VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

OPERATING SYSTEMS

Submitted by

VIJETH M D (1WA23CS041)

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 VIJETH M D (1WA23CS041), 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.

Seema Patil
Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

Index Sheet

FIFO

	b) LRU c) Optimal	

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.

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

CODE:

```
#include <stdio.h>
void main()
{
   int n;
   printf("Enter number of processes: ");
   scanf("%d", &n);
   int AT[n], BT[n], WT[n], TAT[n], CT[n], RT[n];
   printf("Enter AT and BT for each process:\n");
   for (int i = 0; i < n; i++) {
      printf("Enter AT and BT for the process %d: ", i + 1);
      scanf("%d %d", &AT[i], &BT[i]);
   }
   int currentTime = 0;
   float totalWT = 0, totalTAT = 0;
   printf("\nProcess\tAT\tBT\tWT\tTAT\tRT\n");
   for (int i = 0; i < n; i++) {
       if (currentTime < AT[i])
          currentTime = AT[i];
      CT[i] = currentTime + BT[i];
      TAT[i] = CT[i] - AT[i];
      WT[i] = TAT[i] - BT[i];
      RT[i] = CT[i] - AT[i];
      totalWT += WT[i];
      totalTAT += TAT[i];
      printf("%d\t%d\t%d\t%d\t%d\t\t%d\n", i + 1, AT[i], BT[i], WT[i], TAT[i], RT[i]);
      currentTime = CT[i];
   }
```

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

OUTPUT:

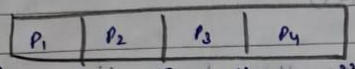
```
Enter number of processes: 5
Enter AT and BT for each process:
Enter AT and BT for the process 1: 0 7
Enter AT and BT for the process 2: 0 3
Enter AT and BT for the process 3: 0 6
Enter AT and BT for the process 4: 0 9
Enter AT and BT for the process 5: 0 2

Process AT      BT      WT      TAT     RT
1       0       7       0       7               7
2       0       3       7       10              10
3       0       6       10      16              16
4       0       9       16      25              25
5       0       2       25      27              27

Average Waiting Time: 11.60
Average Turnaround Time: 17.00
```

LAB Program-1

1 Write a C program to simulate FCFS

Process	AT	BT	CT	TAT	WT
P1	0	5	5	5	0
P2	1	3	8	7	4
P3	2	8	16	14	6
P4	3	6	22	19	13

Gantt chart



$TAT = \frac{5+7+14+19}{4} = 11.25$ ms

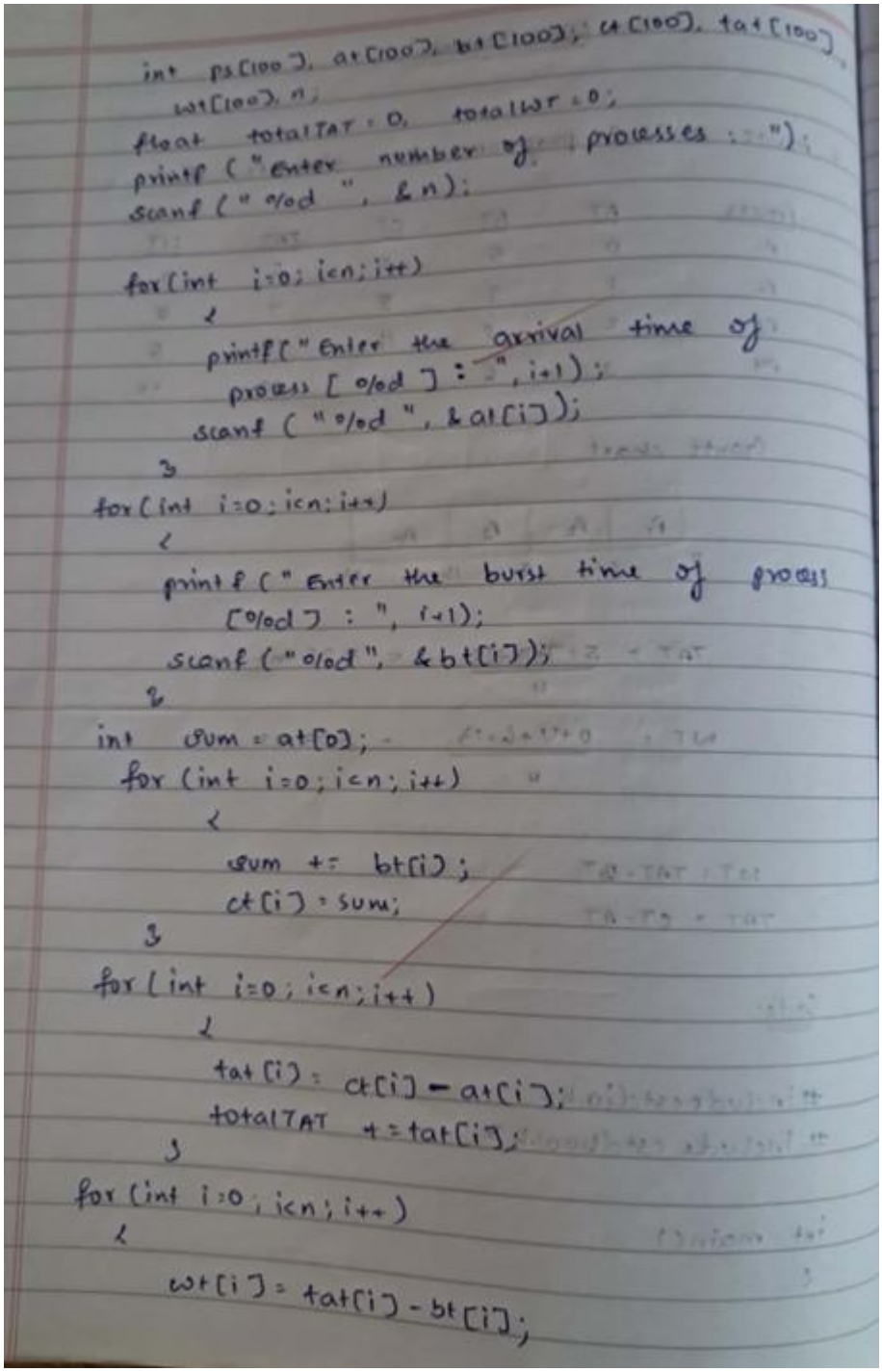
$WT = \frac{0+4+6+13}{4} = 5.75$ ms

WT = TAT - BT

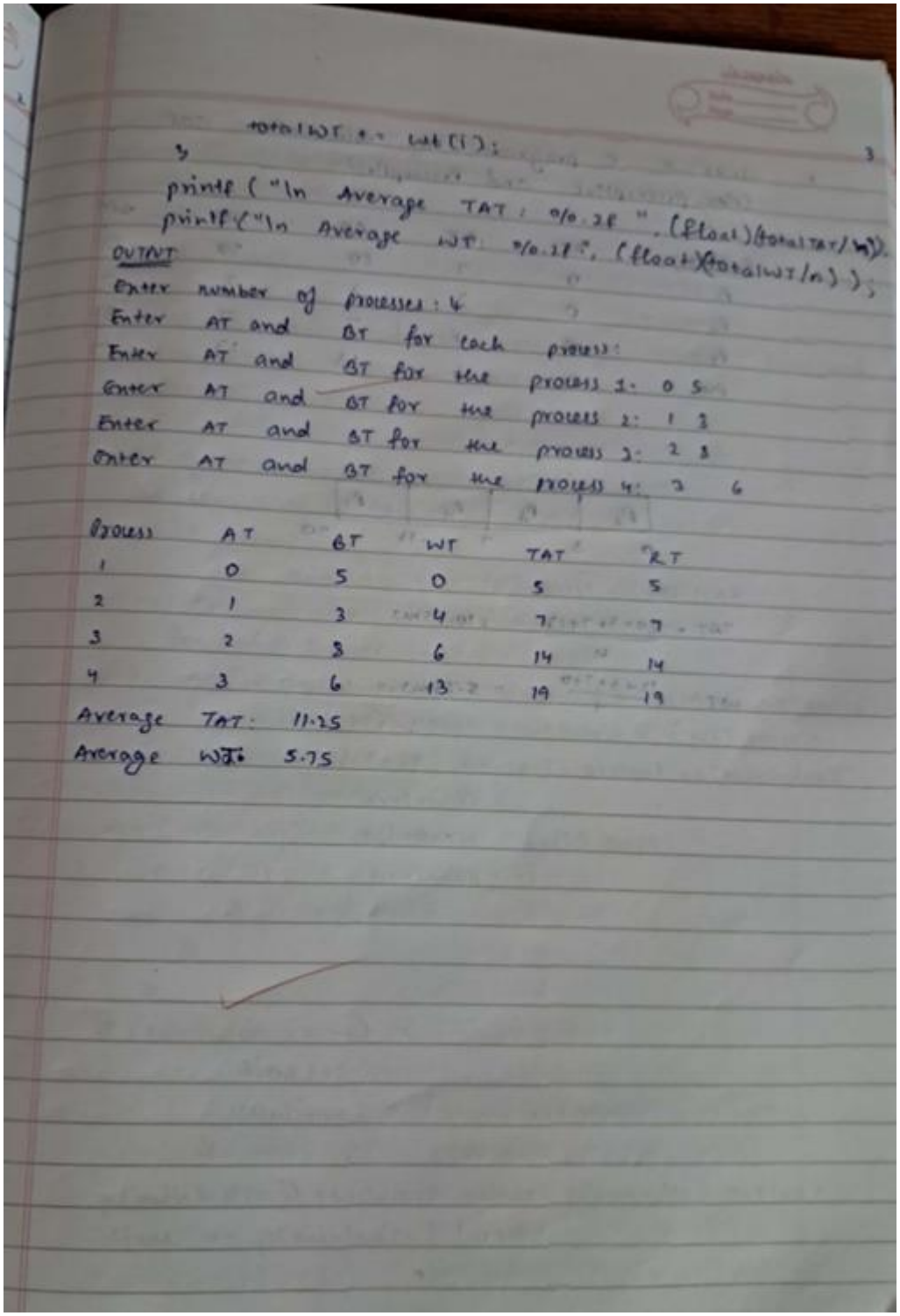
TAT = CT - AT

Code:

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

int main()
{
```



```
int ps[100], at[100], bt[100], ct[100], tat[100]
    wt[100], n;
float totalTAT = 0, totalWT = 0;
printf ("Enter number of processes :-");
scanf ("%d", &n);

for(int i=0; i<n; i++)
    {
    printf(" Enter the arrival time of
     process [%d] : ", i+1);
    scanf ("%d", &at[i]);
    }
for(int i=0; i<n; i++)
    {
    printf (" Enter the burst time of process
       [%d] : ", i+1);
    scanf ("%d", &bt[i]);
    }
int sum = at[0];
 for (int i=0; i<n; i++)
      {
       sum += bt[i];
       ct[i] = sum;
    }
for (int i=0; i<n; i++)
      {
       tat[i] = ct[i] - at[i];
       totalTAT += tat[i];
      }
for (int i=0; i<n; i++)
  {
    wt[i] = tat[i] - bt[i];
```



```
        totalWT += wt[i];
    }
    printf("\n Average TAT : %0.2f ", (float)totalTAT/n);
    printf("\n Average WT : %0.2f", (float)totalWT/n);
```

OUTPUT

```
Enter number of processes : 4
Enter AT and BT for each process:
Enter AT and BT for the process 1: 0 5
Enter AT and BT for the process 2: 1 3
Enter AT and BT for the process 3: 2 3
Enter AT and BT for the process 4: 3 6
```

Process	AT	BT	WT	TAT	RT
1	0	5	0	5	5
2	1	3	4	7	7
3	2	3	6	14	14
4	3	6	13	19	19

Average TAT: 11.25

Average WT: 5.75

Write a C program to simulate the SJF CPU scheduling algorithm to find turnaround time and waiting time.(Non Preemptive)

CODE:

```
#include <stdio.h>
void NPSJF(int n, int AT[], int BT[], int CT[], int TAT[], int WT[], int RT[])
{
   int completed = 0, time = 0, min_BT, shortest, finish_time;
   int remaining_BT[n];
   for (int i = 0; i < n; i++)
   {
      remaining_BT[i] = BT[i];
   }

   while (completed < n)
   {
      min_BT = 9999;
      shortest = -1;
      for (int i = 0; i < n; i++)
      {
         if (AT[i] <= time && remaining_BT[i] > 0 && BT[i] < min_BT)
         {
            min_BT = BT[i];
            shortest = i;
         }
      }
      if (shortest == -1)
      {
         time++;
         continue;
      }
```

```
        time += BT[shortest];
        remaining_BT[shortest] = 0;
        completed++;
        CT[shortest] = time;
        TAT[shortest] = CT[shortest] - AT[shortest];
        WT[shortest] = TAT[shortest] - BT[shortest];
        RT[shortest] = WT[shortest];
    }
}

void display(int n, int AT[], int BT[], int CT[], int TAT[], int WT[], int RT[])
{
    printf("\nProcess\tAT\tBT\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\n", i + 1, AT[i], BT[i], CT[i], TAT[i], WT[i],
RT[i]);
    }
}

int main()
{
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    int AT[n], BT[n], CT[n], TAT[n], WT[n], RT[n];
    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++)
    {
        printf("Enter process %d AT: ", i + 1);
        scanf("%d", &AT[i]);
```

```
        printf("Enter process %d BT: ", i + 1);
        scanf("%d", &BT[i]);
    }
    NPSJF(n, AT, BT, CT, TAT, WT, RT);
    display(n, AT, BT, CT, TAT, WT, RT);
    return 0;
}
```

OUTPUT:

```
Enter number of processes: 5
Enter Arrival Time and Burst Time for each process:
Enter process 1 AT: 5
Enter process 1 BT: 4
Enter process 2 AT: 6
Enter process 2 BT: 9
Enter process 3 AT: 7
Enter process 3 BT: 3
Enter process 4 AT: 7
Enter process 4 BT: 2
Enter process 5 AT: 9
Enter process 5 BT: 6

Process AT      BT      CT      TAT     WT      RT
1       5       4       9       4       0       0
2       6       9       29      23      14      14
3       7       3       14      7       4       4
4       7       2       11      4       2       2
5       9       6       20      11      5       5
```

Write a C program to simulate the SJF CPU scheduling algorithm to find turnaround time and waiting time.(Preemptive)

CODE:

```
#include <stdio.h>
void preemptiveSJF(int n, int AT[], int BT[], int CT[], int TAT[], int WT[], int RT[])
{
   int remaining_BT[n];
   int completed = 0, time = 0, min_BT, shortest;
   int flag[n];
   for (int i = 0; i < n; i++)
   {
      remaining_BT[i] = BT[i];
      flag[i] = 0;
   }

   while (completed < n)
   {
      min_BT = 9999;
      shortest = -1;
      for (int i = 0; i < n; i++)
      {
         if (AT[i] <= time && remaining_BT[i] > 0 && remaining_BT[i] < min_BT &&
flag[i] == 0)
         {
            min_BT = remaining_BT[i];
            shortest = i;
         }
      }
      if (shortest == -1)
      {
         time++;
```

