VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT

on

OPERATING SYSTEMS

Submitted by

VANSH GANDHI (1WA23CS030)

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 VANSH GANDHI(1WA23CS030), who is Bonafide student of B. M. S. College of Engineering. It is in partial fulfilment for the award of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum during the year Feb 2025- June 2025. The Lab report has been approved as it satisfies the academic requirements in respect of a OPERATING SYSTEMS - (23CS4PCOPS) work prescribed for the said degree.

Dr 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) | 1-17 |
| 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) | 18-30 |
| 3. | Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue. | 30-38 |
| 4. | Write a C program to simulate Real-Time CPU Scheduling algorithms:   1. Rate- Monotonic 2. Earliest-deadline First 3. Proportional scheduling | 39-48 |
| 5. | Write a C program to simulate producer-consumer problem using semaphores | 49-54 |
| 6. | Write a C program to simulate the concept of Dining Philosophers problem. | 54-61 |
| 7. | Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance. | 62-67 |
| 8. | Write a C program to simulate deadlock detection | 68-71 |
| 9. | Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit   1. Best-fit 2. First-fit | 72-83 |
| 10. | Write a C program to simulate page replacement algorithms a) FIFO   1. LRU 2. Optimal | 84-92 |

Course Outcomes

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

Program -1

Question:

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

→FCFS

→ SJF (pre-emptive & Non-preemptive)

Code:

#include <stdio.h>

void main() {

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

int arrival[n], burst[n], waiting[n], turnaround[n], completion[n], response[n];

printf("Enter Arrival Time and Burst Time for each process:\n");

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

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

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

}

int currentTime = 0;

float totalWaiting = 0, totalTurnaround = 0;

printf("\nProcess\tArrival\tBurst\tWaiting\tTurnaround\tResponse\n");

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

if (currentTime < arrival[i])

currentTime = arrival[i];

completion[i] = currentTime + burst[i];

turnaround[i] = completion[i] - arrival[i];

waiting[i] = turnaround[i] - burst[i];

response[i] = completion[i] - arrival[i];

totalWaiting += waiting[i];

totalTurnaround += turnaround[i];

printf("%d\t%d\t%d\t%d\t%d\t\t%d\n", i + 1, arrival[i], burst[i], waiting[i], turnaround[i], response[i]);

currentTime = completion[i];

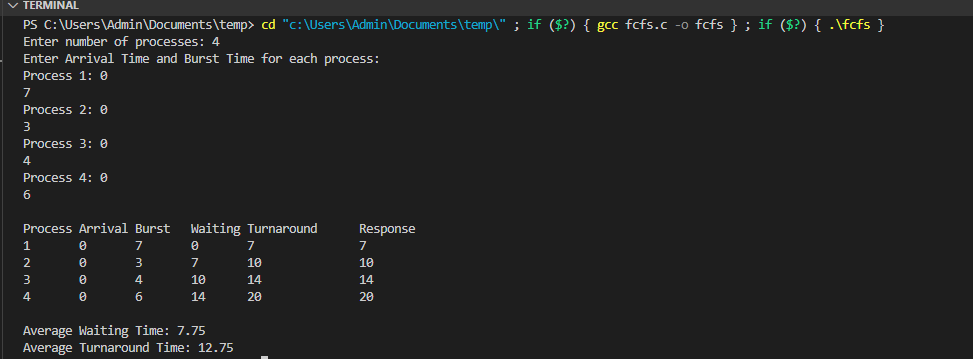
}

printf("\nAverage Waiting Time: %.2f", totalWaiting / n);

printf("\nAverage Turnaround Time: %.2f\n", totalTurnaround / n);

}

Result:



=>SJF(Non-preemptive):

Code:

#include <stdio.h>

void nonPreemptiveSJF(int n, int at[], int bt[], int ct[], int tat[], int wt[], int rt[])

{

int completed = 0, time = 0, min\_bt, shortest, finish\_time;

int remaining\_bt[n];

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

{

remaining\_bt[i] = bt[i];

}

while (completed < n)

{

min\_bt = 9999;

shortest = -1;

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

{

if (at[i] <= time && remaining\_bt[i] > 0 && bt[i] < min\_bt)

{

min\_bt = bt[i];

shortest = i;

}

}

if (shortest == -1)

{

time++;

continue;

}

time += bt[shortest];

remaining\_bt[shortest] = 0;

completed++;

ct[shortest] = time;

tat[shortest] = ct[shortest] - at[shortest];

wt[shortest] = tat[shortest] - bt[shortest];

rt[shortest] = wt[shortest];

}

}

void displayTable(int n, int at[], int bt[], int ct[], int tat[], int wt[], int rt[])

{

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

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

{

printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i], ct[i], tat[i], wt[i], rt[i]);

}

}

int main()

{

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

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

printf("Enter Arrival Time and Burst Time for each process:\n");

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

{

printf("Process %d - Arrival Time: ", i + 1);

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

printf("Process %d - Burst Time: ", i + 1);

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

}

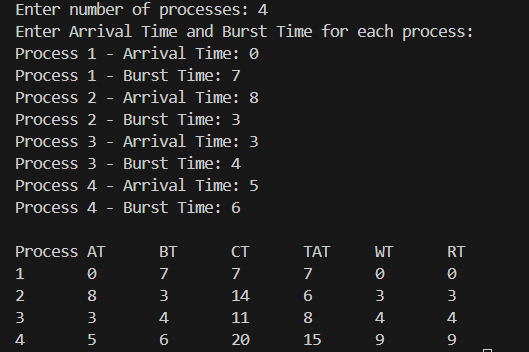
nonPreemptiveSJF(n, at, bt, ct, tat, wt, rt);

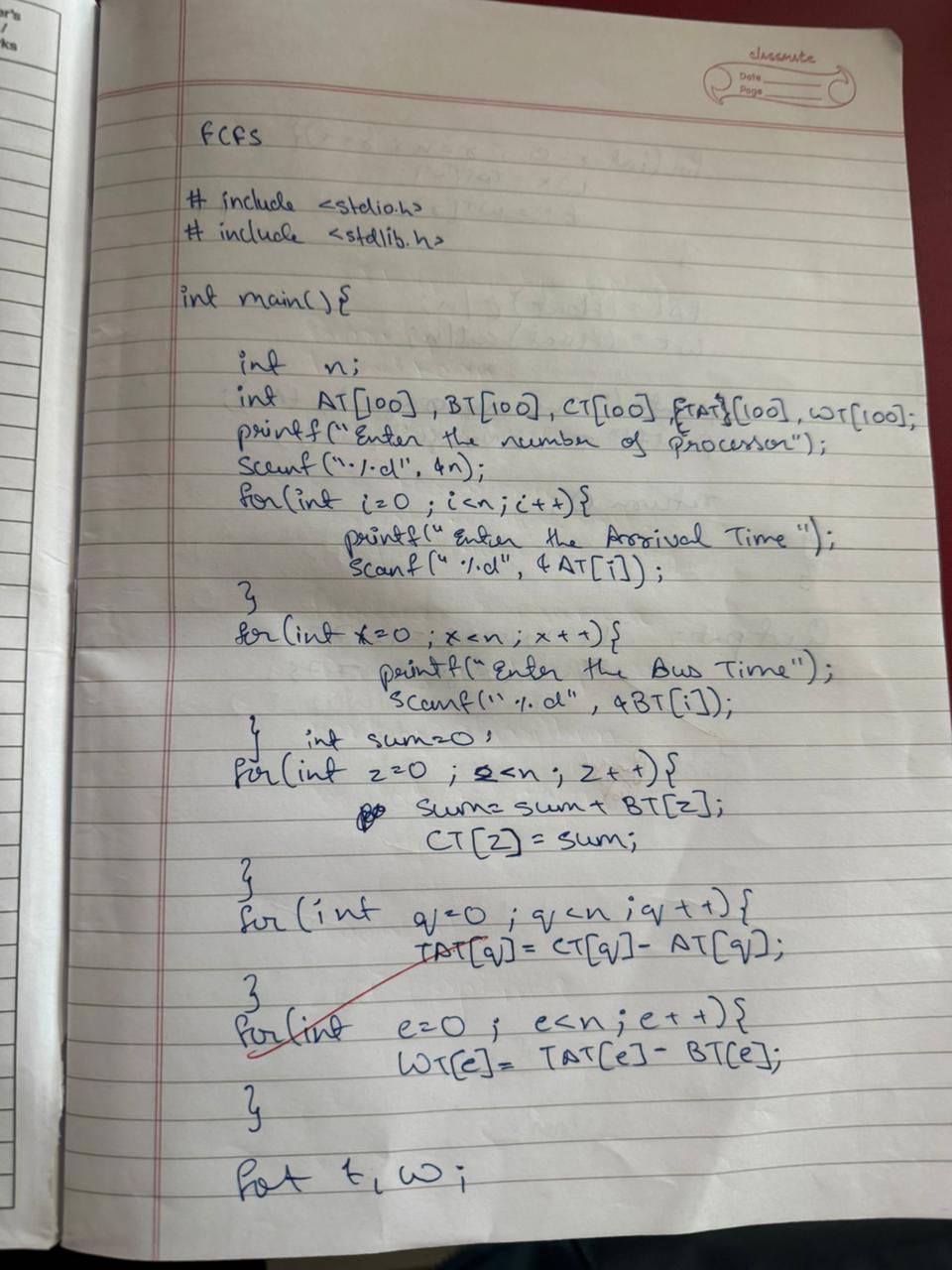
displayTable(n, at, bt, ct, tat, wt, rt);

