VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

OPERATING SYSTEMS

Submitted by

TANAY S HUDDAR (1WA23CS007)

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 TANAY S HUDDAR(1WA23CS007), 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.

Dr.Sema Patil
Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

Index Sheet

Course Outcomes

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

INDEX

Name Tanay Std — Sec —

Roll No. 007 Subject OS School/College — BMS

Sl. No.	Date	Title	Page No.	Teacher Sign/ Remarks
01	6/03/25	First come first Serve	01	
02	6/03/25	SJFC	10	
03	6/03/25	SRTF		
04	7/3/25	Priority (Prempitive)	10	
05	7/3/25	Priority (Non Prempitive)		
06	8/3/25	Earliest Deadline	10	
07	8/3/25	Rate monotonic		
08	9/3/25	~~Dr~~ Producer Consumer	10	
09	9/3/25	Dinning Philosopher		
10		Deadlock allocation	10	
11		Deadlock Detection		
12		~~Memory~~ MU Multilevel Queuing	10	
13		Round Robin		
14		MMU (BestFit)	10	
15		FIFO	10	
16		LRU		
17		Optimal		

Completed

Program -1

Question:

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

→FCFS

→ SJF (pre-emptive & Non-preemptive)

Code:

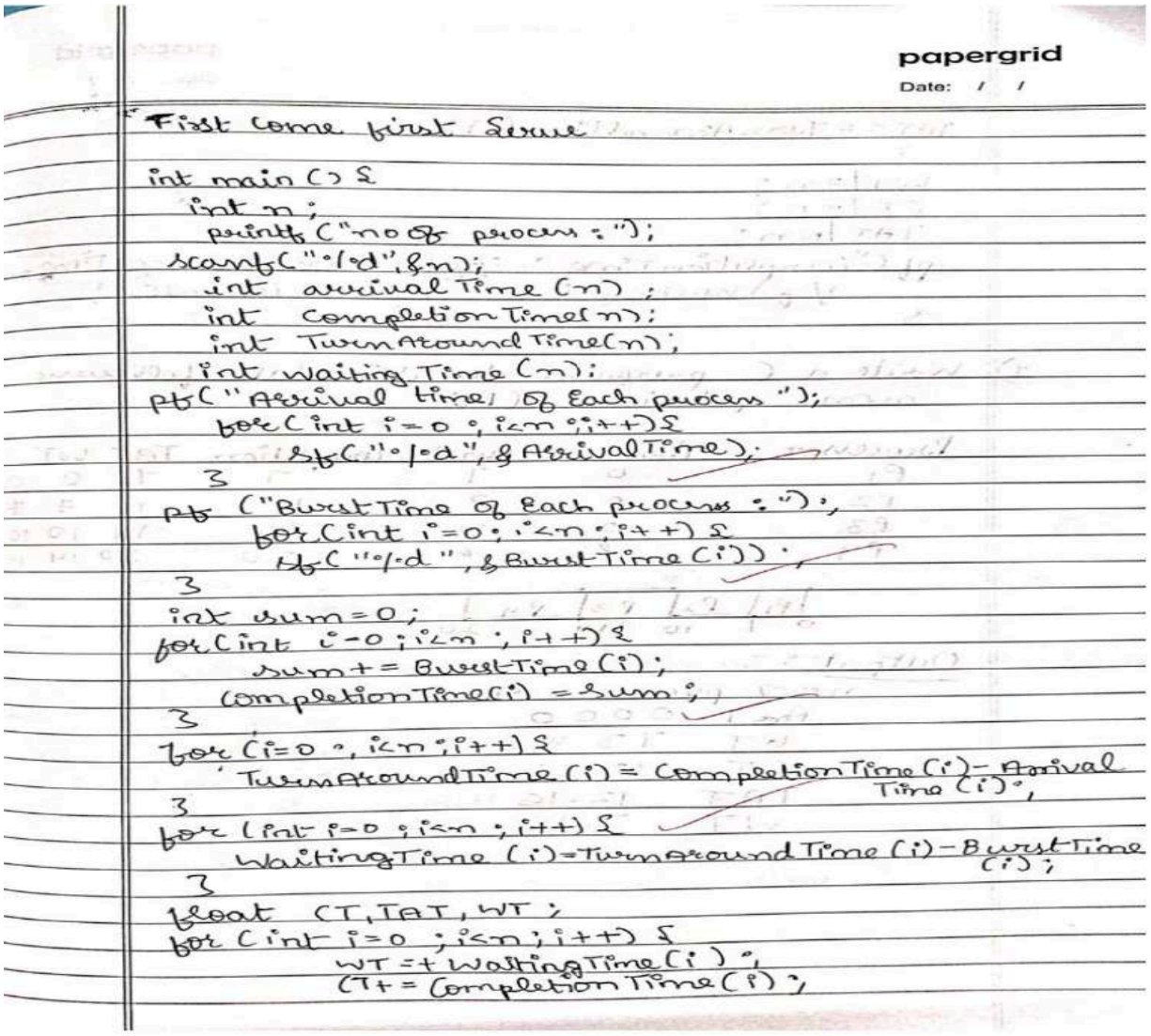
=>FCFS:

```
#include<stdio.h>

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

void main()
{
   int n,c=0;
   printf("Enter number of processes: ");
```

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

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

    printf("FCFS scheduling:\n");
    printf("PID\tAT\tBT\tCT\tTAT\tWT\n");
    for(int i=0;i<n;i++)
       printf("%d\t%d\t%d\t%d\t%d\t%d\n",proc_id[i],at[i],bt[i],ct[i],tat[i],wt[i]);

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

Result:

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

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



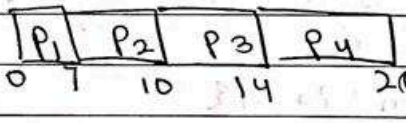
```
papergrid
Date: / /

First come first Serve

int main () {
  int n;
  printf("no of process : ");
scanf("%d",&n);
  int arrivalTime(n);
  int completionTime(n);
  int TurnAroundTime(n);
  int waitingTime(n);
ptf("Arrival time of Each process ");
  for(int i=0;i<n;i++){
  scf("%d",&ArrivalTime);
  }
ptf ("BurstTime of Each process : ");
    for(int i=0;i<n;i++){
      ptf("%d",&BurstTime(i));
}
int sum=0;
for(int i=0;i<n;i++){
    sum+=BurstTime(i);
  completionTime(i)=sum;
}
for(i=0;i<n;i++){
  TurnAroundTime(i)=completionTime(i)-ArrivalTime(i);
}
for(int i=0;i<n;i++){
  WaitingTime(i)=TurnAroundTime(i)-BurstTime(i);
}
float CT,TAT,WT;
for(int i=0;i<n;i++){
        WT+=WaitingTime(i);
        CT+=completionTime(i);
```

Date: / /

```
TAT += TurnAroundTime(i);
}
WT/=n;
CT/=n;
TAT/=n;
Pf("CompletionTime : %f\nTurn around time :
   %f\nWaiting time : %f", CT, TAT, WT);
}
```

1) Write a C program to stimulate following non primitive CPU.

Processor	Arrival	Burst	Completion	TAT	WT
P1	0	7	7	7	0
P2	0	3	10	10	7
P3	0	4	14	14	10
P4	0	6	20	20	14



Output :-

```
no of process  4
AT  0 0 0 0
BT  7 3 4 6
CT  12.75
TAT 12.75
WT  7.75
```

SJF(NON PREEMPITIVE)

CODE:

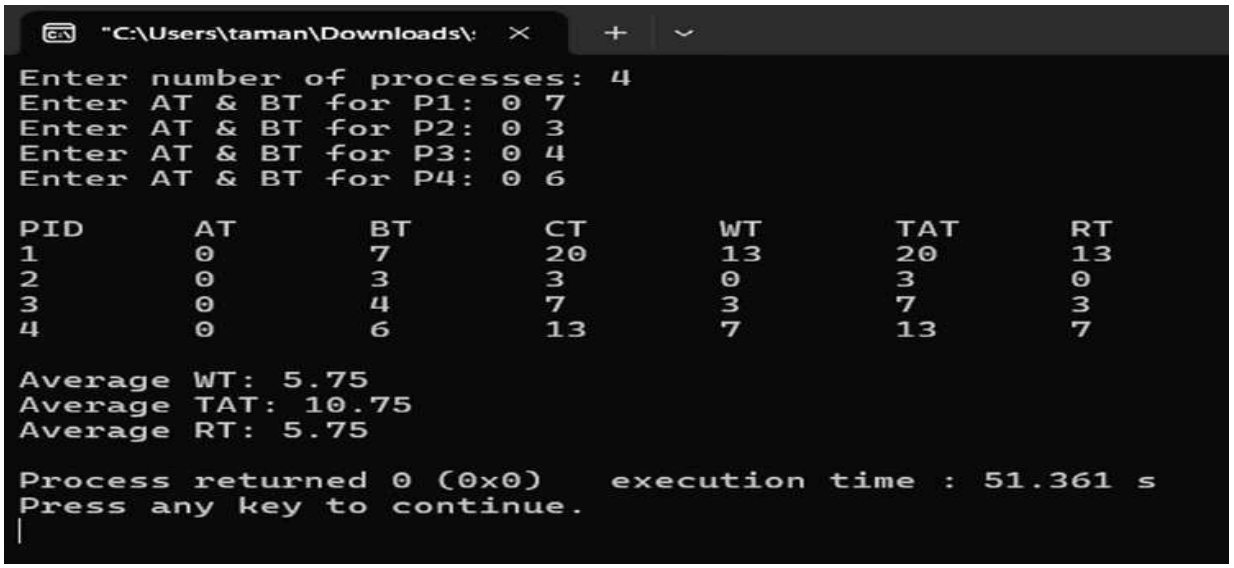
```
#include <stdio.h>

struct Process {

        int pid, bt, at, wt, tat, rt, ct;

};

void sortByArrival(struct Process p[], int n) {

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

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

        if (p[j].at > p[j + 1].at) {

                struct Process temp = p[j];

                p[j] = p[j + 1];

                p[j + 1] = temp;

        }

           }

        }

}

void sjfScheduling(struct Process p[], int n) {

        sortByArrival(p, n);

        int completed = 0, currentTime = 0;
```

```
        int totalWT = 0, totalTAT = 0, totalRT = 0;
while (completed < n) {
        int minIndex = -1, minBT = 9999;
        for (int i = 0; i < n; i++) {
        if (p[i].at <= currentTime && p[i].bt < minBT && p[i].tat == 0) {
                minBT = p[i].bt;
                minIndex = i;
        }
        }
        if (minIndex == -1) {
        currentTime++;
        continue;
        }
        p[minIndex].wt = currentTime - p[minIndex].at;
        p[minIndex].tat = p[minIndex].wt + p[minIndex].bt;
        p[minIndex].rt = p[minIndex].wt;
        p[minIndex].ct = currentTime + p[minIndex].bt;
        currentTime += p[minIndex].bt;
        completed++;
```

```
        totalWT += p[minIndex].wt;
        totalTAT += p[minIndex].tat;
        totalRT += p[minIndex].rt;
        }
    printf("\nPID\tAT\tBT\tCT\tWT\tTAT\tRT\n");
        for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].ct, p[i].wt, p[i].tat, p[i].rt);
        }
        printf("\nAverage WT: %.2f", (float)totalWT / n);
        printf("\nAverage TAT: %.2f", (float)totalTAT / n);
        printf("\nAverage RT: %.2f\n", (float)totalRT / n);
}
int main() {
        int n;
        printf("Enter number of processes: ");
        scanf("%d", &n);
        struct Process p[n];
        for (int i = 0; i < n; i++) {
        printf("Enter AT & BT for P%d: ", i + 1);
```

```
        scanf("%d %d", &p[i].at, &p[i].bt);
        p[i].pid = i + 1;
        p[i].wt = p[i].tat = p[i].rt = p[i].ct = 0;
        }
        sjfScheduling(p, n);
        return 0;
}
```

