 " "
5 ms : " 3 " "
```

Program 4

b)Rate- Monotonic

program:

```
#include <stdio.h>

#define MAX_PROCESSES 10

typedef struct {
    int id;
    int burst_time;
    int period;
    int remaining_time;
    int next_deadline;
} Process;

void sort_by_period(Process processes[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].period > processes[j + 1].period) {
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
}
```

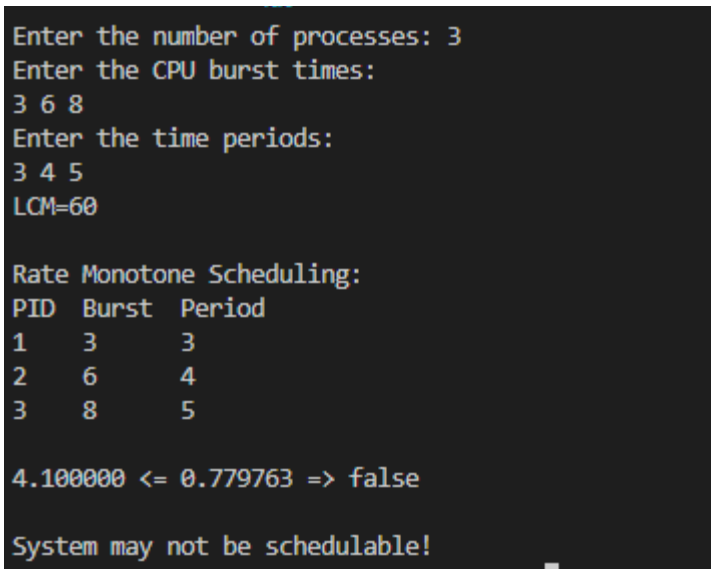
```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

int lcm(int a, int b) {
    return (a * b) / gcd(a, b);
}

int calculate_lcm(Process processes[], int n) {
    int result = processes[0].period;
    for (int i = 1; i < n; i++) {
        result = lcm(result, processes[i].period);
    }
    return result;
}

double utilization_factor(Process processes[], int n) {
    double sum = 0;
    for (int i = 0; i < n; i++) {
        sum += (double)processes[i].burst_time / processes[i].period;
    }
    return sum;
}

double rms_threshold(int n) {
    return n * (pow(2.0, 1.0 / n) - 1);
}

void rate_monotonic_scheduling(Process processes[], int n) {
```

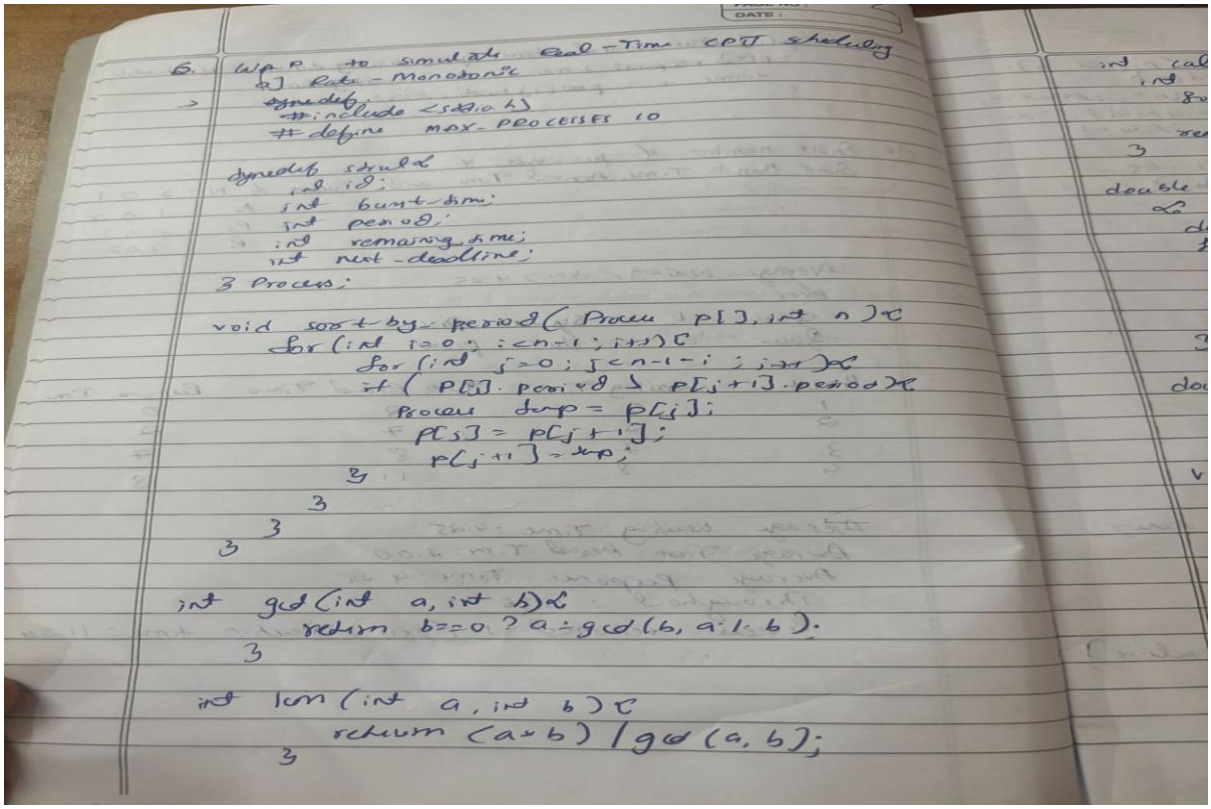
```
    int lcm_period = calculate_lcm(processes, n);
    printf("LCM=%d\n\n", lcm_period);

    printf("Rate Monotone Scheduling:\n");
    printf("PID  Burst  Period\n");
    for (int i = 0; i < n; i++) {
        printf("%d    %d      %d\n", processes[i].id, processes[i].burst_time, processes[i].period);
    }

    double utilization = utilization_factor(processes, n);
    double threshold = rms_threshold(n);
    printf("\n%.6f <= %.6f => %s\n", utilization, threshold, (utilization <= threshold) ? "true" :
"false");

    if (utilization > threshold) {
        printf("\nSystem may not be schedulable!\n");
        return;
    }

    int timeline = 0, executed = 0;
    while (timeline < lcm_period) {
        int selected = -1;
        for (int i = 0; i < n; i++) {
            if (timeline % processes[i].period == 0) {
                processes[i].remaining_time = processes[i].burst_time;
            }
            if (processes[i].remaining_time > 0) {
                selected = i;
                break;
```

```
            }
        }
        if (selected != -1) {
            printf("Time %d: Process %d is running\n", timeline, processes[selected].id);
            processes[selected].remaining_time--;
            executed++;
        } else {
            printf("Time %d: CPU is idle\n", timeline);
        }
        timeline++;
    }
}

int main() {
    int n;
    Process processes[MAX_PROCESSES];

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

    printf("Enter the CPU burst times:\n");
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        scanf("%d", &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;
    }

    printf("Enter the time periods:\n");
    for (int i = 0; i < n; i++) {
```

```
        scanf("%d", &processes[i].period);
    }

    sort_by_period(processes, n);
    rate_monotonic_scheduling(processes, n);

    return 0;
}
```

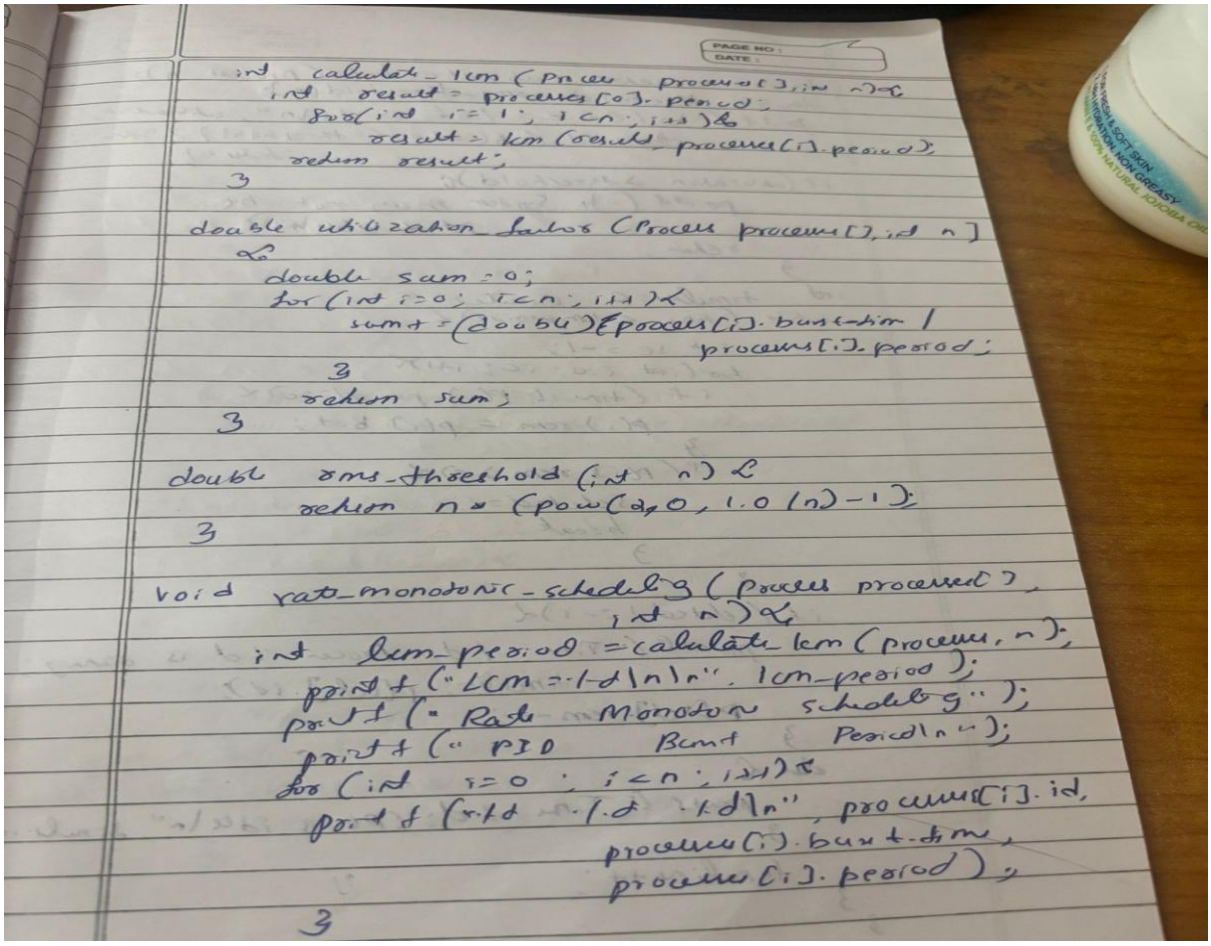
o/p:



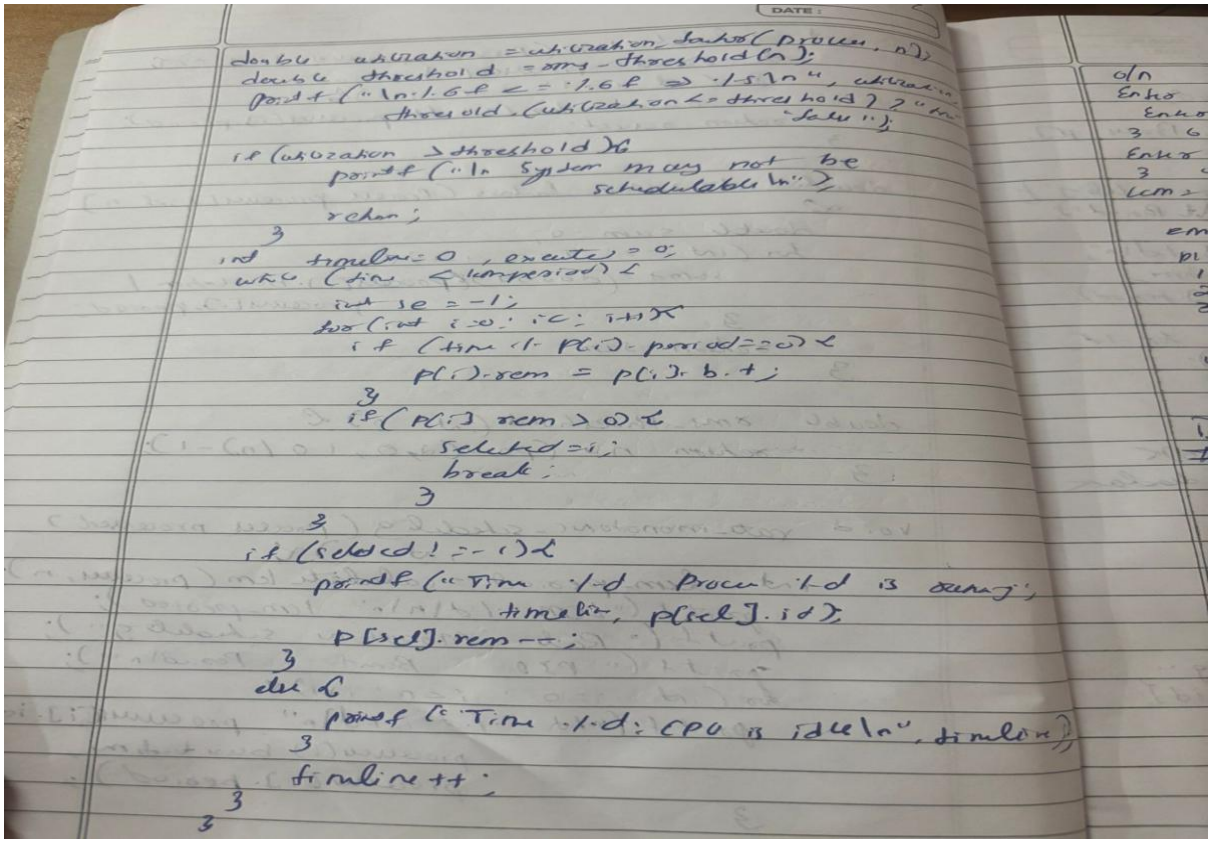


6. WAP to simulate Real-Time CPU scheduling
a) Rate - Monotonic

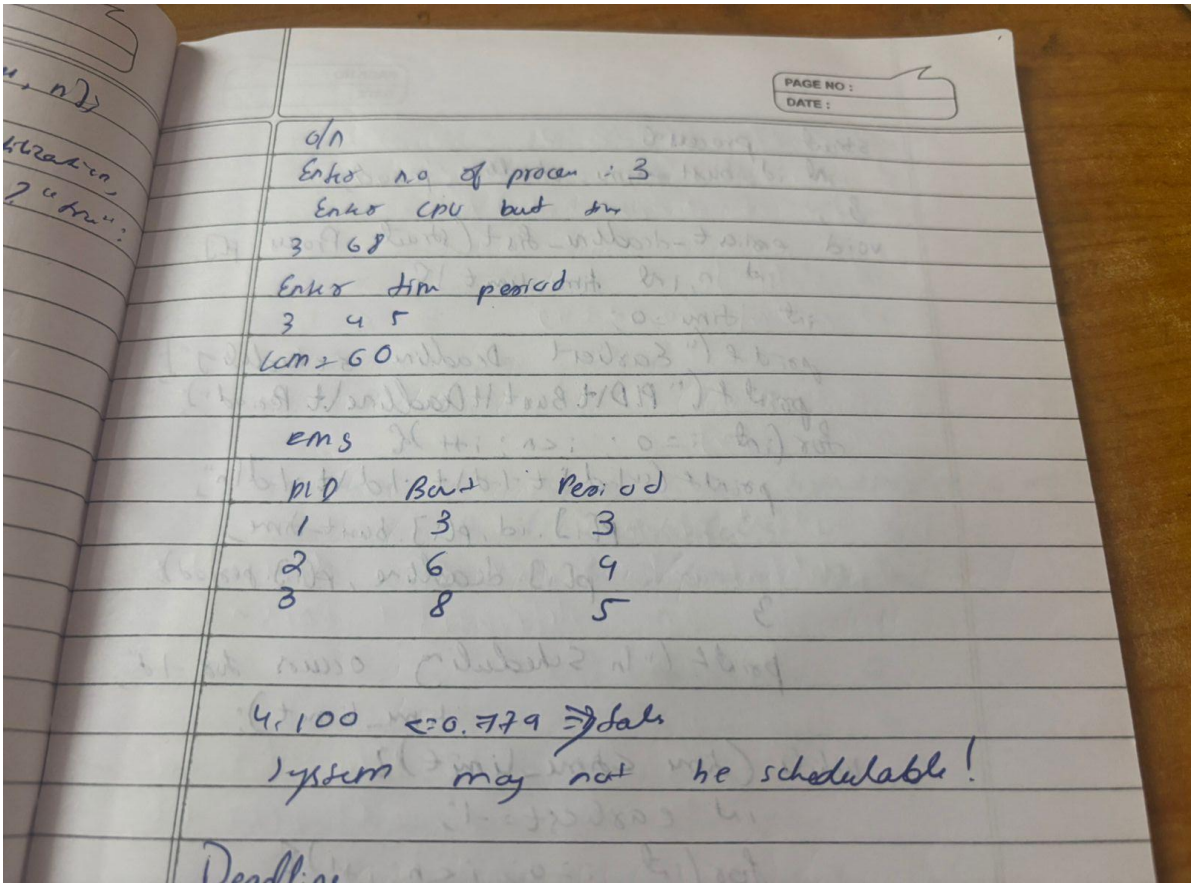
```
#include <stdio.h>
#define MAX_PROCESSES 10

