VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

OPERATING SYSTEMS

Submitted by

Vijay J(1WA23CS040)

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 Vijay J(1WA23CS040), who is Bonafide student of B. M. S. College of Engineering. It is in partial fulfilment for the award of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum during the year Feb 2025- June 2025. The Lab report has been approved as it satisfies the academic requirements in respect of a OPERATING SYSTEMS - (23CS4PCOPS) work prescribed for the said degree.

Prof. Sheetal V A
Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

Index Sheet

10.	Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) Optimal	116-127

Course Outcomes

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

Program 1:

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

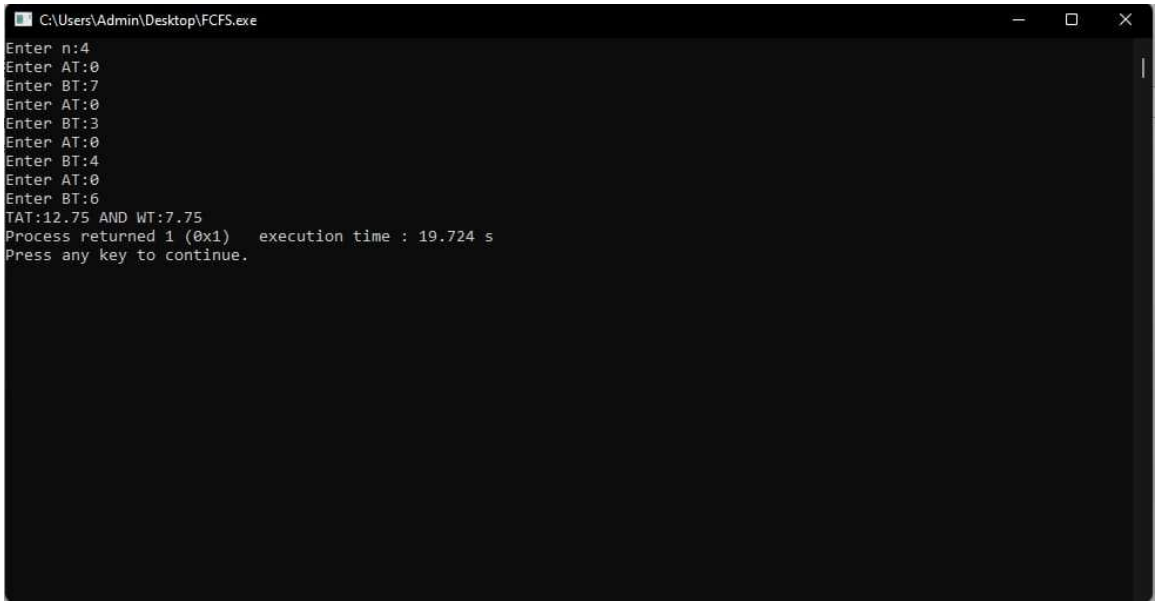
→FCFS

→ SJF (pre-emptive & Non-preemptive)

→FCFS

```
#include <stdio.h>

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

void calculateFCFS(struct Process proc[], int n) {
    int time = 0;
    for (int i = 0; i < n; i++) {
        if (time < proc[i].at) {
            time = proc[i].at;
        }
        proc[i].ct = time + proc[i].bt;
        proc[i].tat = proc[i].ct - proc[i].at;
        proc[i].wt = proc[i].tat - proc[i].bt;
        proc[i].rt = time - proc[i].at;
        time = proc[i].ct;
    }
}
int main() {
```

```
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    struct Process proc[n];
    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("P%d Arrival Time: ", i + 1);
        scanf("%d", &proc[i].at);
        printf("P%d Burst Time: ", i + 1);
        scanf("%d", &proc[i].bt);
    }
    calculateFCFS(proc, n);
    printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n", i + 1, proc[i].at, proc[i].bt, proc[i].ct, proc[i].tat,
proc[i].wt, proc[i].rt);
    }
    float totalWT = 0, totalTAT = 0;
    for (int i = 0; i < n; i++) {
        totalWT += proc[i].wt;
        totalTAT += proc[i].tat;
    }
    printf("\nAverage Waiting Time: %.2f", totalWT / n);
    printf("\nAverage Turnaround Time: %.2f\n", totalTAT / n);
    return 0;
}
```


C:\Users\Admin\Desktop\FCFS.exe
Enter n:4
Enter AT:0
Enter BT:7
Enter AT:0
Enter BT:3
Enter AT:0
Enter BT:4
Enter AT:0
Enter BT:6
TAT:12.75 AND WT:7.75
Process returned 1 (0x1)   execution time : 19.724 s
Press any key to continue.

Lab-01

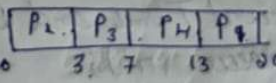
write a C program to simulate non-preemptive CPU scheduling algorithm to find turn around time & waiting time.

FCFS

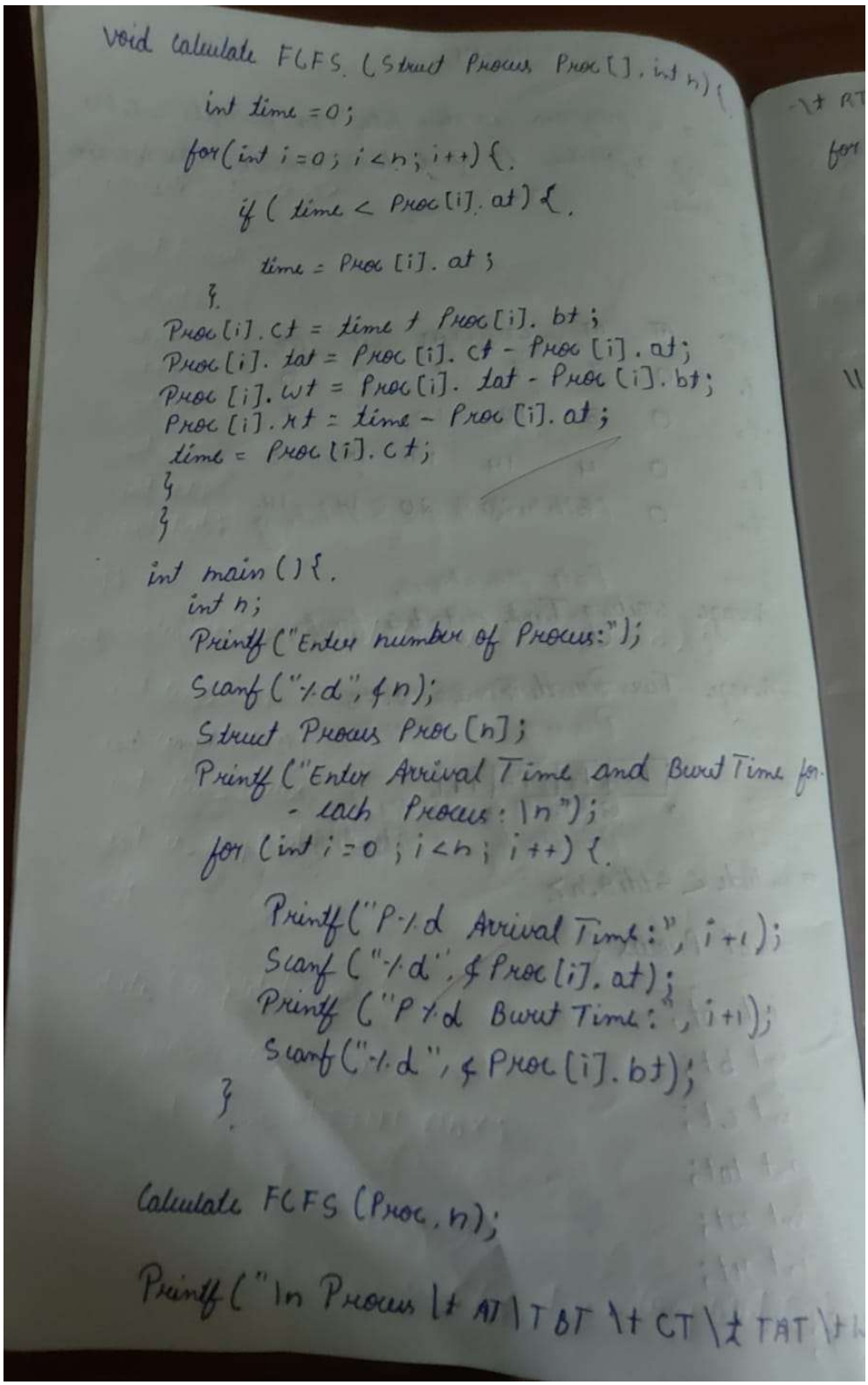
Process	AT	BT	CT	TAT	WT	RT
P_1	0	7	7	7	0	0
P_2	0	3	10	10	7	7
P_3	0	4	14	14	10	10
P_4	0	6	20	20	14	14

Average waiting Time = 7.75

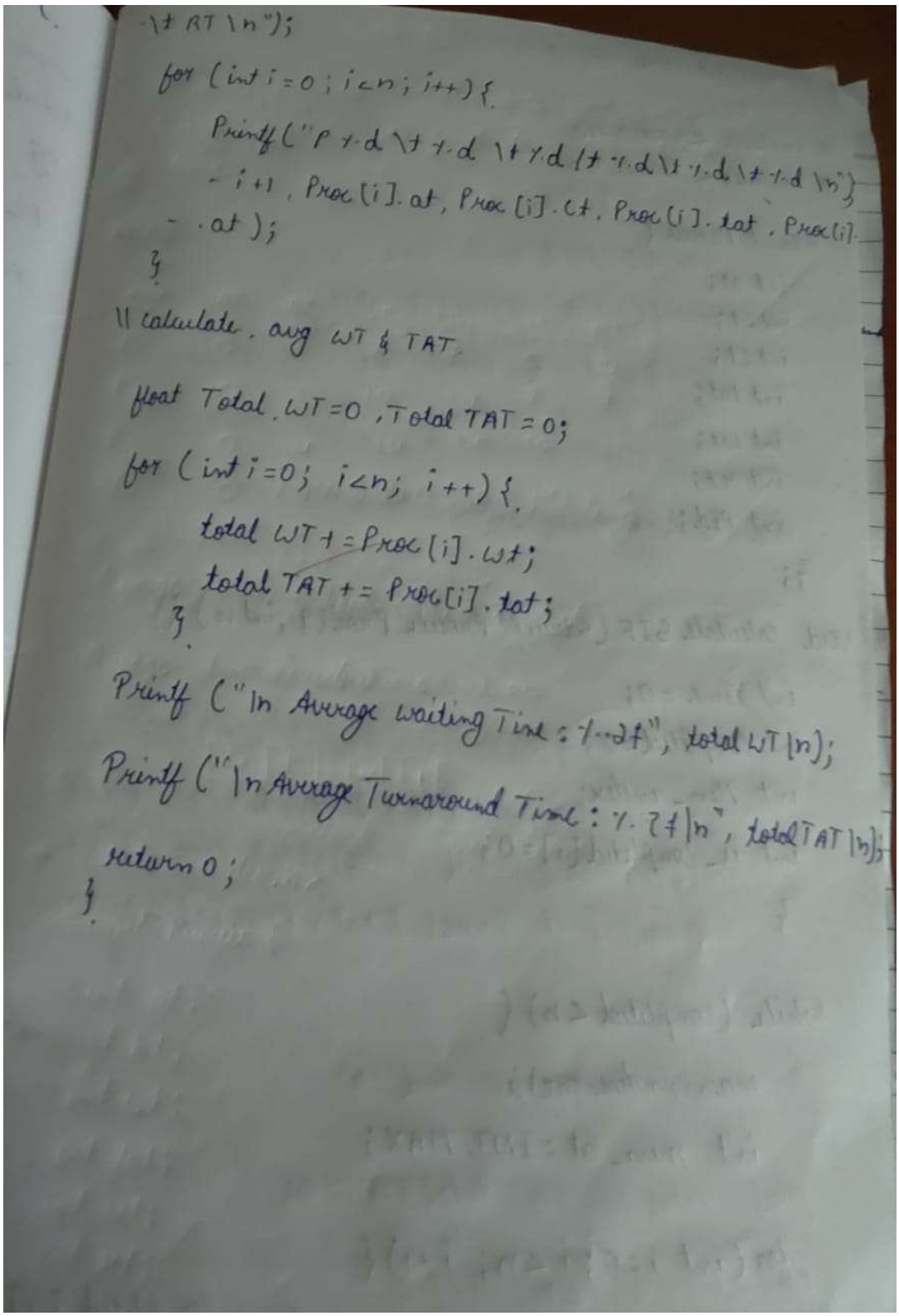
Average Turnaround Time = 12.75




```
#include <stdio.h>
Struct Process {
  int at;
  int bt;
  int ct;
  int tat;
  int wt;
  int rt;
};
```



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

int main(){
    int n;
    Printf("Enter number of Process:");
    Scanf("%d", &n);
    Struct Process Proc[n];
    Printf("Enter Arrival Time and Burst Time for
        - each Process: \n");
    for(int i=0; i<n; i++){
        Printf("P%d Arrival Time:", i+1);
        Scanf("%d", &Proc[i].at);
        Printf("P%d Burst Time:", i+1);
        Scanf("%d", &Proc[i].bt);
    }

Calculate FCFS (Proc, n);

Printf("\n Process \t AT \t BT \t CT \t TAT \t
```



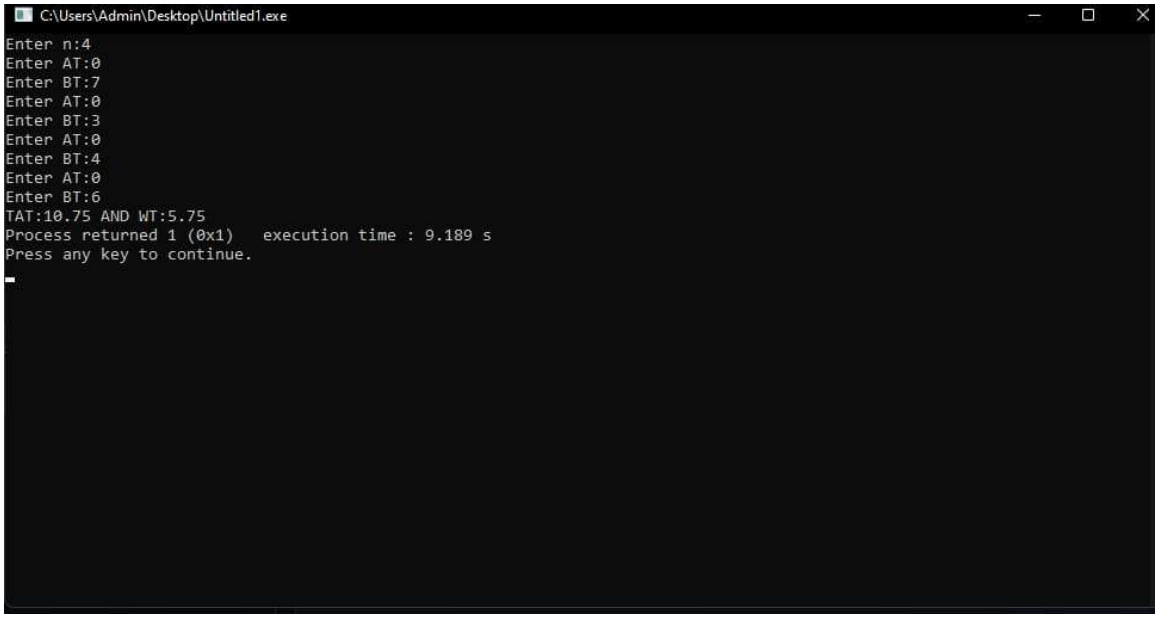
```
\t AT \n");
for (int i=0; i<n; i++){
    Printf("P %d \t %d \t %d \t %d \t %d \t %d \n",
    - i+1, Proc[i].at, Proc[i].ct, Proc[i].tat, Proc[i]
    - .at);
}
// calculate avg WT & TAT
float Total WT=0, Total TAT=0;
for (int i=0; i<n; i++){
    total WT += Proc[i].wt;
    total TAT += Proc[i].tat;
}
Printf ("\n Average waiting Time: %.2f", total WT/n);
Printf ("\n Average Turnaround Time: %.2f\n", total TAT/n);
return 0;
}
```

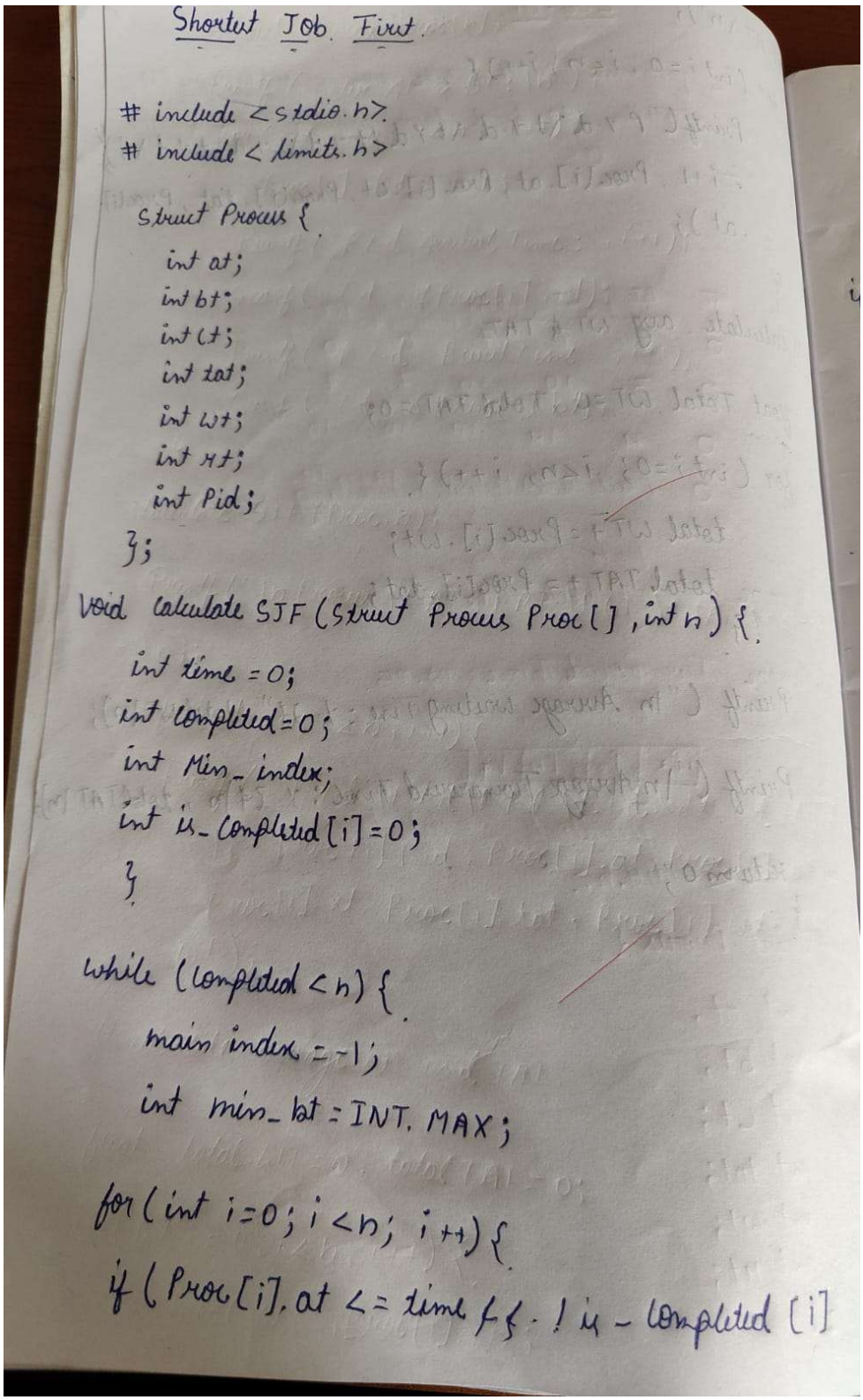
→SJF(Non Preemptive)

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

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