Output:



2) Write a C program to shortest job first (Non-premitive)

```
#include <stdio.h>
#include <stdlib.h>
Struct Process {
    int id;
    int bt;
    int at;
    int ct;
    int tat;
    int wt;
};
int compareArrivalTime(const void *a, const
void *b) {
  return ((struct Process*)a)->at - ((struct
Process*)b)->at;
}
void calculateTimes(Struct Process process[], int n
{
int time=0;
int completed=0;
while (completed<n) {
    int shortest=-1;
    int min_burst=100000;
for (int i=0; i<n; i++) {
        if (processes(i).at <= time && processes(i)
.ct==0) {
            if (processes(i).bt < min_burst) {
```

```
    }
  }
}
if (shortest == -1) {
    time++;
  } else {
    processes[shortest].ct = time + processes[
shortest].ct - processes[shortest].at;
    processes[shortest].wt = processes[shortest].
```

```
      time = processes[shortest].ct;
      completed++;
    }
  }
}
void calculateAvg(struct Process processes[],
    int n) {
  int total_wt = 0, total_tat = 0;

  for (int i = 0; i < n; i++) {
    total_wt += processes[i].wt;
    total_tat += processes[i].tat;
  }
  printf("\n avg wt = %.2f", (float)total_
                                    wt/n);
printf("\n avg tat = %.2f", (float)totat_tat/n);

int main() {
  int n;
  pf("no of processes:");
  scanf("%d", &n);
  struct processes processes[n];
```

```
pf ("Enter bt & at : \n");
  for (int i=0 ; i<n ; i++) {
     processes(i). id = i+1 ;
    pf ("bt %d :", i+1) ;
scanf ("%d", &processes(i). bt ) ;
pf ("at %d :", i+1) ;
   sf("%d", & processes(i). at ) ;
      processes(i). ct = 0 ;
3
 qsort (processes , n , sizeof (struct processes),
compare Arrival Time ) ;
  Calculate Times (processes, n) ;
Calculate Avg (processes, n) ;
   return 0 ;
3
```

Output :-

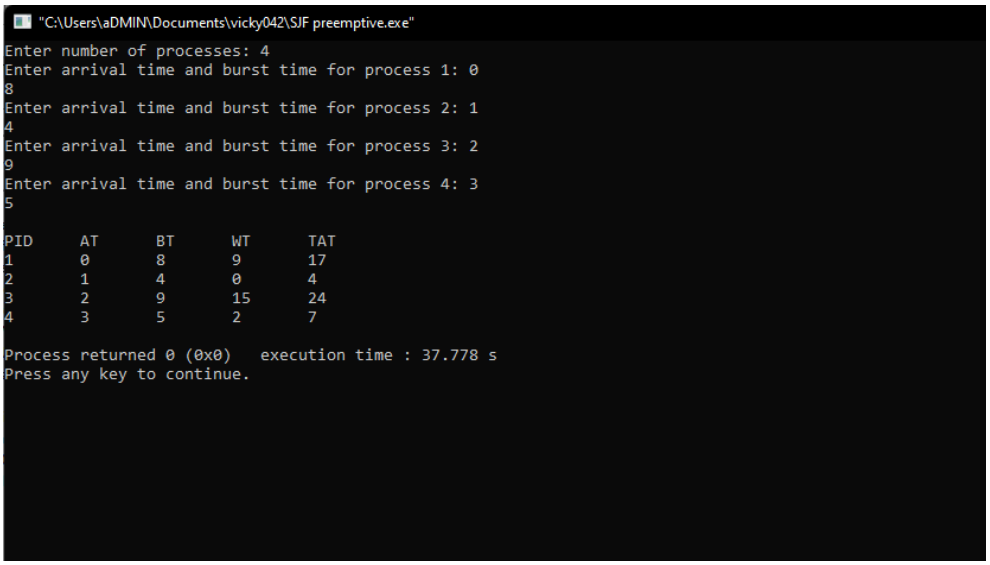
no of process : 4
BT 1 : 7
AT : 0
BT 2 : 3
AT 2 : 8
BT 3 : 4
AT 3 : 3
BT 4 : 6
AT 4 : 5
Avg WT : 4.00
Avg TAT : 9.00

SJFC(PREMETIVE)

CODE:

```
#include <stdio.h>

#define MAX 10

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

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

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

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

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

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

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

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

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

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

    sjf_preemptive(p, n);

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

    return 0;
}
```


"C:\Users\aDMIN\Documents\vicky042\SJF preemptive.exe"
Enter number of processes: 4
Enter arrival time and burst time for process 1: 0
8
Enter arrival time and burst time for process 2: 1
4
Enter arrival time and burst time for process 3: 2
9
Enter arrival time and burst time for process 4: 3
5
PID AT BT WT TAT
1 0 8 9 17
2 1 4 0 4
3 2 9 15 24
4 3 5 2 7
Process returned 0 (0x0) execution time : 37.778 s
Press any key to continue.

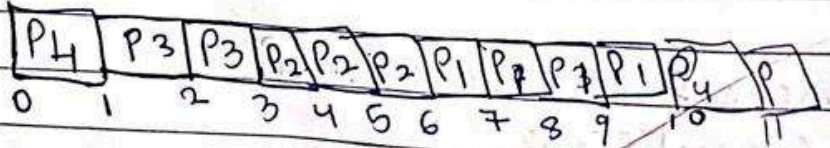
⊙ SJFC (Prempitive)

```
#include <stdio.h>

struct Process {
  int id; int BT; int AT; int CT; int ST;
};
int compareAT(const void *a, void *b) {
  return ((struct Process*))a->AT - st(b)AT
}
int compareRT(const void *a, const void *b){
return ((struct Process*)a)->RT-((structProcess),
        compareAT);

while(completedProcesses < numProcesses) {
  int index = -1;

for(int i = 0; i< numProcesses ;;i++) {
  if( processes(i)-at <=CT&& procs(i).RT>0) {
    if(minindex ==-1)
      minIndex = i ;
  }
}
if(minIndex == -1) {
  CT;
  continue;

processes[minIndex].RT--;
  currentTime++;

if(processes(minIndex).RT==0) {
  processes(minIndex).CT = currentTime ;
  completedProcesses++ ;
```

```
    pf("Processes Completed");
}
}

int main() {
    struct Process (max_Processes);
    int numProcesses;

for (int i=0; i<numProcesses; i++){
    processes(i).id = i+1;
pf("- Enter AT");
sf("%d", &procen(i).BT);
procen(i).RT = processes(i).BT;
}
preemptiveSJF (processes, numProcesses);

return 0;
}
```

Problem :-

			CT	TAT	WT	RT
P1	0	4	17	17	9	0
P2	1	3	5	4	0	0
P3	0	2	26	24	15	15
P4	0	1	10	7	2	2



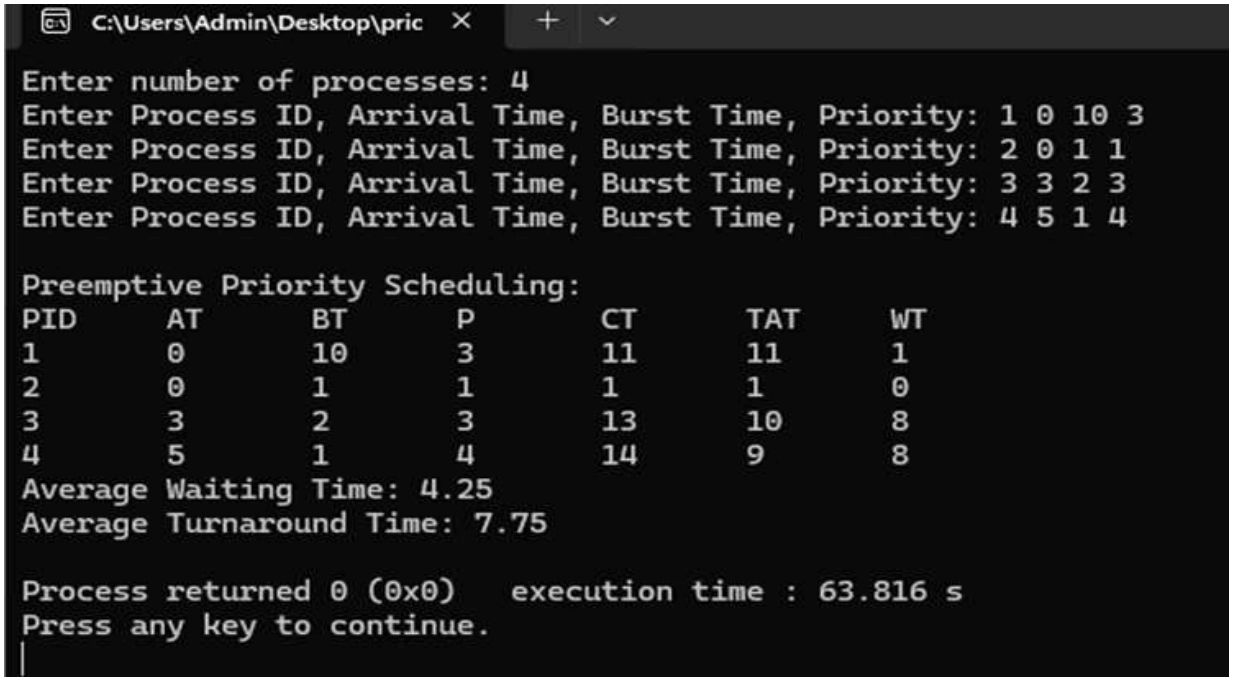
PRIORITY SCHELDUING(PREEMITIVE)