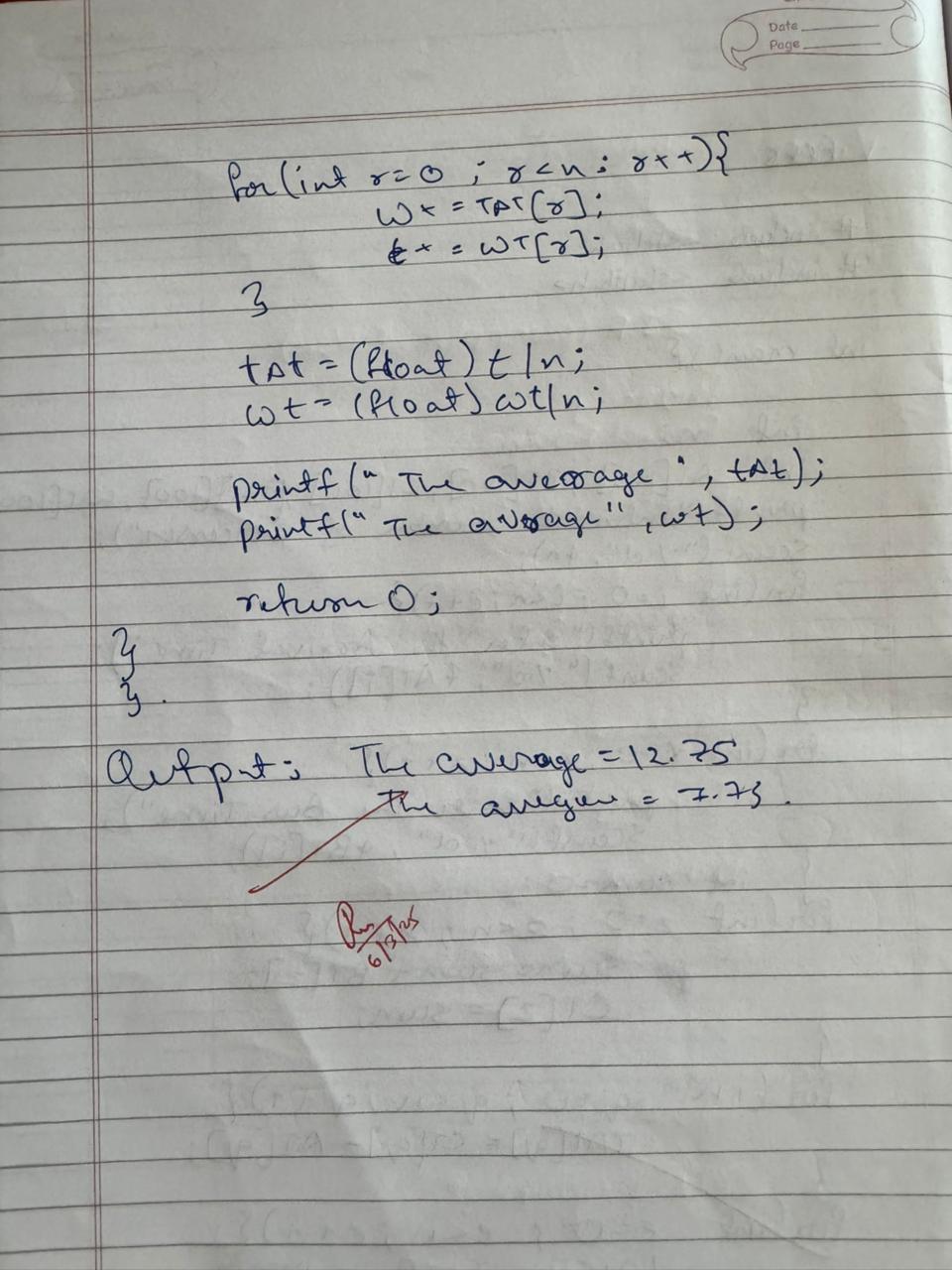
return 0;

}

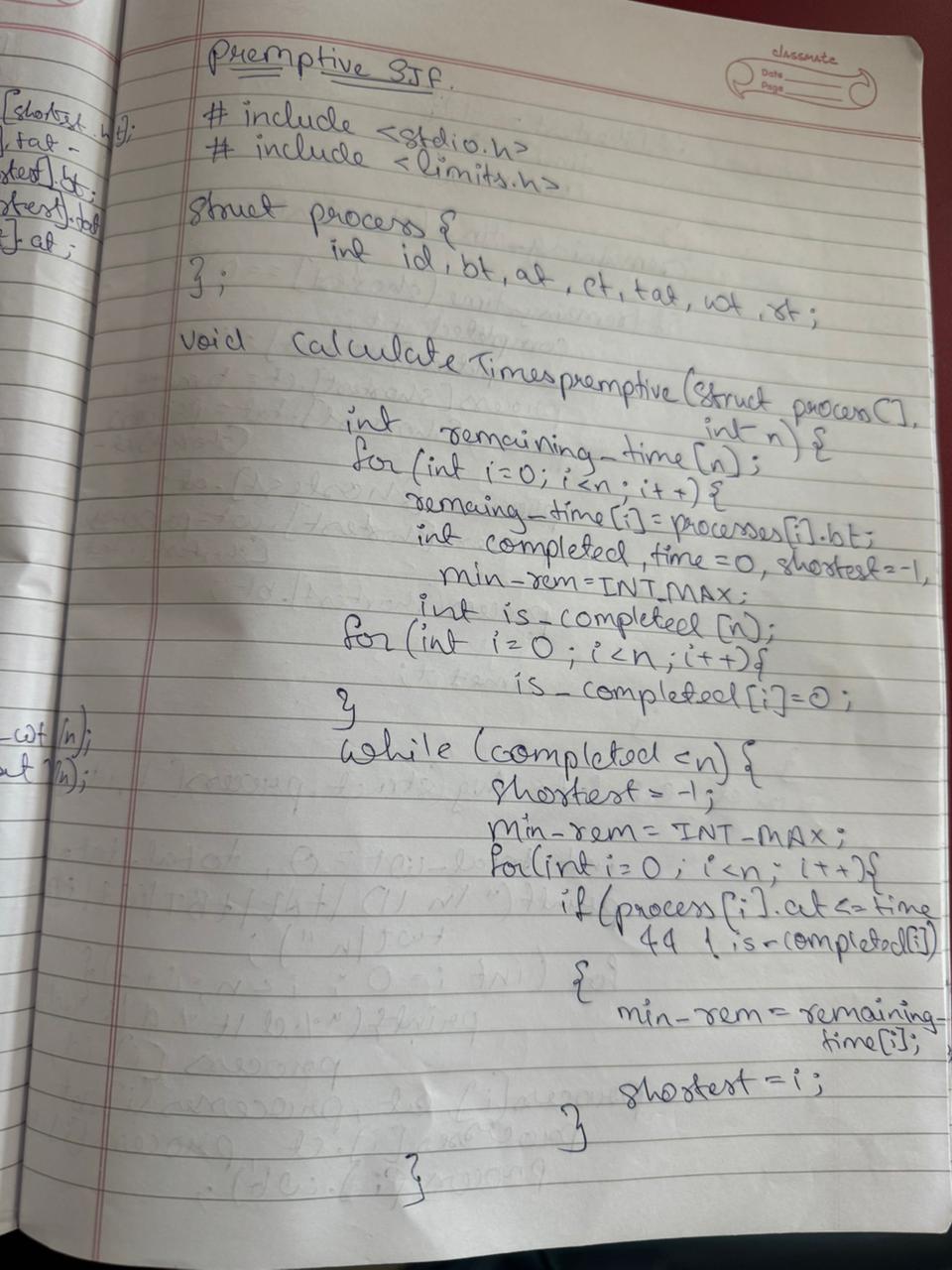
Output:

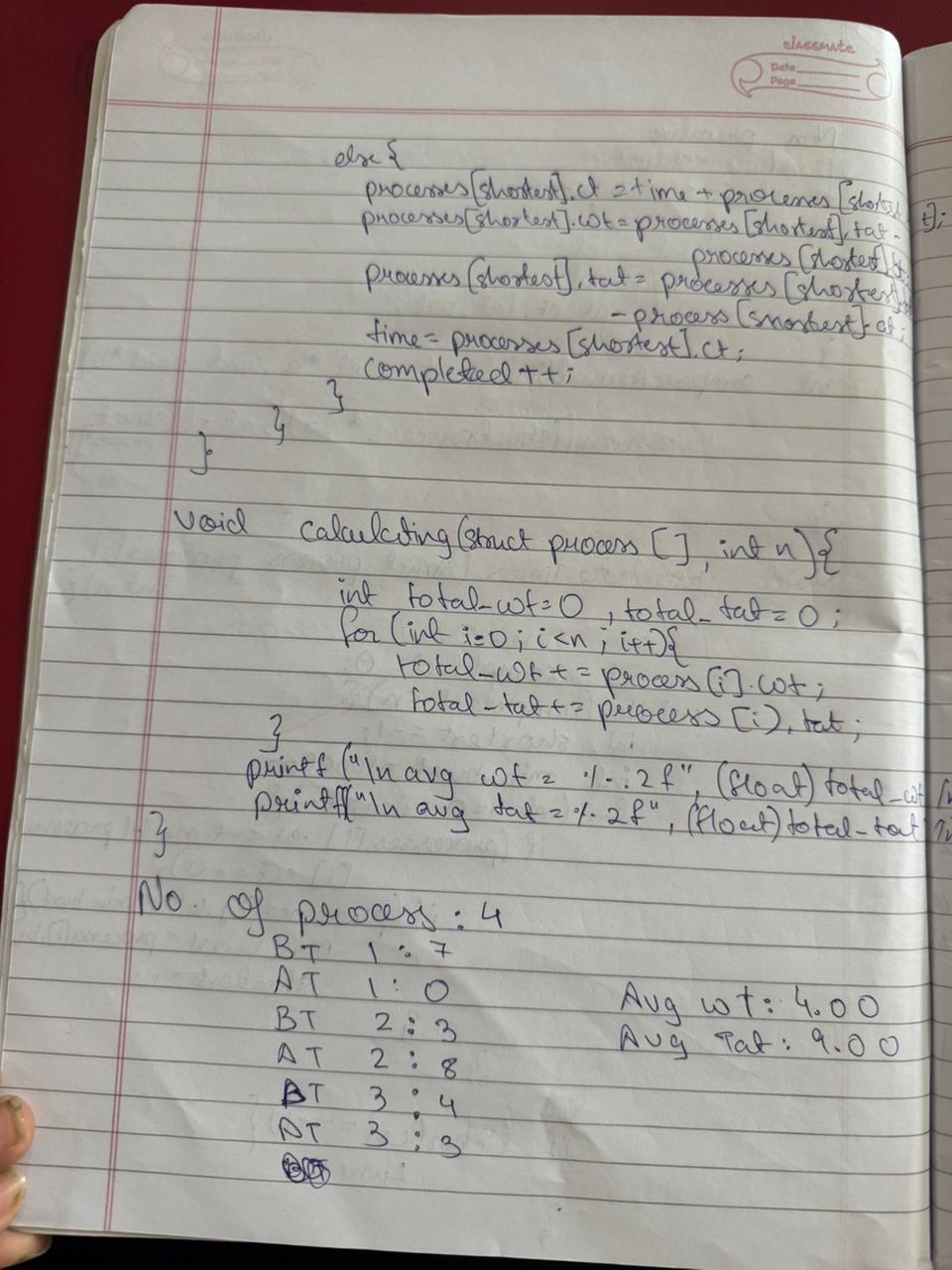


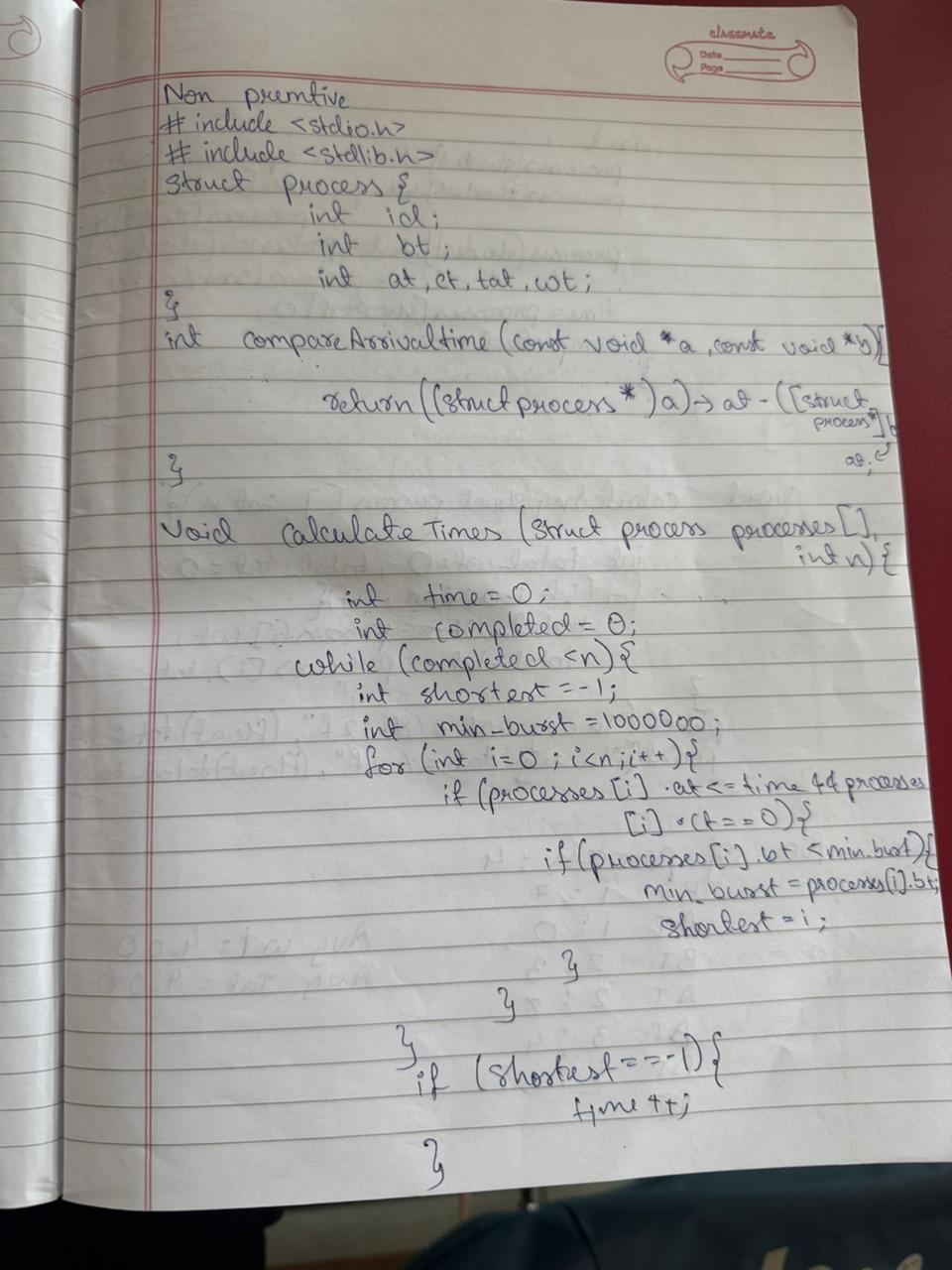


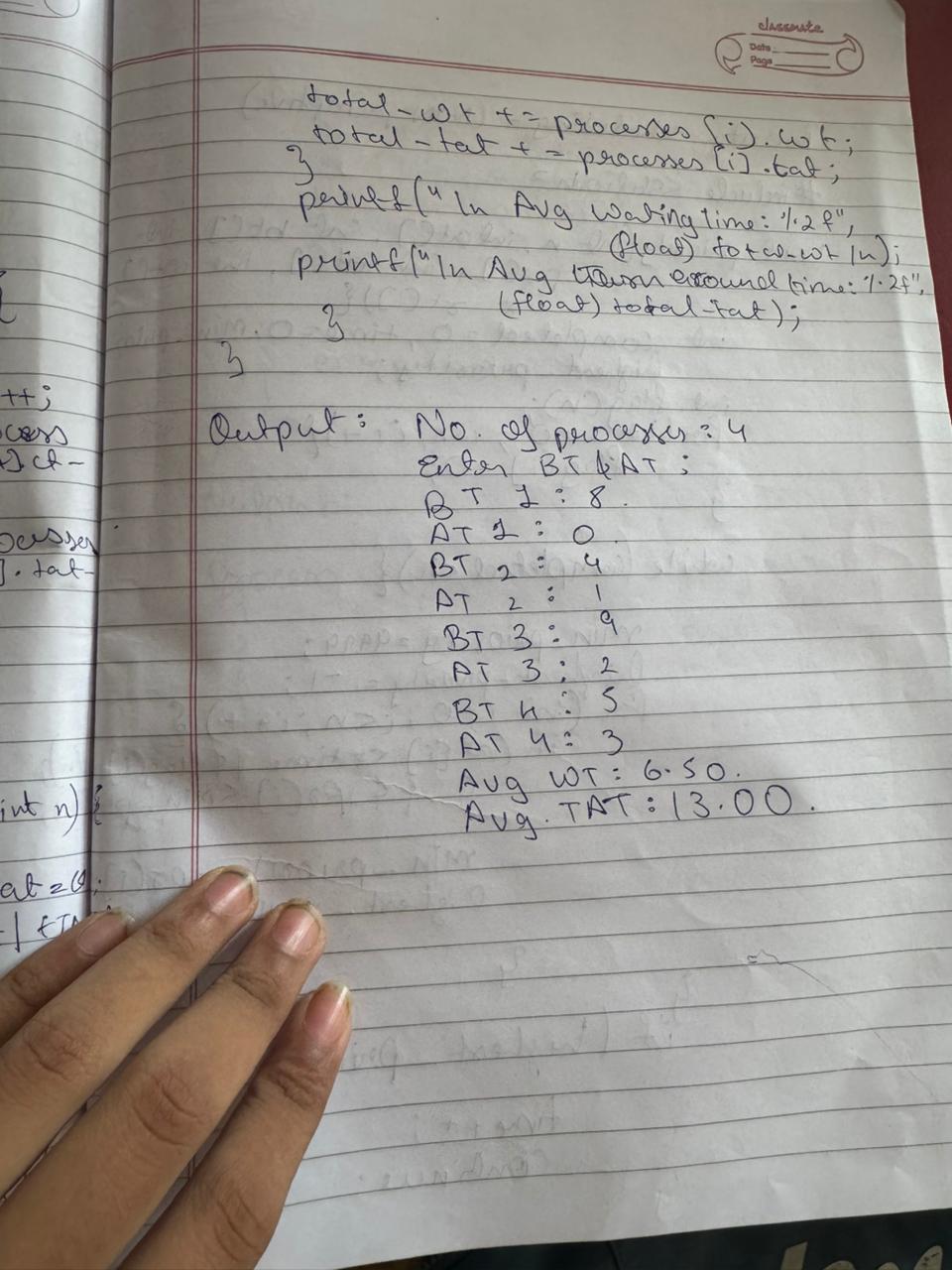










Code

#include <stdio.h>

void preemptiveSJF(int n, int at[], int bt[], int ct[], int tat[], int wt[], int rt[])

{

int remaining\_bt[n];

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

int flag[n];

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

{

remaining\_bt[i] = bt[i];

flag[i] = 0;

}

while (completed < n)

{

min\_bt = 9999;

shortest = -1;

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

{

if (at[i] <= time && remaining\_bt[i] > 0 && remaining\_bt[i] < min\_bt && flag[i] == 0)

{

min\_bt = remaining\_bt[i];

shortest = i;

}

}

if (shortest == -1)

{

time++;

continue;

}

remaining\_bt[shortest]--;

if (remaining\_bt[shortest] == 0)

{

completed++;

flag[shortest] = 1;

ct[shortest] = time + 1;

tat[shortest] = ct[shortest] - at[shortest];

wt[shortest] = tat[shortest] - bt[shortest];

rt[shortest] = wt[shortest];

}

time++;

}

}

void displayTable(int n, int at[], int bt[], int ct[], int tat[], int wt[], int rt[])

{

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

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

{

printf("%d\t%d\t%d\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i], ct[i], tat[i], wt[i], rt[i]);

}

}

int main()

{

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

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

printf("Enter Arrival Time and Burst Time for each process:\n");

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

{

printf("Process %d - Arrival Time: ", i + 1);

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

printf("Process %d - Burst Time: ", i + 1);

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

}

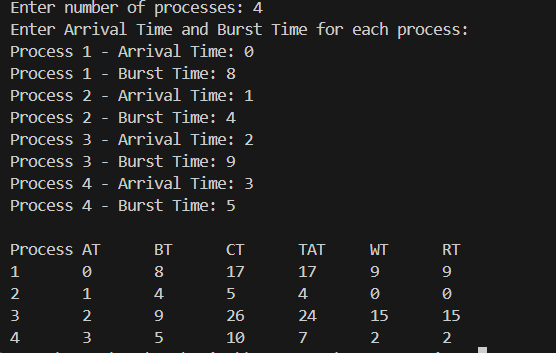
preemptiveSJF(n, at, bt, ct, tat, wt, rt);

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

return 0;

}

Output:



Program 2

Question

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

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

→ Round Robin (Experiment with different quantum sizes for RR algorithm)

=> Priority Scheduling (Non-preemptive):

Code

#include <stdio.h>

//non-preemptive

void priorityScheduling(int n, int at[], int bt[], int pr[], int ct[], int tat[], int wt[], int rt[]) {

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

int flag[n];

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

flag[i] = 0;

}

while (completed < n) {

min\_priority = 9999;

highest\_priority = -1;

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

if (at[i] <= time && flag[i] == 0 && pr[i] < min\_priority) {

min\_priority = pr[i];

highest\_priority = i;

}

}

if (highest\_priority == -1) {

time++;

continue;

