VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT

on

OPERATING SYSTEMS

Submitted by

SUSHMA BT (1WA23CS002)

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 Sushma B T (1WA23CS002), who is Bonafide student of B. M. S. College of Engineering. It is in partial fulfilment for the award of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum during the year Feb 2025- June 2025. The Lab report has been approved as it satisfies the academic requirements in respect of a OPERATING SYSTEMS - (23CS4PCOPS) work prescribed for the said degree.

Seema patil Dr. Kavitha Sooda

Assistant Professor Professor and Head

Department of CSE Department of CSE

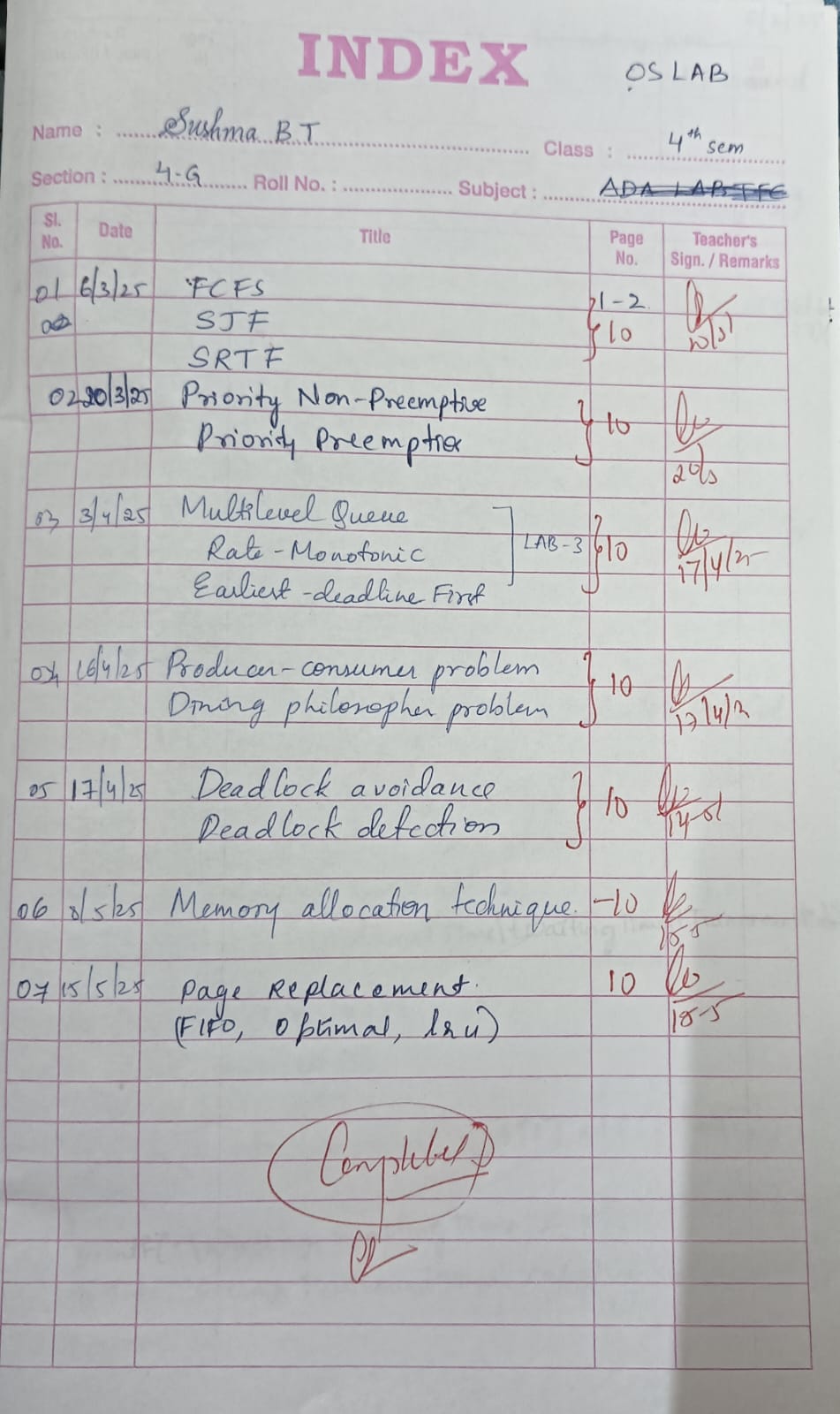
BMSCE, Bengaluru BMSCE, Bengaluru

Index Sheet

|  |  |  |
| --- | --- | --- |
| Sl.  No. | Experiment Title | Page No. |
| 1. | Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.  →FCFS  → SJF (pre-emptive & Non-preemptive) | 1-10 |
| 2. | Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.  → Priority (pre-emptive & Non-pre-emptive)  →Round Robin (Experiment with different quantum sizes for RR algorithm) | 11-19 |
| 3. | Write a C program to simulate the following scheduling   1. Multilevel queue 2. Rate monotonic 3. Earliest deadline first | 20-24 |
| 4. | a) Write a C program to simulate producer-consumer problem using semaphores  b) Write a C program to simulate the concept of Dining Philosophers problem | 24-34 |
| 5. | Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance and deadlock detection | 35-38 |
| 6. | Write a C program to simulate the following contiguous memory allocation techniques  a) Worst-fit   1. Best-fit   c)First-fit | 39-43 |
| 7. | Write a C program to simulate page replacement algorithms a) FIFO   1. LRU   Optimal | 44-47 |

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. |



LAB-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>

struct Process {

int pid;

int arrival\_time;

int burst\_time;

int waiting\_time;

int turnaround\_time;

};

void fcfs(struct Process proc[], int n) {

int completion\_time[n];

// First process

completion\_time[0] = proc[0].arrival\_time + proc[0].burst\_time;

proc[0].turnaround\_time = completion\_time[0] - proc[0].arrival\_time;

proc[0].waiting\_time = proc[0].turnaround\_time - proc[0].burst\_time;

// Remaining processes

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

completion\_time[i] = (completion\_time[i - 1] > proc[i].arrival\_time ? completion\_time[i - 1] : proc[i].arrival\_time) + proc[i].burst\_time;

proc[i].turnaround\_time = completion\_time[i] - proc[i].arrival\_time;

proc[i].waiting\_time = proc[i].turnaround\_time - proc[i].burst\_time;

}

}

void display(struct Process proc[], int n) {

printf("\nPID\tArrival\tBurst\tWaiting\tTurnaround\n");

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

printf("%d\t%d\t%d\t%d\t%d\n", proc[i].pid, proc[i].arrival\_time, proc[i].burst\_time, proc[i].waiting\_time, proc[i].turnaround\_time);

}

// Calculate average waiting time and turnaround time

int total\_waiting\_time = 0, total\_turnaround\_time = 0;

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

total\_waiting\_time += proc[i].waiting\_time;

total\_turnaround\_time += proc[i].turnaround\_time;

}

double avg\_waiting\_time = (double)total\_waiting\_time / n;

double avg\_turnaround\_time = (double)total\_turnaround\_time / n;

// Display average waiting and turnaround times

printf("\nAverage Waiting Time: %.2f", avg\_waiting\_time);

printf("\nAverage Turnaround Time: %.2f\n", avg\_turnaround\_time);

}

int main() {

int n, choice;

printf("Enter the 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("Process %d: ", i + 1);

scanf("%d %d", &proc[i].arrival\_time, &proc[i].burst\_time);

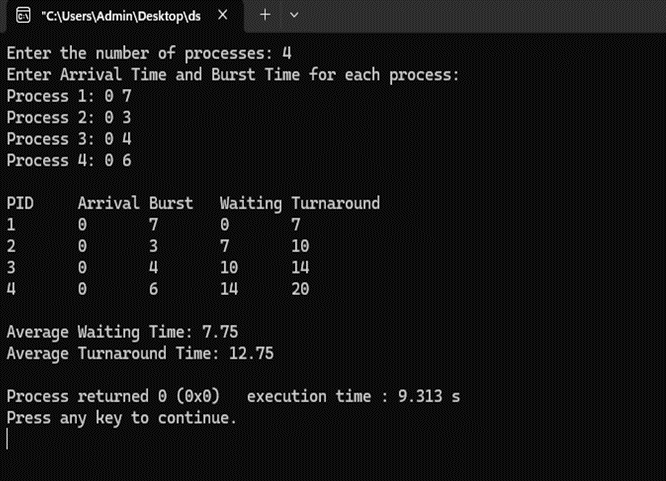
}

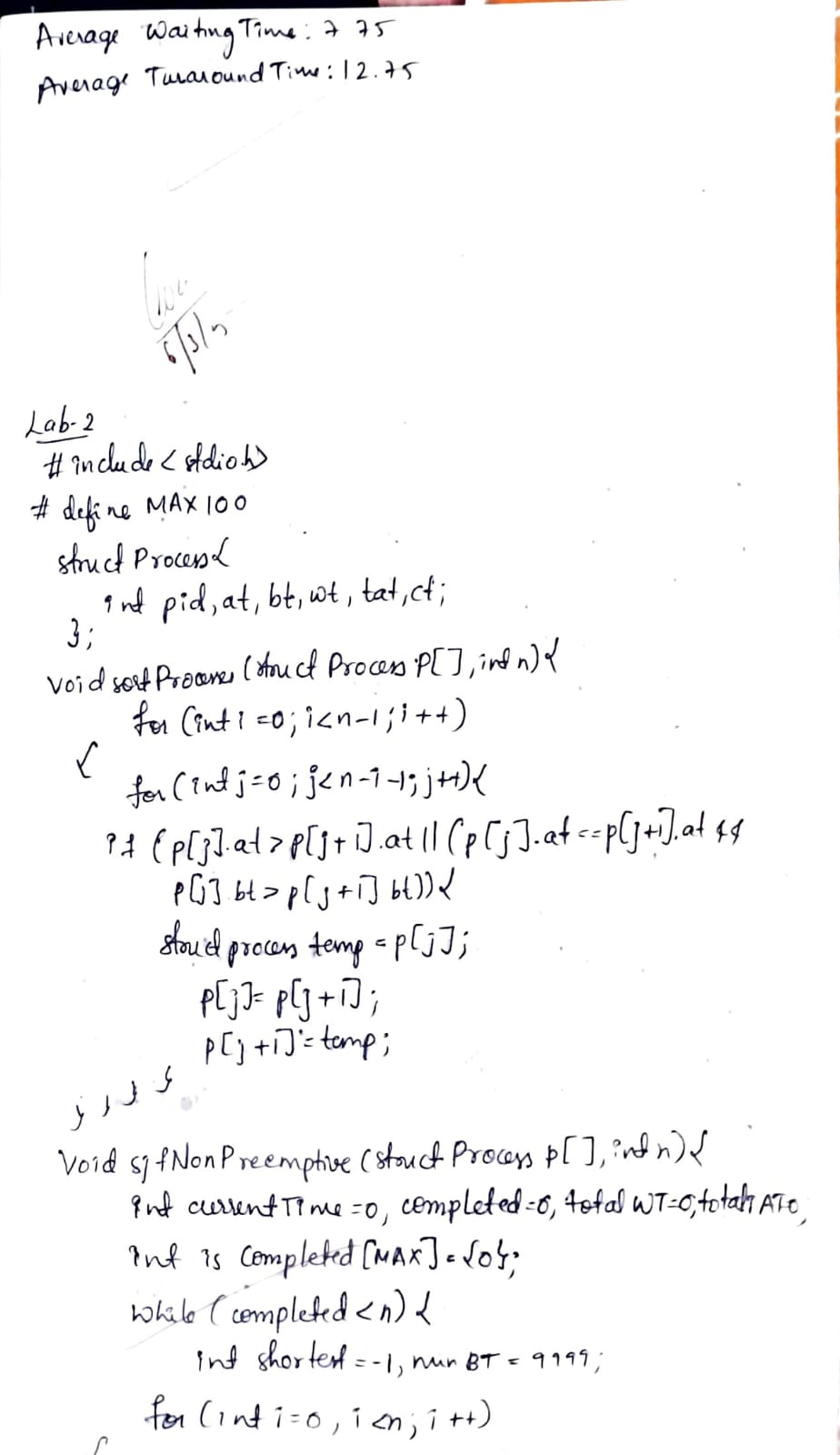
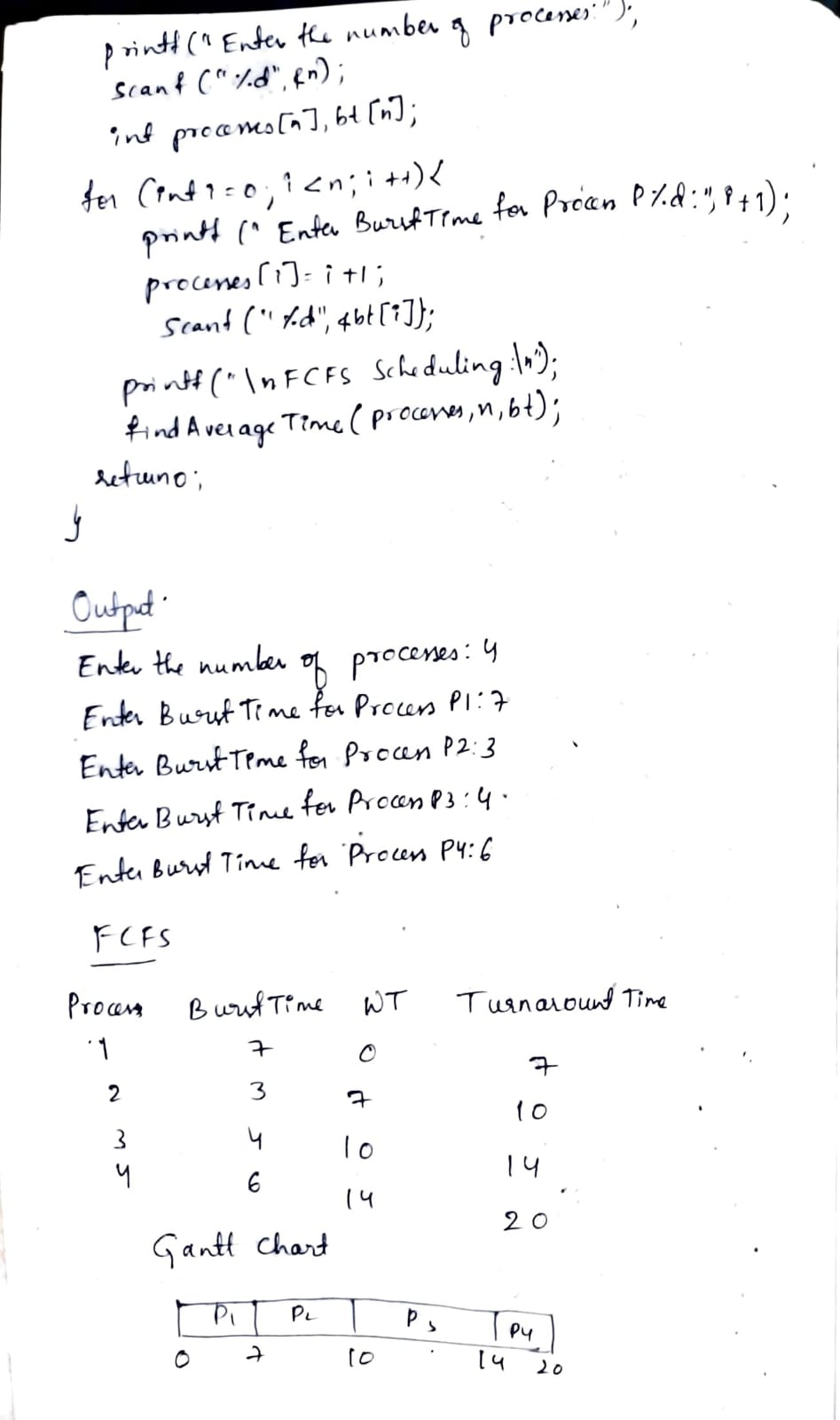
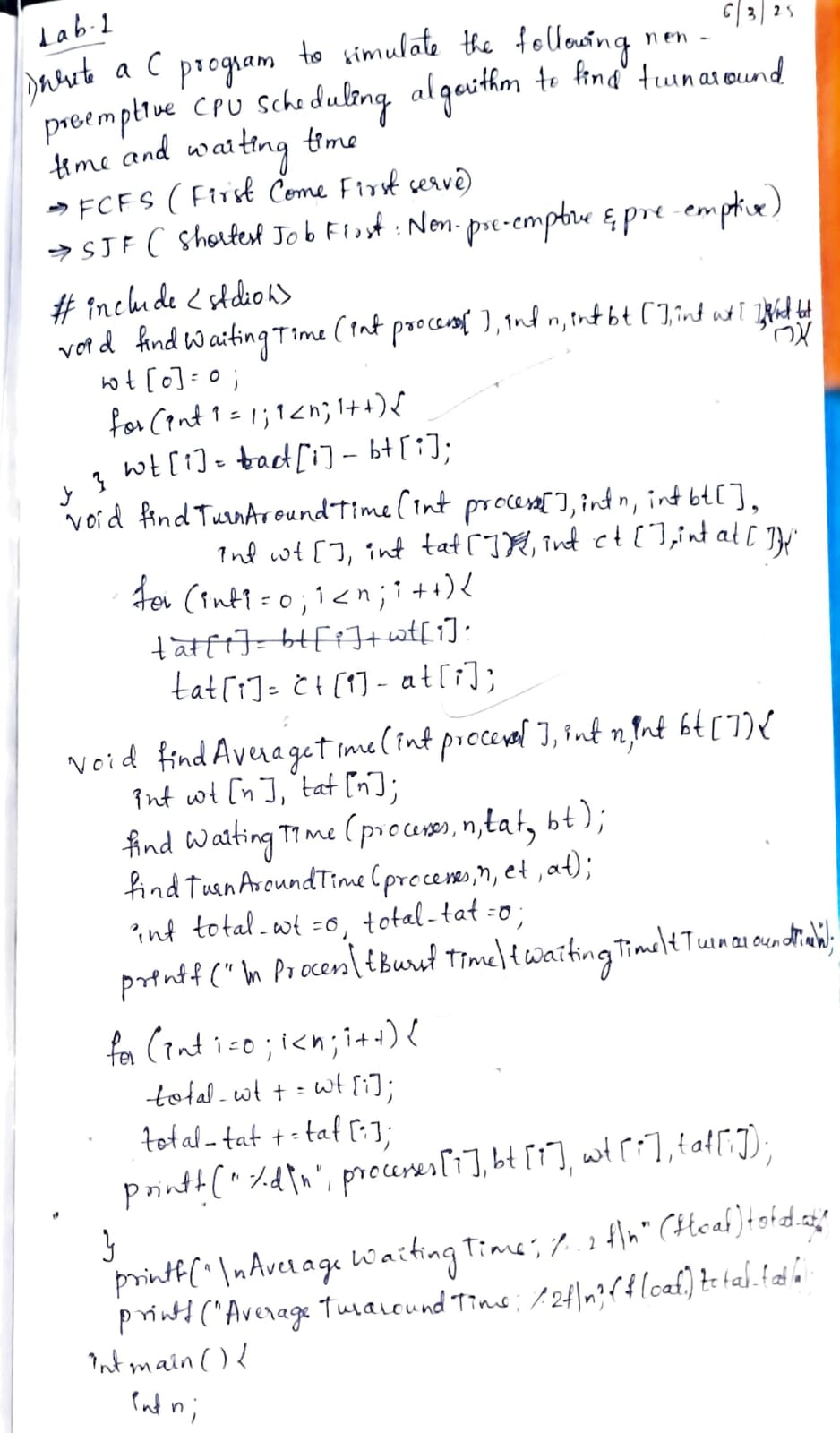
fcfs(proc, n);

display(proc, n);

return 0;

}





CODE:

SJF(non preemptive)

#include<stdio.h>

typedef struct{

int id, AT, BT, TAT, RT, CT, WT;

}Process;

void sorted(Process p[], int n)

{

int i, j;

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

{

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

{

if(p[j].AT > p[j+1].AT)

{

Process temp = p[j];

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

p[j+1] = temp;

}

}

}

}

void SJFNonPreemptive(Process p[], int n)

{

int TotalTAT = 0, TotalWT = 0, time = 0, completed = 0, i;

int remainingBT[n];

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

{

remainingBT[i] = p[i].BT;

}

while(completed < n)

{

int minBTIndex = -1;

int minBT = 9999;

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

{

if(p[i].AT <= time && remainingBT[i] > 0 && remainingBT[i] < minBT)

{

minBT = remainingBT[i];

minBTIndex = i;

}

}

if(minBTIndex != -1)

{

time += p[minBTIndex].BT;

p[minBTIndex].CT = time;

p[minBTIndex].TAT = p[minBTIndex].CT - p[minBTIndex].AT;

p[minBTIndex].WT = p[minBTIndex].TAT - p[minBTIndex].BT;

TotalTAT += p[minBTIndex].TAT;

TotalWT += p[minBTIndex].WT;

remainingBT[minBTIndex] = 0;

completed++;

}

else

{

time++;

}

}

float avgTAT = (float)TotalTAT / n;

float avgWT = (float)TotalWT / n;

printf("Average TurnAround Time: %f\n", avgTAT);

printf("Average Waiting Time: %f\n", avgWT);

}

int main()

{

int n, i;

printf("Enter the number of processes: ");

scanf("%d", &n);

Process p[n];

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

{

p[i].id = i+1;

printf("Enter arrival time and burst time for process %d: ", i+1);

scanf("%d %d", &p[i].AT, &p[i].BT);