```
            continue;
        }
        remaining_BT[shortest]--;
        if (remaining_BT[shortest] == 0)
        {
            completed++;
            flag[shortest] = 1;
            CT[shortest] = time + 1;
            TAT[shortest] = CT[shortest] - AT[shortest];
            WT[shortest] = TAT[shortest] - BT[shortest];
            RT[shortest] = WT[shortest];
        }
        time++;
    }
}

void display(int n, int AT[], int BT[], int CT[], int TAT[], int WT[], int RT[])
{
    printf("\nProcess\tAT\tBT\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\n", i + 1, AT[i], BT[i], CT[i], TAT[i], WT[i],
RT[i]);
    }
}

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

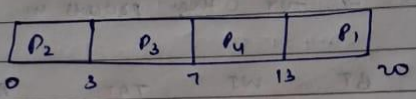
```
    int AT[n], BT[n], CT[n], TAT[n], WT[n], RT[n];
    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++)
    {
        printf("Enter process %d  Arrival Time: ", i + 1);
        scanf("%d", &AT[i]);
        printf("Enter process %d  Burst Time: ", i + 1);
        scanf("%d", &BT[i]);
    }
    preemptiveSJF(n, AT, BT, CT, TAT, WT, RT);
    display(n, AT, BT, CT, TAT, WT, RT);

    return 0;
}
```

OUTPUT:

```
Enter number of processes: 5
Enter Arrival Time and Burst Time for each process:
Enter process 1  Arrival Time: 4
Enter process 1  Burst Time: 2
Enter process 2  Arrival Time: 6
Enter process 2  Burst Time: 3
Enter process 3  Arrival Time: 9
Enter process 3  Burst Time: 1
Enter process 4  Arrival Time: 5
Enter process 4  Burst Time: 3
Enter process 5  Arrival Time: 7
Enter process 5  Burst Time: 2

Process AT      BT      CT      TAT     WT      RT
1       4       2       6       2       0       0
2       6       3       9       3       0       0
3       9       1       10      1       0       0
4       5       3       15      10      7       7
5       7       2       12      5       3       3
```

2. Write a C program to simulate SJF (Non-preemptive and Preemptive).

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



$TAT = \frac{20+3+7+13}{4} = 10.75ms$

$WT = \frac{13+3+7+0}{4} = 5.75ms.$

Code:

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

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

void sjfNonPreemptive (Process p[], int n) {
    int completed = 0, time = 0;
    while (completed < n) {
        int minIndex = -1, minBurst = INT_MAX;
        for (int i=0; i<n; i++) {
            if (p[i].arrival <= time && p[i].completion==0){
                if (p[i].burst < minBurst || (p[i].burst ==
                    minBurst && p[i].arrival < p[minIndex].
                    arrival)) {
                    minBurst = p[i].burst;
                    minIndex = i;
                }
            }
        }
        if (minIndex == -1) {
            time++;
            continue;
        }
        p[minIndex].response = time - p[minIndex].arrival;
        time += p[minIndex].burst;
```

```
p[minIndex].completion = time;
p[minIndex].turnaround = p[minIndex].completion -
                         p[minIndex].arrival;
p[minIndex].waiting = p[minIndex].turnaround -
                      p[minIndex].burst;
completed++;
}
}

void sjfPreemptive (Process p[], int n) {
    int completed = 0, time = 0, minIndex = -1,
        minBurst = INT_MAX;

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

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

```
    if (p[minIndex].started == 0) {
        p[minIndex].response = time - p[minIndex].arrival;
        p[minIndex].started = 1;
    }
    p[minIndex].remaining--;
    time++;

    if (p[minIndex].remaining == 0) {
        p[minIndex].completion = time;
        p[minIndex].turnaround = p[minIndex].completion -
                                 p[minIndex].arrival;
        p[minIndex].waiting = p[minIndex].turnaround -
                              p[minIndex].burst;
        completed++;
    }
}
}

void displayResults(Process p[], int n, const char *title) {
    printf("%s", title);
    printf("\n PID \t AT \t BT \t CT \t TAT \t
           WT \t RT \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 \t %d ",
        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 WT : %0.2f \n", totalWT /n);
    printf ("Average TAT : %0.2f \n", totalTAT /n);
    printf ("Average RT : %0.2f \n", totalRT /n);
}

int main()
{
    int n, choice;
    printf ("Enter no. of processes :");
    scanf ("%d", &n);

    Process p[n], temp[n];

    printf ("Enter AT and BT : \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;
            temp[i] = p[i];
        }

    printf ("RESULTS \n");

    sjfNonPreemptive (p, n);
    displayResults (p, n, "SJF (Non-preemptive)");

    sjfPreemptive (p, n);
    displayResults (p, n, "SJP (preemptive)");
```

OUTPUT:

```
Enter no. of processes : 4
Enter AT and BT:
Enter AT and BT for process 1: 0 5
Enter AT and BT for process 2: 1 3
Enter AT and BT for process 3: 2 8
Enter AT and BT for process 4: 3 6
```

SJF (Non preemptive):

Process	AT	BT	CT	TAT	WT	RT
1	0	5	5	5	0	0
2	1	3	8	7	4	4
3	2	8	22	20	12	12
4	3	6	14	11	5	5

Average WT: 7.00
Average TAT: 14.33
Average RT: 7.00

SJF (Preemptive):

Process	AT	BT	CT	TAT	WT	RT
1	0	5	8	8	3	3
2	1	3	4	3	0	0
3	2	8	22	20	12	12
4	3	6	14	11	5	5

Average WT: 6.67
Average TAT: 14.00
Average RT: 6.67

Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time. PRIORITY (Non Preemptive)

CODE:

```
#include <stdio.h>


void priorityScheduling(int n, int at[], int bt[], int pr[], int ct[], int tat[], int wt[], int rt[]) {
    int completed = 0, time = 0, min_priority, highest_priority;
    int flag[n];
    for (int i = 0; i < n; i++) {
        flag[i] = 0;
    }

    while (completed < n) {
        min_priority = 9999;
        highest_priority = -1;
        for (int i = 0; i < n; i++) {
            if (at[i] <= time && flag[i] == 0 && pr[i] < min_priority) {
                min_priority = pr[i];
                highest_priority = i;
            }
        }
        if (highest_priority == -1) {
            time++;
            continue;
        }
        time += bt[highest_priority];
        flag[highest_priority] = 1;
        ct[highest_priority] = time;
        tat[highest_priority] = ct[highest_priority] - at[highest_priority];
        wt[highest_priority] = tat[highest_priority] - bt[highest_priority];
        rt[highest_priority] = wt[highest_priority];
```

```
        completed++;
    }
}

void displayTable(int n, int at[], int bt[], int pr[], int ct[], int tat[], int wt[], int rt[]) {
    printf("\nProcess\tAT\tBT\tPriority\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i], pr[i], ct[i], tat[i], wt[i],
rt[i]);
    }
}

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

    int at[n], bt[n], pr[n], ct[n], tat[n], wt[n], rt[n];
    printf("Enter Arrival Time, Burst Time, and Priority for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d - Arrival Time: ", i + 1);
        scanf("%d", &at[i]);
        printf("Process %d - Burst Time: ", i + 1);
        scanf("%d", &bt[i]);
        printf("Process %d - Priority: ", i + 1);
        scanf("%d", &pr[i]);
    }
    priorityScheduling(n, at, bt, pr, ct, tat, wt, rt);
    displayTable(n, at, bt, pr, ct, tat, wt, rt);
```

```
    return 0;
}
```

OUTPUT:

```
Enter number of processes: 4
Enter Arrival Time, Burst Time, and Priority for each process:
Process 1 - Arrival Time: 4
Process 1 - Burst Time: 6
Process 1 - Priority: 2
Process 2 - Arrival Time: 3
Process 2 - Burst Time: 7
Process 2 - Priority: 5
Process 3 - Arrival Time: 9
Process 3 - Burst Time: 6
Process 3 - Priority: 1
Process 4 - Arrival Time: 4
Process 4 - Burst Time: 3
Process 4 - Priority: 1

Process AT      BT      Priority        CT      TAT     WT      RT
1       4       6       2               25      21      15      15
2       3       7       5               10      7       0       0
3       9       6       1               16      7       1       1
4       4       3       1               19      15      12      12
```

20/03/2025

⑧ Write a program to implement Non preemptive priority scheduling

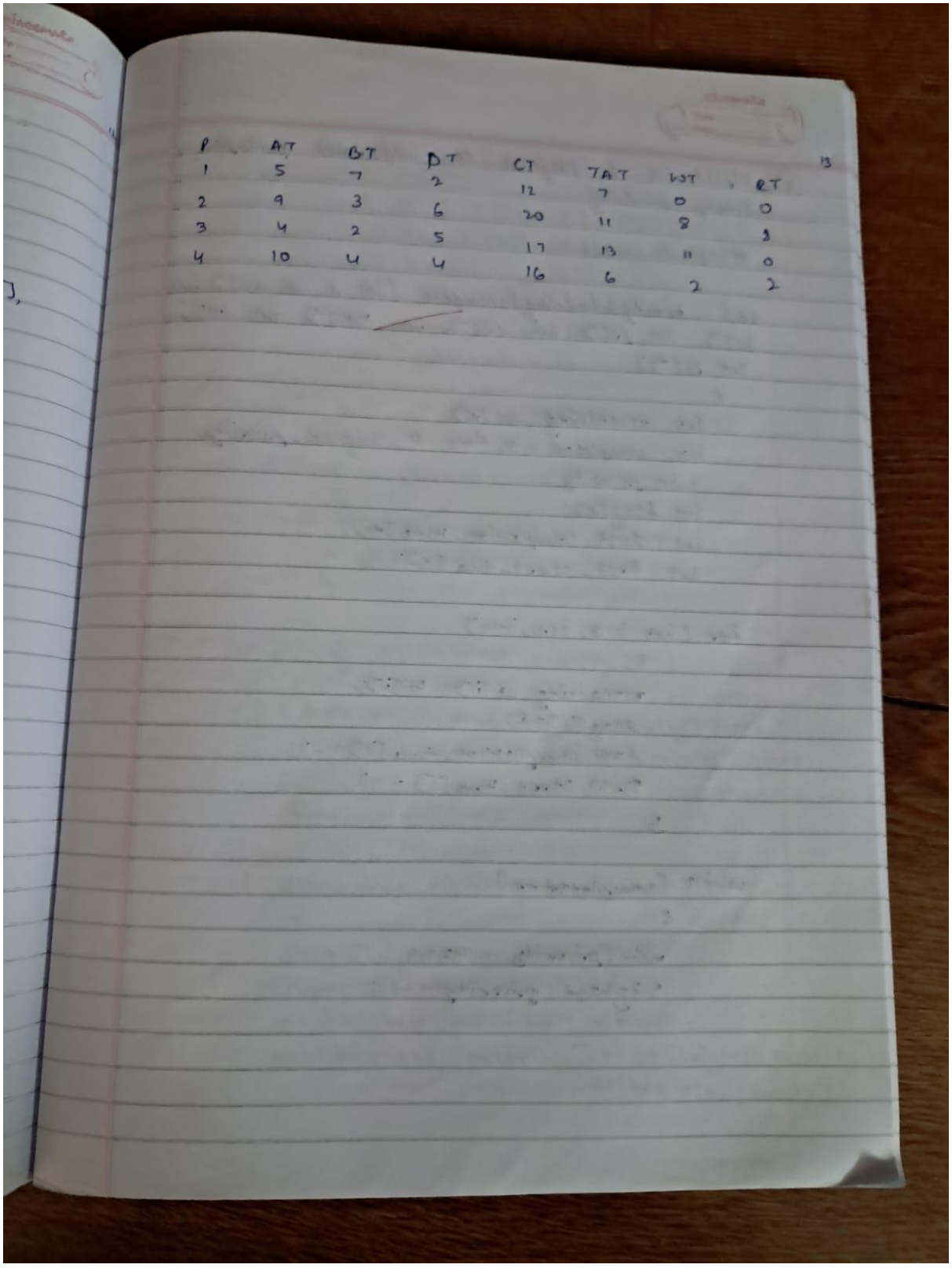
```
#include<stdio.h>

void priorityScheduling(int n, int at[], int bt[],
 int pr[], int ct[], int tat[], int wt[], int rt[])
{
   int completed=0, time=0, min_priority, highest_
   priority;
   int flag[n];
   for(int i=0; i<n; i++){
         flag[i]=0;
    }
  while(completed<n){
     {
      min_priority=9999;
      highest_priority=-1;
      for(int i=0; i<n; i++)
        {
        if(at[i]<=time && flag[i]==0 &&
             pr[i]<min_priority)
             {
              min_priority = pr[i];
              highest_priority = i;
             }
        }
```

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

time += bt[highest_priority];
flag[highest_priority] = 1;
ct[highest_priority] = time;
tat[highest_priority] = ct[highest_priority] -
                        at[highest_priority];
wt[highest_priority] = tat[highest_priority] -
                       bt[highest_priority];
rt[highest_priority] = wt[highest_priority];
completed++;
}
}

void displayTable(int n, int at[], int bt[], int pr[],
int ct[], int tat[], int wt[], int rt[])
{
printf("\n Process \t AT \t BT \t Priority \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 \n",
    i+1, at[i], bt[i], pr[i], ct[i], tat[i], wt[i],
        rt[i]);
}
}
```

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

int at[n], bt[n], pr[n], ct[n], tat[n], wt[n],
 rt[n];
printf ("Enter Arrival Time, Burst Time,
   and Priority for each process: \n");
for (int i=0 ; i<n ; i++) {
   printf ("Process %d - Arrival Time : ", i+1);
   scanf ("%d", & at[i]);
   printf ("Process %d - Burst Time :", i+1);
   scanf ("%d", & bt[i]);
   printf ("Process %d - Priority :", i+1);
   scanf ("%d ", &pr[i]);
}
priorityScheduling (n, at, bt, pr, ct, tat, wt, rt);
displayTable (n, at, bt, pr, ct, tat, wt, rt);

return 0;
}
```

OUTPUT:

```
Enter no of processes : 4
Enter AT, BT and priority of process 1 : 5 7 2
Enter AT, BT and priority of process 2 : 9 3 6
Enter AT, BT and priority of process 3 : 4 2 5
Enter AT, BT and priority of process 4 : 10 4 4
```



P	AT	BT	DT	CT	TAT	WT	RT
1	5	7	2	12	7	0	0
2	9	3	6	20	11	8	3
3	4	2	5	17	13	11	0
4	10	4	4	16	6	2	2

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

CODE:

```
#include <stdio.h>


void prioritySchedulingPreemptive(int n, int at[], int bt[], int pr[], int ct[], int tat[], int wt[], int rt[]) {
    int remaining_bt[n];
    int completed = 0, time = 0, highest_priority, min_priority;
    int flag[n];
    int last_completion_time[n];
    int first_start_time[n];



    for (int i = 0; i < n; i++) {
        remaining_bt[i] = bt[i];
        flag[i] = 0;
        last_completion_time[i] = -1;
        first_start_time[i] = -1;
    }

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



        for (int i = 0; i < n; i++) {
            if (at[i] <= time && flag[i] == 0 && pr[i] < min_priority && remaining_bt[i] > 0) {
                min_priority = pr[i];
                highest_priority = i;
            }
```

```
        }

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


        remaining_bt[highest_priority]--;
        if (first_start_time[highest_priority] == -1) {
            first_start_time[highest_priority] = time;
        }
        time++;

        if (remaining_bt[highest_priority] == 0) {

            flag[highest_priority] = 1;
            completed++;
            ct[highest_priority] = time;
            tat[highest_priority] = ct[highest_priority] - at[highest_priority];
            wt[highest_priority] = tat[highest_priority] - bt[highest_priority];
            rt[highest_priority] = first_start_time[highest_priority] - at[highest_priority];
        }
    }
}

void displayTable(int n, int at[], int bt[], int pr[], int ct[], int tat[], int wt[], int rt[]) {
    printf("\nP\tAT\tBT\tPt\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", i + 1, at[i], bt[i], pr[i], ct[i], tat[i], wt[i],
rt[i]);
```

```
    }
}

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

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

    prioritySchedulingPreemptive(n, at, bt, pr, ct, tat, wt, rt);
    displayTable(n, at, bt, pr, ct, tat, wt, rt);

    return 0;
}
```