typedef struct {
    int id;
    int burst_time;
    int period;
    int remaining_time;
    int next_deadline;
} Process;

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

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 calculate_lcm(Process processes[], int n) {
    int result = processes[0].period;
    for (int i = 1; i < n; i++)
        result = lcm(result, processes[i].period);
    return result;
}

double utilization_factor(Process processes[], int n) {
    double sum = 0;
    for (int i = 0; i < n; i++)
        sum += (double)processes[i].burst_time /
               processes[i].period;
    return sum;
}

double rms_threshold(int n) {
    return n * (pow(2.0, 1.0/n) - 1);
}

void rate_monotonic_scheduling(Process processes[],
                               int n) {
    int lcm_period = calculate_lcm(processes, n);
    printf("LCM = %d\n\n", lcm_period);
    printf("Rate Monotonic Scheduling:\n");
    printf("PID  Burst  Period\n");
    for (int i = 0; i < n; i++)
        printf("%d  %d  %d\n", processes[i].id,
               processes[i].burst_time,
               processes[i].period);
}
```



```
double utilization = utilization_factor(proc, n);
double threshold = rms_threshold(n);
printf("\n%.6f <= %.6f => %s\n", utilization,
        threshold, (utilization <= threshold) ? "Yes"
                                              : "No");
if (utilization > threshold)
    printf("\n System may not be
                schedulable\n");
    return;
}
int timeline = 0, executed = 0;
while (time < lcm_period) {
    int se = -1;
    for (int i = 0; i < n; i++)
        if (time % proc[i].period == 0) {
            proc[i].rem = proc[i].bt;
        }
        if (proc[i].rem > 0) {
            selected = i;
            break;
        }
    }
    if (selected != -1) {
        printf("Time %d Process %d is running\n",
                timeline, proc[selected].id);
        proc[selected].rem--;
    }
    else {
        printf("Time %d: CPU is idle\n", timeline);
    }
    timeline++;
}
}
```



```
o/p
Enter no of process : 3
Enter CPU burst time
3 6 8
Enter time period
3 4 5
lcm = 60

EMS

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

4.100 <= 0.779 => false
System may not be schedulable!
```

Program 5

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

program:

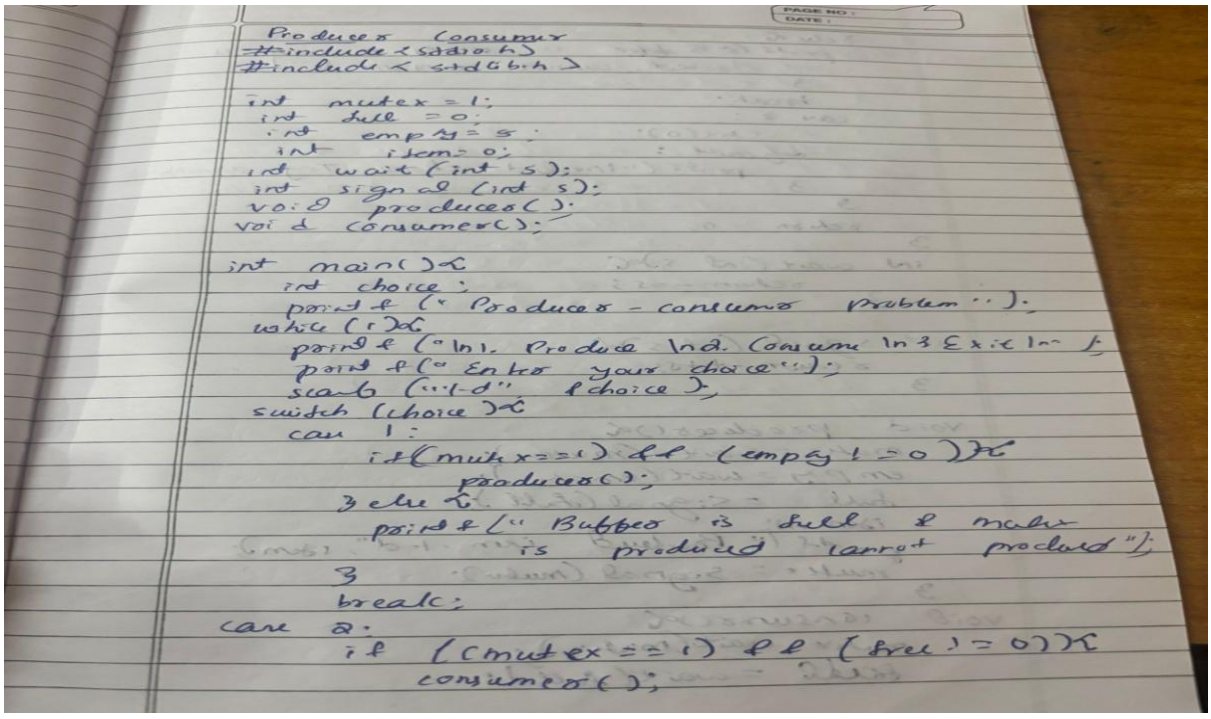
```
#include <stdio.h>
#include <stdlib.h>

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) {
```

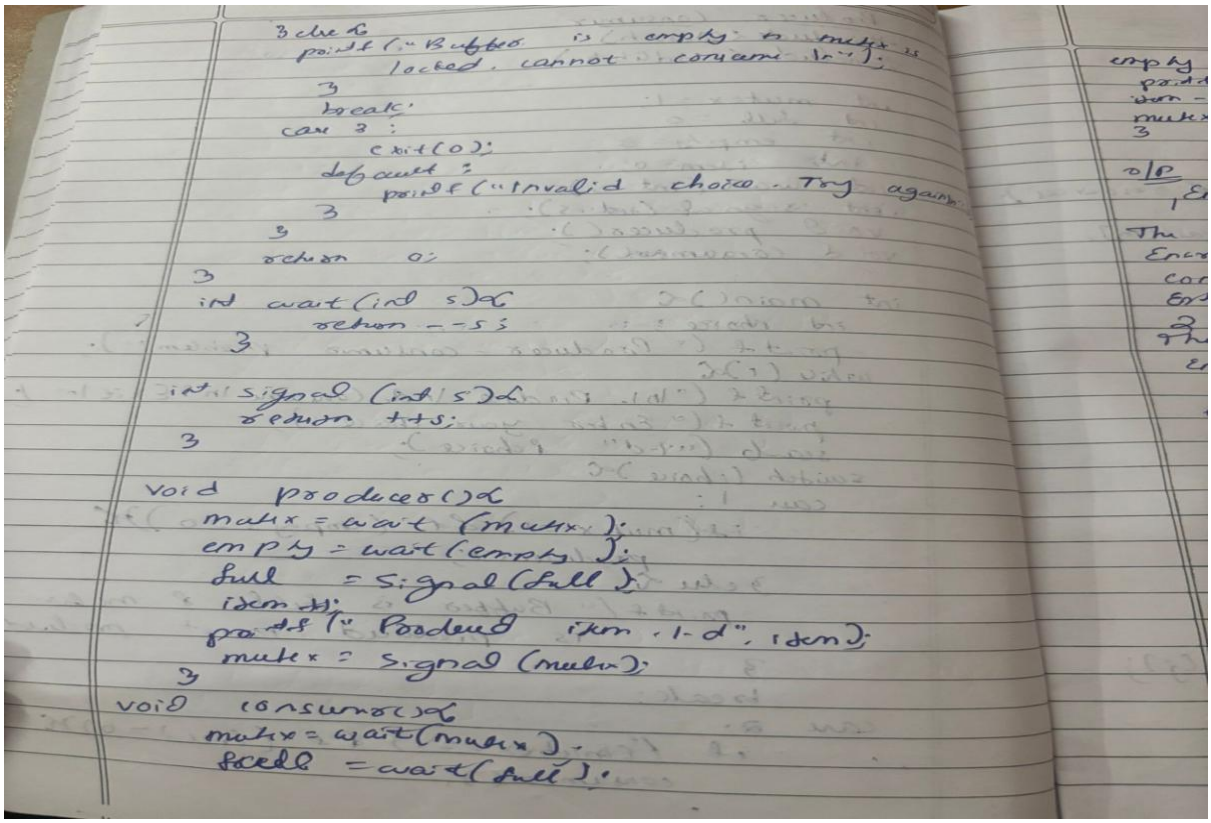
```
    return ++s;
}