```
    }
    return 0;
}
```


C:\Users\Admin\Desktop\Untitled1.exe
Enter n:4
Enter AT:0
Enter BT:7
Enter AT:0
Enter BT:3
Enter AT:0
Enter BT:4
Enter AT:0
Enter BT:6
TAT:10.75 AND WT:5.75
Process returned 1 (0x1)   execution time : 9.189 s
Press any key to continue.



Shortest Job First.

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

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

Void calculate SJF (struct Process Proc[], int n) {
    int time = 0;
    int completed = 0;
    int Min_index;
    int is_completed[i] = 0;
};

while (completed < n) {
    main index = -1;
    int min_bt = INT. MAX;

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

```
        if (Proc[i].bt < min.bt)
        {
            min_bt = Proc[i].bt;
            Min_index = i;
        }
    }
if (min_index == -1) {
    time++;
} else {
Proc[min_index].ct = time + Proc[Min_index].bt;
Proc[Min_index].tat = Proc[Min_index].ct = Proc[-
-min_index].at;
Proc[Min_index].wt = Proc[Min_index].tat - Proc[-
-Min_index].bt;
Proc[Min_index].rt = time - Proc[Min_index].at;
time = Proc[min_index].ct;
is_completed[Min_index] = 1;
completed++;
    }
}
int main(){
int n;
Printf("Enter number of Processes:");
Scanf("%d", &n);
```

```
total TAT+= Proc[i].tat;
}
Printf("\n Average waiting Time: %.2f", total wt/n);
Printf("\n Average Turnaround Time: %.2f\n", total T/n);
return 0;
}
```

O/P

Enter number of Process: 4.

Enter AT and BT for each:

P_1 AT = 6.

P_1 BT = 7.

P_2 AT = 0

P_2 BT = 3.

P_3 AT = 0

P_3 BT = 4

P_4 AT = 0

P_4 BT = 6.

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

→SJF(Pre-emptive)

```
#include <stdio.h>

#define MAX 10

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

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

   while (completed < n) {
      shortest = -1;
      min_bt = 9999;

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

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

      p[shortest].rt--;
      time++;
```

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

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

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

    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("Enter arrival time and burst time for process %d: ", p[i].pid);
        scanf("%d %d", &p[i].at, &p[i].bt);
        p[i].rt = p[i].bt;
        p[i].completed = 0;
    }

    sjf_preemptive(p, n);

    printf("\nPID\tAT\tBT\tWT\tTAT\n");
    for (int i = 0; i < n; i++)
        printf("%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].wt, p[i].tat);
```

```
    return 0;
}
```

```
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
```

SJF Preemptive

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

typedef Struct {
        int Pid, at, bt, rt, wt, tat, completed;
} Process;

void SJF-Preemptive (Process P[], int n) {
        int time=0, completed=0, Shortest = -1;
        min. bt = 9999,

while (completed<n) {
        Shortest = -1;
        min-bt = 9999;

    for (int i=0; i<n; i++) {
        if (P[i].at <= time && P[i].rt > 0 &&
        P[i].rt < min_bt) {
                min bt = P[i].rt;
                Shortest = i;
        }
}

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

```
    P[shortest].rt --;
    time ++;
    if (P[shortest].rt == 0) {
        P[shortest].completed = 1;
        completed++;
        P[shortest].tat = time - P[shortest].at;
        P[shortest].wt = P[shortest].tat - P[shortest
    - est].bt;

int main() {
    Process P[MAX];
    int n;
    Printf("Enter the number of Process:");
    Scanf("%d", &n);
    for(int i=0; i<n; i++) {
        P[i].Pid = i+1;
    Printf("Enter arrival time and burst time for
    - Process %d:", P[i], Pid);
    Scanf("%d %d", &P[i].at, &P[i].bt);
    P[i].rt = P[i].bt;
    P[i].completed = 0;
}
Sjf - Premptive (P, n);
Printf("\n Pid \t AT \t BT \t WT \t TAT \n");
```

```
for (int i=0; i<n; i++){
    Printf("%d \t %d \t %d \t %d \t %d \n");
    for(int i=0; i<n; i++){
        Printf("%d \t %d \t %d \t %d \t %d \n");
P[i].Pid, P[i].at, P[i].bt, P[i], ut, P[i].tat);
return 0;
}
```

O/P

Enter the number of Process : 4

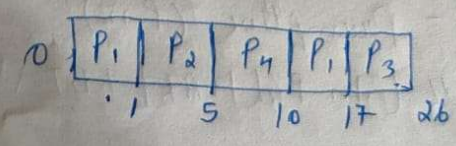
Enter the arrival time and burst time Process 1: 0;
8

Enter the arrival time and burst time for Process 2: 1
4

Enter the arrival time and burst time for Process 3: 2
9

Enter the arrival time and burst time for Process 4: 3
5.

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



Program 2:

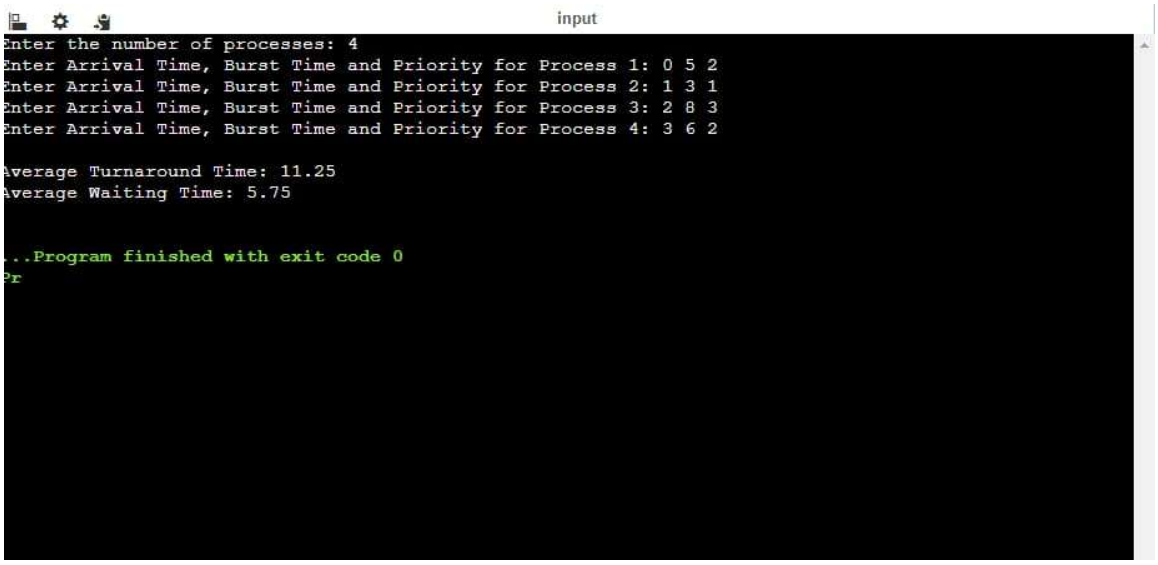
Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.
→ Priority (pre-emptive & Non-pre-emptive)
→Round Robin (Experiment with different quantum sizes for RR algorithm)

→Priority(Non-pre-emptive)

```
#include <stdio.h>


struct Process {
   int pid, at, bt, pr, ct, wt, tat, rt;
   int isCompleted; // Flag to check if process is completed
};


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


void findPriorityScheduling(struct Process p[], int n) {
   sortByArrival(p, n);
   int time = 0, completed = 0;
   float totalWT = 0, totalTAT = 0;
```

```
while (completed < n) {
    int idx = -1, highestPriority = 9999;

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

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

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

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

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

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

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

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

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


input
Enter the number of processes: 4
Enter Arrival Time, Burst Time and Priority for Process 1: 0 5 2
Enter Arrival Time, Burst Time and Priority for Process 2: 1 3 1
Enter Arrival Time, Burst Time and Priority for Process 3: 2 8 3
Enter Arrival Time, Burst Time and Priority for Process 4: 3 6 2
Average Turnaround Time: 11.25
Average Waiting Time: 5.75
...Program finished with exit code 0
Pr

Priority Non-Preemptive :-

```
#include <stdio.h>

struct Process {
    int pid, at, bt, pr, ct, wt, tat, rt;
    int is_completed;
};

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

void findPriorityScheduling(struct Process p[], int n) {
    SortByArrival(p, n);
    int time=0; completed=0;
    float totalWT=0, totalTAT=0;

    while(completed<n) {
    int idx=-1, highestPriority=9999;
    for(int i=0; i<n; i++) {
```

```
        if (P[i].at <= time && P[i].isCompleted == 0) {
            if (P[i].Pr < highestPriority) {
                highestPriority = P[i].Pr;
                idx = i;
            }
        }
    }

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

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

Printf("PID\tAT\tBT\tPR\tCT\tTAT\tWT\tRT\n");
for (int i=0; i<n; i++) {
```

```
Printf ("%d \t %d \t %d \t %d \t %d \t %d \t %d \t %d \n");
    P[i].Pid, P[i].at, P[i].bt, P[i].Pr, P[i].ct, P[i].tat,
    P[i].wt, P[i].wtt);
}
Printf("\n Average Turnaround Time: %.2f \n", total TAT /n);
Printf("Average waiting Time: %.2f \n", total WT /n);
}
int main (){
    int n;
    Printf ("Enter the number of Process:");
    Scanf ("%d", &n);
    Struct Process P[n];
    Printf ("Enter. Arrival Time, Burst Time, and Priority for
    each Process: \n");
    for (int i=0; i<n; i++) {
        P[i].Pid = i+1;
        Printf("Process %d:", i+1);
        Scanf("%d %d %d", &P[i].at, &P[i].bt, &P[i].Pr);
        P[i].is completed=0;
    }
find Priority Scheduling [P, n];
return 0;
}
```

O/P

Enter the number of Process : 5

Enter Arrival time, Burst time, & Priority for each Process,

Process 1: 0 3 5

Process 2; 2 2 3

Process 3; 3 5 2

Process 4: 4 4 4

Process 5 : 6 1 1

PID	AT	BT	PR	CT	TAT	WT	RT
1	0	3	5	3	3	0	0
2	2	2	3	11	9	7	7
3	3	5	2	8	5	0	0
4	4	4	4	15	11	7	7
5	6	1	1	9	3	2	2.

Average Turnaround Time: 6.20

Average waiting Time : 3.20

→Priority(Pre-emptive)

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

struct Process {
    int pid, at, bt, pr, ct, wt, tat, rt, remaining;
};

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

    for (int i = 0; i < n; i++) {
        p[i].remaining = p[i].bt;
        p[i].rt = -1;
    }

    while (completed != n) {
        int min_priority = INT_MAX;
        min_idx = -1;

        for (int i = 0; i < n; i++) {
            if (p[i].at <= time && p[i].remaining > 0 && p[i].pr < min_priority) {
                min_priority = p[i].pr;
                min_idx = i;
            }
        }

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

```
            continue;
        }

        if (p[min_idx].rt == -1) {
            p[min_idx].rt = time - p[min_idx].at;
        }

        p[min_idx].remaining--;
        time++;

        if (p[min_idx].remaining == 0) {
            completed++;
            p[min_idx].ct = time;
            p[min_idx].tat = p[min_idx].ct - p[min_idx].at;
            p[min_idx].wt = p[min_idx].tat - p[min_idx].bt;
            totalWT += p[min_idx].wt;
            totalTAT += p[min_idx].tat;
        }
    }

    printf("PID\tAT\tBT\tPR\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
               p[i].pid, p[i].at, p[i].bt, p[i].pr, p[i].ct, p[i].tat, p[i].wt, p[i].rt);
    }

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

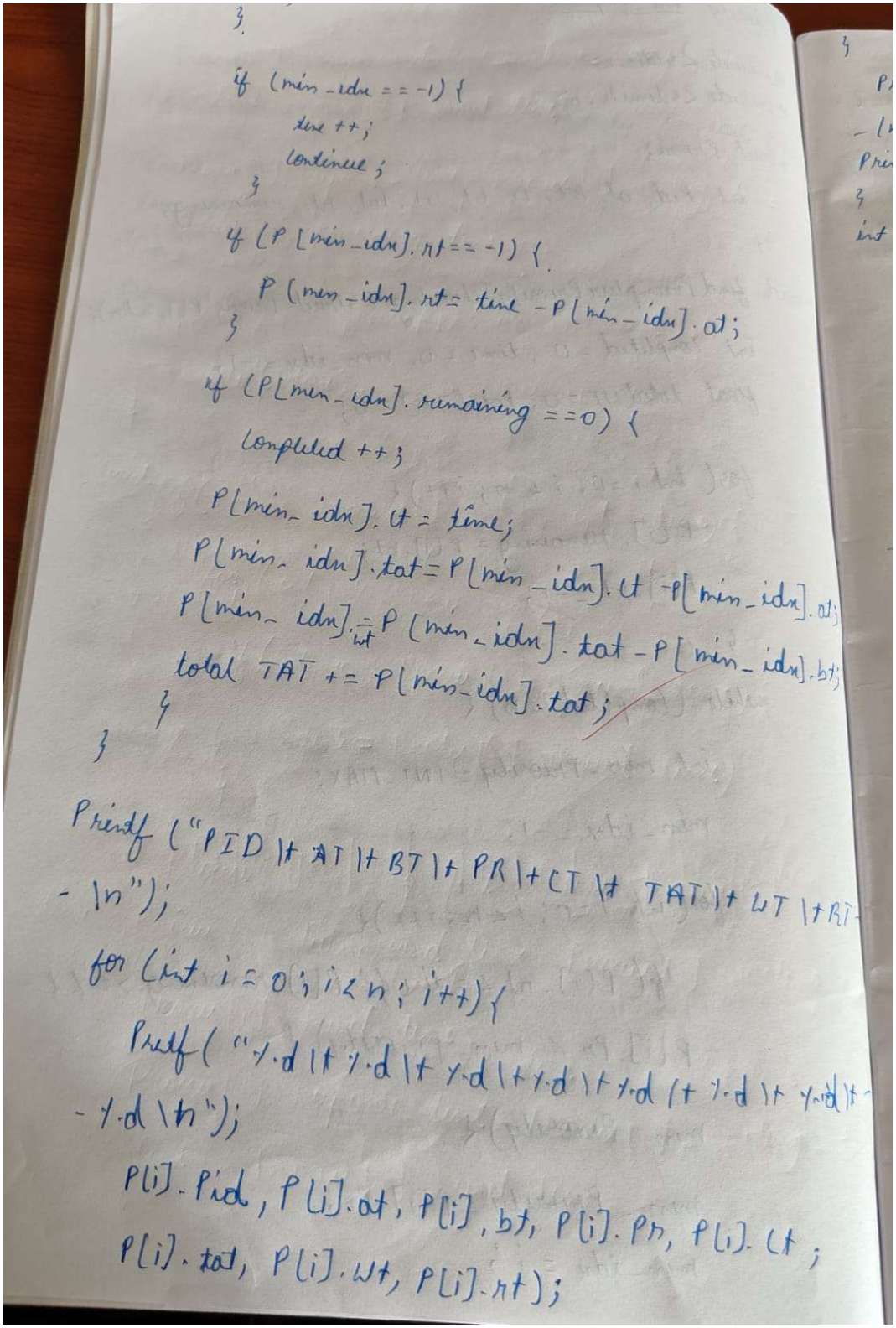
```
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process p[n];

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

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

Output Clear

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

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

Priority Pre-emptive

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

Struct Process{
    int Pid, at, bt, Pr, ct, wt, tat, rt, remaining;
};

void findPreemptivePriorityScheduling (Struct Process P[], int n){
    int completed = 0, time = 0, min_idx = -1;
    float totalWT = 0, totalTAT = 0;

    for( int i = 0; i < n; i++){
        P[i].remaining = P[i].bt;
        P[i].rt = -1;
    }

    while (completed != n) {
        int min_Priority = INT_MAX;
        min_idx = -1;

        for( int i = 0; i < n; i++){
            if( P[i].at <= time && P[i].remaining > 0 && -
            - P[i].Pr < min_Priority) {

            min_Priority) {
            min_Priority = P[i].Pr;
            min_idx = i;
```



```
}

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

if (P[min_idx].rt == -1) {
    P[min_idx].rt = time - P[min_idx].at;
}

if (P[min_idx].remaining == 0) {
    completed++;
    P[min_idx].ct = time;
    P[min_idx].tat = P[min_idx].ct - P[min_idx].at;
    P[min_idx].wt = P[min_idx].tat - P[min_idx].bt;
    total_TAT += P[min_idx].tat;
}
}

Printf("PID \t AT \t BT \t PR \t CT \t TAT \t WT \t RT
- \n");
for (int i = 0; i < n; i++) {
    Printf("%d \t %d \t %d \t %d \t %d \t %d \t %d \t %d
- %d \n");
    P[i].Pid, P[i].at, P[i], bt, P[i].Pr, P[i].ct;
    P[i].tat, P[i].wt, P[i].rt);
```

```
}
    Printf ("\n Average Turnaround Time : %.2f\n", totalTAT
    _[n]);
    Printf("Average waiting Time : %.2f\n", totalWT[n]);
}
int main() {
    int n;
    Printf ("Enter the number of Process: ");
    Scanf("%d", &n);
    Struct Process P[n];

    Printf ("Enter Arrival Time, Burst Time, and Priority for
    each Process: \n");

    for(int i=0; i<n; i++){
        P[i].Pid = i+1;
        Printf ("Process %d: ", i+1);
        Scanf("%d %d %d", &P[i].at, &P[i].bt, &P[i].
    Pr);
    }
find PreemptivePriorityScheduling(P,n);
return 0;
}
```

O/P

Enter the number of Process : 7.