CODE:

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

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

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

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

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

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

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

```
        }

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

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

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

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

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

int main() {
    int n;
    printf("Enter the number of processes: ");
```

```
    scanf("%d", &n);
    struct Process p[n];

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

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



④ 4) write a C program to perform priority (prempetive)

```
#include <stdio.h>

typedef struct {
    int pid, arT, BT, R } process;

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

void sort-by-aT (process) {
  for(int i=0; i<n-1; i++) {
      if (process(j) : arrival time )
  (process(i).priority > pr(j+1).AT {{
      swap (& process(j), & process(j-1);
  }
  }
}

void priority-prempting (process) {
  int completed = 0, CT = 0, MT=0;
}
for(int i=0; i<n; i++) {
    process(i).rT = processes(i).BT;
}
while (completed != n) {
    min priority = 999;
    selected = -1;
```

```
}
while (i=0 ; i<n ; i++){
    if (process(i).AT<= CT && process(i).
    min-priority = process(i).priority ; RT>0){
    selected = i ;
    }
}
}
if( selected == -1){
    CT++;
    continue ;
}
processes(selected).RT--;
  current-Time ++;

if (processes (selected).RT == 0){
    completed++;
    processes (selected).Turnaround_time
        = processes (selected).completion_time
        (selected).arrival time;
    processes (selected).waiting-time =
                processes (selected).
                    TA.-processes
                    (selected).BT;
processes(selected).WT = processes (selected).
                    TA - processes
                    (selected).BT;
}
}
```

Problem :-

No of process : 7
Enter BT & AT & PT
BT1 : 8
AT1 : 0
PT1 : 3
BT2 : 2
AT2 : 1
BT3 : 4
AT3 : 4
BT : 5
AT5 : 5
AT5 :
BT6 : 1
AT7 : 1
AT7 : 7
BT7 : 1

Output :-

Avg TAT : 13.71
Avg WT : 9.86

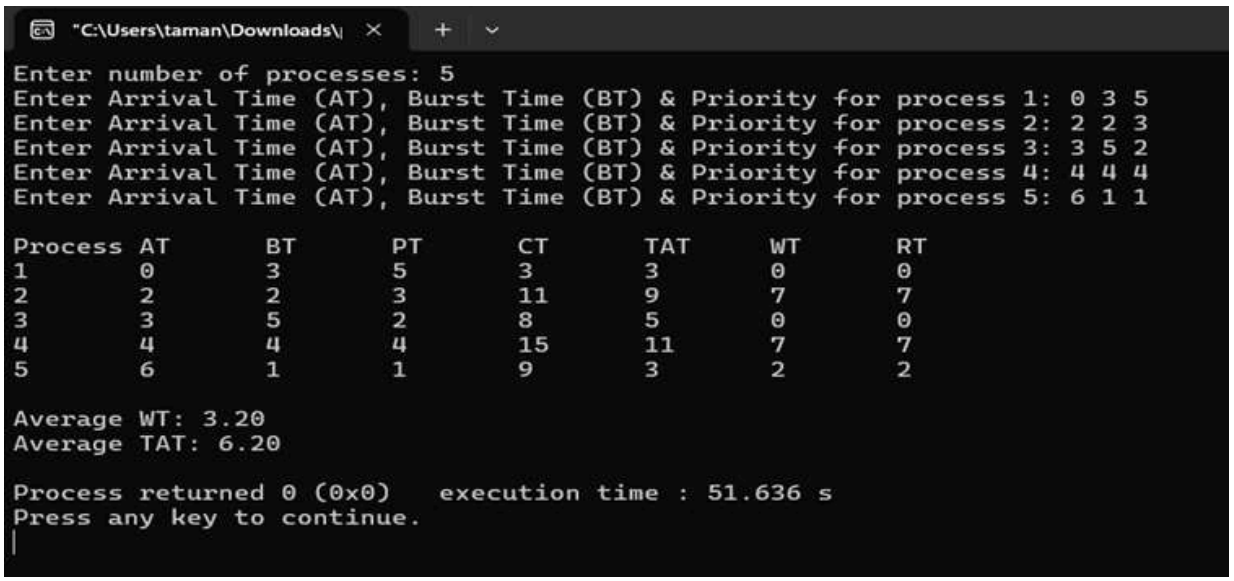
PRIORITY (NON PREEMITIVE)

CODE:

```
#include <stdio.h>

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

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

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

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

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

```
                highestPriority = p[i].pr;
                idx = i;
            }
        }
    }

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

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

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

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

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

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

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



3) write a C program of Priority Scheldulin of non primitive.

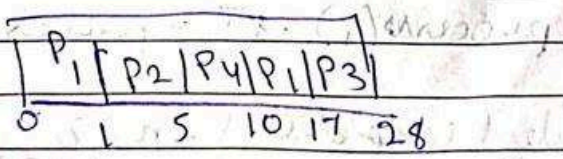
```
#include <stdlib.h>
#include <stdio.h>

int main () {
    int n, i, j;
int burst(20), priority (20), process(20)
int wait(20), turnaround (20),
  completion (20);
int total-wait = 0, total-tat = 0;

printf ("Enter the no of processes:");
sf ("%d", &n);

pf("Enter burst time & priority for each
    process %n");
for (i=0; i<n; i++){
    pf ("Process %d:", i+1);
sf ("%d %d", &burst (i), &priority (i));
process(i) = i+1;
}
for (i=0; i<n; i++) {
    if(priority (i) < priority (j)) {
int temp = priority (i);
    priority(i) = priority (j);
    priority (j) = temp;
    temp = burst(i);
  burst(i) = burst(j);
  burst(i) = temp;
  temp = process;
```

```
wait(0) = 0;
completion(0) = burst(0);
turnaround(0) = completion(0);
total-turnaround = turnaround(0);
for(i=0; i<n; i++) {
  wait(i) = completion(i-1)
  completion(i) = wait(i);
pf("\n Process\t");
  for(i=0; i<n; i++) {
  pf(" Process(i), burst(i), priority(i),
      wait(i), turnaround(i));
}
printf("\n Average time : %.2f "(float)
   total-wait \n);
printf("\n Average Turnaround Time : %.2f\n",
   (float) total-turnaround \n);
  return 0;
}
```

Output :-

P	AT	BT	CT	TAT	WT	RT
P_1	0	8	17	17	9	0
P_2	1	4	5	4	0	0
P_3	2	9	26	24	15	15
P_4	3	5	10	7	2	2

650



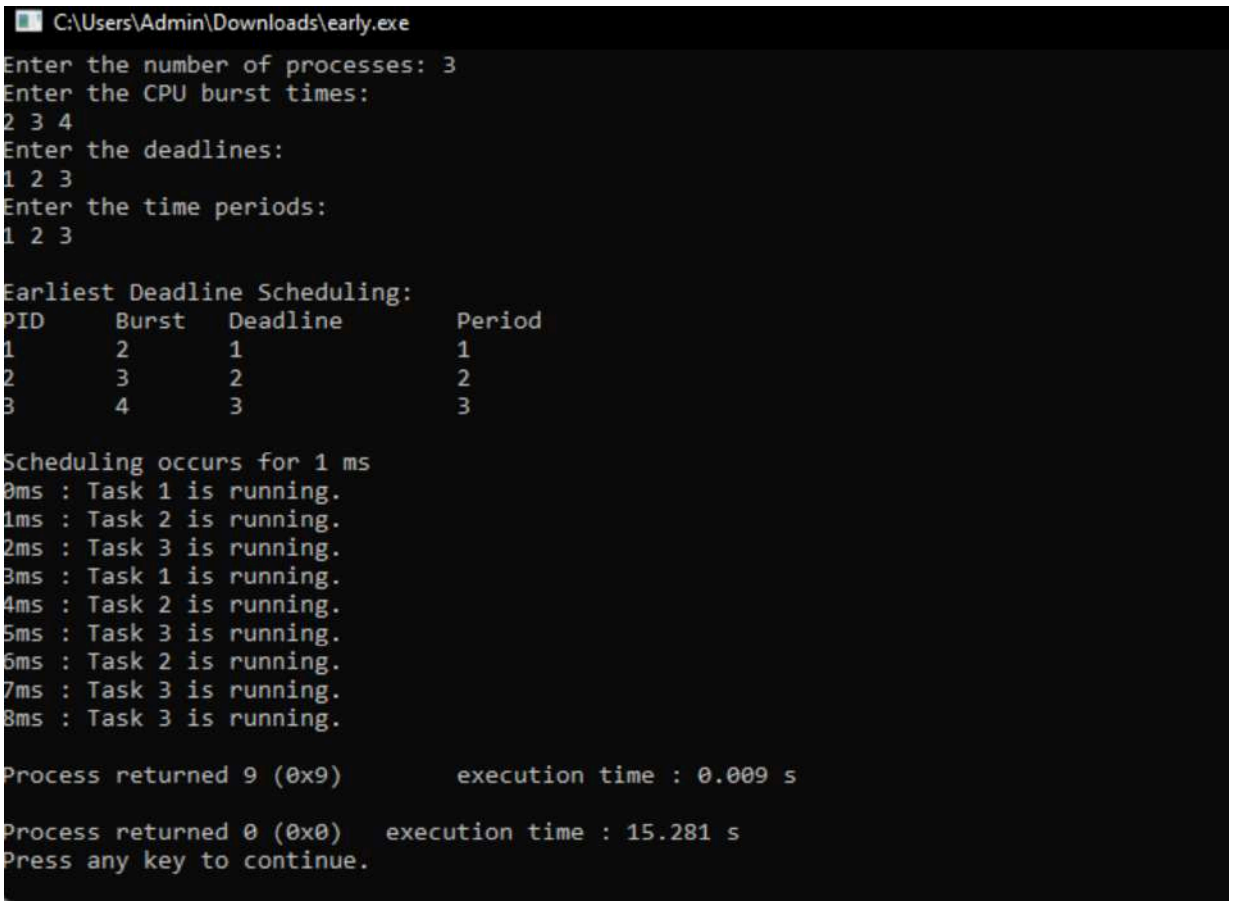
EARILIEST DEADLINE

CODE:

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

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

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

int main() {
    int num_processes;

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

    Process processes[num_processes];

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

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

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

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

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

    int current_time = 0;
    int completed_processes = 0;

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

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