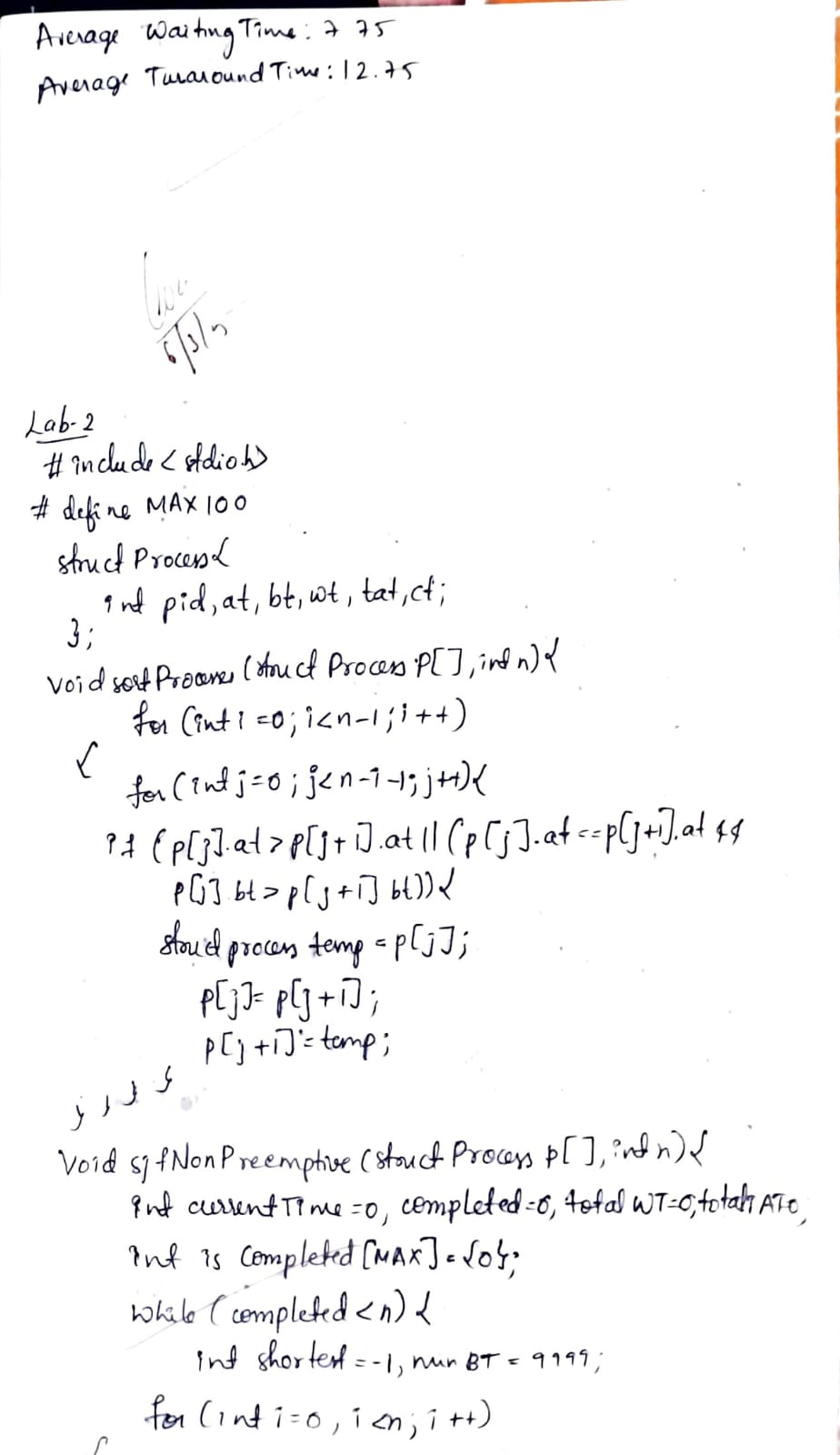
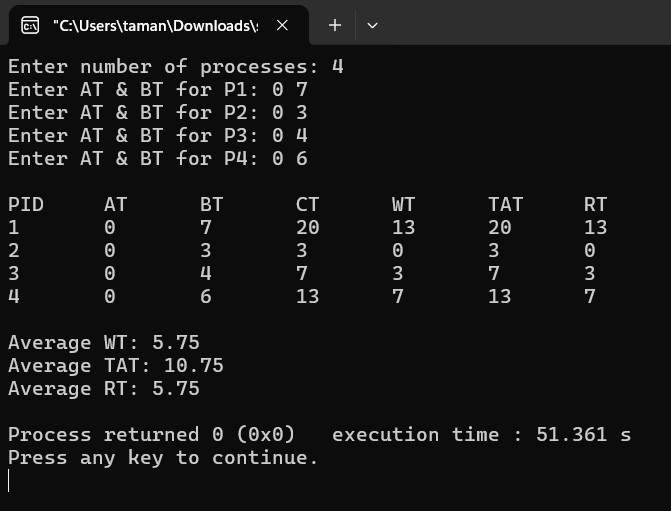
}

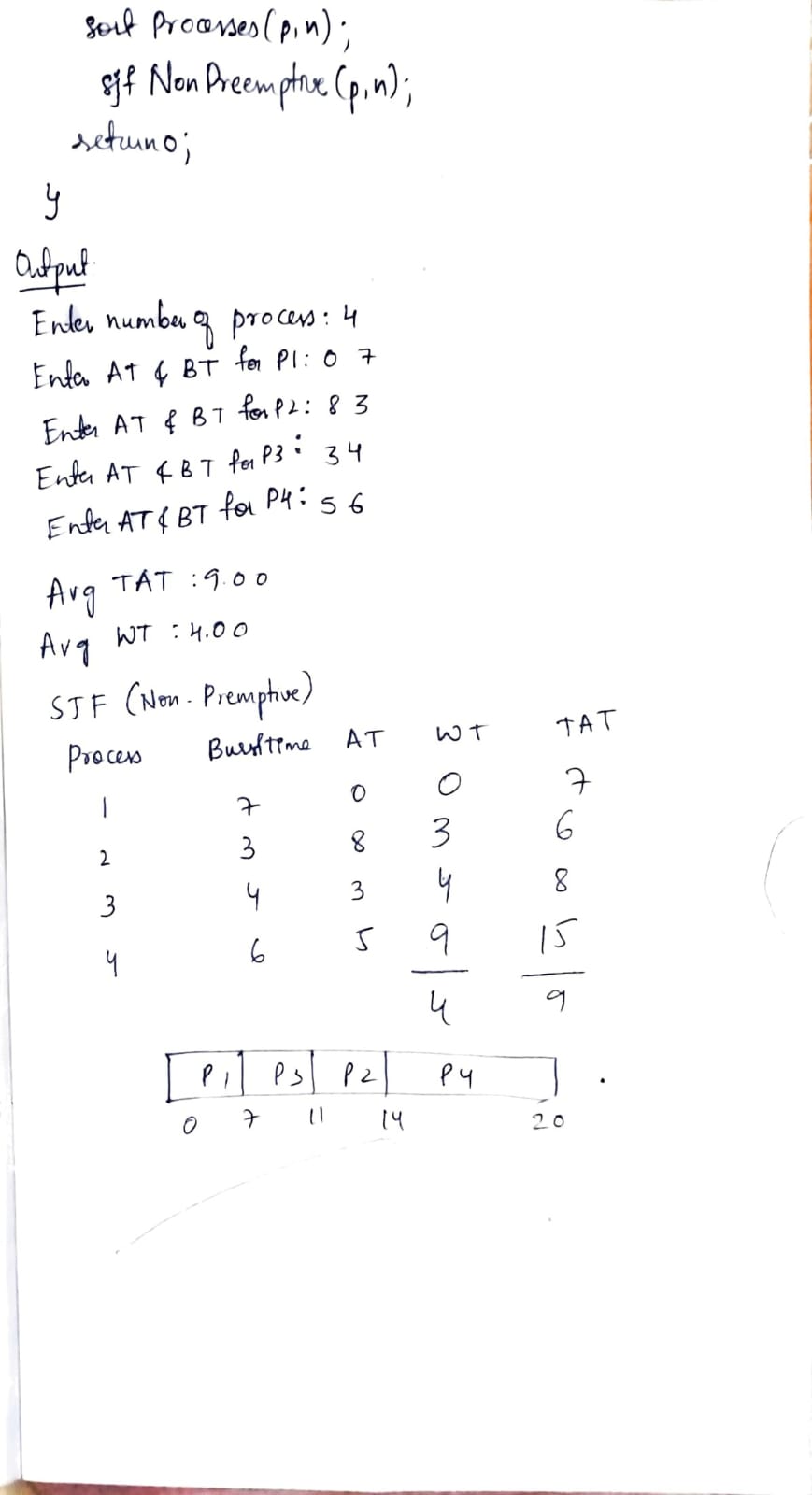
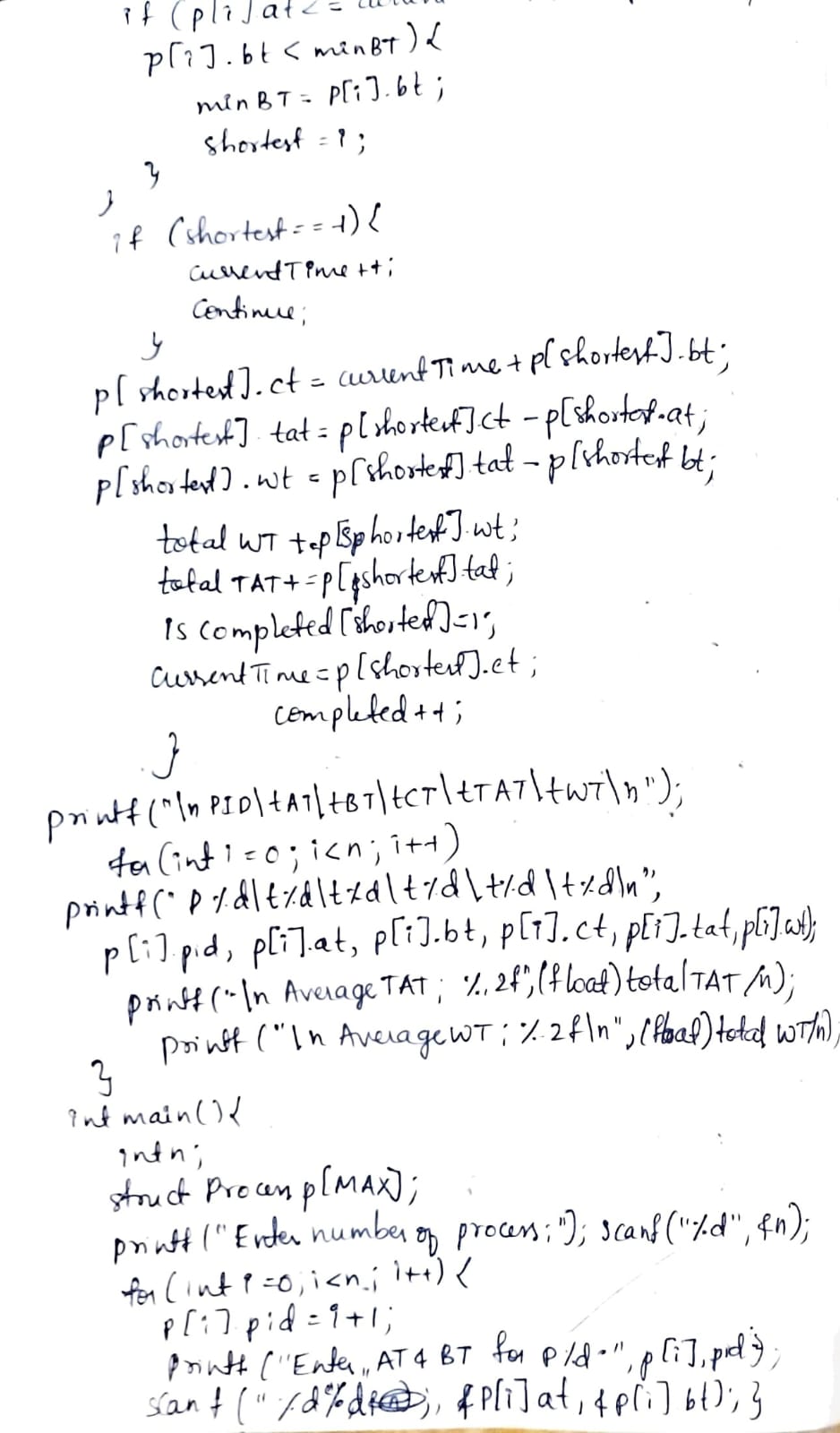
sorted(p, n);

SJFNonPreemptive(p, n);

return 0;

}

RESULT: 



CODE:

SRTF

#include<stdio.h>

typedef struct{

int id, AT, BT, TAT, RT, CT, WT, tempBT;

}Process;

void sorted(Process p[], int n)

{

int i, j;

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

{

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

{

if(p[j].AT > p[j+1].AT)

{

Process temp = p[j];

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

p[j+1] = temp;

}

}

}

}

void SJFPreemptive(Process p[], int n)

{

int TotalTAT = 0, TotalWT = 0, time = 0, completed = 0, i, minBTIndex;

int remainingBT[n];

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

{

remainingBT[i] = p[i].BT;

}

while(completed < n)

{

minBTIndex = -1;

int minBT = 9999;

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

{

if(p[i].AT <= time && remainingBT[i] > 0 && remainingBT[i] < minBT)

{

minBT = remainingBT[i];

minBTIndex = i;

}

}

if(minBTIndex != -1)

{

remainingBT[minBTIndex]--;

time++;

if(remainingBT[minBTIndex] == 0)

{

completed++;

p[minBTIndex].CT = time;

p[minBTIndex].TAT = p[minBTIndex].CT - p[minBTIndex].AT;

p[minBTIndex].WT = p[minBTIndex].TAT - p[minBTIndex].BT;

TotalTAT += p[minBTIndex].TAT;

TotalWT += p[minBTIndex].WT;

}

}

else

{

time++;

}

}

float avgTAT = (float)TotalTAT / n;

float avgWT = (float)TotalWT / n;

printf("Average TurnAround Time: %f\n", avgTAT);

printf("Average Waiting Time: %f\n", avgWT);

}

int main()

{

int n, i;

printf("Enter the number of processes: ");

scanf("%d", &n);

Process p[n];

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

{

p[i].id = i+1;

printf("Enter arrival time and burst time for process %d: ", i+1);

scanf("%d %d", &p[i].AT, &p[i].BT);

}

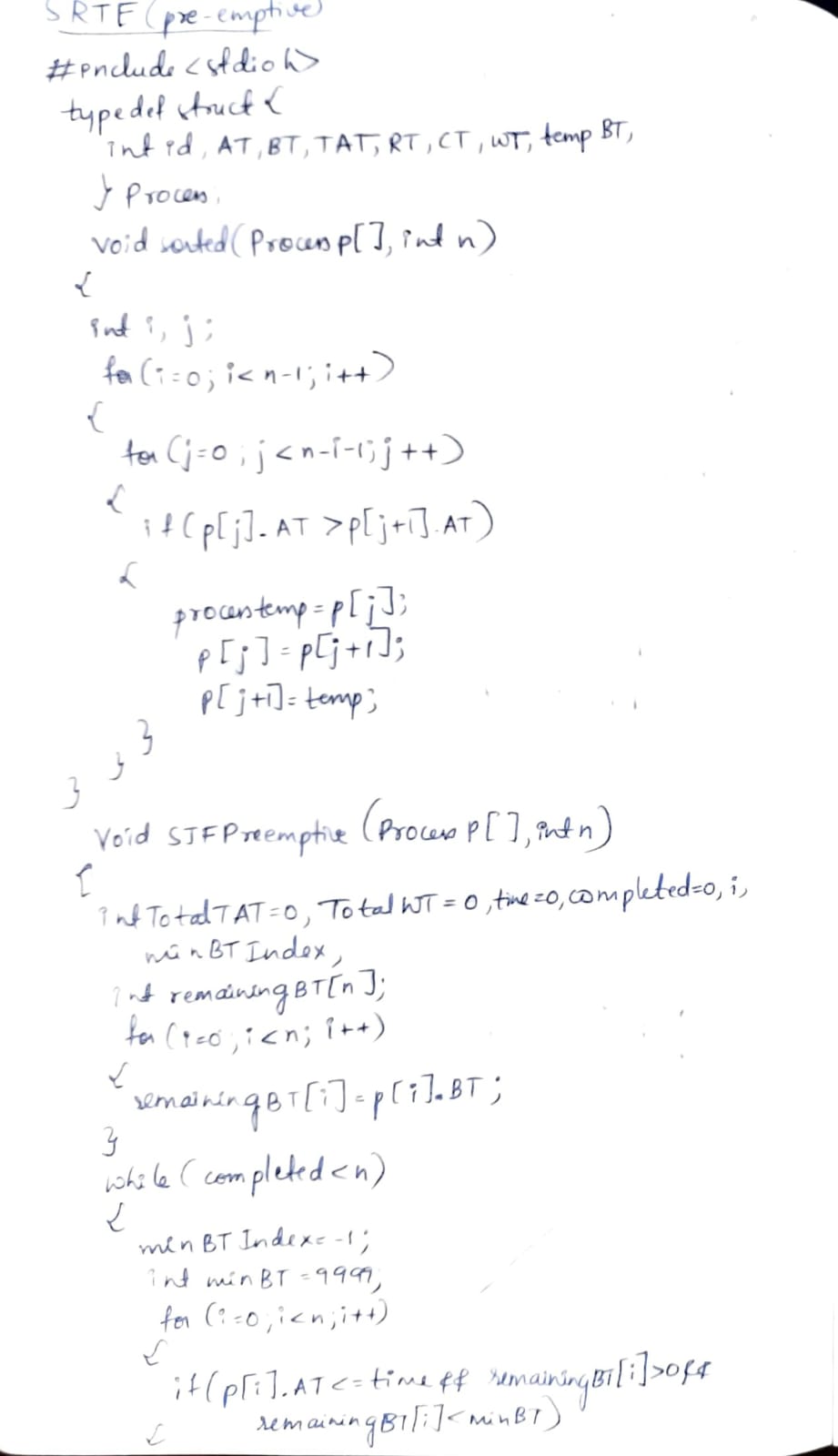
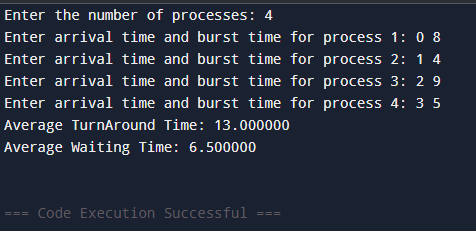
sorted(p, n);