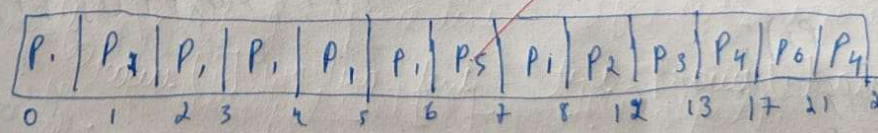
Enter Arrival Time, Burst time, and Priority for each Process:

Process 1 : 0 8 3
Process 2 : 1 2 4
Process 3 : 3 4 4
Process 4 : 4 1 5
Process 5 : 5 6 2
Process 6 : 6 5 6
Process 7 : 7 1 1

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

Average Turnaround Time : 13.71

Average waiting Time : 9.86



20-3-25

→Round Robin

```
#include <stdio.h>

#define MAX 100

void roundRobin(int n, int at[], int bt[], int quant) {
    int ct[n], tat[n], wt[n], rem_bt[n];
    int queue[MAX], front = 0, rear = 0;
    int time = 0, completed = 0, visited[n];

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

    queue[rear++] = 0;
    visited[0] = 1;

    while (completed < n) {
        int index = queue[front++];

        if (rem_bt[index] > quant) {
            time += quant;
            rem_bt[index] -= quant;
        } else {
            time += rem_bt[index];
            rem_bt[index] = 0;
            ct[index] = time;
            completed++;
        }
```

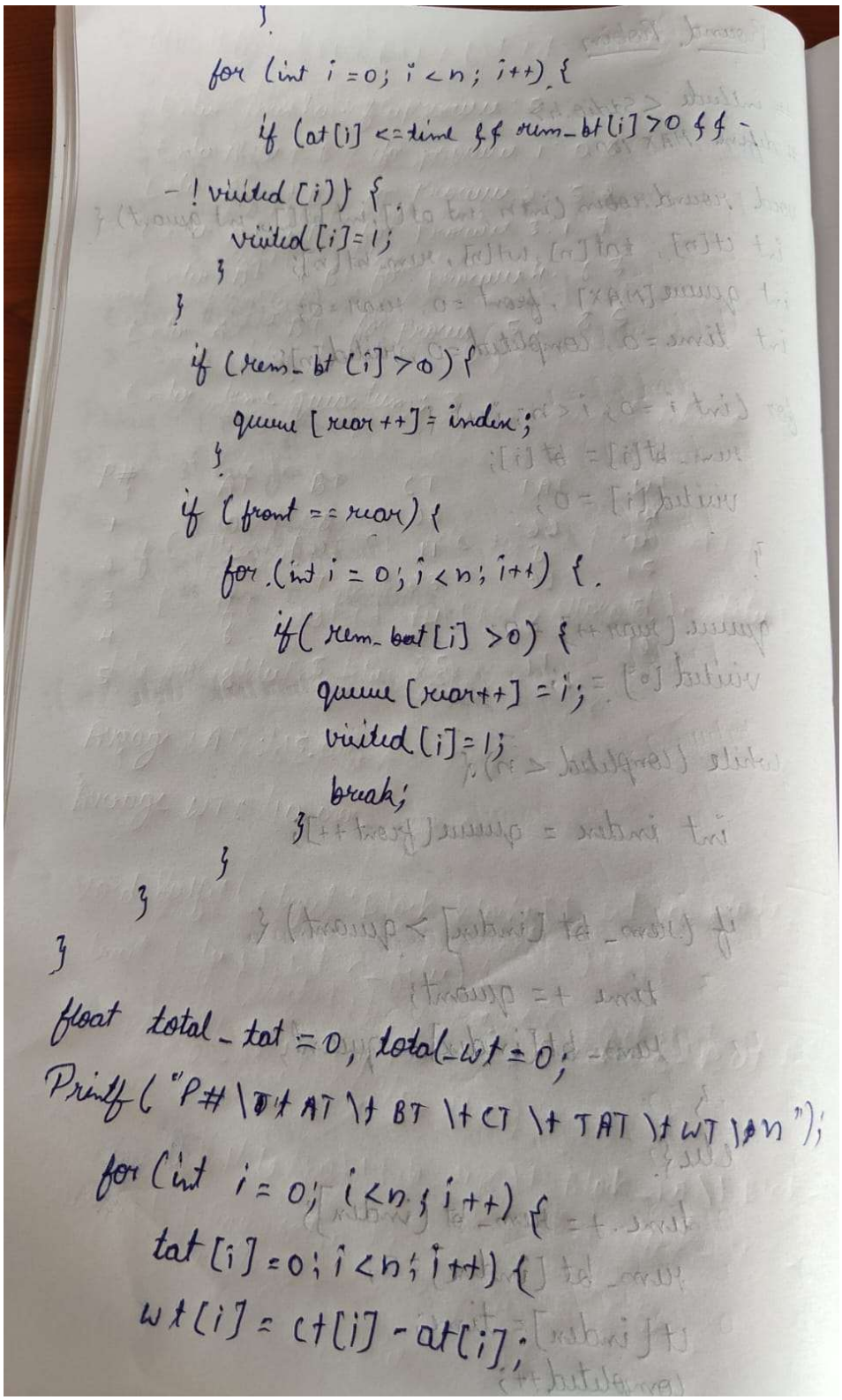
```
    for (int i = 0; i < n; i++) {
        if (at[i] <= time && rem_bt[i] > 0 && !visited[i]) {
            queue[rear++] = i;
            visited[i] = 1;
        }
    }

    if (rem_bt[index] > 0) {
        queue[rear++] = index;
    }

    if (front == rear) {
        for (int i = 0; i < n; i++) {
            if (rem_bt[i] > 0) {
                queue[rear++] = i;
                visited[i] = 1;
                break;
            }
        }
    }
}

float total_tat = 0, total_wt = 0;
printf("P#\tAT\tBT\tCT\tTAT\tWT\n");
for (int i = 0; i < n; i++) {
    tat[i] = ct[i] - at[i];
    wt[i] = tat[i] - bt[i];
    total_tat += tat[i];
    total_wt += wt[i];
```

```
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i], ct[i], tat[i], wt[i]);
    }

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

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

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

    printf("Enter time quantum: ");
    scanf("%d", &quant);

    roundRobin(n, at, bt, quant);
    return 0;
}
```

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

Round Robin

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

void roundrobin (int n, int at[], int bt[], int quant) {
 int ct[n], tat[n], wt[n], rem_bt[n];
 int queue[MAX], front=0, rear=0;
 int time=0, completed=0, visited[n];

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

 queue[rear++] = 0;
 visited[0] = 1;

while (completed < n) {
    int index = queue[front++];

    if (rem_bt[index] > quant) {
        time += quant;
        rem_bt[index] -= quant;
    }
    else {
        time += rem_bt[index];
        rem_bt[index] = 0;
        ct[index] = time;
        completed++;
```



```
    }
    for (int i = 0; i < n; i++) {
        if (at[i] <= time && rem_bt[i] > 0 &&
    - !visited[i]) {
        visited[i] = 1;
    }
}
if (rem_bt[i] > 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# \t AT \t BT \t CT \t TAT \t WT \n");
    for (int i = 0; i < n; i++) {
    tat[i] = 0; i < n; i++) {
    wt[i] = ct[i] - at[i];
```

```
    total_tat += tat[i];
    total_wt += wt[i];
    Printf("%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 Process: ");
    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

Enter number of process : 5
Enter AT & BT for process 1 : 0 8
Enter AT & BT for Process 2 : 5 2
Enter AT & BT for Process 3 : 1 7
Enter AT & BT for Process 4 : 6 3
Enter AT & BT for Proces : 5 : 8 5

Enter time quantum : 3

P#	AT	BT	CT	TAT	WT
1	0	8	22	22	14
2	5	2	11	6	4
3	1	7	23	22	15
4	6	3	14	8	5
5	8	5	25	17	12

Average TAT : 15.00
Average WT : 10.00

Program 3:

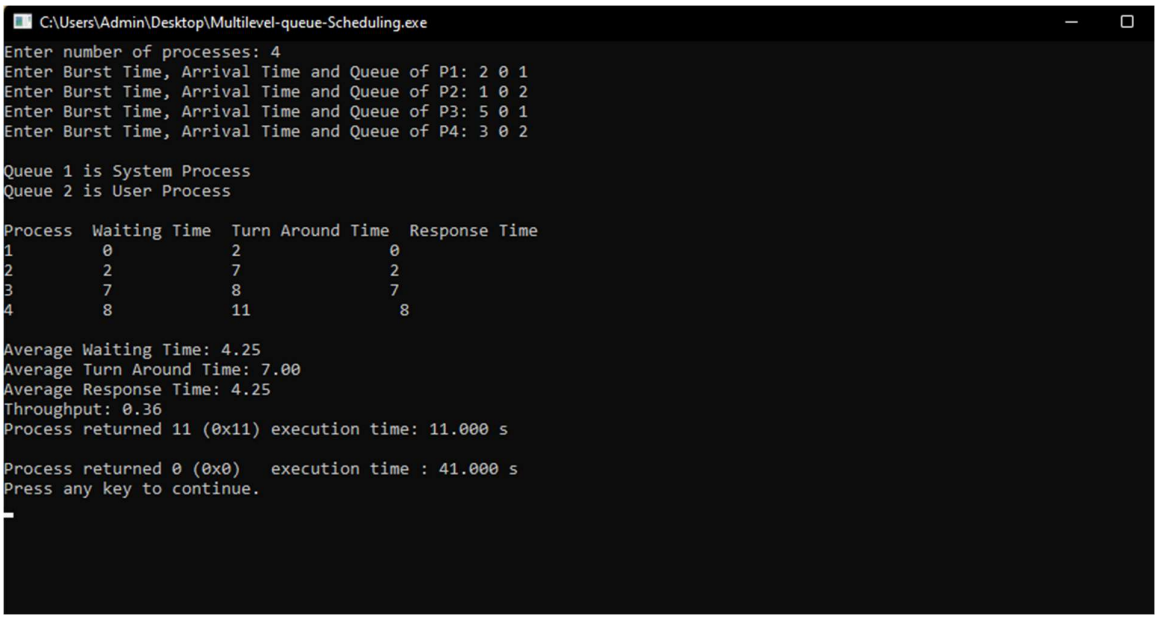
Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue.

```
#include <stdio.h>


#define MAX_PROCESSES 10

#define TIME_QUANTUM 2


typedef struct {

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

} Process;


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

   int done, i;

   do {

      done = 1;

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

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

            done = 0;

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

               *time += time_quantum;

               processes[i].remaining_time -= time_quantum;

            } else {

               *time += processes[i].remaining_time;

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

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

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

               processes[i].remaining_time = 0;
```

```
            }
        }
    }
  } while (!done);
}

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

int main() {
  Process processes[MAX_PROCESSES], system_queue[MAX_PROCESSES],
user_queue[MAX_PROCESSES];
  int n, sys_count = 0, user_count = 0, time = 0;
  float avg_waiting = 0, avg_turnaround = 0, avg_response = 0, throughput;

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

  for (int i = 0; i < n; i++) {
    printf("Enter Burst Time, Arrival Time and Queue of P%d: ", i + 1);
```

```
        scanf("%d %d %d", &processes[i].burst_time, &processes[i].arrival_time,
&processes[i].queue_type);
        processes[i].remaining_time = processes[i].burst_time;

        if (processes[i].queue_type == 1) {
            system_queue[sys_count++] = processes[i];
        } else {
            user_queue[user_count++] = processes[i];
        }
    }

    // Sort user processes by arrival time for FCFS
    for (int i = 0; i < user_count - 1; i++) {
        for (int j = 0; j < user_count - i - 1; j++) {
            if (user_queue[j].arrival_time > user_queue[j + 1].arrival_time) {
                Process temp = user_queue[j];
                user_queue[j] = user_queue[j + 1];
                user_queue[j + 1] = temp;
            }
        }
    }

    printf("\nQueue 1 is System Process\nQueue 2 is User Process\n");
    round_robin(system_queue, sys_count, TIME_QUANTUM, &time);
    fcfs(user_queue, user_count, &time);

    printf("\nProcess  Waiting Time  Turn Around Time  Response Time\n");

    for (int i = 0; i < sys_count; i++) {
        avg_waiting += system_queue[i].waiting_time;
```

```
        avg_turnaround += system_queue[i].turnaround_time;
        avg_response += system_queue[i].response_time;
        printf("%d        %d        %d        %d\n", i + 1, system_queue[i].waiting_time,
system_queue[i].turnaround_time, system_queue[i].response_time);
    }

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

    avg_waiting /= n;
    avg_turnaround /= n;
    avg_response /= n;
    throughput = (float)n / time;

    printf("\nAverage Waiting Time: %.2f", avg_waiting);
    printf("\nAverage Turn Around Time: %.2f", avg_turnaround);
    printf("\nAverage Response Time: %.2f", avg_response);
    printf("\nThroughput: %.2f", throughput);
    printf("\nProcess returned %d (0x%d) execution time: %.3f s\n", time, time, (float)time);

    return 0;
}
```


C:\Users\Admin\Desktop\Multilevel-queue-Scheduling.exe
Enter number of processes: 4
Enter Burst Time, Arrival Time and Queue of P1: 2 0 1
Enter Burst Time, Arrival Time and Queue of P2: 1 0 2
Enter Burst Time, Arrival Time and Queue of P3: 5 0 1
Enter Burst Time, Arrival Time and Queue of P4: 3 0 2
Queue 1 is System Process
Queue 2 is User Process
Process Waiting Time Turn Around Time Response Time
1 0 2 0
2 2 7 2
3 7 8 7
4 8 11 8
Average Waiting Time: 4.25
Average Turn Around Time: 7.00
Average Response Time: 4.25
Throughput: 0.36
Process returned 11 (0x11) execution time: 11.000 s
Process returned 0 (0x0) execution time : 41.000 s
Press any key to continue.

→ Multilevel Queue Scheduling

```
#include <stdio.h>
#define MAX_Process 10
#define TIME_QUANTUM 2

typedef struct {
    int burst_time, arrival_time, queue_time, queue_time,
waiting_time, turnaround_time, response_time, remaining_time; } Process;
void round_robin (Process Processor[], int n, int time_quantum, int *time) {
    int done, i;
    do {
        done = 1;
        for (i=0; i<n; i++) {
            if (Process[i].remaining_time > 0) {
                done = 0;
                if (Process[i].remaining_time > time_quantum) {
                    *time += time_quantum;
                    Process[i].remaining_time -= time_quantum;
                } else {
                    *time += process[i].remaining_time;
                    Process[i].waiting_time = *time - process[i].arrival_
```

```
        time = proces[i].burst_time;
        Proces[i].turnaround_time = *time - Proces[i].arrival
            -time - Proces[i].burst_time;
        Proces[i].response_time = Proces[i].waiting_time;
        Proces[i].remaining_time = 0;
            }
        }
    }
    } while (!done);
}

void fff(Proces Proces[], int n; int *time) {
        for (int i=0; i<n; i++){
            if (*time < Proces[i].arrival_time) {
                *time = Proces[i].arrival_time)
            }
        Proces[i].waiting_time = *time - Proces[i].arrival
_time, Proces[i].turnaround_time = Proces[i].waiting_time
        Proces[i].burst_time;
        Proces[i].response_time = Proces[i].waiting_time
        *time += Proces[i].burst_time;
        }
}
```

```
int main() {
    Proces Process [MAX_Process], system_queue[MAX_
- Process], user_queue [MAX_Process];
    int n, sys_count=0; user_count=0, time=0; float -
- avg_waiting=0; avg_waiting=0; avg_turnaround=0,
    avg_response=0, throughput;
    Printf("Enter the number of Process:");
    Scanf("%d", &n);
    for(int i=0; i<n; i++) {
        Printf("Enter BT, AT and Queue of P%d:", i+1);
        Scanf("%d %d %d", &Process[i].burst_time,
        &Process[i].arrival_time, &Process[i].queue_type);
        Process[i].remaining_time = Process[i].burst_time;
        if(Process[i].queue_type==1){
            system_queue[system_queue[sys_count++]=
        - Process[i];
    } else {
        user_queue[user_count++] = Process[i];
        }
}
```

```
for (int i=0; i<usr_count; i++){
    avg_waiting += usr_queue[i].waiting_time;
    avg_turnaround += usr_queue[i].turnaround_time;
    avg_response += usr_queue[i].response_time;
    Printf("%d \t %d \t %d \n", P+1, system_queue[i]
    - waiting_time, system_queue[i].turnaround
    - time, usr_queue[i].response_time);
}
    avg_waiting /= n;
    avg_turnaround /= n;
    avg_response /= n;
    throughput = float(n)/time;
Printf("\n Avage waiting time %.2f", avg_waiting);
Printf("\n Avage turnaround time: %.2f", avg_turnaro-
-around);
Printf("\n Avrag response time : %.2f", Avg_response);
Printf("\n throughput : %.2f", throughput);
Printf("\n Process returned %d (0x%d) execution
time : %.3f s\n", time, time, (float) time);
return 0;
}
```

output:

Enter the number of process: 4.

