VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

OPERATING SYSTEMS

Submitted by

THILAK K (1WA23CS021)

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 THILAK K (1WA23CS021), 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.

Madhavi R.P.
Associate 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	111-122

Course Outcomes

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

Program -1

Question:

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

→FCFS

→ SJF (pre-emptive & Non-preemptive)

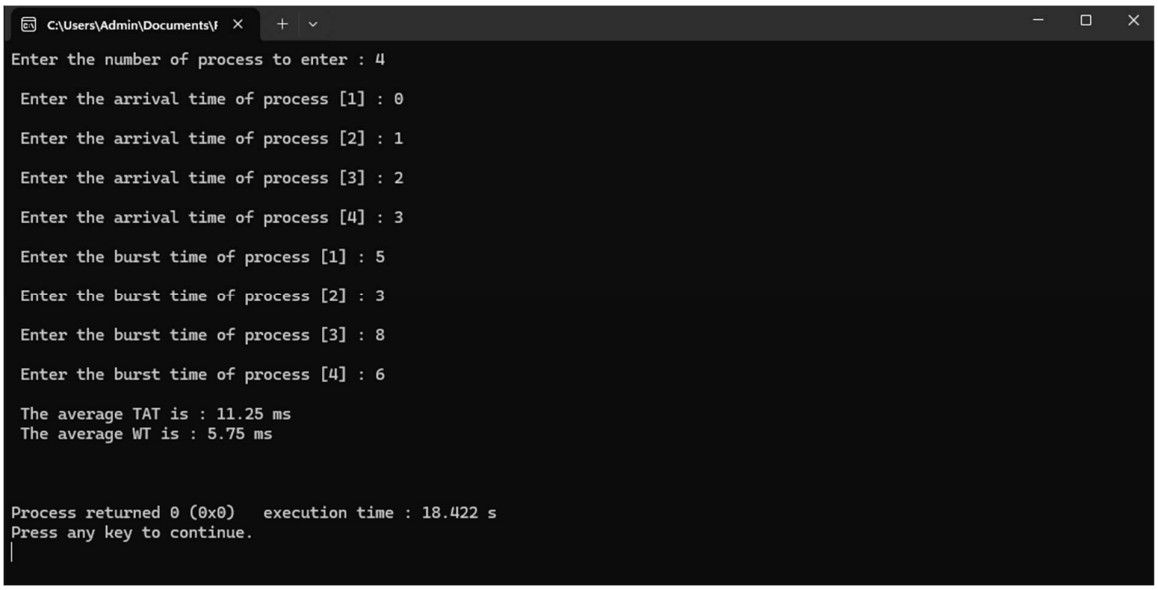
Code:

=>FCFS:

```
#include<stdio.h>
#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 the number of process to enter : ");
   scanf("%d",&n);

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

   for(int i=0;i<n;i++)
   {
      printf("\n 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 The average TAT is : %.2f ms",(float)totalTAT/n);
    printf("\n The average WT is : %.2f ms\n\n\n",(float)totalWT/n);

    return 0;
}
```

Result:


C:\Users\Admin\Documents\F
Enter the number of process to enter : 4
Enter the arrival time of process [1] : 0
Enter the arrival time of process [2] : 1
Enter the arrival time of process [3] : 2
Enter the arrival time of process [4] : 3
Enter the burst time of process [1] : 5
Enter the burst time of process [2] : 3
Enter the burst time of process [3] : 8
Enter the burst time of process [4] : 6
The average TAT is : 11.25 ms
The average WT is : 5.75 ms
Process returned 0 (0x0) execution time : 18.422 s
Press any key to continue.

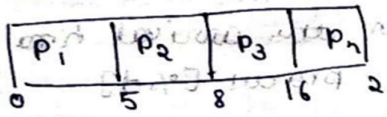
OBSERVATION:

LAB – Program - 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.



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

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

$WT = TAT - BT$

$TAT = CT - AT$

```
#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 no of process's : ");

    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 burst time : ");

      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 : %.2f ", (float) totalTAT/n);
printf("\n Average WT : %.2f ", (float) totalWT/n);
```

=>SJF(Non-preemptive):

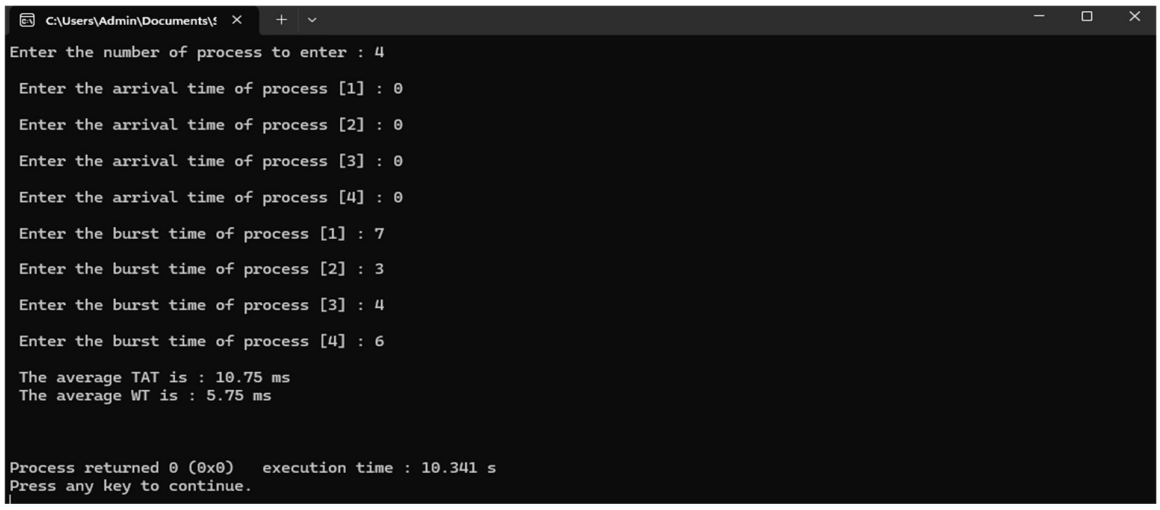
```
#include<stdio.h>
#include<stdbool.h>

struct process
{
   int at;
   int bt;
   int ct;
   int tat;
   int wt;
   int start_time;
} ps[100];

int main()
{
   float totalTAT = 0, totalWT = 0;
   int n;
   int completed = 0;
   bool is_visited[100] = {false};
   int current_time = 0;
   int minimum = 9999999999;
   int min_index = -1;

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

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

```
{
   printf("\n Enter the arrival time of process [%d] : ", i + 1);
   scanf("%d", &ps[i].at);
}

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

while(completed != n)
{
   minimum = 2147483647;
   min_index = -1;

   for(int i = 0; i < n; i++)
   {
      if(ps[i].at <= current_time && !is_visited[i])
      {
         if(ps[i].bt < minimum)
         {
            minimum = ps[i].bt;
            min_index = i;
         }

         if(ps[i].bt == minimum)
         {
```

```
            if(ps[i].at < ps[min_index].at)
            {
               minimum = ps[i].bt;
               min_index = i;
            }
         }
      }
   }

   if(min_index == -1)
   {
      current_time++;
   }
   else
   {
      ps[min_index].start_time = current_time;
      ps[min_index].ct = ps[min_index].start_time + ps[min_index].bt;
      ps[min_index].tat = ps[min_index].ct - ps[min_index].at;
      totalTAT += ps[min_index].tat;
      ps[min_index].wt = ps[min_index].tat - ps[min_index].bt;
      totalWT += ps[min_index].wt;
      is_visited[min_index] = true;
      completed++;
      current_time = ps[min_index].ct;
   }
}

printf("\n The average TAT is : %.2f ms", totalTAT / n);
```

```
    printf("\n The average WT is : %.2f ms\n\n\n", totalWT / n);

}
```

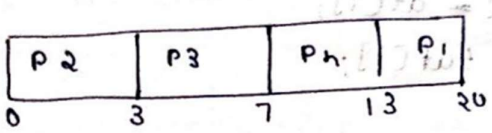
Result:



OBSERVATION:

SJF

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



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

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

6/3/25

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

struct process
{
    int at;
    int bt;
    int ct;
    int tat;
    int wt;
    int start_time;
} ps[100];

int main()
{
    float totalWAT = 0, totalWT = 0;
    int n;
    int completed = 0;
    bool isVisited[100] = {false};
    int current_time = 0;
    int minimum = 999999999;
    int min_indx = -1;

    printf("Enter no of process :");
    scanf("%d", &n);

    for(int i=0; i<n; i++)
    {
        scanf("%d", &ps[i].at);
    }
```

```
for (int i=0; i<n; i++)
{
    printf(" Enter burst ");
    scanf(" %d", &ps[i].bt);
}

while (completed != n);
{
    minimum = 9999999;
    min_index = -1;

    for (int i=0 ; i<n; i++)
    {
        if ( ps[i].at <= current_time &&
                        !is_visited[i])
        {
            if (ps[i].bt < minimum)
            {
                minimum = ps[i].bt;
                min_index = i;
            }

            if (ps[i].bt == minimum)
            {
                if (ps[i].at < ps[min_index].at)
                {
                    minimum = ps[i].bt;
                    min_index = i;
                }
            }
        }
```

```
if (min_index == -1)
{
    current_time++;
}
else
{
    ps[min_index].start_time = current_time;
    ps[min_index].ct = ps[min_index].start_time
                       + ps[min_index].bt;

    ps[min_index].tat = ps[min_index].ct
                        - ps[min_index].at;

    totalTAT += ps[min_index].tat;

    ps[min_index].wt = ps[min_index].tat
                       - ps[min_index].bt;

    totalWT += ps[min_index].wt;
    is_visited[min_index] = true;
    completed++;
    current_time = ps[min_index].ct;
}
}

printf("Average TAT is: %.2f", totalTAT/n);
printf("Average WT is: %.2f", totalWT/n);
}
```

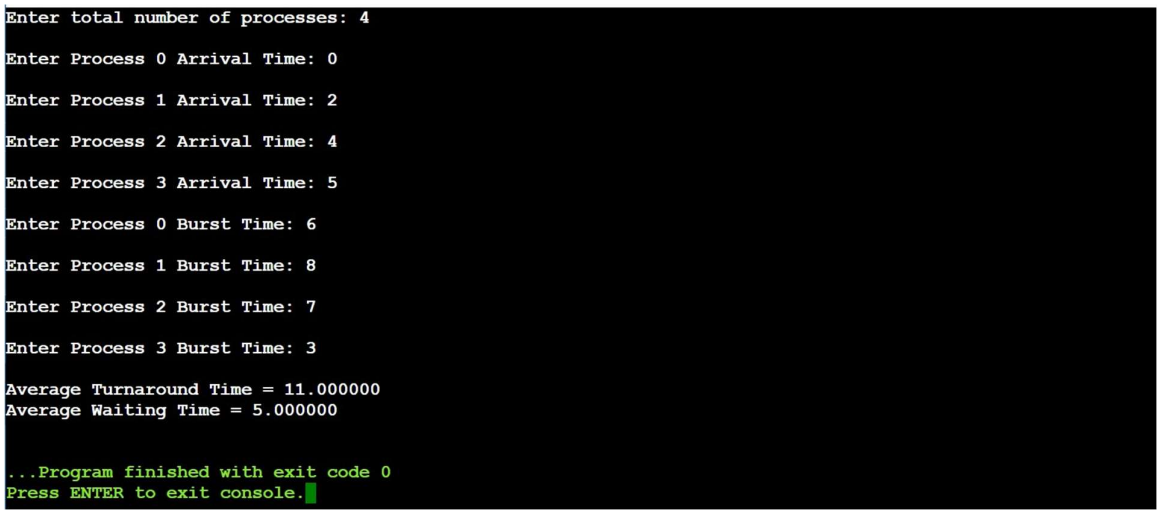
=>SJF(Pre-Emptive[SRTF]):

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

struct process_struct {
  int pid, at, bt, ct, wt, tat, start_time;
} ps[100];

int main() {
    int n;
    float bt_remaining[100];
    bool is_completed[100] = {false};
    int current_time = 0, completed = 0;
    float sum_tat = 0, sum_wt = 0;
    int prev = 0;

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

    for (int i = 0; i < n; i++) {
        printf("\nEnter Process %d Arrival Time: ", i);
        scanf("%d", &ps[i].at);
        ps[i].pid = i;
    }

    for (int i = 0; i < n; i++) {
        printf("\nEnter Process %d Burst Time: ", i);
```

```
        scanf("%d", &ps[i].bt);
        bt_remaining[i] = ps[i].bt;
    }

    while (completed != n) {
        int min_index = -1, minimum = INT_MAX;
        for (int i = 0; i < n; i++) {
            if (ps[i].at <= current_time && !is_completed[i] && bt_remaining[i] <
minimum) {
                minimum = bt_remaining[i];
                min_index = i;
            }
        }

        if (min_index == -1) {
            current_time++;
        } else {
            if (bt_remaining[min_index] == ps[min_index].bt) {
                ps[min_index].start_time = current_time;
            }
            bt_remaining[min_index]--;
            current_time++;
            prev = current_time;
            if (bt_remaining[min_index] == 0) {
                ps[min_index].ct = current_time;
                ps[min_index].tat = ps[min_index].ct - ps[min_index].at;
                ps[min_index].wt = ps[min_index].tat - ps[min_index].bt;
                sum_tat += ps[min_index].tat;
```

```
            sum_wt += ps[min_index].wt;
            completed++;
            is_completed[min_index] = true;
        }
    }
}

printf("\nAverage Turnaround Time = %f", sum_tat / n);
printf("\nAverage Waiting Time = %f\n", sum_wt / n);