```
            current_time++;

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

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

    return 0;
}
```



⑥ Earliest Deadline First

```
#include <stdio.h>
typedef struct {
    int pid, BT, RT;
} proces;
int compare deadlines (const void *a, const void
    return ((proces *)) a->deadline - ((proces *)
        deadline;
    }
int main() {
    int main() {
    int num=processes;
    pf ("Enter no of processes:");
proceses Processes [num-proces];
    pf ("Enter CPU BT:");
        for i = 0; i < num-processes; i++
        sf ("%d", & processes(i).BT);
        processes (i).pid=i+1;
        processes (i).RT = proces(i).BT;
}
pf ("Enter deadline:");
    for i=0; i<num-processes; i++) {
        scanf ("%d", & processes (i).deadline;
    }
    pf( "Enter Time:");
    for (int i=0; i<num; i++) {
        scanf ("%d", & processes (i).period);
q sort (proces es, num-processes, sizeof
    (processes);
pf("\n Enter :%d");
    processes(i).pid  proces (i).BT;
```

```
3
int CT = 0;
int completed_processes = 0;
    pf ("\n Scheduling occurs for %d ms\n",
     processes(0).deadline);
while (completed_processes < num_process) {
    for (int i = 0; i < num_process; i++)
     if (processes(i).RT) {
     printf("%d ms > task %d", 
current_Time processes(i).pid);
        current_Time++;
     if (processes(i).RT = 0) {
        completed_processes++;
    }
}}
```

Output :-

Enter no of process : 3

2 3 4

1 2 3

System : 6 ms

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

Scheduling occurs for 6 ms :

0 ms : Task 1 is running

1 ms : Task 1 is running

2 ms : Task 2 is running

3 ms : Task 2 is running

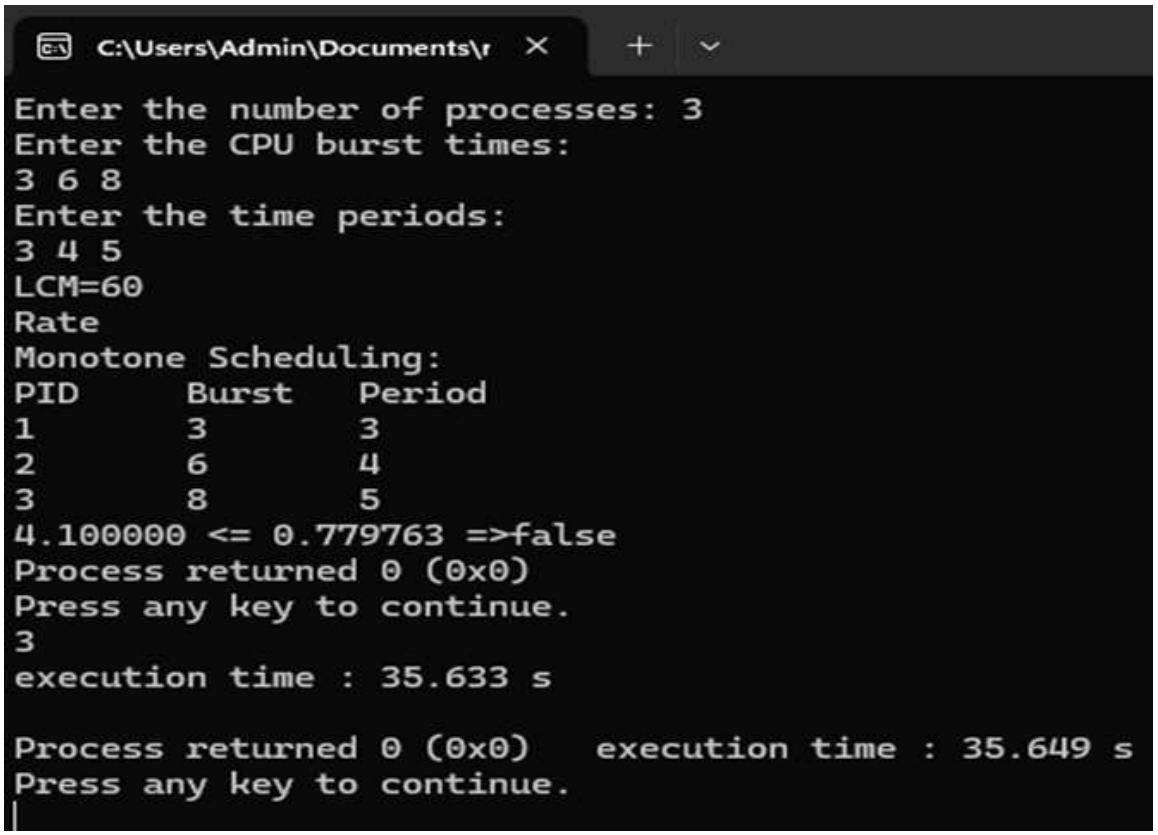
4 ms : Task 2 is running

RATE MONOTONIC

CODE:

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

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

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

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

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

void rateMonotonic(Task tasks[], int n) {
    float utilization = 0;
    printf("\nRate Monotonic Scheduling:\nPID\tBurst\tPeriod\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\n", tasks[i].id, tasks[i].burst, tasks[i].period);
        utilization += (float)tasks[i].burst / tasks[i].period;
    }
```

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

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

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

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

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


C:\Users\Admin\Documents\r
Enter the number of processes: 3
Enter the CPU burst times:
3 6 8
Enter the time periods:
3 4 5
LCM=60
Rate
Monotone Scheduling:
PID Burst Period
1 3 3
2 6 4
3 8 5
4.100000 <= 0.779763 =>false
Process returned 0 (0x0)
Press any key to continue.
3
execution time : 35.633 s
Process returned 0 (0x0) execution time : 35.649 s
Press any key to continue.

① Rate - Monotonic scheduling :-

```
#include <stdio.h>
#define MAX-Process 10
    type def Struct {
        int id;
        int BT;
    } 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 find (Task Task[], int n) {
    int result = tasks(0).period;
    for(int i=1; i<n; i++)
    result= lcm(result, takes(i), period);
        return result;
}
int find LCMC Task tasks[], int n){
    int result = tasks (0).period;
    for (int i=1; i<n; i++)
    result = lcm(result, task-p(i)
}
void rate monotic (task tasks[], int n) {
        float utilisation = 0;
pf ("In Rate Monotonic Scheduling :\n");
    Burst period \n");
    postint i=0; i<n; i++)
        pf ("%d %d , task(i))
task p(i).bT, task p(i).period;
    utilization = (float) tasks (i).bT.period;
}
```

```
printf("!!! Deadline miss for p%d at
        time %d\n", p[current process].pid,
                                time+1);
    }
    else {
        printf("%d\t IDLE\n", time);
    }
}
```

output.

Enter the number of Processes : 3
Enter the CPU burst time;
3 6 8
Enter the Time periods:
3 4 5
LCM = 60

Rate Monotone Scheduling :

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

$y = 100000 \quad x <= 0.779763 \Rightarrow$ false.

17

PRODUCER CONSUMER

CODE:

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

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

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

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

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

```
        printf("Buffer is full\n");
    }
}

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

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

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

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

    srand(time(NULL));


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

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

```
Buffer:47
Consumer 2 consumed 47
Current buffer len: 0
Producer 1 produced 9
Buffer:9
Consumer 2 consumed 9
Current buffer len: 0
Producer 1 produced 44
Buffer:44
Consumer 2 consumed 44
Current buffer len: 0
Producer 1 produced 30
Buffer:30
Consumer 2 consumed 30
Current buffer len: 0
Producer 1 produced 1
Buffer:1
Consumer 2 consumed 1
Current buffer len: 0
Producer 1 produced 14
Buffer:14
Consumer 2 consumed 14
Current buffer len: 0
Producer 1 produced 48
Buffer:48
Consumer 2 consumed 48
Current buffer len: 0
Producer 1 produced 2
Buffer:2
Consumer 2 consumed 2
Current buffer len: 0
Producer 1 produced 31
Buffer:31
Consumer 2 consumed 31
Current buffer len: 0
```

Producer Consumer

```
#include <stdio.h>
#include <stdlib.h>
int mutex=1, full=0, empty x=0°, ;
    int * buffer, buffer size°,
int in=0°, out=0°,
int wait (int s) {
        xturn (s)°,
  }
int signal (int s) {
    xturn ++s°,
  }
void producer (int id) {
   if (mutex ==1) && (empty != 0) {
        mutex = wait (mutex)°;
        full = wait (empty);
        int item= void ()%.50°;
      buffer (in) = item°,
        x++;
printf ("producer %d produced in %d");
printf ("Buffer: %d \n", item)°,
    in = (in++) % buffer -size;
      mutex = signal (mutex)°;
  }else {
  Pf("Buffer is full\n")°,
    }
  }
void consumer (int in) {
      if (mutex == 1 && full != 0) {
```