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);
}
```

O/P:

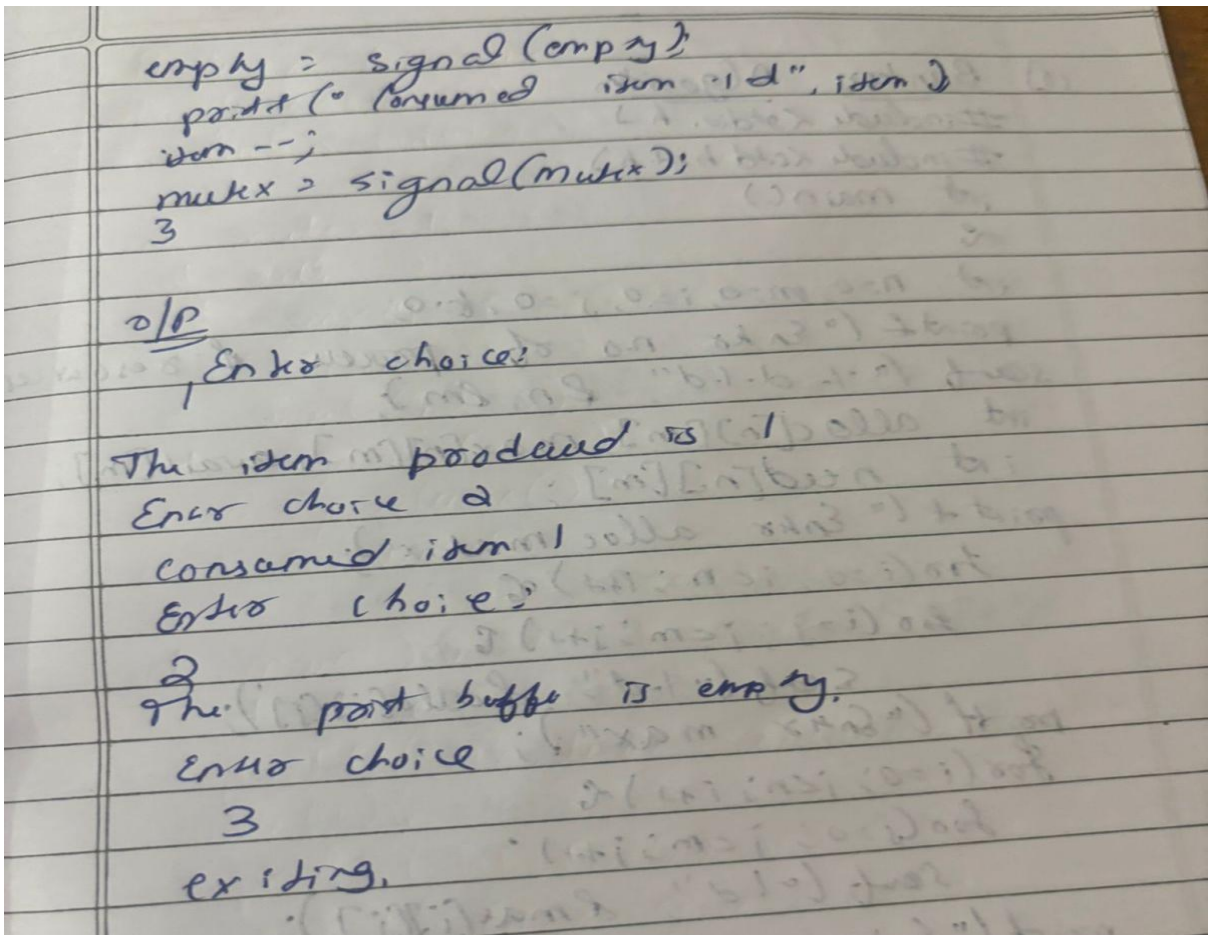
```
Producer-Consumer Problem Simulation

1. Produce
2. Consume
3. Exit
Enter your choice: 1
Produced item 1

1. Produce
2. Consume
3. Exit
Enter your choice: 2
Consumed item 1

1. Produce
2. Consume
3. Exit
Enter your choice: 2
Buffer is empty or mutex is locked. Cannot consume.

1. Produce
2. Consume
3. Exit
Enter your choice:
```



```
Producer Consumer
#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");
  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 & mutex is produced cannot produce");
      }
      break;
  case 2:
      if (mutex==1) && (full != 0){
        consumer();
```



```
3 else {
    printf("Buffer is empty & 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", item);
  mutex = signal(mutex);
}
void consumer(){
  mutex = wait(mutex);
  full = wait(full);
```



```
empty = signal(empty);
printf("Consumed item : %d", item);
item--;
mutex = signal(mutex);
}
```

o/p

```
1
Enter choice:
1
The item produced is 1
Enter choice 2
Consumed item 1
Enter choice:
2
The post buffer is empty.
Enter choice
3
exiting.
```

Program 6

Write a C program to simulate the concept of Dining

Philosophers problem.

Program:

```
#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);
void take_fork(int phnum);
void put_fork(int phnum);
void* philosopher(void* num);

int main() {
```

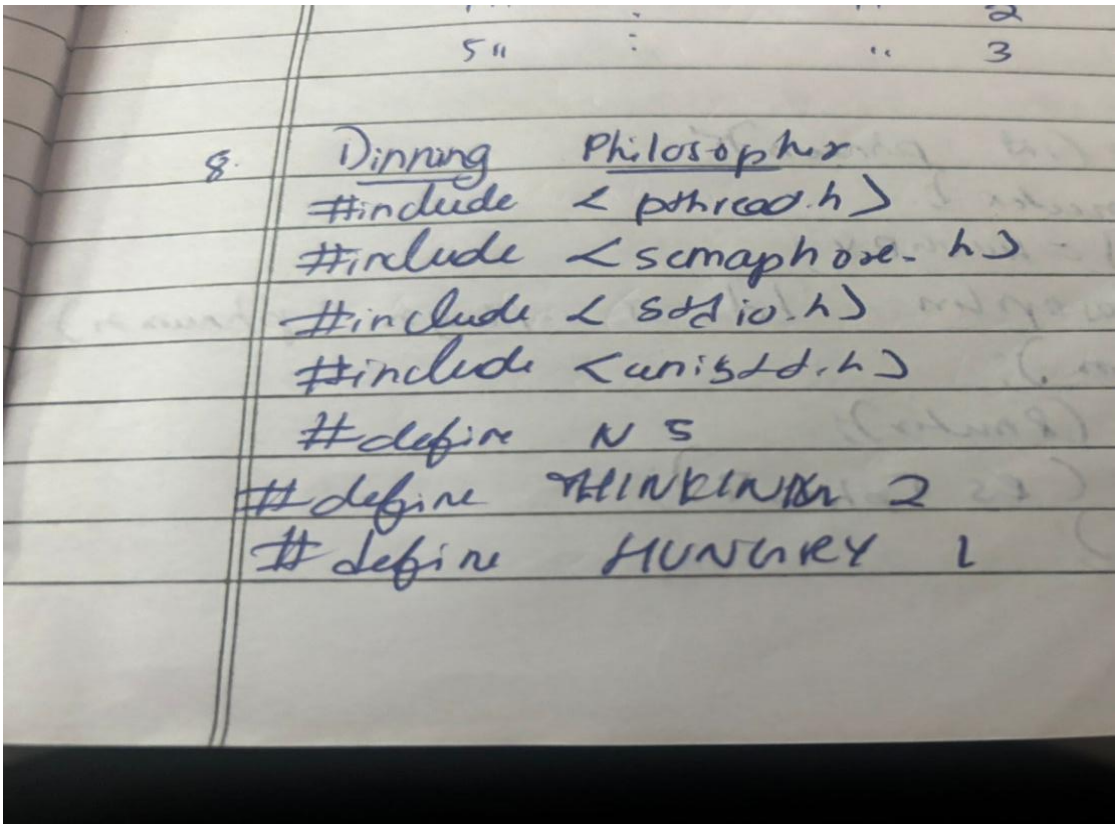
```
    int i;
    pthread_t thread_id[N];

    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;
}

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

        state[phnum] = EATING;

        sleep(2);

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

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

        sem_post(&S[phnum]);
    }
}

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

```
    test(LEFT);
    test(RIGHT);

    sem_post(&mutex);
}

void* philosopher(void* num) {
    int* i = (int*)num;

    while (1) {
        sleep(1);
        take_fork(*i);
        sleep(1);
        put_fork(*i);
    }
}
```

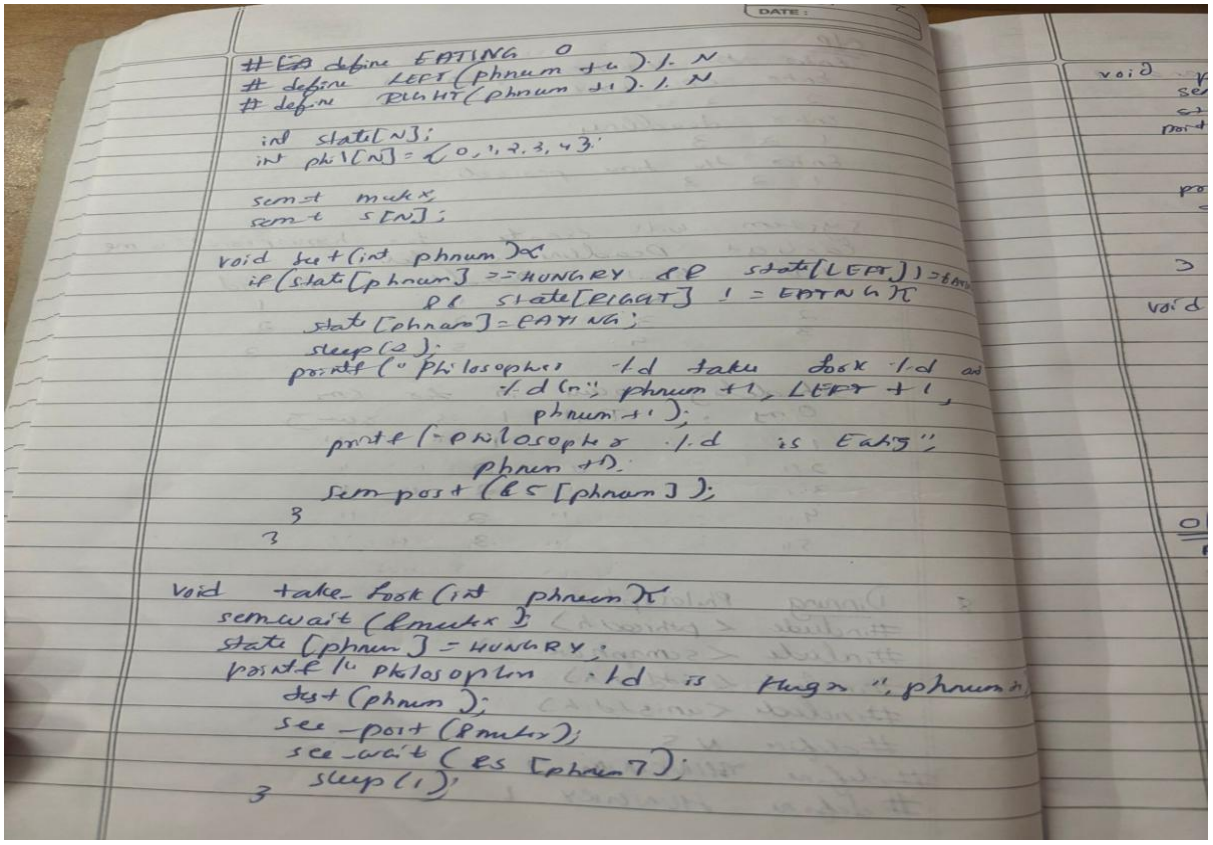
O/P:

```
PS C:\Users\Admin\Desktop\1wa23cs023> gcc dinning_phi.c -o dinning_phi
PS C:\Users\Admin\Desktop\1wa23cs023> ./dinning_phi
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 5 is Hungry
Philosopher 4 is Hungry
Philosopher 2 is Hungry
Philosopher 3 is Hungry
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 1 is Hungry
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 2 takes fork 1 and 2
```

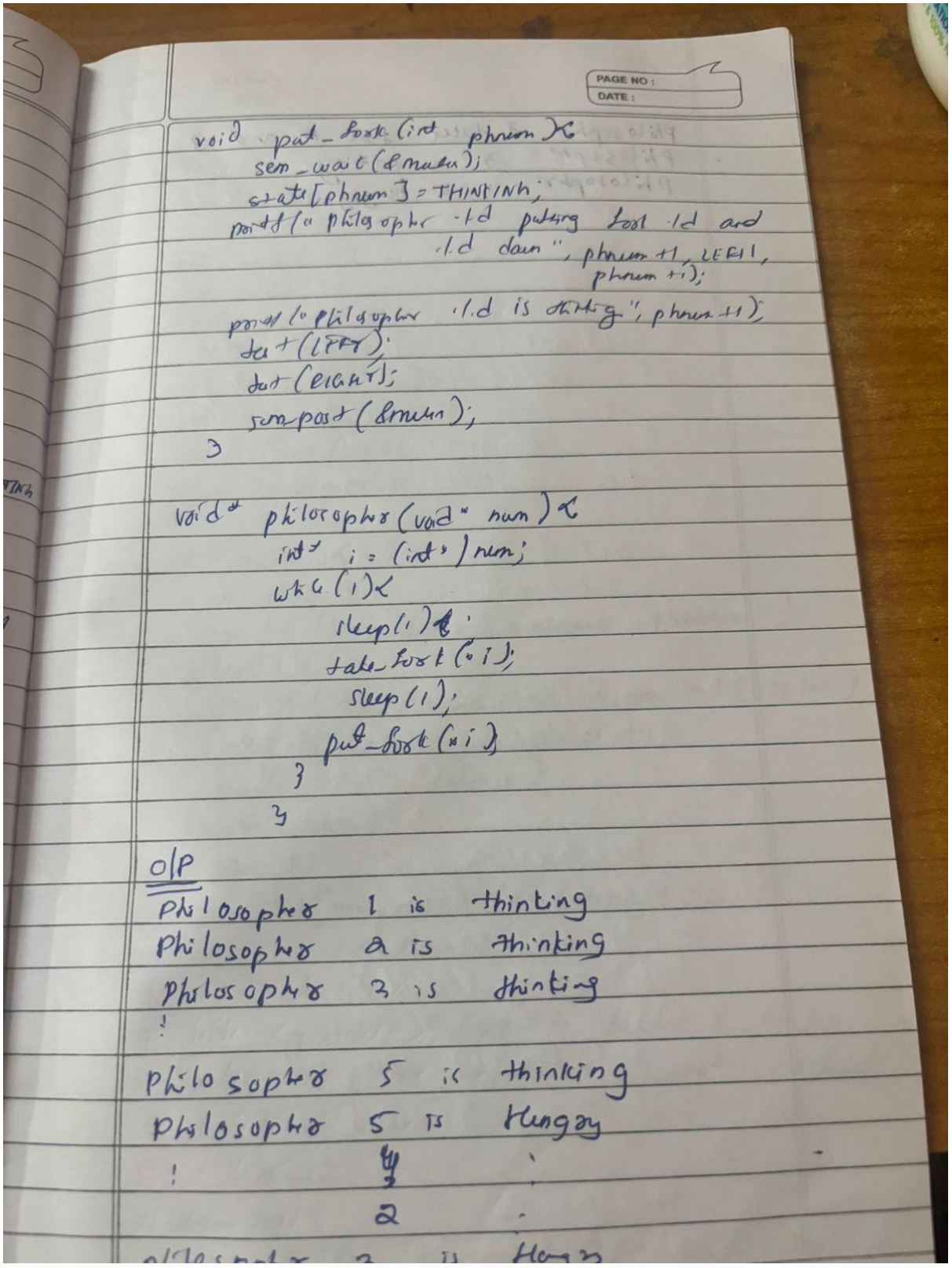


8. Dinning Philosophers

```
#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)
  {
    state[phnum] = EATING;
    sleep(2);
    printf("Philosopher %d takes fork %d and
            %d\n", phnum+1, LEFT+1,
            phnum+1);
    printf("Philosopher %d is Eating",
            phnum+1);
    sem_post(&S[phnum]);
  }
}

void take_fork(int phnum)
{
  sem_wait(&mutex);
  state[phnum] = HUNGRY;
  printf("Philosopher %d is Hungry", 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 fork %d and
            %d down", phnum+1, LEFT,
                        phnum+1);

    printf("Philosopher %d is thinking", phnum+1);
    test(LEFT);
    test(RIGHT);
    sem_post(&mutex);
}

void* philosopher(void* num){
    int* i = (int*)num;
    while(1){
        sleep(1);
        take_fork(*i);
        sleep(1);
        put_fork(*i);
    }
}
```

O/P

Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
!

