VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

OPERATING SYSTEMS

Submitted by

TANISHQ J AMIN (1WA23CS009)

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 TANISHQ J AMIN (1WA23CS009), 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.

Faculty Incharge Seema Patil
Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

Index Sheet

10.	Write a C program to simulate page replacement algorithms a) FIFO b) LRU c) Optimal	57-63

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 Tanisha J Amin Std Sec

1WA23CS009

Roll No. Subject School/College

School/College Tel. No. Parents Tel. No.

Sl. No.	Date	Title	Page No.	Teacher Sign / Remarks
1	06/03/25	CPU scheduling algorithm to Find TAT and WT (a) FCFS (b) SJF (non Preemptive)	10	
2	20/3/25	STF (Preemptive)	10	
3	20/3/25	Round robin, Priority scheduling		
4.	3/4/25	Multilevel Queue	10	
	3/4/25	Rate monotonic		
	3/4/25	earliest deadline first		
5	10/4/25	Producer-consumer	10	
	10/4/25	Dining philosopher		
6	15/5/25	Deadlock avoidance	10	
		Deadlock detection		
7	15/5/25	Memory management (BestFit, WorstFit, FirstFit)	10	
8	15/5/25	FIFO, LRU & optimal	10	
9	15/5/25	Bankers Algorithm	10	

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

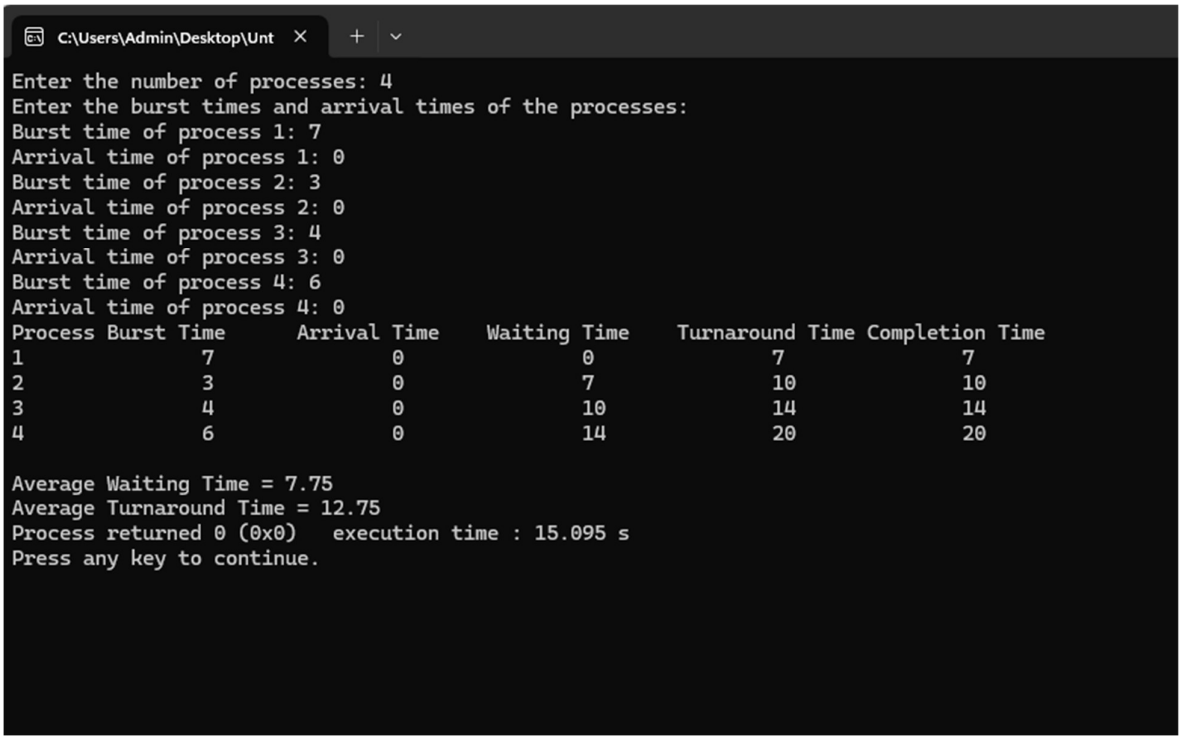
→ SJF (pre-emptive & Non-preemptive)

Code:

=>FCFS:

```
#include <stdio.h>

void findCompletionTime(int processes[], int n, int bt[], int at[], int ct[]) {
    ct[0] = at[0] + bt[0];
    for (int i = 1; i < n; i++) {
        ct[i] = (ct[i - 1] > at[i] ? ct[i - 1] : at[i]) + bt[i];3
    }
}

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

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

void findAvgTime(int processes[], int n, int bt[], int at[]) {
    int wt[n], tat[n], ct[n];
    findCompletionTime(processes, n, bt, at, ct);
    findTurnaroundTime(processes, n, bt, at, ct, tat);
    findWaitingTime(processes, n, bt, at, tat, wt);

    int total_wt = 0, total_tat = 0;

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

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

```
int main() {
    int n;

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

    int processes[n], bt[n], at[n];

    printf("Enter the burst times and arrival times of the processes:\n");
    for (int i = 0; i < n; i++) {
        printf("Burst time of process %d: ", i + 1);
        scanf("%d", &bt[i]);
        printf("Arrival time of process %d: ", i + 1);
        scanf("%d", &at[i]);
        processes[i] = i + 1;  // Process IDs start from 1
    }

    findAvgTime(processes, n, bt, at);

    return 0;
}
```


C:\Users\Admin\Desktop\Unt
Enter the number of processes: 4
Enter the burst times and arrival times of the processes:
Burst time of process 1: 7
Arrival time of process 1: 0
Burst time of process 2: 3
Arrival time of process 2: 0
Burst time of process 3: 4
Arrival time of process 3: 0
Burst time of process 4: 6
Arrival time of process 4: 0
Process Burst Time Arrival Time Waiting Time Turnaround Time Completion Time
1 7 0 0 7 7
2 3 0 7 10 10
3 4 0 10 14 14
4 6 0 14 20 20
Average Waiting Time = 7.75
Average Turnaround Time = 12.75
Process returned 0 (0x0) execution time : 15.095 s
Press any key to continue.

Lab 1

Date __/__/__
Page _____

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