```
    mutex = wait(mutex);
    full = wait(full);
    empty = signal(empty);
    int item = buffer(out);
    printf("consumer %d consumed %d!!");
        x--;
    }
int main(){
    int producer-consumer;
    pf("Enter the no of producers:");
    sf("%d", &producer);
    pf("Enter the buffer capacity");
    sf("%d, &buffer);
    buffer = (int*) malloc(buffer-size);
}
return 0;
}
```

Output:-

1. producer
2. consumer
3. Exit

Enter your choice : 1
Producer produces item 1

Enter your choice : 1
Producer produces item 2
Enter your choice : 1
Buffer is full
Enter your choice : 2
consumer consumes item 3

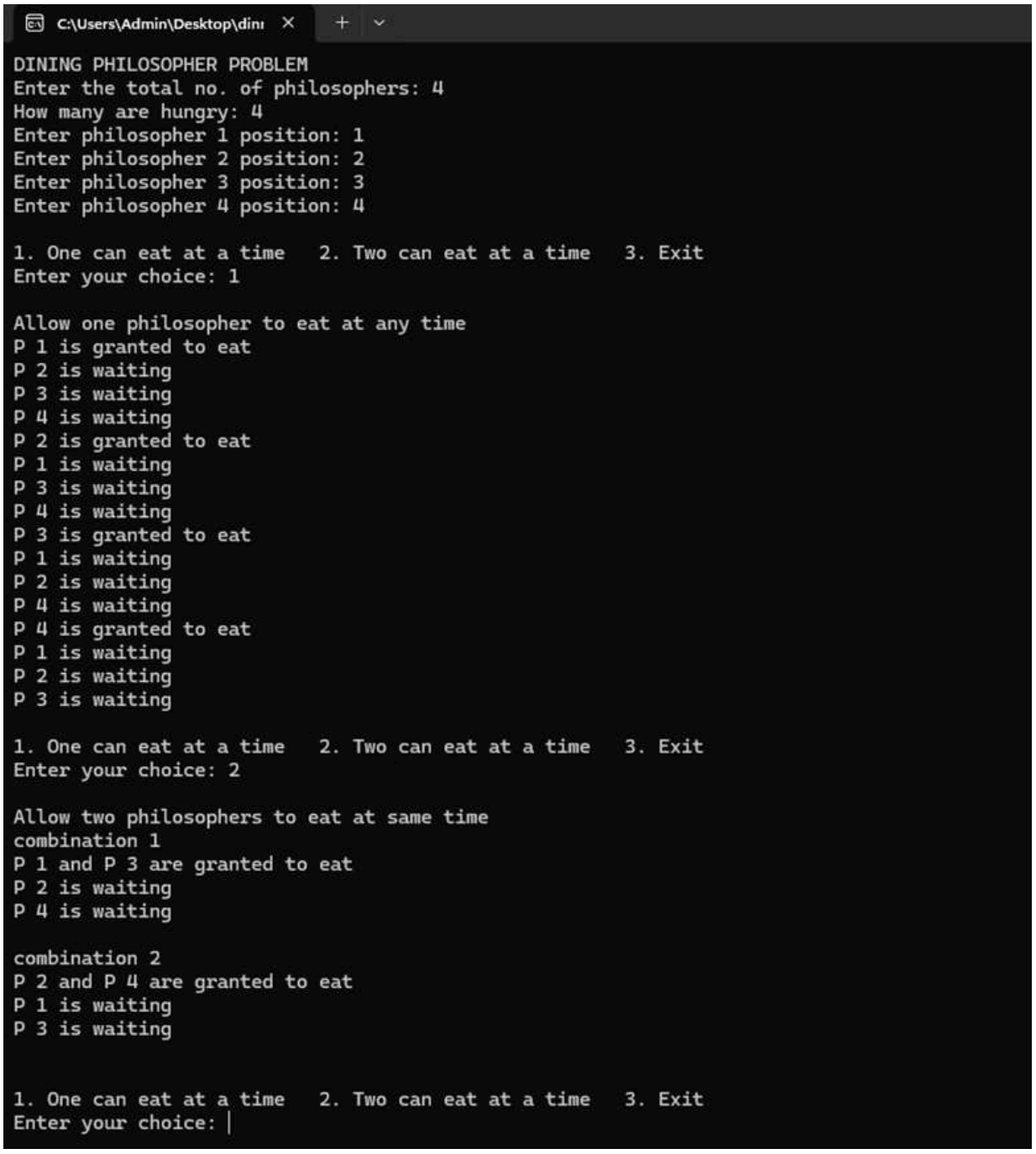
DINNING PHILOSOPHER

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

       state[phnum] = EATING;
       sleep(2);
       printf("Philosopher %d takes fork %d and %d\n", phnum + 1, LEFT + 1, phnum + 1);
       printf("Philosopher %d is Eating\n", phnum + 1);

       sem_post(&S[phnum]);
```

```
    }
}

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

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

    test(phnum);

    sem_post(&mutex);
    sem_wait(&S[phnum]);

    sleep(1);
}

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

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

    test(LEFT);
    test(RIGHT);

    sem_post(&mutex);
}
```

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

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

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;
}
```


C:\Users\Admin\Desktop\din
DINING PHILOSOPHER PROBLEM
Enter the total no. of philosophers: 4
How many are hungry: 4
Enter philosopher 1 position: 1
Enter philosopher 2 position: 2
Enter philosopher 3 position: 3
Enter philosopher 4 position: 4
1. One can eat at a time 2. Two can eat at a time 3. Exit
Enter your choice: 1
Allow one philosopher to eat at any time
P 1 is granted to eat
P 2 is waiting
P 3 is waiting
P 4 is waiting
P 2 is granted to eat
P 1 is waiting
P 3 is waiting
P 4 is waiting
P 3 is granted to eat
P 1 is waiting
P 2 is waiting
P 4 is waiting
P 4 is granted to eat
P 1 is waiting
P 2 is waiting
P 3 is waiting
1. One can eat at a time 2. Two can eat at a time 3. Exit
Enter your choice: 2
Allow two philosophers to eat at same time
combination 1
P 1 and P 3 are granted to eat
P 2 is waiting
P 4 is waiting
combination 2
P 2 and P 4 are granted to eat
P 1 is waiting
P 3 is waiting
1. One can eat at a time 2. Two can eat at a time 3. Exit
Enter your choice:

⑥ Dinning Philosopher

```
#include <stdio.h>
#include <stdlib.h>
int total philosopher;
int hungry (MAX);
  return (abs(a-b) == 1 || abs(a-b) == total philosophers
                                        -1);
}
void option1 (int count) {
    printf ("\n Allow one philosopher to eat at any
    for(int i=0; i<count, i++) {                time");
    printf ("%d is granted to eat \n");
      hungry(i) ;
  for(int j=0; j<count; j++) {
      if(j!=i) {
        printf ("%d is waiting");
    hungry (j));
    }
  }
void option 2 (int count) {
  printf ("\n Allow two philosophers to eat at
                              same time");
  int combination=1;
  for (int i=0; i< count; i++) {
    for(int j=0; j< count; j++) {
  if (!are Neighbours (hungry(i), hungry(j)){
    printf ("Combination %d");
      if ( k1 == 0 && k1 == j) {
      printf("P %d is waiting \n")
          hungry (k1));
    }
  }
  printf ("\n");
  }
}
```

```
if (combination == 1){
    printf("No combination found where
        two non-neighbour philosopher
        can eat.\n");
    }
}
int main() {
int hungry_count;
printf("Enter the total no of philosophers");
scanf("%d", &total philosophers");
printf("How many are hungry");
scanf("%d" &hungry_count);
for(int i=0; i<hungry_count;i++) {
    printf("Enter philosopher %d
                positive i+1);
    scanf("%d" &hungry_[i]);
}
int choice;
do {
pf("\n One can eat at a time \n")
  2. Two can eat a time
  3. Exit \n");
Pf("Enter your choice");
Sf("%d, &choice) {
 case 1:
      option1(hungry_count);
      break;
 case 2:
      option2(hungry_count);
      break;
 case 3:
      printf("Exiting\n");
      break;
```

default
printf("Invalid choice\n");
}
} while (choice != 3);
return 0;
}

Output:-

Dinning Philosopher Problem
Enter the total no. of philosophers : 5
How many are hungry : 3
Enter philosopher 1 pos : 2
Enter philosopher 2 pos : 4
Enter philosopher 3 pos : 5

Allow one philosopher to eat at any time
P2 is allowed
P4 is allowed
P5 is allowed
P4 is allowed
P2 is allowed
P5 is allowed
P2 is allowed
P4 is allowed.

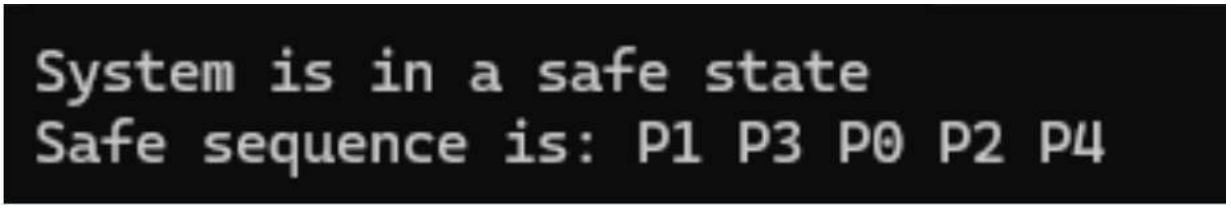
DEADLOCK AVOIDANCE

CODE:

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

#define P 5
#define R 3

bool isSafe(int avail[], int max[][R], int allot[][R]) {
    int need[P][R];
    bool finish[P] = {false};
    int safeSeq[P];
    int work[R];

    for (int i = 0; i < P; i++) {
        for (int j = 0; j < R; j++) {
            need[i][j] = max[i][j] - allot[i][j];
        }
    }

    for (int i = 0; i < R; i++) {
        work[i] = avail[i];
    }

    int count = 0;
    while (count < P) {
        bool found = false;

        for (int p = 0; p < P; p++) {
            if (!finish[p]) {
                bool canAllocate = true;
                for (int r = 0; r < R; r++) {
```

```
                if (need[p][r] > work[r]) {
                    canAllocate = false;
                    break;
                }
            }

            if (canAllocate) {
                for (int r = 0; r < R; r++) {
                    work[r] += allot[p][r];
                }
                safeSeq[count++] = p;
                finish[p] = true;
                found = true;
                break;
            }
        }
    }

    if (!found) {
        printf("System is in an unsafe state\n");
        return false;
    }
  }

  printf("System is in a safe state\nSafe sequence is: ");
  for (int i = 0; i < P; i++) {
    printf("P%d ", safeSeq[i]);
  }
  printf("\n");

  return true;
}
```

```
int main() {
    int avail[] = {3, 3, 2};

    int max[][R] = {
        {7, 5, 3},
        {3, 2, 2},
        {9, 0, 2},
        {2, 2, 2},
        {4, 3, 3}
    };

    int allot[][R] = {
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 2},
        {2, 1, 1},
        {0, 0, 2}
    };

    isSafe(avail, max, allot);

    return 0;
}
```