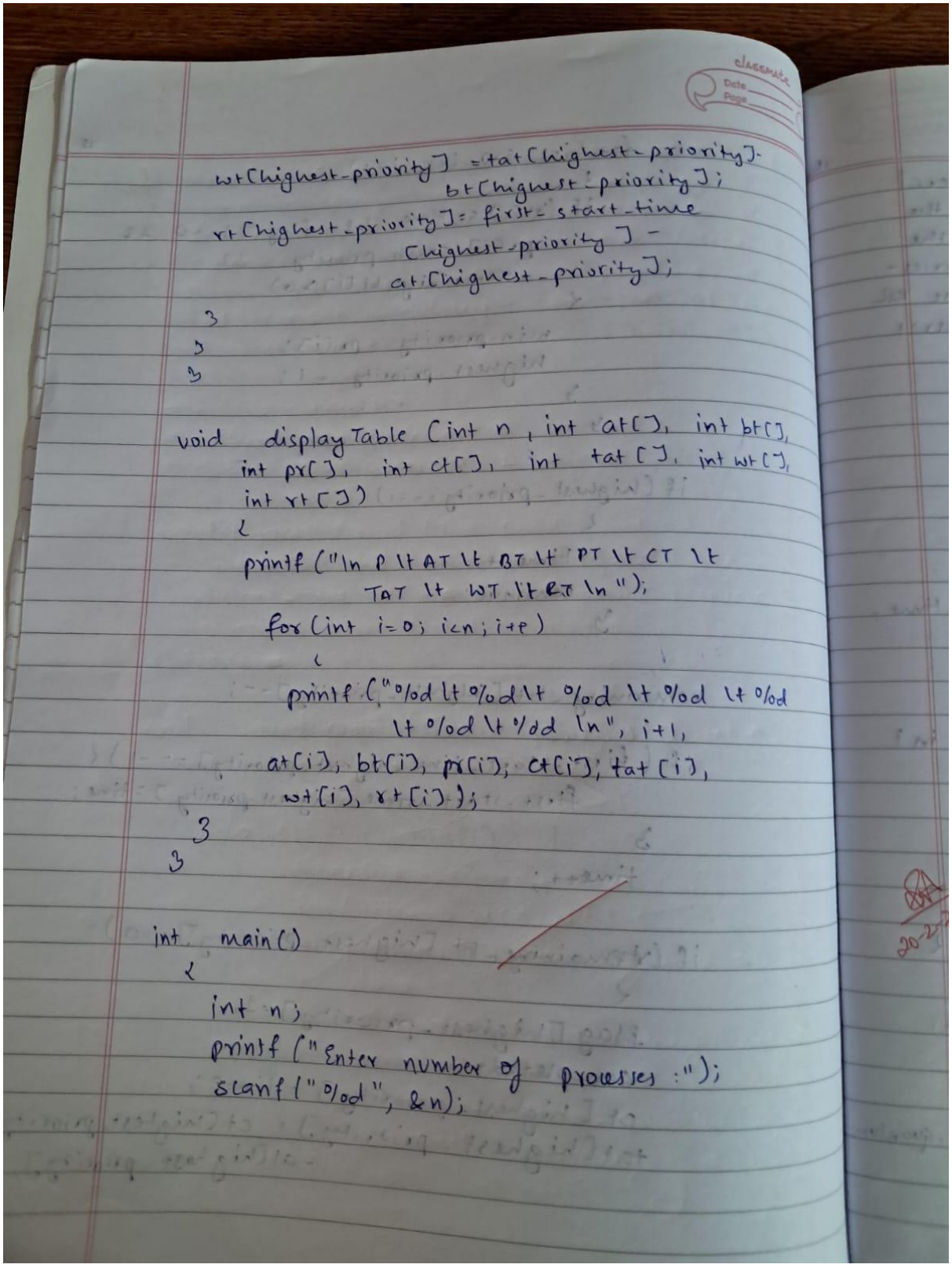
OUTPUT:

```
Enter number of processes: 4
Enter Arrival Time, Burst Time, and Priority for each process:
Process 1 - Arrival Time: 2
Process 1 - Burst Time: 4
Process 1 - Priority: 1
Process 2 - Arrival Time: 8
Process 2 - Burst Time: 7
Process 2 - Priority: 4
Process 3 - Arrival Time: 9
Process 3 - Burst Time: 2
Process 3 - Priority: 3
Process 4 - Arrival Time: 6
Process 4 - Burst Time: 4
Process 4 - Priority: 5

P       AT      BT      Pt      CT      TAT     WT      RT
1       2       4       1       6       4       0       0
2       8       7       4       17      9       2       0
3       9       2       3       11      2       0       0
4       6       4       5       19      13      9       0
```

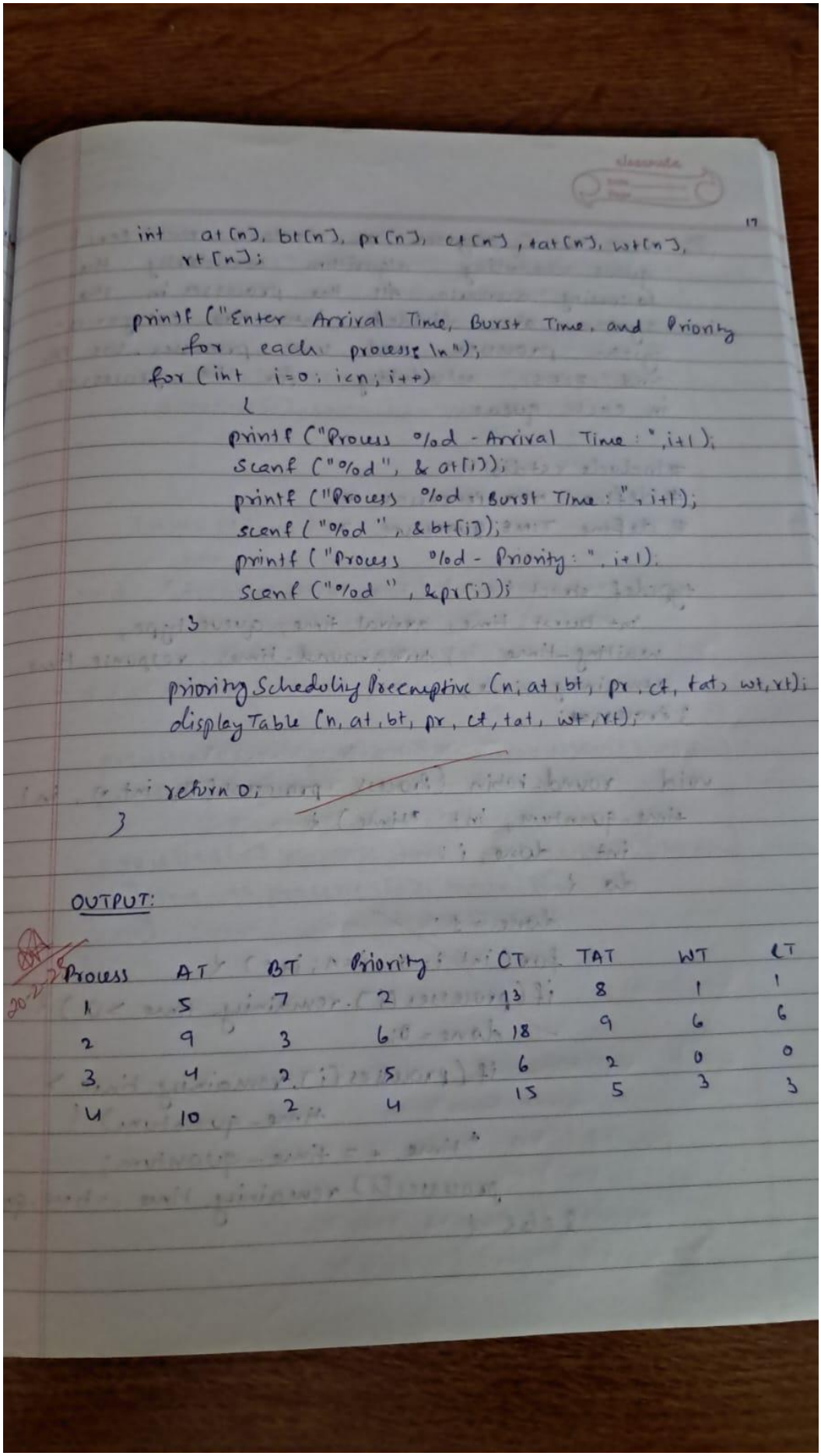
(8) Write a C program to implement preemptive Priority Scheduling.

```
#include<stdio.h>

void priorityScheduling Preemptive (int n, int at[], int
bt[], int pr[], int ct[], int tat[], int wt[],
int rt[])
  {
    int remaining_bt[n];
    int completed = 0, time = 0, highest_priority,
    min_priority;
    int flag[n];
    int last_completion_time[n];
    int first_start_time[n];

for (int i=0; i<n; i++)
      {
        remaining_bt[i]= bt[i];
        flag[i]=0;
        last_completion_time[i]= -1;
        first_start_time[i] = -1;
      }

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

```
for (int i=0; i<n; i++)
    {
    if (at[i] <= time && flag[i]==0 &&
        pr[i] < min_priority &&
        remaining_bt[i] > 0)
    {
        min_priority = pr[i];
        highest_priority = i;
    }
}
if (highest_priority == -1)
{
    time++;
    continue;
}

remaining_bt[highest_priority]--;

if (first_start_time[highest_priority] == -1) {
    first_start_time[highest_priority] = time;
}
time++;

if (remaining_bt[highest_priority] == 0)
{
    flag[highest_priority] = 1;
    completed++;
    ct[highest_priority] = time;
    tat[highest_priority] = ct[highest_priority]
                            - at[highest_priority];
```



```
    wt[highest_priority] = tat[highest_priority] -
                           bt[highest_priority];
    rt[highest_priority] = first_start_time
                           [highest_priority] -
                           at[highest_priority];
  }
}

void displayTable(int n, int at[], int bt[],
    int pr[], int ct[], int tat[], int wt[],
    int rt[])
{
    printf("\n P \t AT \t BT \t PT \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 \n", i+1,
        at[i], bt[i], pr[i], ct[i], tat[i],
        wt[i], rt[i]);
    }
}

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



```
int at[n], bt[n], pr[n], ct[n], tat[n], wt[n],
    rt[n];

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

priorityScheduling Preemptive(n, at, bt, pr, ct, tat, wt, rt);
displayTable(n, at, bt, pr, ct, tat, wt, rt);

return 0;
}
```

OUTPUT:

Process	AT	BT	Priority	CT	TAT	WT	LT
1	5	7	2	13	8	1	1
2	9	3	6	18	9	6	6
3	4	2	5	6	2	0	0
4	10	2	4	15	5	3	3

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

CODE:

```
#include <stdio.h>

void findTurnaroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i];
    }
}

void findWaitingTime(int processes[], int n, int bt[], int wt[], int quantum) {
    int remaining_bt[n];
    for (int i = 0; i < n; i++) {
        remaining_bt[i] = bt[i];
    }

    int time = 0;
    while (1) {
        int done = 1;
        for (int i = 0; i < n; i++) {
            if (remaining_bt[i] > 0) {
                done = 0;
                if (remaining_bt[i] > quantum) {
                    time += quantum;
                    remaining_bt[i] -= quantum;
                } else {
                    time += remaining_bt[i];
                    wt[i] = time - bt[i];
                    remaining_bt[i] = 0;
```

```
            }
        }
    }
    if (done == 1)
        break;
  }
}

void roundRobin(int processes[], int n, int bt[], int quantum) {
  int wt[n], tat[n];

  for (int i = 0; i < n; i++) {
    wt[i] = 0;
  }

  findWaitingTime(processes, n, bt, wt, quantum);
  findTurnaroundTime(processes, n, bt, wt, tat);

  int total_wt = 0, total_tat = 0;
  printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
  for (int i = 0; i < n; i++) {
    total_wt += wt[i];
    total_tat += tat[i];
    printf("P%d\t%d\t\t%d\t\t%d\n", processes[i], bt[i], wt[i], tat[i]);
  }

  printf("\nAverage Waiting Time: %.2f", (float)total_wt / n);
  printf("\nAverage Turnaround Time: %.2f", (float)total_tat / n);
}
```

```
int main() {
    int n, quantum;

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

    int processes[n], bt[n];

    printf("Enter the burst time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d burst time: ", i + 1);
        scanf("%d", &bt[i]);
        processes[i] = i + 1;
    }

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

    roundRobin(processes, n, bt, quantum);

    return 0;
}
```

OUTPUT:

```
Enter the number of processes: 4
Enter the burst time for each process:
Process 1 burst time: 10
Process 2 burst time: 4
Process 3 burst time: 5
Process 4 burst time: 7
Enter the time quantum: 3

Process Burst Time      Waiting Time    Turnaround Time
P1      10              16              26
P2      4               12              16
P3      5               13              18
P4      7               18              25

Average Waiting Time: 14.75
Average Turnaround Time: 21.25
```

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

```
#include<stdio.h>

void findTurnaroundTime(int processes[], int n, int bt[], int wt[],
    int quantum) {
        int remaining_bt[n];
        for(int i=0; i<n; i++) {
            remaining_bt[i] = bt[i]; }
        int time=0;
        while(1) {
            int done=1;
            for(int i=0; i<n; i++) {
                if(remaining_bt[i]>0) {
                    done=0;
                    if(remaining_bt[i]>quantum) {
                        time+=quantum;
                        remaining_bt[i] -= quantum;
                    } else {
                        time += remainig_bt[i];
                        wt[i] = time - bt[i];
                        remainig_bt[i] = 0;
                    }}}
            if(done==1) break;
        }

void roundRobin(int processes[], int n, int bt[], int quantum)
{
    int wt[n], tat[n];
```

```
for(int i=0 ; i<n; i++) {
        wt[i] = 0; }
findWaitingTime(processes, n, bt, wt, quantum);
findTurnaroundTime(processes, n, bt, wt, tat);
int total_wt = 0, total_tat = 0;
printf("\n Process \t Burst \t Waiting Time \t
    Turnaround Time \n");
for(int i=0; i<n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf("%d \t %d \t %d \t %d \n",
        processes[i], bt[i], wt[i], tat[i]);
  }
printf("Avg WT : %.2f", (float) total_wt/n);
printf("\n Avg TAT: %.2f", (float) total_tat/n);
}

int main() {
    int n, quantum;
    printf("Enter the number of processes:");
    scanf("%d", &n);
    int processes[n], bt[n];
    printf("Enter the burst time for each
        process: \n");
    for(int i=0; i<n ; i++) {
        printf("Process %d burst time:", i+1);
        scanf("%d", &bt[i]);
        processes[i] = i+1;
    }
    printf("Enter the time quantum:");
    scanf("%d", &quantum);
    roundRobin(processes, n, bt, quantum);
```

```
return 0; }
```