}

time += bt[highest\_priority];

flag[highest\_priority] = 1;

ct[highest\_priority] = time;

tat[highest\_priority] = ct[highest\_priority] - at[highest\_priority];

wt[highest\_priority] = tat[highest\_priority] - bt[highest\_priority];

rt[highest\_priority] = wt[highest\_priority];

completed++;

}

}

void displayTable(int n, int at[], int bt[], int pr[], int ct[], int tat[], int wt[], int rt[]) {

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

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

printf("%d\t%d\t%d\t%d\t\t%d\t%d\t%d\t%d\n", i + 1, at[i], bt[i], pr[i], ct[i], tat[i], wt[i], rt[i]);

}

}

int main() {

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

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

printf("Enter Arrival Time, Burst Time, and Priority for each process:\n");

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

printf("Process %d - Arrival Time: ", i + 1);

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

printf("Process %d - Burst Time: ", i + 1);

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

printf("Process %d - Priority: ", i + 1);

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

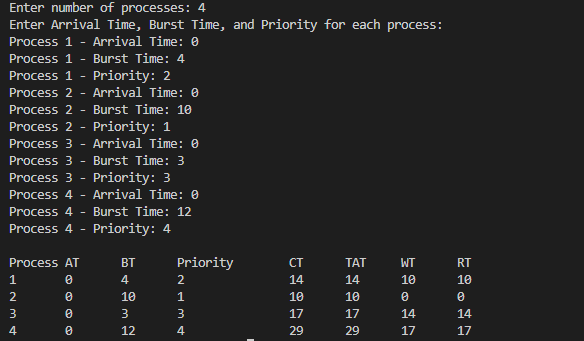
}

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

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

return 0;

}

Output:

=> Priority Scheduling (Preemptive):

Code

#include <stdio.h>

struct Process {

int id, arrivalTime, burstTime, remainingTime, priority;

int waitingTime, turnaroundTime, completionTime;

};

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

int highest = -1;

int highestPriority = 1e9;

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

if (p[i].arrivalTime <= currentTime && p[i].remainingTime > 0) {

if (p[i].priority < highestPriority) {

highestPriority = p[i].priority;

highest = i;

}

}

}

return highest;

}

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

int currentTime = 0, completed = 0;

float totalWaitingTime = 0, totalTurnaroundTime = 0;

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

p[i].remainingTime = p[i].burstTime;

}

while (completed < n) {

int idx = findHighestPriority(p, n, currentTime);

if (idx == -1) {

currentTime++;

continue;

}

p[idx].remainingTime--;

currentTime++;

if (p[idx].remainingTime == 0) {

completed++;

p[idx].completionTime = currentTime;

p[idx].turnaroundTime = p[idx].completionTime - p[idx].arrivalTime;

p[idx].waitingTime = p[idx].turnaroundTime - p[idx].burstTime;

totalWaitingTime += p[idx].waitingTime;

totalTurnaroundTime += p[idx].turnaroundTime;

}

}

printf("\nProcess\tArrival\tBurst\tPriority\tCompletion\tTurnaround\tWaiting\n");

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

printf("%d\t%d\t%d\t%d\t\t%d\t\t%d\t\t%d\n", p[i].id, p[i].arrivalTime, p[i].burstTime,

p[i].priority, p[i].completionTime, p[i].turnaroundTime, p[i].waitingTime);

}

printf("\nAverage Waiting Time: %.2f", totalWaitingTime / n);

printf("\nAverage Turnaround Time: %.2f\n", totalTurnaroundTime / n);

}

int main() {

int n;

printf("Enter number of processes: ");

scanf("%d",&n);

struct Process p[n];

printf("Enter Arrival Time, Burst Time, and Priority (lower number = higher priority) for each process:\n");

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

p[i].id = i + 1;

printf("Process %d: ", p[i].id);

scanf("%d %d %d", &p[i].arrivalTime, &p[i].burstTime, &p[i].priority);

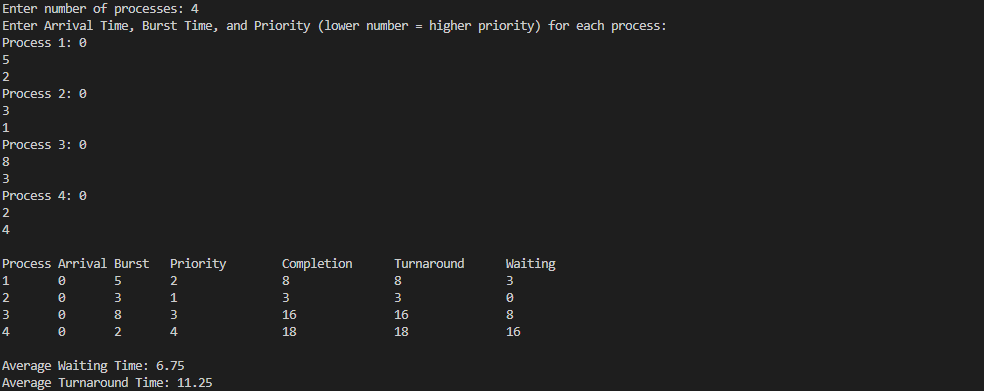
}

priorityScheduling(p, n);

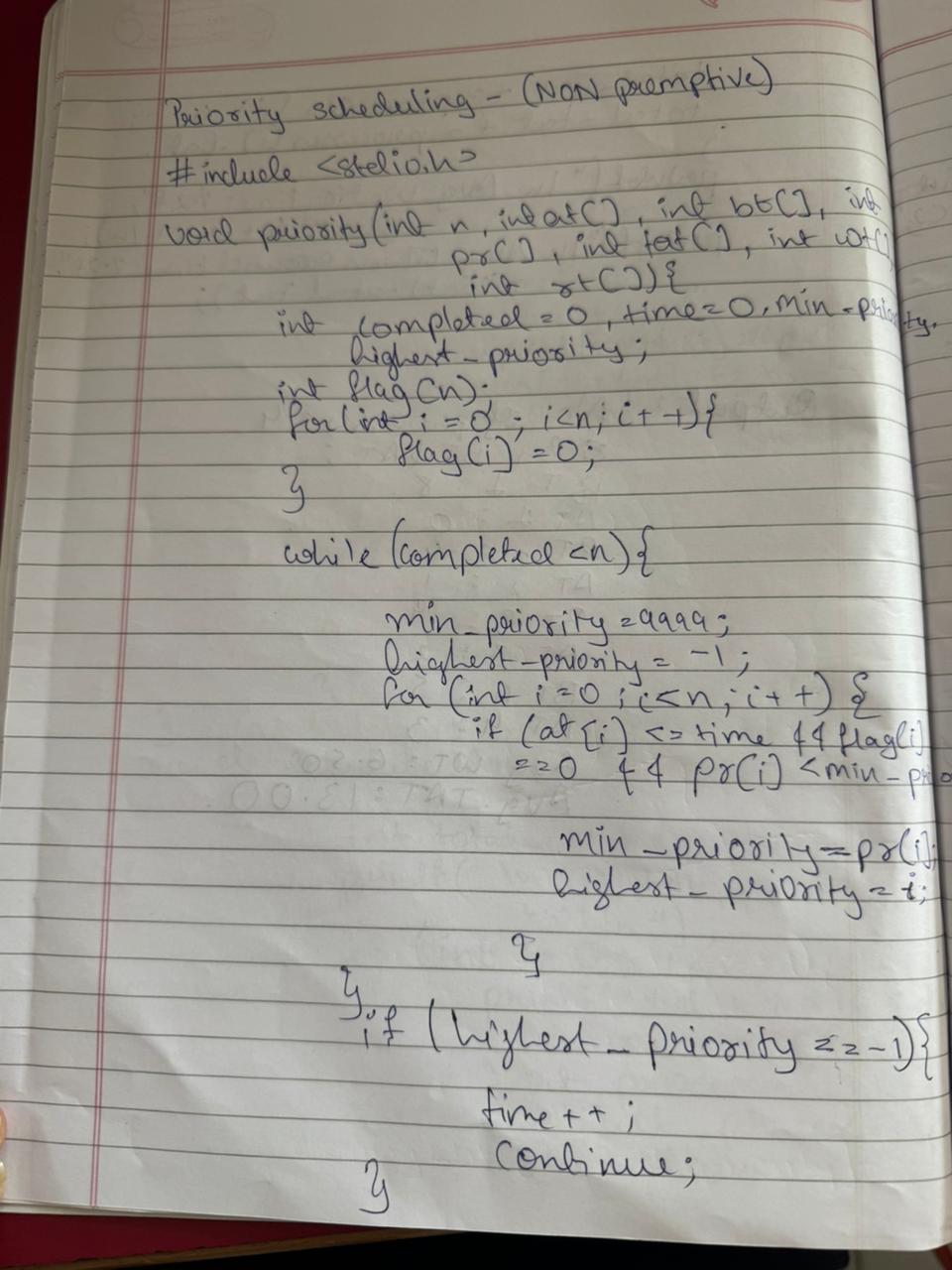
return 0;