Enter BT, AT and Queue of P1: 2 0 1
Enter BT, AT and Queue of P2: 1 0 2
Enter BT, AT and Queue of P3: 5 0 1
Enter BT, AT and Queue of P4: 3 0 2

Queue 1 is system Process.
Queue 2 is User Process.

Process	waiting time	Turnaround time	Response time.
1	0	2	0
2	2	7	2
3	7	8	7
4.	8	11	8

Average waiting Time: 4.25
Avrge Turnaround Tim: 7.00
Average response time: 4.25
Throughput: 0.36
Process returned (0x11) execution time: 11.0005

Program 4:
Write a C program to simulate Real-Time CPU Scheduling algorithms:
a) Rate- Monotonic
b) Earliest-deadline First

→Rate- Monotonic

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

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

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

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

int findLCM(Task tasks[], int n) {
    int result = tasks[0].period;
    for (int i = 1; i < n; i++)
        result = lcm(result, tasks[i].period);
    return result;
}

void rateMonotonic(Task tasks[], int n) {
    float utilization = 0;
    printf("\nRate Monotonic Scheduling:\nPID\tBurst\tPeriod\n");
```

```
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\n", tasks[i].id, tasks[i].burst, tasks[i].period);
        utilization += (float)tasks[i].burst / tasks[i].period;
    }

    float bound = n * (pow(2, (1.0 / n)) - 1);
    printf("\nUtilization: %.6f, Bound: %.6f\n", utilization, bound);
    if (utilization <= bound)
        printf("Tasks are Schedulable\n");
    else
        printf("Tasks are NOT Schedulable\n");
}

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

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

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

    rateMonotonic(tasks, n);
```

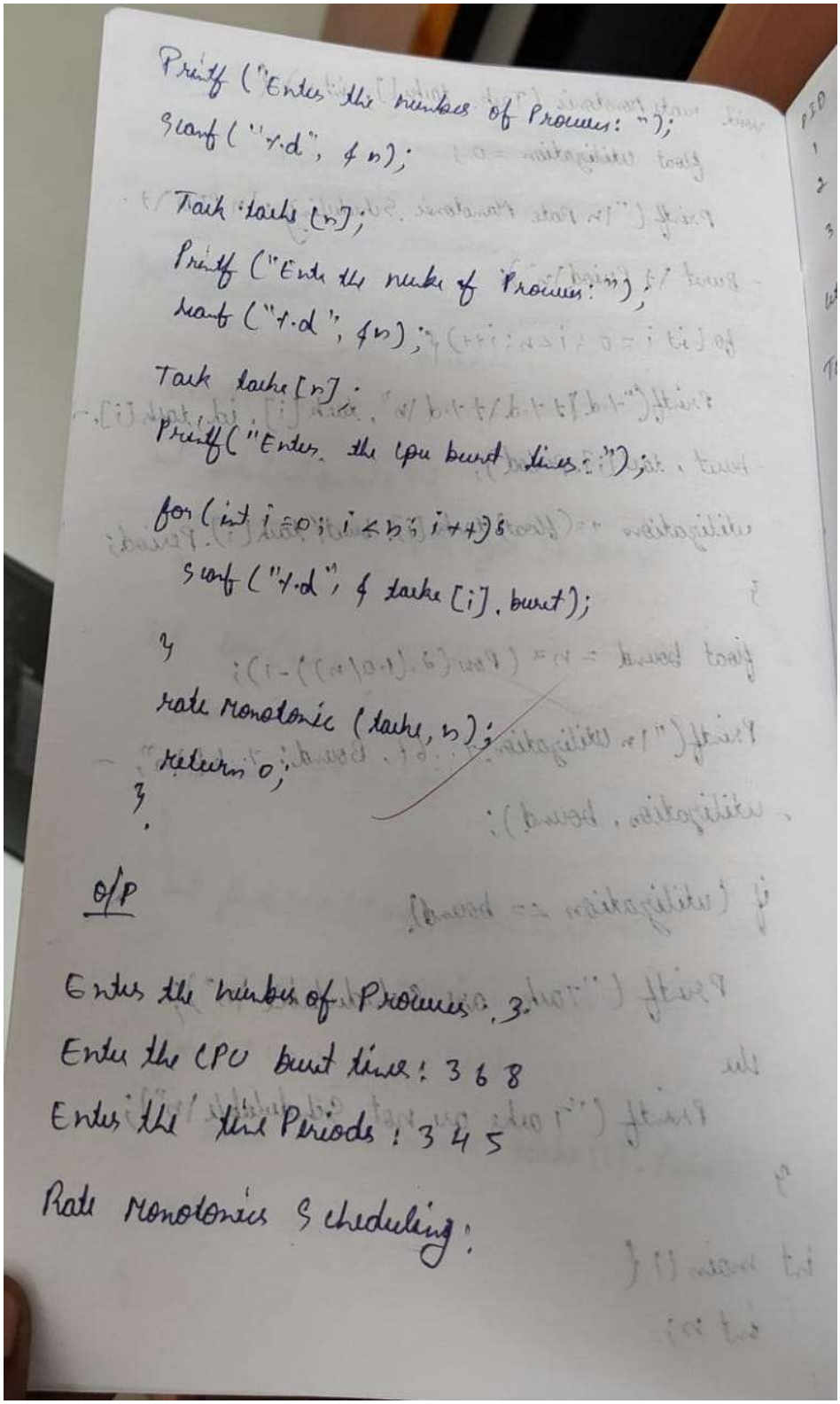
```
    return 0;
}
```

```
Enter the number of processes: 3
Enter the CPU burst times:
3
6 8
Enter the time periods:
3 4 5
LCM=60

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

4.100000 <= 0.779763 => false

System may not be schedulable!

Process returned 0 (0x0)   execution time : 18.410 s
Press any key to continue.
```

Rate - Monotonic

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

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

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

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

int findLCM(Task task[], int n) {
    int result = task[0].period;
    for (int i = 1; i<n; i++)
        result = lcm(result, tasks[i].Period);
    return result;
}
```

```
void rateMonotonic(Task tasks[], int n) {
    float utilization = 0;
    Printf("\n Rate Monotonic Scheduling:\n PID\t
- Burst\t Period\n");
    for(int i=0;i<n;i++) {
    Printf("%d\t %d\t %d\n", task[i].id, task[i].-
- burst, task[i].period);
    utilization += (float) tasks[i].burst/task[i].Period;
}
    float bound = n*(Pow(2,(1.0/n))-1);
    Printf("\n Utilization: %.6f, Bound: %.6f\n", -
- utilization, bound);
    if (utilization <= bound)
        Printf("Tasks are Schedulable\n");
    else
        Printf("Tasks are not Schedulable\n");
}
int main() {
    int n;
```



```
Printf ("Enter the number of Process: ");
Scanf ("%d", &n);
Task tasks[n];
Printf ("Enter the number of Process: ");
Scanf ("%d", &n);
Task tasks[n];
Printf("Enter the cpu burst times: ");
for (int i=0; i<n; i++)
  Scanf ("%d", &tasks[i].burst);
}
rate Monotonic (tasks, n);
return 0;
}
```

O/P

```
Enter the number of Process: 3
Enter the CPU burst time: 3 6 8
Enter the time Periods: 3 4 5
Rate Monotonics Scheduling:
```

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

Utilization : 4.10000, Bound : 0.779763.

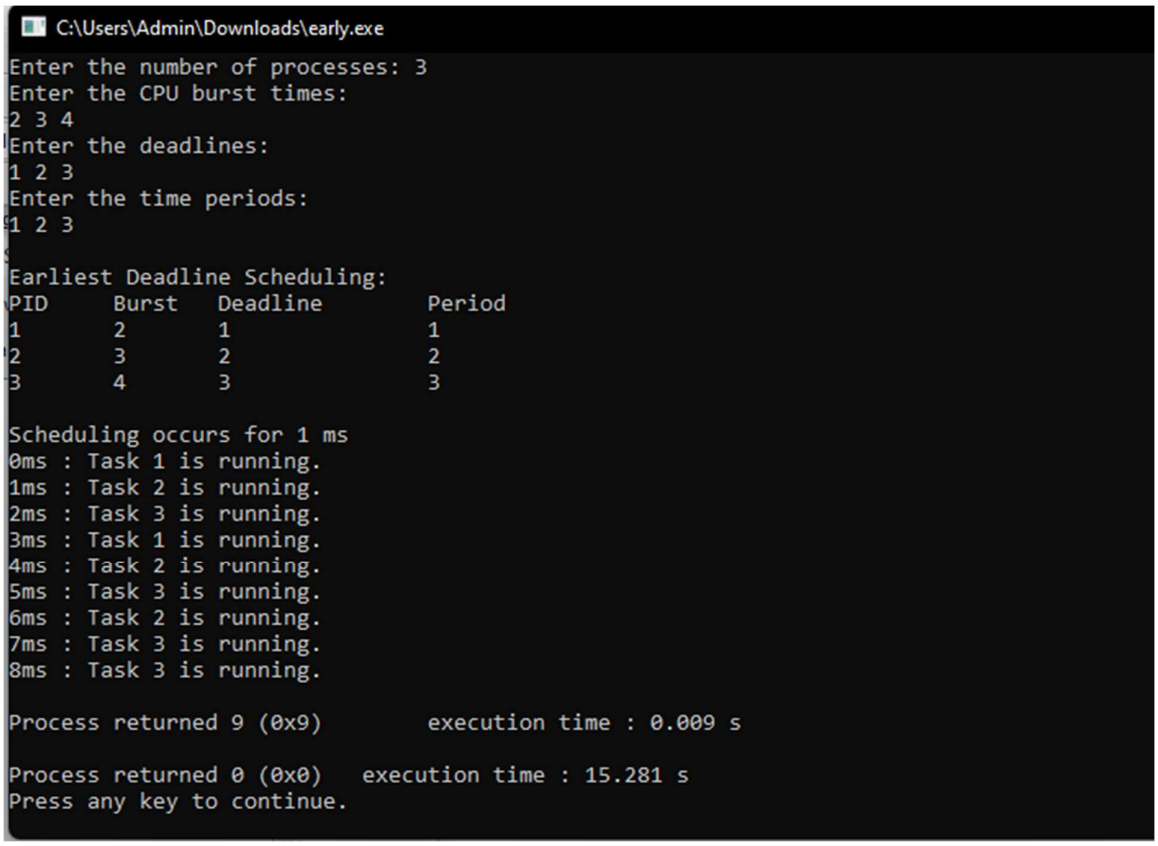
Tasks are not schedulable.

→Earliast-Deadline First

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

typedef struct {
    int pid;
    int burst_time;
    int deadline;
    int period;
    int remaining_time;
} Process;

// Function to compare processes based on their deadlines
int compare_deadlines(const void *a, const void *b) {
    return ((Process *)a)->deadline - ((Process *)b)->deadline;
}

int main() {
    int num_processes;

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

    Process processes[num_processes];

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

```
    }

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

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

    // Sort processes based on deadlines
    qsort(processes, num_processes, sizeof(Process), compare_deadlines);

    printf("\nEarliest Deadline Scheduling:\n");
    printf("PID\tBurst\tDeadline\tPeriod\n");
    for (int i = 0; i < num_processes; i++) {
        printf("%d\t%d\t%d\t\t%d\n", processes[i].pid, processes[i].burst_time, processes[i].deadline,
processes[i].period);
    }

    int current_time = 0;
    int completed_processes = 0;

    printf("\nScheduling occurs for %d ms\n", processes[0].deadline); // Assuming the first process has
the earliest deadline

    while (completed_processes < num_processes) {
        for (int i = 0; i < num_processes; i++) {
```

```
            if (processes[i].remaining_time > 0) {
                printf("%dms : Task %d is running.\n", current_time, processes[i].pid);
                processes[i].remaining_time--;
                current_time++;

                if (processes[i].remaining_time == 0) {
                    completed_processes++;
                }
            }
        }
    }

    printf("\nProcess returned %d (0x%X)\texecution time : %.3f s\n", current_time, current_time,
(float)current_time / 1000.0);

    return 0;
}
```


C:\Users\Admin\Downloads\early.exe
Enter the number of processes: 3
Enter the CPU burst times:
2 3 4
Enter the deadlines:
1 2 3
Enter the time periods:
1 2 3
Earliest Deadline Scheduling:
PID Burst Deadline Period
1 2 1 1
2 3 2 2
3 4 3 3
Scheduling occurs for 1 ms
0ms : Task 1 is running.
1ms : Task 2 is running.
2ms : Task 3 is running.
3ms : Task 1 is running.
4ms : Task 2 is running.
5ms : Task 3 is running.
6ms : Task 2 is running.
7ms : Task 3 is running.
8ms : Task 3 is running.
Process returned 9 (0x9) execution time : 0.009 s
Process returned 0 (0x0) execution time : 15.281 s
Press any key to continue.

Earliest - Deadline

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

typedef struct {
    int Pid;
    int burst_time;
    int deadline;
    int Period;
    int remaining_time;
} Process;

int compare_deadline(const void *a, const void *b) {
    return ((Process *)a) -> deadline - ((Process *)b)
-> deadline;
}

int main() {
    int num_Process;
    Printf("Enter the number of Process : ");
    Scanf("%d", &num_Process);
    Process Process[num_Process];
    Printf("Enter the CPU burst times : \n");
```

```
for (int i=0; i < num_Proones; i++) {
    Scanf("%d", & Proones; i++) {
    Proones[i]. Pid = i+1;
    Proone [i]. remaining_time = Proones[i]. burst_time;

Printf ("Enter the time Periods: \n");
for (int i=0; i < num_Proones; i++) {
    Scanf("%d", & Proones [i]. Period);
}

qsort (Proones, num_Proones, sizeoff (Proones), comp-
-are_deadlines);

Printf ("\n Earliest Deadline scheduling: \n");
Printf("PID \t Burst \t Deadline \t Period \n");
for(int i=0; i < num_Proones; i++) {
Printf("%d \t %d \t %d \t %d \n", Proones[i]. Pid,
- Proones[i] burst_time, Proones [i]. deadline, Proones (i). Pe-
-od.)
}
int current_time = 0;
int completed_Proones = 0;
Printf("\n Scheduling occurs for %d ms \n", Proones [0] dead
```

```
- time);
while (completed - Process < num_Process) {
    for(int i=0; i<num_process; i++) {
    if (Process[i].remaining_time > 0) {
    Printf ("%d ms: Task %d is running \n", current
- .current time process[] %d );
        Process[i].remaining_time --";
        current_time ++;

    if (Process[i].remaining time == 0) {
        completed_Process ++;
    }
}
}
};
Printf ("\n Process returned %d (0x%x) t execution
- time %.3f \n", current_time, current_time, (float) current_
- time / 100.0.0);