```
#include<stdio.h>

void findcompletiontime (int processes[], int n, int bt[],
        int at[], int ct[]) {

    ct[0] = at[0] + bt[0];
    for(int i=0; i<n; i++) {
     ct[i] = (ct[i-1] > at[i] ? ct[i-1] :
                                at[i]) + bt[i]; }
    }
}
void findTurnaroundtime (int process[], int n,
        int bt[], int at[], int ct[], int TAT[])
{
    for (int i=0; i<n; i++) {

        tat[i] = ct[i] - at[i];
    }
}
void int findWaitingtime (int process[], int n, int bt[],
        int tat[], int wt[]) {
        for (int i=0; i<n; i++) {

            wt[i] = tat[i] - bt[i];
    }
}
```

```
void findAvgTime (int processes[], int n, int bt[],
                  int at[]) {

    int wt[n], tat[n], ct[n];

findCompletionTime (processes, n, bt, at, ct);
findTurnAroundTime (processes, n, bt, at, ct, tat);
findWaitingTime (processes, n, bt, at, tat, wt);

int total_wt = 0, total_tat = 0;

printf ("Proces \t burst time \t Arrival time
\t waiting time \t turnaround time \t completion time \n");

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

printf ("\n Average waiting time = %.2f",
        (float) total_wt / n);

printf ("\n Average turnaround time = %.2f",
        (float) total_tat / n);
}
```

```
int main() {

  int n;
  Printf ("Enter no of process: ");
  Scanf ("%d", &n);

  int process[n], bt[n], at[n];

  printf ("Enter BT & AT of the process: \n");
  for (i=0; i<n; i++) {
    pf ("Burst time of process %d", i+1);
    Scanf ("%d", &bt[i]);
    printf ("Arrival time of process %d", i+1);
    Scanf ("%d", &at[i]);
    process[i] = i+1;
  }

  findAvgTime (process, n, bt, at);

  return 0;
}
```

Output:-

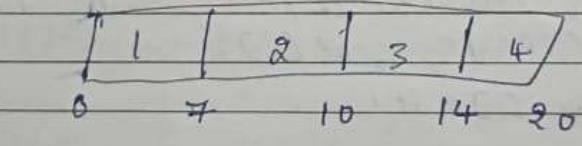
Enter the number of process: 4
Enter the BT & AT of the process:
Burst time of process 1: 7
Arrival time of process 1: 0
Burst time of process 2: 3
Arrival time of process 2: 0
Burst time of process 3: 4
Arrival time of process 3: 0

Date __/__/__
Page _____

Burst time of proces 4: 6
Arrival time of proces 4: 0

Proceses	Burst time	Arrival time	waiting time	TAT	CT
1	7	0	0	7	7
2	3	0	7	10	10
3	4	0	10	14	14
4	6	0	14	20	20

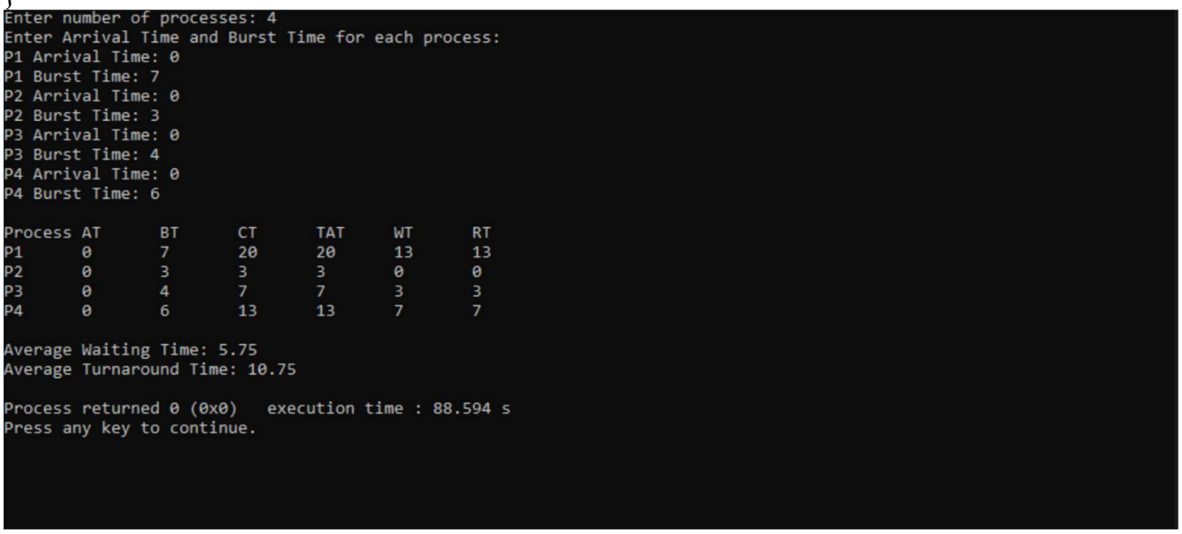
Avege waiting time = 7.75
Average Turnaround Time = 12.25



SJF
Non pre-emptive

```
#include <stdio.h>
#include <limits.h>
struct Process {
    int at;
    int bt;
    int ct;
    int tat;
    int wt;
    int rt;
    int pid;
};
void calculateSJF(struct Process proc[], int n) {
    int time = 0;
    int completed = 0;
    int min_index;
    int is_completed[n];
    for (int i = 0; i < n; i++) {
        is_completed[i] = 0;
    }
    while (completed < n) {
        min_index = -1;
        int min_bt = INT_MAX;
        for (int i = 0; i < n; i++) {
            if (proc[i].at <= time && !is_completed[i] && proc[i].bt < min_bt) {
                min_bt = proc[i].bt;
                min_index = i;
            }
        }
        if (min_index == -1) {
            time++;
        } else {
            proc[min_index].ct = time + proc[min_index].bt;
            proc[min_index].tat = proc[min_index].ct - proc[min_index].at;
            proc[min_index].wt = proc[min_index].tat - proc[min_index].bt;
            proc[min_index].rt = time - proc[min_index].at;
            time = proc[min_index].ct;
            is_completed[min_index] = 1;
            completed++;
        }
    }
}
int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    struct Process proc[n];
    printf("Enter Arrival Time and Burst Time for each process:\n");
    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("P%d Arrival Time: ", i + 1);
        scanf("%d", &proc[i].at);
        printf("P%d Burst Time: ", i + 1);
        scanf("%d", &proc[i].bt);
```