41

OUTPUT:

```
Enter the number of processes : 4
Enter the burst time for each process :
Process 1 burst time : 10
Process 2 burst time : 4
Process 3 burst time : 5
Process 4 burst time : 7
Enter the quantum time : 3
```

Process	Burst Time	Waiting Time	Turnaround Time
P1	10	16	26
P2	4	12	16
P3	5	13	18
P4	7	18	25

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

CODE:

```
#include <stdio.h>


#define MAX_PROCESSES 10

#define TIME_QUANTUM 2


typedef struct {

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

} Process;


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

   int done, i;

   do {

      done = 1;

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

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

            done = 0;

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

               *time += time_quantum;

               processes[i].remaining_time -= time_quantum;

            } else {

               *time += processes[i].remaining_time;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

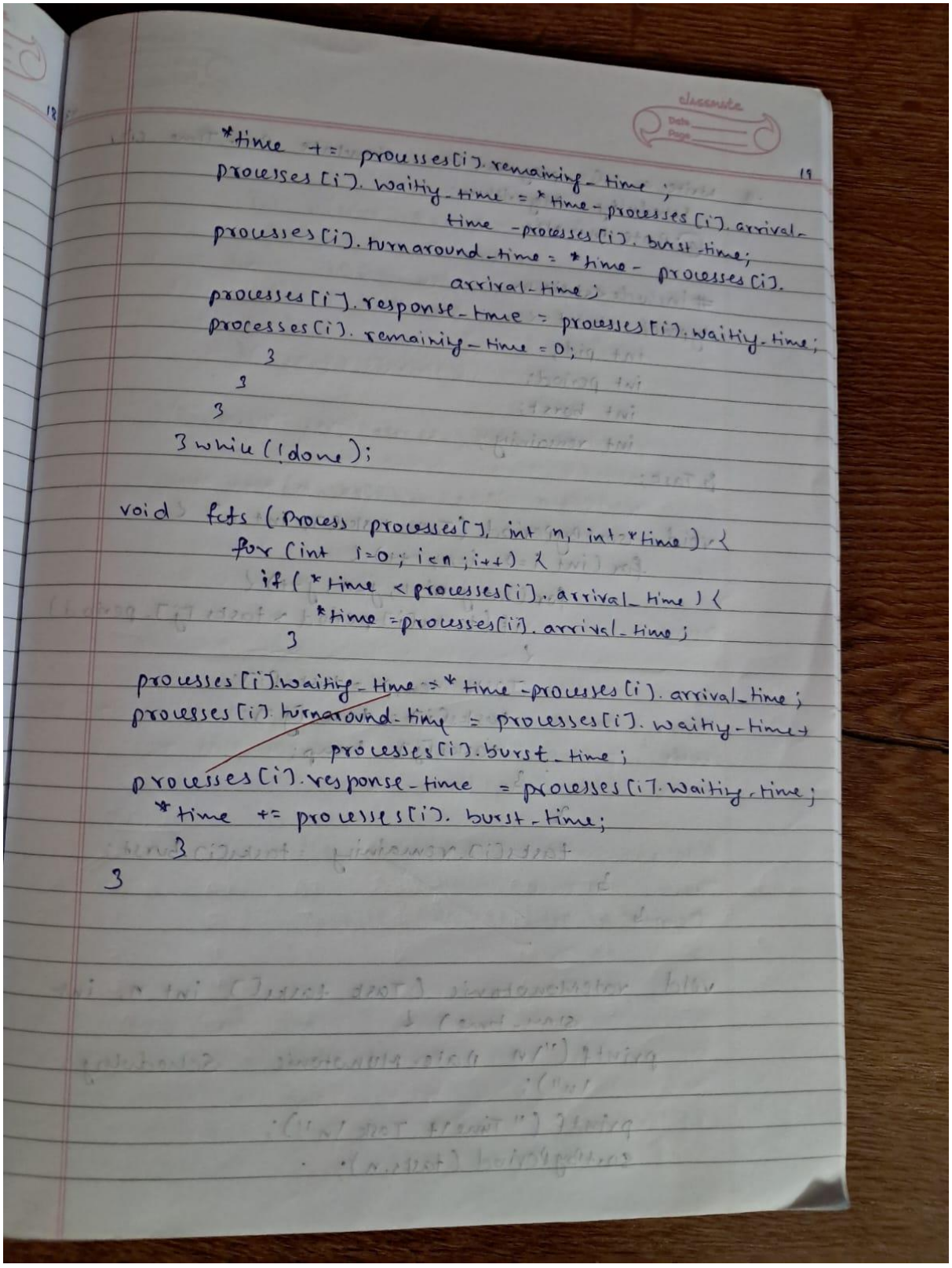
```
        printf("%d        %d        %d        %d\n", i + 1, system_queue[i].waiting_time,
system_queue[i].turnaround_time, system_queue[i].response_time);
    }

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

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

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

    return 0;
}
```

OUTPUT:

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

Queue 1 is System Process
Queue 2 is User Process

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

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

Process returned 0 (0x0)   execution time : 55.213 s
Press any key to continue.
```

(8) 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. Use RR and 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 (int 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_quan
            } 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;
    }
}
```

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

Rate- Monotonic

CODE:

```
#include <stdio.h>

typedef struct {
    int pid;
    int period;
    int burst;
    int remaining;
} Task;

void sortByPeriod(Task tasks[], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            if (tasks[i].period > tasks[j].period) {
                Task temp = tasks[i];
                tasks[i] = tasks[j];
                tasks[j] = temp;
            }
        }
        tasks[i].remaining = tasks[i].burst;
    }
}

void rateMonotonic(Task tasks[], int n, int sim_time) {
    printf("\nRate-Monotonic Scheduling:\n");
    printf("Time\tTask\n");
```

```
    sortByPeriod(tasks, n);

    for (int time = 0; time < sim_time; time++) {
        int scheduled = -1;


        for (int i = 0; i < n; i++) {
            if (time % tasks[i].period == 0) {
                tasks[i].remaining = tasks[i].burst;
            }
        }


        for (int i = 0; i < n; i++) {
            if (tasks[i].remaining > 0) {
                scheduled = i;
                break;
            }
        }


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

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

    Task tasks[n];
    for (int i = 0; i < n; i++) {
        tasks[i].pid = i+1;
        printf("Enter period and burst time for task T%d: ", i+1);
        scanf("%d %d", &tasks[i].period, &tasks[i].burst);
    }

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

    rateMonotonic(tasks, n, sim_time);

    return 0;
}
```

OUTPUT:

```
Enter number of tasks: 2
Enter period and burst time for task T1: 2 1
Enter period and burst time for task T2: 2 3
Enter simulation time: 3

Rate-Monotonic Scheduling:
Time    Task
0       T1
1       T2
2       T1

Process returned 0 (0x0)   execution time : 14.831 s
Press any key to continue.
```

9 Write a C program to simulate Real-Time CPU Scheduling Algorithms:
(a) Rate-Monotonic

```
#include<stdio.h>
typedef struct {
    int pid;
    int period;
    int burst;
    int remaining;
} Task;

void sortByPeriod (Task tasks[], int n) {
    for (int i=0; i<n; i++) {
        for (int j=i+1; j<n; j++) {
            if (tasks[i].period > tasks[j].period)
            {
                Task temp = tasks[i];
                tasks[i] = tasks[j];
                tasks[j] = temp;
            }
        }
        tasks[i].remaining = tasks[i].burst;
    }
}

void rateMonotonic (Task tasks[], int n, int
    sim_time) {
    printf ("\n Rate-Monotonic Scheduling:
    \n");
    printf ("Time\t Task \n");
    sortByPeriod (tasks, n);
```

```
for (int time=0; time<sim_time; time++)
{
    int scheduled = -1;
    for (int i=0; i<n; i++) {
        if (time % tasks[i].period == 0) {
            tasks[i].remaining = tasks[i].burst;
        }
    }

for (int i=0; i<n; i++)
    {
    if (tasks[i].remaining > 0) {
                scheduled = i;
                break;
        }
    }

if (scheduled != -1) {
        tasks[scheduled].remaining --;
        printf ("%d lt T%d\n", time,
            tasks[scheduled].pid);
    }
    else
    {
        printf ("%d lt Idle \n", time);
    }
}
}

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

```
Task tasks[n];

for(int i=0; i<n; i++){
    tasks[i].pid = i+1;
printf("Enter period and burst
    time for task T%d:", i+1);
scanf("%d %d", &tasks[i].period,
        &tasks[i].burst);
}
printf("Enter simulation time:");
scanf("%d", &sim_time);
rateMonotonic(tasks, n, sim_time);
return 0;
}
```

OUTPUT:

```
Enter number of tasks: 2
Enter period and burst time for task T1: 2 1
Enter period and burst time for task T2: 2 1
Enter simulation time: 3

Rate Monotonic Scheduling:
Time    Task
0       T1
1       T2
2       T1
```

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

Earliest-deadline First

CODE:

```
#include <stdio.h>

#define MAX_TASKS 10
#define MAX_JOBS 1000

typedef struct {
    int id, capacity, deadline, period;
} Task;

typedef struct {
    int task_id, remaining_time, absolute_deadline, release_time, completed;
} Job;

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

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

int lcm_of_periods(Task tasks[], int k) {
    int l = tasks[0].period;
    for (int i = 1; i < k; i++) {
        l = lcm(l, tasks[i].period);
    }
```

```
    return l;
}

int main() {
    int k, hyperperiod;
    Task tasks[MAX_TASKS];
    Job jobs[MAX_JOBS];

    printf("Enter the number of tasks (max %d): ", MAX_TASKS);
    scanf("%d", &k);

    for (int i = 0; i < k; i++) {
        tasks[i].id = i + 1;
        printf("Enter capacity, deadline, and period for task %d: ", tasks[i].id);
        scanf("%d %d %d", &tasks[i].capacity, &tasks[i].deadline, &tasks[i].period);
    }

    hyperperiod = lcm_of_periods(tasks, k);
    printf("\nLCM (Hyperperiod) of periods: %d\n", hyperperiod);

    int job_count = 0;
    for (int i = 0; i < k; i++) {
        for (int release_time = 0; release_time < hyperperiod; release_time += tasks[i].period) {
            jobs[job_count++] = (Job){tasks[i].id, tasks[i].capacity, release_time +
tasks[i].deadline, release_time, 0};
        }
    }

    printf("\nTime : Running Task\n");

    for (int time = 0; time < hyperperiod; time++) {
```

```
        int selected_job_index = -1, earliest_deadline = 1e6;

        for (int j = 0; j < job_count; j++) {
            if (!jobs[j].completed && jobs[j].release_time <= time && jobs[j].remaining_time >
0) {
                if (jobs[j].absolute_deadline < earliest_deadline) {
                    earliest_deadline = jobs[j].absolute_deadline;
                    selected_job_index = j;
                }
            }
        }

        if (selected_job_index == -1) {
            printf("%4d : Idle\n", time);
        } else {
            jobs[selected_job_index].remaining_time--;
            printf("%4d : Task %d\n", time, jobs[selected_job_index].task_id);
            if (jobs[selected_job_index].remaining_time == 0) {
                if (time + 1 > jobs[selected_job_index].absolute_deadline) {
                    printf("      --> Task %d missed its deadline at time %d\n",
jobs[selected_job_index].task_id, time + 1);
                }
                jobs[selected_job_index].completed = 1;
            }
        }
    }

    return 0;
}
```

OUTPUT:

```
LCM (Hyperperiod) of periods: 30

Time : Running Task
   0 : Task 2
   1 : Task 2
   2 : Task 1
   3 : Task 1
   4 : Task 3
   5 : Task 3
   6 : Task 2
   7 : Task 2
   8 : Task 1
   9 : Task 1
  10 : Task 2
  11 : Task 2
  12 : Task 1
  13 : Task 1
  14 : Task 3
  15 : Task 3
  16 : Task 2
  17 : Task 2
  18 : Task 1
  19 : Task 1
  20 : Task 2
  21 : Task 2
  22 : Task 3
  23 : Task 3
  24 : Task 1
  25 : Task 1
  26 : Task 2
  27 : Task 2
  28 : Idle
  29 : Idle
```

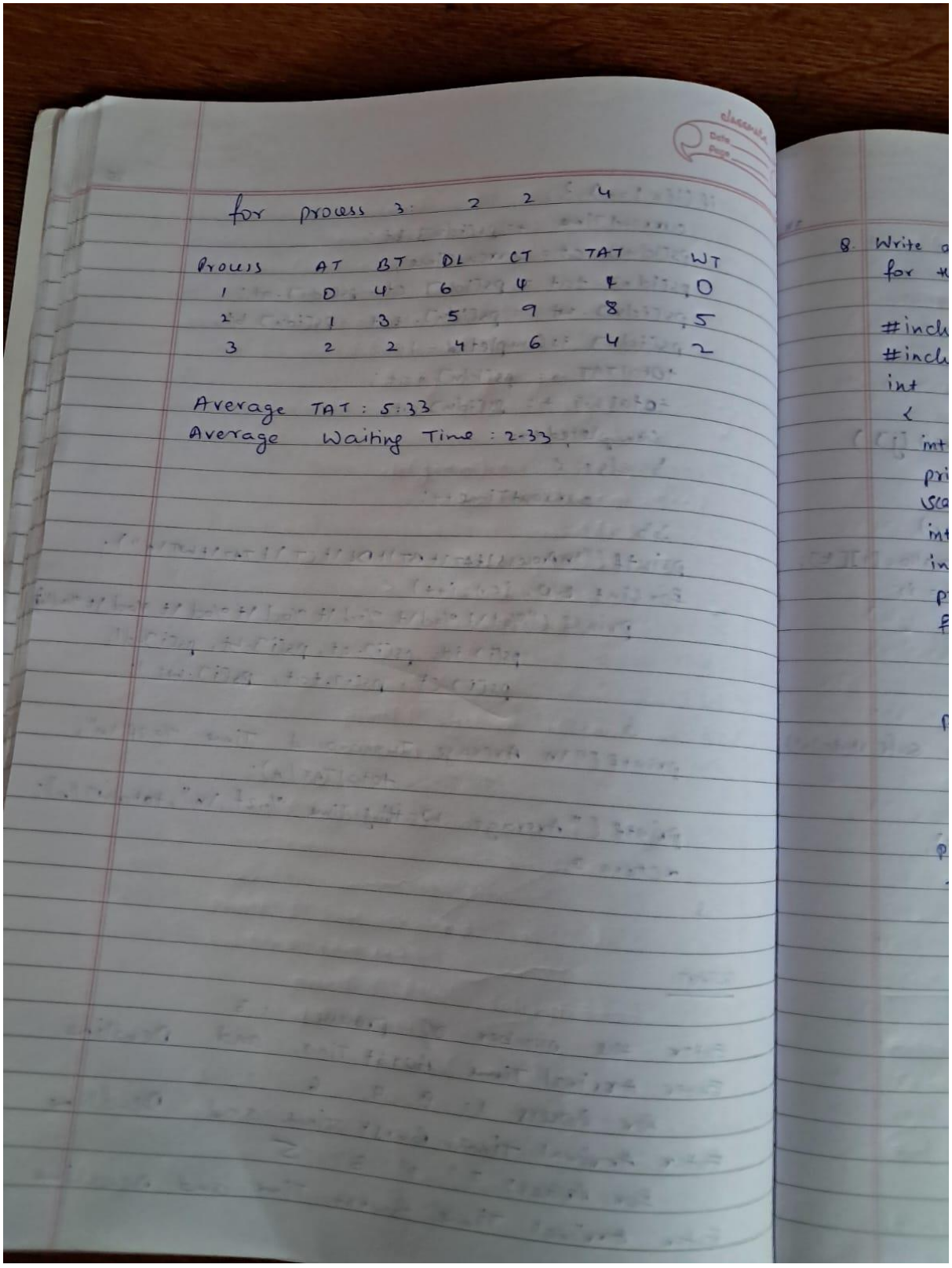
Q. Write a C program to simulate Real-Time CPU scheduling Algorithms: Earliest Deadline First

```
#include <stdio.h>
struct Process {
    int id, at, bt, dl, ct, tat, wt, isCompleted;
};
int main() {
    int n;
    printf("Enter the number of processes : ");
    scanf("%d", &n);
    struct Process ps[n];

    for(int i=0; i<n; i++) {
        ps[i].id = i+1;
        printf("Enter Arrival Time, Burst Time, and
        Deadline for Process %d: ", ps[i].id);
        scanf("%d %d %d", &ps[i].at, &ps[i].bt,
            &ps[i].dl);
        ps[i].isCompleted = 0;
    }

int completed = 0, currentTime = 0;
float totalTAT = 0, totalWT = 0;
while(completed < n) {
    int idx = -1;
    int earliestDeadline = 1eq;

    for(ps[i].isCompleted && ps[i].at <= currentTime
        && ps[i].dl < earliestDeadline) {
        earliestDeadline = ps[i].dl;
        idx = i; }}
```

```
if (idx != -1) {
    currentTime += ps[idx].bt;
    ps[idx].ct = currentTime;
    ps[idx].tat = ps[idx].ct - ps[idx].at;
    ps[idx].wt = ps[idx].tat - ps[idx].bt;
    ps[idx].isCompleted = 1;
    totalTAT += ps[idx].tat;
    totalWT += ps[idx].wt;
    completed++;
} else {
    currentTime++;
}
printf("\nProcess\tAT\tBT\tDL\tCT\tTAT\tWT\n");
for(int i=0; i<n; i++) {
    printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
        ps[i].id, ps[i].at, ps[i].bt, ps[i].dl,
        ps[i].ct, ps[i].tat, ps[i].wt);
}
printf("\n Average Turnaround Time: %.2f\n",
        totalTAT/n);
printf("Average Waiting Time: %.2f\n", totalWT/n);
return 0;
}
```