Philosopher 5 is thinking
Philosopher 5 is Hungry
!
4
2

Program 7

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

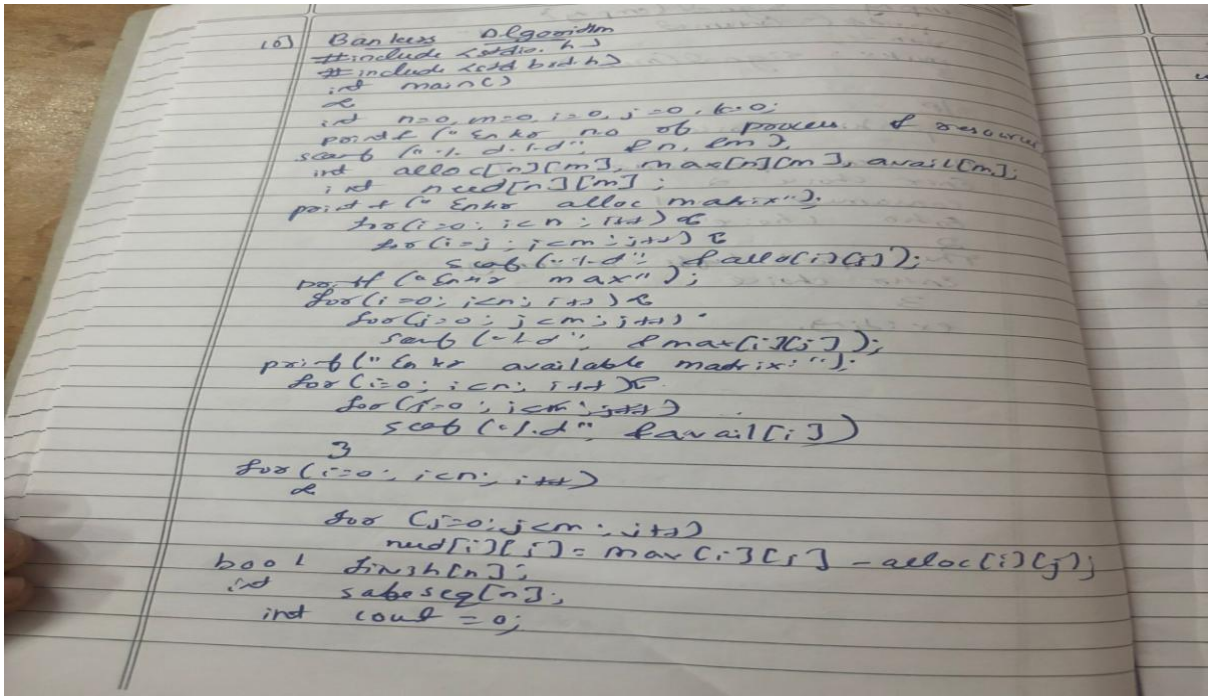
```
#include <stdio.h>
#include <stdbool.h>

int main() {
    int n, m, i, j, k;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    printf("Enter number of resources: ");
    scanf("%d", &m);

    int alloc[n][m], max[n][m], avail[m];
    int need[n][m];

    printf("Enter allocation matrix (%d x %d):\n", n, m);
    for (i = 0; i < n; i++) {
        printf("Allocation for process %d: ", i);
        for (j = 0; j < m; j++)
            scanf("%d", &alloc[i][j]);
    }

    printf("Enter max matrix (%d x %d):\n", n, m);
    for (i = 0; i < n; i++) {
        printf("Max for process %d: ", i);
        for (j = 0; j < m; j++)
            scanf("%d", &max[i][j]);
```

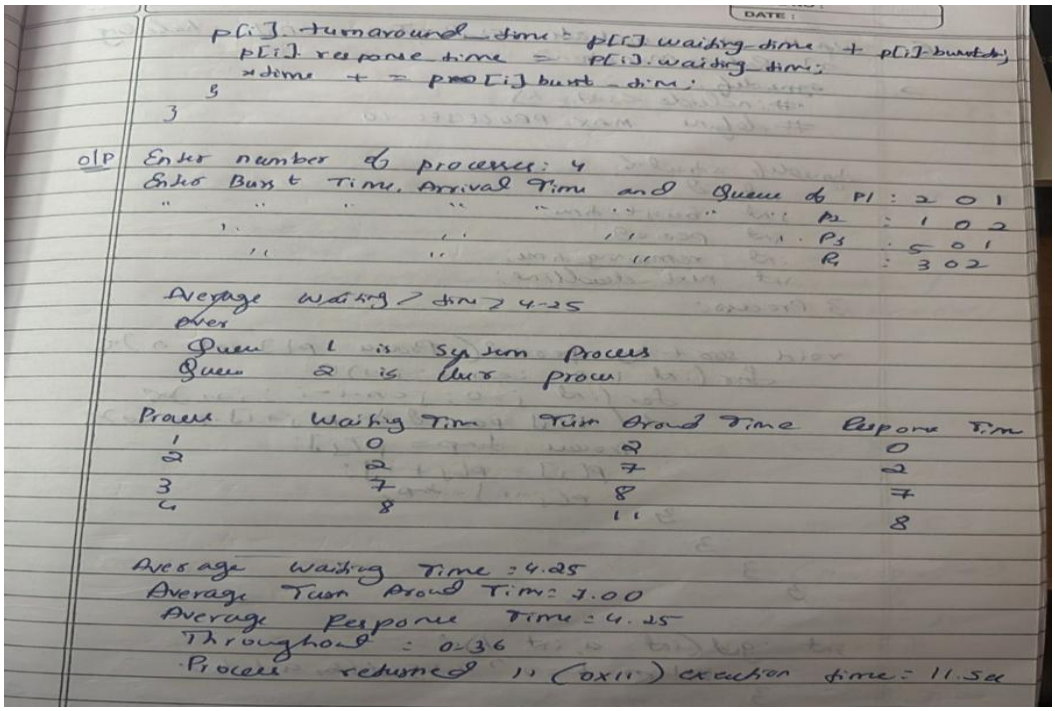
```
}

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

for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
        need[i][j] = max[i][j] - alloc[i][j];

bool finish[n];
int safeSeq[n];
int count = 0;

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

while (count < n) {
    bool found = false;
    for (i = 0; i < n; i++) {
        if (!finish[i]) {
            for (j = 0; j < m; j++)
                if (need[i][j] > avail[j])
                    break;

            if (j == m) {
                for (k = 0; k < m; k++)
                    avail[k] += alloc[i][k];
```

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

    if (!found) {
        printf("System is not in safe state.\n");
        return 1;
    }
}

printf("System is in safe state.\n");
printf("Safe sequence is: ");
for (i = 0; i < n; i++) {
    printf("P%d", safeSeq[i]);
    if (i != n - 1)
        printf(" -> ");
}
printf("\n");

return 0;
}
```

o/p:

```
PS C:\Users\Admin\Desktop\1wa23cs023> ./deadlock
Enter number of processes: 5
Enter number of resources: 3
Enter allocation matrix (5 x 3):
Allocation for process 0: 0 1 0
Allocation for process 1: 2 0 0
Allocation for process 2: 3 0 2
Allocation for process 3: 2 1 1
Allocation for process 4: 0 0 2
Enter max matrix (5 x 3):
Max for process 0: 7 5 3
Max for process 1: 3 2 2
Max for process 2: 9 0 2
Max for process 3: 2 2 2
Max for process 4: 4 3 3
Enter available resources (3 values): 3 3 2
System is in safe state.
Safe sequence is: P1 -> P3 -> P4 -> P0 -> P2
PS C:\Users\Admin\Desktop\1wa23cs023>
```





```
p[i].turnaround_time = p[i].waiting_time + p[i].burst_time;
p[i].response_time = p[i].waiting_time;
*time += p[i].burst_time;
}
}
```

O/P Enter number of processes: 4
Enter Burst Time, Arrival Time and Queue of P1 : 2 0 1
" " " " " " P2 : 1 0 2
" " " " " " P3 : 5 0 1
" " " " " " P4 : 3 0 2

Average Waiting Time: 4.25
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: 11.54

Program 8

Write a C program to simulate deadlock detection

Program:

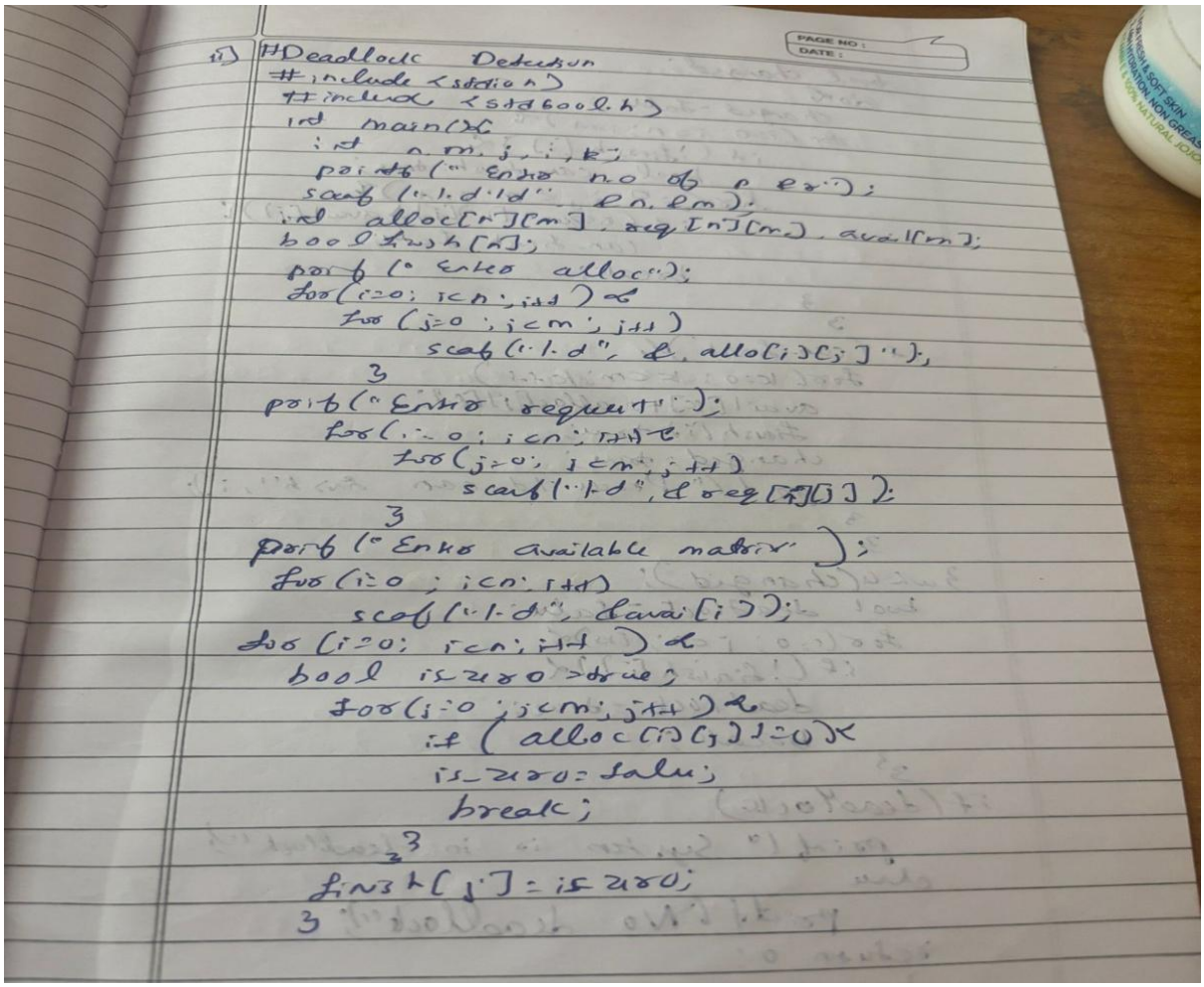
```
#include <stdio.h>
#include <stdbool.h>