return 0;
}
```

O/P

Enter the number of Process: 3

Enter the CPU burst time: 2 3 4

Enter the deadlines : 1 2 3
Enter the time Periods : 1 2 3.

Earliest Deadline Scheduling

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

Scheduling occurs for 1ms.

0 ms : Task 1 is running
1 ms : Task 2 is running
2 ms : Task 3 is running
3ms : Task1 is runing
4ms : Task 2 is running
5 ms : Task 3 —11—
6 ms : Task 2 —11—
7ms : Task 3 —11—
8 ms : Task 3 —11—

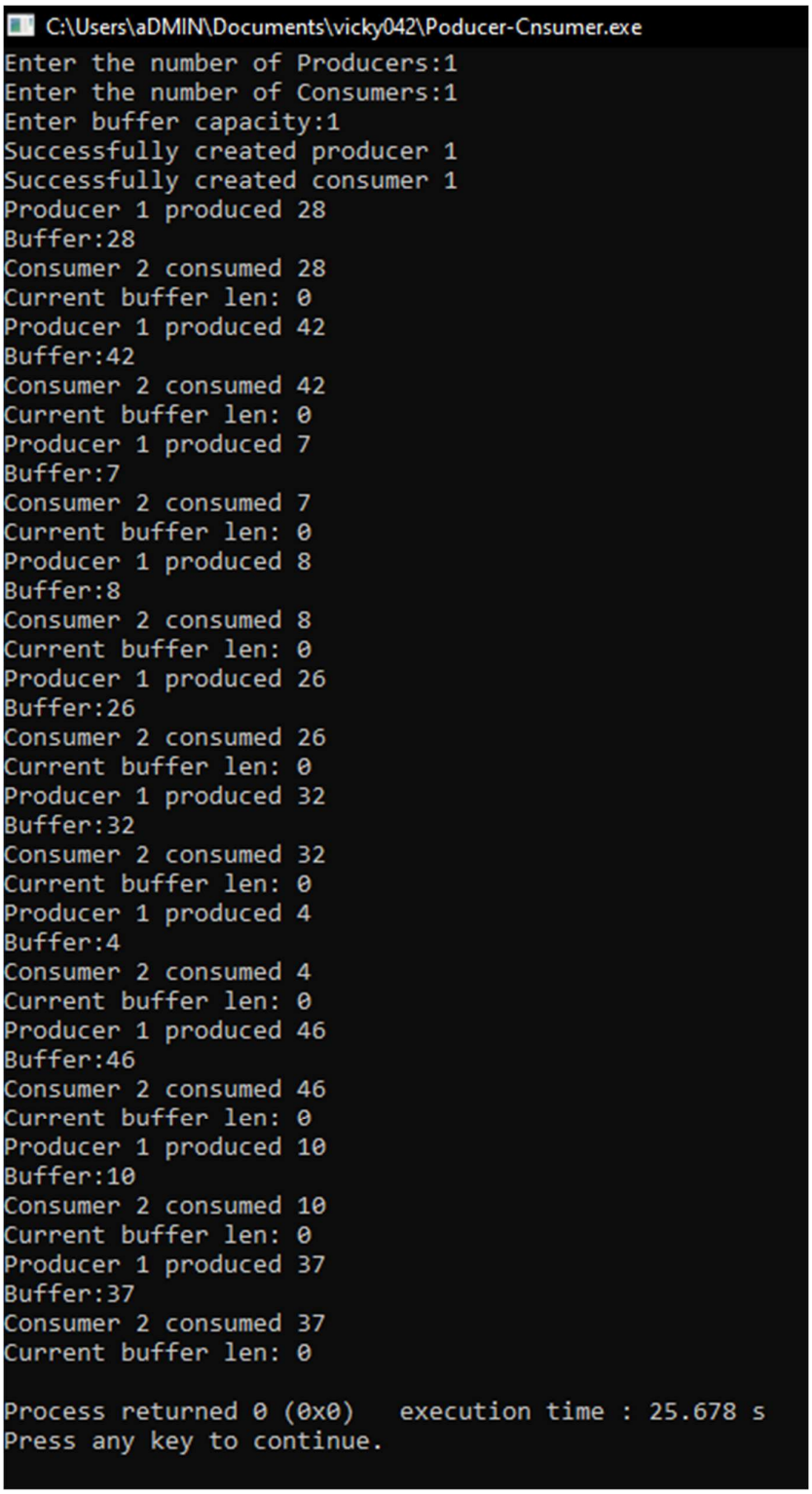
Program 5:

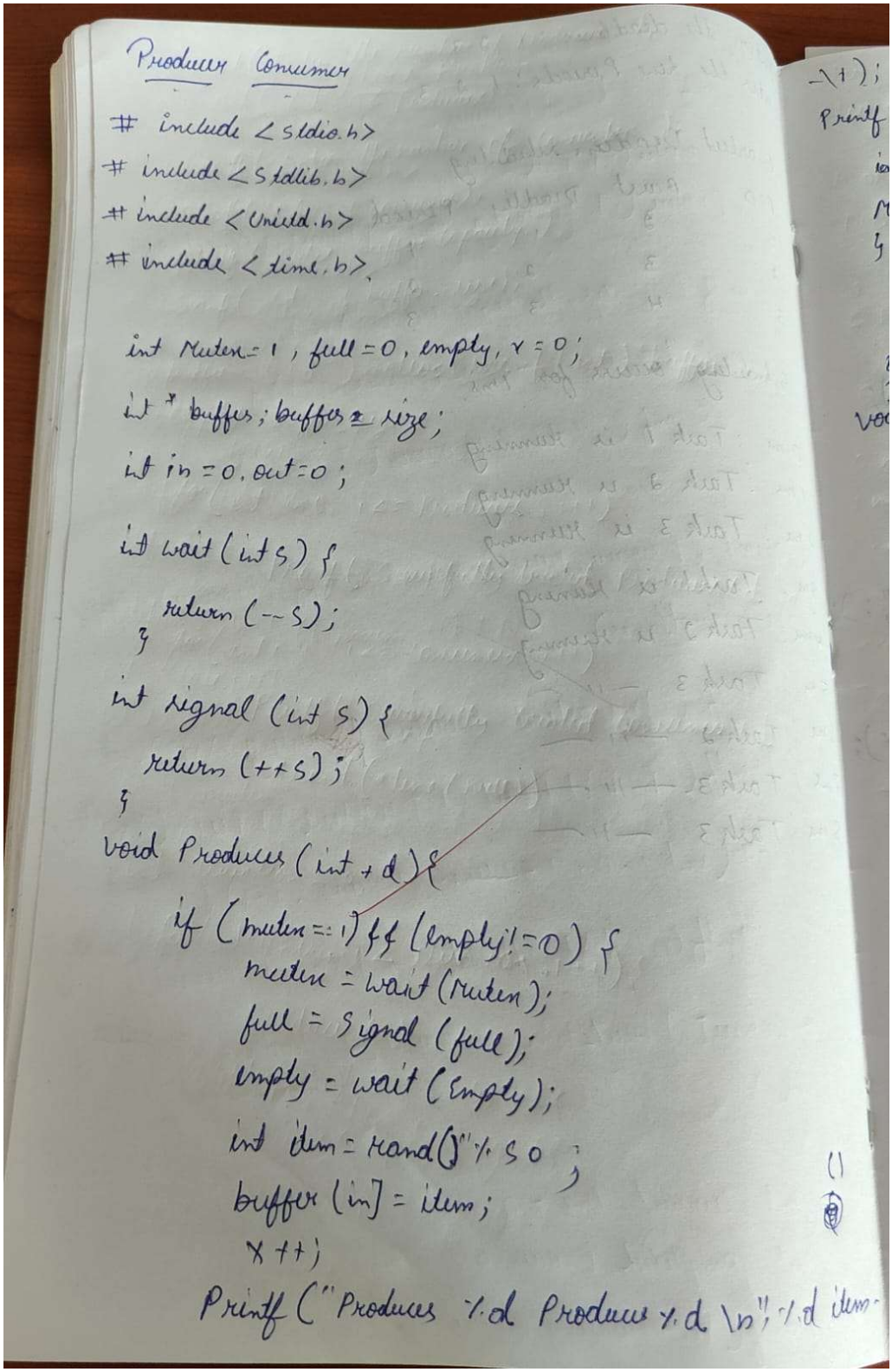
Write a C program to simulate producer-consumer problem using semaphores

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

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

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

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

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

```
        mutex = signal(mutex);
    } else {
        printf("Buffer is full\n");
    }
}

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

int main() {
    int producers, consumers;
    printf("Enter the number of Producers:");
    scanf("%d", &producers);
    printf("Enter the number of Consumers:");
    scanf("%d", &consumers);
    printf("Enter buffer capacity:");
    scanf("%d", &buffer_size);
```

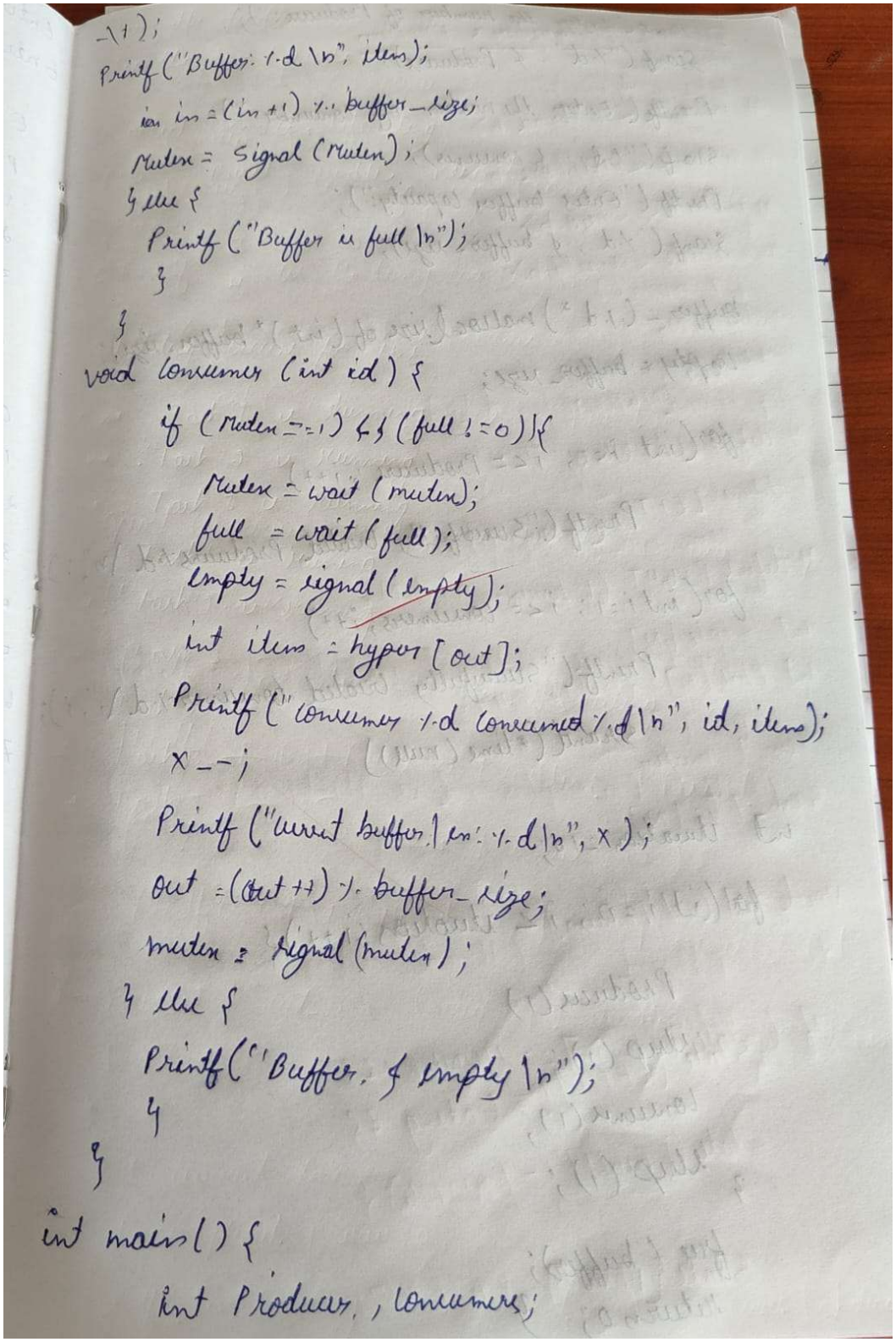
```
    buffer = (int *)malloc(sizeof(int) * buffer_size);
    empty = buffer_size;

    for (int i = 1; i <= producers; i++)
        printf("Successfully created producer %d\n", i);
    for (int i = 1; i <= consumers; i++)
        printf("Successfully created consumer %d\n", i);

    srand(time(NULL));

    int iterations = 10;
    for (int i = 0; i < iterations; i++) {
        producer(1);
        sleep(1);
        consumer(2);
        sleep(1);
    }

    free(buffer);
    return 0;
}
```


C:\Users\aDMIN\Documents\vicky042\Poducer-Cnsumer.exe
Enter the number of Producers:1
Enter the number of Consumers:1
Enter buffer capacity:1
Successfully created producer 1
Successfully created consumer 1
Producer 1 produced 28
Buffer:28
Consumer 2 consumed 28
Current buffer len: 0
Producer 1 produced 42
Buffer:42
Consumer 2 consumed 42
Current buffer len: 0
Producer 1 produced 7
Buffer:7
Consumer 2 consumed 7
Current buffer len: 0
Producer 1 produced 8
Buffer:8
Consumer 2 consumed 8
Current buffer len: 0
Producer 1 produced 26
Buffer:26
Consumer 2 consumed 26
Current buffer len: 0
Producer 1 produced 32
Buffer:32
Consumer 2 consumed 32
Current buffer len: 0
Producer 1 produced 4
Buffer:4
Consumer 2 consumed 4
Current buffer len: 0
Producer 1 produced 46
Buffer:46
Consumer 2 consumed 46
Current buffer len: 0
Producer 1 produced 10
Buffer:10
Consumer 2 consumed 10
Current buffer len: 0
Producer 1 produced 37
Buffer:37
Consumer 2 consumed 37
Current buffer len: 0
Process returned 0 (0x0) execution time : 25.678 s
Press any key to continue.



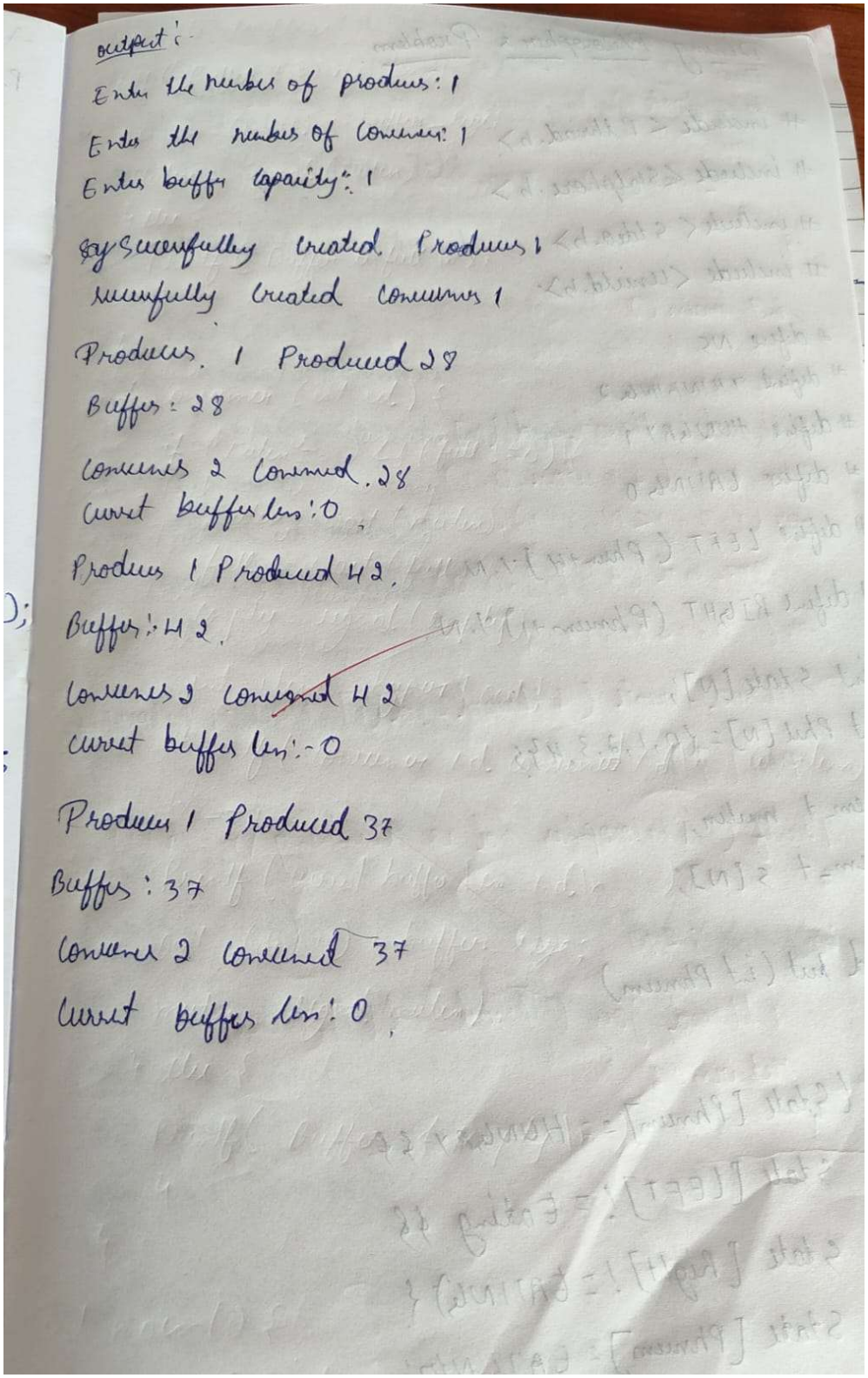
Producer Consumer

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

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

int wait (int s) {
    return (--s);
}
int signal (int s) {
    return (++s);
}
void Produce (int * d) {
    if (mutex == 1) && (empty != 0) {
        mutex = wait (mutex);
        full = signal (full);
        empty = wait (empty);
        int item = rand() % 50;
        buffer [in] = item;
        x++;
    Printf ("Produces %d Produce %d \n", %d item
```



```
-\t);
Printf("Buffer: %d \n", item);
    in= in = (in+1) % buffer_size;
    Mutex = Signal(Mutex);
} else {
    Printf("Buffer is full \n");
    }
}
void consumer (int id) {
    if (Mutex == -1) && (full != 0){
        Mutex = wait(mutex);
        full = wait(full);
        empty = signal(empty);
        int item = hyper[out];
        Printf("consumer %d consumed %d \n", id, item);
        x--;
        Printf("current buffer in: %d \n", x);
        out = (out++) % buffer_size;
        mutex = signal(mutex);
    } else {
        Printf("Buffer, & empty \n");
    }
}
int main() {
    int Producer, consumers;
```

```
Print("Enter the number of Producers:");
Scanf("%d", & Producers);
Printf("Enter the number of consumers:");
Scanf("%d", & consumers);
Printf("Enter buffer capacity:");
Scanf("%d", & buffer_size);

buffer = (int *) malloc(size of(int) * buffer_size);
empty = buffer_size;

for(int i=1; i<= Producers ; i++)
        Printf("Succesfully created Producers %d \n",;);
for(int i=1; i<= consumers; i++)
        Printf("Succesfully created consumer %d \n", i);
        Srand(*time(null))

int duration = 10;
for(int i=0; i< duration ;i++){
        Producer(1)
        sleep(1);
        consumer(1);
    } sleep(1);

    free(buffer);
    return 0; }
```



output:-

Enter the number of producers: 1

Enter the number of consumers: 1

Enter buffer capacity: 1

Successfully created Producers:

Successfully created Consumers 1

Producer 1 Produced 28

Buffer: 28

Consumer 2 Consumed 28

Current buffer len: 0

Producer 1 Produced 42

Buffer: 42

Consumer 2 Consumed 42

Current buffer len: 0

Producer 1 Produced 37

Buffer: 37

Consumer 2 Consumed 37

Current buffer len: 0

Program 6:

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

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

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };

sem_t mutex;
sem_t S[N];