return 0;
}
```

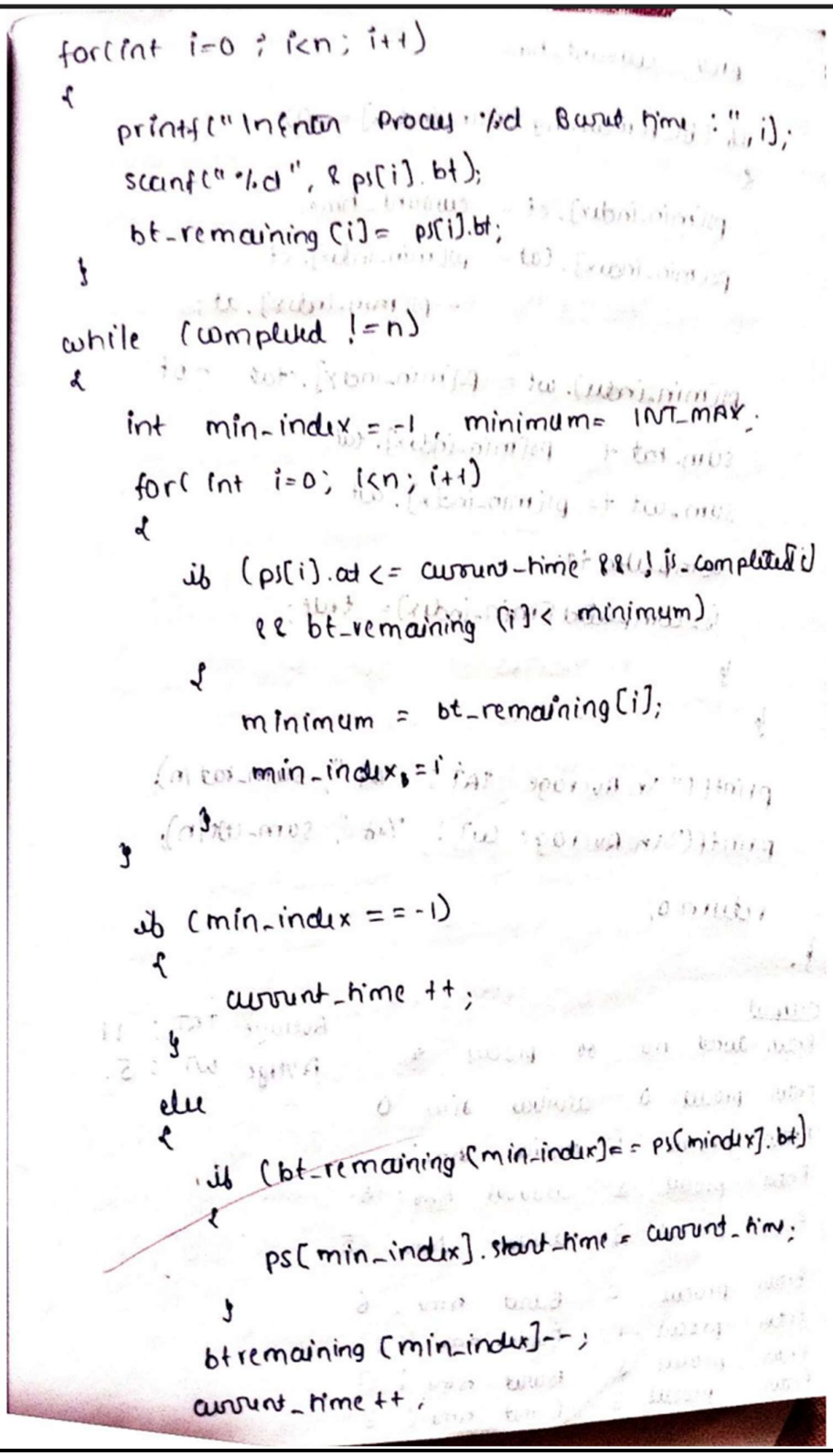
Result:



OBSERVATION:

SJF - Preemptive

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

struct process_struct
{
    int pid, at, bt, ct, wt, tat, start_time;
} ps[100];

int main()
{
    int n;
    float bt_remaining[100];
    bool is_completed[100]={false};
    int current_time = 0, completed=0;
    float sum_tat=0, sum_wt = 0;

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

    for(int i=0; i<n; i++)
    {
        printf("\nEnter process %d arrival ");
        scanf("%d", &ps[i].at);
        ps[i].pid=i;
    }
```



```
for(int i=0 ; i<n; i++)
{
    printf("\nEnter Process %d Burst time : ",i);
    scanf("%d", &ps[i].bt);
    bt_remaining[i] = ps[i].bt;
}

while (completed != n)
{
    int min_index = -1, minimum = INT_MAX;
    for( int i=0; i<n; i++)
    {
        if (ps[i].at <= current_time && !is_completed[i]
            && bt_remaining[i] < minimum)
        {
            minimum = bt_remaining[i];
            min_index = i;
        }
    }

    if (min_index == -1)
    {
        current_time ++;
    }
    else
    {
        if (bt_remaining[min_index] == ps[minindex].bt)
        {
            ps[min_index].start_time = current_time;
        }
        btremaining[min_index]--;
        current_time ++;
```

```
        prev = current_time;
        if (bt_remaining[min_index] == 0)
        {
            ps[min_index].ct = current_time;
            ps[min_index].tat = ps[min_index].ct
                                - ps[min_index].at;
            ps[min_index].wt = ps[min_index].ctat - bt;
            sum_tat += ps[min_index].tat;
            sum_wt += ps[min_index].wt
            completed++;
            is_completed[min_index] = true;
        }
    }
    printf("\n Average TAT : %f", sum_tat/n);
    printf("\n Average WT : %f", sum_wt/n);
    return 0;
}.
```

Output:

```
Enter total no of process : 4
Enter process 0 arrival time : 0
Enter process 1 arrival time : 2
Enter process 2 arrival time : 4
Enter process 3 arrival time : 5

Enter process 0 Burst time : 6
Enter process 1 Burst time : 8
Enter process 2 Burst time : 7
Enter process 3 Burst time : 3

Average TAT : 11
Averge WT : 5.
```

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)

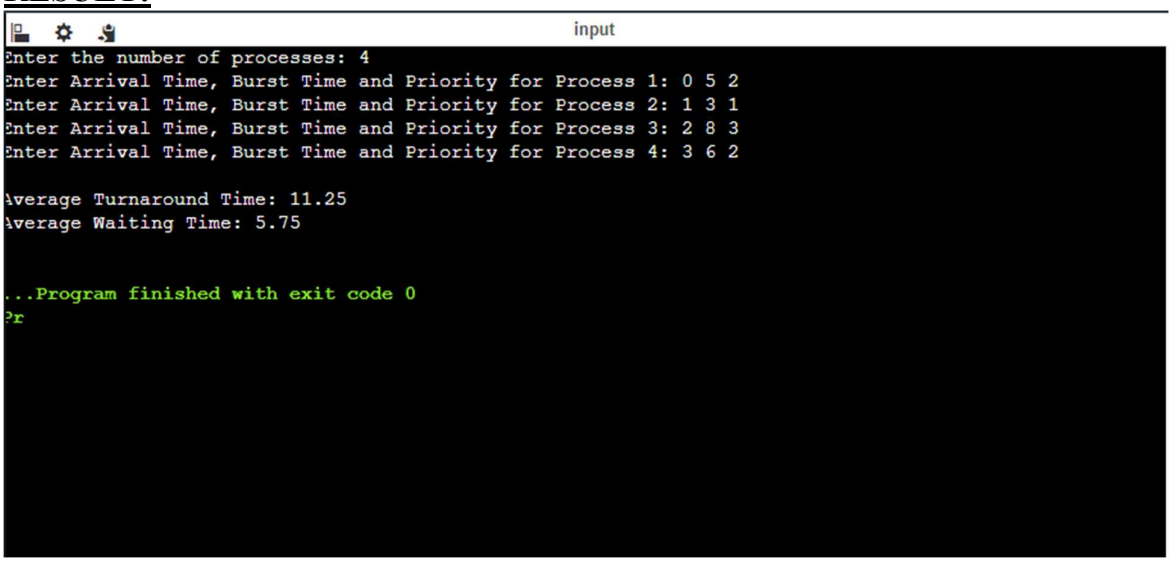
=>Priority(Non-PreEmptive)

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

struct Process {
   int id, at, bt, priority, ct, tat, wt;
};

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

   struct Process ps[n];
   int bt_remaining[n];

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

   int completed = 0, currentTime = 0;
   float totalTAT = 0, totalWT = 0;

   while (completed < n) {
      int min_index = -1;
      int minPriority = INT_MAX;
```

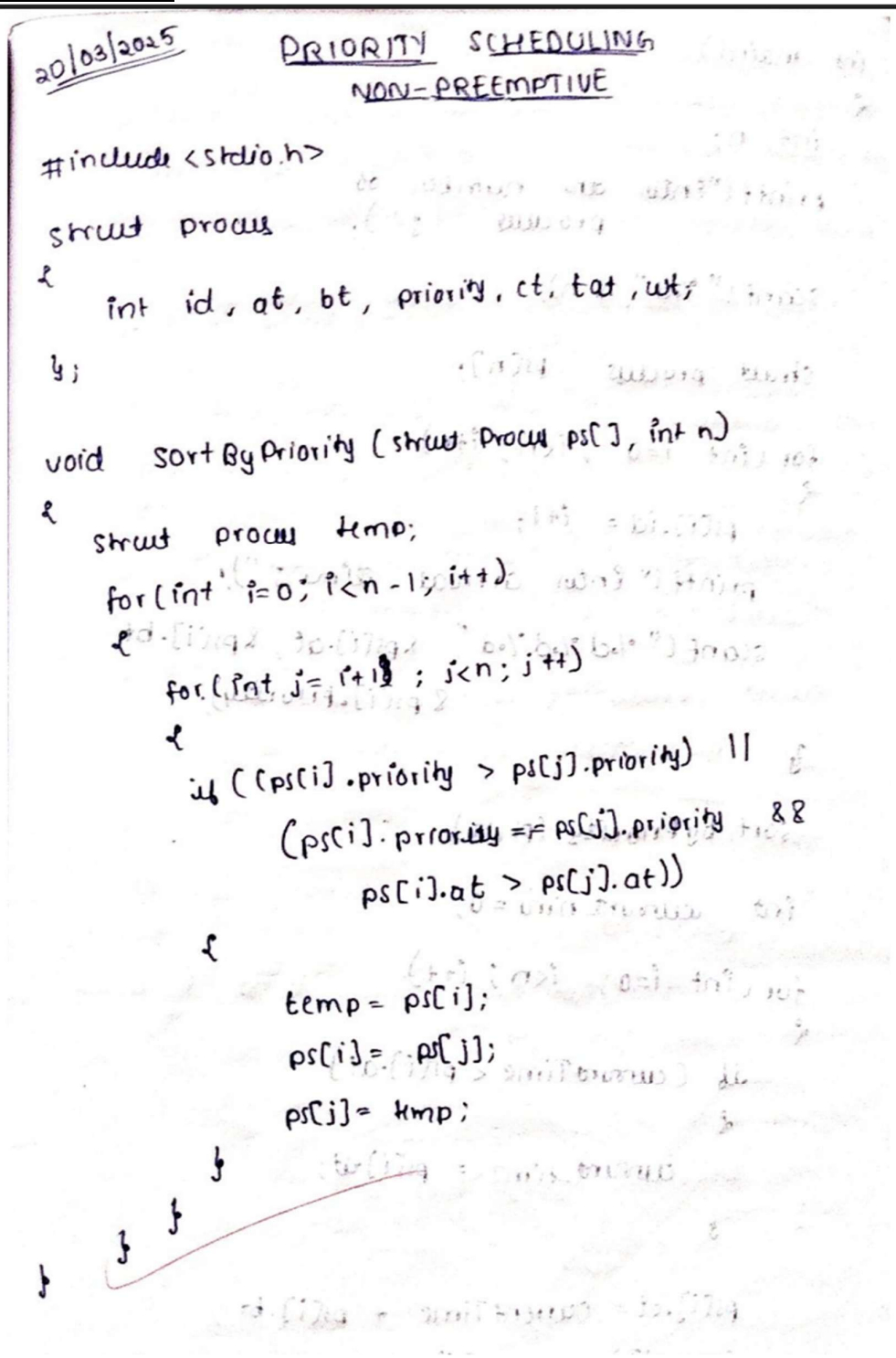
```
    for (int i = 0; i < n; i++) {
        if (ps[i].at <= currentTime && bt_remaining[i] > 0) {
            if (ps[i].priority < minPriority) {
                minPriority = ps[i].priority;
                min_index = i;
            }
        }
    }

    if (min_index != -1) {
        bt_remaining[min_index]--;
        currentTime++;


        if (bt_remaining[min_index] == 0) {
            ps[min_index].ct = currentTime;
            ps[min_index].tat = ps[min_index].ct - ps[min_index].at;
            ps[min_index].wt = ps[min_index].tat - ps[min_index].bt;
            totalTAT += ps[min_index].tat;
            totalWT += ps[min_index].wt;
            completed++;
        }
    } else {
        currentTime++;
    }
}


printf("\nProcess\tAT\tBT\tPriority\tCT\tTAT\tWT\n");
for (int i = 0; i < n; i++) {
    printf("%d\t%d\t%d\t%d\t\t%d\t%d\t%d\n",
           ps[i].id, ps[i].at, ps[i].bt, ps[i].priority,
           ps[i].ct, ps[i].tat, ps[i].wt);
}

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

```
    return 0;
}
```

RESULT:



OBSERVATION:



20/03/2025

PRIORITY SCHEDULING
NON-PREEMPTIVE

```
#include <stdio.h>

struct process
{
    int id, at, bt, priority, ct, tat, wt;
};

void sortByPriority (struct Process ps[], int n)
{
    struct process temp;
    for(int i=0; i<n-1; i++)
    {
        for(int j= i+1 ; j<n; j++)
        {
          if ((ps[i].priority > ps[j].priority) ||
              (ps[i].prrority == ps[j].priority &&
                  ps[i].at > ps[j].at))
          {
              temp = ps[i];
              ps[i] = ps[j];
              ps[j] = tmp;
          }
        }
    }
}
```

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

    struct process ps[n];

    for (int i=0 ; i<n; i++)
    {
        ps[i].id = i+1;
        printf("Enter arrival time : ");
        scanf("%d %d %d", &ps[i].at, &ps[i].bt,
                        &ps[i].priority);
    }

    sortByPriority(ps, n);

    int currentTime = 0;

    for(int i=0; i<n; i++)
    {
        if (currentTime < ps[i].at)
        {
            currentTime = ps[i].at;
        }

        ps[i].ct = currentTime + ps[i].bt;
        currentTime = ps[i].ct;
```

```
ps[i].tat = ps[i].ct - ps[i].at,
ps[i]. wt = ps[i].tat - ps[i].bt,

totalTAT+= ps[i].tat;
totalWT+= ps[i]. wt;
}

printf(" Average TAT : %f", totalTAT/n);
printf(" Average WT : %f ", totalWT/n);

};
```

=>Priority(Pre-Emptive):

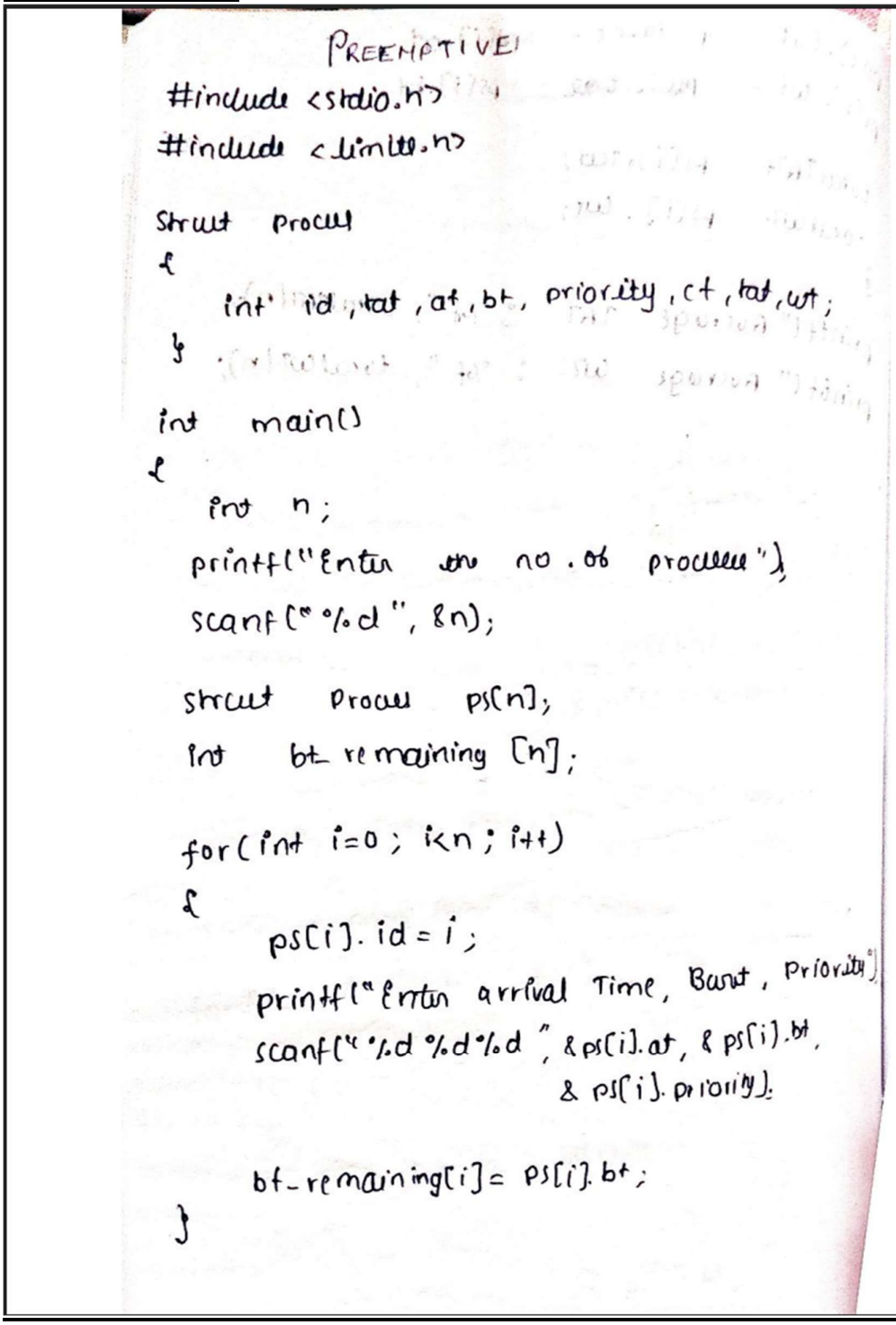
```
#include <stdio.h>
#include <limits.h>

struct Process {
   int id, at, bt, priority, ct, tat, wt;
};