OUTPUT:

```
Enter the number of processes : 3
Enter Arrival Time, Burst Time, and Deadline
    for Process 1: 0 4 6
Enter Arrival Time, Burst Time and Deadline
    for Process 2: 1 3 5
Enter Arrival Time, Burst Time and Deadline
```



for process 3 2 2 4

Process	AT	BT	DL	CT	TAT	WT
1	0	4	6	4	4	0
2	1	3	5	9	8	5
3	2	2	4	6	4	2

Average TAT : 5.33
Average Waiting Time : 2.33

8. Write a
for

#incl
#incl
int
{
int
pri
sca
int
in
p
f

Write a C program to simulate:

Producer-Consumer problem using semaphores.

CODE:

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



int mutex = 1;
int full = 0;
int empty = 3;
int x = 0;



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

int main() {
   int choice;

   printf("Producer-Consumer Problem using Semaphores (Buffer size = 3)\n");

   while (1) {
      printf("\nMenu:\n");
      printf("1. Produce\n");
      printf("2. Consume\n");
      printf("3. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);
```

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

            case 2:
                if ((mutex == 1) && (full != 0))
                    consumer();
                else
                    printf("Buffer is empty! 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);
   full = signal(full);
   empty = wait(empty);

   x++;
   printf("Producer produced item %d\n", x);

   mutex = signal(mutex);
}


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

   printf("Consumer consumed item %d\n", x);
   x--;

   mutex = signal(mutex);
}
```

OUTPUT:

```
Menu:
1. Produce
2. Consume
3. Exit
Enter your choice: 2
Consumer consumed item 3

Menu:
1. Produce
2. Consume
3. Exit
Enter your choice: 2
Consumer consumed item 2

Menu:
1. Produce
2. Consume
3. Exit
Enter your choice: 2
Consumer consumed item 1

Menu:
1. Produce
2. Consume
3. Exit
Enter your choice: 2
Buffer is empty! Cannot consume.

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

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

Code:

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

int mutex = 1;
int full = 0;
int empty = 3;
int x = 0;

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

int main()
{
    int choice;
    printf ("Producer-Consumer Problem using
        Semaphores (Buffer size = 3) \n");

    while(1)
    {
        printf ("\n Menu: \n");
        printf ("1. Produce \n");
        printf ("2. Consume \n");
        printf ("3. Exit \n");
        printf ("Enter your choice: ");
        scanf ("%d", &choice);
```

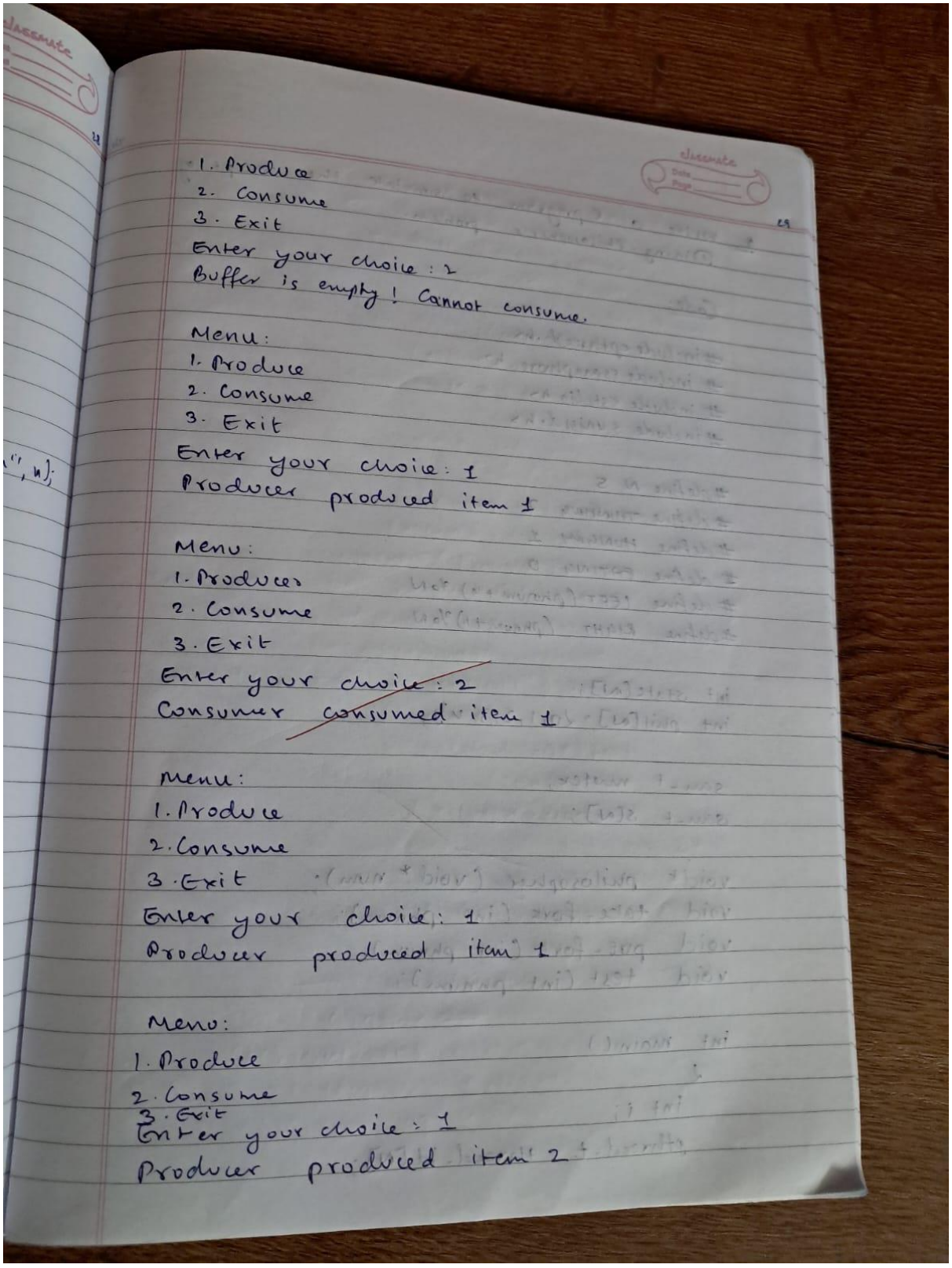
```
switch (choice) {
    case 1:
        if ((mutex == 1) && (empty != 0))
            producer();
        else
            printf("Buffer is full! Cannot
                produce.\n");
        break;
    case 2:
        if((mutex == 1) && (full != 0))
            consumer();
        else
          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

    case 3:
        exit(0);

    default:
        printf("\nInvalid choice. Try again.\n");
    }
}
return 0;
}

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

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

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

      x++;
      printf ("Producer produced item %d \n", x);
      mutex = signal (mutex);
    }

void consumer()
    {
      mutex = wait (mutex);
      full = wait (full);
      empty = signal (empty);
      /
      printf ("Consumer consumed item %d \n", x);
      x--;
      mutex = signal (mutex);
    }
```

OUTPUT:

Producer - Consumer Problem using semaphores (Buffer size = 3)

Menu:



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

Menu:
1. Produce
2. Consume
3. Exit
Enter your choice: 1
Producer produced item 1

Menu:
1. Produce
2. Consume
3. Exit
Enter your choice: 2
Consumer consumed item 1

Menu:
1. Produce
2. Consume
3. Exit
Enter your choice: 1
Producer produced item 1

Menu:
1. Produce
2. Consume
3. Exit
Enter your choice: 1
Producer produced item 2

Write a C program to simulate:

Dining-Philosopher's problem

CODE:

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

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

```
    // initialize the semaphores
    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 that 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 = num;

    while (1) {
        sleep(1);
        take_fork(*i);
        sleep(0);
        put_fork(*i);
    }
}
```

OUTPUT:

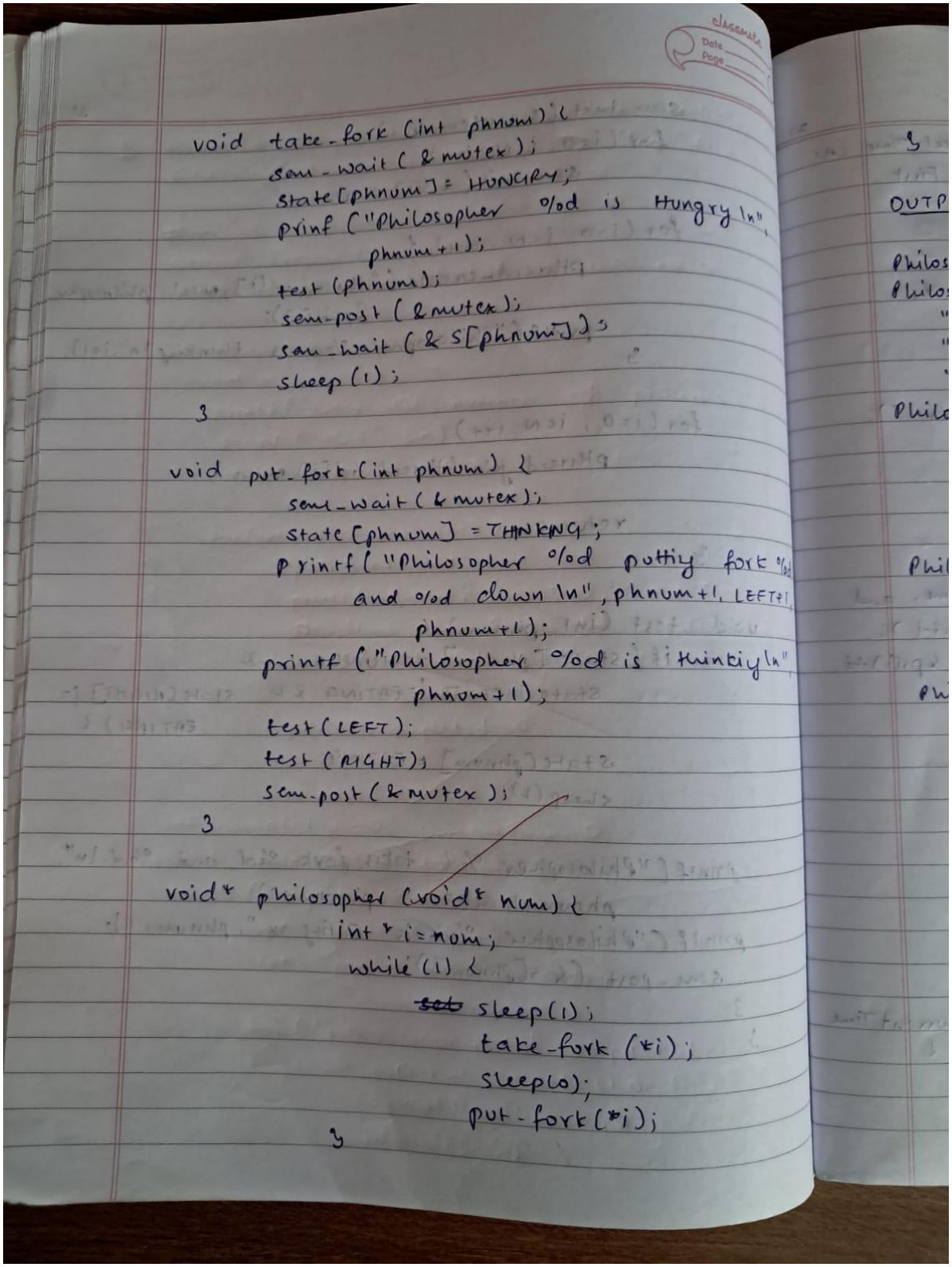
```
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is Hungry
Philosopher 2 is Hungry
Philosopher 5 is Hungry
Philosopher 3 is Hungry
Philosopher 4 is Hungry
Philosopher 4 takes fork 3 and 4
Philosopher 4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

Q. Write a C program to simulate the concept of Dining-Philosopher's problem.

Code:

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

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 (1);

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 %od 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 %od putting fork %od
            and %od down \n", phnum+1, LEFT+1,
                phnum+1);
    printf ("Philosopher %od is thinking \n",
                phnum+1);
    test (LEFT);
    test (RIGHT);
    sem_post (&mutex);
}

void* philosopher (void* num) {
    int *i = num;
    while (1) {
        sleep(1);
        take_fork (*i);
        sleep(0);
        put_fork (*i);
    }
```

9

OUTPUT:

Philosopher 1 is thinking
Philosopher 2 is thinking
" 3 "
" 4 "
" 5 "
Philosopher 1 is hungry
" 2 "
" 5 "
" 3 "
" 4 "
Philosopher 4 takes fork 3 and 4
" is Eating
" putting fork 3 and 4 down
" is thinking
Philosopher 3 takes fork 2 and 3
" is Eating

Write a C program to simulate:

Bankers' algorithm for the purpose of deadlock avoidance.

CODE:

```
#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], max[n][m], avail[m];
    int need[n][m];

    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++)
```

```
    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;
}
```

OUTPUT:

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

Process returned 0 (0x0)   execution time : 43.465 s
Press any key to continue.
```

8. Write a C program to simulate : Banker's algorithm for the purpose of deadlock avoidance.