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

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

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

        sem_post(&S[phnum]);
    }
}

void take_fork(int phnum)
{
    sem_wait(&mutex);

    state[phnum] = HUNGRY;
    printf("Philosopher %d is Hungry\n", phnum + 1);

    test(phnum);

    sem_post(&mutex);

    sem_wait(&S[phnum]);
    sleep(1);
}

void put_fork(int phnum)
{
    sem_wait(&mutex);

    state[phnum] = THINKING;
    printf("Philosopher %d putting chopstick %d and %d down\n", phnum + 1, LEFT + 1, phnum + 1);
    printf("Philosopher %d is thinking\n", phnum + 1);

    test(LEFT);
```

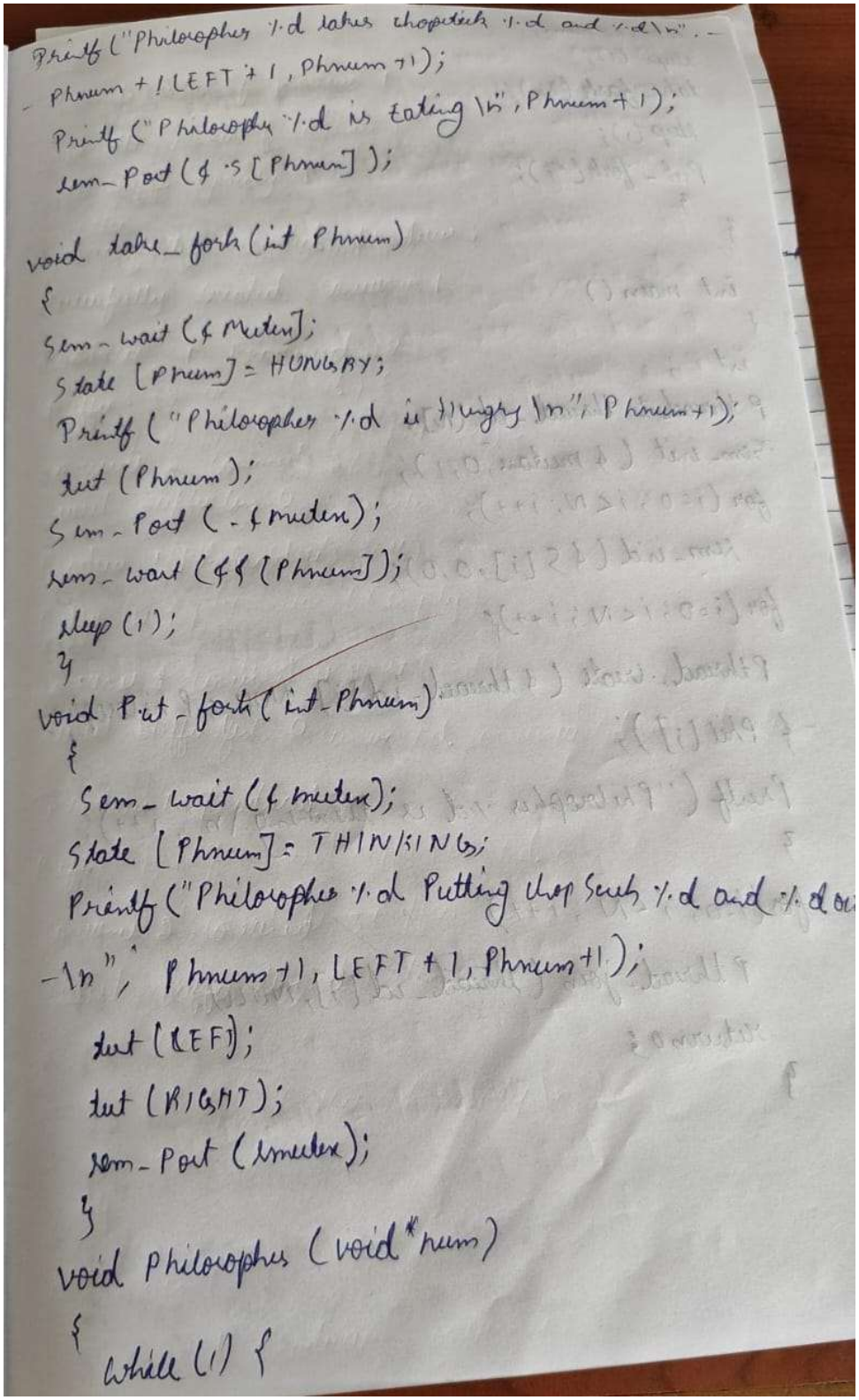
```
    test(RIGHT);

    sem_post(&mutex);
}

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

        sleep(1);
        take_fork(*i);
        sleep(1);
        put_fork(*i);
    }
}

int main()
{
    int i;
    pthread_t thread_id[N];

    sem_init(&mutex, 0, 1);

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

    for (i = 0; i < N; i++) {
        pthread_create(&thread_id[i], NULL, philosopher, &phil[i]);
        printf("Philosopher %d is thinking\n", i + 1);
```

```
    }

    for (i = 0; i < N; i++)
        pthread_join(thread_id[i], NULL);

    return 0;
}
```

```
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
```

Dining Philosopher's Problem

```
#include <Pthread.h>
#include <Semphore.h>
#include <stdio.h>
#include <unistd.h>

#define N5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum+4)%N
#define RIGHT (phnum+1)%N.

int state[N];
int phil[N]={0,1,2,3,4};

sem_t mutex;
sem_t s[N];

void test(int phnum)
{
if (state[phnum]==HUNGRY &&
    state[LEFT]!=Eating &&
    state[Right]!=EATING){
    state[phnum]=EATING;
```



```
Printf ("Philosopher %d takes chopstick %d and %d\n".
Phnum+1 LEFT + 1, Phnum+1);
Printf ("Philosopher %d is Eating \n", Phnum+1);
sem_Post (& S [Phnum]);

void take_fork (int Phnum)
{
Sem_wait (& Mutex);
State [Phnum] = HUNGRY;
Printf ("Philosopher %d is Hungry \n", Phnum+1);
test (Phnum);
Sem_Post (& mutex);
sem_wait (& S [Phnum]);
sleep (1);
}
void Put_fork (int Phnum)
{
Sem_wait (& mutex);
State [Phnum] = THINKING;
Printf ("Philosopher %d Putting chop Sticks %d and %d on
-\n", Phnum+1, LEFT + 1, Phnum+1);
test (LEFT);
test (RIGHT);
sem_Post (& mutex);
}
void Philosophes (void* num)
{
while (1) {
```

```
    int *i = num;
    sleep (1);
    take_fork (*i);
    sleep (1);
    Put_fork (*i);
    }
}
int main ()
{
int i;
Pthread_t thread_id [N];
sem_init (& mutex, 0, 1);
for (i=0; i<N; i++);
    sem_init (& S [i], 0, 0);
for (i=0; i<N; i++){
  Pthread_create (& thread_id [i], NULL, Philosopher,
& Phil [i]);
  Printf ("Philosopher %d is thinking \n", i++);
}
for (i=0; i<N; i++)
    Pthread_join (thread_id [i], NULL);
    return 0;
}
```

output:-

Dining philosophy problem.

Enter the total no of philosophers: 5

How many are hungry: 3.

Enter philosopher 1 position: 2

Enter philosopher 2 Position: 4

Enter philosopher 3 Position: 5

1. One can eat at a time

2. Two can eat at a time.

3. Exit.

Enter your choice: 1

How one philosophers to eat at time.

P_2 is granted to eat

P_4 is waited

P_5 is waited.

P_4 is granted to eat.

P_2 is granted to eat.

P_2 is waiting

P_5 is granted to eat

P_2 is waiting.

P_4 is waiting

1. One can eat at a time

2. Two can eat at a time.

3. Exit

Program 7:

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

```
#include <stdio.h>

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

   int alloc[n][m], max[n][m], avail[m];
   int need[n][m], f[n], ans[n], ind = 0;

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

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

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

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

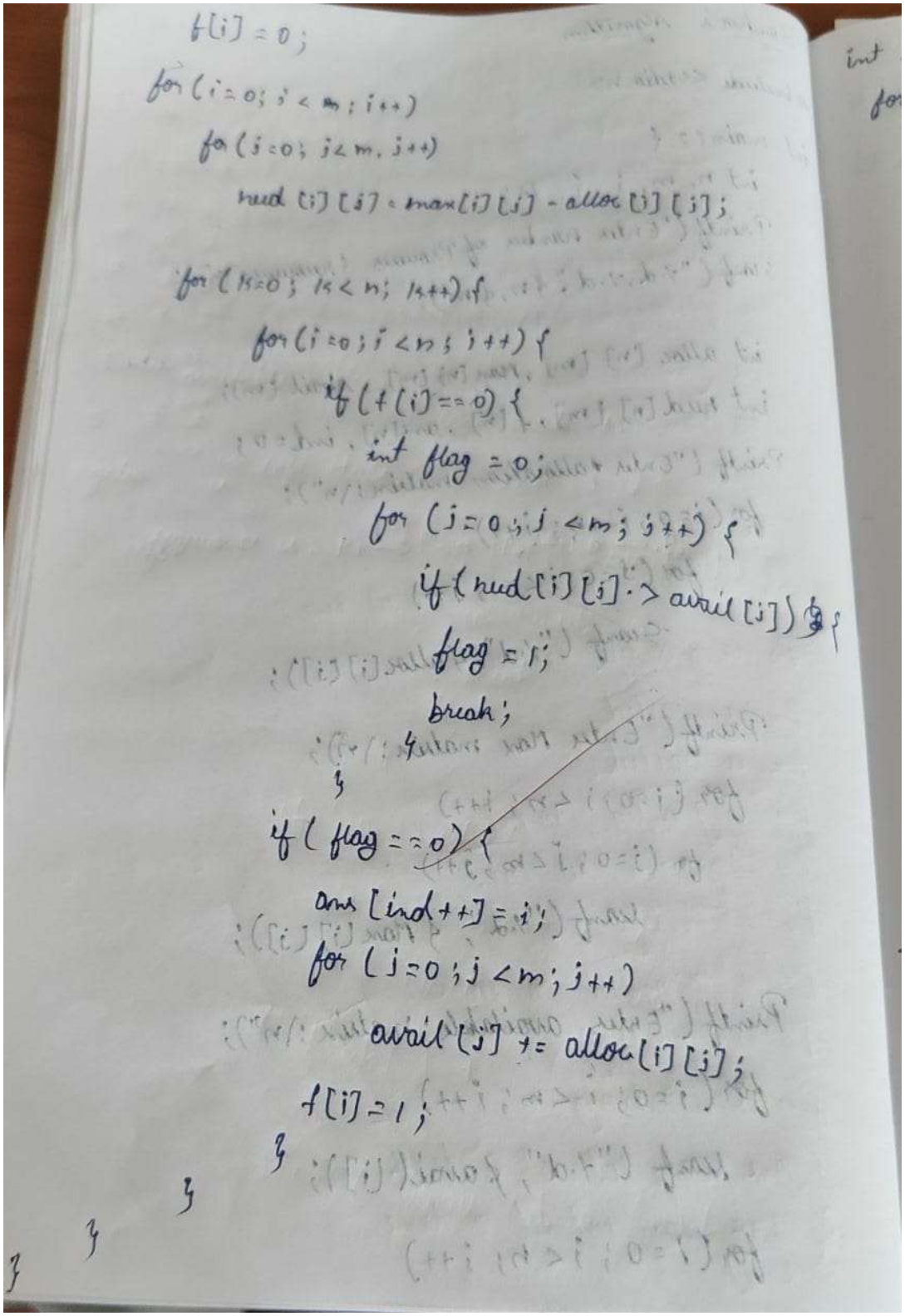
```
for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
        need[i][j] = max[i][j] - alloc[i][j];

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

int safe = 1;
for (i = 0; i < n; i++)
    if (f[i] == 0)
        safe = 0;

if (safe) {
```

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

    return 0;
}
```

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

Banker's Algorithm

```
#include <stdio.h>
int main() {
    int n, m, i, j, k;
    Printf("Enter number of Process & resources:\n");
    Scanf("%d %d", &n, &m);

    int alloc[n][m], Max[n][m], avail[m];
    int need[n][m], f[n], ans[n], ind=0;
    Printf("Enter allocation matrix:\n");
    for(i=0; i<n; i++)
        for(j=0; j<m; j++)
            Scanf("%d", &alloc[i][j]);
    Printf("Enter Max matrix:\n");
    for(i=0; i<n; i++)
        for(j=0; j<m; j++)
            scanf("%d", &Max[i][j]);
    Printf("Enter available matrix:\n");
    for(i=0; i<m; i++)
        scanf("%d", &avail[i]);
    for(i=0; i<n; i++)
```



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

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

O/P.

Enter number of process and resources:

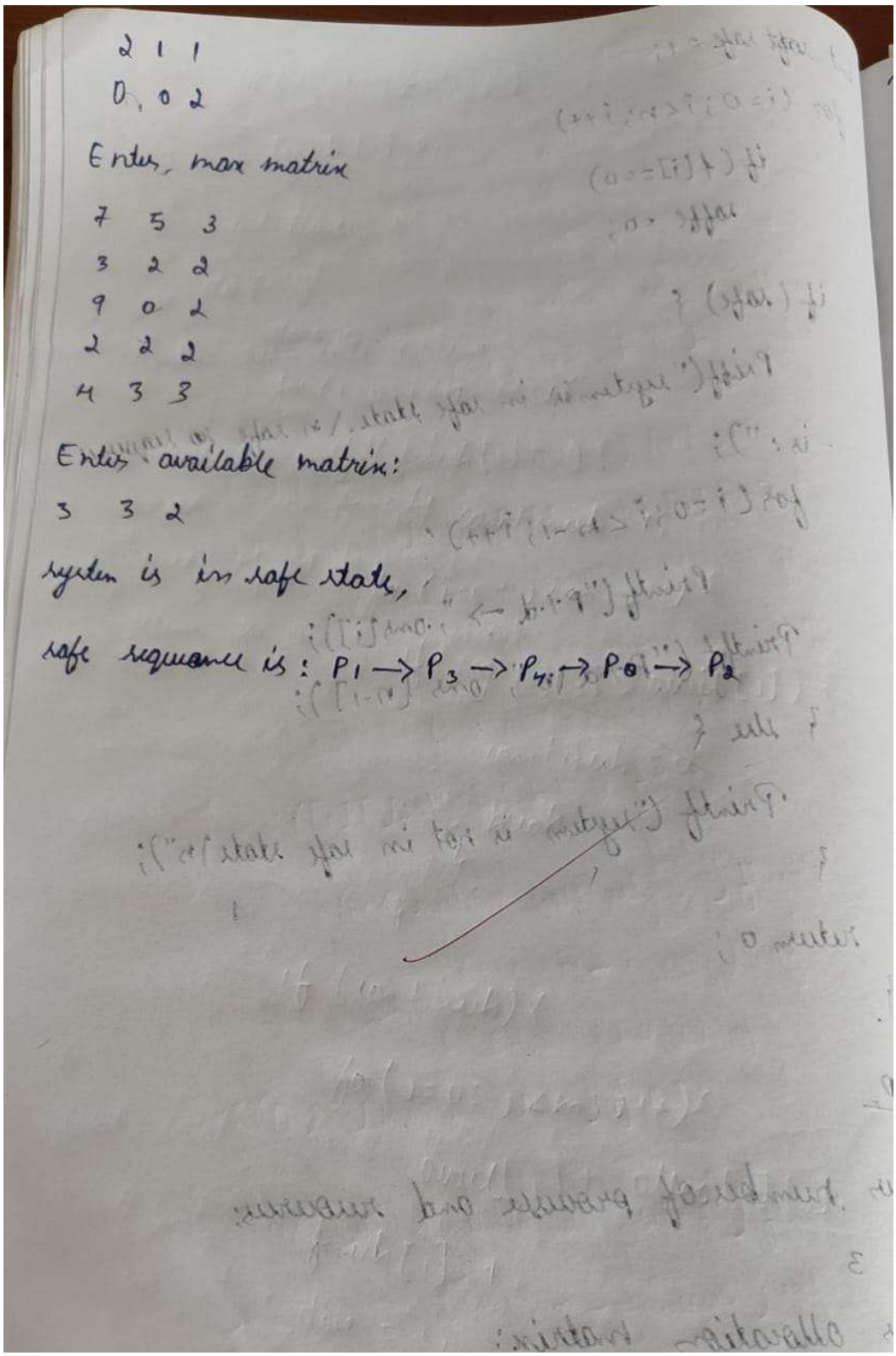
5 3

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

Program 8:

Write a C program to simulate deadlock detection

```
#include <stdio.h>

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

   int alloc[n][m], req[n][m], avail[m], finish[n];

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

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

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

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

   int done;
   do {
```

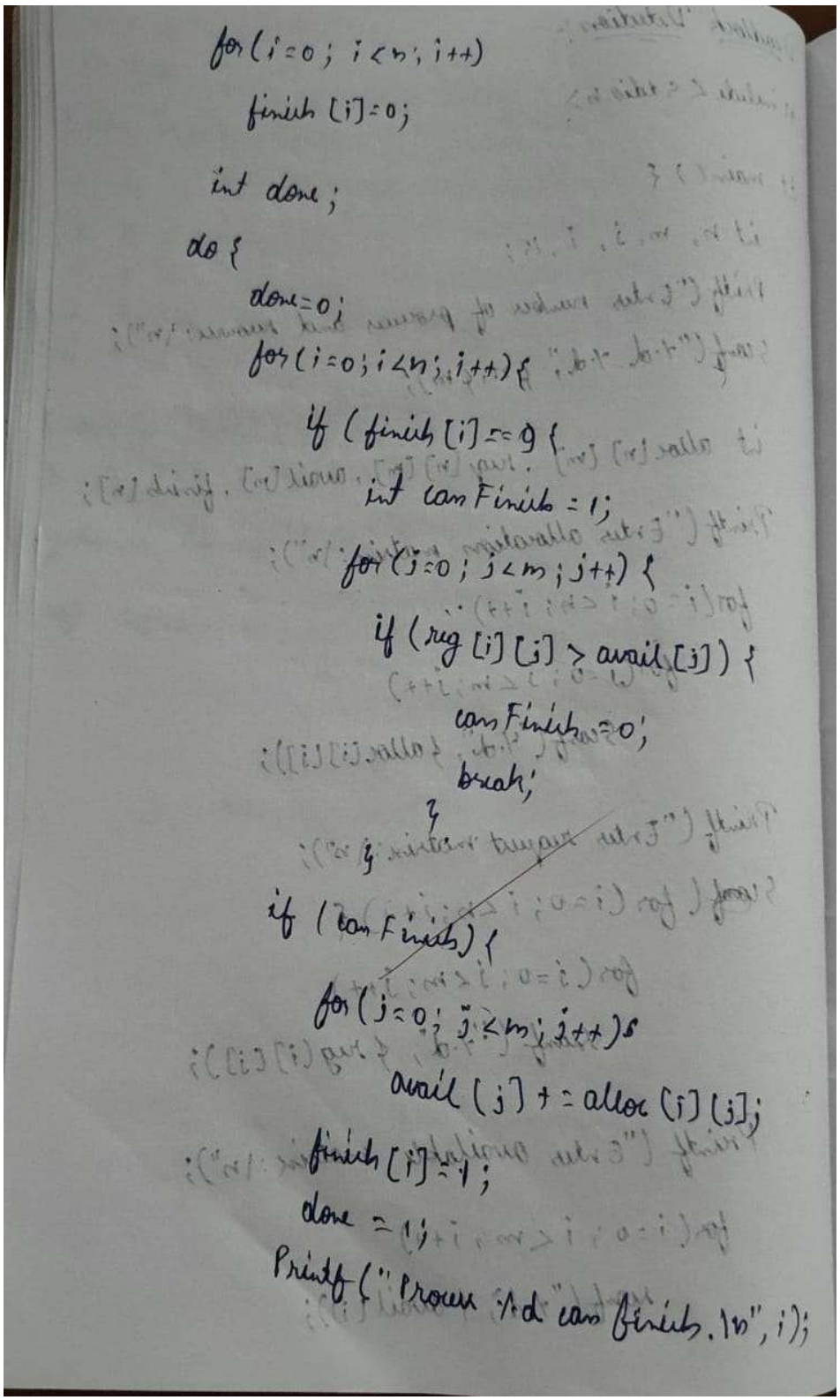
```
    done = 0;
    for (i = 0; i < n; i++) {
        if (finish[i] == 0) {
            int canFinish = 1;
            for (j = 0; j < m; j++) {
                if (req[i][j] > avail[j]) {
                    canFinish = 0;
                    break;
                }
            }
            if (canFinish)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

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

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

```
    return 0;
}
```

```
Enter number of processes and resources:
5 3
Enter allocation matrix:
0 1 0
2 0 0
3 0 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>

int main() {
    int n, m, j, i, k;
    Printf("Enter number of process and resource: \n");
    Scanf("%d %d", &n, &m);

    int alloc[n][m], req[n][m], avail[m], finish[n];
    Printf("Enter allocation matrix: \n");
    for(i=0; i<n; i++)
        for(j=0; j<m; j++)
            Scanf("%d", &alloc[i][j]);
    Printf("Enter request matrix: \n");
    Scanf( for(i=0; i<n; i++) {
            for(j=0; j<m; j++)
            scanf("%d", &req[i][j]);
    Printf("Enter available matrix: \n");
    for(i=0; i<m; i++)
        scanf("%d", &avail[i]);
```



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

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

```
    }
  }
}
while (done);
int deadlock = 0;
for (i=0; j<n; i++)
  if (finish[i] == 0)
    deadlock = 1;
if (deadlock)
  Printf("System is in a deadlock state.\n");
else
  Printf("System is not in a deadlock state.\n");
  return 0;
}
```

O/P

Enter number of Process and resources;
5 3.
Enter allocation matrix
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2

Enter request matrix:

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

Enter available matrix:

3 3 2

Process 1 can finish
Process 3 can finish
Process 4 can finish
System is in a deadlock.