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

   struct Process ps[n];
   int bt_remaining[n];

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

   int completed = 0, currentTime = 0;
   float totalTAT = 0, totalWT = 0;

   while (completed < n) {
      int min_index = -1;
      int minPriority = INT_MAX;


      for (int i = 0; i < n; i++) {
         if (ps[i].at <= currentTime && bt_remaining[i] > 0) {
            if (ps[i].priority < minPriority) {
               minPriority = ps[i].priority;
               min_index = i;
            }
```

```
            }
        }

        if (min_index != -1) {
            bt_remaining[min_index]--;
            currentTime++;


            if (bt_remaining[min_index] == 0) {
                ps[min_index].ct = currentTime;
                ps[min_index].tat = ps[min_index].ct - ps[min_index].at;
                ps[min_index].wt = ps[min_index].tat - ps[min_index].bt;
                totalTAT += ps[min_index].tat;
                totalWT += ps[min_index].wt;
                completed++;
            }
        } else {
            currentTime++;
        }
    }

    printf("\nProcess\tAT\tBT\tPriority\tCT\tTAT\tWT\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t\t%d\t%d\t%d\n",
               ps[i].id, ps[i].at, ps[i].bt, ps[i].priority,
               ps[i].ct, ps[i].tat, ps[i].wt);
    }

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

    return 0;
}
```

RESULT:

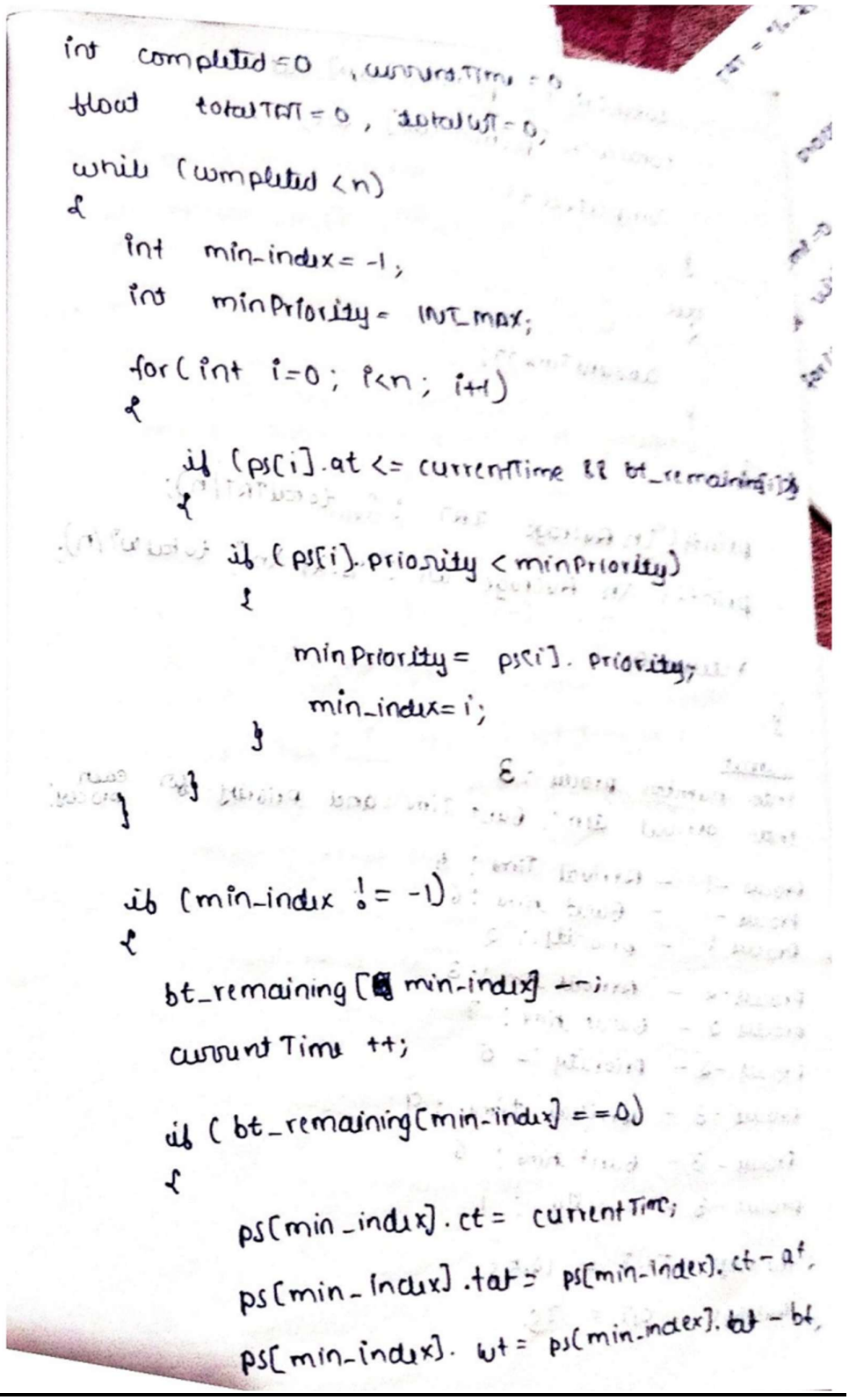
```
Enter the number of processes: 3
Enter Arrival Time, Burst Time and Priority for Process 1: 0 5 3
Enter Arrival Time, Burst Time and Priority for Process 2: 1 3 1
Enter Arrival Time, Burst Time and Priority for Process 3: 2 4 2

Process AT      BT      Priority        CT      TAT     WT
1       0       5       3               12      12      7
2       1       3       1               4       3       0
3       2       4       2               8       6       2

Average Turnaround Time: 7.00
Average Waiting Time: 3.00

Process returned 0 (0x0)   execution time : 21.094 s
Press any key to continue.
```

OBSERVATION:



PREEMPTIVE

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

Struct Process
{
    int id, bat, at, bt, priority, ct, tat, wt;
};

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

    struct Process ps[n];
    int bt_remaining [n];

    for(int i=0; i<n; i++)
    {
        ps[i].id = i;
        printf("Enter arrival Time, Burst, Priority");
        scanf("%d %d %d", &ps[i].at, &ps[i].bt,
                          &ps[i].priority);

        bt_remaining[i] = ps[i].bt;
    }
```



```
int completed = 0 , currentTime = 0
float totalTAT = 0 , totalWT = 0;

while (completed < n)
{
    int min_index = -1;
    int minPriority = INT_MAX;

    for(int i=0; i<n; i++)
    {
        if (ps[i].at <= currentTime && bt_remaining[i] > 0)
        {
            if (ps[i].priority < minPriority)
            {
                minPriority = ps[i].priority;
                min_index = i;
            }
        }
    }

    if (min_index != -1)
    {
        bt_remaining[min_index] --;
        currentTime ++;

        if ( bt_remaining[min_index] == 0)
        {
            ps[min_index].ct = currentTime;
            ps[min_index].tat = ps[min_index].ct - ps[min_index].at;
            ps[min_index].wt = ps[min_index].tat - bt[
```

```
        totalTAT += ps[minIndex].tat;
        totalWT += ps[minIndex].wt;
        completed++;
    }
    else
    {
        currentTime++;
    }
}

printf("\n Average TAT : %f", totalTAT/n);
printf("\n Average WT : %f\n", totalWT/n);

return 0;
}
```

Output

Enter number process : 3
Enter arrival time, Burst Time and priority for each proc

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

Average TAT = 12.5s.
Average WT = 7s.

ROUND-ROBIN:

Code:

```
#include <stdio.h>

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

int main() {
    int n, quantum;
    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 and Burst Time for Process %d: ", i + 1);
        scanf("%d %d", &ps[i].at, &ps[i].bt);
        ps[i].rt = ps[i].bt;
    }

    printf("Enter Time Quantum: ");
    scanf("%d", &quantum);

    int completed = 0, currentTime = 0;
    float totalTAT = 0, totalWT = 0;

    while (completed < n) {
        int done = 1;
        for (int i = 0; i < n; i++) {
            if (ps[i].rt > 0 && ps[i].at <= currentTime) {
```

```
            done = 0;
            if (ps[i].rt > quantum) {
                currentTime += quantum;
                ps[i].rt -= quantum;
            } else {
                currentTime += ps[i].rt;
                ps[i].rt = 0;
                ps[i].ct = currentTime;
                ps[i].tat = ps[i].ct - ps[i].at;
                ps[i].wt = ps[i].tat - ps[i].bt;
                totalTAT += ps[i].tat;
                totalWT += ps[i].wt;
                completed++;
            }
        }
    }
    if (done) currentTime++;
}

printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\n");
for (int i = 0; i < n; i++) {
    printf("%d\t%d\t%d\t%d\t%d\t%d\n",
           ps[i].id, ps[i].at, ps[i].bt, ps[i].ct, ps[i].tat, ps[i].wt);
}

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

return 0;
}
```

RESULT:

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

Process AT      BT      CT      TAT     WT
1       0       5       11      11      6
2       1       4       10      9       5
3       2       2       6       4       2

Average Turnaround Time: 8.00
Average Waiting Time: 4.33


...Program finished with exit code 0
Press ENTER to exit console.
```

OBSERVATION:

Round robin

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

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

    struct Process ps[n];
    for( int i=0 ; i<n; i++)
    {
        ps[i].id = i+1;
        printf("Enter the arrival time & BT");
        scanf("%d %d", &ps[i].at, &ps[i].bt);
        ps[i].rt = ps[i].bt;
    }
    printf("Enter time quantum :");
    scanf("%d", &quantum);

    int completed = 0, currentTime = 0;
    float totalTAT = 0, totalWT = 0;
```

```
while (completed < n)
{
    int done = 1;
    for (int i=0 ; i<n; i++)
    {
        if (ps[i].rt >0 && ps[i].at <= currentTime)
        {
            done =0;
            if (ps[i].rt > quantum)
            {
                currentTime += quantum;
                ps[i].rt-= quantum;
            }
            else
            {
                currentTime += ps[i].rt;
                ps[i].rt = 0;
                ps[i].ct = currentTime;
                ps[i].tat = ps[i].ct - ps[i].at;
                ps[i].wt = ps[i].tat - ps[i].bt;
                totalTAT += ps[i].tat;
                totalWT += ps[i].wt
                completed++;
            }
        }
    }
    if (done) currentTime++;
}
```

```
    printf(" Average TAT : %.2f ", total_tat/n);
    printf(" Average WT : %.2f ", total_wt/n);
    return 0;
}
```

Output:

Enter number of process : 3

Enter Arrival time and Burst Time Process 1: 0 5

Enter arrival time and Burst Process 2 : 1 4

Enter arrival time and Burst time Process 3 : 2 2

Process	AT	BT	CT	TAT	WT
1	0	5	11	11	6
2	1	4	10	9	5
3	2	2	6	4	2

Average turn around time : 8

Average waiting time : 4.33

PROGRAM 3:

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

CODE:

```
#include<stdio.h>

struct Process
{
   int at,bt,ct,tat,id,wt,queue;
};

int main()
{
   int n, quantum1;
   printf("Enter the number of processes: ");
   scanf("%d", &n);

   struct Process ps[n];
   int bt_remaining[n];

   for (int i = 0; i < n; i++) {
      ps[i].id = i + 1;
      printf("Enter Arrival Time, Burst Time, and Queue (1: RR, 2: FCFS) for
Process %d: ", i + 1);
      scanf("%d %d %d", &ps[i].at, &ps[i].bt, &ps[i].queue);
      bt_remaining[i] = ps[i].bt;
   }

   printf("Enter the quantum for Round Robin(Queue 1) : ");
   scanf("%d", &quantum1);

   int currentTime=0,completed=0;
   float totalTAT=0,totalWT=0;
```

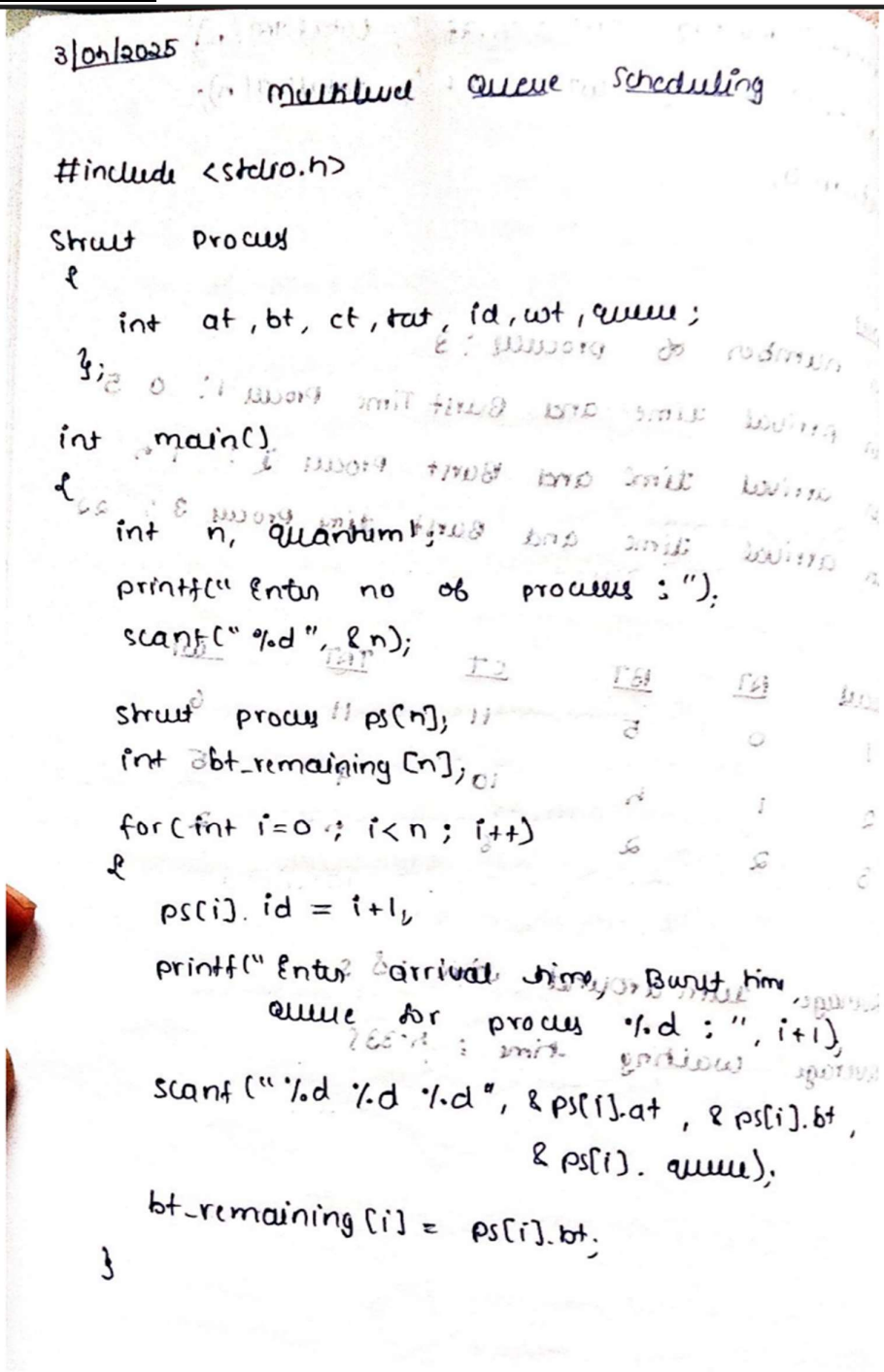
```
while(completed<n)
{
   int done=1;
   for(int i=0;i<n;i++)
   {
      if(ps[i].queue==1 && ps[i].at<=currentTime && bt_remaining[i]>0)
      {
         done=0;
         if(bt_remaining[i]>quantum1)
         {
            currentTime+=quantum1;
            bt_remaining[i]-=quantum1;
         }
         else{
            currentTime+=quantum1;
            bt_remaining[i]=0;
            ps[i].ct=currentTime;
            ps[i].tat=ps[i].ct-ps[i].at;
            ps[i].wt=ps[i].tat-ps[i].bt;
            totalTAT+=ps[i].tat;
            totalWT+=ps[i].wt;
            completed++;
         }
      }
   }
   if(done)
   {
      break;
   }
}

while(completed<n)
{
   int done = 1;
   for (int i = 0; i < n; i++)
      {
```

```
        if (ps[i].queue == 2 && bt_remaining[i] > 0 && ps[i].at <= currentTime)
         {
            done = 0;
            currentTime += bt_remaining[i];
            bt_remaining[i] = 0;
            ps[i].ct = currentTime;
            ps[i].tat = ps[i].ct - ps[i].at;
            ps[i].wt = ps[i].tat - ps[i].bt;
            totalTAT += ps[i].tat;
            totalWT += ps[i].wt;
            completed++;
        }
      }
      if (done){break;}
   }

    printf("\nProcess\tAT\tBT\tQueue\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].queue, ps[i].ct, ps[i].tat, ps[i].wt);
   }

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

   return 0;
}
```