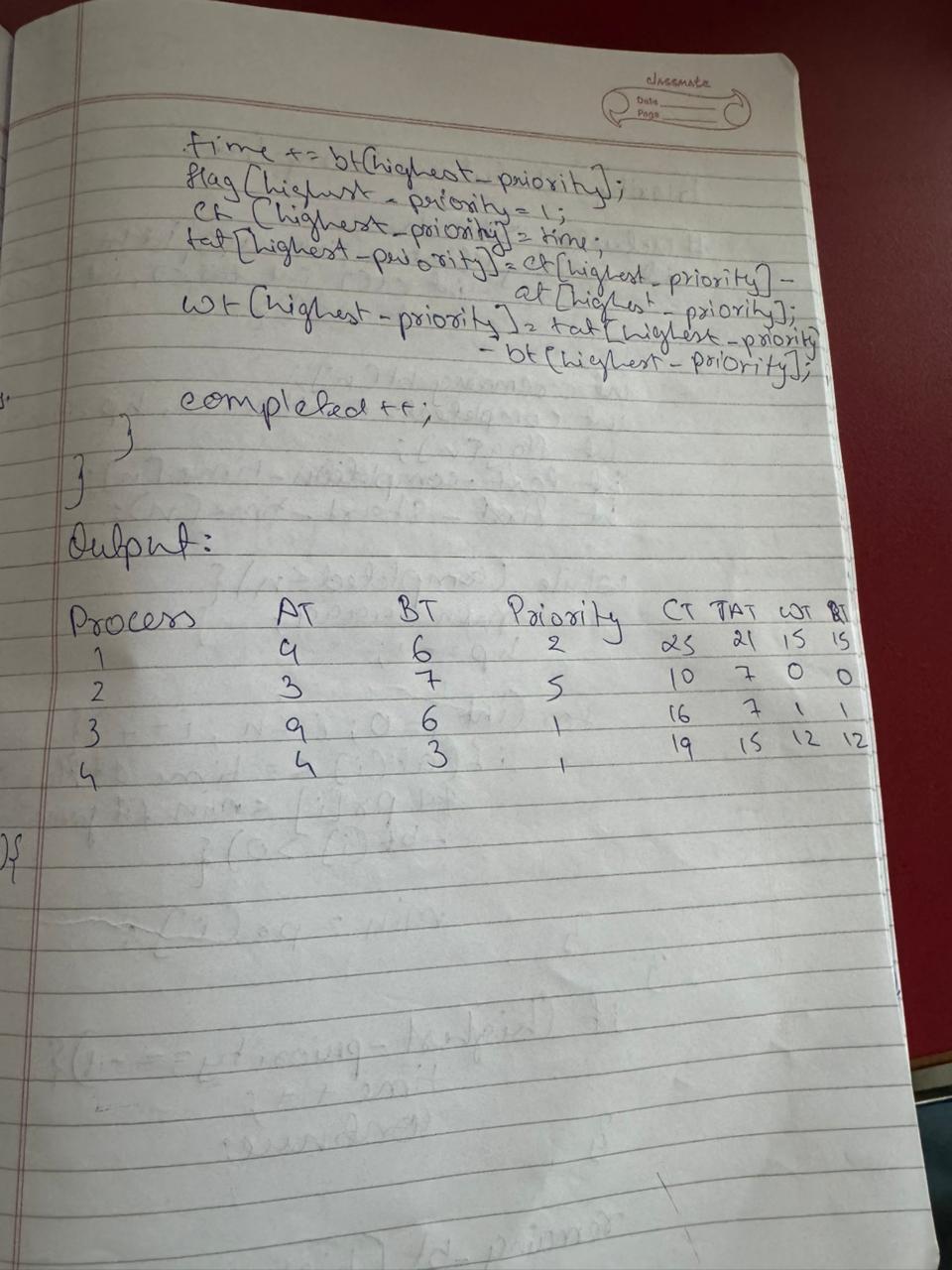
}

Output:



=> Round Robin:





#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[], int quantum) {

int rem\_bt[n];

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

rem\_bt[i] = bt[i];

wt[i] = 0;

wt++;

}

int t = 0;

while (1) {

int done = 1;

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

if (rem\_bt[i] > 0) {

done = 0;

if (rem\_bt[i] > quantum) {

rem\_bt[i] -= quantum;

//++quantum;

t += quantum;

} else {

t += rem\_bt[i];

wt[i] = t - bt[i];

rem\_bt[i] = 0;

}

}

}

if (done) break;

}

}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {

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

tat[i] = bt[i] + wt[i];

}

}

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

int wt[n], tat[n];

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

findTurnAroundTime(processes, n, bt, wt, tat);

int total\_wt = 0, total\_tat = 0;

printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

total\_wt += wt[i];

total\_tat += tat[i];

printf("%d\t%d\t\t%d\t\t%d\n", processes[i], bt[i], wt[i], tat[i]);

}

printf("\nAverage Waiting Time: %.2f", (float)total\_wt / n);

printf("\nAverage Turnaround Time: %.2f\n", (float)total\_tat / n);

}

int main() {

int n, quantum;

printf("Enter number of processes: ");

scanf("%d", &n);

int processes[n];

int burst\_time[n];

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

processes[i] = i + 1;

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

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

}

printf("Enter time quantum: ");

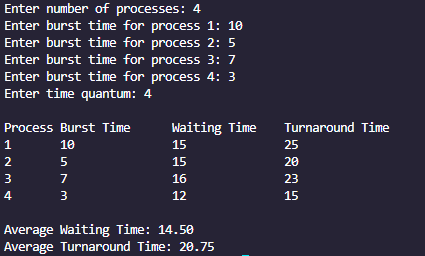
scanf("%d", &quantum);

findAvgTime(processes, n, burst\_time, quantum);

return 0;

}

Output:



Program 3

Question

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.

=> Multilevel queue Scheduling

Code

#include <stdio.h>

#define TIME\_QUANTUM 2

typedef struct {

int pid, burst\_time, arrival\_time, queue;

int waiting\_time, turnaround\_time, response\_time, remaining\_time;

} Process;

void sort\_by\_arrival(Process p[], int n) {

Process temp;

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

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

if (p[i].arrival\_time > p[j].arrival\_time) {

temp = p[i];

p[i] = p[j];

p[j] = temp;

}

}

}

}

void round\_robin(Process p[], int n, int \*time) {

int done, i;

do {

done = 1;

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

if (p[i].remaining\_time > 0) {

done = 0;

if (p[i].remaining\_time > TIME\_QUANTUM) {

\*time += TIME\_QUANTUM;

p[i].remaining\_time -= TIME\_QUANTUM;

} else {

\*time += p[i].remaining\_time;

p[i].waiting\_time = \*time - p[i].arrival\_time - p[i].burst\_time;

p[i].turnaround\_time = p[i].waiting\_time + p[i].burst\_time;

p[i].response\_time = p[i].waiting\_time;

p[i].remaining\_time = 0;

}

}

}

} while (!done);

}

void fcfs(Process p[], int n, int \*time) {

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

if (\*time < p[i].arrival\_time)

\*time = p[i].arrival\_time;

p[i].waiting\_time = \*time - p[i].arrival\_time;

p[i].turnaround\_time = p[i].waiting\_time + p[i].burst\_time;

p[i].response\_time = p[i].waiting\_time;

\*time += p[i].burst\_time;

}

}

int main() {

int n, i, time = 0;

printf("Enter number of processes: ");

scanf("%d", &n);

Process p[n], system\_processes[n], user\_processes[n];

int sys\_count = 0, user\_count = 0;

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

printf("Enter Burst Time, Arrival Time and Queue of P%d: ", i + 1);

p[i].pid = i + 1;

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

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

if (p[i].queue == 0)

system\_processes[sys\_count++] = p[i];

else

user\_processes[user\_count++] = p[i];

}

sort\_by\_arrival(system\_processes, sys\_count);

sort\_by\_arrival(user\_processes, user\_count);

printf("\nQueue 1 is System Process\nQueue 2 is User Process\n");

round\_robin(system\_processes, sys\_count, &time);

fcfs(user\_processes, user\_count, &time);

Process final\_list[n];

int index = 0;

for (i = 0; i < sys\_count; i++)

final\_list[index++] = system\_processes[i];

for (i = 0; i < user\_count; i++)

final\_list[index++] = user\_processes[i];

printf("\nProcess\tWaiting Time\tTurn Around Time\tResponse Time\n");

float avg\_wt = 0, avg\_tat = 0, avg\_rt = 0;

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

printf("%d\t%d\t\t%d\t\t\t%d\n", final\_list[i].pid, final\_list[i].waiting\_time, final\_list[i].turnaround\_time, final\_list[i].response\_time);

avg\_wt += final\_list[i].waiting\_time;

avg\_tat += final\_list[i].turnaround\_time;

avg\_rt += final\_list[i].response\_time;

}

avg\_wt /= n;

avg\_tat /= n;

avg\_rt /= n;

float throughput = (float)n / time;

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

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

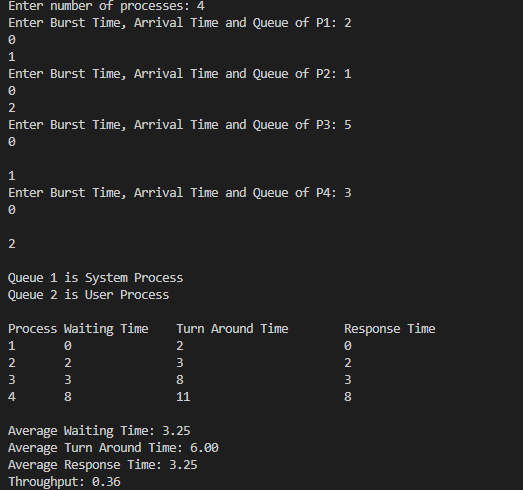
printf("\nAverage Response Time: %.2f", avg\_rt);

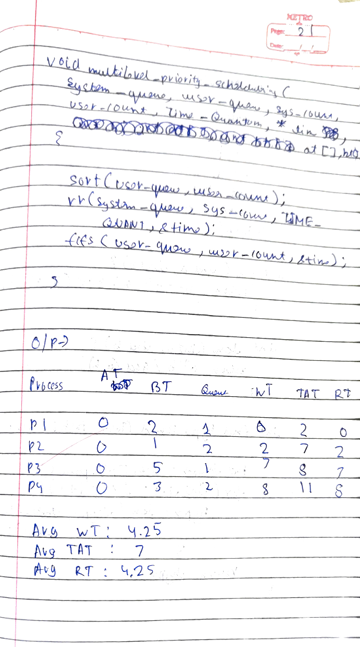
printf("\nThroughput: %.2f", throughput);

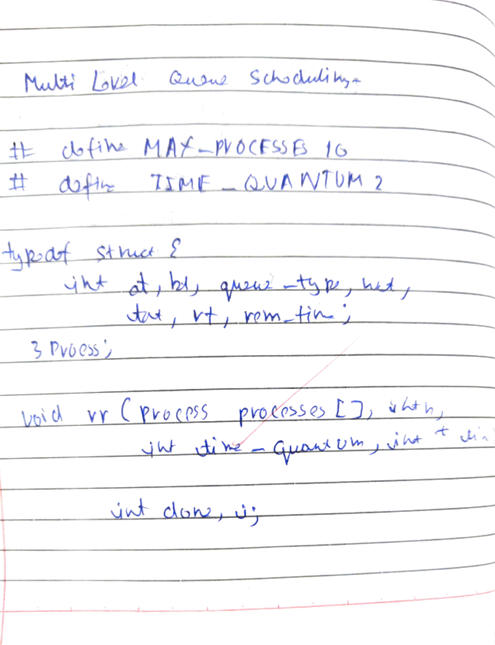
return 0;