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], request[n][m], avail[m];
    bool 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", &request[i][j]);

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

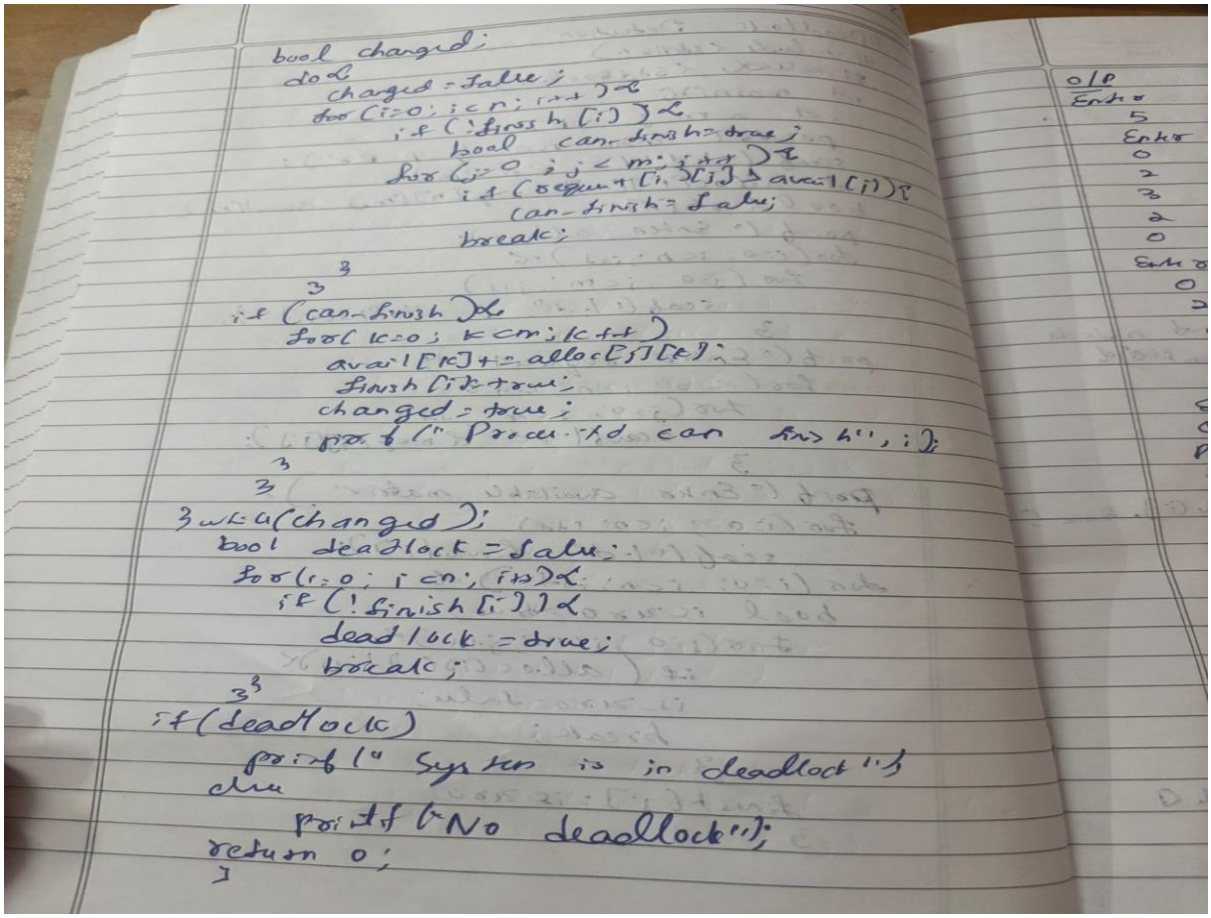
```
for (i = 0; i < n; i++) {
    bool is_zero = true;
    for (j = 0; j < m; j++) {
        if (alloc[i][j] != 0) {
            is_zero = false;
            break;
        }
    }
    finish[i] = is_zero;
}

bool changed;
do {
    changed = false;
    for (i = 0; i < n; i++) {
        if (!finish[i]) {
            bool can_finish = true;
            for (j = 0; j < m; j++) {
                if (request[i][j] > avail[j]) {
                    can_finish = false;
                    break;
                }
            }

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

return 0;
}
```

O/P:

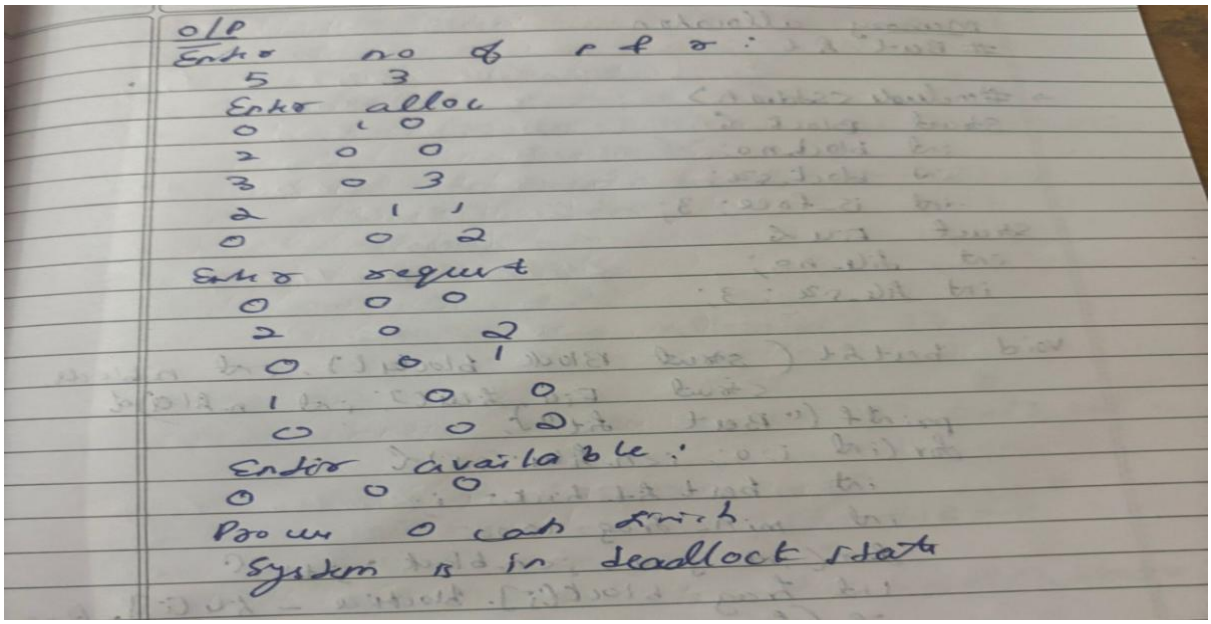
```
PS C:\Users\Admin\Desktop\1wa23cs023> gcc deadlock2.c -o deadlock2
PS C:\Users\Admin\Desktop\1wa23cs023> ./deadlock2
Enter number of processes and resources:
5 3
Enter allocation matrix:
0 1 0
2 0 0
3 0 3
2 1 1
0 0 2
Enter request matrix:
0 0 0
2 0 2
0 0 1
1 0 0
0 0 2
Enter available matrix:
0 0 0
Process 0 can finish.
System is in a deadlock state.
```





```
bool changed;
do{
    changed = false;
    for(i=0; i<n; i++){
        if(!finish[i]){
            bool can_finish = true;
            for(j=0; j<m; j++){
                if(request[i][j] > avail[j]){
                    can_finish = false;
                    break;
                }
            }
            if(can_finish){
                for(k=0; k<m; k++)
                    avail[k] += alloc[i][k];
                finish[i] = true;
                changed = true;
                printf("Process %d can finish", 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
5
Enter
0
2
3
2
0
Enter
0
2
```



```
o/p
Enter no of resources:
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
```

Program 9

Write a C program to simulate the following contiguous

memory allocation techniques

a) Worst-fit

b) Best-fit

c) First-fit

Programs:

```
#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 = 100000;
```

```
        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);
        }
    }
}

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);
        }
    }
}

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;
```

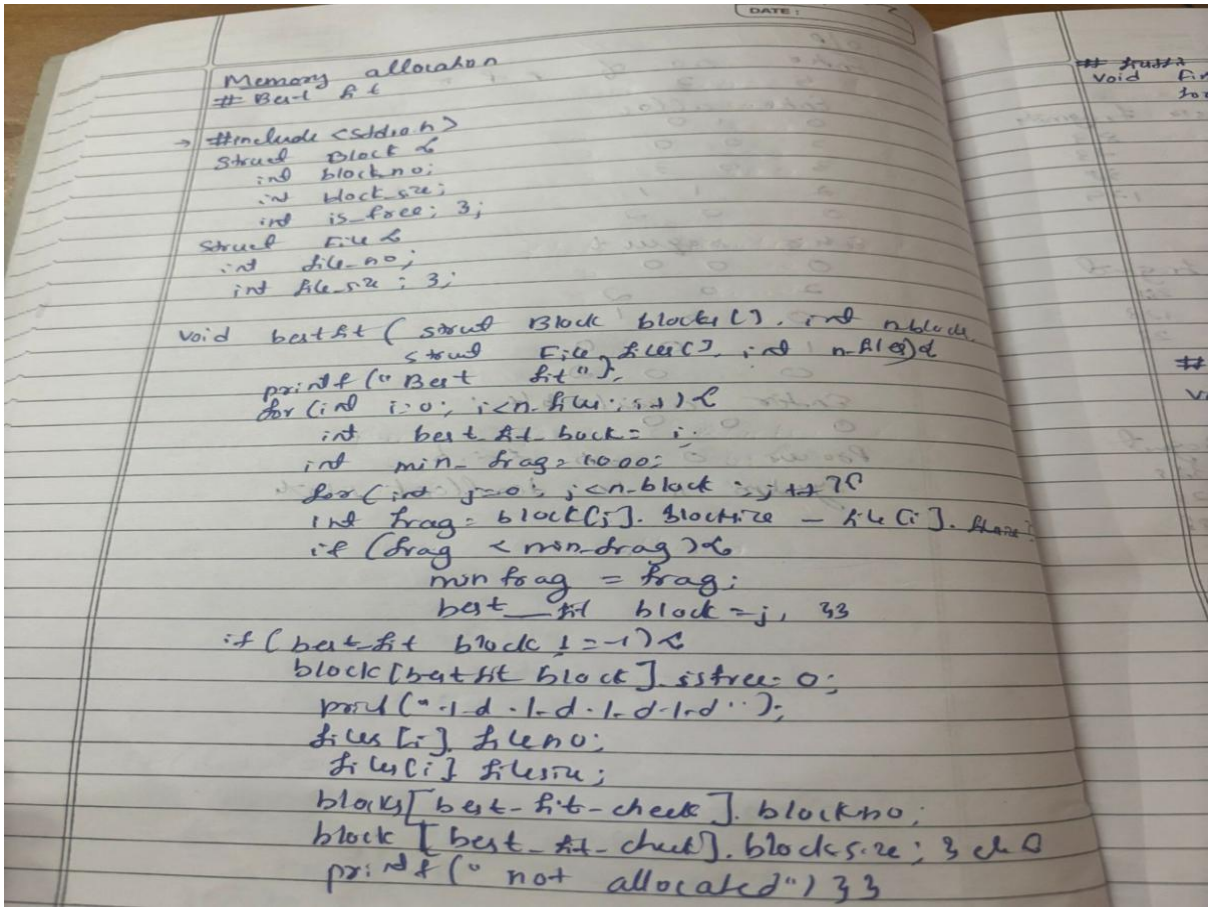
```
        for (int j = 0; j < n_blocks; j++) {
            if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
                int fragment = blocks[j].block_size - files[i].file_size;
                blocks[j].is_free = 0;
                printf("%d\t%d\t\t%d\t\t%d\t\t%d\n",
                        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);
        }
    }
}

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

int main() {
    int n_blocks, n_files, choice;

    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);
}

printf("\nChoose Memory Allocation Technique:\n");
printf("1. First Fit\n");
printf("2. Best Fit\n");
printf("3. Worst Fit\n");
printf("Enter choice (1/2/3): ");
scanf("%d", &choice);

resetBlocks(blocks, n_blocks);
```

```
    switch (choice) {
        case 1:
            firstFit(blocks, n_blocks, files, n_files);
            break;
        case 2:
            bestFit(blocks, n_blocks, files, n_files);
            break;
        case 3:
            worstFit(blocks, n_blocks, files, n_files);
            break;
        default:
            printf("Invalid choice\n");
    }

    return 0;
}
```