RESULT:

```
Enter the number of processes: 4
Enter Arrival Time, Burst Time, and Queue (1: RR, 2: FCFS) for Process 1: 0 2 1
Enter Arrival Time, Burst Time, and Queue (1: RR, 2: FCFS) for Process 2: 0 1 2
Enter Arrival Time, Burst Time, and Queue (1: RR, 2: FCFS) for Process 3: 0 5 1
Enter Arrival Time, Burst Time, and Queue (1: RR, 2: FCFS) for Process 4: 0 3 2
Enter the quantum for Round Robin(Queue 1) : 2

Process AT      BT      Queue   CT      TAT     WT
1       0       2       1       2       2       0
2       0       1       2       9       9       8
3       0       5       1       8       8       3
4       0       3       2       12      12      9

Average Turnaround Time: 7.75
Average Waiting Time: 5.00

Process returned 0 (0x0)   execution time : 77.902 s
Press any key to continue.
```

OBSERVATION:



3/04/2025

Multilevel Queue Scheduling

```
#include <stdio.h>

Struct Process
{
    int at, bt, ct, tat, id, wt, queue;
};
int main()
{
    int n, quantum;
    printf(" Enter no of processes : ");
    scanf(" %d ", &n);

    Struct process ps[n];
    int bt_remaining[n];

    for(int i=0 ; i<n ; i++)
    {
        ps[i].id = i+1;
        printf(" Enter arrival time, Burst time,
                queue for process %d : ", i+1);
        scanf(" %d %d %d ", &ps[i].at, &ps[i].bt,
                            &ps[i].queue);
        bt_remaining[i] = ps[i].bt;
    }
```

```
printf("Enter quantum for Round Robin: ");
scanf("%d", &quantum);

int currentTime=0, completed=0;
float totalTAT=0, totalWT=0;

while (completed < n)
{
    int done=1;
    for (int i=0; i<n; i++)
    {
        if (ps[i].queue==1 && ps[i].at<=currentTime
            && bt_remaining[i]>0)
        {
            done=0;
            if (bt_remaining[i] > quantum)
            {
                currentTime += quantum;
                bt_remaining[i] -= quantum;
            }
            else {
                currentTime += quantum;
                bt_remaining[i]=0;
                ps[i].ct = currentTime;
                ps[i].tat = ps[i].ct - ps[i].at;
                ps[i].wt = ps[i].tat - ps[i].bt;
                totalTAT += ps[i].tat
                totalWT += ps[i].wt;
                completed ++;
            }
        }
    }
```

```
        if (done)
        {
            break;
        }
    }

    while (completed < n)
    {
        int done = 1;
        for(int i=0; i<n; i++)
        {
            if (ps[i].queue == 2 && bt_remaining[i] > 0
                    && ps[i].at <= currentTime)
            {
                done = 0;
                currentTime += bt_remaining[i];
                bt_remaining[i] = 0;
                ps[i].ct = currentTime;
                ps[i].tat = ps[i].ct - ps[i].at;
                ps[i].wt = ps[i].tat - ps[i].bt;
                totalTAT += ps[i].tat;
                totalWT += ps[i].wt;
                completed++;
            }
        }
        if (done)
        {
            break;
        }
    }
```

```
printf("\nProcess\t AT\t BT\t Queue\t CT\t TAT\t WT\n");
for(int i=0; i<n; i++)
{
printf("%d\t%d\t %d\t %d\t %d\t %d\t %d",
      ps[i].id, ps[i].at, ps[i].bt, ps[i].queue, ps[i].ct,
      ps[i].tat; ps[i].wt);

printf(" TAT : %.2f ", totalTAT/n);
printf(" Average WT : %.2f ", totalWT(n);

}.
```

Output:

Enter the number of processes : 4

Enter Arrival time, BT, and Queue (1:RR, 2:FCFS): 0 2 1

Enter arrival time, BT, and Queue (1:RR, 2:FCFS): 0 1 2

Enter Arrival time, BT and Queue (1:RR, 2:FCFS): 0 5 1

Enter arrival time, BT and Queue (1:RR, 2: FCFS): 0 3 2

Process	AT	BT	Queue	CT	TAT	WT
1	0	2	1	2	2	0
2	0	1	2	9	9	8
3	0	5	1	8	8	3
4	0	3	2	12	12	9

Average TurnAround Time : 7.75

Average waiting Time : 5.00

PROGRAM 4:

4.	**Write a C program to simulate Real-Time CPU Scheduling algorithms:** d) **Rate- Monotonic** e) **Earliest-deadline First** f) **Proportional scheduling**

CODE:

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

typedef struct {
    int pid;
    int period;
    int burst;
    int remaining;
} 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 findHyperPeriod(Task tasks[], int n) {
```

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

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;
    }
}

// Function to calculate CPU utilization
float calculateCPUUtilization(Task
tasks[], int n) {
    float utilization = 0.0;
    for (int i = 0; i < n; i++) {
```

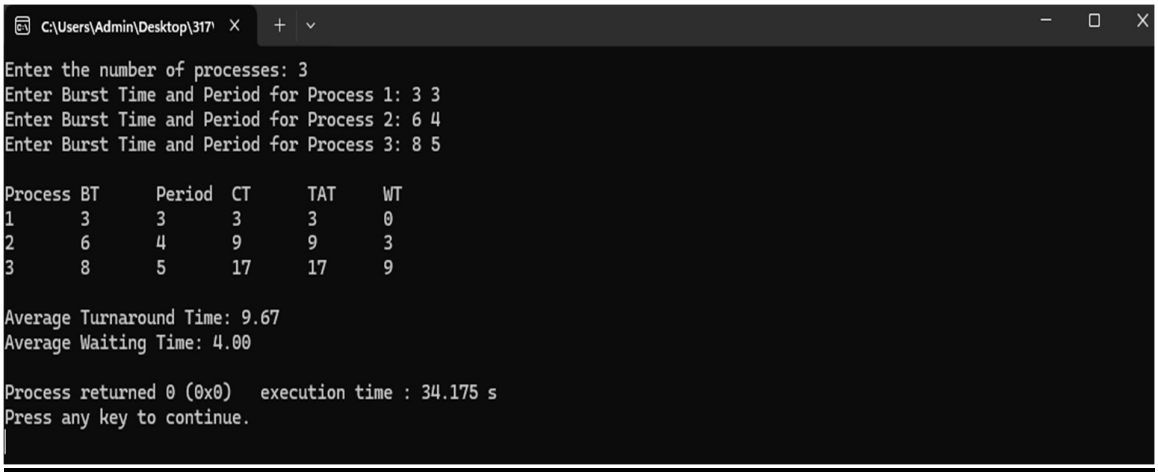
```
        utilization += (float)tasks[i].burst /
tasks[i].period;
    }
    return utilization;
}

// Function to check if the system is
schedulable
int isSchedulable(Task tasks[], int n) {
    float utilization =
calculateCPUUtilization(tasks, n);
    // Schedulability bound for RMS
    float threshold = n * (pow(2, 1.0 / n) -
1);

    printf("CPU Utilization: %.2f\n",
utilization);
    printf("Schedulability Threshold:
%.2f\n", threshold);

    if (utilization <= threshold) {
        return 1;  // System is schedulable
    } else {
        return 0;  // System is not schedulable
    }
}
```

```
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) {
```

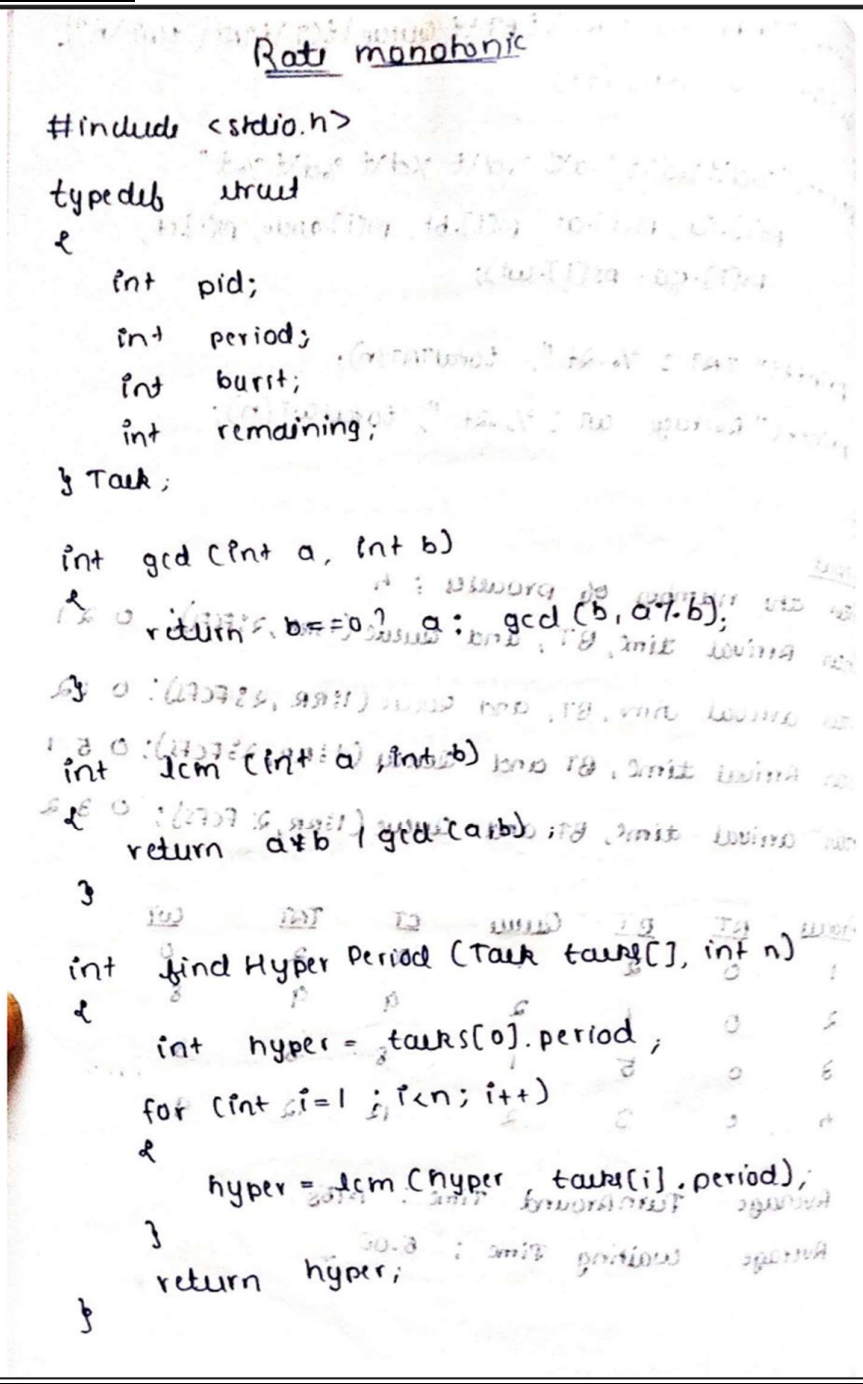
```
            tasks[scheduled].remaining--;
            printf("%d\tT%d\n", time,
tasks[scheduled].pid);
        } else {
            printf("%d\tIdle\n", time);
        }
    }
}

int main() {
    int n;
    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);
    }

    // Check schedulability before
proceeding
    if (!isSchedulable(tasks, n)) {
```

```
        printf("The task set is not schedulable
under Rate-Monotonic Scheduling.\n");
        return 0;
    }

    int sim_time = findHyperPeriod(tasks,
n);
    printf("Simulation will run for %d units
(LCM of periods)\n", sim_time);

    rateMonotonic(tasks, n, sim_time);

    return 0;
}
```

RESULT:



OBSERVATION:



Rate monotonic

```
#include <stdio.h>

typedef struct
{
    int pid;
    int period;
    int burst;
    int remaining;
} 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 findHyperPeriod(Task tasks[], int n)
{
    int hyper = tasks[0].period;
    for(int i=1; i<n; i++)
    {
        hyper = lcm(hyper, tasks[i].period);
    }
    return hyper;
}
```

```
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;
    }
}

//Function to calculate CPU utilisation
float calculateCPUUtilization(Task tasks[], int n)
{
    float utilization = 0.0;
    for(int i=0; i<n; i++)
    {
        utilization += (float) tasks[i].burst / tasks[i].
                                                period;
    }
    return utilisation;
}
```

```
int isSchedulable (Task tasks[], int n)
{
    float utilization = calculateCPUUtilization (tasks,
    float threshold = n * (pow(2, 1.0/n) - 1);
    printf("CPU utilization : %.2f \n", utilization);
    printf("Schedulability threshold %.2f\n", threshold
    if (utilization <= threshold)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
void rateMonotonic (Task tasks[], int n, int sim_t
{
    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 \t T %d \n ", time, tasks[scheduled].pid);
    }
    else
    {
        printf(" %d \t Idle \n", time);
    }
}
```

```
int main()
{
    int n;
    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].bu
    }

    if(!isSchedulable(tasks, n))
    {
        printf("The task set is not schedulable
        under Rate-Monotonic Scheduling.\n");
        return 0;
    }

    int sim_time = findHyperPeriod(tasks, n);
    printf("Simulation will run for %d
        units (LCM of period)\n", sim_time);

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

Output :

Enter number of tasks : 2

Enter period and burst time task T1 : 2 1

Enter period and burst time task T2 : 2 3

Enter simulation time : 3

Rate monotonic scheduling

Time	Task
0	T1
1	T2
2	T1

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) {
```