```
    }
    calculateSJF(proc, n);
    printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\tRT\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n", proc[i].pid, proc[i].at, proc[i].bt, proc[i].ct,
proc[i].tat, proc[i].wt, proc[i].rt);
    }
    return 0;
}
```



Date / /
Page

SJF (non-premtive)

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

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

int compare arrival_Time (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 minburst = 100000;
  for (int i = 0; i < n; i++)
}
  if (Process (i). at <= time && process (i). ct
                                    == 0)
```

```
}
    if (Processes[i].bt < min_burst)
    {
        min_burst = process[i].bt;
        shortest = i;
    }
}
    if (shortest == -1) {
        time++;
    }
    else {
    process[shortest].ct = time + process[shortest].bt
                - process[shortest].at;
    process[shortest].wt = process[shortest].bt;
        time = process[shortest].ct;
        completed++;
    }
}

void calculateAvg (struct process process[],
                   int n)
{
    int total_wt = 0, total_tat = 0;
    for (int i=0; i<n; i++) {
        total_wt += process[i].wt;
        total_tat += process[i].tat;
    }
    printf("\n avg wt = %0.2f", (float)total_wt/n);
    printf("\n avg tat = %0.2f", (float)total_tat/n);
```

O/P

```
int main()
{
    int n;
    printf("no of process:");
    scanf("%d", &n);
    struct process p[n];
    printf("Enter BT & AT : \n");
    for(int i=0; i<n; i++){
        p[i].id = i+1;
        printf("bt %d: ", i+1);
        scanf("%d", &p[i].bt);
        printf("at %d", i+1);
        scanf("%d", &p[i].at);
        p[i].ct=0;
    }
    qSort(p, n, sizeof(struct process));
    compute arrival time();
    calculateTimes(p, n);
    calculateAvg(p, n);

    return 0;
}
```

O/P:

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

P	AT	BT	CT	TAT	WT	RT
P_1	0	7	7	7	0	0
P_2	8	3	14	6	3	3
P_3	3	4	11	8	4	5
P_4	5	6	20	15	9	9

SJF pre-emptive

```
#include <stdio.h>

#define MAX 10

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

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

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

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

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

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

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

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

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

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

```
    sjf_preemptive(p, n);

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

    return 0;
}
```

```
Enter number of processes: 4
Enter arrival time and burst time for 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.
```

Date __/__/__
Page _____

(5) SJF((Prempitive)

```
#include <studio.h>

struct process {
    int id; int BT; int AT; int CT;
    int ST;
};
int compareAT ( const void *a, void *b){
  return ((struct procers*)) a -> AT - ST(b))AT
}
int compare RT (const void*a, const void *b) {
  retur (Struct Procers*) a) -> RT - ((Struct Procers)
      compare AT);
while (completd procers < num Procers) {
  int index = -1;
for (int i = 0; i < numprocers ; i++) {
  if (procers (i). at <= CT && procers (i). RT > 0) {
      if (minimdex == -1)
          minindex = i;
    }
}
    if (minIndex == -1) {
        CT;
        continue;

Procers [minIndex]. RT --;
    current time ++;

if (Procers (minIndex). RT == 0) {
    procers (minIndex) . CT = current time;
    completed procount++;
```

```
    printf("Process completed");
}
int main() {
    struct process (max_processes);
    int numprocess;
for(int i=0; i<numprocess; i++){
    process(i).id = i+1;
pf("Enter AT");
sf("%d", &process(i).BT);
process(i).RT = process(i).BT;
}
    preemptiveSJF(Process, numprocesses);
    return 0;
}
```

O/P

```
Enter number of process : 4

Enter AT of process 1 : 0
Enter BT of process 1 : 8
Enter AT of process 2 : 1
Enter BT of process 2 : 4
Enter AT of process 3 : 2
Enter BT of process 3 : 9
Enter AT of process 4 : 3
Enter BT of process 4 : 5
```

Process	AT	BT	CT	TAT	WT	RT
1	0	8	17	17	9	0
2	1	4	5	4	0	0
3	2	9	26	24	15	15
4	3	5	10	7	2	2

avg tat : 13.0 avg wt = 6.50

Priority Non-pre-emptive

```
#include <stdio.h>

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

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

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

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

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

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

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

    printf("PID\tAT\tBT\tPR\tCT\tTAT\tWT\tRT\n");
```

```
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\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;
}
```

```
Enter the number of processes: 5
Enter Arrival Time, Burst Time, and Priority for each process:
Process 1: 0
10
3
Process 2: 0
1
1
Process 3: 0
2
4
Process 4: 0
1
5
Process 5: 0
5
2
PID AT  BT  PR  CT  TAT WT  RT
1   0   10  3   16  16  6   6
2   0   1   1   1   1   0   0
3   0   2   4   18  18  16  16
4   0   1   5   19  19  18  18
5   0   5   2   6   6   1   1

Average Turnaround Time: 12.00
Average Waiting Time: 8.20
```

Lab-2

(13/3/25)

Date
Page

3 Priority scheduling of non-primtive

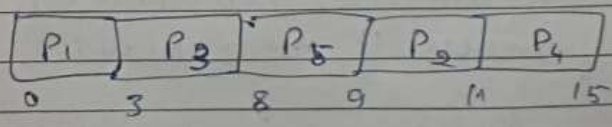
```
#include <stdio.h>
int main() {
int n, i, j;
int burst (20), Priority (20), process (20)
int wait (20), turnaround (20), completion (20)
int total_wt = 0, total_tat = 0;
printf ("Enter the no of processes: ");
scanf ("%d", &n);
printf ("Enter burst time & priority from
        each process: \n");
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 (j) = temp;
       temp = process;
       print
        Process (i) = process (i)
        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)
    printf("In process %d", i);
    for(i=0; i<n; i++){
    printf("%d Process(i), burst(i), priority(i),
        wait(i), turnaround(i));
}
printf("In average time: %0.2f", (float)
                total_wait/n);
printf("In Avege turnaround time: %0.2f\n",
        (float) total_turnaround/n);

    return 0;
}
```

O/P

Process	AT	BT	Pri	CT	TAT	WT	RT
P_1	0	3	5	3	3	0	0
P_2	2	2	3	11	9	7	7
P_3	3	5	2	8	5	0	0
P_4	4	4	4	15	11	7	7
P_5	6	1	1	9	3	2	2

Avg = 6.2 Avg = 3.2



Priority pre