Dead Lock Avoidance

```
#include <stdio.h>
#include <stdbool.h>
int main(){
    int n,m;
printf("process & resources:\n");
scanf("%d %d",&n,&m);
int alloc[n][m],max[n][m],avail[m],
    need[n][m];
printf("Enter allocation matrix:\n");
for(int i=0;i<n;i++)
    for(int j=0;j<m;j++)
    scanf("%d",&allocation[i][j]);
for(int i=0;i<n;i++)
    for(int j=0;j<m;j++)
    scanf("%d",&max[i][j]);

printf("Enter available matrix\n");
for(int i=0;i<m;i++)
    scanf("%d",&avail[i]);
    need[i][j]=max[i][j],alloca[i][i];
bool finish[n];
for(int i=0;i<n;i++) finish[i]=false;

int safe seq[n];
    int count=0;
while(count<n){
    bool found=false;
for(int p=0;p<n;p++){
    if(!finish[p]){
        bool can alloc=true;
for(int j=0;j<m;j++){
```

```
if (need(p)(j) > avail canAlloc = false;
    break;
}
if (canAllocate) {
    for (int k=0; k<m; k++)
        avail[k] += alloc[p][k];
    safeseq[count++] = p;
    finish[p] = true;
    found = true;
}
}
if(!found) {
    printf("System is not in safe
        state:");
    return 1;
}
```

Output:-

Enter no. of processes & 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

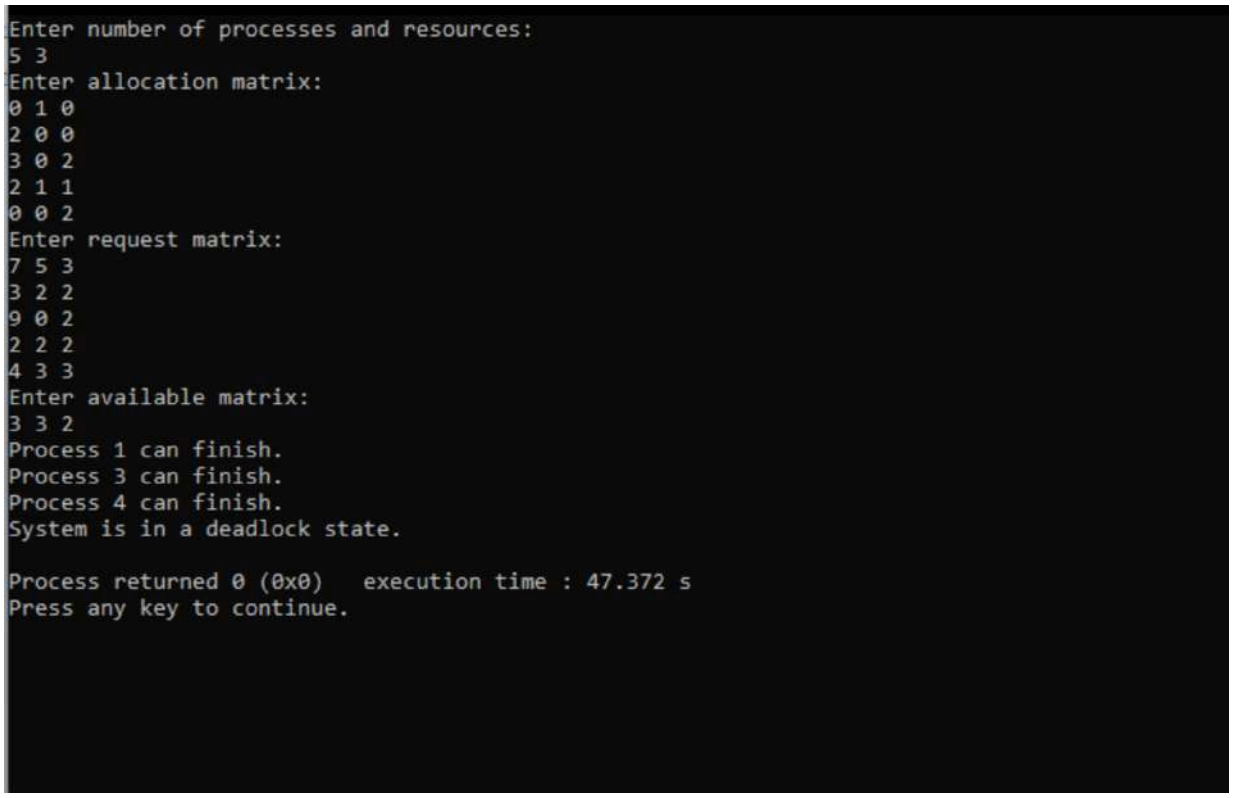
P1 → P3 → P4 → P0 → P2

DEADLOCK DETECTION

CODE:

```
#include <stdio.h>

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

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

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

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

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

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

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

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

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

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

    return 0;
}
```


Enter number of processes and resources:
5 3
Enter allocation matrix:
0 1 0
2 0 0
3 0 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 : 47.372 s
Press any key to continue.

Deadlock Detection

```
#include <stdio.h>
#include <stbool.h>
void deadlock-detection (int allocation (i)(R),
                         int req(i)(R) )
    int work(P)",
    bool finish(P) & false};
  bool deadlock = false;
  for(int i=0", i<R ;i++) {
    work (i)= available (i);
  }
int count = 0;
(count<P) }
  bool found = false;
  for(int j=0 ;j<R ;j++) {
  if (can) {
    for(int j=0 ;j<R ;j++) {
work(j)+ = allocation (i)(j); }
    finish (i)= true;
    Pf("Proces P%d is finished");
      count++ ;
        }}
if (! found){
    }}
for (int i=0 ;i<P ;i++) {
      if (! finished (i) ){
        deadlock = true;
      printf ("Processes %d deadlock(n")
      if (! deadlock) {
      Pf " no deadlock detected proces
        complete");
      }else {
```

```
int main() {
    int allocation [5][3] = {
        {0,1,0},
        {2,0,0},
        {3,0,3},
        {2,1,1},
        {0,0,2}};
    int request [5][3] = {
        {0,0,0},
        {2,0,2},
        {0,0,1},
        {1,0,0},
        {0,0,2}};
    int available [3] = {0,0,0};
    deadlockdetection(allocation, request,
                      available)
    return;
}
```

Output :-

Process P_0 is finished
Process P_1 is finished
P_2 is finished
P_3 is finished
P_4 is finished
P_1 is finished.

No deadlock detected.

MULTILEVEL QUIENING

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;
}
```

```
Enter number of processes: 4
Enter Burst Time, Arrival Time and Queue of P1: 2
0
1
Enter Burst Time, Arrival Time and Queue of P2: 1
0
2
Enter Burst Time, Arrival Time and Queue of P3: 5
0

1
Enter Burst Time, Arrival Time and Queue of P4: 3
0

2

Queue 1 is System Process
Queue 2 is User Process

Process Waiting Time    Turn Around Time        Response Time
1       0               2                       0
2       2               3                       2
3       3               8                       3
4       8               11                      8

Average Waiting Time: 3.25
Average Turn Around Time: 6.00
Average Response Time: 3.25
Throughput: 0.36
```

ROUND ROBIN

CODE:

```
#include <stdio.h>

#define MAX_PROCESSES 10

typedef struct {
    int pid;
    int bt;
    int at;
    int queue;
} Process;

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

```
    }
}
void roundRobin(Process p[], int n, int quantum, int wt[], int tat[], int rt[]) {
    int remaining_bt[MAX_PROCESSES];
    for (int i = 0; i < n; i++)
        remaining_bt[i] = p[i].bt;
    int t = 0, completed = 0;
    while (completed < n) {
        int executed = 0;
        for (int i = 0; i < n; i++) {
        if (remaining_bt[i] > 0) {
            if (rt[i] == -1) rt[i] = t;
            if (remaining_bt[i] > quantum) {
           t += quantum;
            remaining_bt[i] -= quantum;
            } else {
            t += remaining_bt[i];
            tat[i] = t - p[i].at;
            wt[i] = tat[i] - p[i].bt;
```