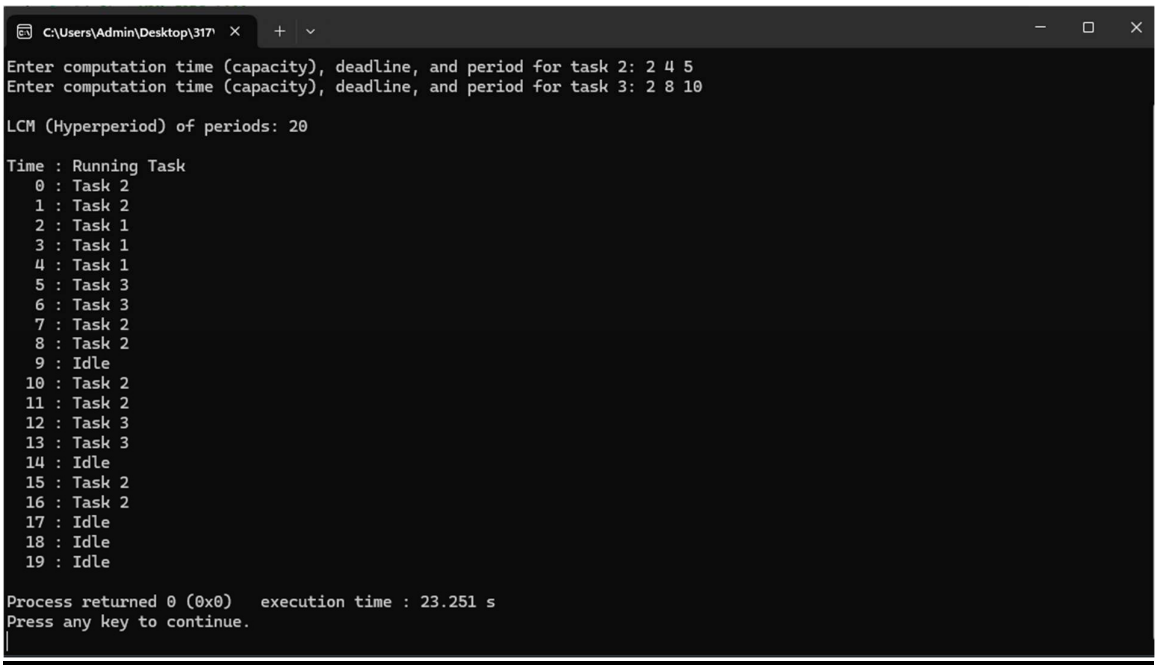
```
    int l = tasks[0].period;
    for (int i = 1; i < k; i++) {
        l = lcm(l, tasks[i].period);
    }
    return l;
}

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;
```

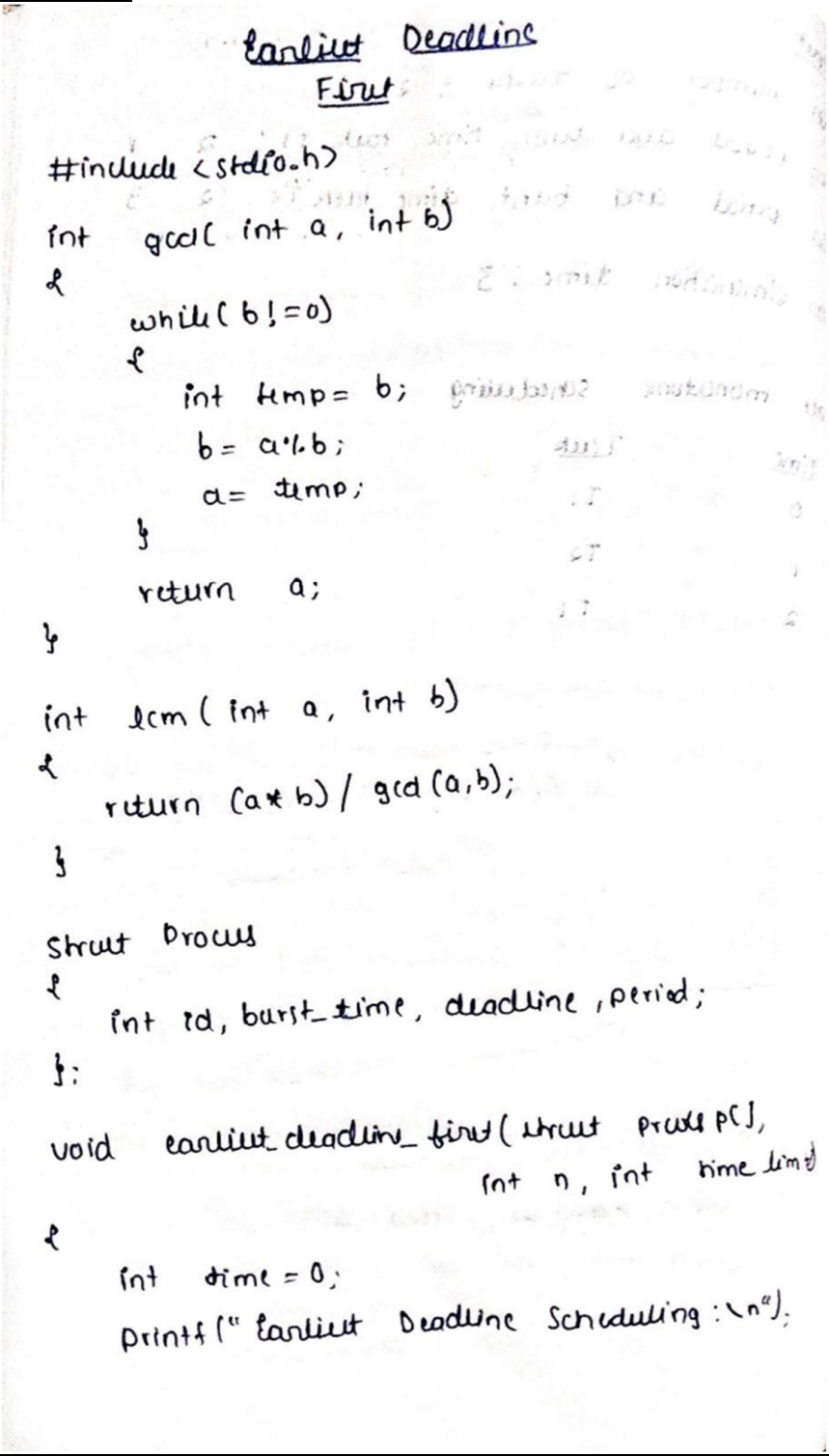
```
                selected_job_index = j;
            }
        }
    }

    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;
}
```

RESULT:



OBSERVATION:



Earliest Deadline First

```
#include <stdio.h>
int gcd( int a, int b)
{
    while( b!=0)
    {
       int temp= b;
       b= a%b;
       a= temp;
    }
    return a;
}
int lcm ( int a, int b)
{
   return (a* b) / gcd (a,b);
}
Struct Procss
{
   int id, burst_time, deadline , period;
};
void earliest_deadline_first ( struct procss p[],
                               int n, int time limit)
{
    int time = 0;
    Printf (" Earliest Deadline Scheduling :\n");
```

```
printf(" PID \t Burst \t Deadline \t Period \n");
for ( int i=0; i<n; i++)
{
  printf("%d \t %d \t \t %d \t \t %d \n", P[i].id,
        P[i].burst_time, P[i].deadline, P[i].period);
}

printf(" \n Scheduling occurs for %d ms \n");
                                    time_limit;

while ( time < time_limit)
{
   int earliest = -1;
   for ( int i=0 ; i<n; i++)
   {
      if (P[i].burst_time > 0)
      {
         if (earliest == -1 || P[i].deadline
                          < P[earliest].deadline)
         {
            earliest = i;
         }
      }
   }
   if (earliest == -1)
      break;
   printf("%d ms : Task %d is running. \n",
                  time, P[earliest].id);
   P[earliest].burst_time --;
   time ++;
}
```

```
int main()
{
    int n;
    printf("Enter the number of process :");
    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].completed = 0;
    }

    int completed = 0, currentTime = 0;
    float totalTAT = 0, totalWT = 0;

    while (completed != n)
    {
      int idx = -1;
      int earliestDeadline = 1e9;

      for(int i=0; i<n; i++)
      {
        if(!ps[i].isCompleted && ps[i].at <=
currentTime && ps[i].dl < earliestDeadline)
        {
          earliest = ps[i].dl;
          idx = i;
        }
}
```

```
if (idx != -1) {
    currentTime += ps[i].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\t%d\n", ps[i].id,
           ps[i].at, ps[i].bt, ps[i].dl, ps[i].ct,
           ps[i].ct, ps[i].tat, ps[i].wt);
}
printf("\nAverage TAT: %.2f\n", totalTAT/n);
printf("Average WT: %.2f\n", totalWT/n);
return 0;
}
```

Output:

Enter the no of process : 3

Enter arrival, Burst and deadline : 0 4 6

Enter arrival, Burst and deadline
for process 2 : 1 3 5

Enter Arrival, Burst 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

PROGRAM 5:

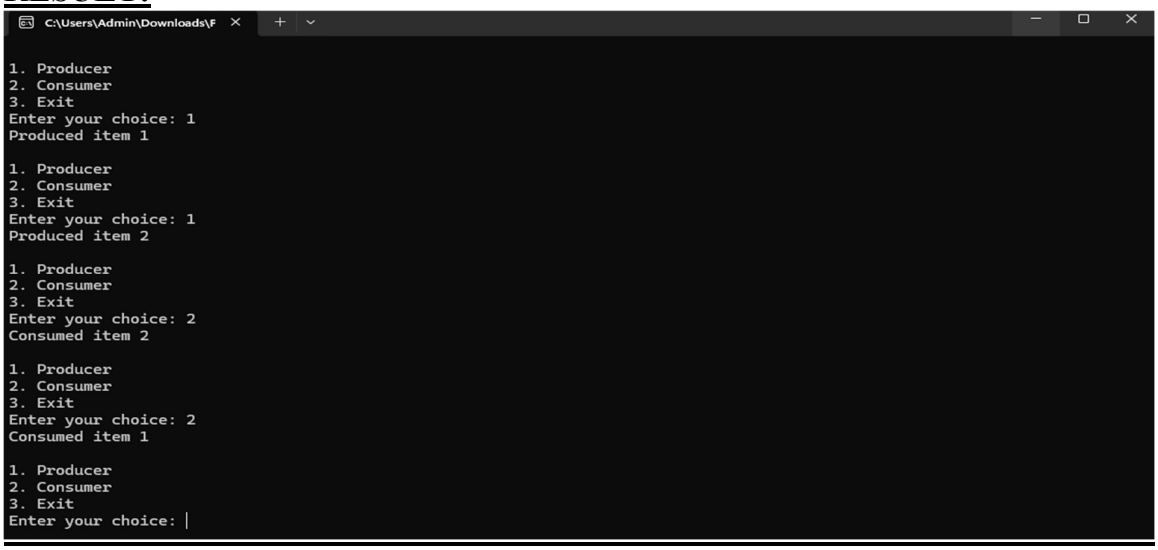
Write a C program to simulate producer-consumer problem using semaphores

CODE:

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

int mutex=1,full=0,empty=3,x=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("Produced item %d\n", x);
   mutex = signal(mutex);
}

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

```
int main() {
   int choice;
   while (1) {
      printf("\n1. Producer\n2. Consumer\n3. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            if ((mutex == 1) && (empty != 0))
               producer();
            else
               printf("Buffer is full\n");
            break;
         case 2:
            if ((mutex == 1) && (full != 0))
               consumer();
            else
               printf("Buffer is empty\n");
            break;
         case 3:
            exit(0);
         default:
            printf("Invalid choice\n");
      }
   }
   return 0;
}
```

RESULT:


C:\Users\Admin\Downloads\F
1. Producer
2. Consumer
3. Exit
Enter your choice: 1
Produced item 1
1. Producer
2. Consumer
3. Exit
Enter your choice: 1
Produced item 2
1. Producer
2. Consumer
3. Exit
Enter your choice: 2
Consumed item 2
1. Producer
2. Consumer
3. Exit
Enter your choice: 2
Consumed item 1
1. Producer
2. Consumer
3. Exit
Enter your choice:

OBSERVATION :

16-04-2025

Producer Consumer

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

int mutex = 1 , full = 0 , empty = 3 , x = 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(" Produced item : %d ", x);
    mutex = signal (mutex);
}
```

```
Void consumer()
{
    mutex = wait(mutex);
    full = wait(full);
    empty = signal(empty);
    printf("Consumed item %d\n", x);
    x--;
    mutex = signal(mutex);
}

int main()
{
    int choice;
    while(1)
    {
      printf("\n1. Producer \n2. Consumer \n3. Exi
      printf("Enter your choice");
      scanf("%d", &choice);
      switch(choice)
      {
        case 1:
            if((mutex==1) && (empty!=0))
                producer();
            else
                printf("Buffer full");
            break;
```

```
case 2:
    if ((mutex==1) && (full!=0))
        consumer();
    else
        printf(" Buffer empty ");
    break;
case 3:
    exit(0);
default:
    printf(" Invalid choice ");
    }
}
return 0;
}
```

Output:

1. Producer
2. Consumer
3. exit

Enter your choice : 1

Produced item : 1

Enter your choice : 2

Produced item : 2

Enter your choice : 3

exited..

PROGRAM -6:

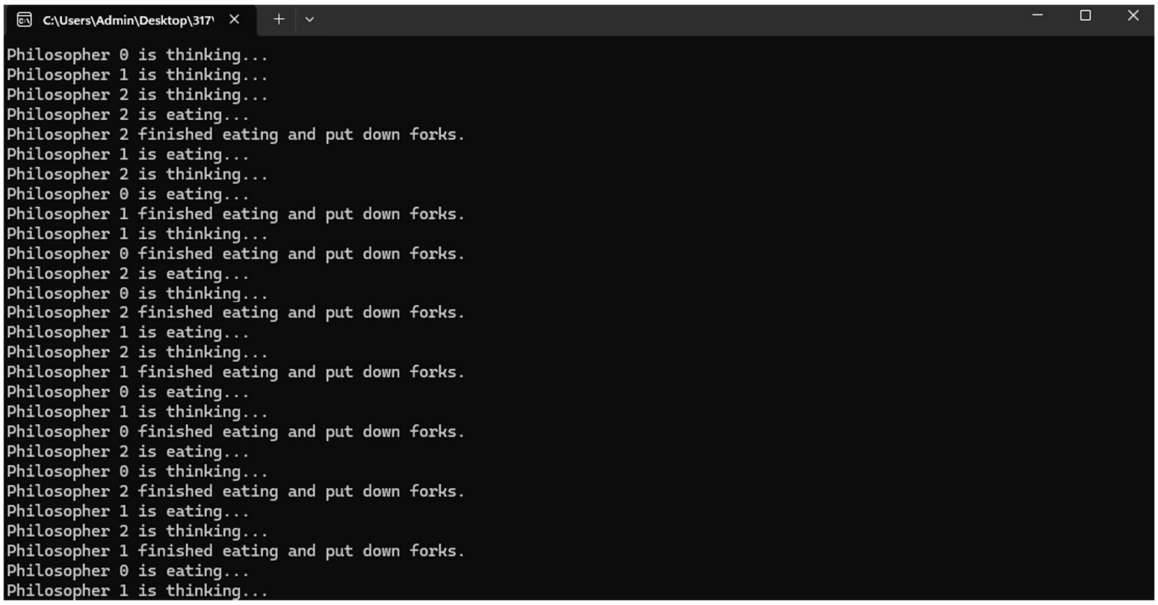
Write a C program to simulate the concept of Dining Philosophers problem.

CODE:

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

#define N 3  // Number of philosophers

sem_t forks[N];        // One semaphore per fork
sem_t mutex;           // Global mutex to avoid deadlock

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

   while (1) {
      printf("Philosopher %d is thinking...\n", id);
      sleep(1);  // Simulate thinking

      sem_wait(&mutex);          // Request permission to pick up forks
      sem_wait(&forks[id]);      // Pick up left fork
      sem_wait(&forks[(id + 1) % N]); // Pick up right fork
      sem_post(&mutex);          // Release global lock

      printf("Philosopher %d is eating...\n", id);
      sleep(2);  // Simulate eating

      sem_post(&forks[id]);      // Put down left fork
      sem_post(&forks[(id + 1) % N]); // Put down right fork

      printf("Philosopher %d finished eating and put down forks.\n", id);
      sleep(1);
   }
```

```
}

int main() {
    pthread_t tid[N];
    int ids[N];

    // Initialize semaphores
    for (int i = 0; i < N; i++) {
        sem_init(&forks[i], 0, 1);  // One fork available
    }
    sem_init(&mutex, 0, 1);  // Binary semaphore

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

    // Join threads (optional, since they run forever)
    for (int i = 0; i < N; i++) {
        pthread_join(tid[i], NULL);
    }

    // Destroy semaphores (not reached here)
    for (int i = 0; i < N; i++) {
        sem_destroy(&forks[i]);
    }
    sem_destroy(&mutex);

    return 0;
}
```

RESULT:


C:\Users\Admin\Desktop\317
Philosopher 0 is thinking...
Philosopher 1 is thinking...
Philosopher 2 is thinking...
Philosopher 2 is eating...
Philosopher 2 finished eating and put down forks.
Philosopher 1 is eating...
Philosopher 2 is thinking...
Philosopher 0 is eating...
Philosopher 1 finished eating and put down forks.
Philosopher 1 is thinking...
Philosopher 0 finished eating and put down forks.
Philosopher 2 is eating...
Philosopher 0 is thinking...
Philosopher 2 finished eating and put down forks.
Philosopher 1 is eating...
Philosopher 2 is thinking...
Philosopher 1 finished eating and put down forks.
Philosopher 0 is eating...
Philosopher 1 is thinking...
Philosopher 0 finished eating and put down forks.
Philosopher 2 is eating...
Philosopher 0 is thinking...
Philosopher 2 finished eating and put down forks.
Philosopher 1 is eating...
Philosopher 2 is thinking...
Philosopher 1 finished eating and put down forks.
Philosopher 0 is eating...
Philosopher 1 is thinking...