```
#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;
}
```

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

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

(4) Priority (Premitive)

```
#include <stdio.h>

typedef struct {
    int pid, at, 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(i+1), AT &&
        swap (&process(i), &process(i-1);
    }
}

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

Date __/__/__
Page _____

```
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
  = process(selected).completion_time(selected)
                    . arrival time;
process(selected).waiting_time = process(selected)
.TA_process(selected).BT;

processes(selected).WT = process(selected).
                    TA_process(selected).BT;
}
}
```

Date / /
Page

No of process: 7
Enter BT and AT and PT:
BT 1 : 8
AT 1 : 0
PT 1 : 3
BT 1 : 9
AT 2 : 1
PT 2 : 4
BT 1 : 4
AT 3 : 3
BT 3 : 4
PT 3 : 1
AT 4 : 4
BT 4 : 5
PT 4 : 6
AT 5 : 5
BT 5 : 2
PT 5 : 5
AT 6 : 6
BT 6 : 6
PT 6 : 1
AT 7 : 7
BT 7 : 1
PT 7 : 4

AVG TAT : 13.71
AVG WT : 4.86

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

```
#include <stdio.h>

#define MAX_PROCESSES 10

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;
}
```

3/04/2025

Lab 4

Date / /
Page

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 process: ");
  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;
  }

Pf("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~~
        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 -= 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_remaing[i] > 0
        && ps[i].at <= currentTime)
        {
            done = 0;
            currentTime += bt_remaing[i];
            bt_remaing[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("\n Process \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 \t",
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);
}
```

O/P:

```
Enter the number of process: 4
Enter AT, BT, and Queue (1: RR, 2: FCFS): 0 2 1
Enter AT, BT and Queue (1: RR, 2: FCFS): 0 1 2
Enter AT, BT and Queue (1: RR, 2: FCFS): 0 5 1
Enter AT, 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

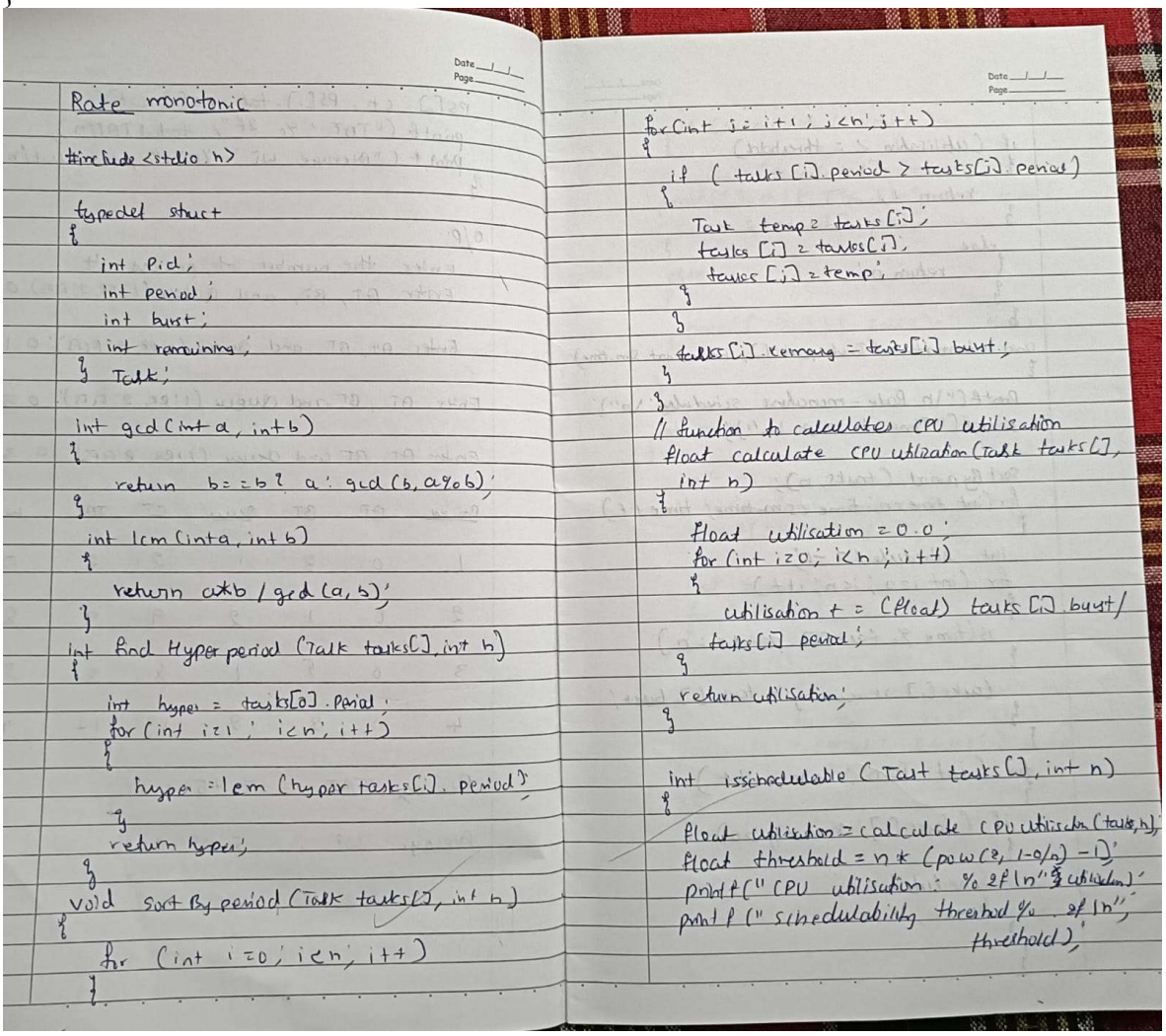
Average TAT = 7.75 s

Average WT : 5.00 s

RATE MONOTONIC

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

#define MAX_PROCESSES 10
#define MAX_TIME 50  // Maximum simulation time

typedef struct {
    int pid;
    int burst;
    int period;
    int remaining_time;
    int next_arrival;
} Process;

void rate_monotonic_scheduling(Process p[], int n) {
    int time = 0, executed;
    printf("\nRate Monotonic Scheduling:\n");
    printf("PID\tBurst\tPeriod\n");
    for (int i = 0; i < n; i++)
        printf("%d\t%d\t%d\n", p[i].pid, p[i].burst, p[i].period);

    while (time < MAX_TIME) {
        executed = -1;

        for (int i = 0; i < n; i++) {
            if (p[i].next_arrival <= time && p[i].remaining_time > 0) {
                if (executed == -1 || p[i].period < p[executed].period)
                    executed = i;
            }
        }

        if (executed != -1) {
            printf("%dms : Task %d is running.\n", time, p[executed].pid);
            p[executed].remaining_time--;
            if (p[executed].remaining_time == 0) {
                p[executed].next_arrival += p[executed].period;
                p[executed].remaining_time = p[executed].burst; // Reset for periodic execution
            }
        }
        time++;
    }
}