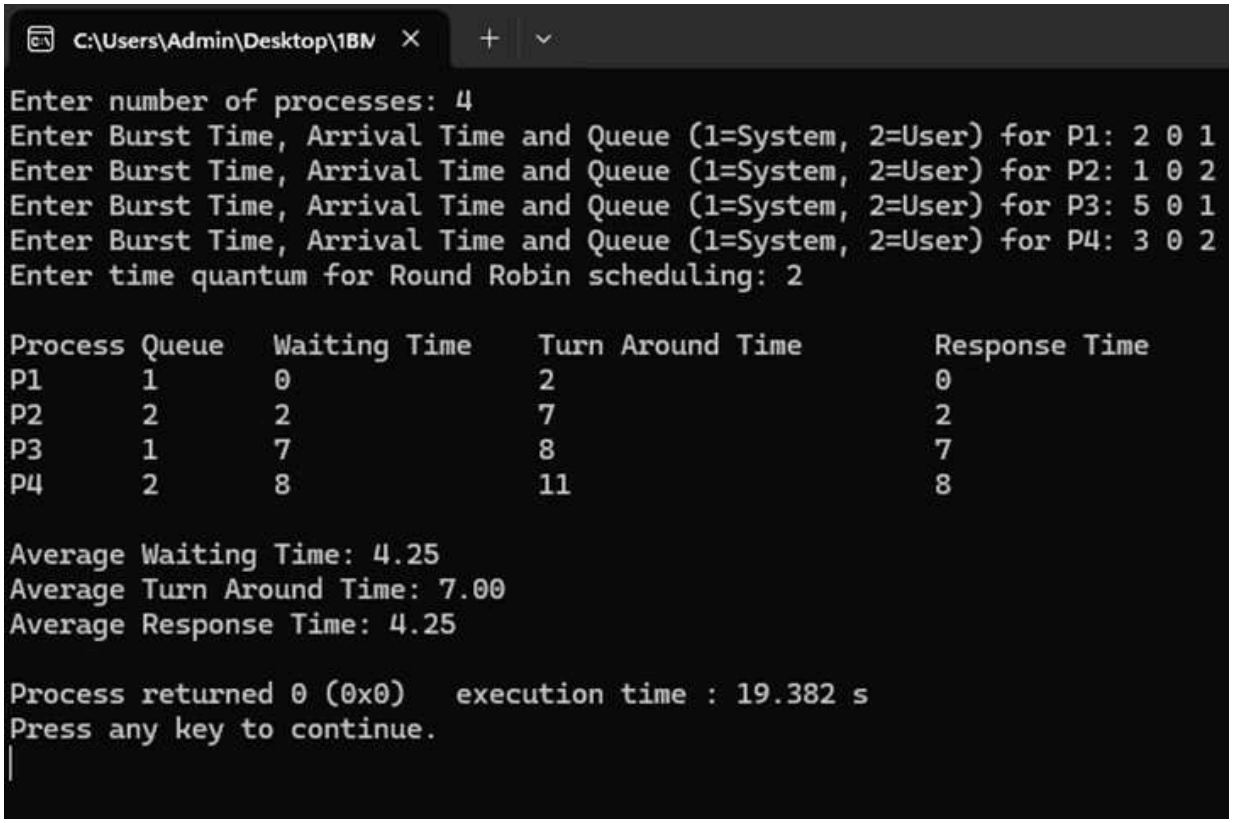
```
                remaining_bt[i] = 0;
                completed++;
            }
                executed = 1;
            }
            }
            if (!executed) t++;
        }
    }
    void fcfs(Process p[], int n, int start_time, int wt[], int tat[], int rt[]) {
        int time = start_time;
        for (int i = 0; i < n; i++) {
            if (time < p[i].at) time = p[i].at;
            rt[i] = time - p[i].at;
            wt[i] = rt[i];
            tat[i] = wt[i] + p[i].bt;
            time += p[i].bt;
        }
    }
```

```
int main() {
    int n, quantum;
    Process p[MAX_PROCESSES], sys[MAX_PROCESSES], usr[MAX_PROCESSES];
    int sys_count = 0, usr_count = 0;
    int wt[MAX_PROCESSES], tat[MAX_PROCESSES], rt[MAX_PROCESSES];
    printf("Enter number of processes: ");
   scanf("%d", &n);
for (int i = 0; i < n; i++) {
        printf("Enter Burst Time, Arrival Time and Queue (1=System, 2=User) for P%d: ", i +
1);
        p[i].pid = i + 1;
        scanf("%d %d %d", &p[i].bt, &p[i].at, &p[i].queue);
        if (p[i].queue == 1)
        sys[sys_count++] = p[i];
        else
        usr[usr_count++] = p[i];
        wt[i] = 0;
        tat[i] = 0;
        rt[i] = -1;
    }
```

```
    printf("Enter time quantum for Round Robin scheduling: ");
    scanf("%d", &quantum);
    sortByArrival(sys, sys_count);
    sortByArrival(usr, usr_count);
    roundRobin(sys, sys_count, quantum, wt, tat, rt);
    int last_sys_time = (sys_count > 0) ? tat[sys_count - 1] + sys[sys_count - 1].at : 0;
    fcfs(usr, usr_count, last_sys_time, &wt[sys_count], &tat[sys_count], &rt[sys_count]);
    printf("\nProcess\tQueue\tWaiting Time\tTurn Around Time\tResponse Time\n");
    for (int i = 0; i < n; i++)
        printf("P%d\t%d\t%d\t\t%d\t\t\t%d\n", p[i].pid, p[i].queue, wt[i], tat[i], rt[i]);
 float avg_wt = 0, avg_tat = 0, avg_rt = 0;
for (int i = 0; i < n; i++) {
            avg_wt += wt[i];
            avg_tat += tat[i];
            avg_rt += rt[i];
    }
    printf("\nAverage Waiting Time: %.2f", avg_wt / n);
    printf("\nAverage Turn Around Time: %.2f", avg_tat / n);
    printf("\nAverage Response Time: %.2f\n", avg_rt / n);
```

```
    return 0;
}
```

Output:



3/04/25

LAB NO. 3

Q. Write a C pgm to simulate multilevel queue scheduling algorithm considering the following scenario. All the processes in the system are divided into 2 categories System processes and user processes. System process - higher priority. Use RR and FCFS scheduling for the processes in each queue.

Soln

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

typedef struct {
        int pid;
        int bt;
        int at;
        int queue;
} process;

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

```
void RoundRobin (Process p[], int n, int quantum,
                 int wt[], int tat[], int rt[]){
    int remaining_bt[MAX_PROCESSES];
    for (int i=0; i<n; i++){
        remaining_bt[i] = p[i].bt;
    int t=0, completed=0;
    while (completed <n){
        int executed=0.
        for (int i=0; i<n; i++){
            if (remaining_bt[i] >0){
                if (rt[i] == -1) rt[i]=t;
                if (remaining_bt[i] > quantum)
                    t+= quantum.
                    remaining_bt[i] -= quantum;
                else{
                    t+= remaining_bt[i];
                tat[i] = t - p[i].at;
                wt[i] = tat[i] - p[i].bt.
                remaining_bt[i]=0;
                completed++;
            }
            executed = 1;
        }
        if (!executed) t++;
    }
}

void fcfs (process p[], int n, int start_time,
  int wt[], int tat[], int rt[]){
    int time = start_time;
    for( int i=0; i<n; i++){
        if (time <p[i].at) time
                = p[i].at;
```

```
        rt[i] = time - p[i].at;
        wt[i] = rt[i];
        tat[i] = wt[i] + p[i].bt;
        time += p[i].bt;
    }
}

int main() {
    int n, quantum;
    Process p[MAX_Processes], sys[MAX_Processes],
    usr[MAX_processes];
    int sys_count = 0, usr_count = 0;
    int wt[MAX_Processes], tat[MAX_Processes],
    rt[MAX_processes];
    printf("Enter number of process:");
    scanf("%d", &n);

    for (int i=0; i<n; i++) {
        printf("Enter Burst time,
            Arrival Time, and Queue(1=
            System; 2= User) for P%d:",
            i+1);
        p[i].pid = i+1;
        scanf("%d %d %d", &p[i].bt, &p[i].at,
        &p[i].queue);

        if (p[i].queue == 1) {
            sys[sys_count++] = p[i];
        else
            usr[sys_count++] = p[i];
        wt[i] = 0;
        tat[i] = 0;
        rt[i] = -1;
    }
```

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

sort_by_arrival(sys, sys_count);
sort_by_arrival(usr, usr_count);

printf("\n Process \t queue \t waiting \t
  turn around time \t response time\n");
float avg_wt = 0; avg_tat = 0, avg_rt = 0;
for (int i = 0; i < n; i++){
      avg_wt += wt[i];
      avg_tat += tat[i];
      avg_rt += rt[i];
  }

printf("\n Average Waiting Time: %.2f",
                          avg_wt/n);
printf("\n Average turnaround time: %.2f",
                          avg_tat/n);
printf("\n Average response time: %.2f",
                          avg_rt/n);
return 0;
```

output

```
Enter number of process : 4
Enter burst time, arrival time, Queue (1 =
  System, 2 = user) for P1 : 20 1
Enter burst time, arrival time, Queue (1 =
  System, 2 = user) for P2 : 1 0 2
Enter burst time, arrival time, Queue (1 = System,
```

Round
Enter time quantum for Robin Scheduling: 2

Process	Queue	WT	TAT	RT
P1	1	0	2	0
P2	2	2	7	2
P3	1	7	8	7
P4	2	8	11	8

Average WT: 4.25
Average TAT: 7.00
Average RT: 4.25

MEMORY MANAGEMENT UNIT (BEST FIT,WORST FIRST,FIRST FIT)

CODE:

```
#include <stdio.h>

#define MAX_BLOCKS 20
#define MAX_PROCESSES 10

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

    printf("\nWorst Fit Allocation:\n");
    for (int i = 0; i < n; i++) {
        if (allocation[i] != -1) {
            printf("Process %d allocated to Block %d\n", i + 1, allocation[i] + 1);
```

```
        } else {
            printf("Process %d not allocated\n", i + 1);
        }
    }
}

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

    printf("\nBest Fit Allocation:\n");
    for (int i = 0; i < n; i++) {
        if (allocation[i] != -1) {
            printf("Process %d allocated to Block %d\n", i + 1, allocation[i] + 1);
        } else {
```

```
            printf("Process %d not allocated\n", i + 1);
        }
    }
}

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("\nFirst Fit Allocation:\n");
    for (int i = 0; i < n; i++) {
        if (allocation[i] != -1) {
            printf("Process %d allocated to Block %d\n", i + 1, allocation[i] + 1);
        } else {
            printf("Process %d not allocated\n", i + 1);
        }
    }
}

int main() {
```

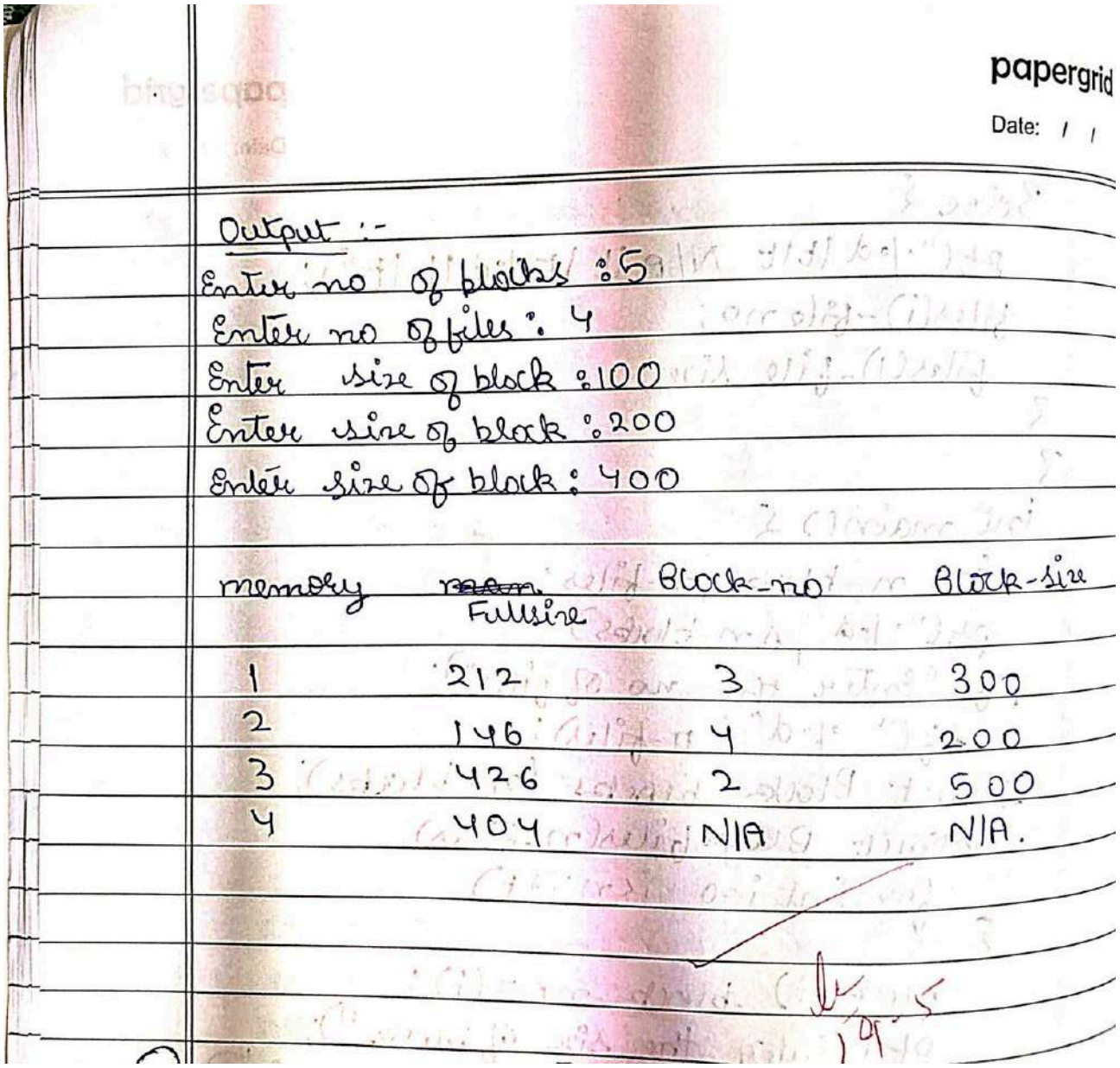
```
    int blockSize[MAX_BLOCKS], processSize[MAX_PROCESSES];
    int m, n;