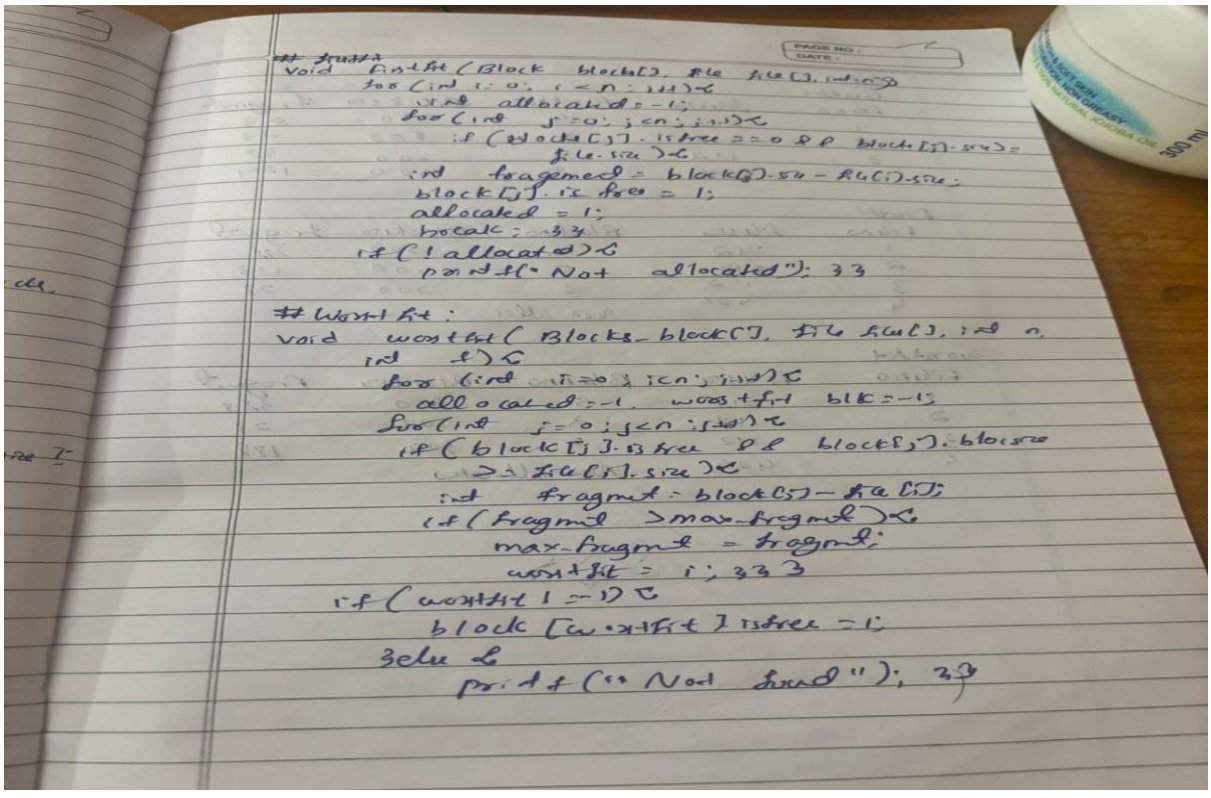
O/P:

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

1. First Fit
2. Best Fit
3. Worst Fit
4. Exit
Enter your choice: 1

        Memory Management Scheme û First Fit
File_no:        File_size       Block_no:       Block_size:
1               212             2               500
2               417             5               600
3               112             3               200
4               426             -               -

1. First Fit
2. Best Fit
3. Worst Fit
4. Exit
Enter your choice: 2

        Memory Management Scheme û Best Fit
File_no:        File_size       Block_no:       Block_size:
1               212             4               300
2               417             2               500
3               112             3               200
4               426             5               600

1. First Fit
2. Best Fit
3. Worst Fit
4. Exit
Enter your choice: 3

        Memory Management Scheme û Worst Fit
File_no:        File_size       Block_no:       Block_size:
1               212             5               600
2               417             2               500
3               112             4               300
4               426             -               -
```



Memory allocation
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 nBlocks,
             struct File files[], int nFiles) {
    printf("Best fit\n");
    for (int i = 0; i < nFiles; i++) {
        int bestFitBlock = -1;
        int minFrag = 10000;
        for (int j = 0; j < nBlocks; j++) {
            int frag = blocks[j].blockSize - files[i].fileSize;
            if (frag < minFrag) {
                minFrag = frag;
                bestFitBlock = j;
            }
        }
        if (bestFitBlock != -1) {
            blocks[bestFitBlock].isFree = 0;
            printf("%d %d %d %d\n", ...);
            files[i].fileNo;
            files[i].fileSize;
            blocks[bestFitCheck].blockNo;
            blocks[bestFitCheck].blockSize;
        } else
            printf("not allocated\n");
    }
}
```



```
# Worst fit
void firstFit(Block blocks[], File files[], int n,
              int m) {
    for (int i = 0; i < n; i++) {
        int allocated = -1;
        for (int j = 0; j < m; j++) {
            if (blocks[j].isFree == 0 && blocks[j].blockSize >=
                files[i].size) {
                int fragment = blocks[j].blockSize - files[i].size;
                blocks[j].isFree = 1;
                allocated = 1;
                break;
            }
        }
        if (!allocated)
            printf("Not allocated\n");
    }
}

# Worst fit:
void worstFit(Blocks blocks[], int files[], int n,
              int m) {
    for (int i = 0; i < n; i++) {
        allocated = -1;
        worstFitBlk = -1;
        for (int j = 0; j < m; j++) {
            if (blocks[j].isFree && blocks[j].blockSize
                >= files[i].size) {
                int fragment = blocks[j] - files[i];
                if (fragment > maxFragment) {
                    maxFragment = fragment;
                    worstFit = i;
                }
            }
        }
        if (worstFit != -1)
            blocks[worstFit].isFree = 1;
        else
            printf("Not found\n");
    }
}
```

PAGE NO :
DATE :

o/p

Best fit

File no	File size	Block no	Block size	Fragment
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	Fragment
1	212	2	500	288
2	417	5	600	183
3	112	3	200	88
4	426	Not alloc		

Worst fit

File no	File size	Block no	Block size	Fragment
1	212	5	650	338
2	417	2	500	83
3	112	4	300	188
4	426	Not allocated		

Program 10

Write a C program to simulate page replacement algorithms

a)FIFO

b) LRU

c) Optimal

a)FIFO

Program:

```
#include <stdio.h>

int main() {
    int pages[100], frames[10];
    int n_pages, n_frames, i, j, k, page_faults = 0, index = 0, found;

    printf("Enter the number of pages: ");
    scanf("%d", &n_pages);

    printf("Enter the page reference string:\n");
    for (i = 0; i < n_pages; i++) {
        scanf("%d", &pages[i]);
    }

    printf("Enter the number of frames: ");
    scanf("%d", &n_frames);

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

```
    frames[i] = -1;
}

printf("\nPage\tFrames\n");

for (i = 0; i < n_pages; i++) {
    found = 0;

    for (j = 0; j < n_frames; j++) {
        if (frames[j] == pages[i]) {
            found = 1;
            break;
        }
    }

    if (!found) {
        frames[index] = pages[i];
        index = (index + 1) % n_frames;
        page_faults++;

        printf("%d\t", pages[i]);
        for (k = 0; k < n_frames; k++) {
            if (frames[k] != -1)
                printf("%d ", frames[k]);
            else
                printf("- ");
        }
        printf("\n");
    } else {
```

```
            printf("%d\tNo Page Fault\n", pages[i]);
        }
    }

    printf("\nTotal Page Faults: %d\n", page_faults);

    return 0;
}
```

O/P:

```
PS C:\Users\Admin\Desktop\1wa23cs023> gcc FIFO.c -o FIFO
PS C:\Users\Admin\Desktop\1wa23cs023> ./FIFO
Enter the number of pages: 12
Enter the page reference string:
1 3 0 3 5 6 3 0 1 2 4 5
Enter the number of frames: 3

Page    Frames
1       1 - -
3       1 3 -
0       1 3 0
3       No Page Fault
5       5 3 0
6       5 6 0
3       5 6 3
0       0 6 3
1       0 1 3
2       0 1 2
4       4 1 2
5       4 5 2

Total Page Faults: 11
PS C:\Users\Admin\Desktop\1wa23cs023>
```

b)LRU

Program:

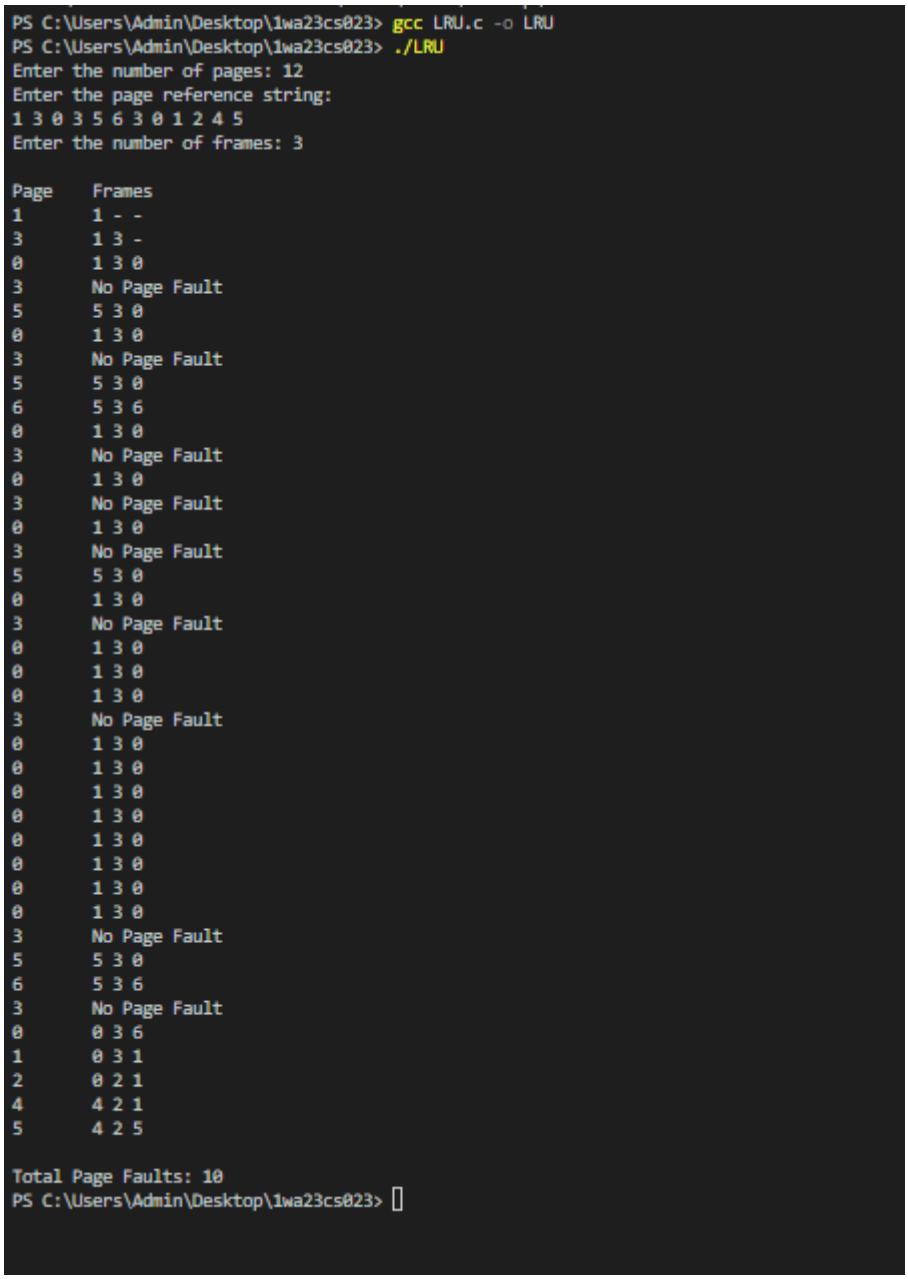
```
#include <stdio.h>

int findLRU(int time[], int n) {
    int i, min = time[0], pos = 0;
    for (i = 1; i < n; i++) {
        if (time[i] < min) {
            min = time[i];
            pos = i;
        }
    }
    return pos;
}

int main() {
    int pages[100], frames[10], time[10];
    int n_pages, n_frames, i, j, pos, page_faults = 0, counter = 0, found;

    printf("Enter the number of pages: ");
    scanf("%d", &n_pages);

    printf("Enter the page reference string:\n");
    for (i = 0; i < n_pages; i++) {
        scanf("%d", &pages[i]);
```

```
}

printf("Enter the number of frames: ");
scanf("%d", &n_frames);

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

printf("\nPage\tFrames\n");

for (i = 0; i < n_pages; i++) {
   found = 0;

   for (j = 0; j < n_frames; j++) {
      if (frames[j] == pages[i]) {
         counter++;
         time[j] = counter;
         found = 1;
         break;
      }
   }

   if (!found) {
      int empty_found = 0;
      for (j = 0; j < n_frames; j++) {
         if (frames[j] == -1) {
            counter++;
            frames[j] = pages[i];
```

```
            time[j] = counter;
            page_faults++;
            empty_found = 1;
            break;
        }
    }

    if (!empty_found) {
        pos = findLRU(time, n_frames);
        counter++;
        frames[pos] = pages[i];
        time[pos] = counter;
        page_faults++;
    }

    printf("%d\t", pages[i]);
    for (j = 0; j < n_frames; j++) {
        if (frames[j] != -1)
            printf("%d ", frames[j]);
        else
            printf("- ");
    }
    printf("\n");
  } else {
    printf("%d\tNo Page Fault\n", pages[i]);
  }
}

printf("\nTotal Page Faults: %d\n", page_faults);
```

```
    return 0;
}
```

O/P:



```
PS C:\Users\Admin\Desktop\1wa23cs023> gcc LRU.c -o LRU
PS C:\Users\Admin\Desktop\1wa23cs023> ./LRU
Enter the number of pages: 12
Enter the page reference string:
1 3 0 3 5 6 3 0 1 2 4 5
Enter the number of frames: 3

Page    Frames
1       1 - -
3       1 3 -
0       1 3 0
3       No Page Fault
5       5 3 0
0       1 3 0
3       No Page Fault
5       5 3 0
6       5 3 6
0       1 3 0
3       No Page Fault
0       1 3 0
3       No Page Fault
0       1 3 0
3       No Page Fault
5       5 3 0
0       1 3 0
3       No Page Fault
0       1 3 0
0       1 3 0
0       1 3 0
3       No Page Fault
0       1 3 0
0       1 3 0
0       1 3 0
0       1 3 0
0       1 3 0
0       1 3 0
0       1 3 0
0       1 3 0
3       No Page Fault
5       5 3 0
6       5 3 6
3       No Page Fault
0       0 3 6
1       0 3 1
2       0 2 1
4       4 2 1
5       4 2 5