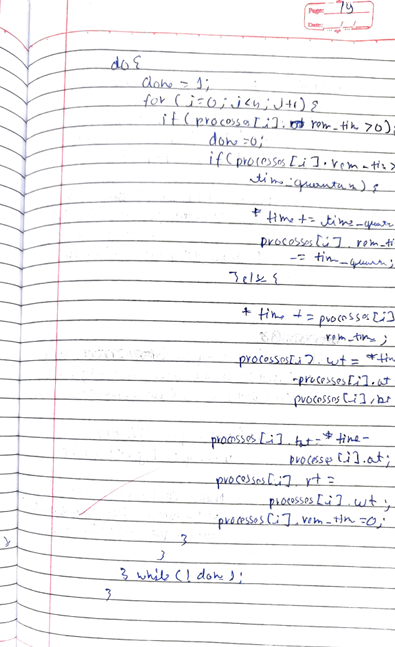
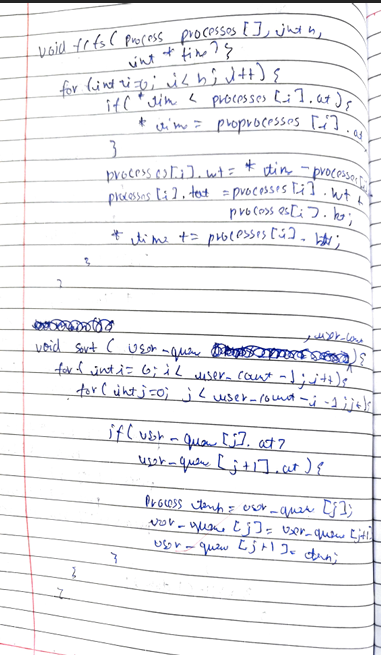
}

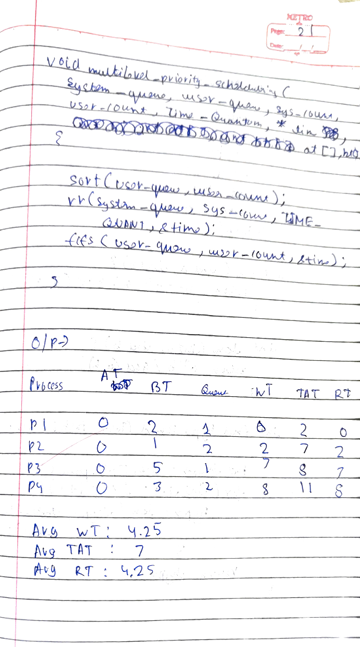
Output:









Program 4

Question

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

-> Rate- Monotonic

-> Earliest-deadline First

-> Proportional scheduling

=> Rate Monotonic Scheduling

Code

#include <stdio.h>

#include <stdlib.h>

typedef struct {

int id;

int period;

int execution\_time;

int next\_deadline;

int executed;

} Task;

int compare\_tasks(const void \*a, const void \*b) {

return ((Task \*)a)->period - ((Task \*)b)->period;

}

void rate\_monotonic\_scheduling(Task tasks[], int num\_tasks, int total\_time) {

qsort(tasks, num\_tasks, sizeof(Task), compare\_tasks);

/\*

for(int i = 0; i < num\_tasks; i++)

printf("Task %d: %d %d\n", tasks[i].id, tasks[i].execution\_time, tasks[i].period);

\*/

for (int i = 0; i < num\_tasks; i++)

tasks[i].next\_deadline = tasks[i].period;

printf("Time\t");

for (int i = 0; i < num\_tasks; i++)

printf("Task %d\t", tasks[i].id);

printf("\n");

for (int current\_time = 0; current\_time < total\_time; current\_time++)

{

printf("%d\t", current\_time);

int executed\_task = -1;

for (int i = 0; i < num\_tasks; i++)

{

if (current\_time % tasks[i].period == 0)

{

tasks[i].next\_deadline = current\_time + tasks[i].period;

tasks[i].executed = 0;

}

if (current\_time < tasks[i].next\_deadline)

{

if(tasks[i].executed < tasks[i].execution\_time)

{

executed\_task = i;

tasks[i].executed++;

break;

}

}

}

if (executed\_task != -1)

{

for (int i = 0; i < num\_tasks; i++)

{

if (i == executed\_task) {

printf("Exec\t");

} else {

printf("\t");

}

}

} else {

for (int i = 0; i < num\_tasks; i++) {

printf("\t");

}

}

printf("\n");

}

}

int main() {

Task tasks[] = {

{1, 20, 3},

{2, 5, 2},

{3, 10, 2}

};

int num\_tasks = sizeof(tasks) / sizeof(tasks[0]);

int total\_time = 20;

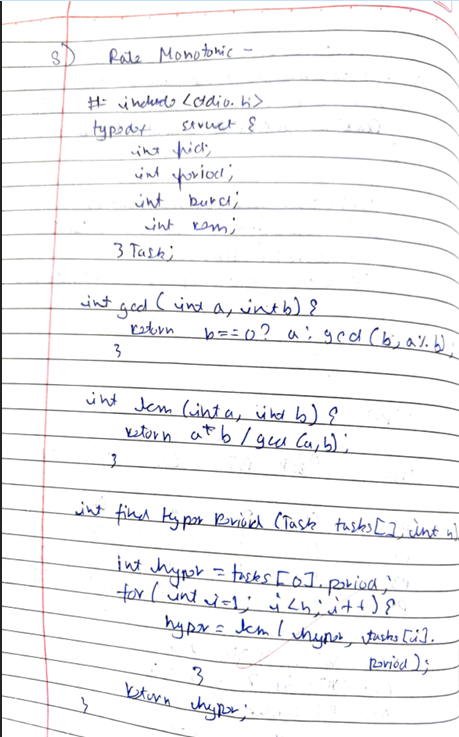
rate\_monotonic\_scheduling(tasks, num\_tasks, total\_time);

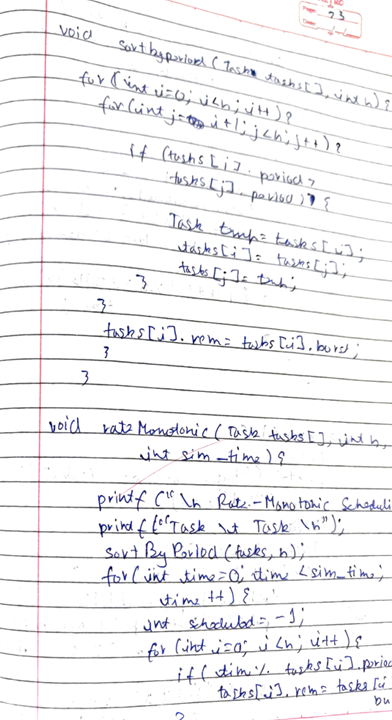
return 0;

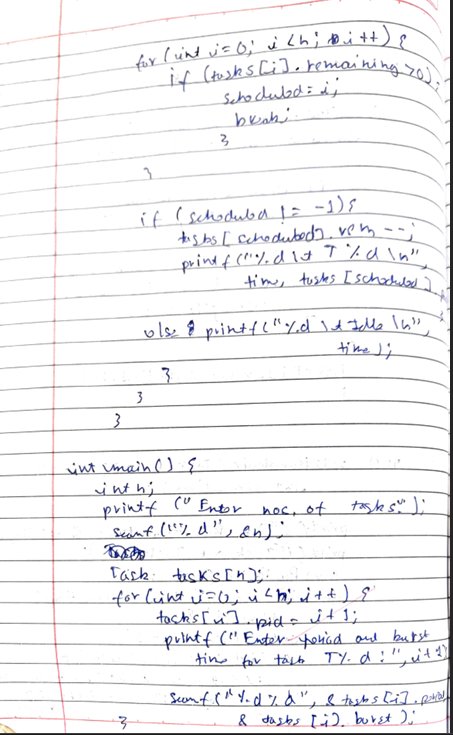
}

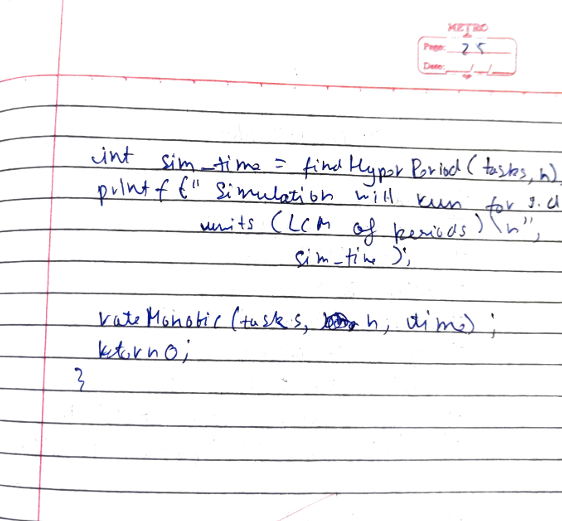
Output:











=> Earliest Deadline First

Code

#include <stdio.h>

#include <stdlib.h>

typedef struct {

int id;

int period;

int execution\_time;

int deadline;

int executed;

} Task;

int compare\_tasks(const void \*a, const void \*b) {

return ((Task \*)a)->deadline - ((Task \*)b)->deadline;

}

void earliest\_deadline\_first\_scheduling(Task tasks[], int num\_tasks, int total\_time) {

printf("Time\t");

for (int i = 0; i < num\_tasks; i++)

printf("Task %d\t", tasks[i].id);

printf("\n");

for (int current\_time = 0; current\_time < total\_time; current\_time++) {

printf("%d\t", current\_time);

int executed\_task = -1;

for (int i = 0; i < num\_tasks; i++) {

if (current\_time % tasks[i].period == 0) {

tasks[i].deadline = current\_time + tasks[i].period;

tasks[i].executed = 0;

}

}

qsort(tasks, num\_tasks, sizeof(Task), compare\_tasks);

for (int i = 0; i < num\_tasks; i++) {

if (current\_time < tasks[i].deadline && tasks[i].executed < tasks[i].execution\_time) {

executed\_task = i;

tasks[i].executed++;

break;

}

}

if (executed\_task != -1) {

for (int i = 0; i < num\_tasks; i++) {

if (i == executed\_task) {

printf("Exec\t");

} else {

printf("\t");

}

}

} else {

for (int i = 0; i < num\_tasks; i++) {

printf("\t");

}

}

printf("\n");

}

}

int main() {

Task tasks[] = {

{1, 20, 3, 20, 0},

{2, 5, 2, 5, 0},

{3, 10, 2, 10, 0} //task

};

int num\_tasks = sizeof(tasks) / sizeof(tasks[0]);

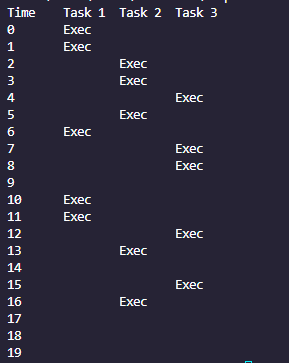
int total\_time = 20;

earliest\_deadline\_first\_scheduling(tasks, num\_tasks, total\_time);

return 0;