Program 9:

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

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

→Worst Fit

```
#include <stdio.h>

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

struct File {
   int file_no;
   int file_size;
};

void worstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
   printf("\nMemory Management Scheme - Worst Fit\n");
   printf("File_no\tFile_size\tBlock_no\tBlock_size\tFragment\n");

   for (int i = 0; i < n_files; i++) {
      int worst_fit_block = -1;
      int max_fragment = -1;

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

      if (worst_fit_block != -1) {
         blocks[worst_fit_block].is_free = 0;
         printf("%d\t%d\t\t%d\t\t%d\t\t%d\n",
               files[i].file_no,
               files[i].file_size,
               blocks[worst_fit_block].block_no,
               blocks[worst_fit_block].block_size,
               max_fragment);
```

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

int main() {
    int n_blocks, n_files;

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

    struct Block blocks[n_blocks];

    for (int i = 0; i < n_blocks; i++) {
        blocks[i].block_no = i + 1;
        printf("Enter the size of block %d: ", i + 1);
        scanf("%d", &blocks[i].block_size);
        blocks[i].is_free = 1;
    }

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

    struct File files[n_files];

    for (int i = 0; i < n_files; i++) {
        files[i].file_no = i + 1;
        printf("Enter the size of file %d: ", i + 1);
        scanf("%d", &files[i].file_size);
    }

    worstFit(blocks, n_blocks, files, n_files);

    return 0;
}
```

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

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

→Best Fit

```
#include <stdio.h>

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

struct File {
    int file_no;
    int file_size;
};

void bestFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf("\nMemory Management Scheme - Best Fit\n");
    printf("File_no\tFile_size\tBlock_no\tBlock_size\tFragment\n");

    for (int i = 0; i < n_files; i++) {
        int best_fit_block = -1;
        int min_fragment = 10000; // Large initial value

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

```
            }
        }

        if (best_fit_block != -1) {
            blocks[best_fit_block].is_free = 0;
            printf("%d\t%d\t\t%d\t\t%d\t\t%d\n",
                   files[i].file_no,
                   files[i].file_size,
                   blocks[best_fit_block].block_no,
                   blocks[best_fit_block].block_size,
                   min_fragment);
        } else {
            printf("%d\t%d\t\tNot Allocated\n", files[i].file_no, files[i].file_size);
        }
    }
}

int main() {
    int n_blocks, n_files;

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

    struct Block blocks[n_blocks];

    for (int i = 0; i < n_blocks; i++) {
        blocks[i].block_no = i + 1;
        printf("Enter the size of block %d: ", i + 1);
        scanf("%d", &blocks[i].block_size);
        blocks[i].is_free = 1;
    }

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

    struct File files[n_files];

    for (int i = 0; i < n_files; i++) {
        files[i].file_no = i + 1;
        printf("Enter the size of file %d: ", i + 1);
        scanf("%d", &files[i].file_size);
    }

    bestFit(blocks, n_blocks, files, n_files);

    return 0;
}
```

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

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

→First Fit

```
#include <stdio.h>

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

struct File {
    int file_no;
    int file_size;
};

void firstFit(struct Block blocks[], int n_blocks, struct File files[], int n_files) {
    printf("\nMemory Management Scheme - First Fit\n");
    printf("File_no\tFile_size\tBlock_no\tBlock_size\tFragment\n");

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

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

                printf("%d\t%d\t\t%d\t\t%d\t\t%d\n",
                       files[i].file_no,
```

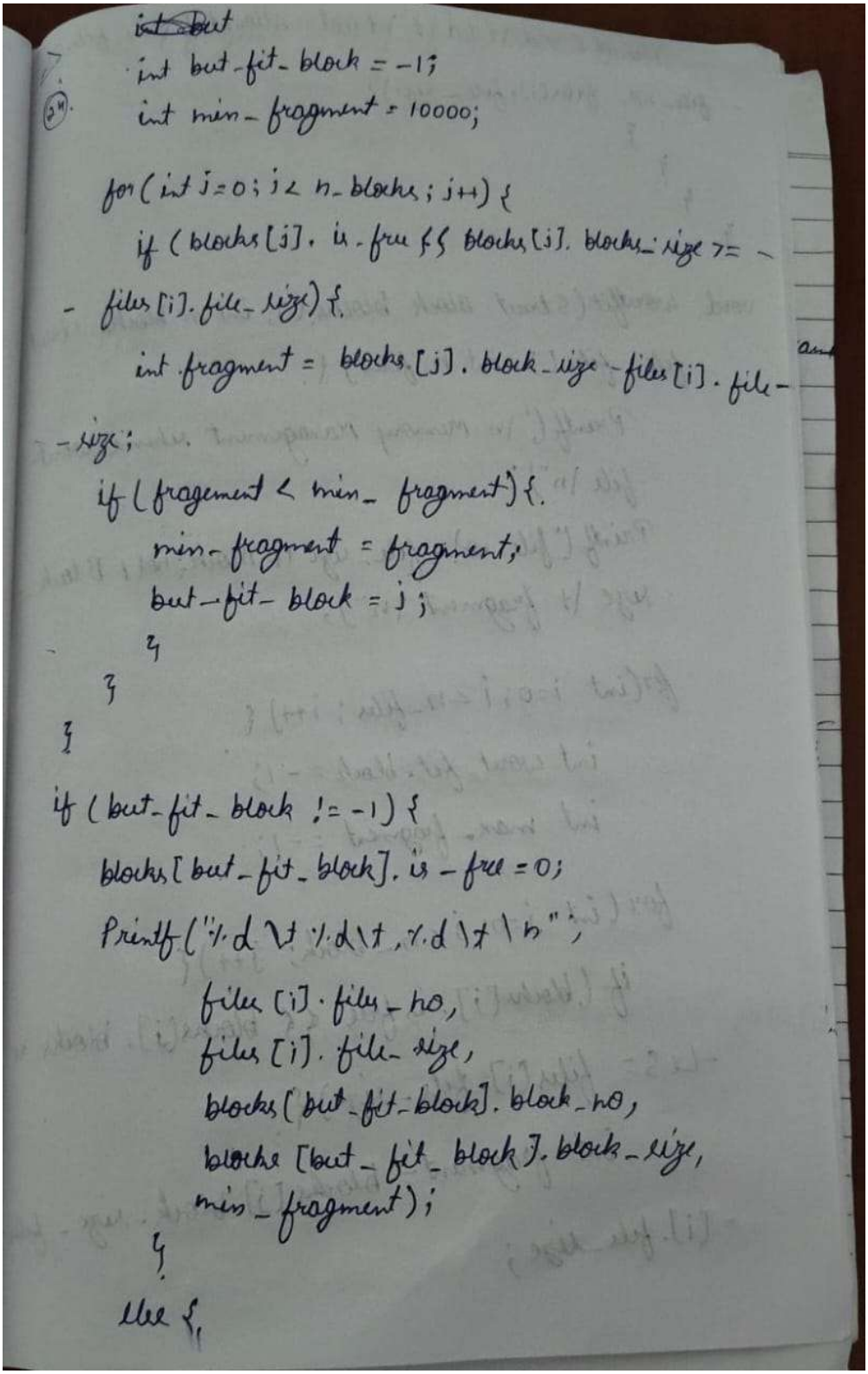
```
                files[i].file_size,
                blocks[j].block_no,
                blocks[j].block_size,
                fragment);

            allocated = 1;
            break;
        }
    }

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

int main() {
  int n_blocks, n_files;

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

  struct Block blocks[n_blocks];

  for (int i = 0; i < n_blocks; i++) {
    blocks[i].block_no = i + 1;
    printf("Enter the size of block %d: ", i + 1);
    scanf("%d", &blocks[i].block_size);
    blocks[i].is_free = 1;
  }

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

  struct File files[n_files];

  for (int i = 0; i < n_files; i++) {
    files[i].file_no = i + 1;
    printf("Enter the size of file %d: ", i + 1);
    scanf("%d", &files[i].file_size);
  }

  firstFit(blocks, n_blocks, files, n_files);

  return 0;
}
```

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

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

Bestfit, worst fit, first fit.

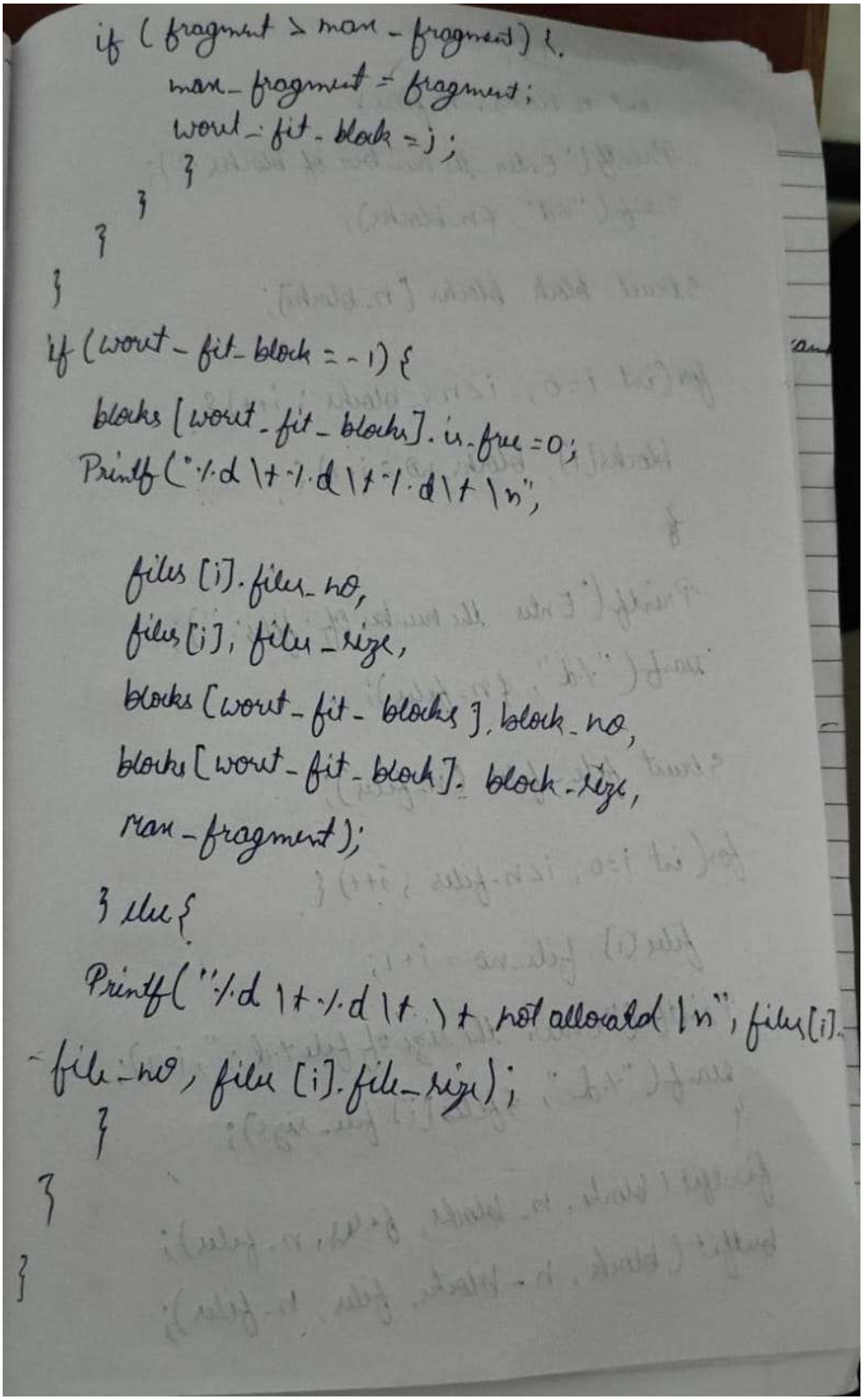
```
#include<stdio.h>
struct Block{
    int block_no;
    int block_size;
    int is_free;
};
struct file{
  int file_no;
  int file_size;
};
void firstfit(struct Block blocks[], int n_blocks, struct file
-files[], int n_files){
 Printf("\n Memory Management Scheme - first fit\n");
Printf("File_no\t File_size\t Block_no\t Block\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].files_size){
    int fragment=blocks[j].block_size-files[i].file_size;
    blocks[j].is_free=0;
```

```
Printf("%d \t %d \t %d \t %d \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 \t Not Allocated \n", files[i].
- file_no, file[i].files_size);
    }
}

void bestfit (struct 'Block blocks[], int n_blocks,
- struct File [], int n_files) {
    Printf("\n Memory Management scheme - Best fit \n
-);
    Printf(" File_no \t File_size \t Block_no \t Block_
- size \t Fragment \n");

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



```
int but
int but_fit_block = -1;
int min_fragment = 10000;

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

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

```
        Printf("%d\t%d\t\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("\n Memory Management scheme - Worst-
- fit\n");
        Printf("file_no\t- file_size\t Block_no\t Block_
- size\t fragment\n");

        for(int i=0; i<n_files; i++) {
            int worst_fit_block = -1;
            int max_fragment = -1;

        for (int j=0; j<n_block; j++) {
            if (blocks[j].is free && blocks[j].block_u-
-LES= files[i].file_size) {
                int fragment = blocks[j].block_size - file
- [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%d\t\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\t not allocated\n", files[i].
file_no, file[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], blocks_no = i+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(block, n_blocks, files, n_files);
bestfit(block, n_blocks, files, n_files);
```

worstfit(blocks, m, blocks, files, m, files);
return 0;
}

O/P :-

Enter the number of block : 5
Enter the size of block 1 : 100
" " " 2 : 200
" " " 3 : 300.
" " " 4 : 300
" " " 5 : 600.
Enter the number of files : 4.
Enter the size of file 1 : 212
" " " 2 : 417
" " " 3 : 112
" " " 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		

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.

Memory Management scheme - Worker.

file_no	File_size	Block_no	Block_size	fragment.
1	212	5	600	388.
2	417	2	500	83.
3	112	4	300	181.
4	426	Not allocated.		