Total Page Faults: 10
PS C:\Users\Admin\Desktop\1wa23cs023>
```

c) Optimal

Program:

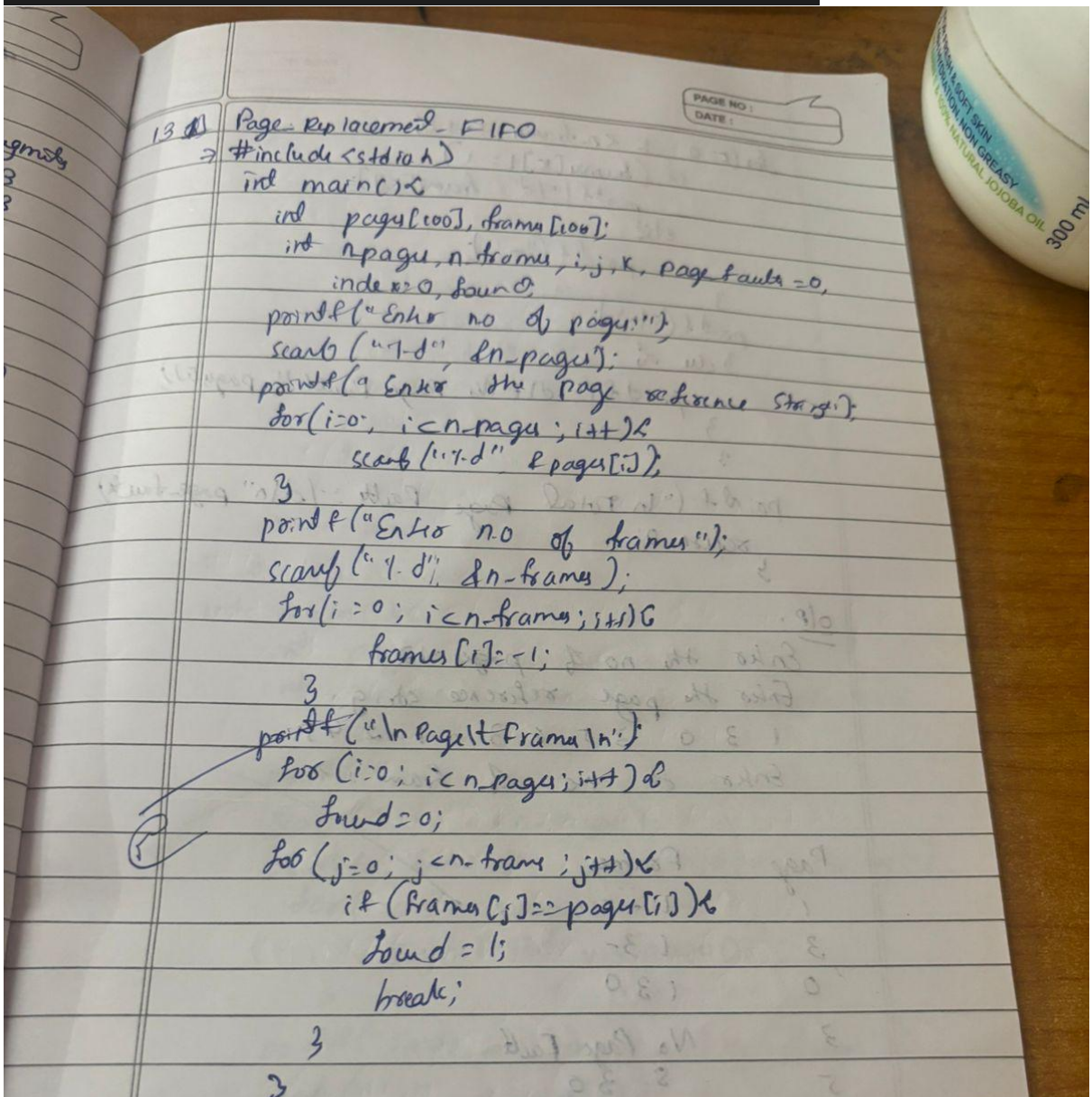
```
#include <stdio.h>

int predict(int pages[], int frames[], int n_pages, int index, int n_frames) {
    int i, j, farthest = index, result = -1;

    for (i = 0; i < n_frames; i++) {
        int found = 0;
        for (j = index + 1; j < n_pages; j++) {
            if (frames[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    result = i;
                }
                found = 1;
                break;
            }
        }
        if (!found)
            return i;
    }

    return (result == -1) ? 0 : result;
}
```

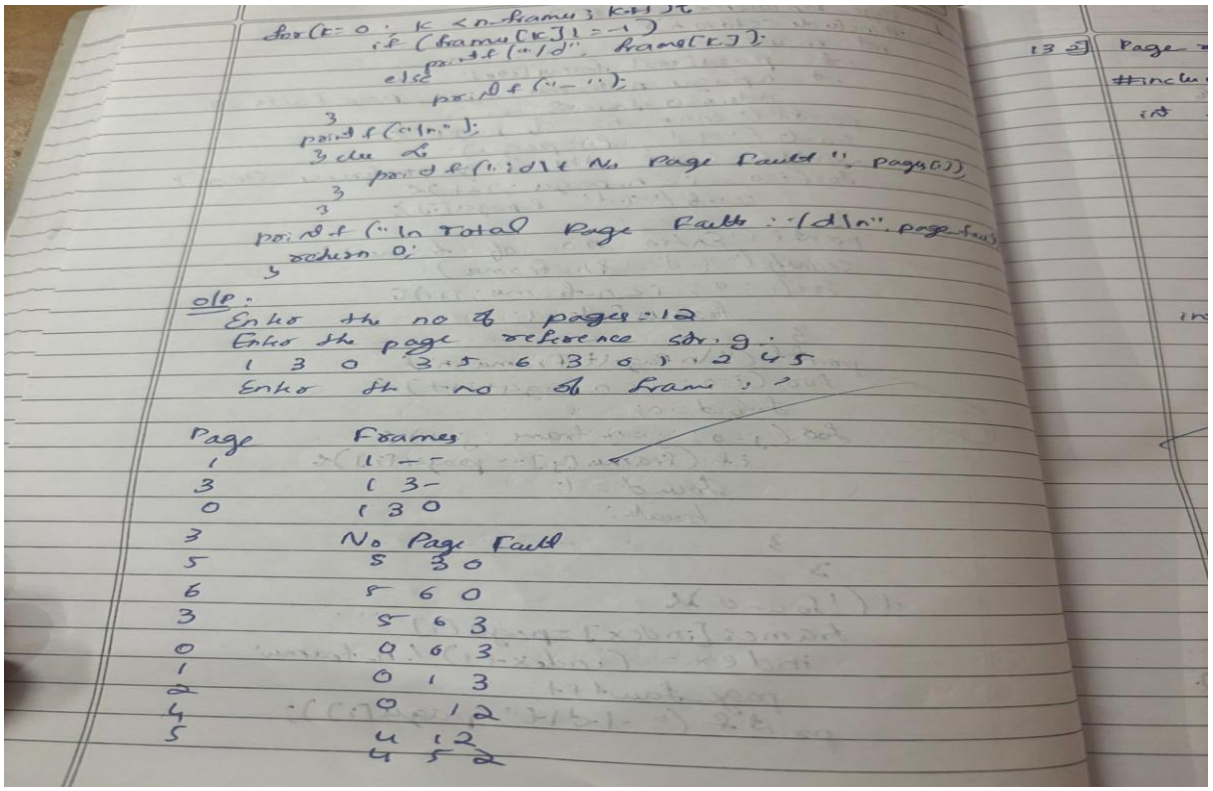
```
int main() {
    int pages[100], frames[10];
    int n_pages, n_frames, i, j, page_faults = 0, filled = 0, found;

    printf("Enter the number of pages: ");
    scanf("%d", &n_pages);

    printf("Enter the page reference string:\n");
    for (i = 0; i < n_pages; i++) {
        scanf("%d", &pages[i]);
    }

    printf("Enter the number of frames: ");
    scanf("%d", &n_frames);

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

    printf("\nPage\tFrames\n");

    for (i = 0; i < n_pages; i++) {
        found = 0;

        for (j = 0; j < n_frames; j++) {
            if (frames[j] == pages[i]) {
                found = 1;
                break;
            }
```

```
        }

        if (!found) {
            if (filled < n_frames) {
                frames[filled++] = pages[i];
            } else {
                int pos = predict(pages, frames, n_pages, i, n_frames);
                frames[pos] = pages[i];
            }

            page_faults++;

            printf("%d\t", pages[i]);
            for (j = 0; j < n_frames; j++) {
                if (frames[j] != -1)
                    printf("%d ", frames[j]);
                else
                    printf("- ");
            }
            printf("\n");
        } else {
            printf("%d\tNo Page Fault\n", pages[i]);
        }
    }

    printf("\nTotal Page Faults: %d\n", page_faults);

    return 0;
}
```

O/P:

```
PS C:\Users\Admin\Desktop\1wa23cs023> gcc optimal.c -o optimal
PS C:\Users\Admin\Desktop\1wa23cs023> ./optimal
Enter the number of pages: 12
Enter the page reference string:
1 3 0 3 5 6 3 0 1 2 4 5
Enter the number of frames: 3

Page    Frames
1       1 - -
3       1 3 -
0       1 3 0
3       No Page Fault
5       5 3 0
6       6 3 0
3       No Page Fault
0       No Page Fault
1       1 3 0
2       2 3 0
4       4 3 0
5       5 3 0

Total Page Faults: 9
PS C:\Users\Admin\Desktop\1wa23cs023>
```



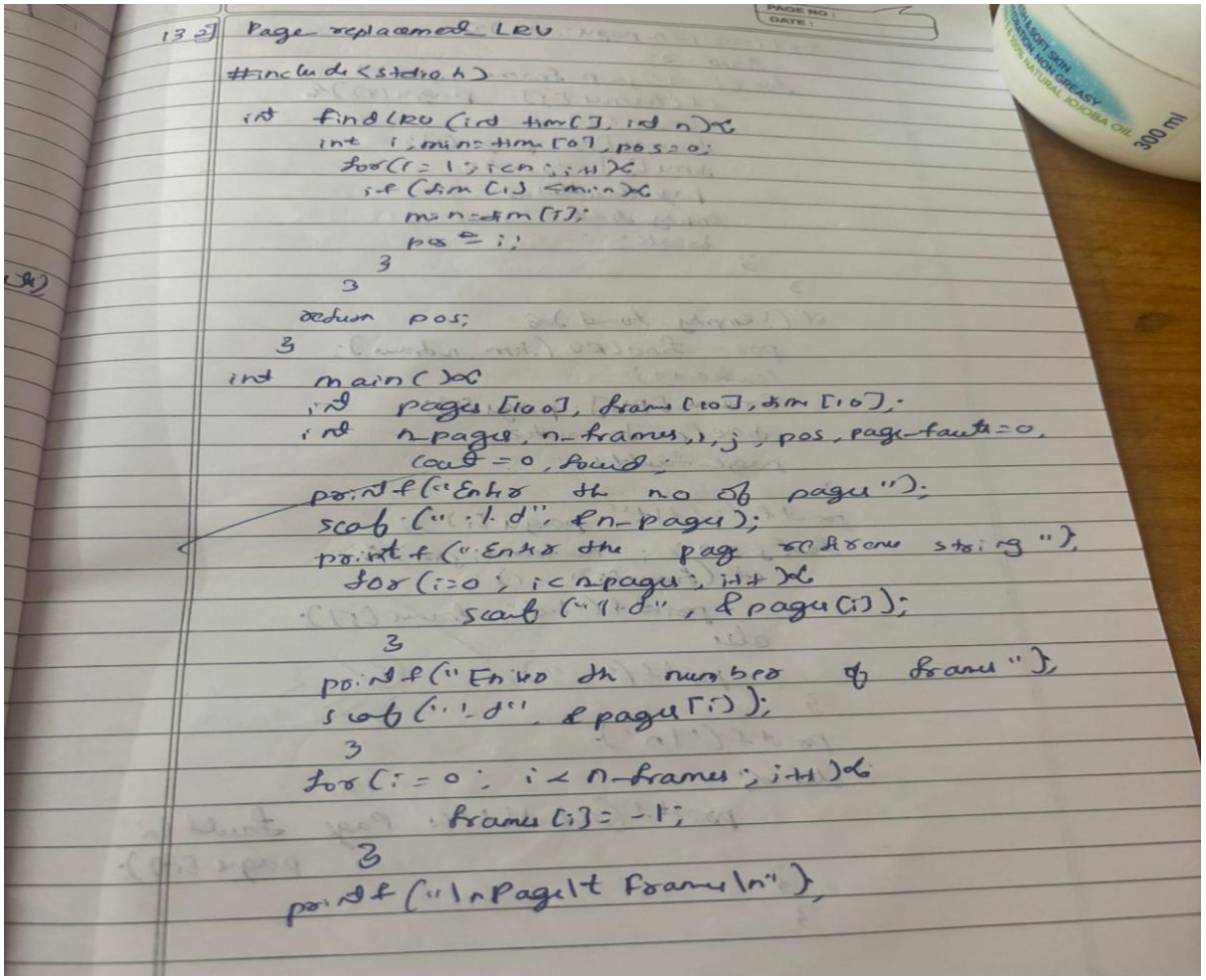