OBSERVATION:

16-04-2025

Dining Philosopher

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

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

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

sem_t  mutex;
sem_t  S[N];

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

  {
     state [phnum]= EATING;
     sleep(2);
     printf (" philosopher %d takes %d and %d \n",
           phnum+1,  LEFT+1,  phnum+1);
```

```
    printf(" Philosopher %d is Eating ", phnum+1);
    sem_post (&S[phnum]);
    }
}

void take_fork(int phnum)
{
    sem_wait (&mutex);
    state [phnum]= HUNGRY;
    printf(" Philosopher %d is hungry \n", phnum+1);
    test (phnum);
    sem_post (&mutex);
    sem_wait (&S[phnum]);
    sleep(1);
}

void putfork (int phnum)
{
    sem_wait (&mutex);
    state [phnum] = THINKING;
    printf(" Philosopher %d putting fork %d & %d down",
            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(0);
        put_fork (*i);
    }
}

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

    sem_init(&mutex,0,1);

    for(int 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;
}
```

Output:

```
philosopher 0 is thinking...
philosopher 1 is thinking...
philosopher 2 is thinking...
philosopher 2 finished eating and put down forks
philosopher 1 is eating
philosopher 2 is thinking
:
:
:
:
```

P 22/4 infinite loop.

PROGRAM-7:

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

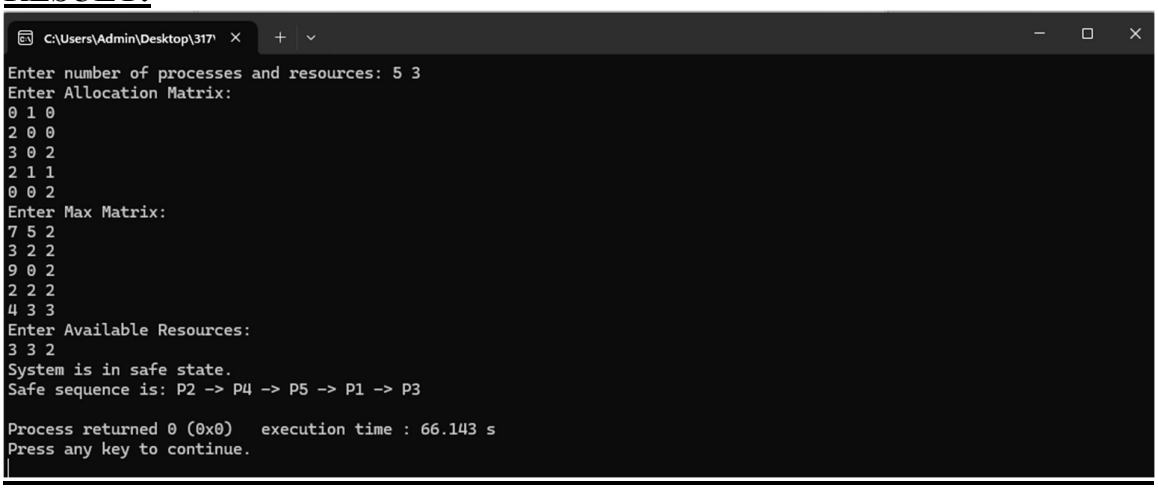
CODE:

```
#include <stdio.h>

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

   int alloc[n][m], max[n][m], avail[m];
   int finish[n], work[m], safeSequence[n];
   int count = 0;

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

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

   printf("Enter Available Resources:\n");
   for (int j = 0; j < m; j++)
      scanf("%d", &avail[j]);

   for (int i = 0; i < m; i++)
      work[i] = avail[i];

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

   int changed;
```

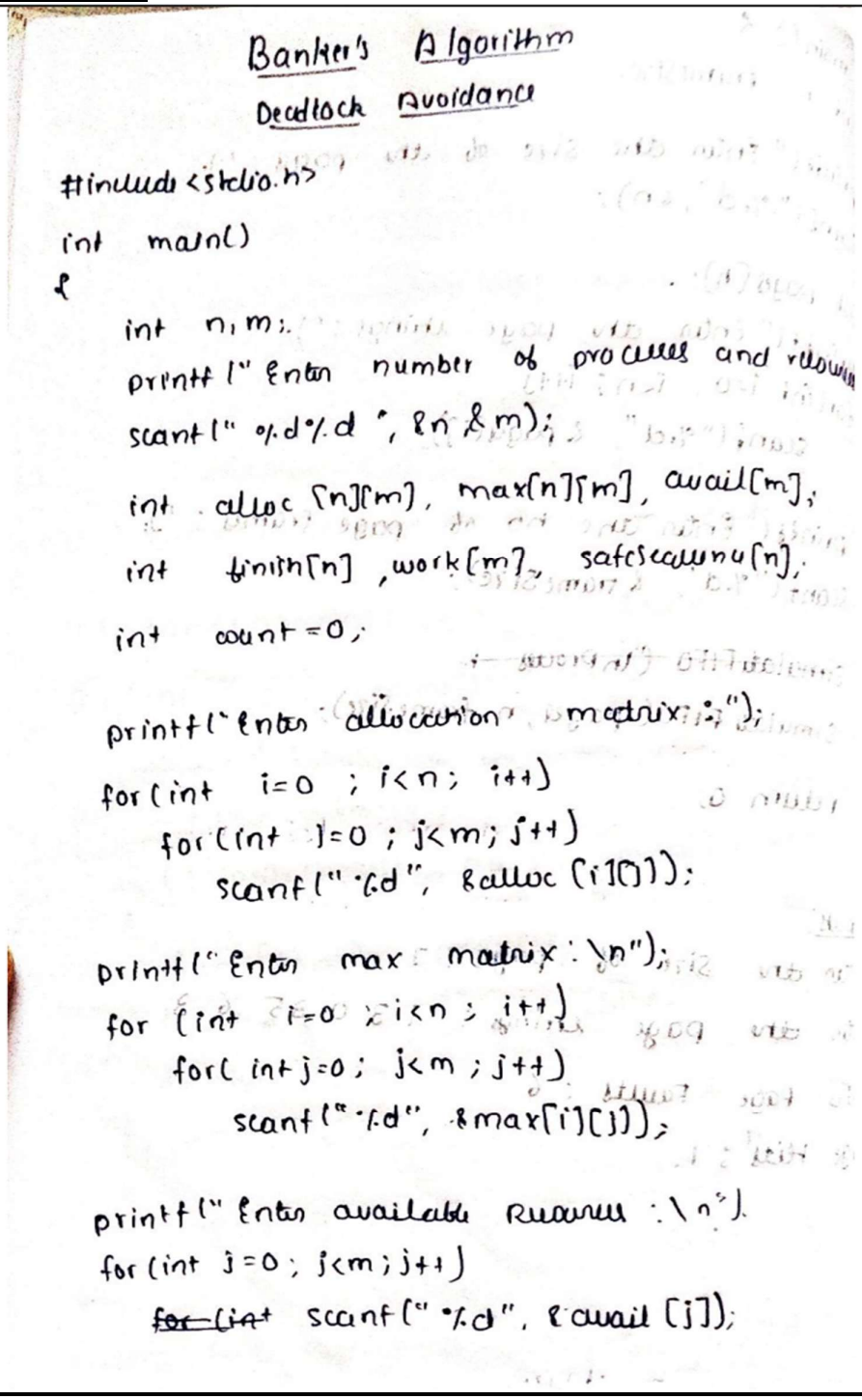
```
do {
   changed = 0;
   for (int i = 0; i < n; i++)
      {
         if (!finish[i])
         {
            int j;
            int canAllocate = 1;
            for (j = 0; j < m; j++)
            {
                int need = max[i][j] - alloc[i][j];
                if (need > work[j])
                {
                      canAllocate = 0;
                      break;
                }
         }

         if (canAllocate)
         {
            for (int k = 0; k < m; k++)
               work[k] += alloc[i][k];
            finish[i] = 1;
            safeSequence[count++] = i;
            changed = 1;
         }
      }
   }
} while (changed);

int deadlock = 0;
for (int i = 0; i < n; i++)
{
      if (!finish[i])
      {
         deadlock = 1;
         printf("Process P%d is in deadlock.\n", i);
```

```
            }
        }

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

    return 0;
}
```

RESULT:



OBSERVATION:

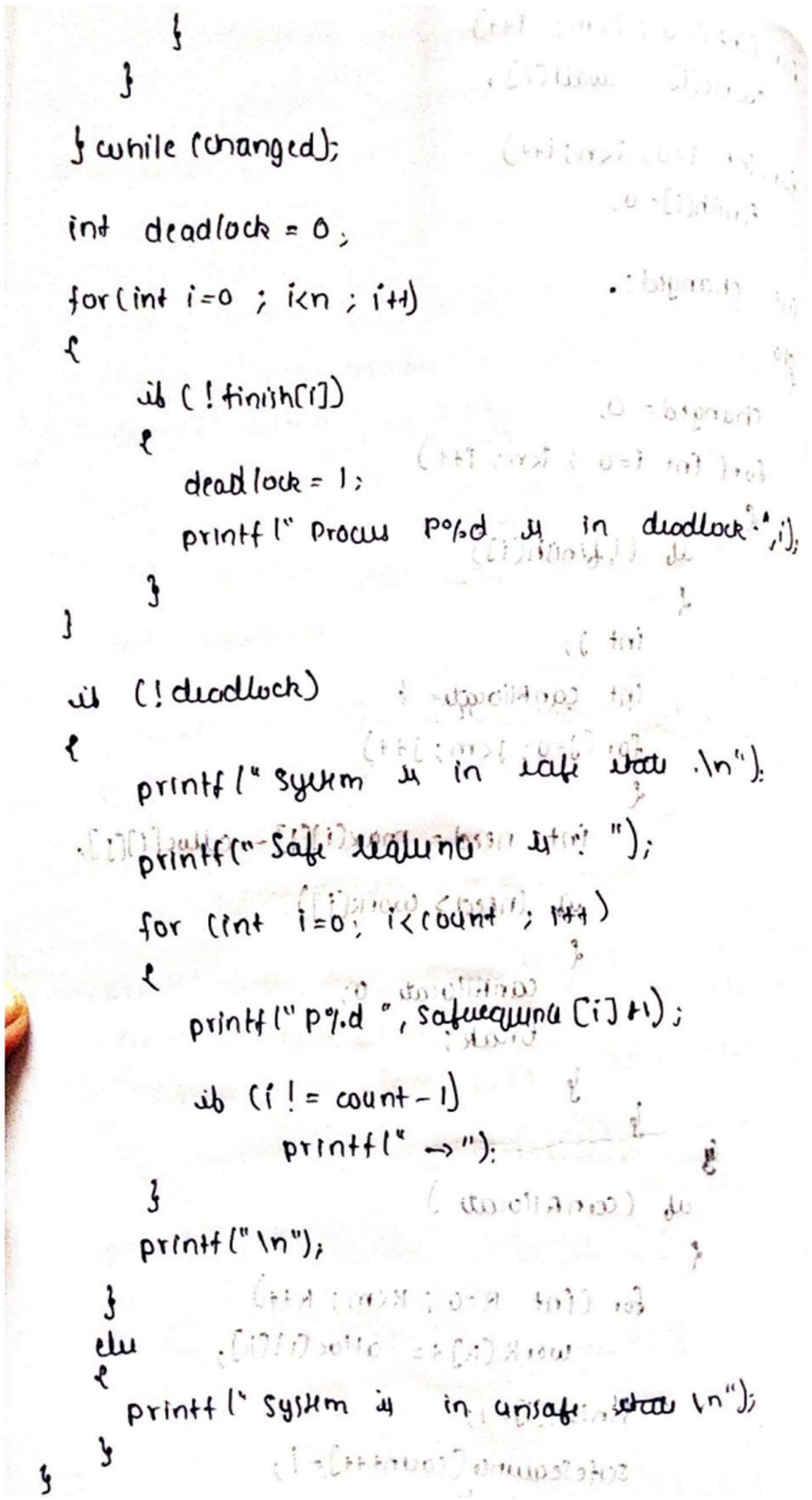


Banker's Algorithm

Deadlock Avoidance

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

```
for (int i=0 ; i<m; i++)
    work[i]= avail[i];

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

int changed;

do
{
    changed = 0;
    for( int i=0 ; i<n; i++ )
    {
        if (!finish[i])
        {
            int j;
            int canAllocate= 1;
            for(j=0; j<m; j++)
            {
                int need = max[i][j] - alloc[i][j];
                if (need > work[j])
                {
                    canAllocate = 0;
                    break;
                }
            }
            if (canAllocate )
            {
                for (int k=0 ; k<m; k++)
                    work[k] += alloc[i][k];

                finish[i]= 1;
                safeSequence[count++]= i;
                changed= 1;
```



```
        }
    }
} while (changed);

int deadlock = 0;
for(int i=0 ; i<n ; i++)
{
    if (!finish[i])
    {
        deadlock = 1;
        printf(" Process P%d is in deadlock.",i);
    }
}
if (!deadlock)
{
    printf(" System is in safe state.\n");
    printf(" Safe sequence is: ");
    for (int i=0; i<count ; i++)
    {
        printf(" P%d ", safequence[i]+1);
        if (i != count-1)
            printf(" -> ");
    }
    printf("\n");
}
else
{
    printf(" System is in unsafe state.\n");
}
}
```

Output:

Enter number of process and resource : 5 3

Enter allocation matrix:

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

Enter the max matrix:

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

Enter available Resources:

3 3 2

System is in safe state.

Safe sequence : P2 → P4 → P5 → P1 → P3.

PROGRAM-8:

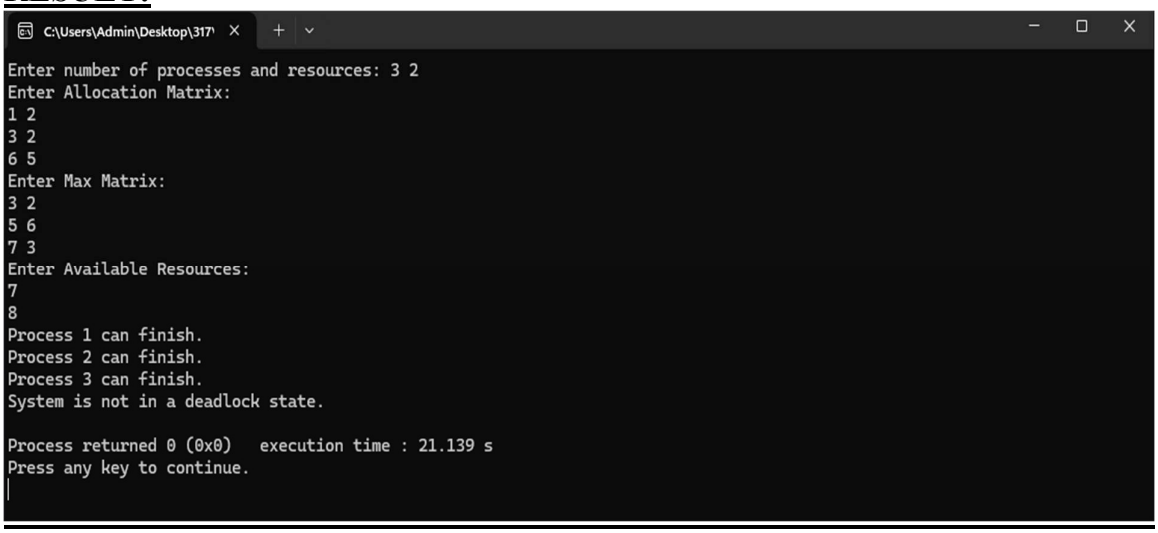
Write a C program to simulate deadlock detection

CODE:

```
#include <stdio.h>

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

   int alloc[n][m], max[n][m], avail[m];
   int finish[n], work[m];
   int count = 0;

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

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

   printf("Enter Available Resources:\n");
   for (int j = 0; j < m; j++)
      scanf("%d", &avail[j]);

   for (int i = 0; i < m; i++)
      work[i] = avail[i];

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

   int changed;
   do {
```

```
        changed = 0;
        for (int i = 0; i < n; i++) {
            if (!finish[i]) {
                int canAllocate = 1;
                for (int j = 0; j < m; j++) {
                    int need = max[i][j] - alloc[i][j];
                    if (need > work[j]) {
                        canAllocate = 0;
                        break;
                    }
                }
                if (canAllocate) {
                    for (int k = 0; k < m; k++)
                        work[k] += alloc[i][k];
                    finish[i] = 1;
                    printf("Process %d can finish.\n", i + 1);
                    changed = 1;
                }
            }
        }
    } while (changed);

    int deadlock = 0;
    for (int i = 0; i < n; i++) {
        if (!finish[i]) {
            deadlock = 1;
            printf("Process %d is in deadlock.\n", i + 1);
        }
    }

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

    return 0;
}
```

RESULT:


C:\Users\Admin\Desktop\317
Enter number of processes and resources: 3 2
Enter Allocation Matrix:
1 2
3 2
6 5
Enter Max Matrix:
3 2
5 6
7 3
Enter Available Resources:
7
8
Process 1 can finish.
Process 2 can finish.
Process 3 can finish.
System is not in a deadlock state.
Process returned 0 (0x0) execution time : 21.139 s
Press any key to continue.

OBSERVATION:

Deadlock detection

```
#include <stdio.h>

int main()
{
   int n, m;
   printf (" Enter the number of process and
                resource : ");
   scanf (" %d %d ", &n, &m);

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

   printf ("Enter max matrix");
   for(int i=0 ; i<n; i++)
      for( int j=0 ; j<n; j++)
         scanf(" %d", & ~~avail~~[j] & max[i][j]);

   printf ("Enter Available Resources :\n");
   for (int j=0; j<m ; j++)
      scanf (" %d", &avail[j]);
```

```
for(int i=0 ; i<n ; i++)
    work[i] = avail[i];

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

int changed;

do
{
    changed = 0;
    for(int i=0 ; i<n; i++){
        if (!finish[i]) {
            int canAllocate = 1;
            for(int j=0 ; j<m; j++)
                int need = max[i][j] - alloc[i][j];
                if (need > work[j])
                {
                    canAllocate = 0;
                    break;
                }
            }
            if (canAllocate)
            {
                for(int k=0 ; k<m; k++)
                    work[k] += alloc[i][k];
                finish[i] = 1;
                printf("Process %d can finish\n", i+1);
                changed = 1;
            }
    }
}
```

```
} while (changed);

int deadlock = 0;
for (int i=0 ; i<n; i++)
{
    if (! finish[i])
    {
        deadlock = 1 ;
        printf(" Process %d is in deadlock \n", i);
    }
}
if (! deadlock)
{
    printf( " System is not in ~~safe~~ deadlock");
}

return 0;
}
```

output

Enter number of process and resources : 3 2

Enter Allocation matrix :

1 2
3 2
6 5

Enter max matrix :

3 2
5 6
7 3

Enter available Resources :

7
8

Process 1 can finish.

Process 2 can finish.

Process 3 can finish.

System is not in a deadlock state.

PROGRAM-9:

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

a) Worst-fit

d) **Best-fit**

e) **First-fit**

CODE:

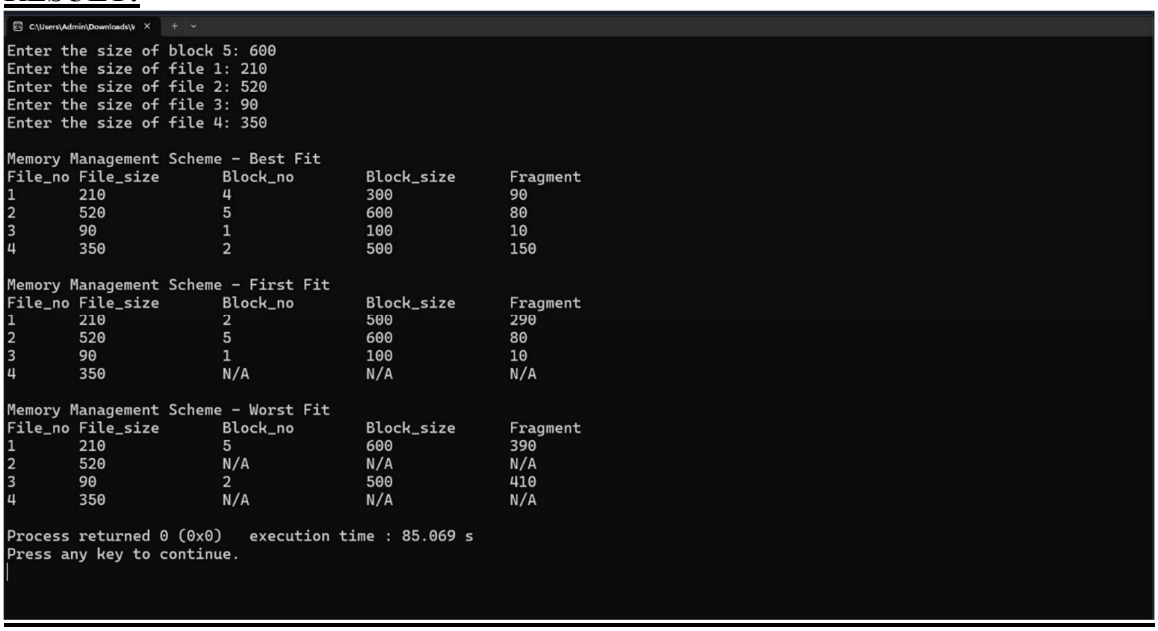
```
#include <stdio.h>

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, 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 (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\tN/A\t\tN/A\t\tN/A\n", files[i].file_no, files[i].file_size);
        }
    }
}

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

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

        for (int j = 0; j < n_blocks; j++) {
            if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
                int fragment = blocks[j].block_size - files[i].file_size;
                blocks[j].is_free = 0;
                printf("%d\t%d\t\t%d\t\t%d\t\t%d\n",
                       files[i].file_no,
                       files[i].file_size,
                       blocks[j].block_no,
                       blocks[j].block_size,
                       fragment);
                allocated = 1;
                break;
            }
        }

        if (!allocated) {
            printf("%d\t%d\t\tN/A\t\tN/A\t\tN/A\n", files[i].file_no, files[i].file_size);
        }
    }
}
```

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

   for (int i = 0; i < n_files; i++) {
      int worst_fit_block = -1, 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\tN/A\t\tN/A\t\tN/A\n", files[i].file_no, files[i].file_size);
      }
   }
}

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

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

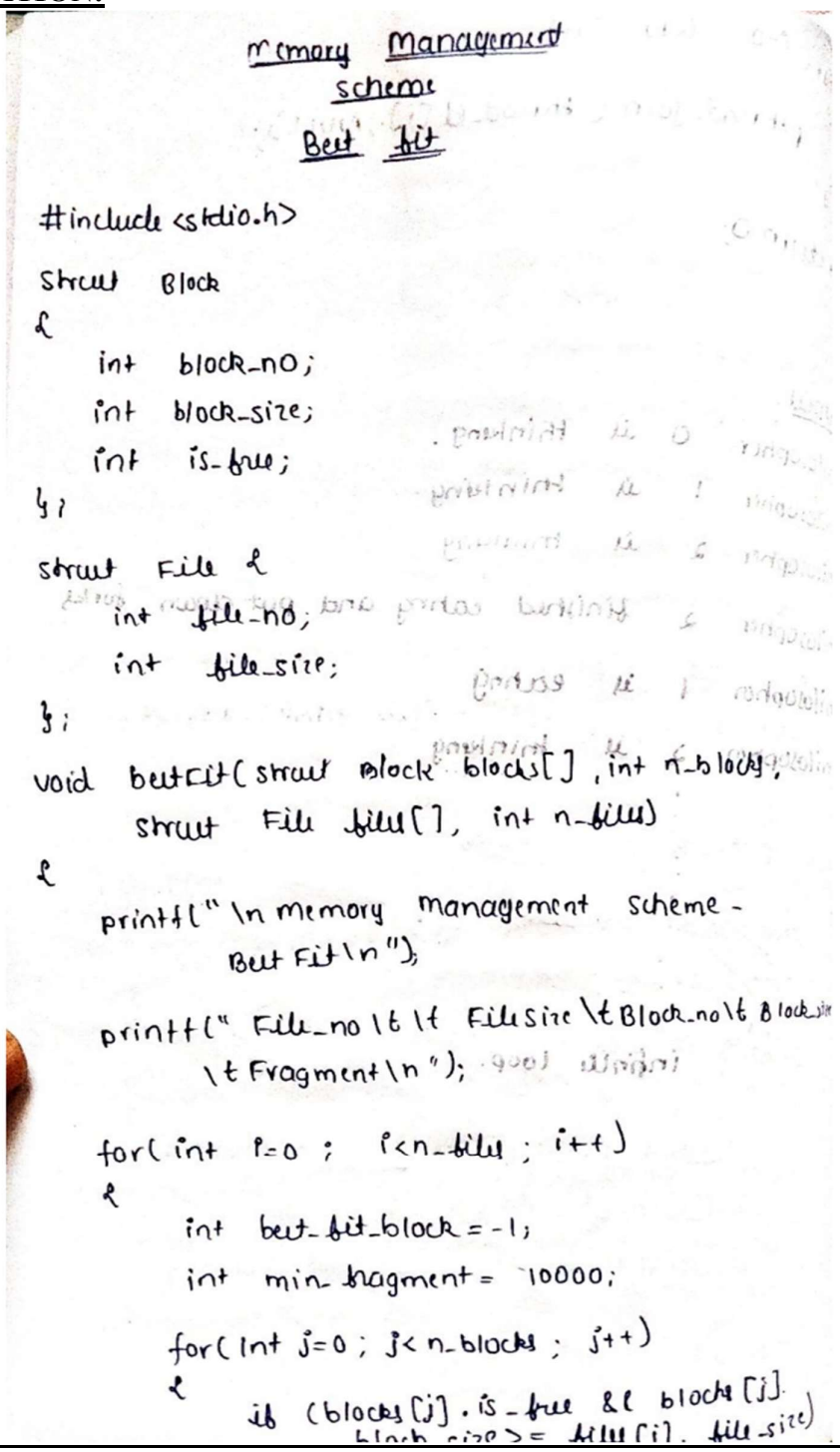
```
    printf("Enter the number of blocks: ");
    scanf("%d", &n_blocks);

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

    struct Block blocks[n_blocks];
    struct File files[n_files];

    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;
    }

    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);
    }

    resetBlocks(blocks, n_blocks);
    bestFit(blocks, n_blocks, files, n_files);

    resetBlocks(blocks, n_blocks);
    firstFit(blocks, n_blocks, files, n_files);

    resetBlocks(blocks, n_blocks);
    worstFit(blocks, n_blocks, files, n_files);

    return 0;
}
```

RESULT:



```
C:\Users\Admin\Downloads\b
Enter the size of block 5: 600
Enter the size of file 1: 210
Enter the size of file 2: 520
Enter the size of file 3: 90
Enter the size of file 4: 350

Memory Management Scheme - Best Fit
File_no File_size       Block_no        Block_size      Fragment
1       210             4               300             90
2       520             5               600             80
3       90              1               100             10
4       350             2               500             150

Memory Management Scheme - First Fit
File_no File_size       Block_no        Block_size      Fragment
1       210             2               500             290
2       520             5               600             80
3       90              1               100             10
4       350             N/A             N/A             N/A

Memory Management Scheme - Worst Fit
File_no File_size       Block_no        Block_size      Fragment
1       210             5               600             390
2       520             N/A             N/A             N/A
3       90              2               500             410
4       350             N/A             N/A             N/A

Process returned 0 (0x0)   execution time : 85.069 s
Press any key to continue.
```

OBSERVATION:



Memory Management scheme

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(" \n memory management scheme -
          Best Fit \n ");

    printf(" File_no \t\t File Size \t Block_no \t Block_size
        \t Fragment \n ");

    for( int i=0 ; i<n_files ; i++ )
    {
        int best_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 (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\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\t%d\t\tN/A\t\t\tN/A\t\tN/A\n",
           files[i].fileno,
           files[i].file_size);
}
}
}
```

```
int main()
{
  int n_blocks, n_files;
  printf("Enter the number of blocks: ");
  scanf("%d", &n_blocks);
  printf("Enter the no of files: ");
  scanf("%d", &n_files);
  struct Block blocks[n_blocks];
  struct File files[n_files];
  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].isfree = 1;
  }
  for( int i=0 ; i<n_files; i++)
  {
     files[i].file_no = i+1;
     printf("Enter size of file %d", i+1);
     scanf("%d", &files[i].file_size);
  }
  for (int i=0; i<n_blocks; i++)
  {
     blocks[i].is_full = 1;
  }
```

```
bestFit(blocks, n_blocks, files, n_files);
return 0;
}
```