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

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

    printf("Enter the CPU burst times:\n");
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
```

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

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

  rate_monotonic_scheduling(processes, n);

  return 0;
}
```



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 find_Hyperperiod(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 sort_By_period(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].remaing = tasks[i].burst;
  }
}

// function to calculates CPU utilisation
float calculate_CPU_utilization(Task tasks[],
  int n)
{
  float utilisation = 0.0;
  for(int i=0; i<n; i++)
  {
    utilisation += (float) tasks[i].burst/
    tasks[i].period;
  }
  return utilisation;
}

int isSchedulable (Task tasks[], int n)
{
  float utilisation = calculate_CPU_utilisation(tasks,n);
  float threshold = n * (pow(2, 1.0/n) - 1);
  printf("CPU utilisation: %.2f\n", utilisation);
  printf("scheduability threshold %.2f\n",
                threshold);
```

```
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

```
#include <stdio.h>

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

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

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

void earliest_deadline_first(struct Process p[], int n, int time_limit) {
    int time = 0;
    printf("Earliest Deadline Scheduling:\n");
    printf("PID\tBurst\tDeadline\tPeriod\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", p[i].id, p[i].burst_time, p[i].deadline, p[i].period);
    }

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

        if (earliest == -1) break;

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

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

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

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

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

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

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

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

    earliest_deadline_first(processes, n, hyperperiod);

    return 0;
```

```
Enter the number of processes: 3
Enter the CPU burst times:
2 3 4
Enter the deadlines:
1 2 3
Enter the time periods:
1 2 3

System will execute for hyperperiod (LCM of periods): 6 ms
Earliest Deadline Scheduling:
PID     Burst   Deadline        Period
1       2               1               1
2       3               2               2
3       4               3               3

Scheduling occurs for 6 ms
0ms: Task 1 is running.
1ms: Task 1 is running.
2ms: Task 2 is running.
3ms: Task 2 is running.
4ms: Task 2 is running.
5ms: Task 3 is running.

Process returned 0 (0x0)   execution time : 9.656 s
Press any key to continue.
```

```
}
```

```
total TAT += ps[idx].tat;
total WT += ps[idx].wt;
completed++;
}
else
{ currentTime++;
}
}
printf("In process \t at \t BT \t DL \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 \t",
ps[i].id, ps[i].at, ps[i].bt, ps[i].dt,
ps[i].ct, ps[i].ct, ps[i].tat, ps[i].wt);
}
printf("\n Avg TAT : %.2f \n", totalTAT/n);
printf("\n Avg WT : %.2f \n", totalWT/n);
return 0;
}
```

Q8

```
Enter the no of process : 3
Enter arrival, Burst & deadline : 0 4 6
Enter arrival, Burst & deadline : 1 3 5
Enter arrival, Burst & deadline : 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

Avg TAT : 5.33 Avg WT : 2.33

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

```
#include <stdio.h>
#include <stdlib.h>
int mutex = 1;
int full = 0;
int empty = 3;
int 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("Producer produces item %d\n", x);
   mutex = signal(mutex);
}
void consumer() {
   mutex = wait(mutex);
   full = wait(full);
   empty = signal(empty);
   printf("Consumer consumes 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;
}
```

```
1. Producer
2. Consumer
3. Exit
Enter your choice: 1
Producer produces item 1