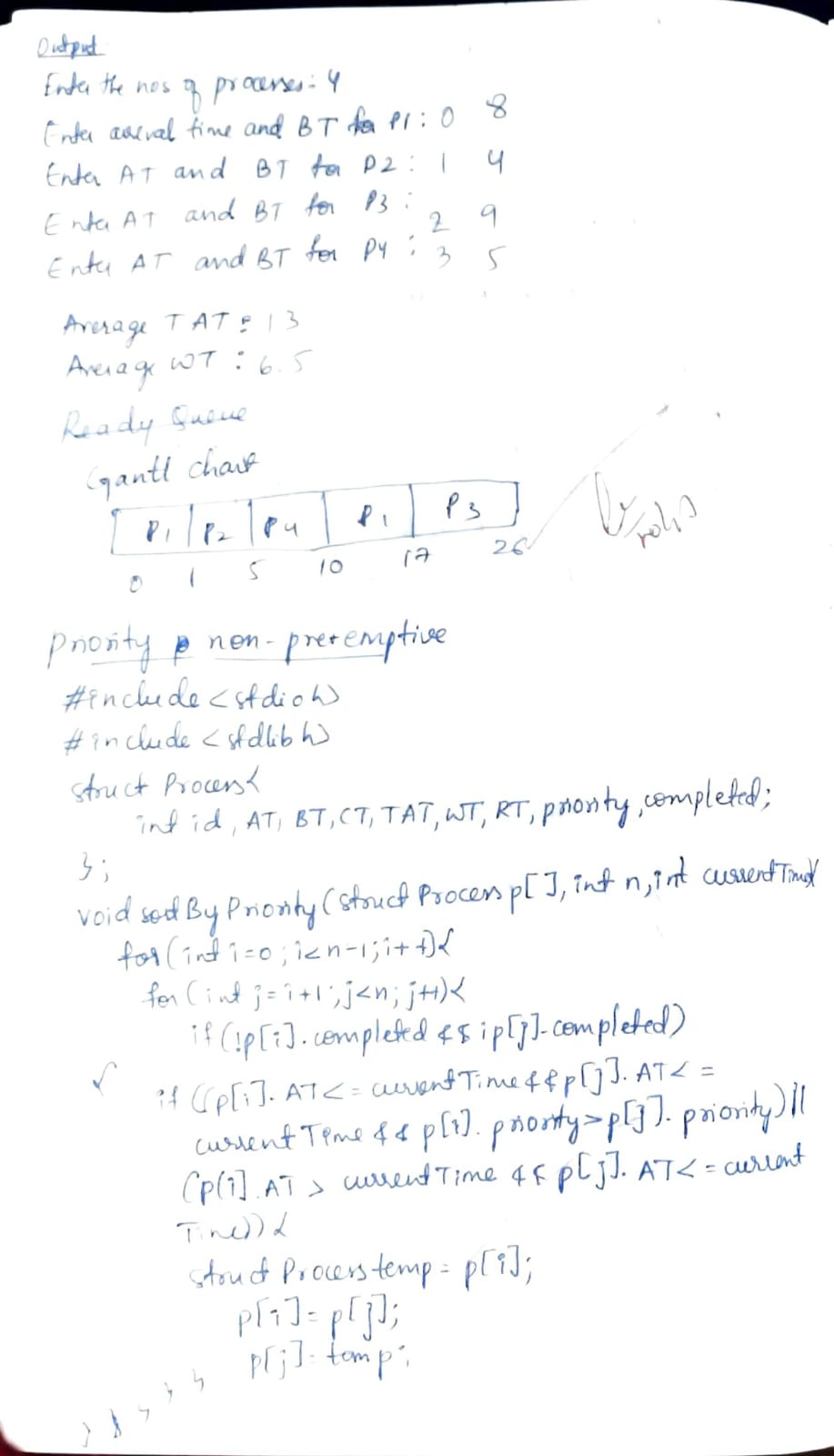
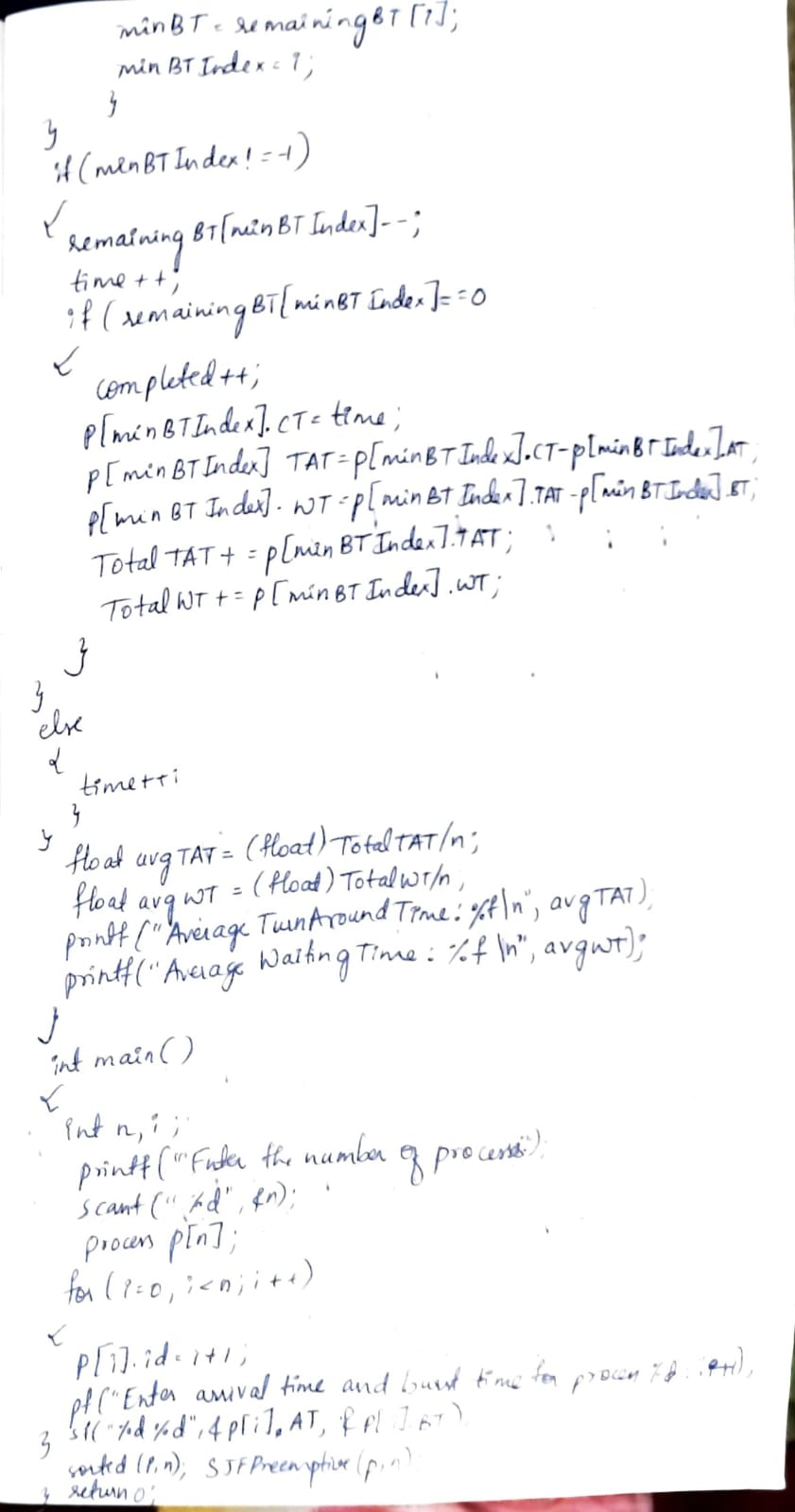
SJFPreemptive(p, n);

return 0;

}

RESULT:





LAB-2

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

→ Priority (pre-emptive & Non-pre-emptive)

CODE:

PRIORITY(pre emptive)

#include <stdio.h>

struct Process {

int pid, arrival\_time, burst\_time, priority, remaining\_time, completion\_time, waiting\_time, turnaround\_time;

};

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

int time = 0, completed = 0, min\_priority, shortest;

float total\_waiting = 0, total\_turnaround = 0;

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

p[i].remaining\_time = p[i].burst\_time;

while (completed < n) {

min\_priority = 9999;

shortest = -1;

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

if (p[i].arrival\_time <= time && p[i].remaining\_time > 0 && p[i].priority < min\_priority) {

min\_priority = p[i].priority;

shortest = i;

}

}

if (shortest == -1) {

time++;

continue;

}

p[shortest].remaining\_time--;

if (p[shortest].remaining\_time == 0) {

completed++;

p[shortest].completion\_time = time + 1;

p[shortest].turnaround\_time = p[shortest].completion\_time - p[shortest].arrival\_time;

p[shortest].waiting\_time = p[shortest].turnaround\_time - p[shortest].burst\_time;

total\_waiting += p[shortest].waiting\_time;

total\_turnaround += p[shortest].turnaround\_time;

}

time++;

}

printf("\nPreemptive Priority Scheduling:\n");

printf("PID\tAT\tBT\tP\tCT\tTAT\tWT\n");

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

printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].arrival\_time, p[i].burst\_time, p[i].priority, p[i].completion\_time, p[i].turnaround\_time, p[i].waiting\_time);

printf("Average Waiting Time: %.2f\n", total\_waiting / n);

printf("Average Turnaround Time: %.2f\n", total\_turnaround / 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 Process ID, Arrival Time, Burst Time, Priority: ");

scanf("%d %d %d %d", &p[i].pid, &p[i].arrival\_time, &p[i].burst\_time, &p[i].priority);

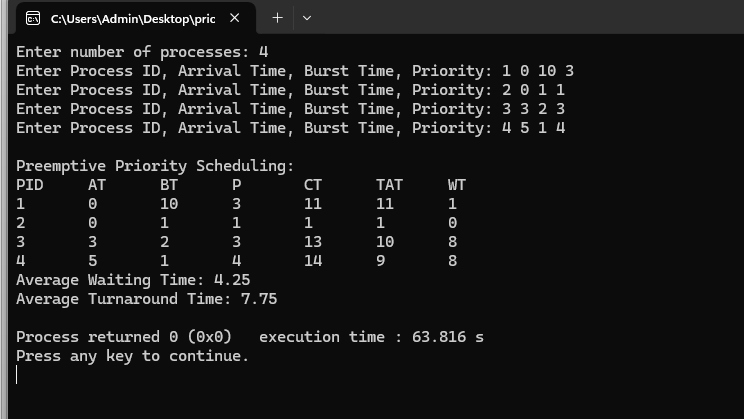
}

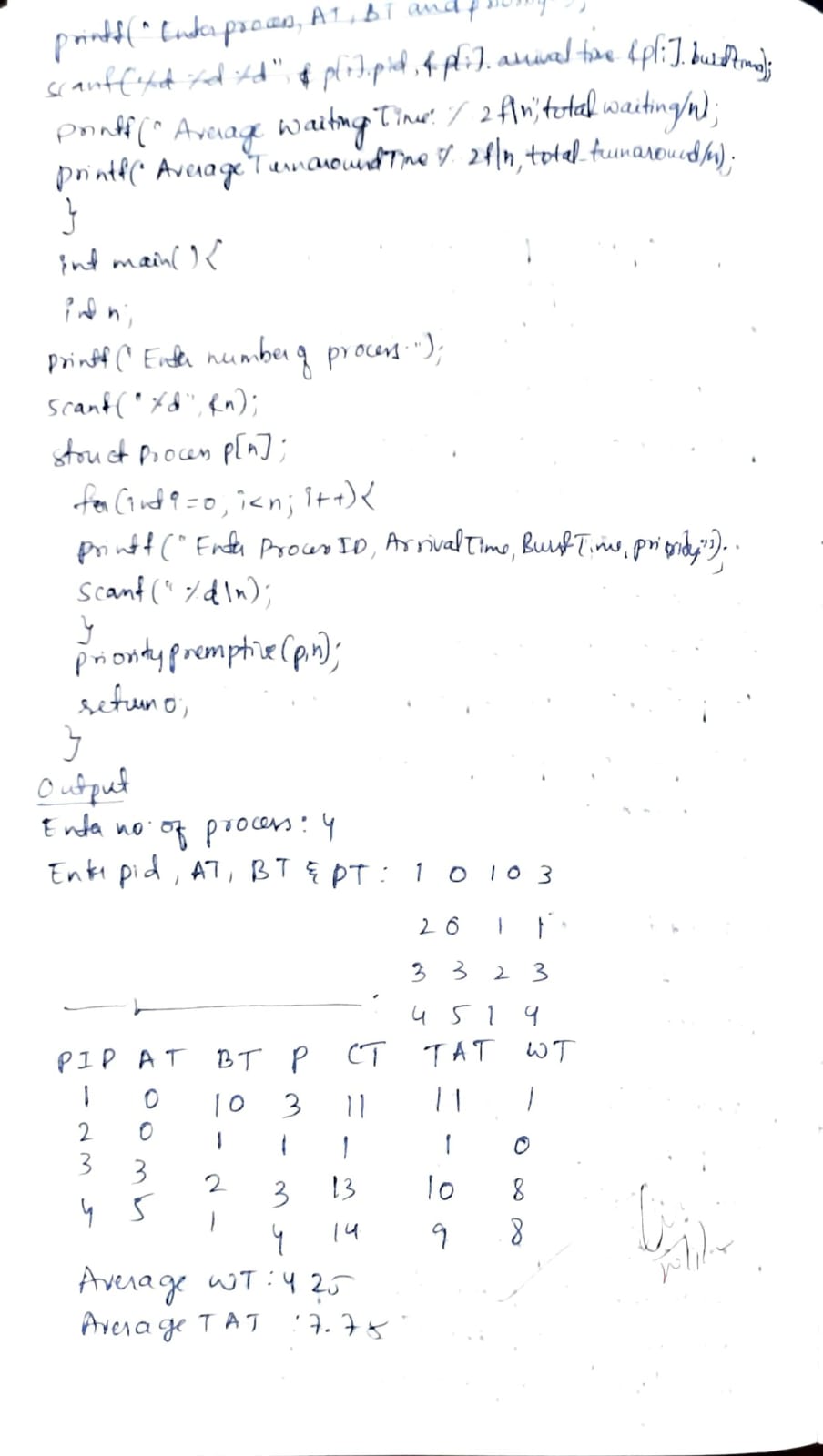
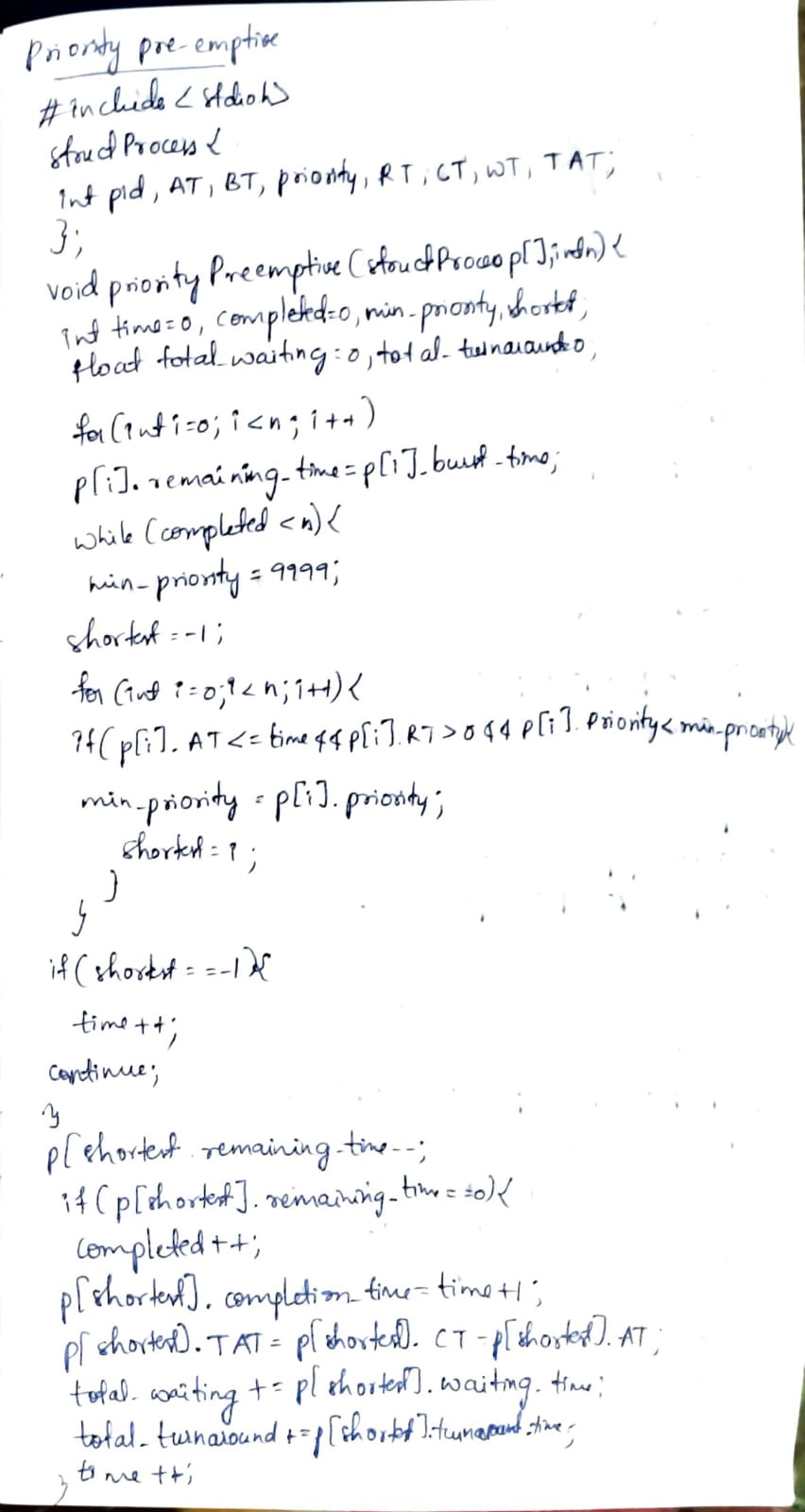
priorityPreemptive(p, n);

return 0;

}

RESULT:





CODE:

PRIORITY(non pre emptive)

#include <stdio.h>

#include <stdlib.h>

struct Process {

int id, AT, BT, CT, TAT, WT, RT, priority, completed;

};

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

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

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

if (!p[i].completed && !p[j].completed) {

if ((p[i].AT <= currentTime && p[j].AT <= currentTime && p[i].priority > p[j].priority) ||

(p[i].AT > currentTime && p[j].AT <= currentTime)) {

struct Process temp = p[i];

p[i] = p[j];

p[j] = temp;

}

}

}

}

}

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

int completed = 0, currentTime = 0;

float totalWT = 0, totalTAT = 0;

while (completed < n) {

sortByPriority(p, n, currentTime);

int index = -1;

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

if (!p[i].completed && p[i].AT <= currentTime) {

index = i;

break;

}

}

if (index == -1) {

currentTime++;

} else {

p[index].CT = currentTime + p[index].BT;

p[index].TAT = p[index].CT - p[index].AT;

p[index].WT = p[index].TAT - p[index].BT;

p[index].RT = currentTime - p[index].AT;

p[index].completed = 1;

totalWT += p[index].WT;

totalTAT += p[index].TAT;

currentTime = p[index].CT;

completed++;

}

}

printf("\nProcess\tAT\tBT\tPT\tCT\tTAT\tWT\tRT\n");

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

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

if (p[j].id == i + 1) {

printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",

p[j].id, p[j].AT, p[j].BT, p[j].priority, p[j].CT, p[j].TAT, p[j].WT, p[j].RT);

break;

}

}

}

printf("\nAverage WT: %.2f", totalWT / n);

printf("\nAverage TAT: %.2f\n", totalTAT / n);

}

int main() {

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

struct Process p[n];

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

p[i].id = i + 1;

printf("Enter Arrival Time (AT), Burst Time (BT) & Priority for process %d: ", i + 1);

scanf("%d %d %d", &p[i].AT, &p[i].BT, &p[i].priority);

p[i].completed = 0;