}

Output:



Program 5

Question

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

=> Producer Consumer

Code

#include <stdio.h>

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

void wait(int \*S)

{

(\*S)--;

}

void signal(int \*S)

{

(\*S)++;

}

void producer()

{

wait(&mutex);

if (empty > 0)

{

wait(&empty);

signal(&full);

printf("Item produced: %d\n", x++);

} else {

printf("Buffer is Full\n");

}

signal(&mutex);

}

void consumer() {

wait(&mutex);

if (full > 0) {

wait(&full);

signal(&empty);

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

} else {

printf("Buffer is Empty\n");

}

signal(&mutex);

}

int main() {

int ch;

printf("1. Produce\n2. Consume\n3. Exit\n");

while (1) {

printf("Enter Choice: ");

scanf("%d", &ch);

switch (ch) {

case 1: producer(); break;

case 2: consumer(); break;

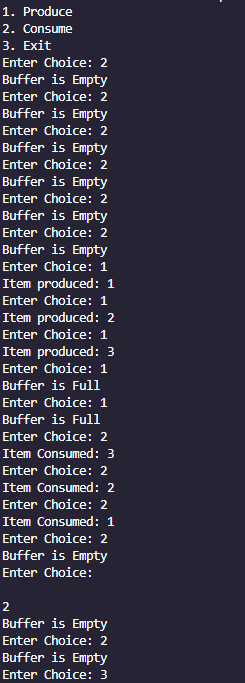
default: return 0;

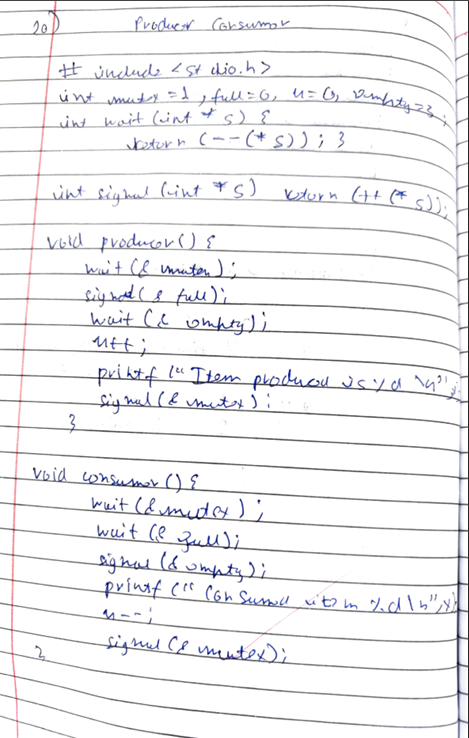
}

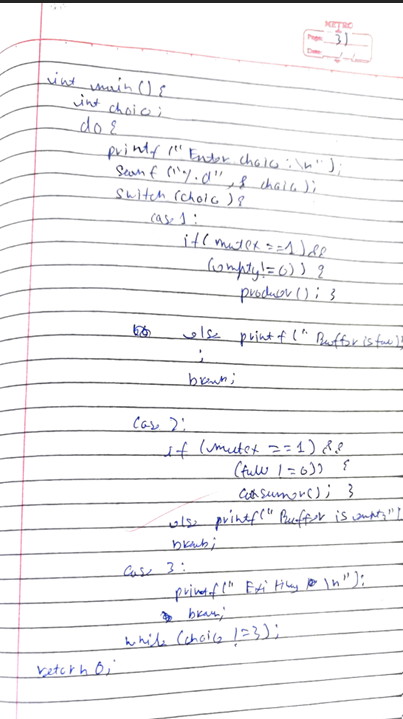
}

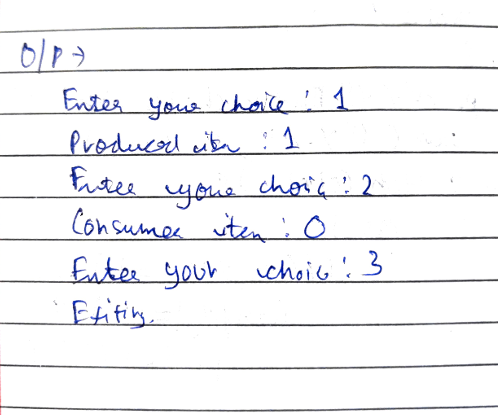
}

Output:









Program 6

Question

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

=> Dining Philosophers

Code

//PTHRED AND SEMAPHORE LIBRARY ONLY WORK IN CODEBLOCKS, NOT VSC

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <unistd.h>

#define N 5

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (phnum + 4) % N

#define RIGHT (phnum + 1) % N

int state[N];

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

sem\_t mutex;

sem\_t S[N];

void test(int phnum) {

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

state[phnum] = EATING;

sleep(2);

printf("Philosopher %d takes fork %d and %d\n", phnum + 1, LEFT + 1, phnum + 1);

printf("Philosopher %d is Eating\n", phnum + 1);

sem\_post(&S[phnum]);

}

}

void take\_fork(int phnum) {

sem\_wait(&mutex);

state[phnum] = HUNGRY;

printf("Philosopher %d is Hungry\n", phnum + 1);

test(phnum);

sem\_post(&mutex);

sem\_wait(&S[phnum]);

sleep(1);

}

void put\_fork(int phnum) {

sem\_wait(&mutex);

state[phnum] = THINKING;

printf("Philosopher %d putting fork %d and %d down\n", phnum + 1, LEFT + 1, phnum + 1);

printf("Philosopher %d is thinking\n", phnum + 1);

test(LEFT);

test(RIGHT);

sem\_post(&mutex);

}

void\* philosopher(void\* num) {

while (1) {

int\* i = (int\*)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 (i = 0; i < N; i++) {

sem\_init(&S[i], 0, 0);

}

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

pthread\_create(&thread\_id[i], NULL, philosopher, (void\*)&phil[i]);

printf("Philosopher %d is thinking\n", i + 1);

}

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

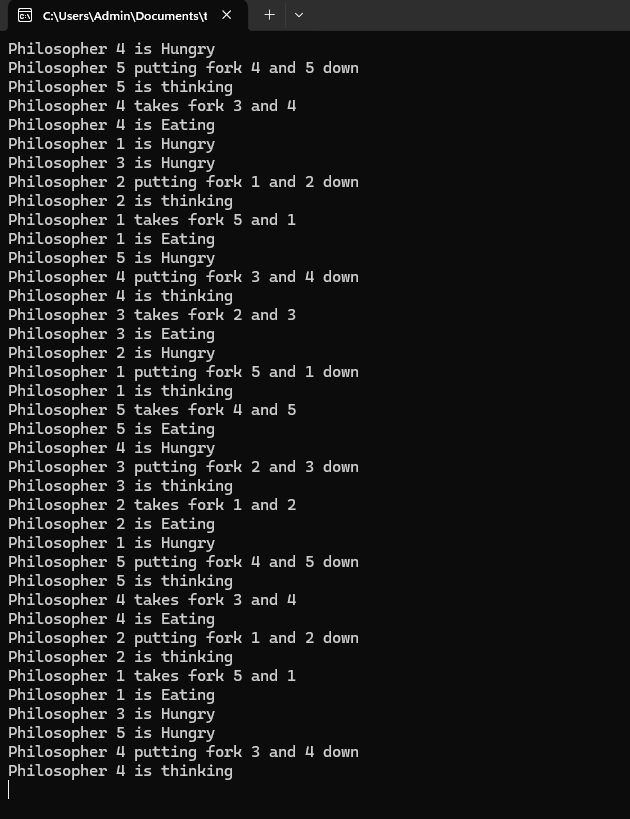
pthread\_join(thread\_id[i], NULL);

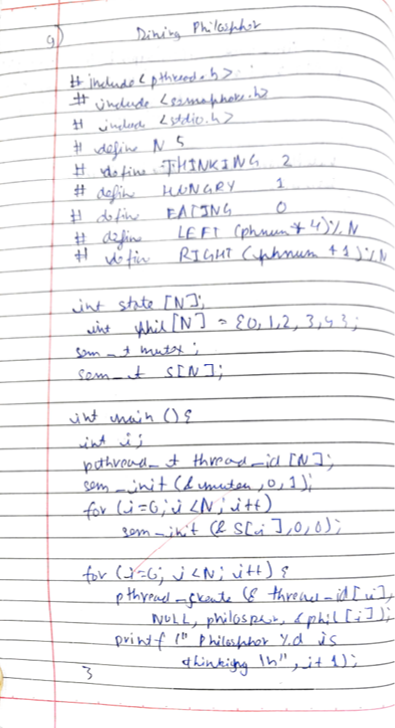
}

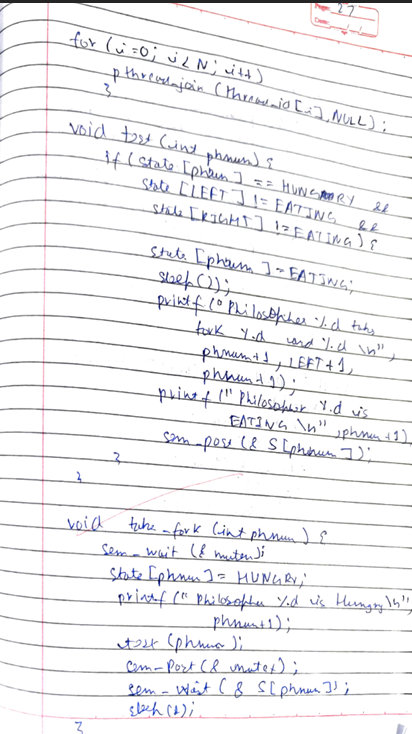
return 0;