1. Producer
2. Consumer
3. Exit
Enter your choice: 1
Producer produces item 2

1. Producer
2. Consumer
3. Exit
Enter your choice: 1
Producer produces item 3

1. Producer
2. Consumer
3. Exit
Enter your choice: 1
Buffer is full!

1. Producer
2. Consumer
3. Exit
Enter your choice: 2
Consumer consumes item 3

1. Producer
2. Consumer
3. Exit
Enter your choice: 1
Producer produces item 3

1. Producer
2. Consumer
3. Exit
Enter your choice: 2
Consumer consumes item 3

1. Producer
2. Consumer
3. Exit
Enter your choice: 2
Consumer consumes item 2

1. Producer
2. Consumer
3. Exit
Enter your choice: 2
Consumer consumes item 1

1. Producer
2. Consumer
3. Exit
Enter your choice: 2
Buffer is empty!

1. Producer
2. Consumer
```

Lab 5

Date __/__/__
Page _____

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);
    Pf("consumed item %d\n", x);
    x--;
    mutex = signal(mutex);
}
```

```
int main()
{
  int choice;
  while(1)
  {
   Pf("\n 1. Producer \n 2. Consumer
\n 3 exit");
   pf("Enter your choice:");
   scanf("%d", &choice);

   switch(choice);
   {
    case 1: if((mutex==1)&&(empty!=0))
              producer();
    else
       printf("Buffer free");
    break;

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

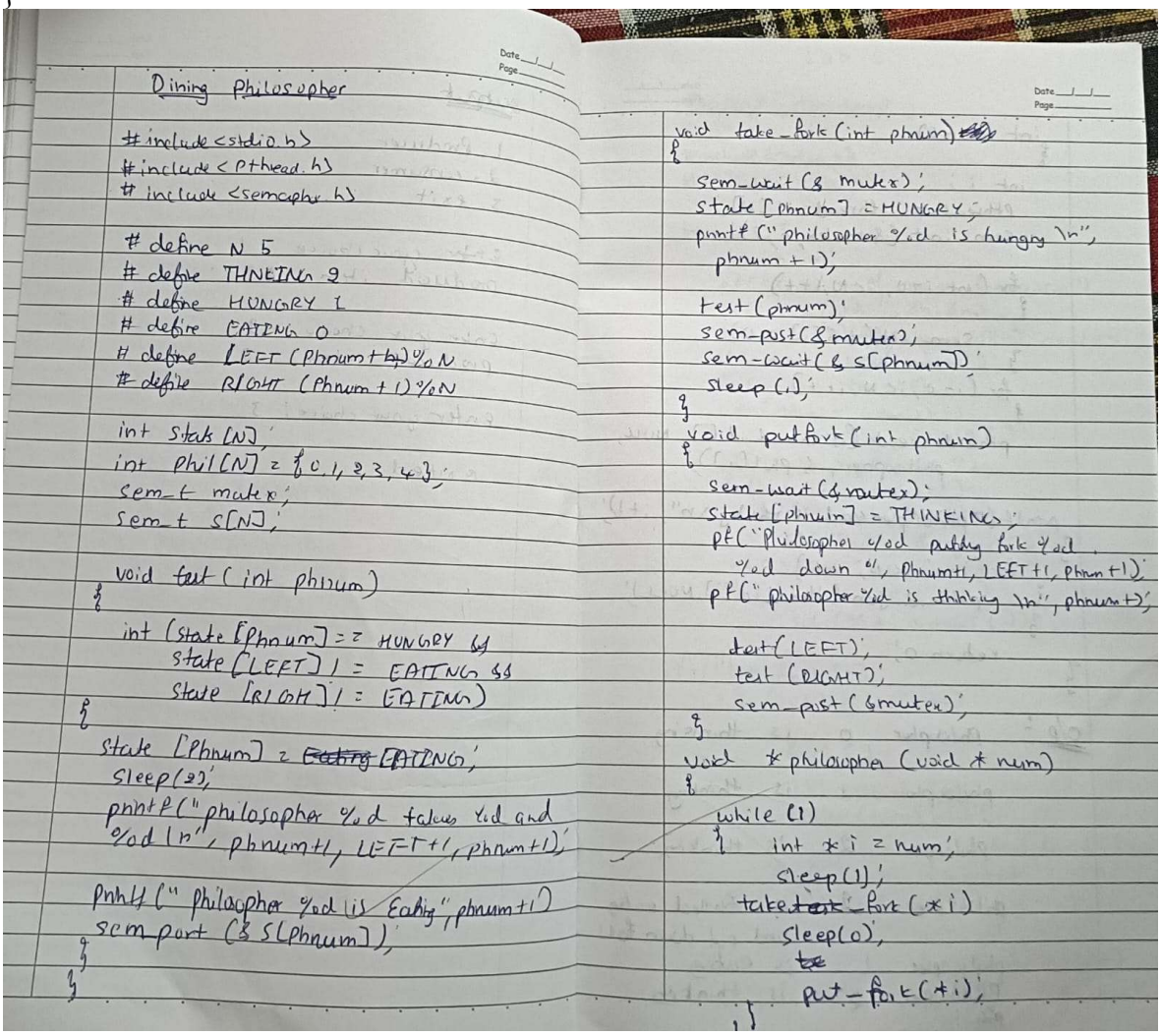
output

```
1. Producer
2. consumer
3. exit

Enter your choice : 1
produced item : 21

Enter your choice : 2
produced item : 2

Enter your choice : 3

exited
```

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

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
int totalPhilosophers;
int hungry[MAX];
int areNeighbors(int a, int b) {
   return (abs(a - b) == 1 || abs(a - b) == totalPhilosophers - 1);
}
void option1(int count) {
   printf("\nAllow one philosopher to eat at any time\n");

   for (int i = 0; i < count; i++) {
      printf("P %d is granted to eat\n", hungry[i]);
      for (int j = 0; j < count; j++) {
         if (j != i) {
            printf("P %d is waiting\n", hungry[j]);
         }
      }
   }
}
void option2(int count) {
   printf("\nAllow two philosophers to eat at same time\n");
   int combination = 1;
   for (int i = 0; i < count; i++) {
      for (int j = i + 1; j < count; j++) {
         if (!areNeighbors(hungry[i], hungry[j])) {
            printf("combination %d\n", combination++);
            printf("P %d and P %d are granted to eat\n", hungry[i], hungry[j]);
            for (int k = 0; k < count; k++) {
               if (k != i && k != j) {
                  printf("P %d is waiting\n", hungry[k]);
               }
            }
            printf("\n");
         }
      }
   }
   if (combination == 1) {
      printf("No combinations found where two non-neighbor philosophers can eat.\n");
   }
}
int main() {
   int hungryCount;
   printf("DINING PHILOSOPHER PROBLEM\n");
   printf("Enter the total no. of philosophers: ");
   scanf("%d", &totalPhilosophers);
   printf("How many are hungry: ");
   scanf("%d", &hungryCount);
   for (int i = 0; i < hungryCount; i++) {
      printf("Enter philosopher %d position: ", i + 1);
      scanf("%d", &hungry[i]);
   }
   int choice;
```

```
    do {
        printf("\n1. One can eat at a time   2. Two can eat at a time   3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            option1(hungryCount);
            break;
        case 2:
            option2(hungryCount);
            break;
        case 3:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice!\n");
        }
    } while (choice != 3);

    return 0;
}
```