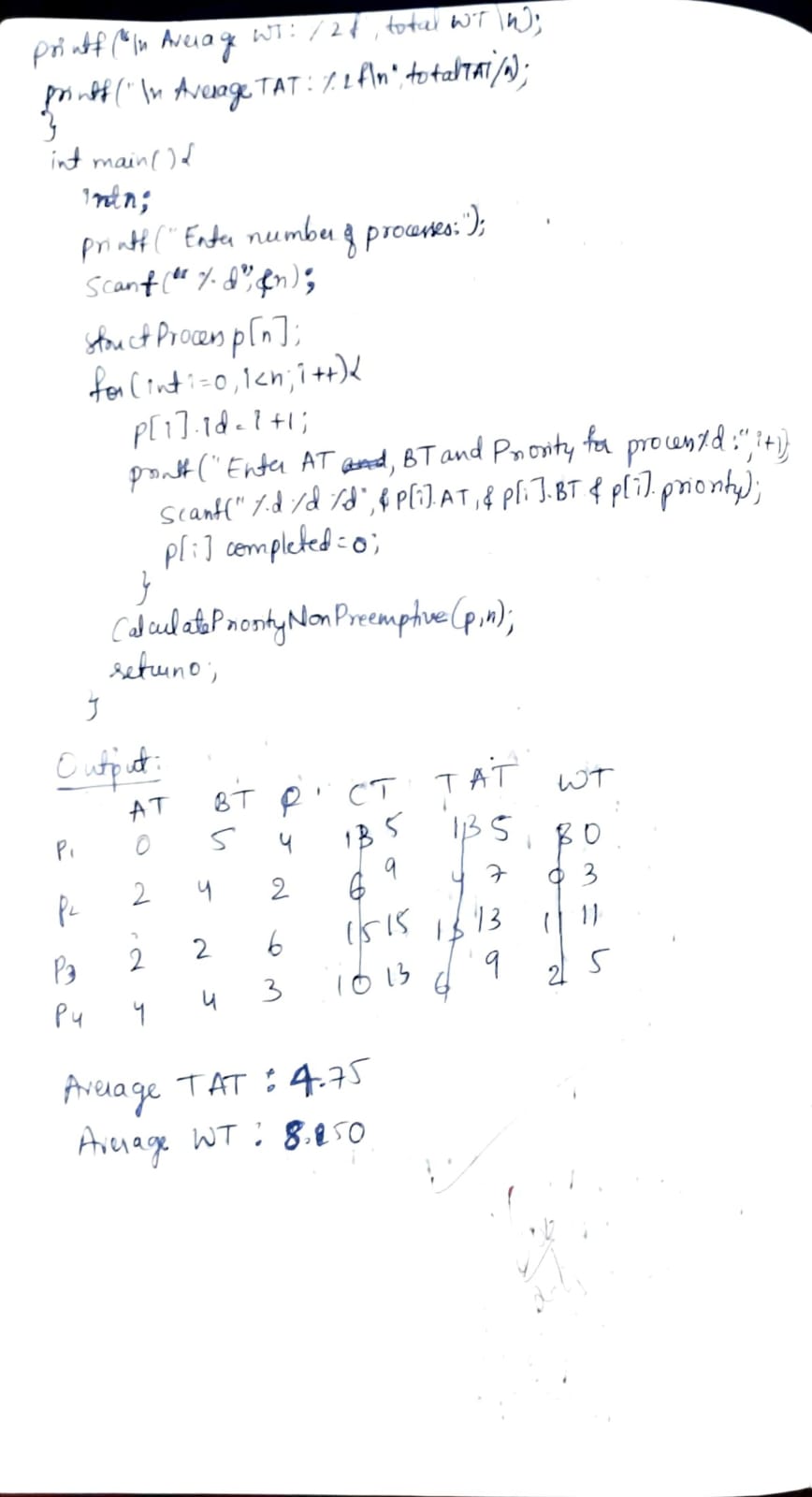
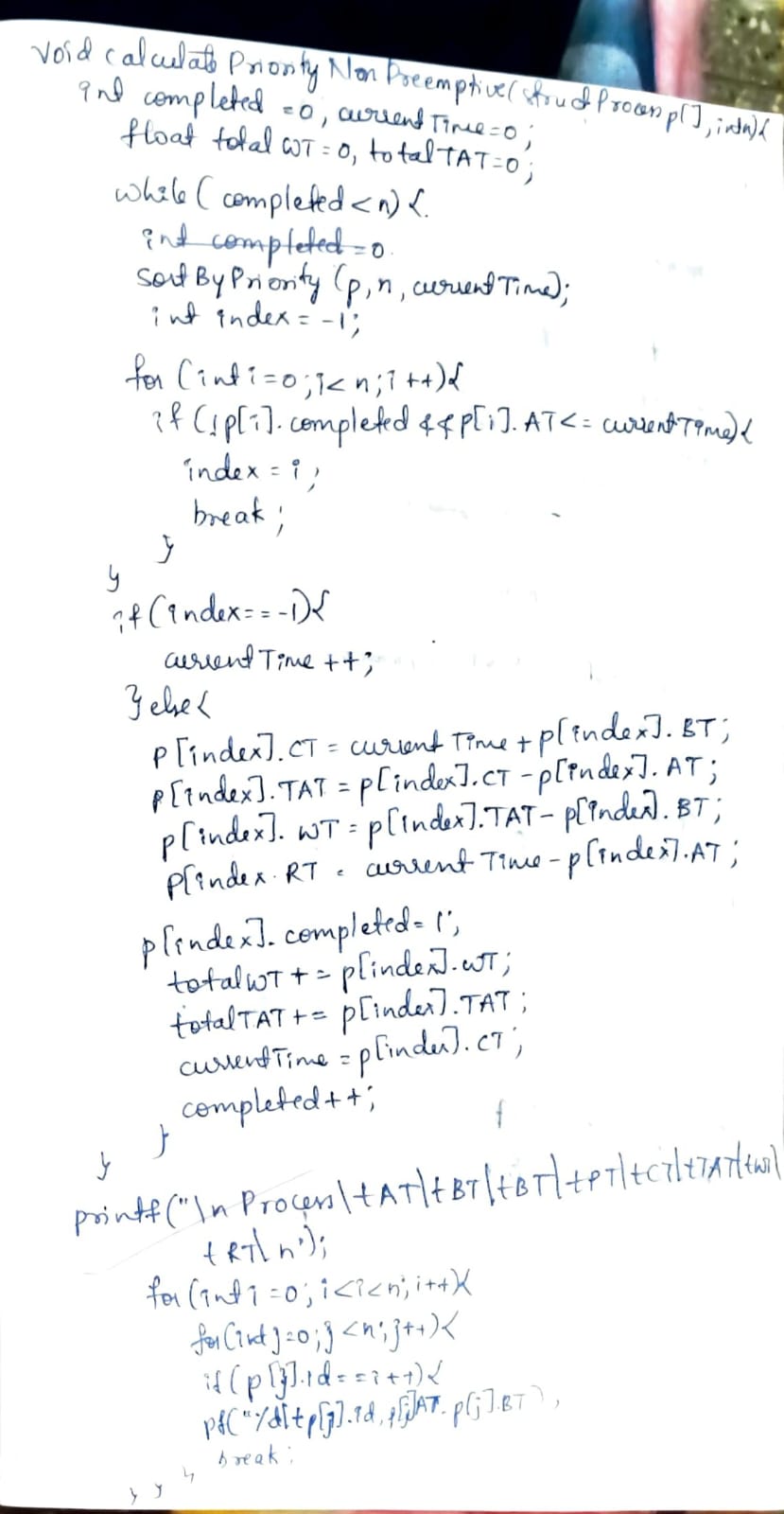
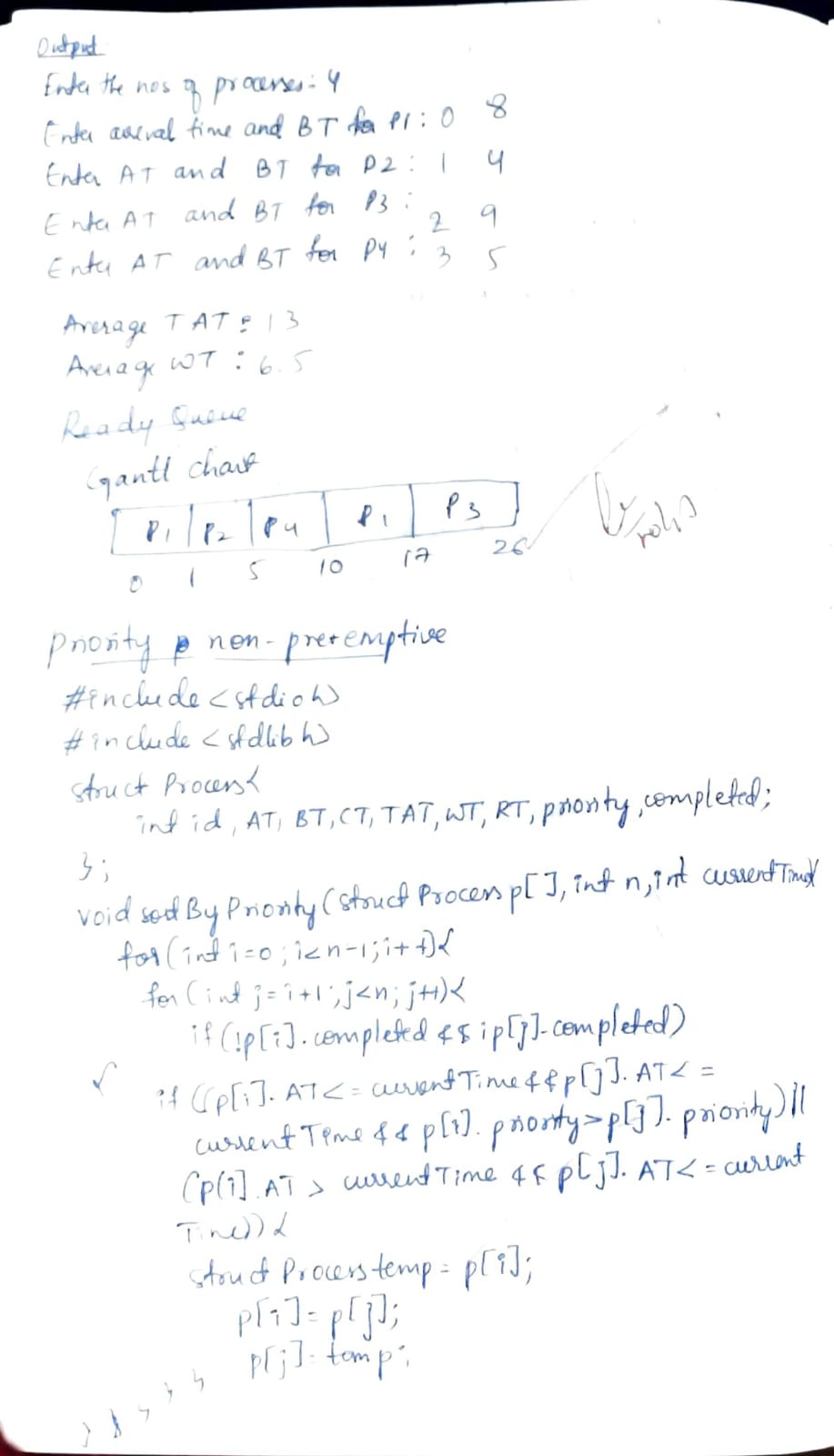
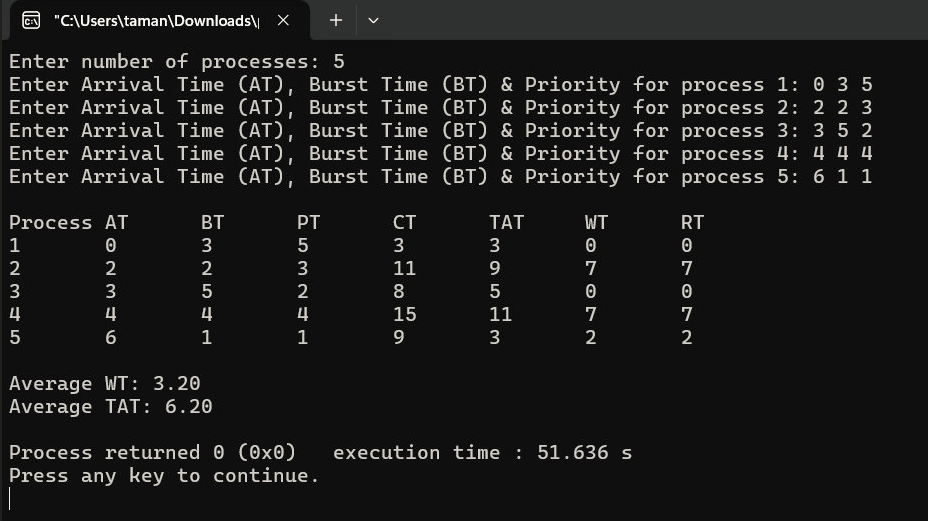
}

calculatePriorityNonPreemptive(p, n);

return 0;

}

RESULT:



LAB-3

Write a C program to simulate the following scheduling

1. Multilevel queue
2. Rate monotonic
3. Earliest deadline first

CODE:

MULTILEVEL QUEUE SCHEDULING:

#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);

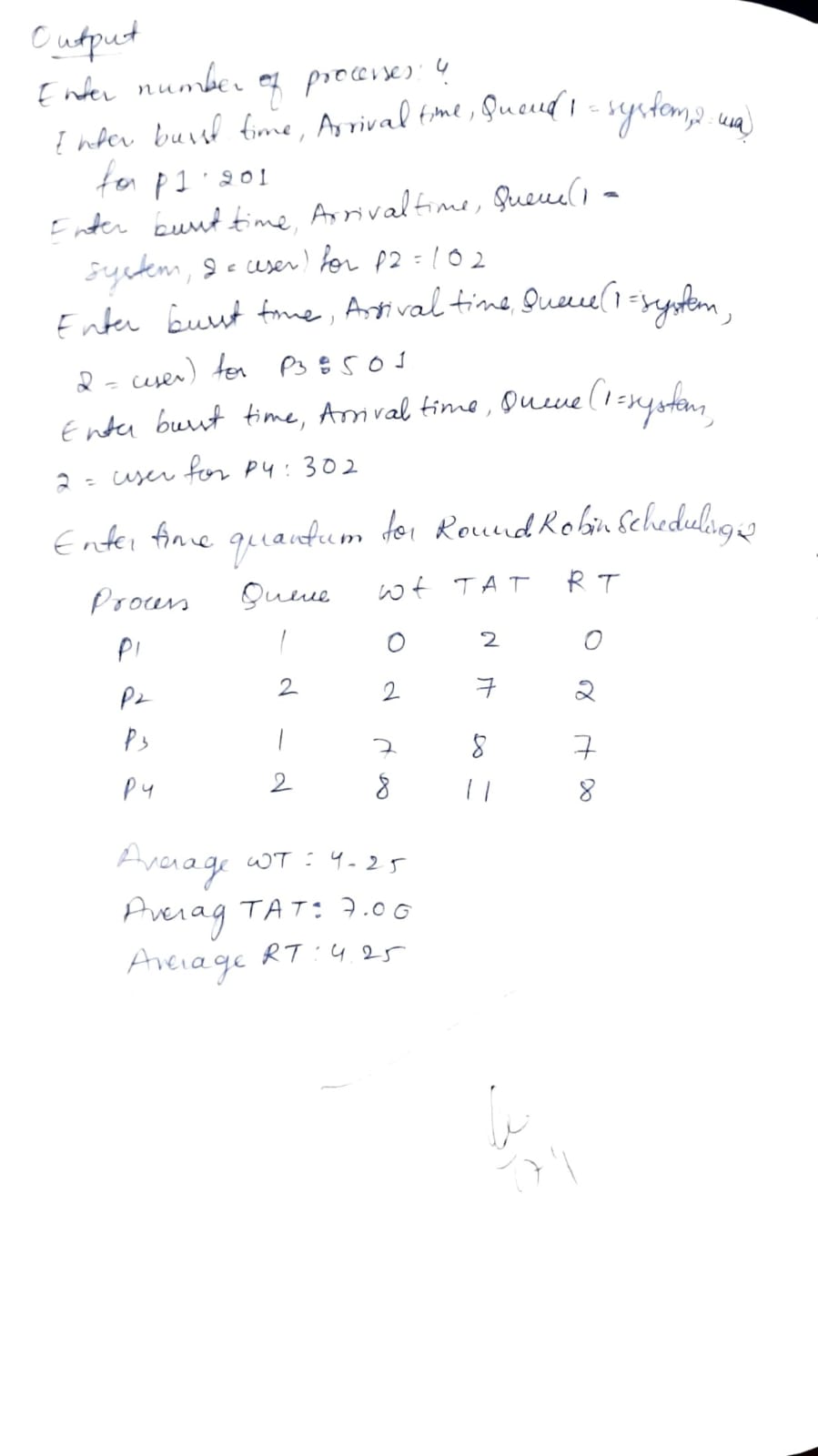
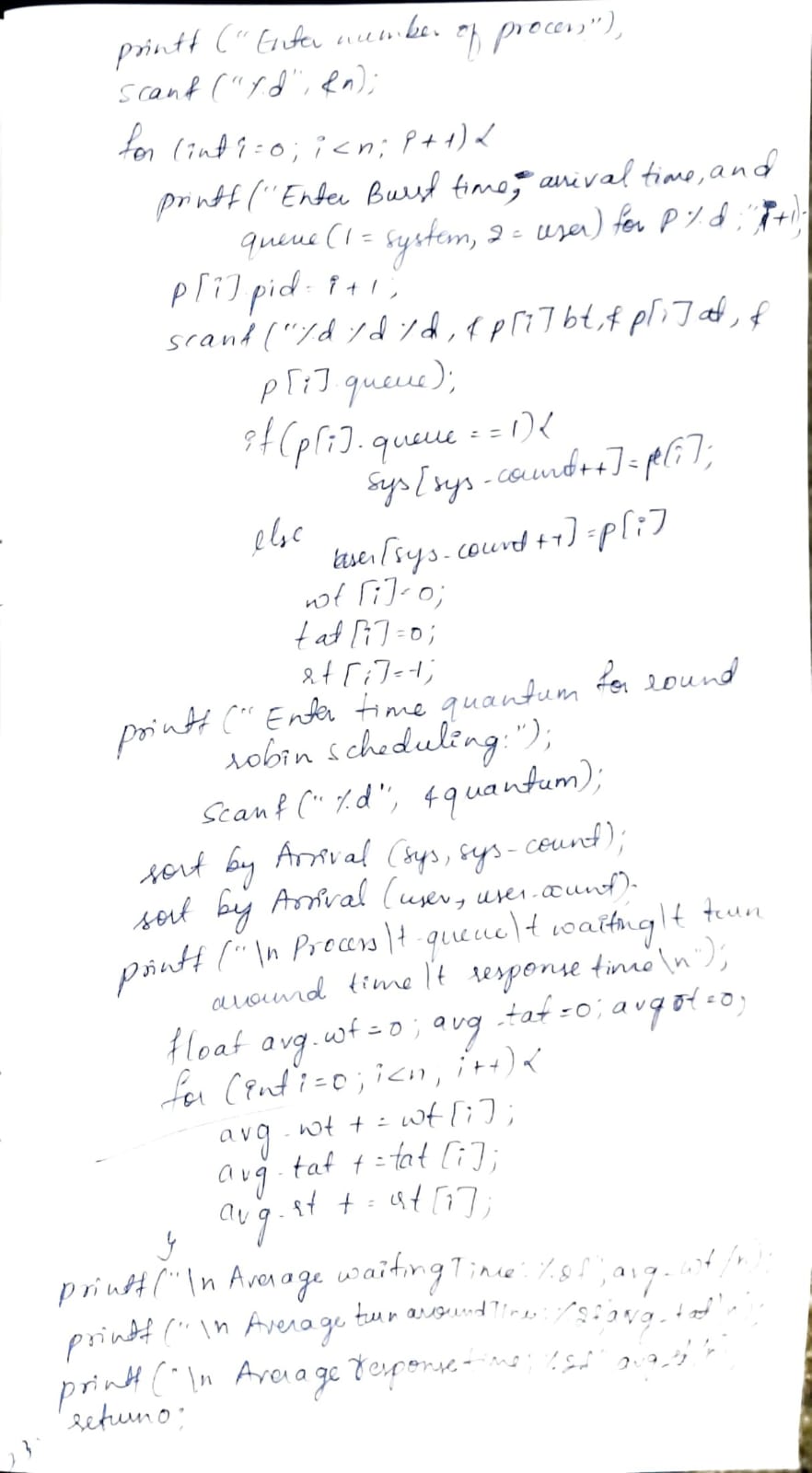
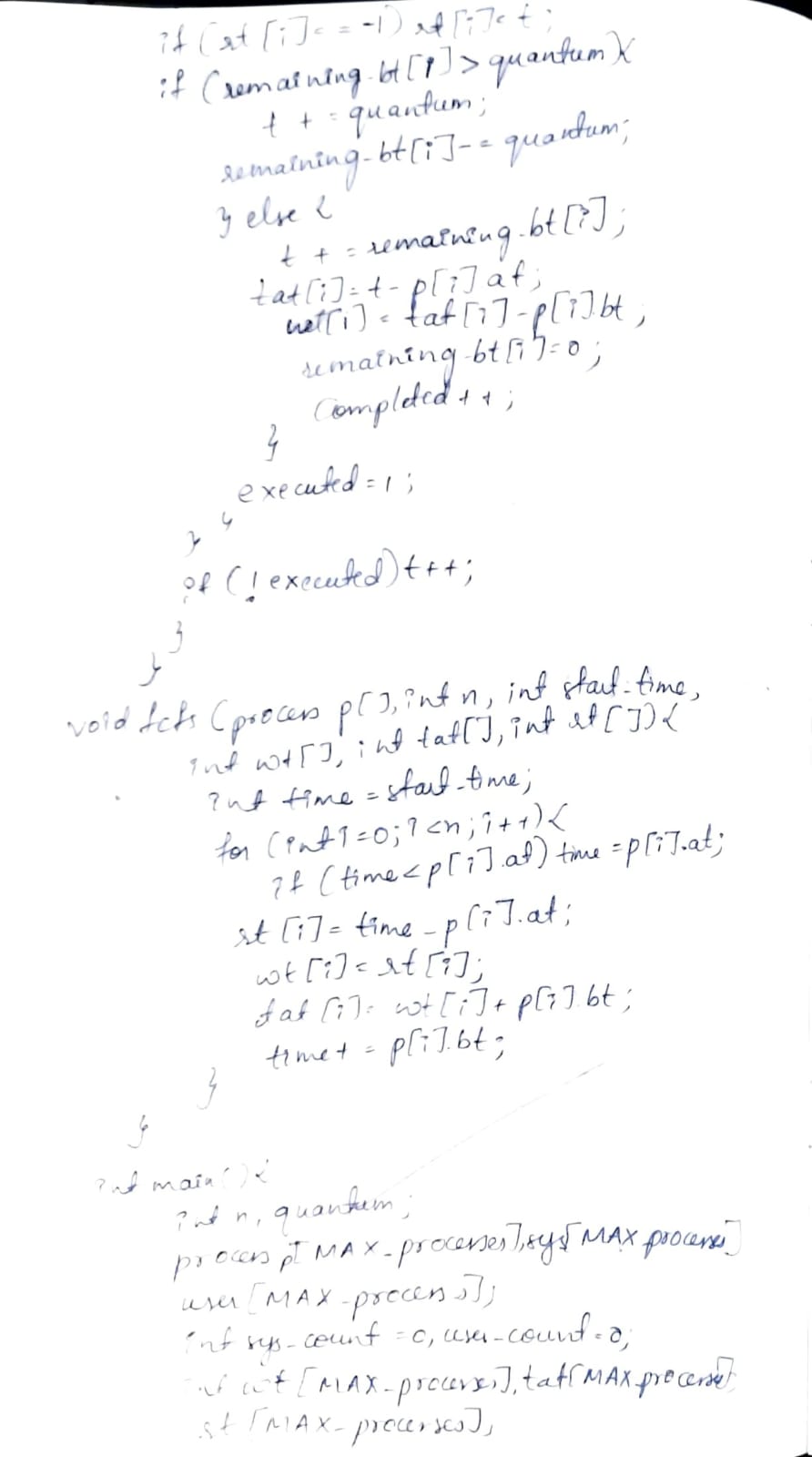
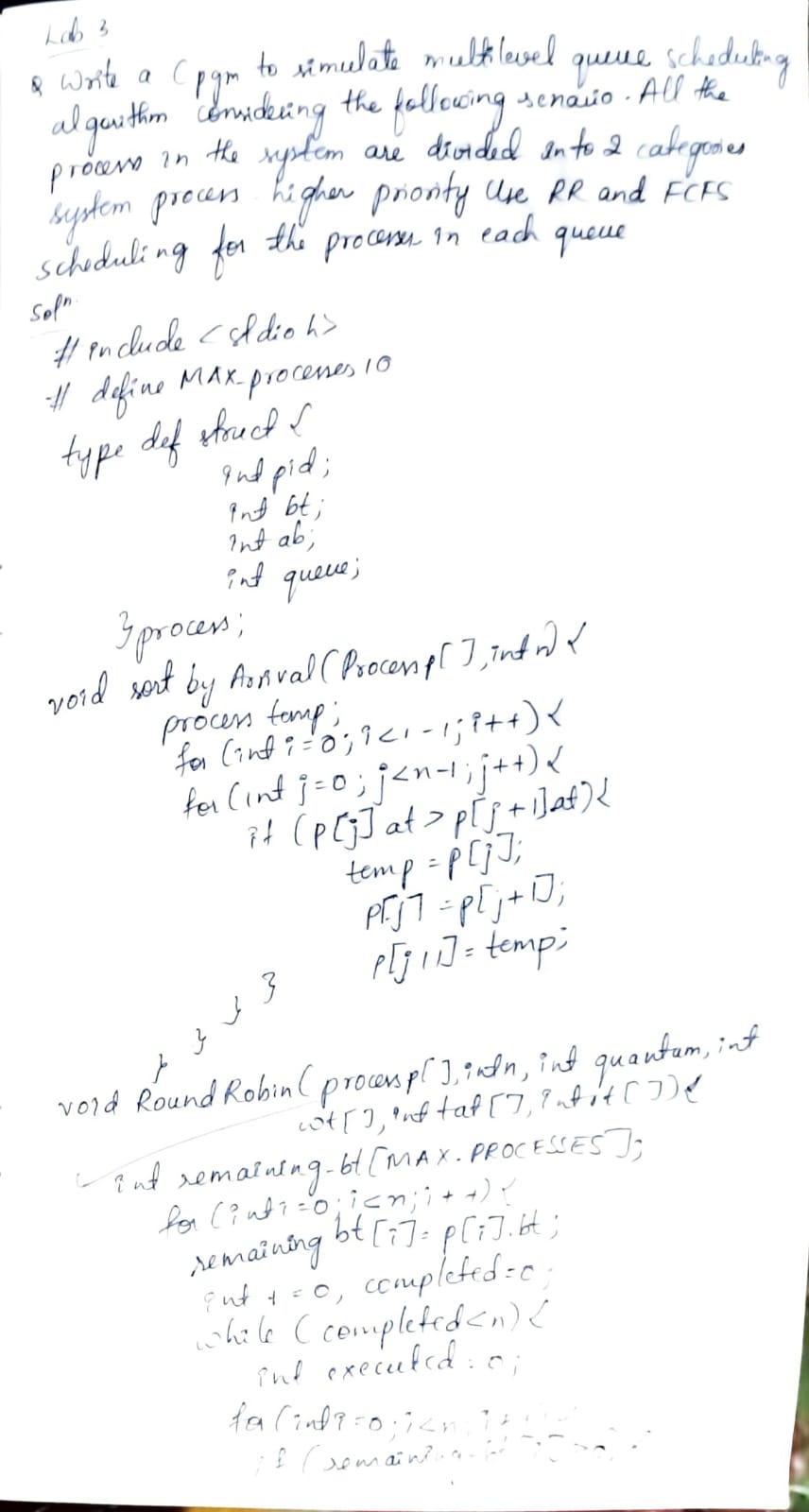
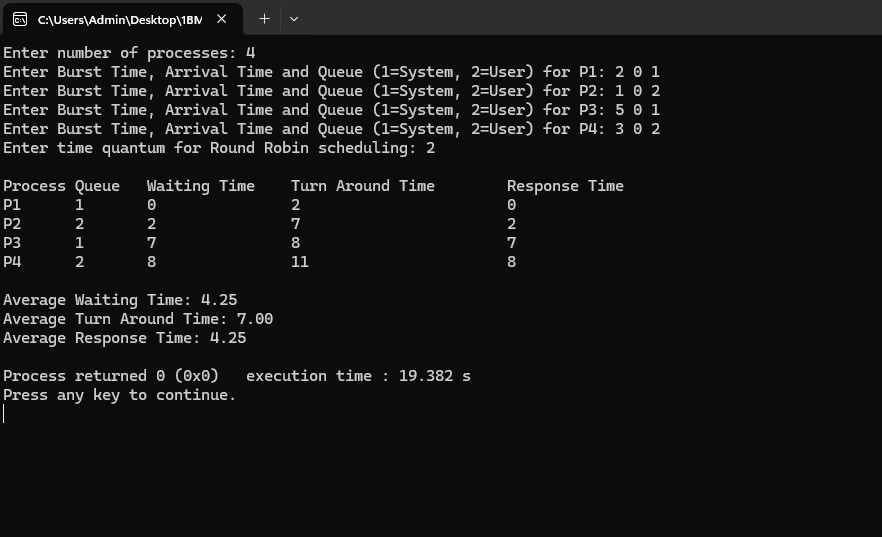
printf("\nAverage Turn Around Time: %.2f", avg\_tat / n);