```
#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], max[n][m], avail[m];
  int need[n][m];
  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++)
      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");
        return 1;
    }
}
printf ("System is in safe state\n");
    return 1;
}
printf ("Safe sequence is : ");
for (i=0; i<n; i++)
        printf ("%d", safeSeq[i]);
        if (i != n-1)
            printf (" -> ");
}
printf ("\n");
return 0;
}
```

Write a C program to simulate:

Deadlock Detection

CODE:

```
#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 allocation[n][m], request[n][m], available[m];
    int work[m];
    bool finish[n];

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

```
}

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

bool found_process;
do {
    found_process = false;
    for (i = 0; i < n; i++) {
        if (!finish[i]) {
            bool can_allocate = true;
            for (j = 0; j < m; j++) {
                if (request[i][j] > work[j]) {
                    can_allocate = false;
                    break;
                }
            }
            if (can_allocate) {
                for (k = 0; k < m; k++)
                    work[k] += allocation[i][k];
                finish[i] = true;
                printf("Process %d can finish.\n", i);
```

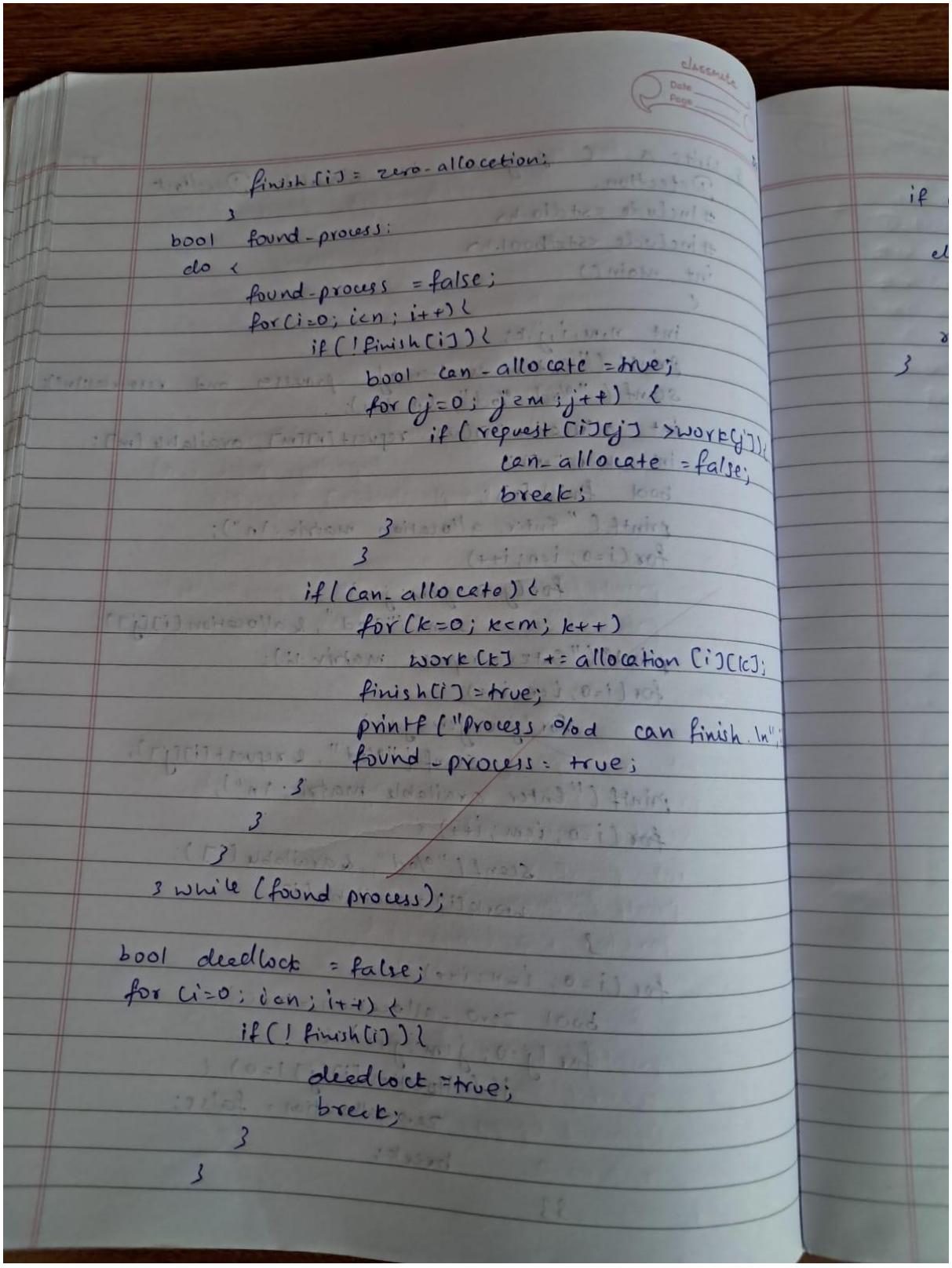
```
                found_process = true;
            }
        }
    }
} while (found_process);

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;
}
```

OUTPUT:

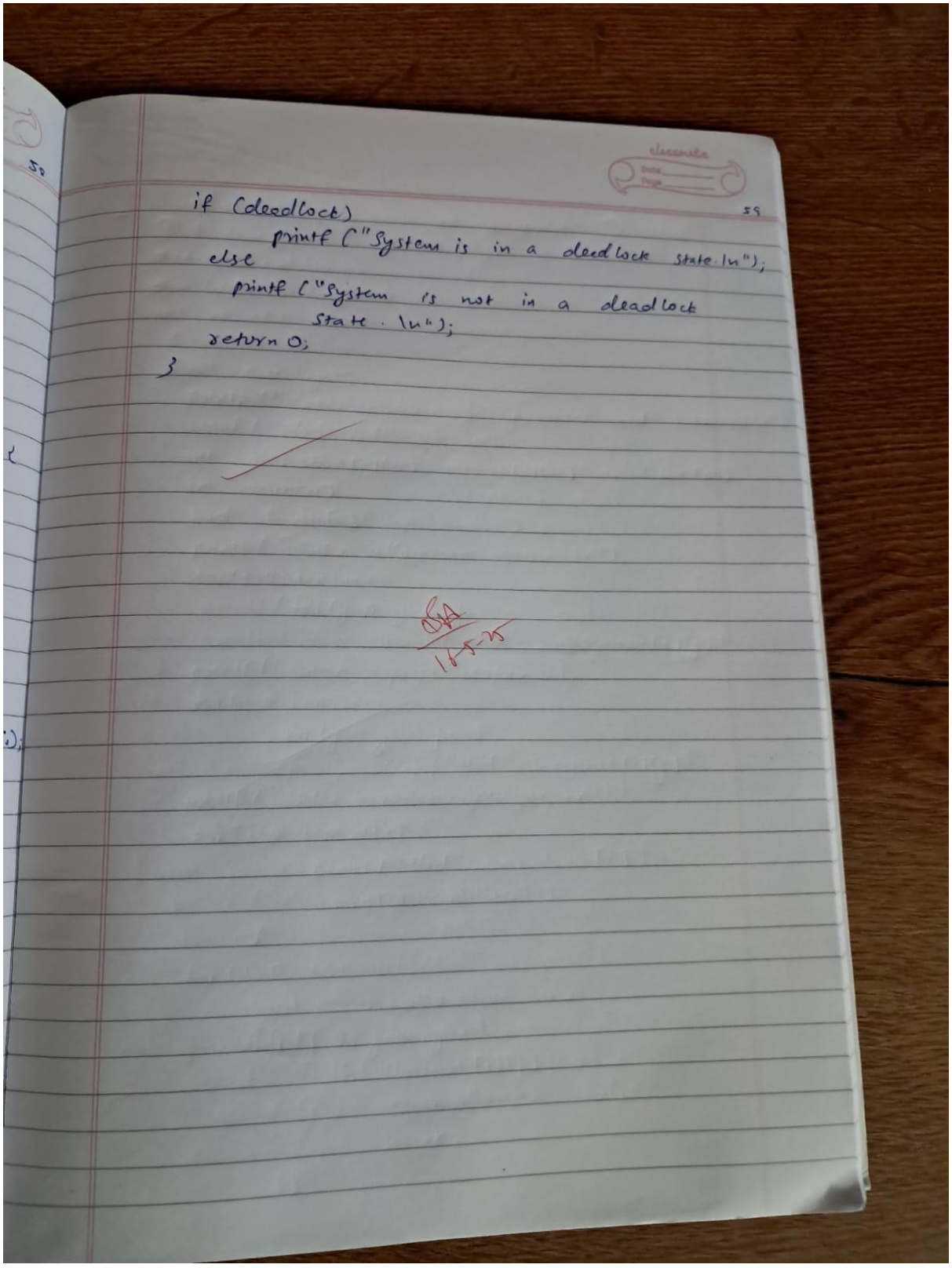
```
Enter number of processes and resources:
5 3
Enter allocation matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter request matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter available matrix:
3 3 2
Process 1 can finish.
Process 3 can finish.
Process 4 can finish.
System is in a deadlock state.

Process returned 0 (0x0)   execution time : 161.813 s
Press any key to continue.
```

Q. Write a C program to simulate Deadlock Detection.

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
    int n,m,i,j,k;
    printf("Enter number of proesses and resources:\n");
    scanf("%od %od", &n, &m);
    int allocation[n][m], request[n][m], available[m];
    int work[m];
    bool finish[n];
    printf("Enter allocation matrix:\n");
    for(i=0; i<n; i++)
        for(j=0; j<m; j++)
            scanf("%od", &allocation[i][j]);
    printf("Enter request matrix:\n");
    for(i=0; i<n; i++)
        for(j=0; j<m; j++)
            scanf("%od", &request[i][j]);
    printf("Enter available matrix:\n");
    for(i=0; i<m; i++) {
        scanf("%od", &available[i]);
        work[i] = available[i];
    }
    for(i=0; i<n; i++) {
        bool zero_allocation = true;
        for(j=0; j<m; j++) {
            if(allocation[i][j] != 0) {
                zero_allocation = false;
                break;
            }
        }
```



```
        finish[i] = zero_allocation;
    }
bool found_process;
do {
    found_process = false;
    for(i=0; i<n; i++) {
        if(!finish[i]) {
            bool can_allocate = true;
            for(j=0; j<m; j++) {
                if(request[i][j] > work[j]) {
                    can_allocate = false;
                    break;
                }
            }
            if(can_allocate) {
                for(k=0; k<m; k++)
                    work[k] += allocation[i][k];
                finish[i] = true;
                printf("Process %d can finish.\n",
                found_process = true;
            }
        }
    }
} while(found_process);

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;
}
```

OSA
16-5-25

Write a C program to simulate the following contiguous memory allocation techniques.
Worst-fit

CODE:

```
#include <stdio.h>

void worstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];


    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }


    for (int i = 0; i < n; i++) {
        int worstIdx = -1;
        for (int j = 0; j < m; j++) {

            if (blockSize[j] >= processSize[i]) {
                if (worstIdx == -1 || blockSize[j] > blockSize[worstIdx]) {
                    worstIdx = j;
                }
            }
        }


        if (worstIdx != -1) {

            allocation[i] = worstIdx;
            blockSize[worstIdx] -= processSize[i];
        }
```

```
    }


    printf("\nProcess No\tProcess Size\tBlock No\tBlock Size\n");
    for (int i = 0; i < n; i++) {
        if (allocation[i] != -1) {
            printf("P%d\t\t%d\t\tB%d\t\t%d\n", i + 1, processSize[i],
                    allocation[i] + 1, blockSize[allocation[i]]);
        } else {
            printf("P%d\t\t%d\t\tNot Allocated\t\t-\n", i + 1, processSize[i]);
        }
    }
}

int main() {
    int m, n;


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

    int blockSize[m];

    printf("Enter the size of each block:\n");
    for (int i = 0; i < m; i++) {
        printf("Block %d size: ", i + 1);
        scanf("%d", &blockSize[i]);
    }
```

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

    int processSize[n];

    printf("Enter the size of each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d size: ", i + 1);
        scanf("%d", &processSize[i]);
    }


    worstFit(blockSize, m, processSize, n);

    return 0;
}
```

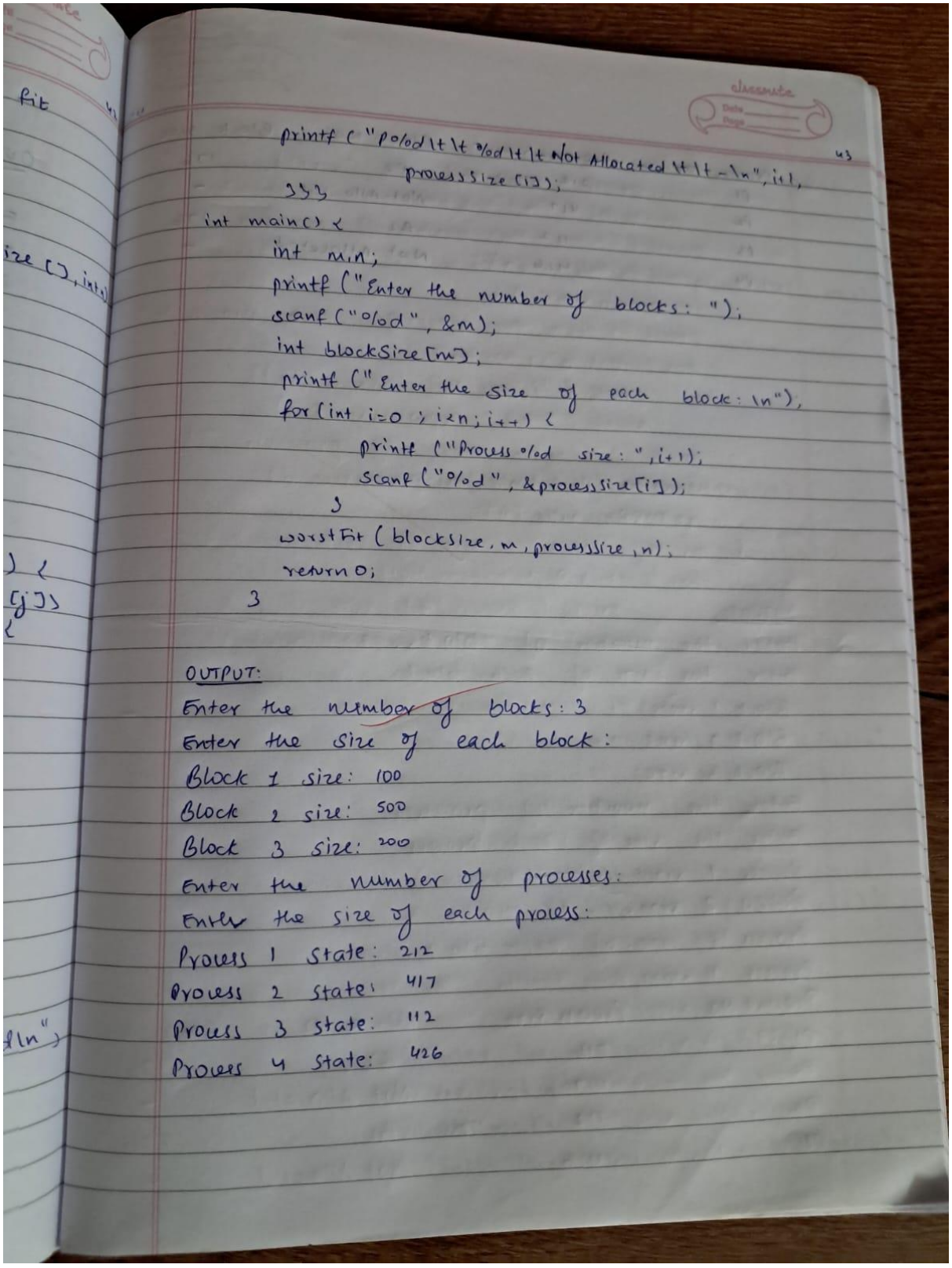
OUTPUT:

```
Enter the number of blocks: 3
Enter the size of each block:
Block 1 size: 100
Block 2 size: 500
Block 3 size: 200
Enter the number of processes: 4
Enter the size of each process:
Process 1 size: 212
Process 2 size: 417
Process 3 size: 112
Process 4 size: 426

Process No      Process Size    Block No        Block Size
P1              212             B2              176
P2              417             Not Allocated           -
P3              112             B2              176
P4              426             Not Allocated           -
```