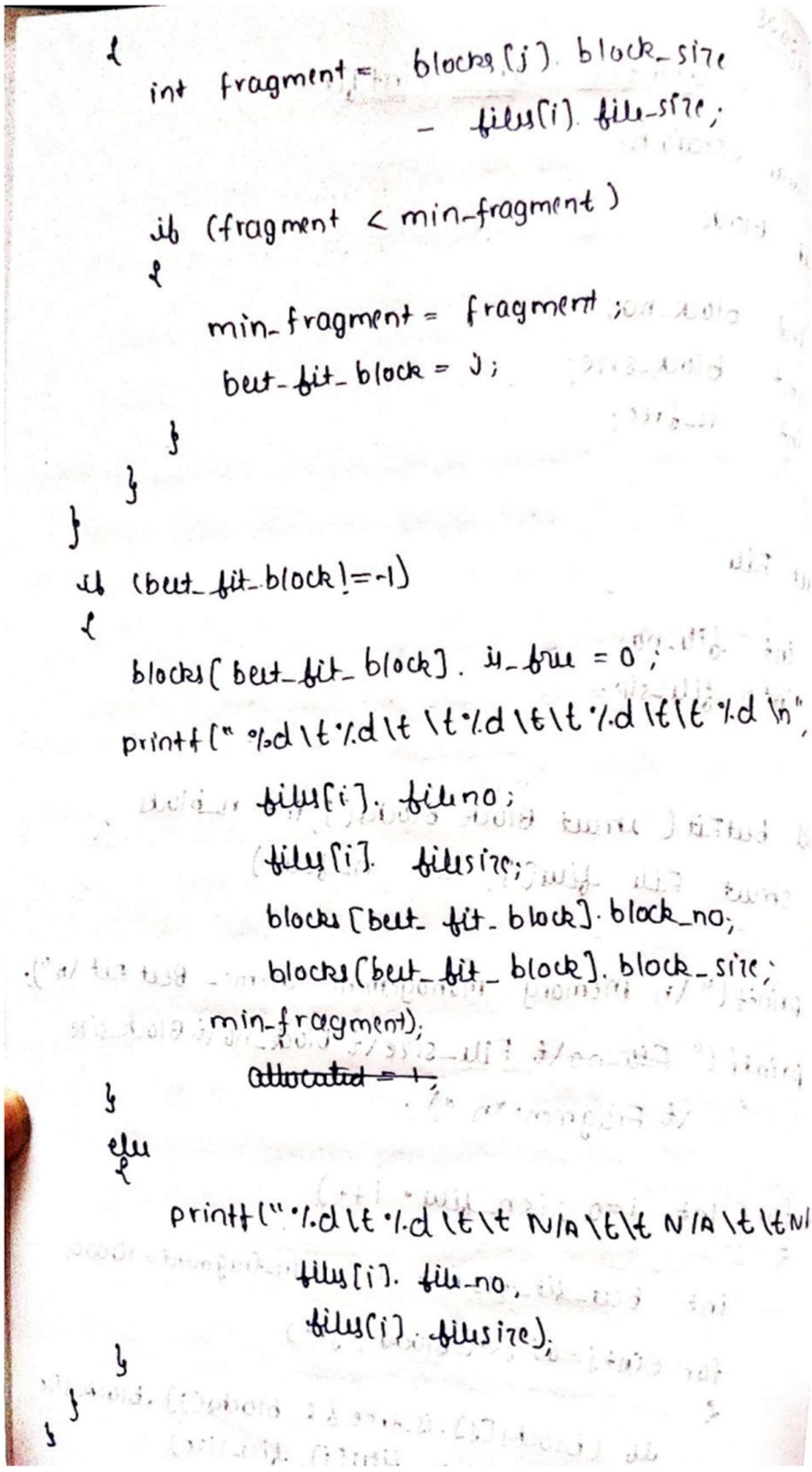
output:

```
Enter no of blocks: 5
Enter the no of files: 4
Enter the size of block 1: 100
Enter the size of block 2: 500
Enter the size of block 3: 300
Enter the size of block 4: 200
Enter the size of block 5: 400
Enter the size of file 1: 212
Enter the size of file 2: 176
Enter the size of file 3: 426
Enter the size of file 4: 404.

Memory management Scheme - Best Fit
File_no  File_size  Block_no  Block_size  Fragment
1        212        3         300         88
2        176        4         200         54
3        426        2         500         74
4        404        N/A       N/A.        N/A.
```

15/05/2025

Worst Fit and 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 bestFit( struct Block blocks[], int n_blocks,
    struct File files[], 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 best_fit_block = -1; 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 (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].fileno;
        files[i].filesize;
        blocks[best_fit_block].block_no;
        blocks[best_fit_block].block_size;
  min_fragment);
        Allocated = 1;
}
else
{
    printf("%d\t%d\t\tN/A\t\tN/A\t\tN/
          files[i].file_no,
          files[i].filesize);
}
}
}
```

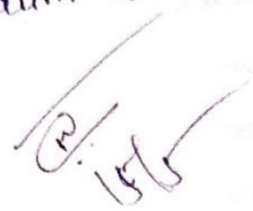
```
void first_fit (struct Block blocks[], int n_blocks,
                struct File files[], int n_files)
{
    printf("Memory management Scheme - First Fit\n");
    printf("File_no\tFile_size\tBlock_no\tBlock_size\tFragment");

    for(int i=0; i<n_files; i++)
    {
        int allocated = 0;
        for(int j=0; j<n_blocks; j++)
        {
            if (blocks[j].isfree && 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\tN/A\t\tN/A\t\tN/A\n",
                files[i].file_no,
                files[i].file_size);
        }
    }
}

void worstFit(struct Block blocks[], int n_blocks,
    struct File files[], int n_files[])
{
    printf("Memory management - Worst Fit\n");
    printf("File_no\tFile_size\tBlock_no\tBlock_size\t
        Fragment\n");

    for(int i=0; i<n_files; i++)
    {
        int worst_fit_block=-1, 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\tN/A\t\tN/A\t\tN/A\n",
           files[i].file_no,
           files[i].file_size);
}
}
}

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

```
int main()
{
    int n_blocks, n_files;
    printf("Enter the number of blocks: ");
    scanf("%d", &n_blocks);

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

    struct Block blocks[n_blocks];
    struct File files[n_files];

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

    resetBlocks(blocks, n_blocks);
    bestFit(blocks, n_blocks, files, n_files);

    resetBlocks(blocks, n_blocks);
    firstFit(blocks, n_blocks, files, n_files);
```

```
resetBlocks (blocks, n_blocks);
worstFit (blocks, n_blocks, files, n_files);

return 0;
}
```



PROGRAM-10:

Write a C program to simulate page replacement algorithms

a) FIFO

d) **LRU**

e) **Optimal**

CODE:

```
#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, 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 (fragment < min_fragment) {
               min_fragment = fragment;
               best_fit_block = j;
            }
         }
      }
```

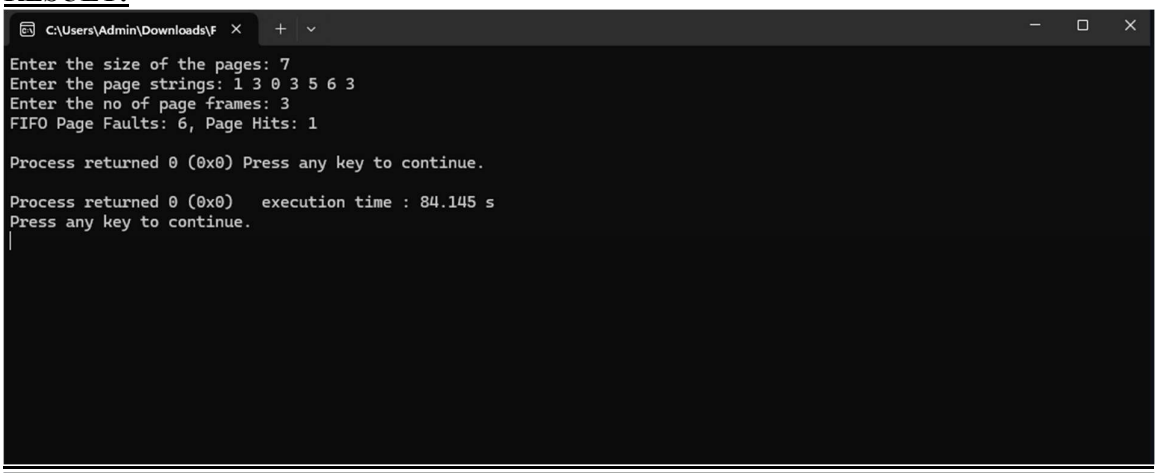
```
        if (best_fit_block != -1) {
            blocks[best_fit_block].is_free = 0;
            printf("%d\t%d\t\t%d\t\t%d\t\t%d\n",
                   files[i].file_no,
                   files[i].file_size,
                   blocks[best_fit_block].block_no,
                   blocks[best_fit_block].block_size,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

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

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

        for (int j = 0; j < n_blocks; j++) {
            if (blocks[j].is_free && blocks[j].block_size >= files[i].file_size) {
                int fragment = blocks[j].block_size - files[i].file_size;
                blocks[j].is_free = 0;
                printf("%d\t%d\t\t%d\t\t%d\t\t%d\n",
                       files[i].file_no,
                       files[i].file_size,
                       blocks[j].block_no,
                       blocks[j].block_size,
                       fragment);
                allocated = 1;
                break;
            }
        }

        if (!allocated) {
```

```
            printf("%d\t%d\t\tN/A\t\tN/A\t\tN/A\n", files[i].file_no, files[i].file_size);
        }
    }
}

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

    for (int i = 0; i < n_files; i++) {
        int worst_fit_block = -1, 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\tN/A\t\tN/A\t\tN/A\n", files[i].file_no, files[i].file_size);
        }
    }
}

void resetBlocks(struct Block blocks[], int n_blocks) {
```

```
    for (int i = 0; i < n_blocks; i++) {
        blocks[i].is_free = 1;
    }
}

int main() {
    int n_blocks, n_files;

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

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

    struct Block blocks[n_blocks];
    struct File files[n_files];

    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;
    }

    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);
    }

    resetBlocks(blocks, n_blocks);
    bestFit(blocks, n_blocks, files, n_files);

    resetBlocks(blocks, n_blocks);
    firstFit(blocks, n_blocks, files, n_files);

    resetBlocks(blocks, n_blocks);
```

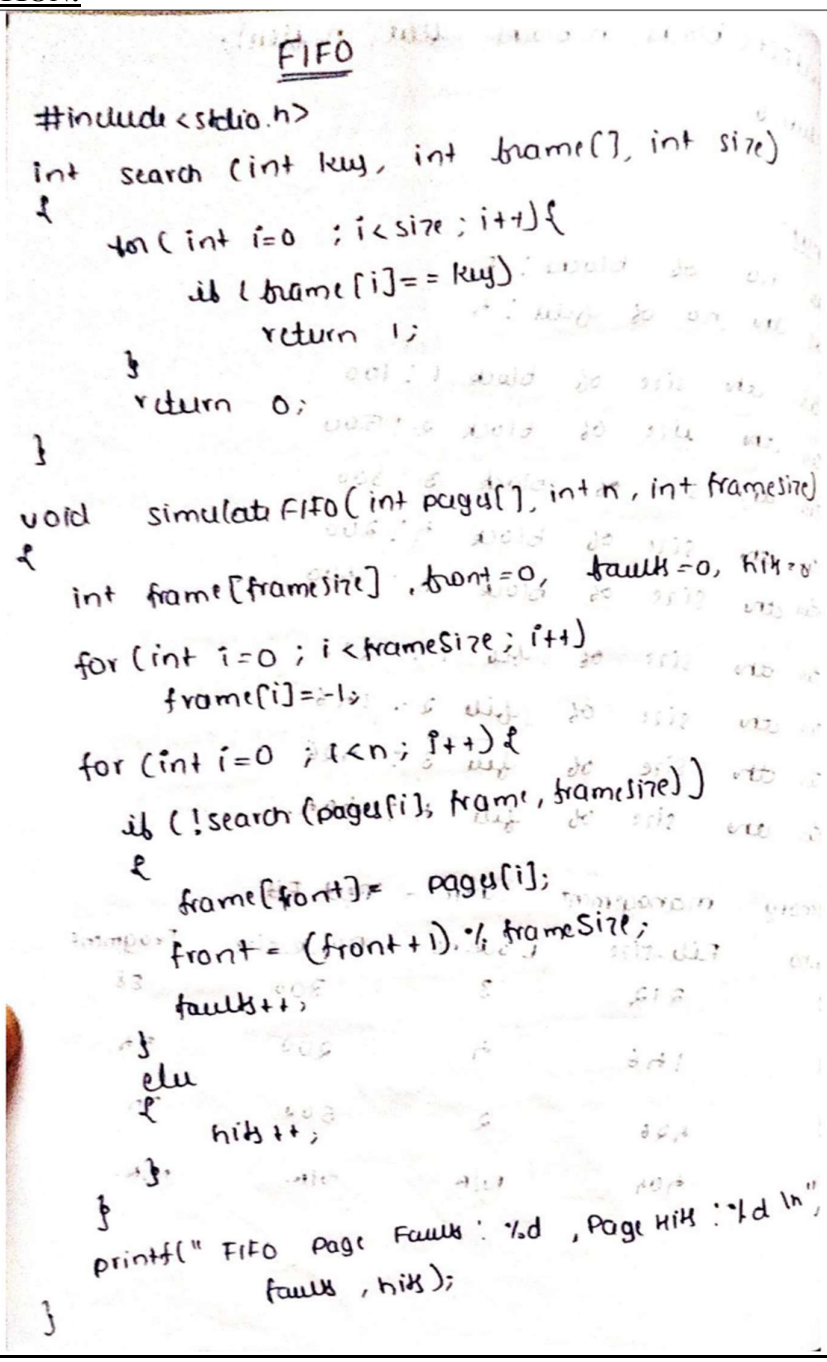
```
    worstFit(blocks, n_blocks, files, n_files);

    return 0;
}
```

RESULT:




```
Enter the size of the pages: 12
Enter the page strings: 7 0 1 2 0 3 0 4 2 3 0 3
Enter the no of page frames: 3
Optimal Page Faults: 7, Page Hits: 5
LRU Page Faults: 9, Page Hits: 3
```

OBSERVATION:



FIFO

```
#include <stdio.h>
int search (int key, int frame[], int size)
{
    for ( int i=0 ; i<size ; i++){
        if ( frame[i]== key)
            return 1;
    }
    return 0;
}
void simulateFIFO ( int pages[], int n , int framesize)
{
   int frame[framesize] , front=0, faults=0, hits=0
   for (int i=0 ; i <framesize ; i++)
       frame[i]=-1;
   for (int i=0 ; i<n ; i++) {
      if (!search (pages[i], frame, framesize) )
      {
         frame[front]= pages[i];
         front = (front+1) % framesize;
         faults++;
      }
      else
      {
          hits++;
      }
   }
   printf(" FIFO Page Faults : %d , Page Hits : %d \n",
              faults , hits);
}
```

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

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

    int pages[n];
    printf("Enter the page strings :");
    for(int i=0; i<n; i++)
        scanf("%d", &pages[i]);

    printf("Enter the no of page frame : ");
    scanf("%d", &frameSize);

    ~~simulateFIFO("InProcess");~~
    simulateFIFO(pages, n, frameSize);

    return 0;
}
```

Outputs:

Enter the size of pages : 7

Enter the page strings : 1 3 0 3 5 6 3

FIFO Page Faults : 6

Page Hits : 1.

LRU and optimal

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

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

```
    if (j==n)
        return i;

  }
  return (pos==-1)? 0 : pos;

}

void simulateLRU(int pagu[], int n, int framesize)
{
    int frame[framesize], time[framesize], count=0, hit=0,
    for(int i=0 ; i<framesize ; i++)
    {
        frame[i]= -1;
        time[i]= 0;
    }

    for(int i=0 ; i<n ; i++)
    {
        int pos = search(pagu[i], frame, framesize);
        if (pos==-1)
        {
            int least= 0;
            for( int j=1 ; j<framesize ; j++)
            {
                if (time[j] < time[least])
                    least= j ;
            }
            frame[least]= pagu[i];
            time [least]= i;
            count++;
```

```
        else
        {
            hits ++;
            time[pos] = i;
        }
    }
    printf(" LRU Page faults : %d , Page Hits : %d\n",
                                    count, hits);
}
void simulateOptimal (int pages , int n , int framesize)
{
    int frame [framesize] , count=0 , hits=0;
    for(int i=0 ; i< framesize ; i++)
        frame[i] = -1;
    for(int i=0 ; i<n ; i++)
    {
        if (search (pages[i], frame, framesize) == -1)
        {
            int indx = -1;
            for( int j= 0 ; j< framesize ; j++)
            {
                if (frame[j] == -1)
                {
                    indx = j;
                    break;
                }
            }
            if (indx != -1)
            {
                frame[indx] = pages[i];
```

```
else
{
    int replaceIndex = findOptimal(pages, frame, n, i+1,
                                   frameSize);

    frame[replacement Index] = pages[i];
}

count ++

}
else
{
    hits ++;
}
}
printf("Optimal Page faults : %d, Page hits %d \n",
                count, hits);
}
int main()
{
    int n, frameSize;
    printf("Enter the size of the pages :");
    scanf("%d", &n);

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

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

    simulateOptimal(page, n, framesize);
    simulateLRU(page, n, framesize);

    return 0;
}
```

Output:

```
Enter the size of pages: 12
Enter the page strings: 7 0 1 2 0 3 0 4 2 3
Enter the no of frames: 3
Optimal Page Faults: 7, Page Hits: 5
LRU Page Faults: 9, Page Hits: 3
```