printf("\nAverage Response Time: %.2f\n", avg\_rt / n);

return 0;

}

RESULT:



CODE:

RATE-MONOTONIC

#include <stdio.h>

#include <math.h>

#include <time.h>

// GCD and LCM helper functions

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 main() {

clock\_t start = clock();

int n = 3;

int burst[3], period[3];

float utilization = 0.0;

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

printf("Enter the CPU burst times:\n");

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

scanf("%d", &burst[i]);

}

printf("Enter the time periods:\n");

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

scanf("%d", &period[i]);

}

// LCM of all periods

int lcm\_val = period[0];

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

lcm\_val = lcm(lcm\_val, period[i]);

}

printf("LCM=%d\n", lcm\_val);

printf("Rate\n");

printf("Monotone Scheduling:\n");

// Matrix-style header

printf("PID\tBurst\tPeriod\n");

// Table values + utilization

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

printf("%d\t%d\t%d\n", i + 1, burst[i], period[i]);

utilization += (float)burst[i] / period[i];

}

float bound = n \* (pow(2, 1.0 / n) - 1);

printf("%.6f <= %.6f =>%s\n", utilization, bound, utilization <= bound ? "true" : "false");

printf("Process returned 0 (0x0)\n");

printf("Press any key to continue.\n");

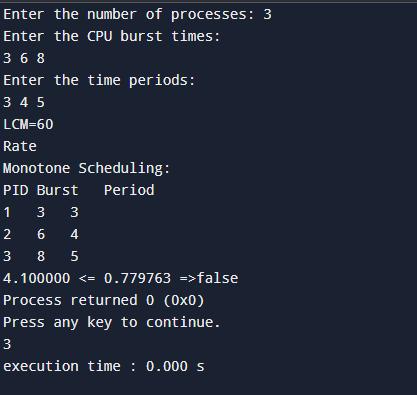
printf("%d\n", n);

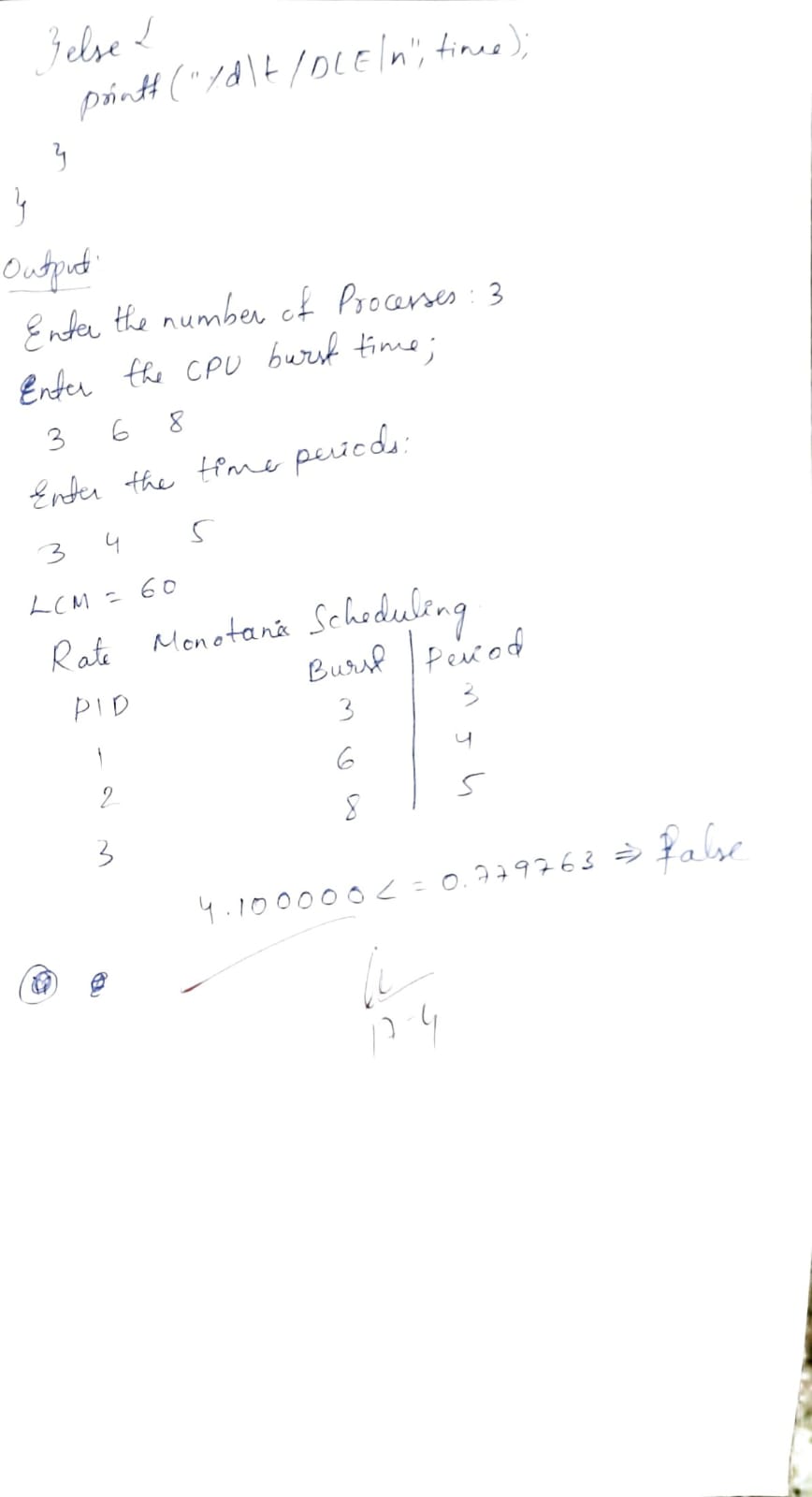
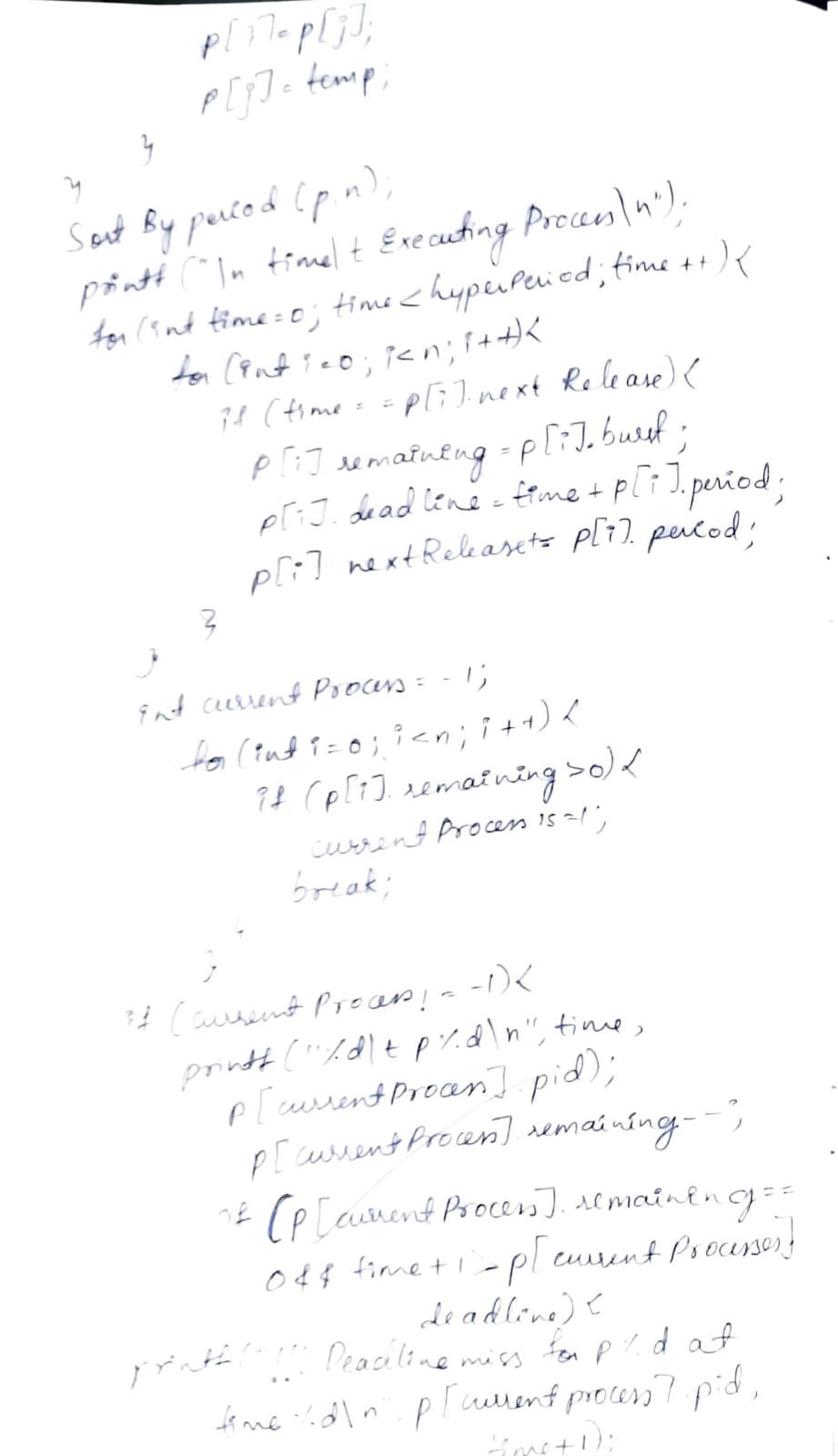
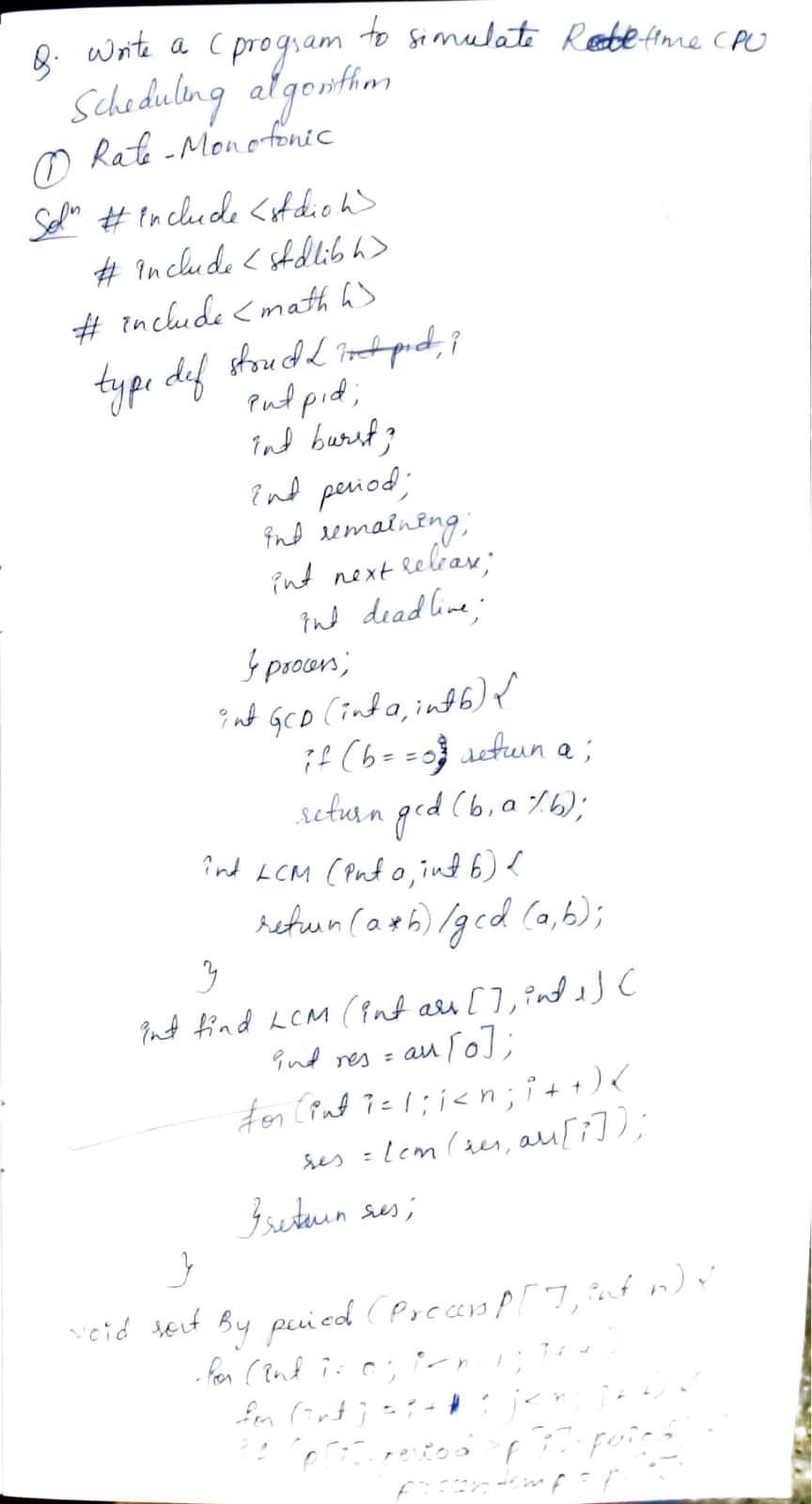
double exec\_time = (double)(clock() - start) / CLOCKS\_PER\_SEC;

printf("execution time : %.3f s\n", exec\_time);

return 0;

}

RESULT: 



C)EARLIST DEADLINE FIRST

#include <stdio.h>

#define MAX 10

typedef struct {

int pid;

int burst;

int period;

int deadline;

int remaining;

int nextRelease;

int absoluteDeadline;

} Task;

void sortByDeadline(Task tasks[], int n) {

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

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

if (tasks[i].absoluteDeadline > tasks[j].absoluteDeadline) {

Task temp = tasks[i];

tasks[i] = tasks[j];

tasks[j] = temp;

}

}

int lcm(int a, int b) {

int temp\_a = a, temp\_b = b;

while (temp\_b != 0) {

int temp = temp\_b;

temp\_b = temp\_a % temp\_b;

temp\_a = temp;

}

return (a \* b) / temp\_a;

}

int findLCM(int arr[], int n) {

int result = arr[0];

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

result = lcm(result, arr[i]);

}

return result;

}

int main() {

int n;

printf("Enter the number of processes: ");

scanf("%d", &n);

Task tasks[MAX];

int periods[MAX];

printf("Enter the CPU burst times:\n");

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

scanf("%d", &tasks[i].burst);

tasks[i].pid = i + 1;

tasks[i].remaining = 0;

}

printf("Enter the deadlines:\n");

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

scanf("%d", &tasks[i].deadline);

}

printf("Enter the time periods:\n");

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

scanf("%d", &tasks[i].period);

tasks[i].nextRelease = 0;

tasks[i].absoluteDeadline = tasks[i].deadline;

periods[i] = tasks[i].period;

}

int hyperPeriod = findLCM(periods, n);

printf("\nEarliest Deadline First Scheduling:\n");

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

printf("\nTime\tStatus\n");

for (int time = 0; time < hyperPeriod; time++) {

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

if (time == tasks[i].nextRelease) {

tasks[i].remaining = tasks[i].burst;

tasks[i].absoluteDeadline = time + tasks[i].deadline;

tasks[i].nextRelease += tasks[i].period;

}

}

sortByDeadline(tasks, n);

int executed = 0;

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

if (tasks[i].remaining > 0) {

printf("%dms:\tTask %d is running.\n", time, tasks[i].pid);

tasks[i].remaining--;

executed = 1;

if (time + 1 > tasks[i].absoluteDeadline && tasks[i].remaining > 0) {

printf("!!! Deadline MISS for Task %d at time %d\n", tasks[i].pid, time + 1);

}

break;

}

}

if (!executed) {

printf("%dms:\tIDLE\n", time);