Dining Philosopher

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.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)
{
    int (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;
    pf("Philosopher %d putting fork %d
    %d down", phnum+1, LEFT+1, phnum+1);
    pf("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;
}
```

o/p :

philosopher 0 is thinking

philosopher 1 is thinking

philosopher 2 is thinking

philosopher 2 is finished eating and put down forks

philosopher 1 is eating

Philosopher 2 is thinking

etc ;

loop.

Lab 6

Deadlock avoidance

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

int main(){
    int n, m;
    pf("Enter no of process & resources");
    sf("%d%d", &n, &m);
    int alloc[n][m], request[n][m], avail[m];
    pf("Enter allocation matrix");
    for(i=0; i<n; i++)
        for(int j=0; j<m; j++)
            sf("%d", &alloc[i][j]);
    pf("Enter req matrix:");
    for(int i=0; i<n; i++)
        for(int j=0; j<m; j++)
            sf("%d", &request[i][j]);
    pf("Enter avail matrix");
    for(int i=0; i<m; i++)
        sf("%d", &avail[i]);

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

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

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

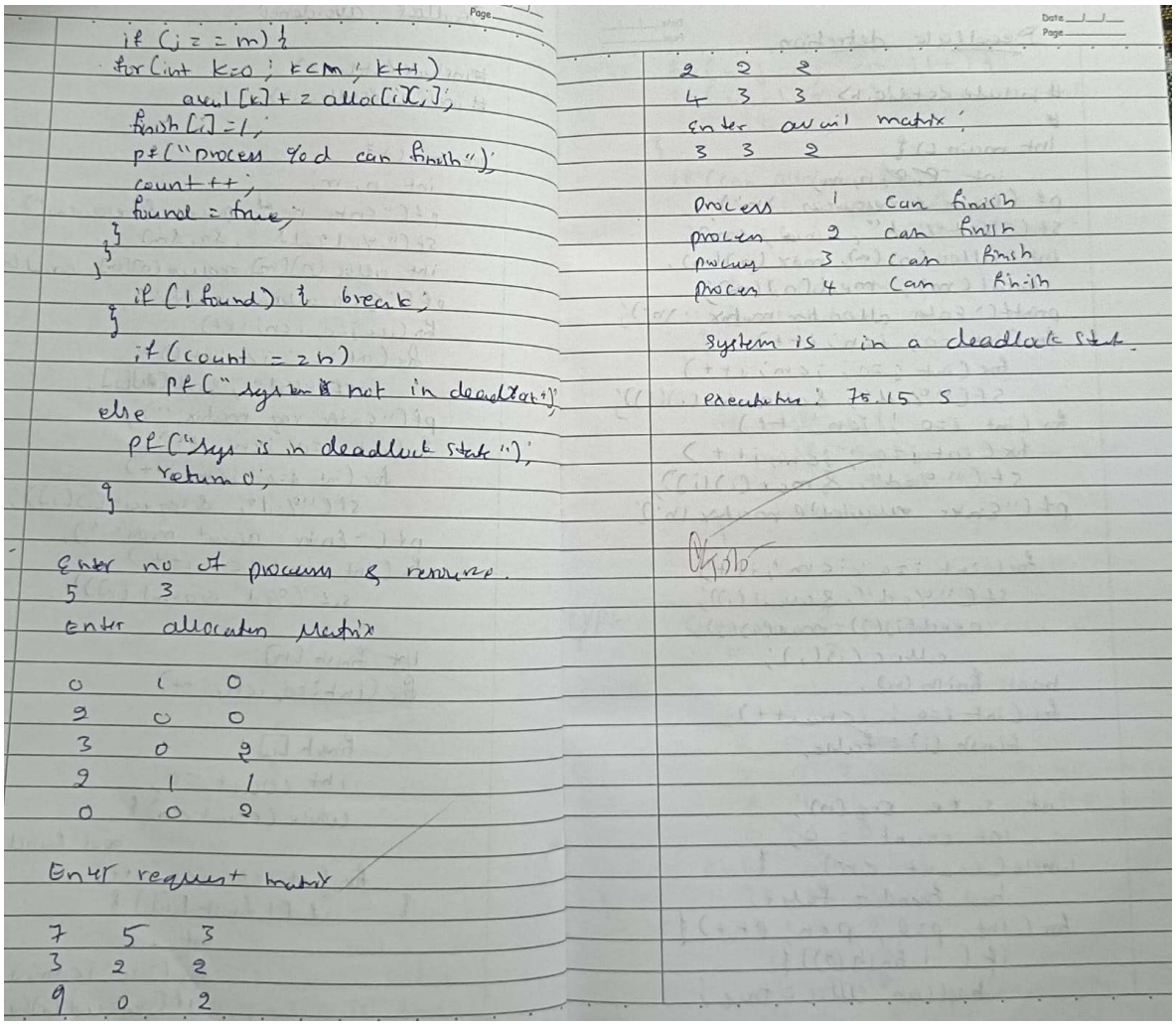
```
#include <stdio.h>

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++)
        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;
}
```

Banker Algorithm

```
#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;

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

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

Pf("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][
        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])
```

Write a C program to simulate deadlock detection



```
if (j == m) {
for(int k=0; k<m; k++)
    avail[k] += alloc[i][j];
finish[i] = 1;
p+("Process %d can finish");
count++;
found = true;
}
}
if (!found) { break;
}
if (count == n)
  p+("system is not in deadlock");
else
  p+("sys is in deadlock state");
  return 0;
}

Enter no of process & resource.
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 avail matrix;
3  3  2

Process 1 can finish
process 2 can finish
process 3 can finish
process 4 can finish

system is in a deadlock state.

Execution: 75.15 s
```

Deadlock detection

```
#include <stdio.h>
#
int main(){
    int n, m;
pf("Process & resources: \n");
sf("%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++)
    sf("%od", &allocation(i)(j));
for(int i=0; i<n; i++)
    for(int j=0; j<m; j++)
    sf("%od", &max(i)(j));
pf("Enter available matrix \n");

for(int i=0; i<m; i++)
    sf("%od", &avail(i));
    need(i)(i)=max(i)(i),
      alloc(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 can alloc = false;
        break;
}
if(can Allocate){
    for(int k=0; k<(m; k++)
    avail(k) + = alloc(p)(k);
    safe seq(count++)=p;
    finish(p) = true;
    found = true;
}
}
}
    if(!found){
        printf("System is not in safe
              State: ");
        return 1;
    }
}
```

O/P

```
Enter no of process & resources.
  5   3
Enter allocation matrix:
  0   1   0
  2   0   0
  3   0   2
  2   1   1
  0   0   2
Enter max matrix:
  7   5   3
  3   2   2
  9   0   2
  2   2   2
  4   3   3
```

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

Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit
d) Best-fit
e) 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("\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;
}
```

Memory management Lab 7

Best fit, first fit, worst fit.

Date / /
Page

```
#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  Filesize  block no ");

for(int i=0 ; i<n_files, i++){
int best_fit_block = -1, min_fragment=1000;

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

if(fragment < min_fragment){
min_frag = fragment;
bestfitblock = j;
}
}
}
```