9. Write a C program to simulate the Worst fit contiguous memory allocation techniques.

Worst fit

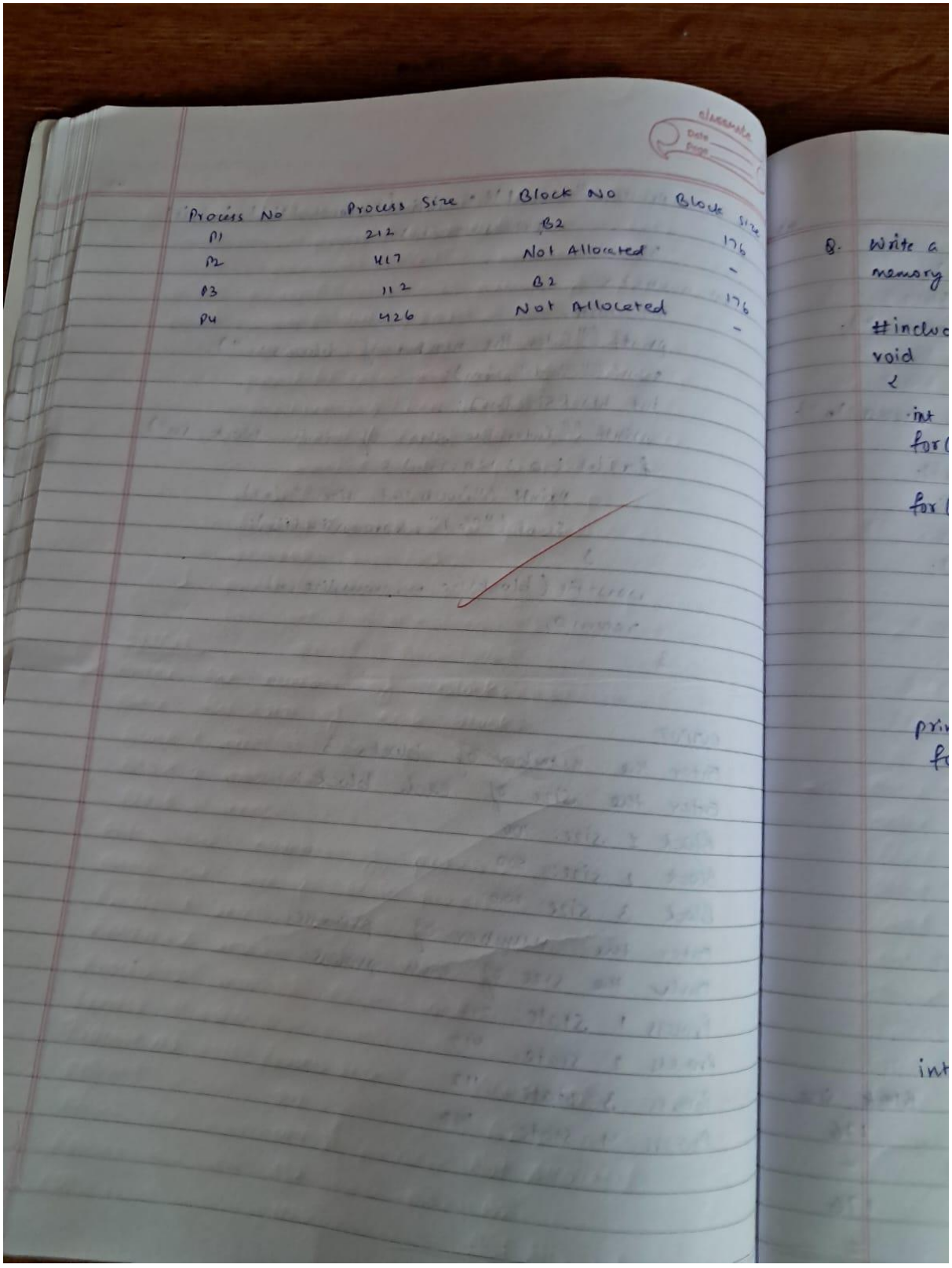
```
#include <stdio.h>
void worstFit(int blocksize[], int m, int processSize[],
{
    int allocation[n];
    for(int i=0; i<n; i++) {
        allocation[i] = -1;
    }
    for(int i=0; i<n; i++) {
        int worstIdx = -1
        for(int j=0; j<m; j++) {
            if(blocksize[j] >= processSize[i]) {
                if(worstIdx == -1 || blockSize[j]
                    blockSize[worstIdx]) {
                    worstIdx = j;
                }
            }
        }
    if(worstIdx != -1) {
        allocation[i] = worstIdx;
        blocksize[worstIdx] -= processSize[i];
    }
printf("\n Process No \t Process Size \t Block No \t
        Block size \n");
for(int i=0; i<n; i++) {
    if( allocation[i] != -1) {
        printf("%d\t\t %d \t\t %d \t\t %d\n
            i+1, processSize[i], allocation[i]+1,
            blocksize[allocation[i]]);
    } else {
```



```
        printf("%d\t\t%d\t\tNot Allocated\t\t-\n", i+1,
                processSize[i]);
    }
}
int main() {
    int m;
    printf("Enter the number of blocks: ");
    scanf("%d", &m);
    int blockSize[m];
    printf("Enter the size of each block:\n");
    for(int i=0; i<m; i++) {
        printf("Process %d size: ", i+1);
        scanf("%d", &processSize[i]);
    }
    worstFit(blockSize, m, processSize, n);
    return 0;
}
```

OUTPUT:

```
Enter the number of blocks: 3
Enter the size of each block:
Block 1 size: 100
Block 2 size: 500
Block 3 size: 200
Enter the number of processes:
Enter the size of each process:
Process 1 state: 212
Process 2 state: 417
Process 3 state: 112
Process 4 state: 426
```



Process No	Process Size	Block No	Block Size
P1	212	B2	176
P2	417	Not Allocated	-
P3	112	B2	176
P4	426	Not Allocated	-

Q. Write a
memory

#inclu
void
{
int
for(
for(

pri
fo

int

Write a C program to simulate the following contiguous memory allocation techniques.
First-fit

CODE:

```
#include <stdio.h>


void firstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];


    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }


    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                allocation[i] = j;
                blockSize[j] -= processSize[i];
                break;
            }
        }
    }


    printf("\nProcess No\tProcess Size\tBlock No\tBlock Size\n");
    for (int i = 0; i < n; i++) {
        if (allocation[i] != -1) {
            printf("P%d\t\t%d\t\tB%d\t\t%d\n", i + 1, processSize[i],
                    allocation[i] + 1, blockSize[allocation[i]]);
        } else {
            printf("P%d\t\t%d\t\tNot Allocated\t\t-\n", i + 1, processSize[i]);
        }
    }
```

```
}

int main() {
    int m, n;

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

    int blockSize[m];
    printf("Enter the size of each block:\n");
    for (int i = 0; i < m; i++) {
        printf("Block %d size: ", i + 1);
        scanf("%d", &blockSize[i]);
    }

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

    int processSize[n];
    printf("Enter the size of each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d size: ", i + 1);
        scanf("%d", &processSize[i]);
    }

    firstFit(blockSize, m, processSize, n);

    return 0;
}
```

OUTPUT:

```
Enter the number of blocks: 3
Enter the size of each block:
Block 1 size: 100
Block 2 size: 500
Block 3 size: 200
Enter the number of processes: 4
Enter the size of each process:
Process 1 size: 212
Process 2 size: 417
Process 3 size: 112
Process 4 size: 426

Process No      Process Size    Block No        Block Size
P1              212             B2              176
P2              417             Not Allocated           -
P3              112             B2              176
P4              426             Not Allocated           -
```

8. Write a C program to simulate the First fit contiguous memory allocation techniques.

```
#include<stdio.h>
void firstFit(int blocksize[], int m, int processSize[], int n)
{
    int allocation[n];
    for(int i=0; i<n; i++) {
        allocation[i] = -1; }
    for(int i=0; i<n; i++) {
        for(int j=0; j<m; j++) {
            if( blocksize[j] >= processSize[i]) {
                allocation[i] = j;
                blocksize[j] -= processSize[i];
                break;
            }}}
    printf("\n Process No \t Process size \t Block No \t Block Size \n");
    for(int i=0; i<n; i++) {
        if(allocation[i] != -1) {
            printf("%d \t\t %d \t\t %d \t\t %d \n",
                i+1, processSize[i], allocation[i]+1,
                blocksize[allocation[i]]);
        } else {
            printf("%d \t\t %d \t\t Not Allocated
                \t\t - \n", i+1, processSize[i]);
        }}}
int main() {
    int m,n;
    printf("Enter the number of blocks: ");
    scanf("%d", &m);
    int blocksize[m];
    printf("Enter the size of each block: \n");
```

```
for(int i=0; i<m; i++) {
    printf("Block %d size: ", i+1);
    scanf("%d", &blocksize[i]); }
printf("Enter the number of processes: ");
scanf("%d", &n);
int processSize[n];
printf("Enter the size of each process: ");
for(int i=0; i<n; i++) {
    printf("Process %d size: ", i+1);
    scanf("%d", &processSize[i]);
}
firstFit(blocksize, m, processSize, n);
return 0;
}
```

OUTPUT:

```
Enter the number of blocks: 3
Enter the size of each block:
Block 1 size:
Block 2 size:
Block 3 size:
Enter the number of processes:
Enter the size of each process:
Process 1 size:
Process 2 size:
Process 3 size:
Process 4 size:
```

Process NO	Process Size	Block NO	Block Size
P1	212	B2	176
P2	417	Not Allocated	-
P3	112	B2	176
P4	426	Not Allocated	-

Write a C program to simulate the following contiguous memory allocation techniques.
Best-fit

CODE:

```
#include <stdio.h>


void bestFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];

    for (int i = 0; i < n; i++) {
        allocation[i] = -1;
    }

    for (int i = 0; i < n; i++) {
        int bestIdx = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (bestIdx == -1 || blockSize[j] < blockSize[bestIdx]) {
                    bestIdx = j;
                }
            }
        }

        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            blockSize[bestIdx] -= processSize[i];
        }
    }

    printf("\nProcess No\tProcess Size\tBlock No\tBlock Size\n");
    for (int i = 0; i < n; i++) {
        if (allocation[i] != -1) {
```

```
            printf("P%d\t\t%d\t\tB%d\t\t%d\n", i + 1, processSize[i],
                    allocation[i] + 1, blockSize[allocation[i]]);
        } else {
            printf("P%d\t\t%d\t\tNot Allocated\t\t-\n", i + 1, processSize[i]);
        }
    }
}

int main() {
    int m, n;

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

    int blockSize[m];
    printf("Enter the size of each block:\n");
    for (int i = 0; i < m; i++) {
        printf("Block %d size: ", i + 1);
        scanf("%d", &blockSize[i]);
    }

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

    int processSize[n];
    printf("Enter the size of each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d size: ", i + 1);
        scanf("%d", &processSize[i]);
    }
```

```
    bestFit(blockSize, m, processSize, n);

    return 0;
}
```

OUTPUT:

```
Enter the number of blocks: 3
Enter the size of each block:
Block 1 size: 100
Block 2 size: 500
Block 3 size: 200
Enter the number of processes: 4
Enter the size of each process:
Process 1 size: 212
Process 2 size: 417
Process 3 size: 112
Process 4 size: 426

Process No      Process Size    Block No        Block Size
P1              212             B2              288
P2              417             Not Allocated           -
P3              112             B3              88
P4              426             Not Allocated           -
```

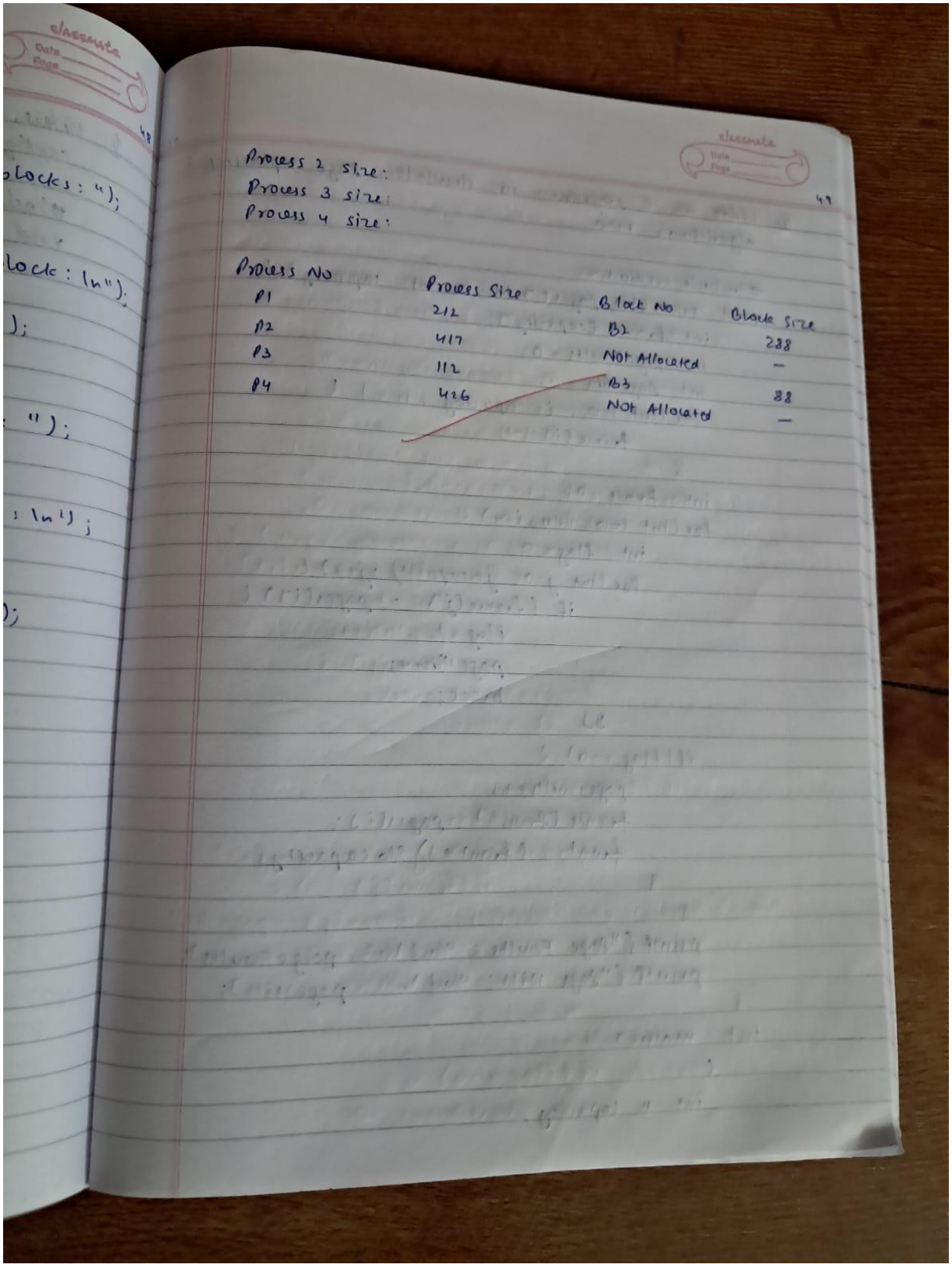
Q. Write a C program to simulate the Best Fit contiguous memory allocation techniques.

```
#include <stdio.h>
void bestFit(int blocksize[], int m, int processsize[], int n){
    int allocation[n];
    for(int i=0; i<n; i++){
        allocation[i] = -1; }
    for(int i=0; i<n; i++){
        int bestIdx = -1;
        for(int j=0; j<m; j++){
            if(blocksize[j] >= processsize[j]){
                if(bestIdx == -1 || blocksize[j] <
                    blocksize[bestIdx]){
                    bestIdx = j;
                }}}
    if(bestIdx != -1){
        allocation[i] = bestIdx;
        blocksize[bestIdx] -= processSize[i];
    }
}
printf("\n Process No \t Process Size \t Block No \t
        Block size \n");
for(int i=0; i<n; i++){
    if(allocation[i] != -1){
        printf("%d \t\t %d \t\t %d \t\t %d\n",
        i+1, processSize[i], allocation[i]+1,
        blocksize[allocation[i]]);
    } else {
        printf("%d \t\t %d \t\t Not Allocated
            \t\t -\n", i+1, processSize[i]);
}}}
```

```
int main() {
    int m,n;
    printf("Enter the number of blocks: \n");
    scanf("%d", &m);
    int blocksize[m];
    printf("Enter the size of each block: \n");
    for(int i=0; i<m; i++) {
        printf("Block %d size: ", i+1);
        scanf("%d", &blocksize[i]);
    }
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int processsize[n];
    printf("Enter the size of each process: \n");
    for(int i=0; i<n; i++) {
        printf("Process %d size: ", i+1);
        scanf("%d", &processsize[i]);
    }
    bestFit(blocksize, m, processsize, n);
    return 0;
}
```