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

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

    firstFit(blockSize, m, processSize, n);
    bestFit(blockSize, m, processSize, n);
    worstFit(blockSize, m, processSize, n);

    return 0;
}
```

```
Block 5: 600
Enter the number of processes: 4
Enter the size of each process:
Process 1: 212
Process 2: 417
Process 3: 112
Process 4: 426

First Fit Allocation:
Process 1 allocated to Block 2
Process 2 allocated to Block 5
Process 3 allocated to Block 2
Process 4 not allocated

Best Fit Allocation:
Process 1 allocated to Block 4
Process 2 not allocated
Process 3 allocated to Block 2
Process 4 not allocated

Worst Fit Allocation:
Process 1 not allocated
Process 2 not allocated
Process 3 allocated to Block 3
Process 4 not allocated
```

⑤ Memory Management unit Scheme (Best fit) (worst fit) (First fit)

```
#include <stdio.h>
struct Block {
int block_no;
int block_size;
 int is_free;};
struct File {
   int file_no;
    int file_size;
};
void batt(struct Block blocks(), int n_block,
          struct File file(), int n_files)
{
  printf("\n Memory Management scheme");
  printf("Filenolt File Size| Block_no \n");
  for(int i=0; i<n_files;i++)
   {
    int best_fit_block = -1;
     int min_nagement = 10000;
    for(int j=0;j<n;j++) {
    int nagement = blocks(j) -blocksize. -files(i)
     if (fragement < min_fragement) {
         min_fragement = fragement;
             best_fit_block = j;
     }
  if(best_fit_block1 = -1) {
       blocks(best_fit_block) is_free = 0;
  printf("%d lt|t %d lt lt %d");
     files(i)_file no;
     files(i)_file size;
  blocks(best_fit_block)_block_no
  block(best_fit_block) block_size
        min_fragement;
```

```
} else {
  pf("%d \t\t N/A \t\t %d \t\t");
  files(i)-file no;
  files(i)-file size;
}
}
int main() {
  int n-blocks, n-files;
  pf("%d", &n-blocks)
pf("Enter the no of files");
  sf("%d" & n-files);
struct Block blocks (n-blocks);
 struct Block files(n-files);
   for(int i=0 ; i<n;i++)
{
  blocks(i) -block-no = (i);
  pf("Enter the size of block");
  sf("%d" &blocks(i), block-size);
    blocks(i)-isfree=1;
}
for (int i=0; i<n-files;i++) {
    files(i) file-no = 1;
  pf("Enter size of file %d");
}
for( int i=0 ; i<n-blocks ;i++)
{
  blocks(i)- is-free = 1;
}
```



papergrid

Date: / /

Output :-

Enter no of blocks : 5

Enter no of files : 4

Enter size of block : 100

Enter size of block : 200

Enter size of block : 400

memory	Fullsize	Block-no	Block-size
1	212	3	300
2	146	4	200
3	426	2	500
4	404	N/A	N/A.

PAGE REPLACEMENT PROGRAM(LRU,OPTIMAL,FIFO)

CODE:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_FRAMES 10
#define MAX_PAGES 50


bool isPageInFrame(int frames[], int page, int frameCount) {
   for (int i = 0; i < frameCount; i++) {
      if (frames[i] == page) {
         return true;
      }
   }
   return false;
}


int fifo(int pages[], int pageCount, int frameCount) {
   int frames[MAX_FRAMES], front = 0, pageFaults = 0;

   for (int i = 0; i < pageCount; i++) {
      if (!isPageInFrame(frames, pages[i], frameCount)) {
         frames[front] = pages[i];
         front = (front + 1) % frameCount;
         pageFaults++;
      }
   }
   return pageFaults;
}
```

```
int lru(int pages[], int pageCount, int frameCount) {
    int frames[MAX_FRAMES], lastUsed[MAX_FRAMES], pageFaults = 0;

    for (int i = 0; i < pageCount; i++) {
        if (!isPageInFrame(frames, pages[i], frameCount)) {
            int leastRecentlyUsedIndex = 0;
            for (int j = 1; j < frameCount; j++) {
                if (lastUsed[j] < lastUsed[leastRecentlyUsedIndex]) {
                    leastRecentlyUsedIndex = j;
                }
            }
            frames[leastRecentlyUsedIndex] = pages[i];
            pageFaults++;
        }
        for (int j = 0; j < frameCount; j++) {
            lastUsed[j]++;
        }
        lastUsed[pages[i] % frameCount] = 0;
    }
    return pageFaults;
}


int optimal(int pages[], int pageCount, int frameCount) {
    int frames[MAX_FRAMES], pageFaults = 0;

    for (int i = 0; i < pageCount; i++) {
        if (!isPageInFrame(frames, pages[i], frameCount)) {
            int farthestIndex = -1, replaceIndex = -1;
            for (int j = 0; j < frameCount; j++) {
                int nextUse = pageCount;
```

```
            for (int k = i + 1; k < pageCount; k++) {
                if (frames[j] == pages[k]) {
                    nextUse = k;
                    break;
                }
            }
            if (nextUse > farthestIndex) {
                farthestIndex = nextUse;
                replaceIndex = j;
            }
        }
        frames[replaceIndex] = pages[i];
        pageFaults++;
    }
  }
  return pageFaults;
}

int main() {
  int pages[MAX_PAGES], pageCount, frameCount;

  printf("Enter number of pages: ");
  scanf("%d", &pageCount);
  printf("Enter page sequence: ");
  for (int i = 0; i < pageCount; i++) {
    scanf("%d", &pages[i]);
  }
  printf("Enter number of frames: ");
  scanf("%d", &frameCount);

  printf("FIFO Page Faults: %d\n", fifo(pages, pageCount, frameCount));
  printf("LRU Page Faults: %d\n", lru(pages, pageCount, frameCount));
```

```
    printf("Optimal Page Faults: %d\n", optimal(pages, pageCount, frameCount));

    return 0;
}
```

```
Enter number of pages: 12
Enter page sequence: 1 3 0 3 5 6 3 0 3 5 2 3
Enter number of frames: 3
```

19/05/25

LAB-7

Page Replacement

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

bool search(int key, int fr[], int capacity){
    for(int i=0; i<capacity; i++){
        if(fr[i]==key)
            return true;
        return false;
    }
int predict(int pages[], int fr[], int pn,
    int index, int capacity){
        int res=-1; farthest=index;
    for(int i=0; i<capacity; i++){
        int j;
        for(j=index; j<pn; j++){
            if(fr[i]==pages[j]){
                if(j>farthest){
                    farthest=j;
                    res=i;
                }
                break;
            }
        }
        if(j==pn)
            return i;
    }
    return (res==-1)?0:res;
}
```

```
        continue;
    }
    if (filled < capacity) {
        fr[filled++] = pages[i];
    } else {
        int j = predict(pages, fr, n, i+1,
                        capacity);
        fr[j] = pages[i];
    }
    page_faults++;
}
printf("Optimal Page Faults: %d,
        Page Hits: %d\n",
        page_faults, page_hits);
}

void lru(int pages[], int n, int
capacity){
    int fr[capacity];
    int recent[capacity];
    int page_faults = 0, page_hits=0;
    int time=0.

    for(int i=0; i< capacity; i++){
        fr[i] = -1;
        recent[i] = -1;
    }
    for(int i= 0; i<n; i++) {
        bool hit= false;
        for(int j=0; j< capacity; j++){
            if (fr[i] == pages[i]){
                time++;
```

```
void fifo ( int pages[], int n, int capacity)
{ int fr[capacity];
   int page_faults = 0, page_hits = 0;
   int index=0;

for (int i=0; i<n; i++ ) {
    if (! search (pages [i], fr, capacity))
    { fr[index] = pages[i];
      index = (index +1) % capacity;
      page_faults ++;
    } else {
        page_hits++;
    }
  }
  printf (" FIFO Page Faults %d,
        Page Hits : %d", page_faults,
        page_hits);
}

void optimal (int pages[], int n, int
capacity) {
   int fr[capacity];
   int page_faults=0, page_hits=0;
   int filled = 0;
  for (int i=0; i<n; i++){
     if ( search ( pages[i], fr,
        capacity)){
      page_hits ++;
        continue;
     }
     if ( filled < capacity) {
          fr[filled++] = pages[i];
```

```
faults++;
} else
{
hits++;
  }
}
printf("FIFO pages faults: %d, page hits: %d\n"
        faults, hit);
int main() {
    int n, frame size;
printf("Enter the size of the pages");
  scanf("%d", &n);
   int pages(n);
  printf("Enter page strings:");
for(int i=0; i<n; i++)
  scanf("%d", & pages(i));
pf("Enter the no of page frames:");
    sf("%d", & frame size(1));
  return 0;
}
```

Output :-

Enter the six of pages : 4
Enter the page strings : 1 30 35 6 3.
FIFO Page Faults : 6
Page hits : 1