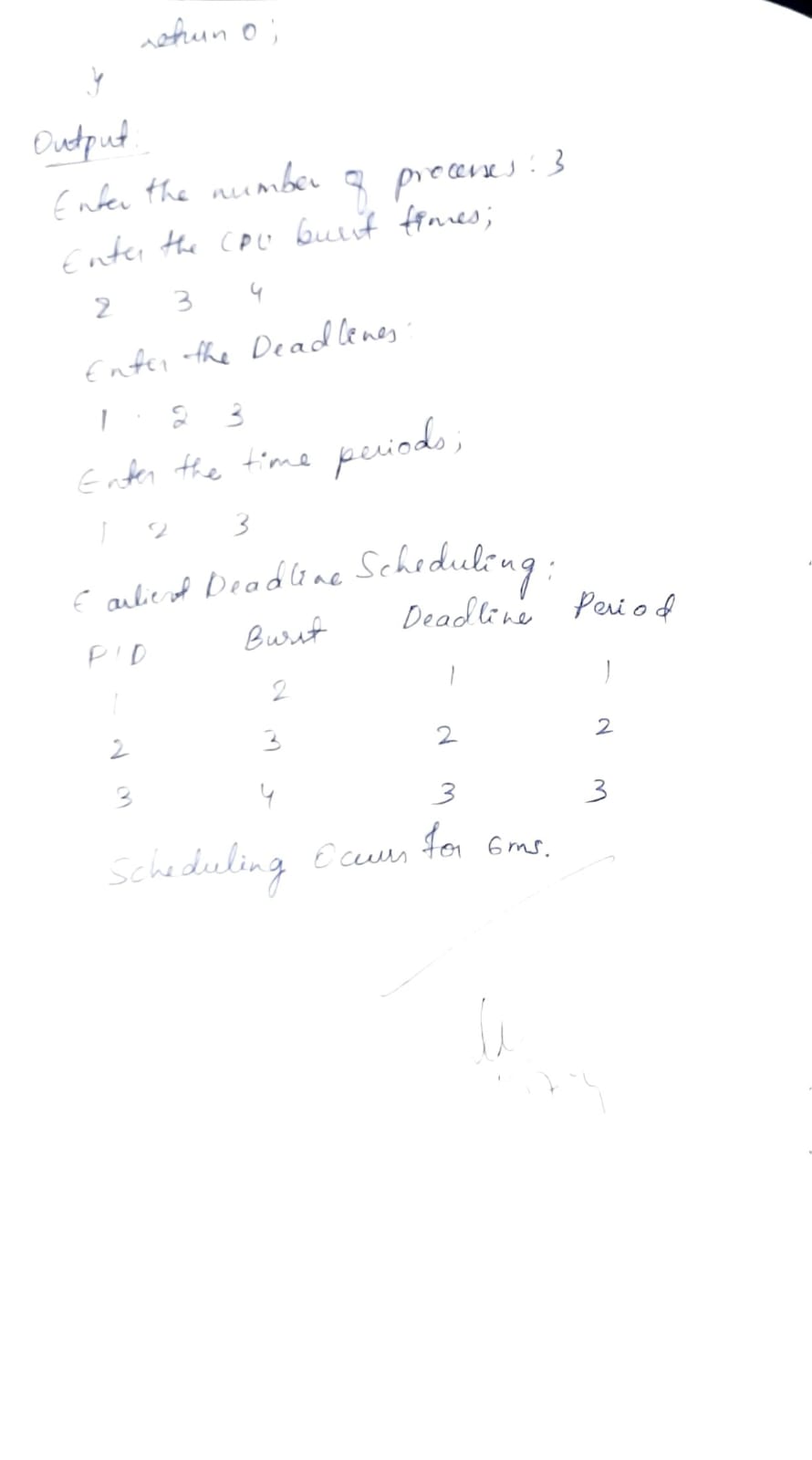
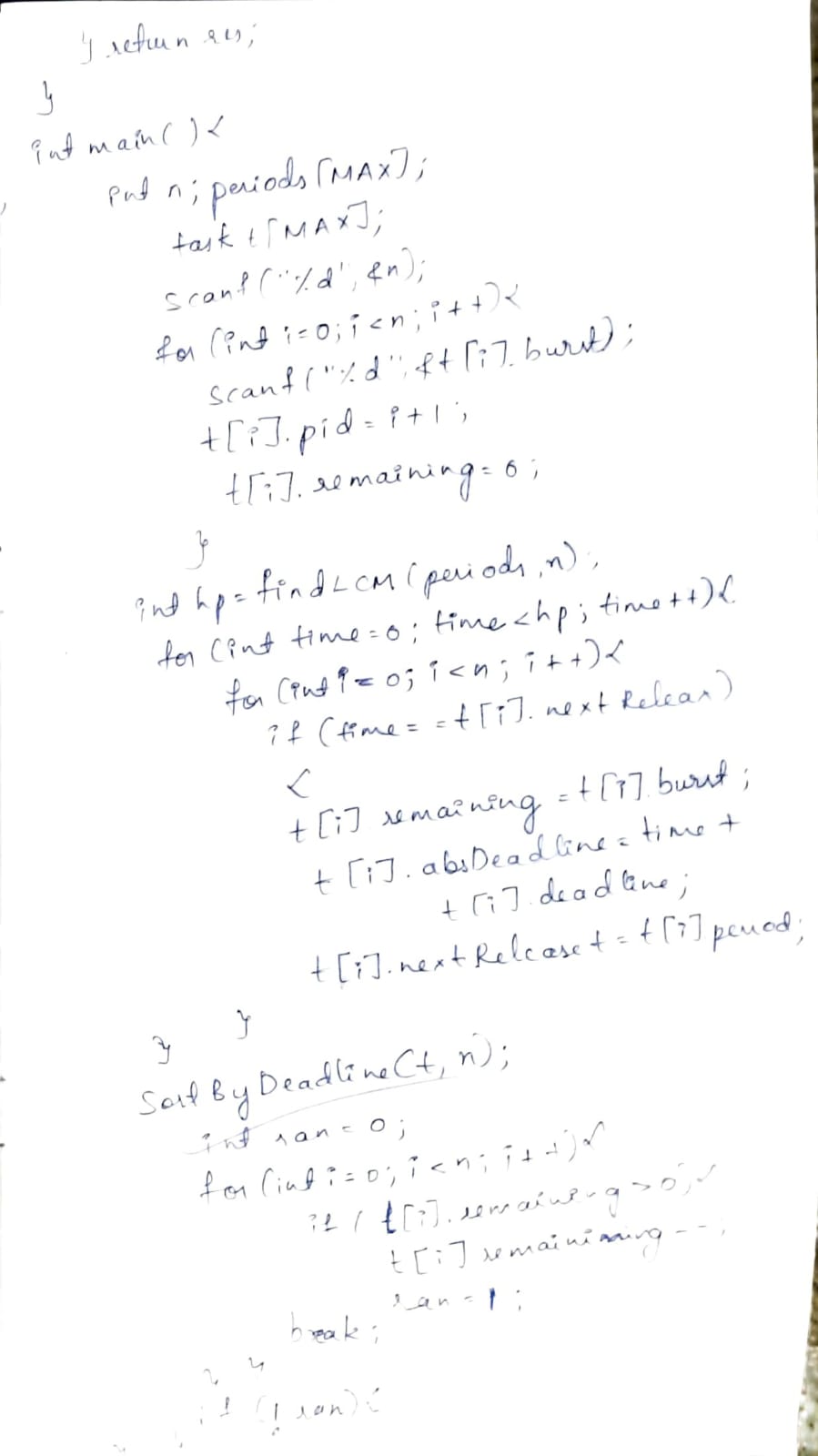
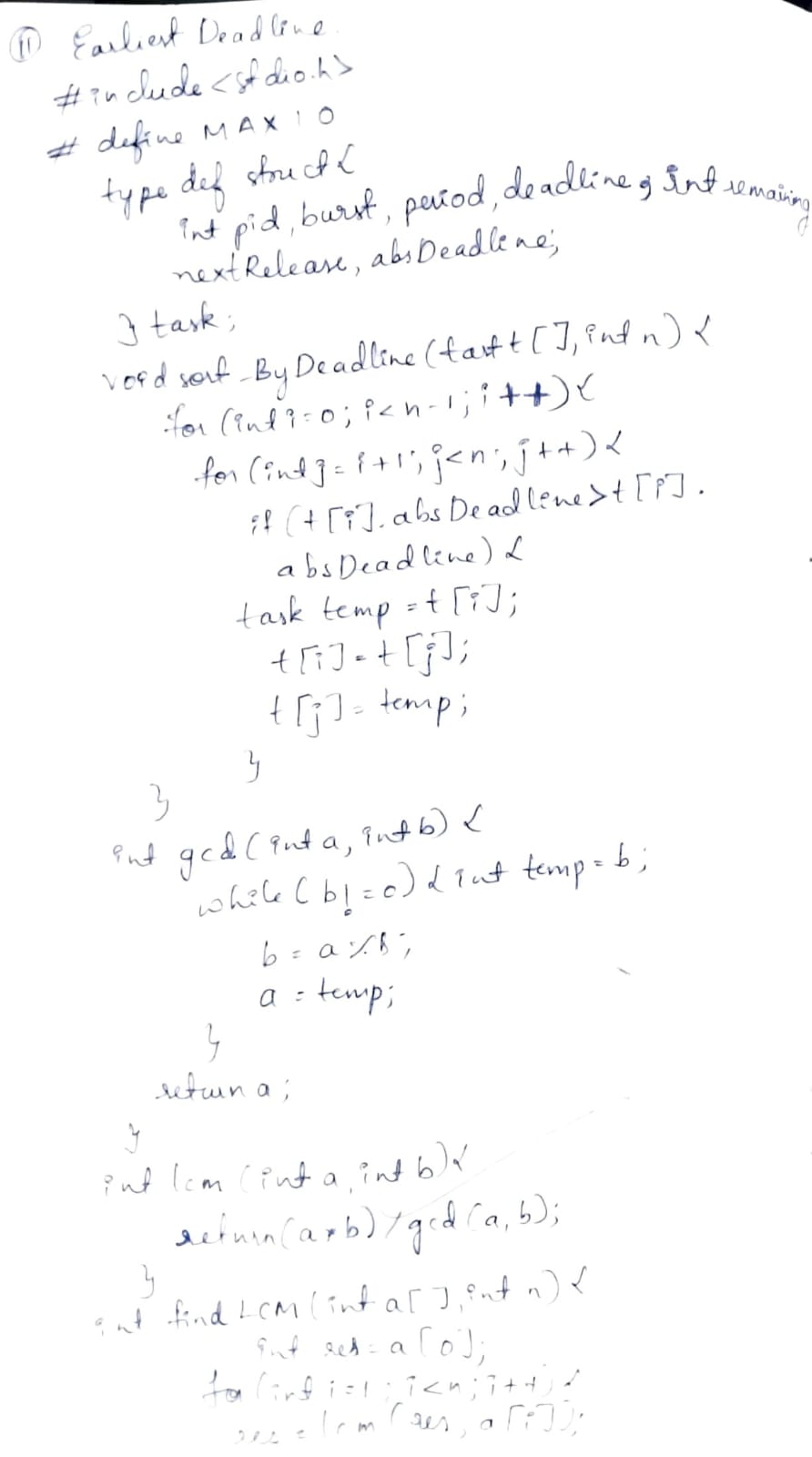
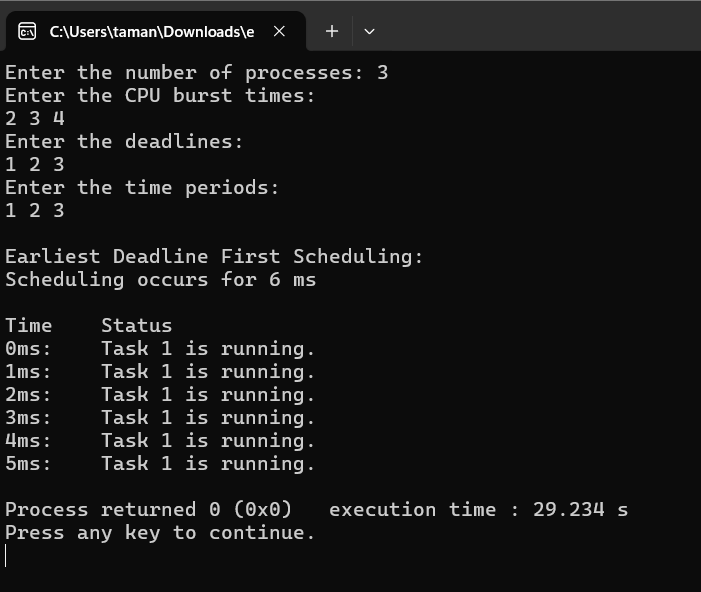
}

}

return 0;

}

RESULT:



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

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

PRODUCER CONSUMER

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

sem\_t mutex;

sem\_t full;

sem\_t empty;

int x = 0;

void producer();

void consumer();

int main() {

int choice;

int buffer\_capacity;

printf("Enter buffer capacity: ");

scanf("%d", &buffer\_capacity);

if (buffer\_capacity <= 0) {

printf("Invalid buffer capacity. Exiting...\n");

return 1;

}

sem\_init(&mutex, 0, 1);

sem\_init(&full, 0, 0);

sem\_init(&empty, 0, buffer\_capacity);

while (1) {

printf("\nChoose an option:\n");

printf("1. Producer\n");

printf("2. Consumer\n");

printf("3. Exit\n");

printf("Enter choice: ");

scanf("%d", &choice);

if (choice == 1) {

producer();

} else if (choice == 2) {

consumer();

} else if (choice == 3) {

break;

} else {

printf("Invalid choice, please try again.\n");

}

}

// Clean up semaphores

sem\_destroy(&mutex);

sem\_destroy(&full);

sem\_destroy(&empty);

return 0;

}

void producer() {

// Wait on mutex for mutual exclusion

sem\_wait(&mutex);

// Try to consume from the empty semaphore (check if there's space in the buffer)

if (sem\_trywait(&empty) == 0) {

x++;

printf("Produced item: %d\n", x);

sem\_post(&full);

} else {

printf("Buffer is full. Cannot produce more items.\n");

}

sem\_post(&mutex);

}

void consumer() {

sem\_wait(&mutex);

if (sem\_trywait(&full) == 0) {

printf("Consumed item: %d\n", x);

x--;

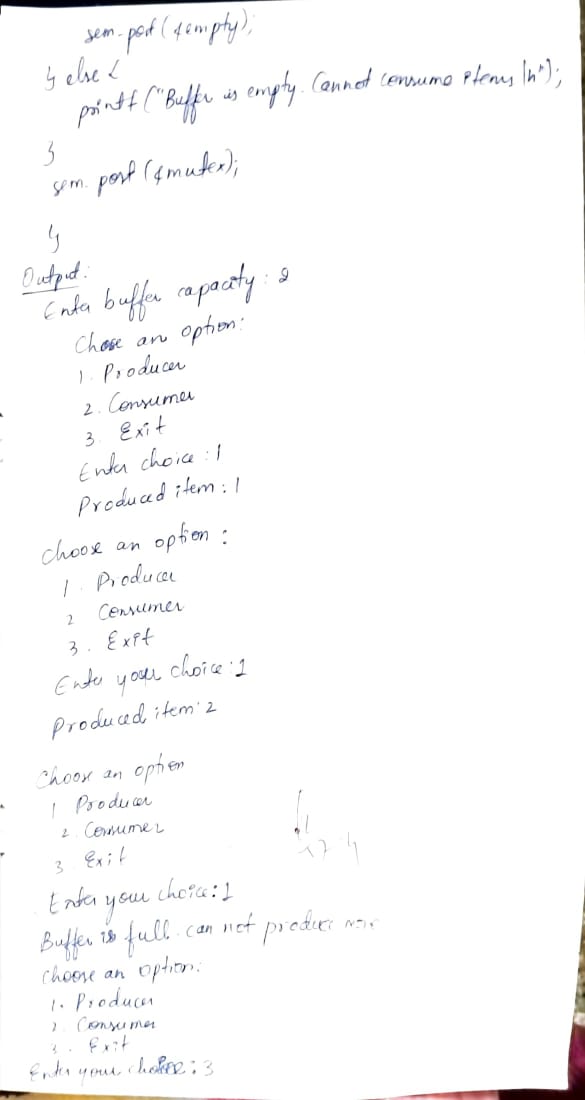
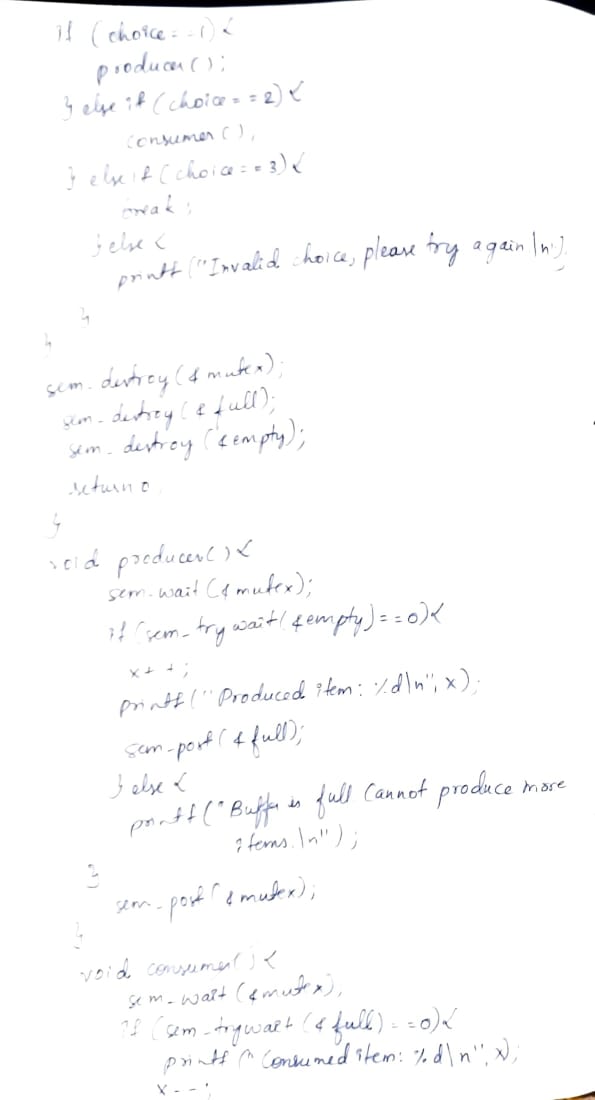
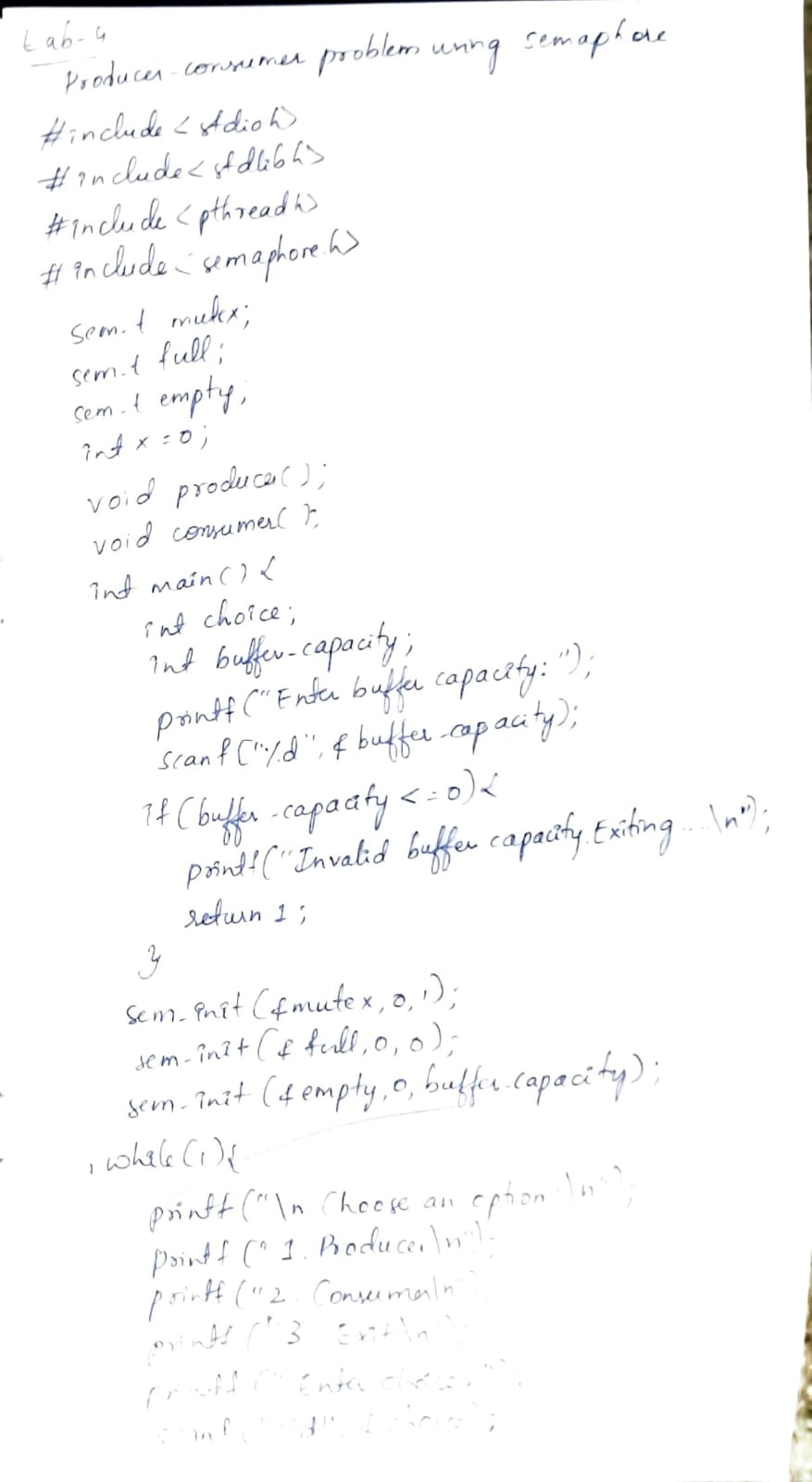
sem\_post(&empty);

} else {

printf("Buffer is empty. Cannot consume items.\n");

}

sem\_post(&mutex);

}

DINING PHILOSOPHER

#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");