OUTPUT:

```
Enter the number of blocks: 3
Enter the size of each block:
Block 1 size:
Block 2 size:
Block 3 size:
Enter the number of processes:
Enter the size of each process:
Process 1 size:
```



Process 2 size:
Process 3 size:
Process 4 size:

Process No	Process Size	Block No	Block Size
P1	212	B2	288
P2	417	Not Allocated	—
P3	112	B3	88
P4	426	Not Allocated	—

Write a C program to simulate page replacement algorithms.

a) FIFO

CODE:

```
#include <stdio.h>

void FIFO(int pages[], int n, int capacity) {
    int frame[capacity];
    int pageFaults = 0;
    int pageHits = 0;

    for (int i = 0; i < capacity; i++) {
        frame[i] = -1;
    }

    int front = 0;

    for (int i = 0; i < n; i++) {
        int flag = 0;
        for (int j = 0; j < capacity; j++) {
            if (frame[j] == pages[i]) {
                flag = 1;
                pageHits++;
                break;
            }
        }

        if (flag == 0) {
            pageFaults++;
            frame[front] = pages[i];
            front = (front + 1) % capacity;
```

```
        }
    }

    printf("Page Faults: %d\n", pageFaults);
    printf("Page Hits: %d\n", pageHits);
}

int main() {
    int n, capacity;

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

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

    printf("Enter frame capacity: ");
    scanf("%d", &capacity);

    FIFO(pages, n, capacity);

    return 0;
}
```

OUTPUT:

```
Enter number of pages: 7
Enter pages:
7 0 1 2 0 3 0
Enter frame capacity: 3
Page Faults: 6
Page Hits: 1
```

Q. Write a C program to simulate page replacement algorithms. FIFO

```
#include <stdio.h>
void FIFO(int pages[], int n, int capacity) {
    int frame[capacity];
    int pageFaults = 0;
    int pageHits = 0;
    for(int i=0; i<capacity; i++) {
        frame[i] = -1;
    }
    int front = 0;
    for(int i=0; i<n; i++) {
        int flag = 0;
        for(int j=0; j<capacity; j++) {
            if(frame[j] == pages[i]) {
                flag = 1;
                pageHits++;
                breet;
            }
        }
        if(flag == 0) {
            pageFaults++;
            frame[front] = pages[i];
            front = (front + 1) % capacity;
        }
    }
    printf("Page Faults : %d \n", pageFaults);
    printf("Page Hits : %d \n", pageHits);
}
int main()
{
    int n, capacity;
```

```
printf("Enter number of pages: ");
scanf("%d", &n);
int pages[n];
printf("Enter pages: \n");
for(int i=0; i<n; i++) {
    scanf("%d", &pages[i]);
}
printf("Enter frame capacity: ");
scanf("%d", &capacity);
FIFO(pages, n, capacity);
return 0;
}
```

OUTPUT:

```
Enter number of pages: 7
Enter pages:
7 0 1 2 0 3 0
Enter frame capacity: 3
Page Faults: 6
Page Hits: 1
```

Write a C program to simulate page replacement algorithms.

LRU

CODE:

```
#include <stdio.h>

void LRU(int pages[], int n, int capacity) {
    int frame[capacity], time[capacity], pageFaults = 0, pageHits = 0;

    for (int i = 0; i < capacity; i++) {
        frame[i] = -1;
        time[i] = -1;
    }

    for (int i = 0; i < n; i++) {
        int flag = 0, leastRecent = 0;
        for (int j = 0; j < capacity; j++) {
            if (frame[j] == pages[i]) {
                flag = 1;
                pageHits++;
                time[j] = i;
                break;
            }
        }

        if (flag == 0) {
            pageFaults++;
            for (int j = 0; j < capacity; j++) {
                if (frame[j] == -1) {
                    frame[j] = pages[i];
                    time[j] = i;
```

```
                break;
            }
        }
        if (flag == 0) {
            leastRecent = 0;
            for (int j = 1; j < capacity; j++) {
                if (time[j] < time[leastRecent]) {
                    leastRecent = j;
                }
            }
            frame[leastRecent] = pages[i];
            time[leastRecent] = i;
        }
    }
}

    printf("Page Faults: %d\n", pageFaults);
    printf("Page Hits: %d\n", pageHits);
}

int main() {
    int n, capacity;

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

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

```
    }

    printf("Enter frame capacity: ");
    scanf("%d", &capacity);

    LRU(pages, n, capacity);

    return 0;
}
```

OUTPUT:

```
Enter number of pages: 7
Enter pages:
7 0 1 2 0 3 0
Enter frame capacity: 3
Page Faults: 5
Page Hits: 2
```

9. Write a C program to simulate page replacement algorithms. LRU.

```
#include <stdio.h>
void LRU (int pages[], int n, int capacity) {
    int frame[capacity], time[capacity], pageFaults, pageHits;
    pageHits = 0;
    for (int i=0; i<capacity; i++) {
        frame[i] = -1;
        time[i] = -1;
    }
    for (int i=0; i<n; i++) {
        int flag=0, leastRecent=0;
        for (int j=0; j<capacity; j++) {
            if (frame[j] == pages[i]) {
                flag = 1;
                pageHits++;
                time[j] = i;
                break;
            }
        }
        if (flag==0) {
            pageFaults++;
            for (int j=0; j<capacity; j++) {
                if (frame[j] == -1) {
                    frame[j] = pages[i];
                    time[j] = i;
                    break;
                }
            }
        }
        if (flag==0) {
            leastRecent=0;
```

```
for (int j=1; j<capacity; j++){
        if(time[j] < time[leastRecent]) {
            leastRecent = j;
        }
    }
    frame[leastRecent] = pages[i];
    time[leastRecent] = i;
}}}
printf("Page Faults: %d\n", pageFaults);
printf("Page Hits: %d\n", pageHits);
}
int main()
{
    int n, capacity;
    printf("Enter number of pages:");
    scanf("%d", &n);
    int pages[n];
    printf("Enter pages:\n");
    for(int i=0; i<n; i++){
        scanf("%d", &pages[i]);
    }
    printf("Enter frame capacity: ");
    scanf("%d", &capacity);
    LRU(pages, n, capacity);
    return 0;
}
```

OUTPUT:

```
Enter number of pages: 7
Enter pages:
7 0 1 2 0 3 0
Enter frame capacity: 3
Page Faults: 5
Page Hits: 2
```

Write a C program to simulate page replacement algorithms.

Optimal

CODE:

```
#include <stdio.h>

int findOptimal(int frame[], int capacity, int pages[], int current, int n) {
    int farthest = -1, index = -1;
    for (int i = 0; i < capacity; i++) {
        int j;
        for (j = current + 1; j < n; j++) {
            if (frame[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    index = i;
                }
                break;
            }
        }
        if (j == n) {
            return i;
        }
    }
    return index;
}

void Optimal(int pages[], int n, int capacity) {
    int frame[capacity], pageFaults = 0, pageHits = 0;

    for (int i = 0; i < capacity; i++) {
        frame[i] = -1;
```

```
    }

    for (int i = 0; i < n; i++) {
        int flag = 0;
        for (int j = 0; j < capacity; j++) {
            if (frame[j] == pages[i]) {
                flag = 1;
                pageHits++;
                break;
            }
        }

        if (flag == 0) {
            pageFaults++;
            int replaceIndex = findOptimal(frame, capacity, pages, i, n);
            frame[replaceIndex] = pages[i];
        }
    }

    printf("Page Faults: %d\n", pageFaults);
    printf("Page Hits: %d\n", pageHits);
}

int main() {
    int n, capacity;

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

    int pages[n];
```

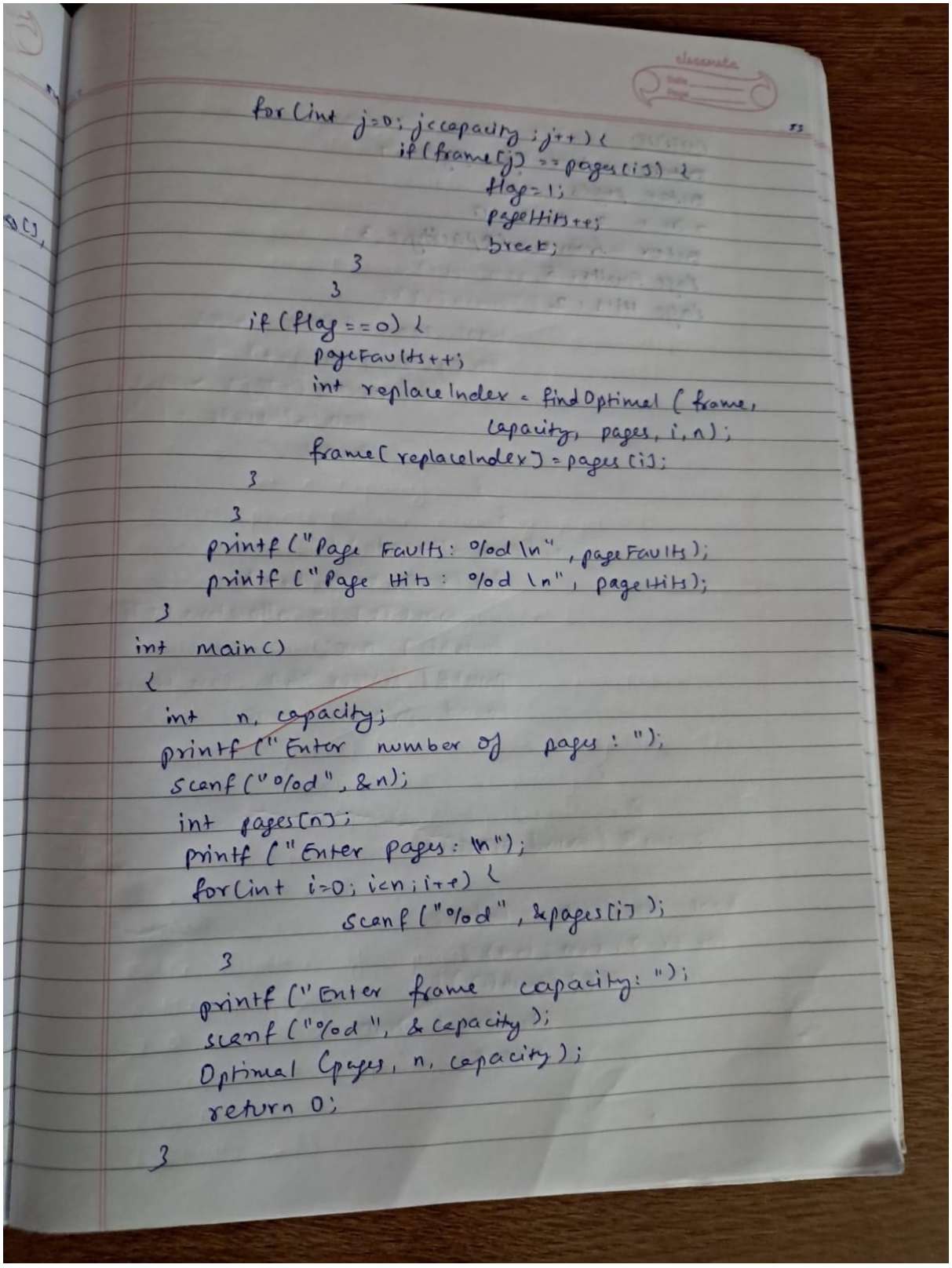
```
    printf("Enter pages:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &pages[i]);
    }

    printf("Enter frame capacity: ");
    scanf("%d", &capacity);

    Optimal(pages, n, capacity);

    return 0;
}
```

OUTPUT:

```
Enter number of pages: 7
Enter pages:
7 0 1 2 0 3 0
Enter frame capacity: 3
Page Faults: 5
Page Hits: 2
```

9. Write a C program to simulate page replacement algorithm. Optimal

```
#include <stdio.h>
int findOptimal(int frame[], int capacity, int pages[],
  int current, int n) {
    int farthest = -1, index = -1;
    for(int i=0; i<capacity; i++) {
      int j;
        for(j= current+1; j<n; j++) {
            if (frame[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    index = i;
                }
                break;
            }
        }
        if (j==n) {
            return i;
        }
    }
    return index;
}

void Optimal(int pages[], int n, int capacity){
    int frame[capacity], pageFaults = 0, pageHits = 0;
    for(int i=0; i<capacity; i++) {
        frame[i] = -1;
    }
    for(int i=0; i<n; i++) {
        int flag=0;
```



```
for(int j=0; j<capacity; j++) {
        if(frame[j] == pages[i]) {
            flag=1;
            pageHits++;
            break;
        }
    }
    if(flag==0) {
        pageFaults++;
        int replaceIndex = findOptimal(frame,
                capacity, pages, i, n);
        frame[replaceIndex] = pages[i];
    }
}
printf("Page Faults: %d\n", pageFaults);
printf("Page Hits: %d\n", pageHits);
}
int main()
{
int n, capacity;
printf("Enter number of pages: ");
scanf("%d", &n);
int pages[n];
printf("Enter pages:\n");
for(int i=0; i<n; i++) {
        scanf("%d", &pages[i]);
}
printf("Enter frame capacity: ");
scanf("%d", &capacity);
Optimal(pages, n, capacity);
return 0;
}
```