Program 10:
Write a C program to simulate page replacement algorithms

a) **FIFO**
b) **LRU**
c) **Optimal**

→FIFO

```
#include <stdio.h>

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

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

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

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

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

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

→LRU

```
#include <stdio.h>

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

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

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

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

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

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

```
            used[lru] = i;
            page_faults++;
        }
    }

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

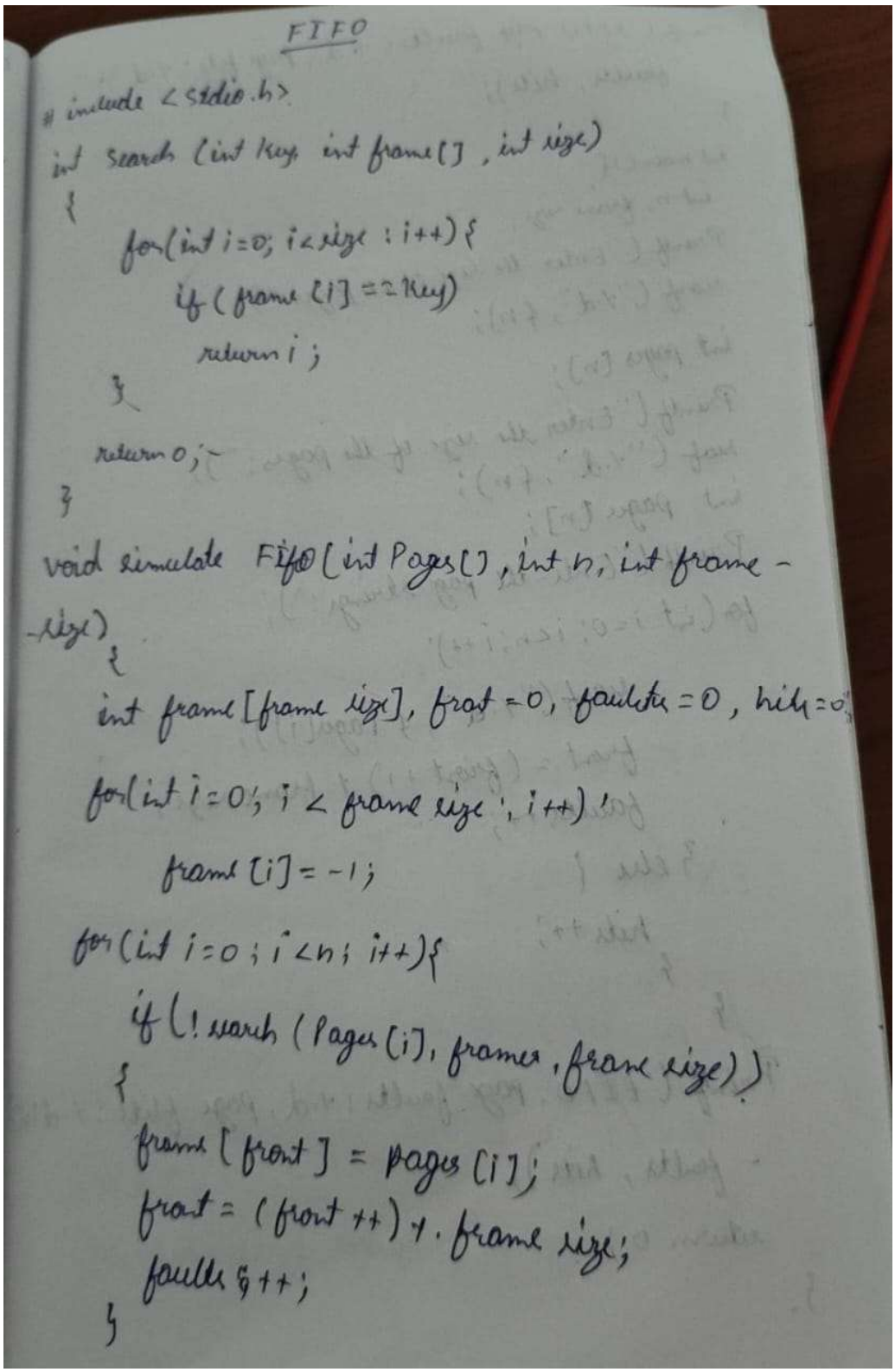
```
Enter the size of the pages:
7
Enter the page strings:
1303563
Enter the no of page frames:
3
LU Page Faults: 5, Page Hits: 2
```

→Optimal

```
#include <stdio.h>

int main() {
    int n, frames, i, j, k, page_faults = 0, hits = 0;

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

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

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

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

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

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

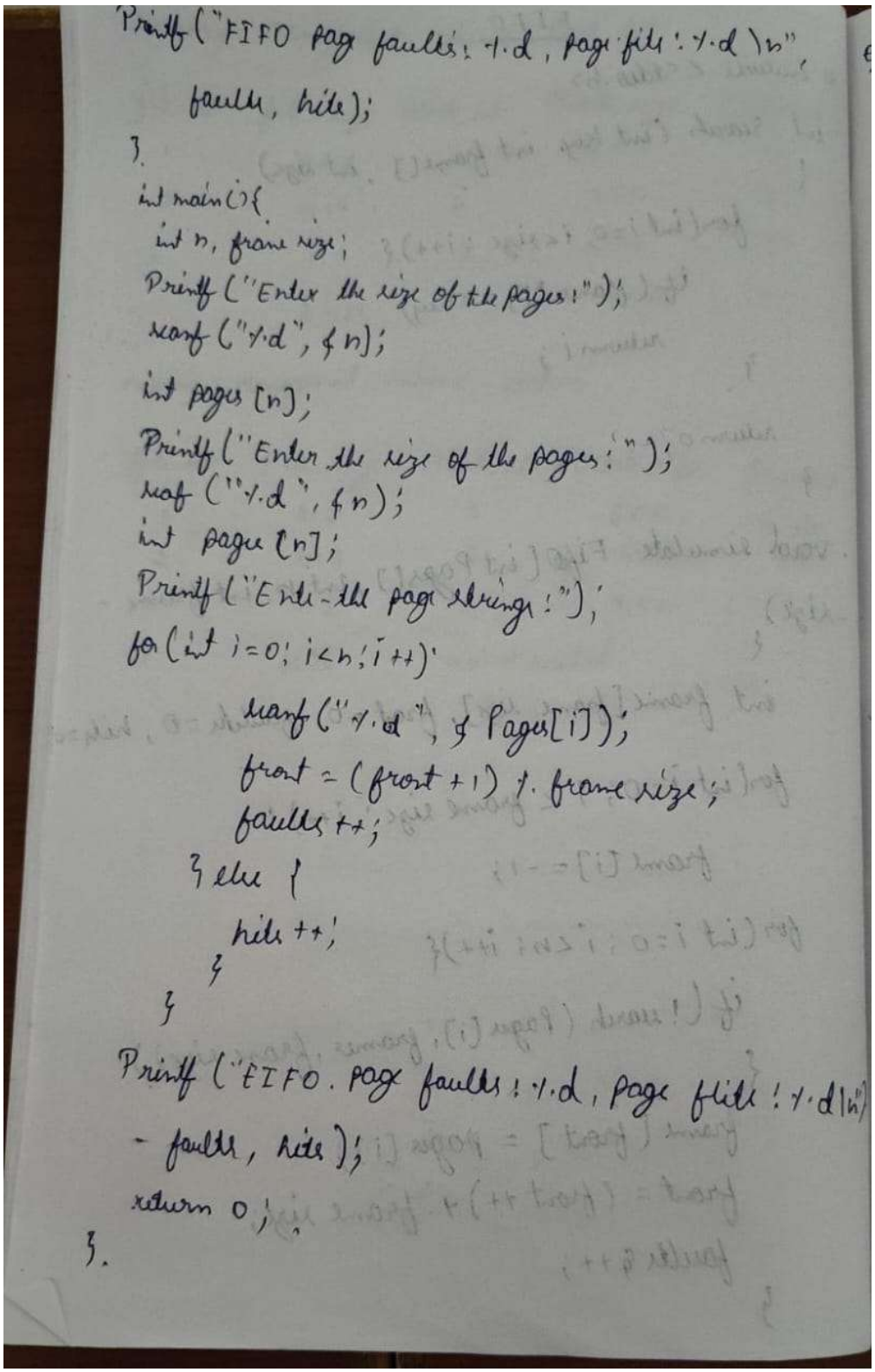
```
        }

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

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

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

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

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



FIFO

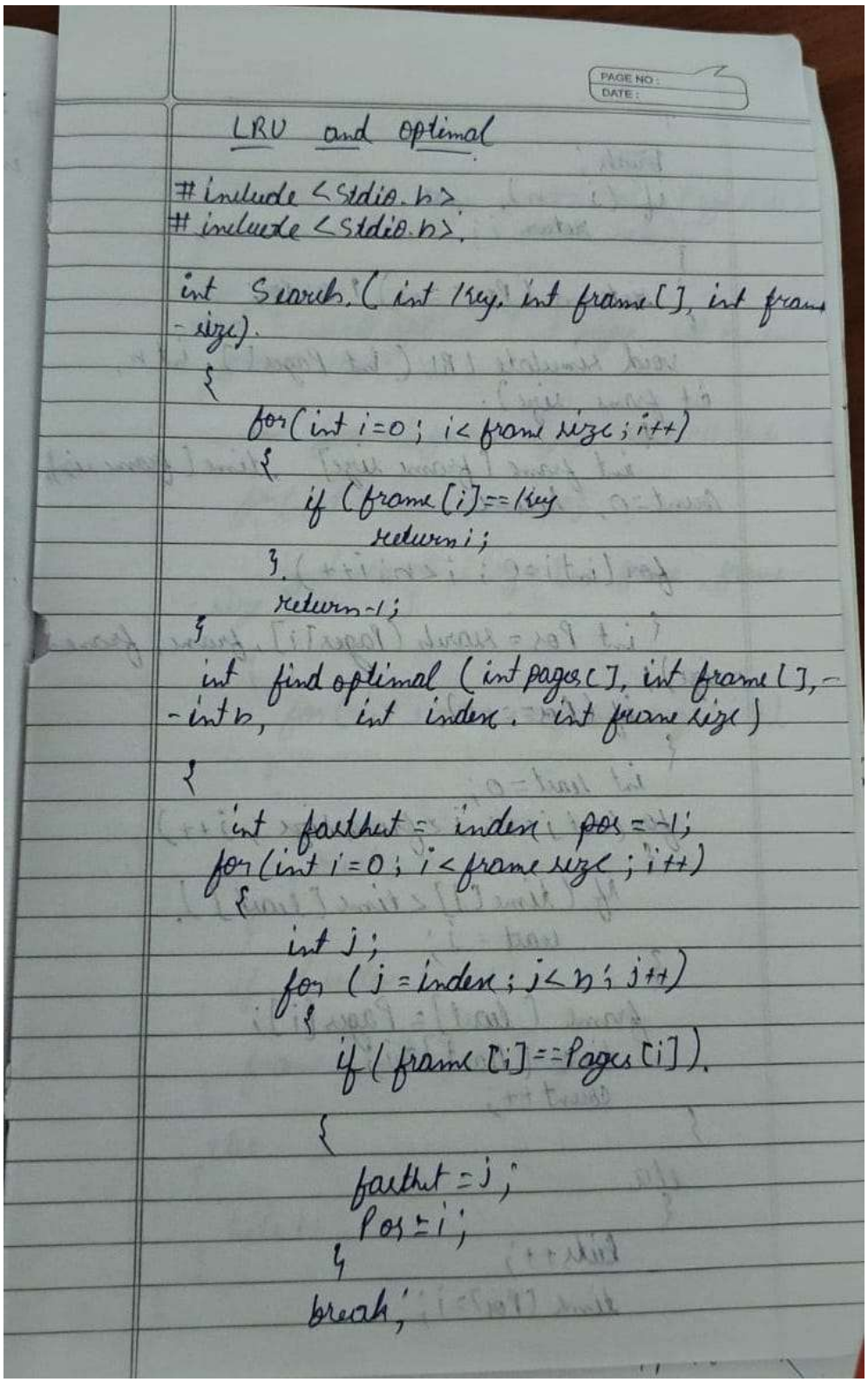
```
#include <stdio.h>
int search (int key, int frame[], int size)
{
    for(int i=0; i<size ; i++){
        if (frame[i] == key)
            return i;
    }
    return 0;
}
void simulate FIFO(int Pages[], int n, int frame_size)
{
    int frame[frame_size], front=0, faults=0, hits=0;
    for(int i=0; i < frame_size ; i++)
        frame[i] = -1;
    for(int i=0; i<n; i++){
        if (!search (Pages[i], frames, frame_size))
        {
            frame[front] = Pages[i];
            front = (front++) % frame_size;
            faults++;
        }
```



```
Printf("FIFO Page faults: %d, Page hits: %d\n",
    faults, hits);
}
int main(){
 int n, frame size;
 Printf("Enter the size of the pages:");
 scanf("%d", &n);
 int pages[n];
 Printf("Enter the size of the pages:");
 scanf("%d", &n);
 int page[n];
 Printf("Enter the page strings:");
for(int i=0; i<n; i++)
        scanf("%d", &Pages[i]);
        front = (front+1) % frame size;
        faults++;
    } else {
      hits++;
    }
  }
Printf("FIFO. Page faults: %d, Page hits: %d\n",
- faults, hits);
return 0;
}
```

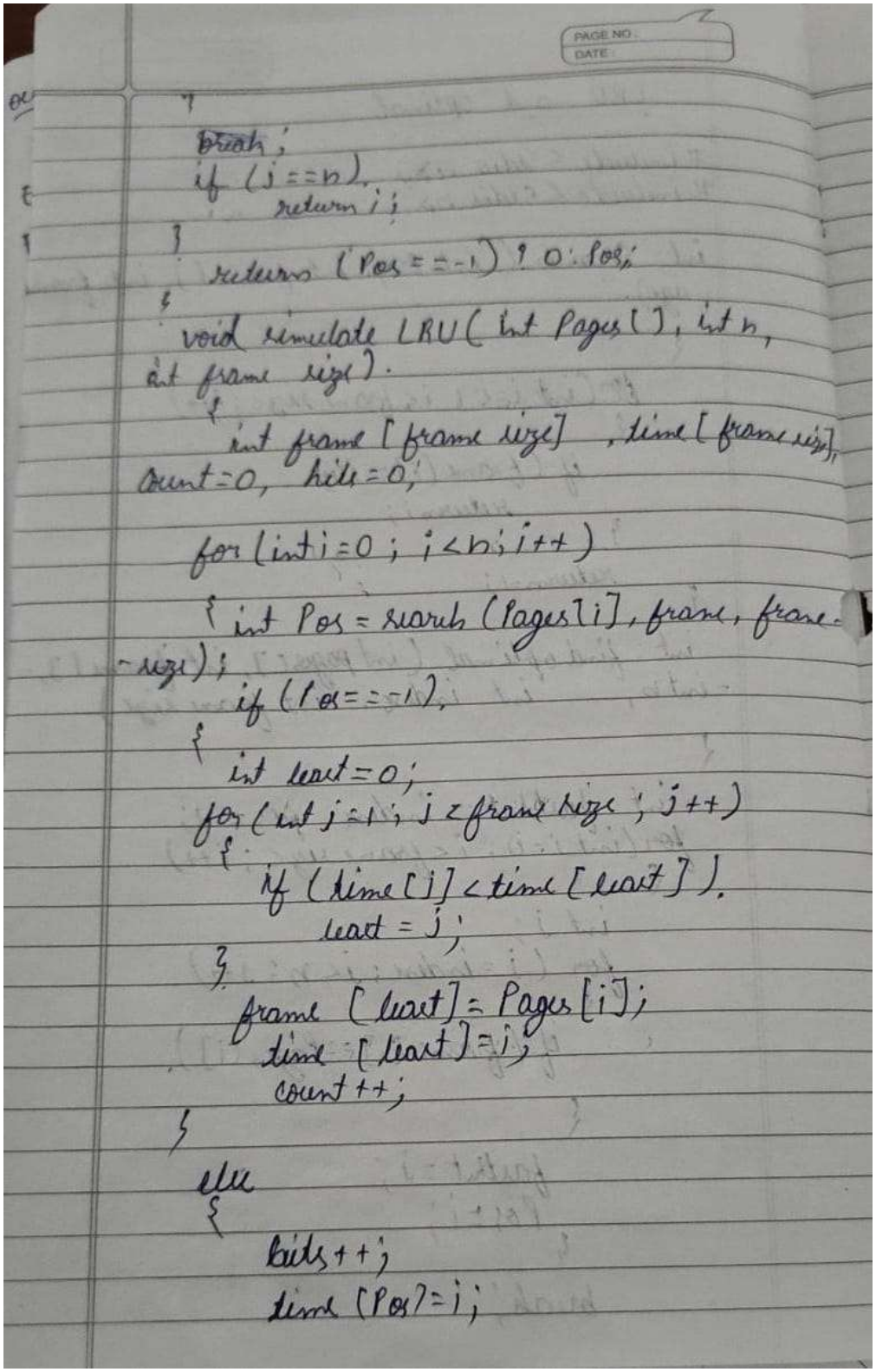
output:

Enter the size of pages : 1
Enter the pages strings : 1 3 0 3 5 6 3.
FIFO Pages faults : 6
Page hits : 1.



LRU and optimal

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

int Search( int key, int frame[], int frame_size)
{
    for(int i=0; i<frame size; i++)
    {
        if (frame[i]==key
            return i;
    }
    return -1;
}
int find optimal (int pages[], int frame[], int n, int index, int frame size)
{
    int farthut = index, pos = -1;
    for(int i=0; i<frame size; i++)
    {
        int j;
        for (j = index; j<n; j++)
        {
            if (frame[i]==Pages[i]).
            {
                farthut = j;
                Pos = i;
            }
            break;
```



```
    break;
    if (j==n)
        return i;
}
    return (pos==-1) ? 0 : pos;
}
void simulate LRU( int Pages[], int n,
int frame size).
{
    int frame [frame size] , time [frame size],
count=0, hit=0;

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

{ int Pos = search (Pages[i], frame, frame
-size);
    if (Pos==-1),
{
    int least=0;
for (int j=1 ; j<frame size ; j++)
{
    if (time[j] < time[least]).
        least = j;
}
    frame [least] = Pages [i];
    time [least] = i;
    count++;
}
else
{
    hits++;
    time [Pos] = i;
```

```
    }
  }
  Printf("RU Page fault : %d , Page Hits :
      %d \n", count, hits);
}

void Simulate optimal (int pages , int n, int
- frame size)
{
  int frame [frame size], count = 0; hits = 0;
  for(int i=0; i<n; i++)
  {
    if (search (Pages [i], frame , frame -
- size ) == -1)
    {
      for (int j = 0 ; j < frame size ; j++)
      {
        index = j ;
        break;
      }
    }
    if (index != -1)
    {
      frame [index] = Pages [i];
    }
    else
    {
      hits ++ ;
    }
}
```

```
int main()
{
    int n, framesize;
    Printf ("Enter the size of the -
        Pages :");
    scanf ("%d", &n);
    int Pages [n];
    Printf ("Enter the page stringe : \n");
    for (int i=0 ; i<n; i++)
        scanf("%d", & pages [i]);

    Printf ("Enter the no of pages, n, frame-
    -size );
        scanf ("%d", & framesize);
    Printf
    simulate optimal (Pages, n, framesize
    simulate LRU (Pages, n, frame size);
    return 0;
}
```

O/P

Enter the size of Pages = 12
Enter the page stringe : 7 0 1 2 0 3 0 4 2 3
0 3

Enter the no of frames : 3
optimal page Faults : 7. page hits 5
LRU Page Faults : 9,

Page hits : 3,

```
int main()
{
    int n, framesize;
    Printf ("Enter the size of the -
        Pages :");
    scanf ("%d", &n);
    int Pages [n];
    Printf ("Enter the page stringe : \n");
    for (int i=0 ; i<n; i++)
        scanf("%d", & pages [i]);

    Printf ("Enter the no of pages, n, frame-
    -size );
        scanf ("%d", & framesize);
    Printf
    simulate optimal (Pages, n, framesize
    simulate LRU (Pages, n, frame size);
    return 0;
}
```

O/P

Enter the size of Pages = 12
Enter the page stringe : 7 0 1 2 0 3 0 4 2 3
0 3

Enter the no of frames : 3
optimal page Faults : 7. page hits 5
LRU Page Faults : 9,

Page hits : 3,