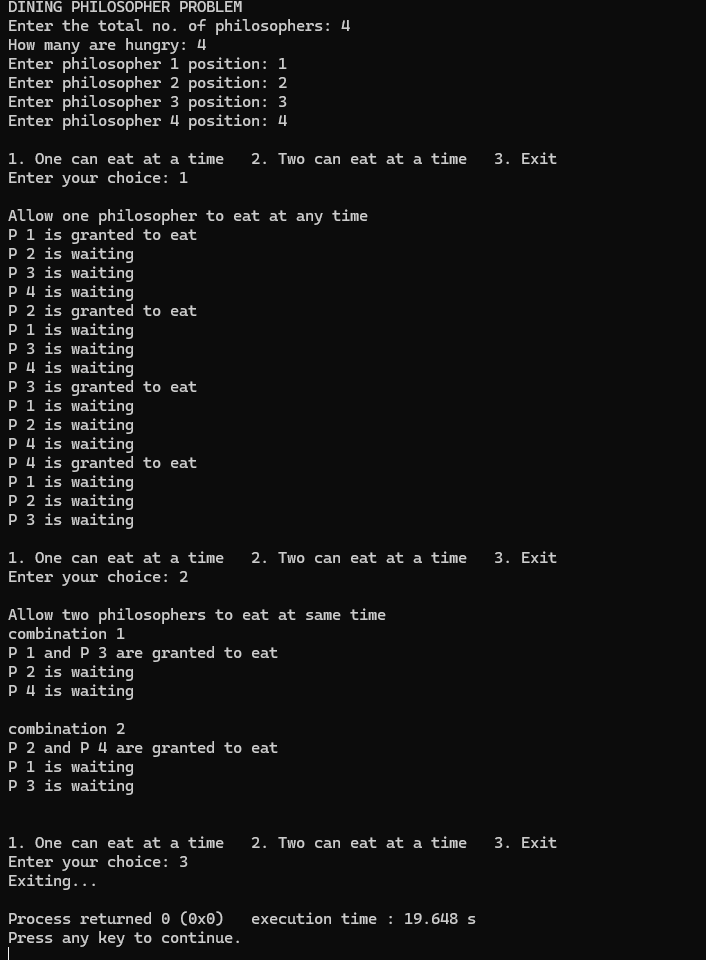
}

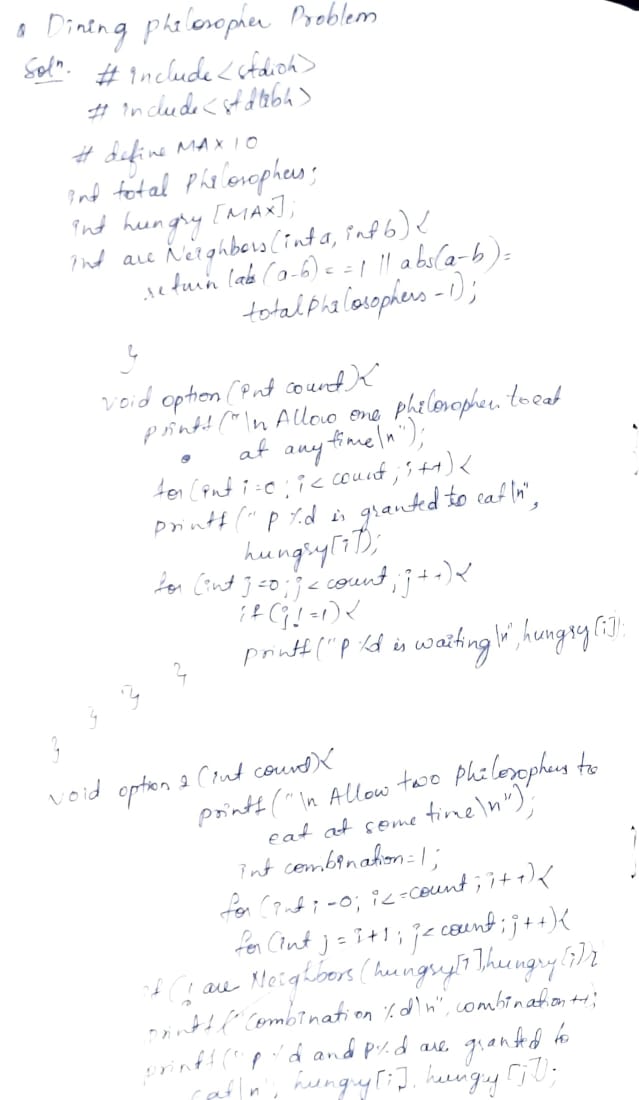
} while (choice != 3);

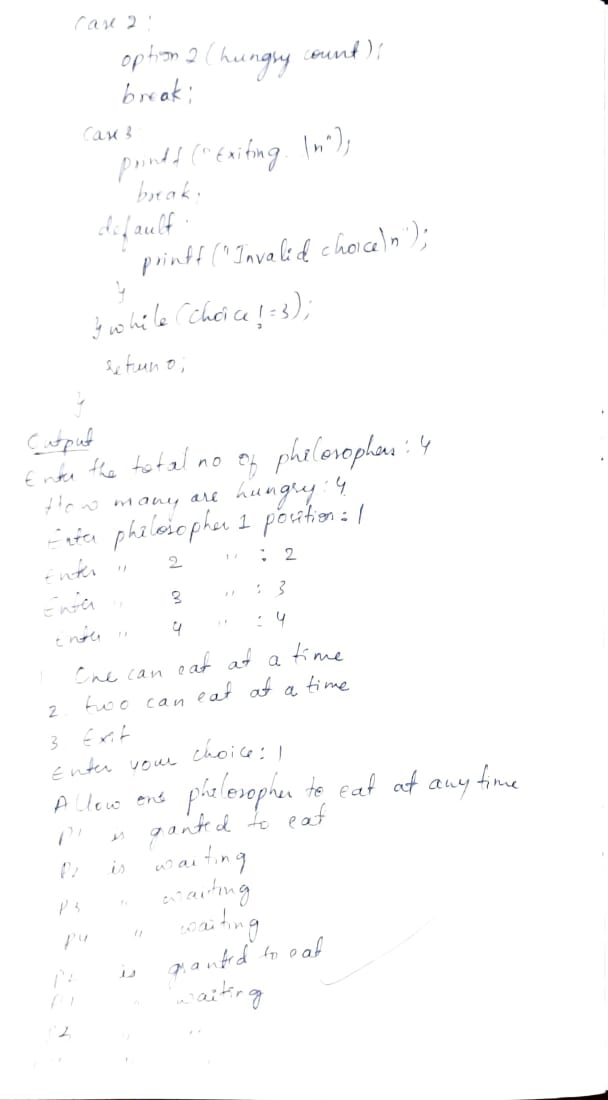
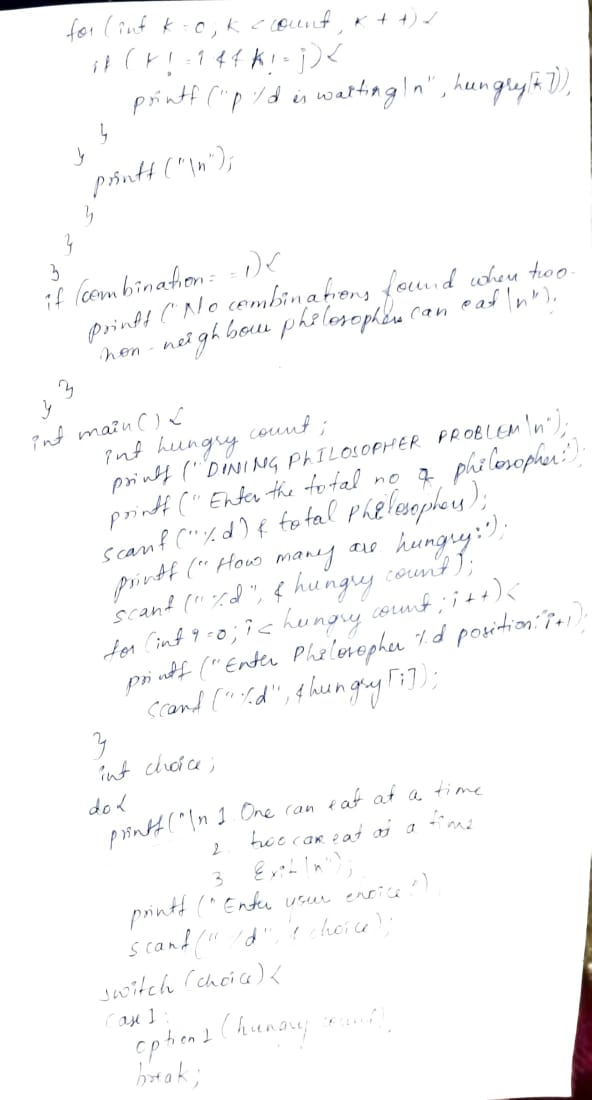
return 0;

}

RESULT:







LAB-5

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

DEADLOCK AVOIDENCE:

#include <stdio.h>

#include <stdbool.h>

int main() {

int n, m;

printf("Enter number of processes and 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", &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 matrix:\n");

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

scanf("%d", &avail[i]);

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

for (int j = 0; j < m; j++)

need[i][j] = max[i][j] - alloc[i][j];

bool finish[n];

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

int safeSeq[n];

int count = 0;

while (count < n) {

bool found = false;

for (int p = 0; p < n; p++) {

if (!finish[p]) {

bool canAllocate = true;

for (int j = 0; j < m; j++) {

if (need[p][j] > avail[j]) {

canAllocate = 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 a safe state.\n");

return 1;

}

}

printf("System is in safe state.\n");

printf("Safe sequence is: ");

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

printf("P%d", safeSeq[i]);

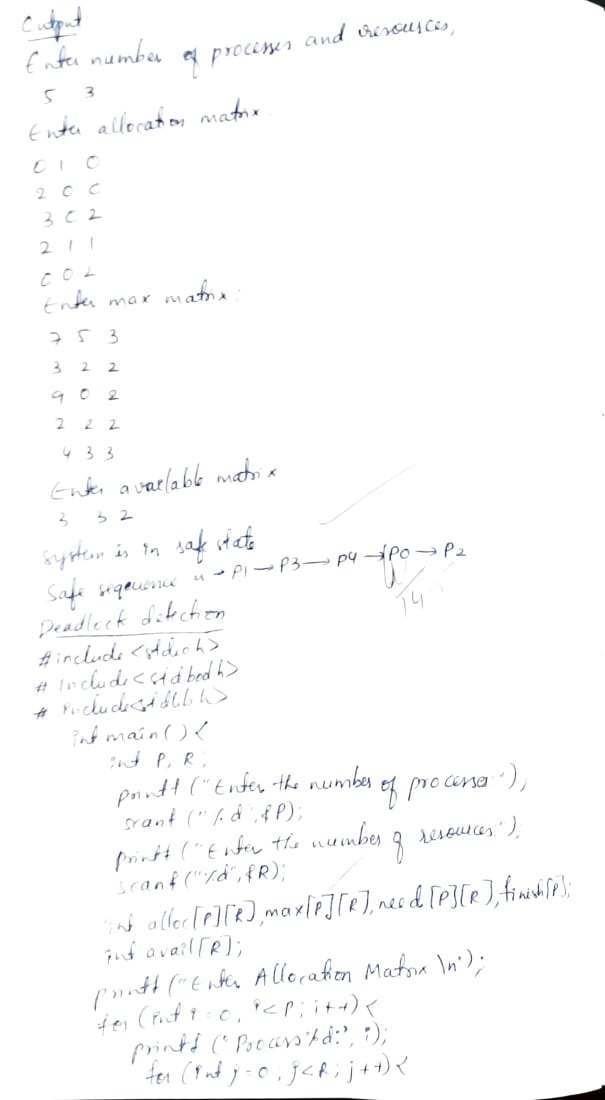
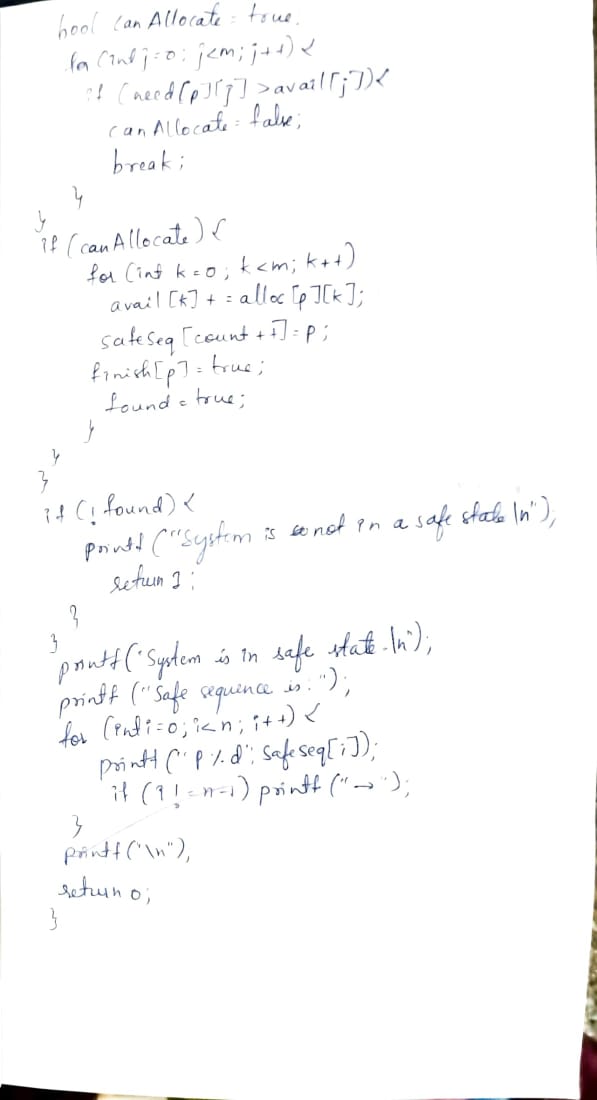
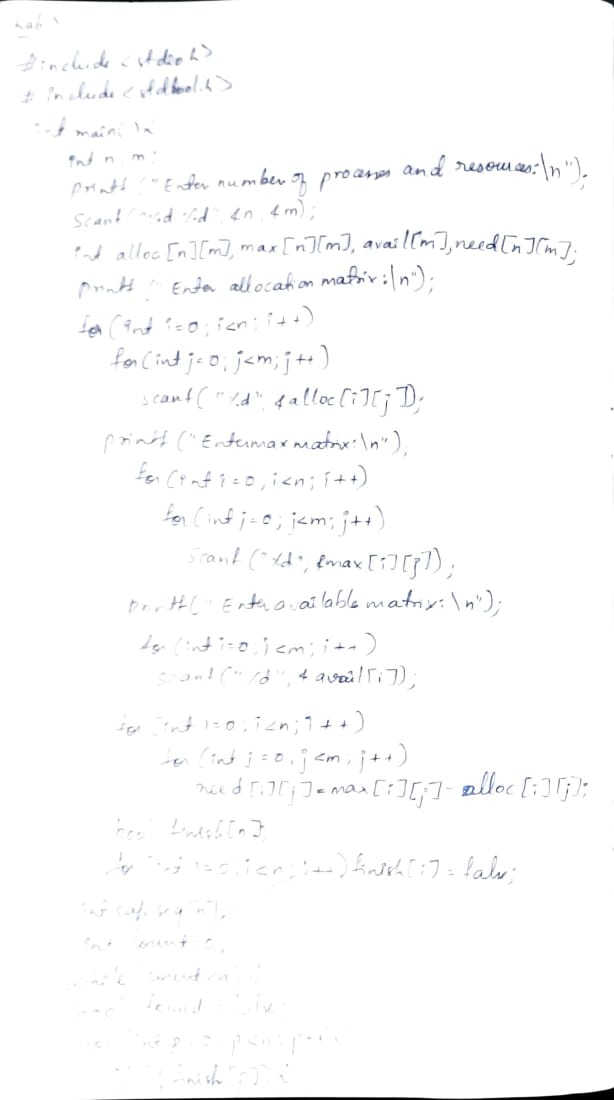
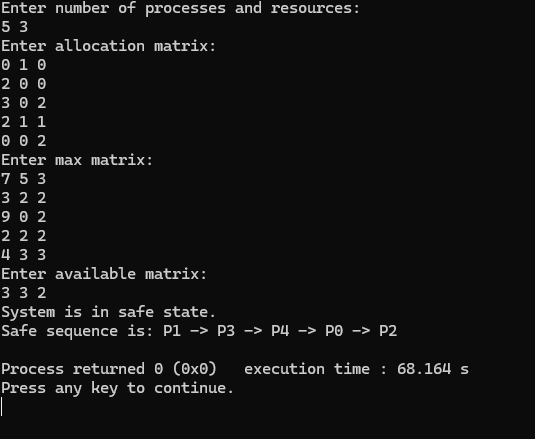
if (i != n - 1) printf(" -> ");

}

printf("\n");

return 0;

}



DEADLOCK DETECTION:

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

int main() {

int P, R;

printf("Enter the number of processes: ");

scanf("%d", &P);

printf("Enter the number of resources: ");

scanf("%d", &R);

int alloc[P][R], max[P][R], need[P][R], finish[P];

int avail[R];

printf("Enter Allocation Matrix:\n");

for (int i = 0; i < P; i++) {

printf("Process %d: ", i);

for (int j = 0; j < R; j++) {

scanf("%d", &alloc[i][j]);

}

}

printf("Enter Maximum Matrix:\n");

for (int i = 0; i < P; i++) {

printf("Process %d: ", i);

for (int j = 0; j < R; j++) {

scanf("%d", &max[i][j]);

}

}

printf("Enter Available Resources:\n");

for (int i = 0; i < R; i++) {

scanf("%d", &avail[i]);

}

for (int i = 0; i < P; i++) {

finish[i] = 0;

for (int j = 0; j < R; j++) {

need[i][j] = max[i][j] - alloc[i][j];

}

}

int count = 0;

bool deadlock = false;

while (count < P) {

bool found = false;

for (int i = 0; i < P; i++) {

if (!finish[i]) {

int j;

for (j = 0; j < R; j++) {

if (need[i][j] > avail[j])

break;

}

if (j == R) {

for (int k = 0; k < R; k++)

avail[k] += alloc[i][k];

finish[i] = 1;

found = true;

count++;

printf("Process %d can finish.\n", i);

}

}

}

if (!found) {

deadlock = true;

break;

}

}

if (!deadlock)