Date / /
Page

```
if(bestfitblock != -1){

blocks[bestfitblock].isfree = 0;
printf("%d\t%d\t\t%d\t%d\t\n",

files[i].fileno,
files[i].filesize,
blocks[bestfitblock].blockno,
blocks[bestfitblock].blocksize,
min_fragment);
}
else{
pf("%d\t%d\tN/A\t\tN\t\tN/A\n"

files[i].fileno, files[i].filesize);
}
}

void firstfit(struct Block blocks[],
int nblocks, struct File files[], int n_files)

pf("\n Memory Mgmt scheme - First fit\n
pf("File no\t file\tblocksize\tsize\tfragment\n");

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

for(int j=0; j<nblocks; j++){
if(blocks[j].is free && block[j].
blocksize >= files[i].filesize){

int fragment=block[j].blocksize -
        files[i].filesize;
```

```
blocks[j].isfree=0;
Pf("%d\t%d\t%d\t%d\n",
    files[i].fileno,
    files[i].filesize,
    blocks[j].fileno,
    blocks[j].filesize,
    fragment);
    allocated=1;
    break;
}
}
    if(!allocated){
Pf("%d\t%d\t\t\tN/A\t\t\tN/A
    \t\t\tN/A\n", files[i].fileno,
        files[i].filesize);
}
}
}
void WorstFit(struct Block blocks[],
int nblocks, struct File files[], int nfiles){

Pf("In memory management scheme -
    Worst fit:");

for(int i=0;i<nfiles;i++){
    int worstfitblock=-1, maxfrag=-1;

for(int j=0;j<nblocks;j++){
    if(fragment>maxfrag){
maxfrag=fragment;
worstfitblock=j;
}
}
}
    if(worstfitblock!=-1){
blocks[worstfitblock].isfree=0;

printf("%d\t%d\t%d\t%d\n",
    files[i].fileno,
    files[i].filesize,
    block[worstfitblock].blockno,
    block[worstfitblock].blocksize,
    maxfrag);
}
    else{
printf("%d\t%d\t\t\tN/A\t\t\tN/A\
    t\t\tN/A\n"),
    files[i].fileno, files[i].filesize);
}
}
}
void reset Blocks(struct Block bl[],
    int nblocks){
for(int i=0;i<nblocks;i++){
    blocks[i].isfree=1;
}
}
```

```
int main(){

int n_blocks, nfiles;
pf("Enter the no of block:");
sf("%d", &nblocks);

pf(" enter the number of files:");
sf("%d", &nfiles);

struct Blakblock [n_blocks];
struct Filesblks [nfiles];

for(int i=0; i<nblocks; i++){
blocks[i].blockno = i+1;
pf("Enter the size of bk", i+1);
sf("%d", &block[i].blocksize);
blocks[i].isfree = 1;
}
for(int i=0; i<nfiles; i++){
files[i].fileno = i+1;
pf("Enter size", i+1);
sf("%d", &file[i].filesize);
}
resetBlocks(blocks, n blocks);
bestfit(blocks, nblocks, files, nfiles);

resetBlocks(blocks, nblocks)
firstfit(blocks, nblock, files, nfiles);

resetblock(blocks, n blocks)
WORSTfit(blocks, nblock, file, nfiles);

return 0;
}
```

O/P

```
Enter the size of the blocks:
Blocks 1 : 100
Blocks 2 : 500
Blocks 3 : 200
Blocks 4 : 300
Blocks 5 : 600

Enter the size of the file

File 1 : 212
File 2 : 417
File 3 : 112
File 4 : 426

1. First Fit
2. Best Fit
3. Worst Fit
4. Exit

Enter your choice: 1

Memory managment scheme First fit
```

File no	File size	Block no	Block size
1	212	2	500
2	417	5	600
3	112	3	200
4	426	-	-

Date __/__/__
Page _____

Enter your choice: 2

Memory management Scheme Best fit

File no	file size	block no	block size
1	212	4	300
2	417	2	500
3	112	3	200
4	426	5	600

Enter your choice: 3

Memory management scheme worst fit.

File no	File size	block no	block size
1	212	5	600
2	417	2	500
3	112	4	300
4	426	-	=

Write a C program to simulate page replacement algorithms a) FIFO
d) LRU
e) 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 simulateFIFO(int pages[], int n, int frameSize) {
    int frame[frameSize], front = 0, 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) {
            frame[front] = pages[i];
            front = (front + 1) % frameSize;
            count++;
        } else {
            hits++;
        }
    }
    printf("FIFO Page Faults: %d, Page Hits: %d\n", count, hits);
}

void simulateLRU(int pages[], int n, int frameSize) {
    int frame[frameSize], time[frameSize], count = 0, hits = 0;
```

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

    for (int i = 0; i < n; i++) {
        int pos = search(pages[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] = pages[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 index = -1;
            for (int j = 0; j < frameSize; j++) {
                if (frame[j] == -1) {
                    index = j;
                    break;
                }
            }
            if (index != -1) {
                frame[index] = pages[i];
            } else {
                int replaceIndex = findOptimal(pages, frame, n, i + 1, frameSize);
                frame[replaceIndex] = 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: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &pages[i]);

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

    simulateFIFO(pages, n, frameSize);
    simulateOptimal(pages, n, frameSize);
    simulateLRU(pages, n, frameSize);

    printf("\nProcess returned 0 (0x0) Press any key to continue.\n");
    return 0;
}
```

1. FIFO, 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 simulate FIFO (int pages[], int n,
int framesize) {

int frame[framesize], front = 0, count = 0,
hits = 0;

for (int i = 0; i < n; i++) {
  if (search(pages[i], frame, framesize) == -1) {
    frame[front] = pages[i];
    front = (front + 1) % framesize;
    count++;
  }
  else {
    hits++;
  }
}
printf("FIFO page Faults %d, page
  Hits %d \n", count, hits);
}

void simulate LRU (int pages[], int n, int
  framesize) {
int frame[framesize], time[framesize],
count = 0, hits = 0;

for (int i = 0; i < framesize; i++) {
  frame[i] = -1;
  time[i] = 0;
}
for (int i = 0; i < n; i++) {
  int pos = search(pages[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] = pages[i];
  time[least] = i;
  count++;
  }
  else {
      hits++
      time[pos] = i;
  }
  } printf("LRU Page Faults:
    %d, Page hits, %d \n", count, hits);
}
Void simulate optimal (int page[], 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(page[i], frame, framesize) == -1) {
    int index = -1;
for (int j = 0; j < framesize; j++) {
    if (frame[j] == -1) {
        index = j;
        break;
    }
  }
```

```
        if (index != -1) {
            frame[index] = pages[i];
        } else {
            int replaceIndex = findOptimal(pages,
            frame, n, i+1, frameSize);
            frame[replaceIndex] = 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:");
    for(int i=0; i<n; i++)
        scanf("%d", &pages[i]);

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

    simulateFIFO(pages, n, frameSize);
    simulateOptimal(pages, n, frameSize);
    simulateLRU(pages, n, frameSize);
}
return 0;
```

Output :-

```
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
```