```
for(k=0; k<n_frames; k++)
    if(frame[k]!=-1)
        printf("%d", frame[k]);
    else
        printf("-");
printf("\n");
else
    printf("%d No Page Fault", page[i]);
printf("\nTotal Page Fault: %d\n", page_fault);
return 0;
```

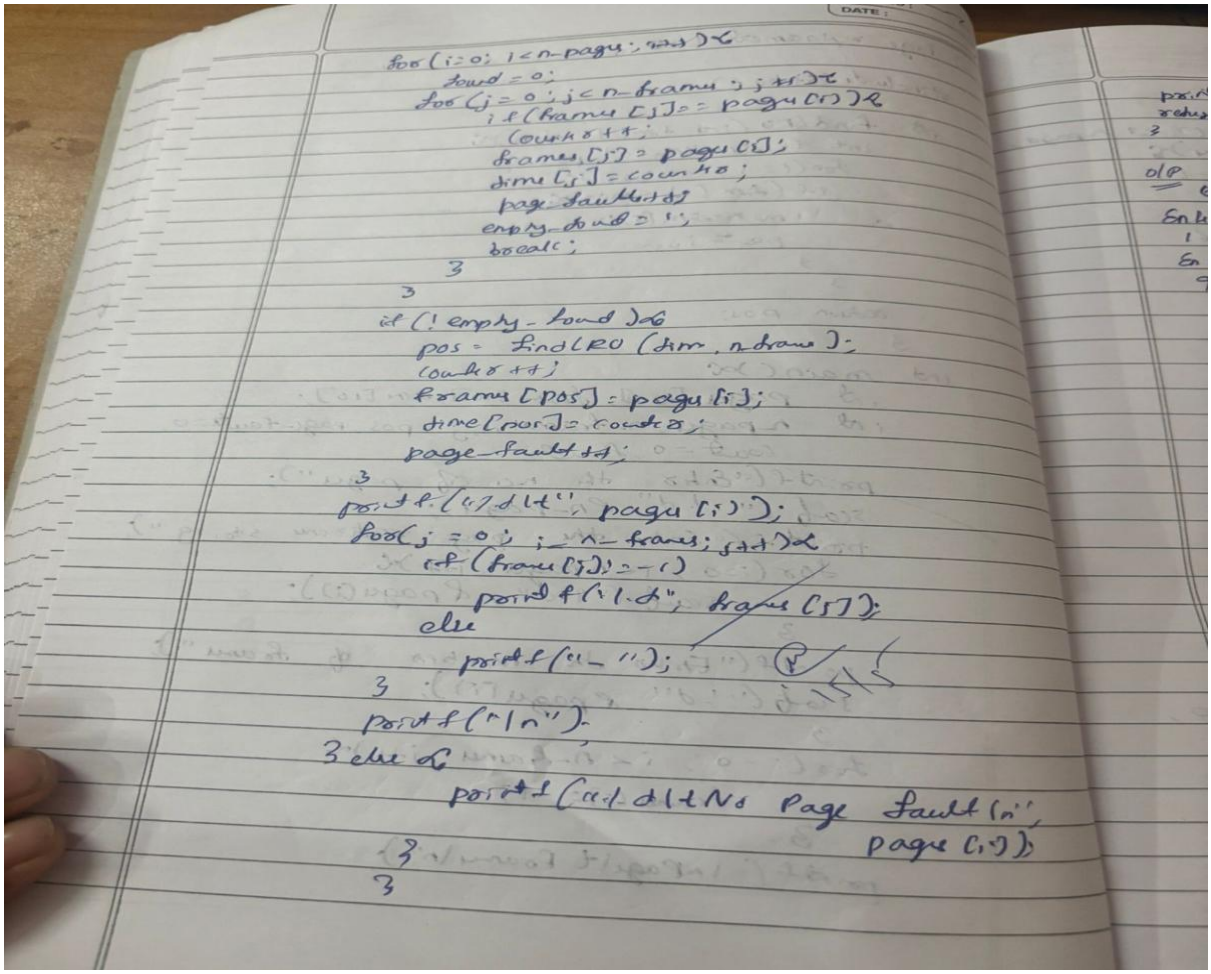
o/p:

Enter the no of pages: 12
Enter the page reference str: 1 3 0 3 5 6 3 0 1 2 4 5
Enter the no of frame: 3

Page	Frames
1	1 - -
3	1 3 -
0	1 3 0
3	No Page Fault
5	5 3 0
6	5 6 0
3	5 6 3
0	0 6 3
1	0 1 3
2	0 1 2
4	4 1 2
5	4 5 2



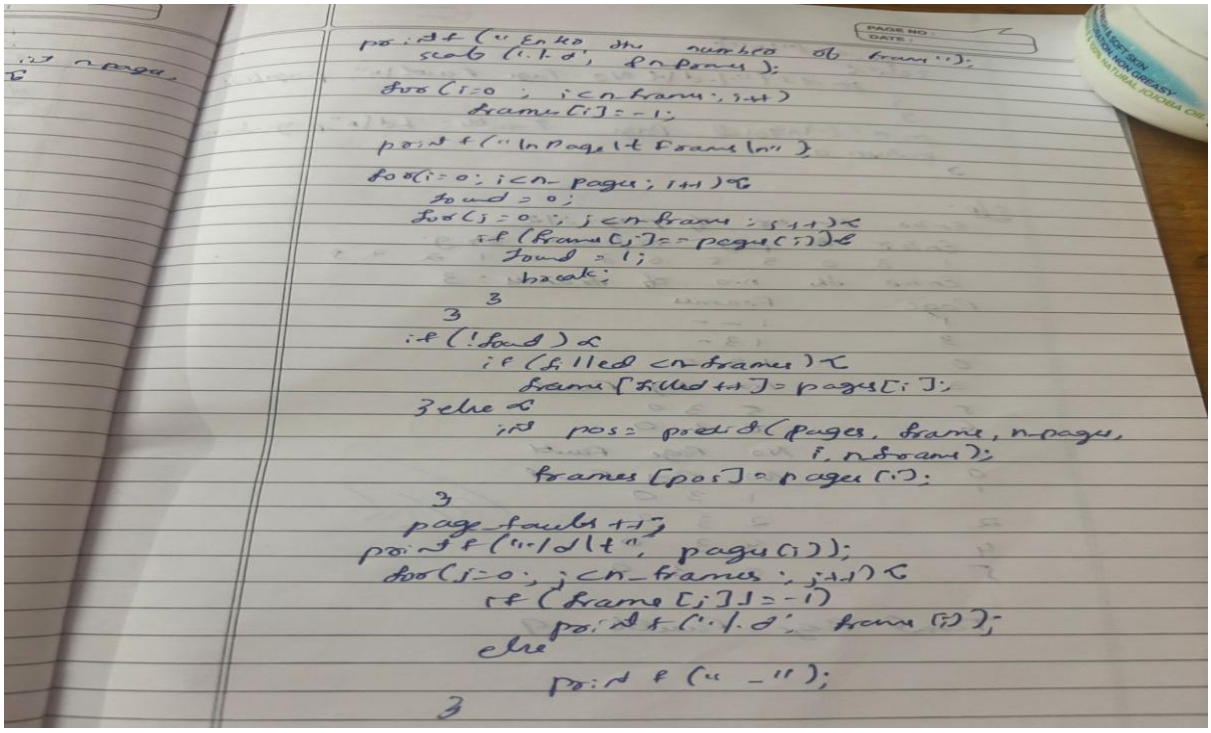
```
Page replacement LRU

#include<stdio.h>

int findLRU(int time[], int n){
    int i, min=time[0], pos=0;
    for(i=1; i<n; ++i){
        if(time[i]<min){
            min=time[i];
            pos=i;
        }
    }
    return pos;
}
int main(){
    int pages[100], frames[100], time[100];
    int n_pages, n_frames, i, j, pos, page_fault=0,
        count=0, found;
    printf("Enter the no of pages");
    scanf("%d", &n_pages);
    printf("Enter the page reference string");
    for(i=0; i<n_pages; i++){
        scanf("%d", &pages[i]);
    }
    printf("Enter the number of frames");
    scanf("%d", &pages[i]);
    for(i=0; i<n_frames; i++){
        frames[i]=-1;
    }
    printf("\nPage\t Frame\n");
```



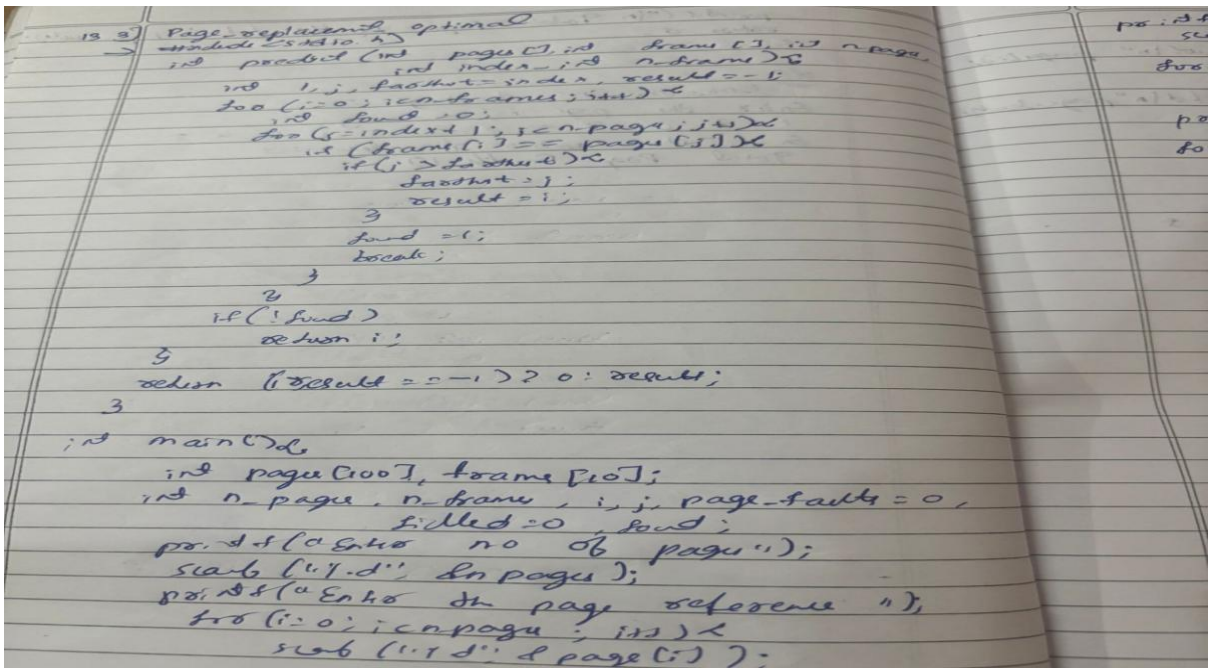
```
for(i=0; i<n_pages; i++){
    found = 0;
    for(j=0; j<n_frames; j++){
        if(frames[j] == pages[i]){
            counter++;
            frames[j] = pages[i];
            time[j] = counter;
            page_faults++;
            empty_found = 1;
            break;
        }
    }
    if(!empty_found){
        pos = findLRU(time, n_frames);
        counter++;
        frames[pos] = pages[i];
        time[pos] = counter;
        page_faults++;
    }
    printf("%d\t", pages[i]);
    for(j=0; j<n_frames; j++){
        if(frames[j] != -1)
            printf("%d\t", frames[j]);
        else
            printf("-\t");
    }
    printf("\n");
} else {
    printf("No Page Fault\n");
}
}
printf("\nTotal Page Faults: %d\n", page_faults);
return 0;
}
```

O/P

```
Enter the no of pages: 12
Enter the page reference string:
1 3 0 3 5 6 3 0 1 2 4 5
Enter the number of frames: 3
Total Page Faults: 10
```



```
printf("Enter the number of frames:");
scanf("%d", &n_frames);
for(i=0; i<n_frames; i++)
    frame[i] = -1;
printf("\n Page \t Frame\n");
for(i=0; i<n_pages; i++){
    found = 0;
    for(j=0; j<n_frames; j++){
        if(frame[j] == page[i]){
            found = 1;
            break;
        }
    }
    if(!found){
        if(filled < n_frames){
            frame[filled++] = page[i];
        }else{
            int pos = predict(pages, frame, n_pages,
                              i, n_frames);
            frames[pos] = page[i];
        }
        page_faults++;
    printf("%d\t%d\t", page[i]);
    for(j=0; j<n_frames; j++){
        if(frame[j] != -1)
            printf("%d", frame[j]);
        else
            printf("_");
    }
}
```



```
13 a) Page replacement optimal
#include<stdio.h>
int predict(int page[], int frame[], int n_pages,
            int index, int n_frames){
    int i, j, farthest = index, result = -1;
    for(i=0; i<n_frames; i++){
        int found = 0;
        for(j=index+1; j<n_pages; j++){
            if(frame[i] == page[j]){
                if(j > farthest){
                    farthest = j;
                    result = i;
                }
                found = 1;
                break;
            }
        }
        if(!found)
            return i;
    }
    return (result == -1) ? 0 : result;
}
int main(){
    int page[100], frame[10];
    int n_page, n_frame, i, j, page_faults = 0,
        filled = 0, found;
    printf("Enter no of pages:");
    scanf("%d", &n_pages);
    printf("Enter the page reference");
    for(i=0; i<n_pages; i++)
        scanf("%d", &page[i]);
```

```
p[i].turnaround_time = p[i].waiting_time + p[i].burst_time;
p[i].response_time = p[i].waiting_time;
*time += p[i].burst_time;
}
}
```

O/P Enter number of processes: 4
Enter Burst Time, Arrival Time and Queue of P1 : 2 0 1
" " " " " " P2 : 1 0 2
" " " " " " P3 : 5 0 1
" " " " " " P4 : 3 0 2

Average waiting time = 4.25
Over
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 : 11.5s