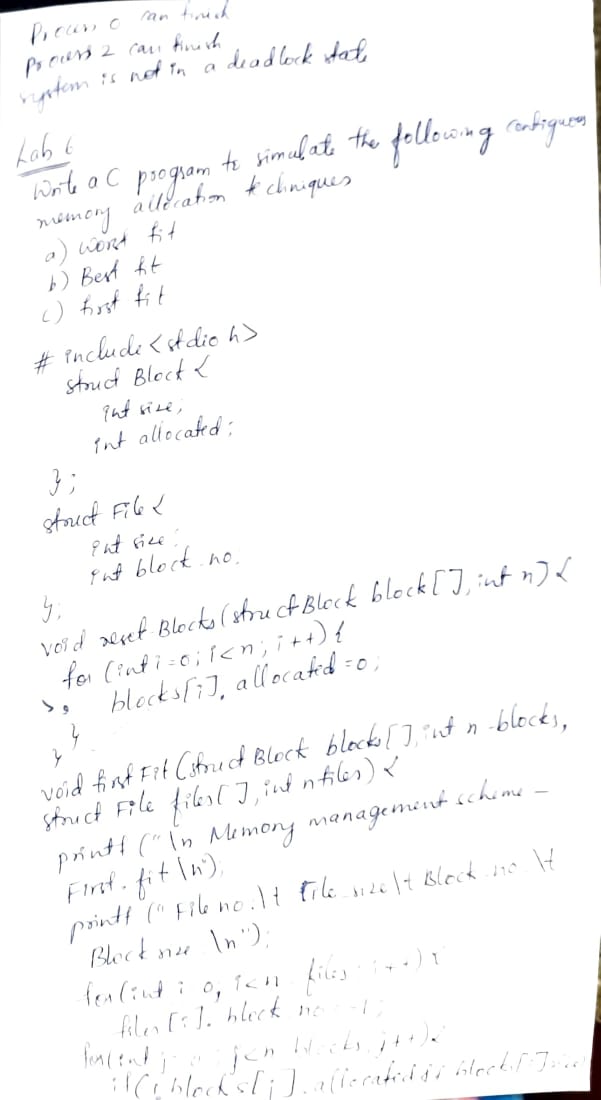
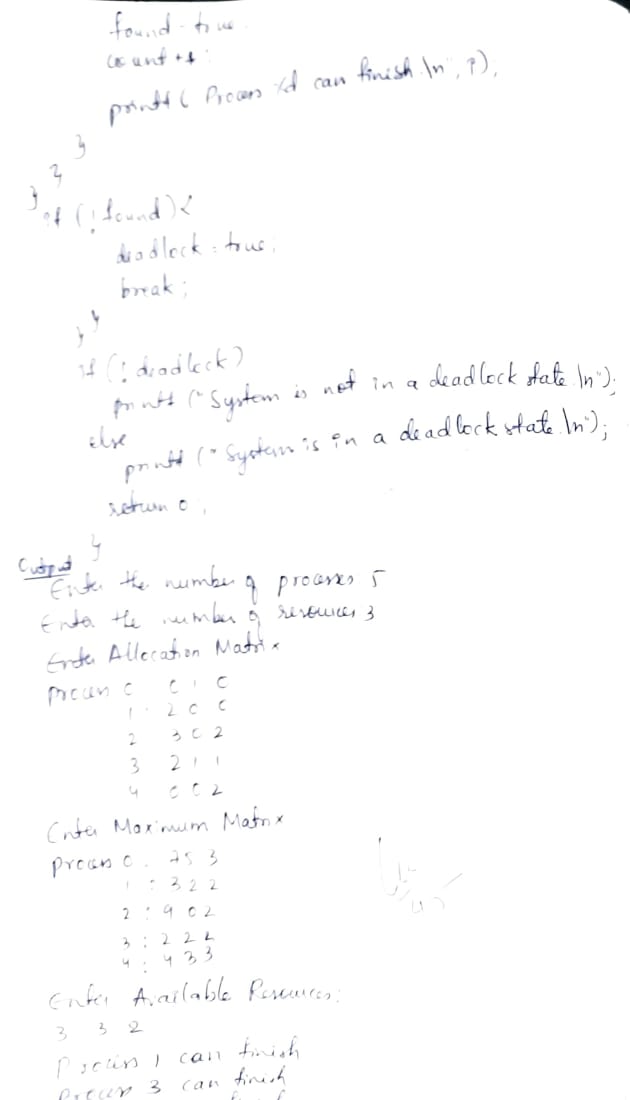
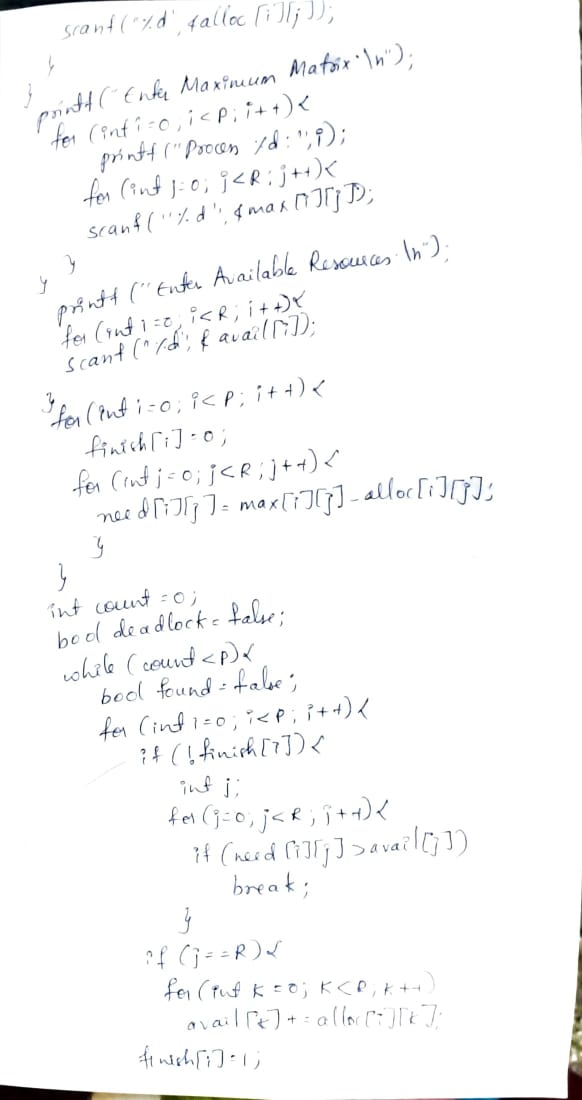
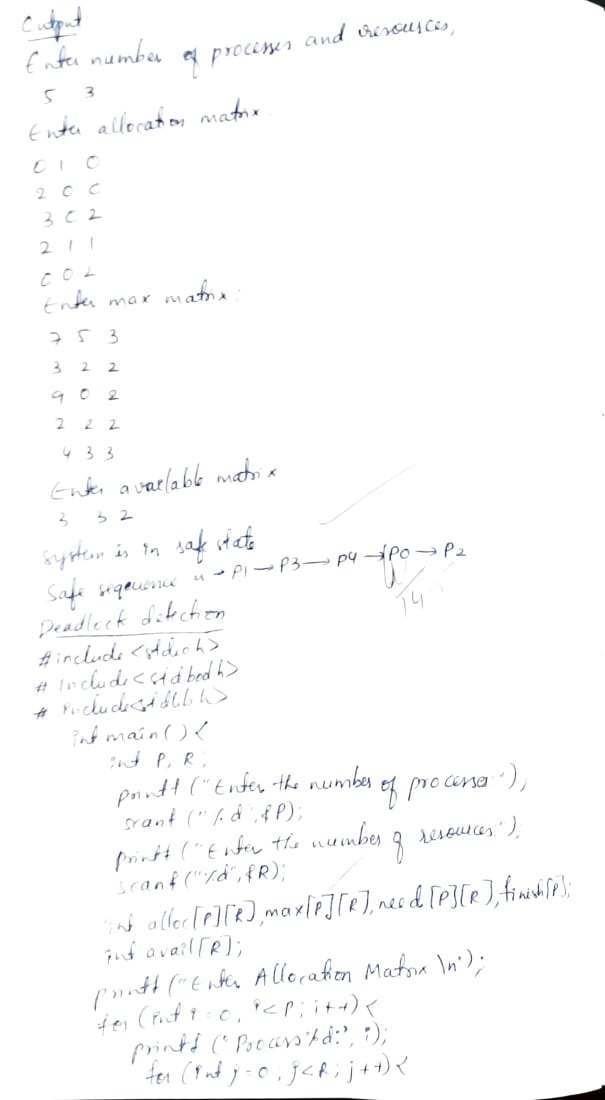
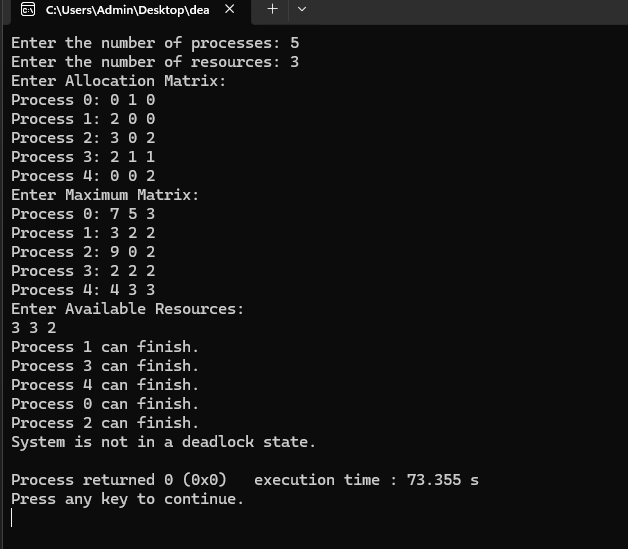
printf("System is not in a deadlock state.\n");

else

printf("System is in a deadlock state.\n");

return 0;

}



LAB-6

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

a) Worst-fit

1. Best-fit

c)First-fit

MEMORY ALLOCATION:

#include <stdio.h>

struct Block {

int size;

int allocated;

};

struct File {

int size;

int block\_no;

};

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

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

blocks[i].allocated = 0;

}

}

void firstFit(struct Block blocks[], int n\_blocks, struct File files[], int n\_files) {

printf("\n\tMemory Management Scheme – First Fit\n");

printf("File\_no:\tFile\_size\tBlock\_no:\tBlock\_size:\n");

for (int i = 0; i < n\_files; i++) {

files[i].block\_no = -1;

for (int j = 0; j < n\_blocks; j++) {

if (!blocks[j].allocated && blocks[j].size >= files[i].size) {

files[i].block\_no = j + 1;

blocks[j].allocated = 1;

printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, files[i].size, j + 1, blocks[j].size);

break;

}

}

if (files[i].block\_no == -1) {

printf("%d\t\t%d\t\t\_\t\t\_\n", i + 1, files[i].size);

}

}

}

void bestFit(struct Block blocks[], int n\_blocks, struct File files[], int n\_files) {

printf("\n\tMemory Management Scheme – Best Fit\n");

printf("File\_no:\tFile\_size\tBlock\_no:\tBlock\_size:\n");

for (int i = 0; i < n\_files; i++) {

int bestIdx = -1;

for (int j = 0; j < n\_blocks; j++) {

if (!blocks[j].allocated && blocks[j].size >= files[i].size) {

if (bestIdx == -1 || blocks[j].size < blocks[bestIdx].size) {

bestIdx = j;

}

}

}

if (bestIdx != -1) {

blocks[bestIdx].allocated = 1;

files[i].block\_no = bestIdx + 1;

printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, files[i].size, bestIdx + 1, blocks[bestIdx].size);

} else {

printf("%d\t\t%d\t\t\_\t\t\_\n", i + 1, files[i].size);

}

}

}

void worstFit(struct Block blocks[], int n\_blocks, struct File files[], int n\_files) {

printf("\n\tMemory Management Scheme – Worst Fit\n");

printf("File\_no:\tFile\_size\tBlock\_no:\tBlock\_size:\n");

for (int i = 0; i < n\_files; i++) {

int worstIdx = -1;

for (int j = 0; j < n\_blocks; j++) {

if (!blocks[j].allocated && blocks[j].size >= files[i].size) {

if (worstIdx == -1 || blocks[j].size > blocks[worstIdx].size) {

worstIdx = j;

}

}

}

if (worstIdx != -1) {

blocks[worstIdx].allocated = 1;

files[i].block\_no = worstIdx + 1;

printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, files[i].size, worstIdx + 1, blocks[worstIdx].size);

} else {

printf("%d\t\t%d\t\t\_\t\t\_\n", i + 1, files[i].size);

}

}

}

int main() {

int n\_blocks, n\_files, choice;

printf("Memory Management Scheme\n");

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];

printf("\nEnter the size of the blocks:\n");

for (int i = 0; i < n\_blocks; i++) {

printf("Block %d: ", i + 1);

scanf("%d", &blocks[i].size);

blocks[i].allocated = 0;

}

printf("Enter the size of the files:\n");

for (int i = 0; i < n\_files; i++) {

printf("File %d: ", i + 1);

scanf("%d", &files[i].size);

}

do {

printf("\n1. First Fit\n2. Best Fit\n3. Worst Fit\n4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

resetBlocks(blocks, n\_blocks); // Reset block allocation before each strategy

switch (choice) {

case 1:

firstFit(blocks, n\_blocks, files, n\_files);

break;

case 2:

bestFit(blocks, n\_blocks, files, n\_files);

break;

case 3:

worstFit(blocks, n\_blocks, files, n\_files);

break;

case 4:

printf("\nExiting...\n");

break;

default:

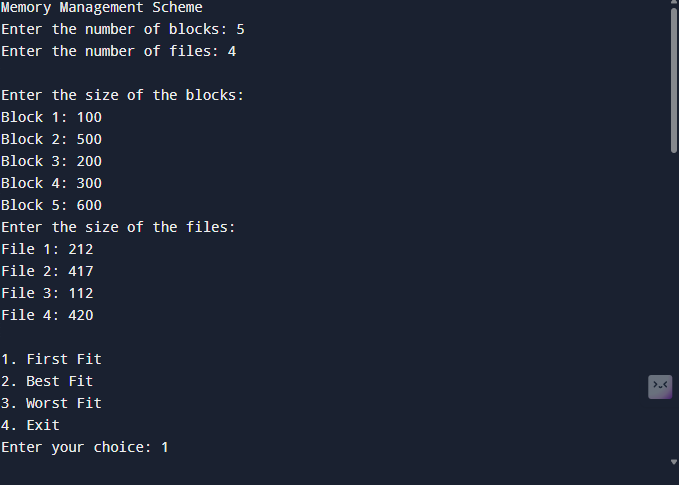
printf("Invalid choice.\n");

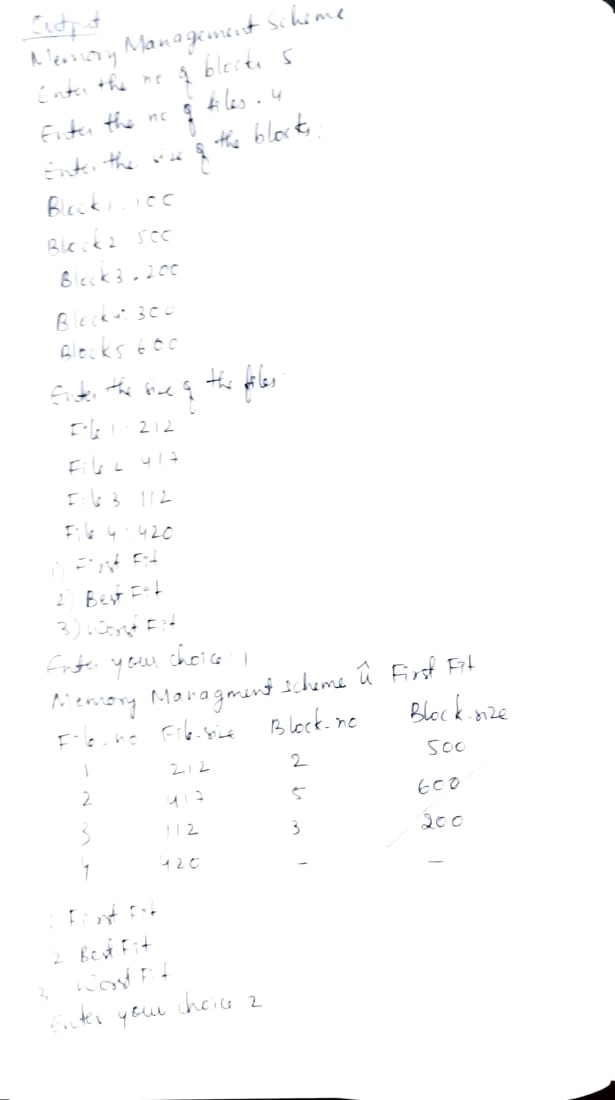
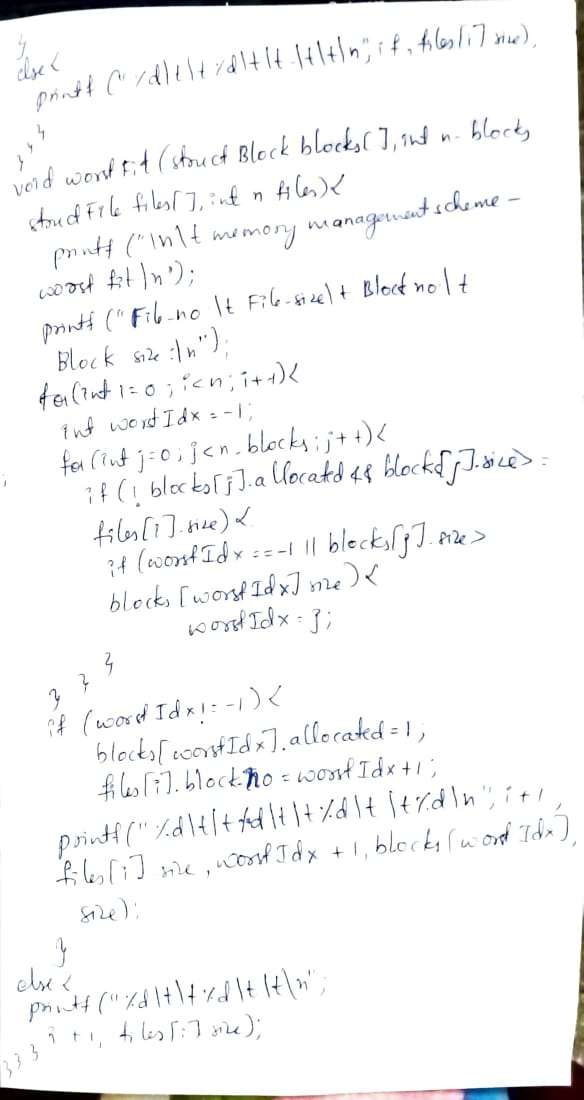
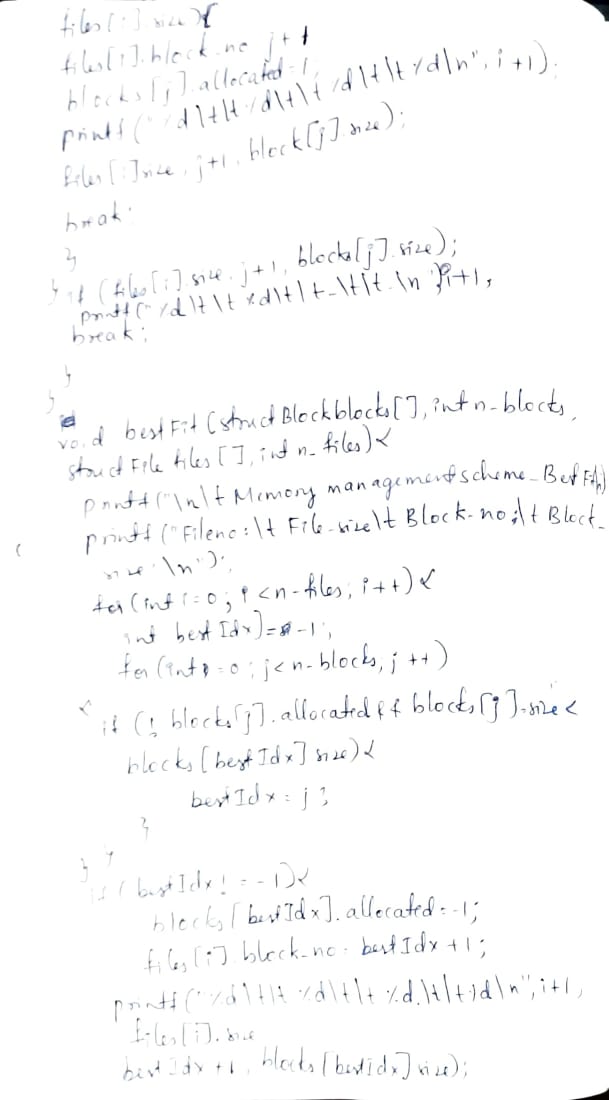
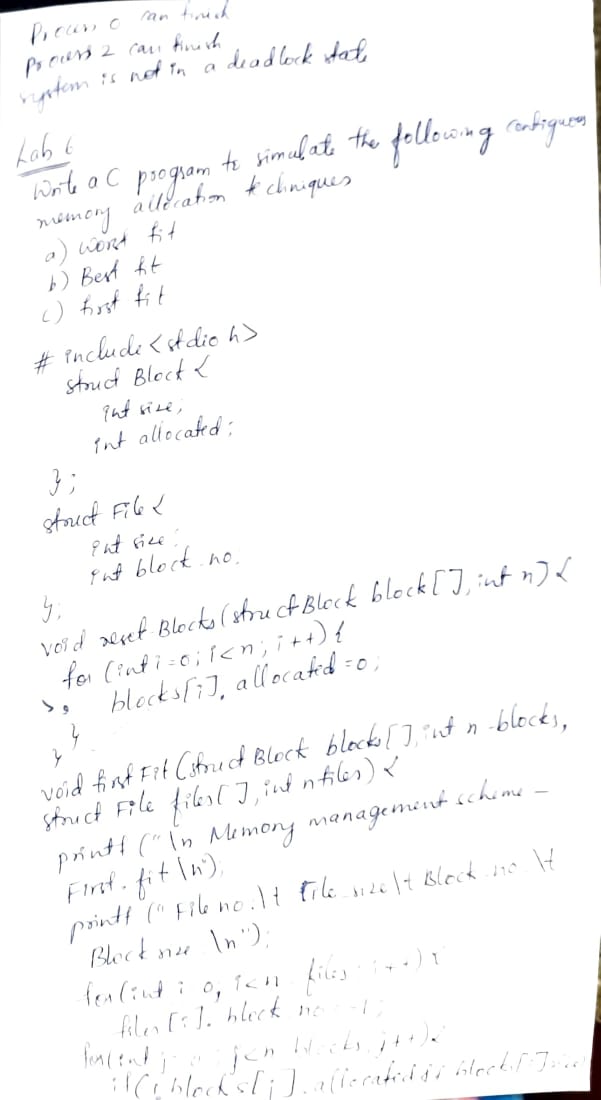
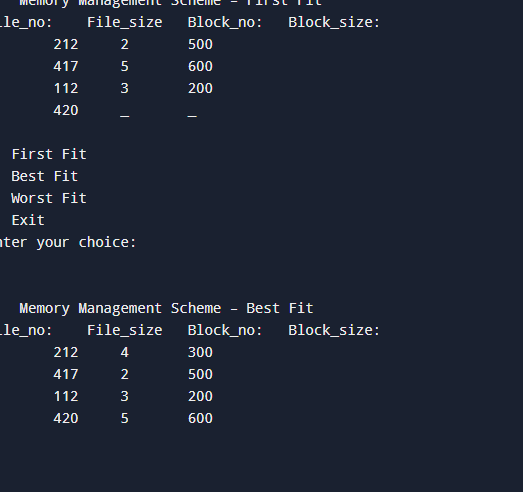
}

} while (choice != 4);

return 0;

}

RESULT:



LAB-7

Write a C program to simulate page replacement algorithms

a) FIFO

1. LRU

Optimal

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;

}

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\n", 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];

} 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[j] == pages[i]) {

time++;

recent[j] = time;

page\_hits++;

hit = true;

break;

}

}

if (!hit) {

int lru = 0;

for (int j = 1; j < capacity; j++)

if (recent[j] < recent[lru])

lru = j;

time++;

fr[lru] = pages[i];

recent[lru] = time;

page\_faults++;

}

}

printf("LRU Page Faults: %d, Page Hits: %d\n", page\_faults, page\_hits);

}

int main() {

int n, capacity;

printf("Enter the size of the pages:\n");

scanf("%d", &n);

int pages[n];

printf("Enter the page strings:\n");

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

scanf("%d", &pages[i]);

printf("Enter the no of page frames:\n");

scanf("%d", &capacity);

fifo(pages, n, capacity);

optimal(pages, n, capacity);

lru(pages, n, capacity);

return 0;

}