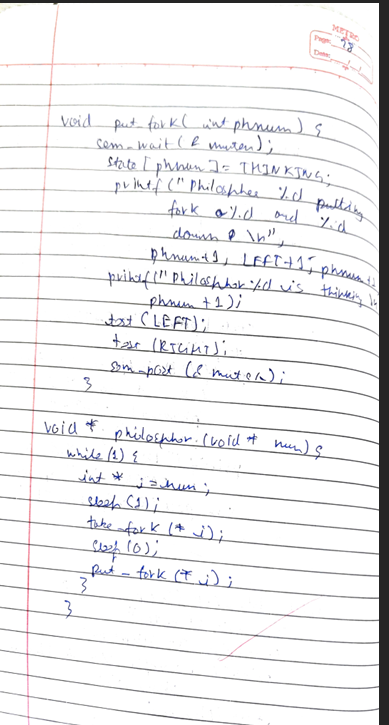
}

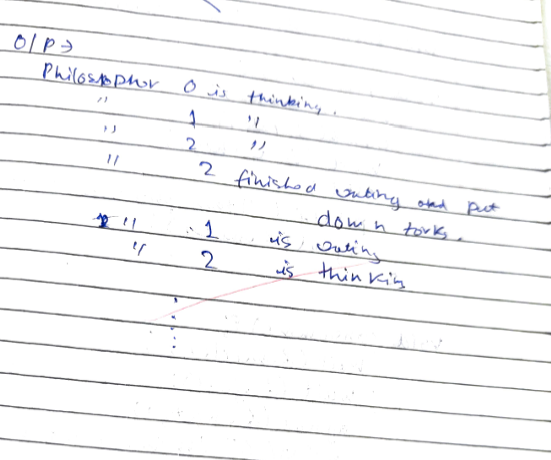
Output:











Program 7

Question

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

=> Banker’s Algorithm / Deadlock Avoidance

Code

#include <stdio.h>

#include <stdlib.h>

int condition(int \*\*need, int \*work, int i, int m)

{

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

{

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

return 0;

}

return 1;

}

int safety(int m, int n, int \*\*allocated, int \*\*max, int \*available, int \*sequence)

{

// Need Matrix

int \*\*need = (int\*\*) malloc(n \* sizeof(int\*));

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

{

need[i] = (int\*) malloc(m \* sizeof(int));

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

{

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

}

}

// Work array

int \*work = (int\*) malloc(m \* sizeof(int));

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

{

work[i] = available[i];

}

// Finish array

int \*finish = (int\*) malloc(n \* sizeof(int));

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

{

finish[i] = 0;

}

int safeIndex = 0;

int changed;

do {

changed = 0;

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

{

if (!finish[i] && condition(need, work, i, m))

{

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

{

work[j] += allocated[i][j];

}

finish[i] = 1;

sequence[safeIndex++] = i;

changed = 1;

}

}

} while (changed);

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

{

if (!finish[i])

{

return 0;

}

}

return 1;

}

int main()

{

int n, m;

printf("Enter number of processes and resources (n x m order): ");

scanf("%d",&n);

scanf("%d",&m);

// Allocation Matrix

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

int \*\*allocated = (int \*\*) malloc(n \* sizeof(int\*));

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

{

allocated[i] = (int\*) malloc(m \* sizeof(int));

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

{

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

}

}

// Max Matrix

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

int \*\*max = (int \*\*) malloc(n \* sizeof(int\*));

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

{

max[i] = (int\*) malloc(m \* sizeof(int));

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

{

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

}

}

// Available Matrix

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

int \*available = (int \*) malloc(m \* sizeof(int));

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

{

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

}

// Sequence Matrix

int \*sequence = (int \*) malloc(n \* sizeof(int));

int safe = safety(m, n, allocated, max, available, sequence);

if (safe)

{

printf("System is in a Safe State.\nSafe Sequence: ");

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

{

printf("P%d\t", sequence[i]);

}

printf("\n");

}

else

{

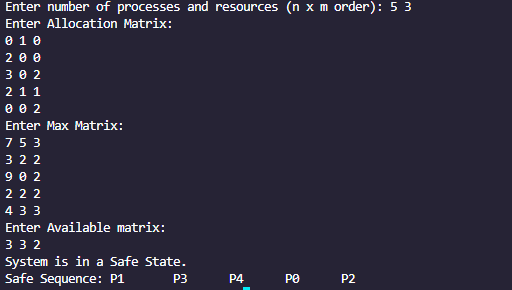
printf("System is not in a Safe State.\n");

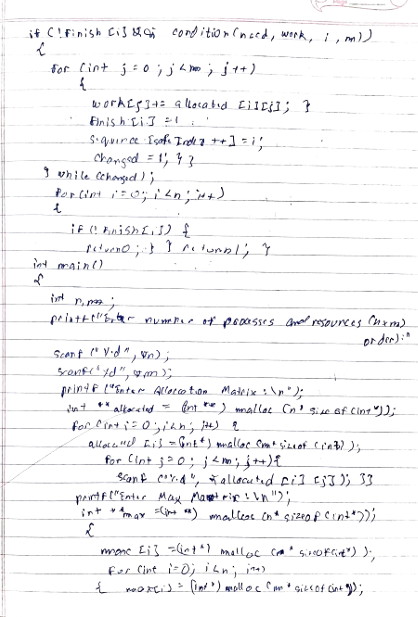
}

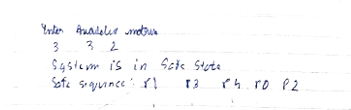
return 0;

}

Output:







Program 8

Question

Write a C program to simulate deadlock detection

=> Deadlock Detection

Code

#include <stdio.h>

#include <stdbool.h>

#define P 5

#define R 3

int main() {

int finish[P] = {0};

int work[R];

int need[P][R] = {

{7, 5, 3},

{3, 2, 2},

{9, 0, 2},

{2, 2, 2},

{4, 3, 3}

};

int allocation[P][R] = {

{0, 1, 0},

{2, 0, 0},

{3, 0, 2},

{2, 1, 1},

{0, 0, 2}

};

int available[R] = {3, 3, 2};

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

work[i] = available[i];

}

bool deadlock = false;

int count = 0;

while (count < P) {

bool found = false;

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

if (finish[p] == 0) {

bool canFinish = true;

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

if (need[p][r] - allocation[p][r] > work[r]) {

canFinish = false;

break;

}

}

if (canFinish) {

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

work[r] += allocation[p][r];

}

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

finish[p] = 1;

found = true;

count++;

}

}

}

if (!found) {

deadlock = true;

break;

}

}

if (deadlock) {

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

} else {

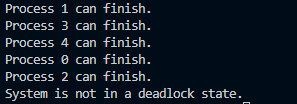
printf("System is not in a deadlock state.\n");

}

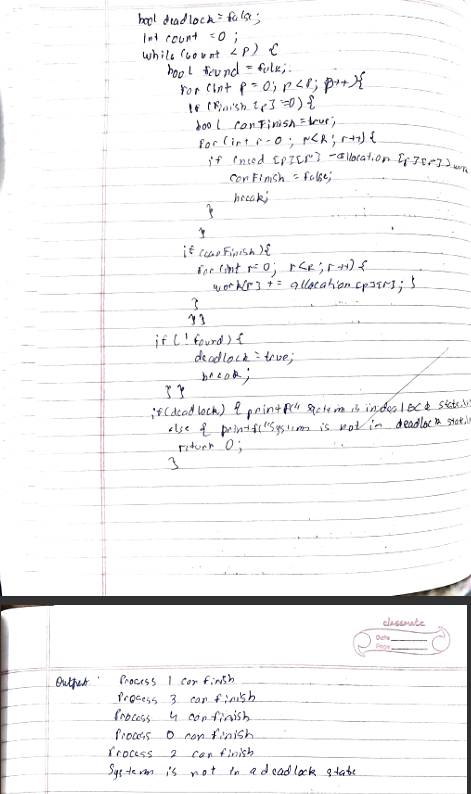
return 0;

}

Output:







Program 9

Question

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

1. Best-fit
2. First-fit

=> Best fit, worst fit, first fit

Code

#include <stdio.h>

struct Block {

int block\_no;

int block\_size;

int is\_free;

};

struct File {

int file\_no;

int file\_size;

};

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

printf("Memory 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;

int min\_fragment = 10000; // Initialize with a large value

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

if (blocks[j].is\_free && blocks[j].block\_size >= files[i].file\_size) {

int fragment = blocks[j].block\_size - files[i].file\_size;

if (fragment < min\_fragment) {

min\_fragment = fragment;

best\_fit\_block = j;

}

}

}

if (best\_fit\_block != -1) {

blocks[best\_fit\_block].is\_free = 0;

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", files[i].file\_no, files[i].file\_size,

blocks[best\_fit\_block].block\_no, blocks[best\_fit\_block].block\_size, min\_fragment);

}

}

}

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

printf("Memory Management Scheme - First Fit\n");

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

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

int found = 0;

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

if (blocks[j].is\_free && blocks[j].block\_size >= files[i].file\_size) {

blocks[j].is\_free = 0;

int fragment = blocks[j].block\_size - files[i].file\_size;

printf("%d\t\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);

found = 1;

break;

}

}

if (!found) {

printf("No suitable block found for File %d\n", files[i].file\_no);

}

}

}

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

printf("Memory 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;

int max\_fragment = -1; // Initialize with a small value

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\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);

}

}

}

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];

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;

}

struct File files[n\_files];

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);

}

while(1) {

int choice;

printf("Choose Memory Management Scheme:\n");

printf("1. Best Fit\n");

printf("2. First Fit\n");

printf("3. Worst Fit\n");

printf("[ANY KEY]. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

// Reset blocks for allocation scheme

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

blocks[i].is\_free = 1;

}

switch (choice) {

case 1:

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

break;

case 2:

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

break;

case 3:

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

break;

default:

printf("Closing...");

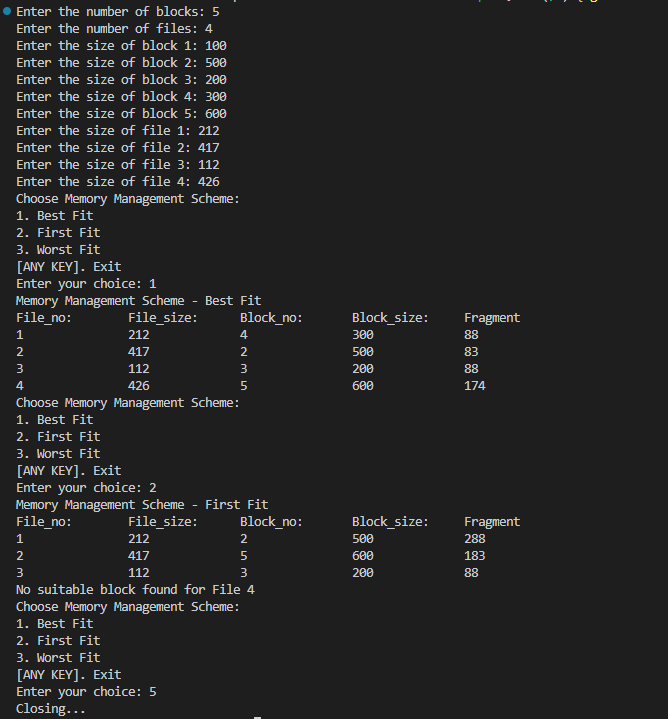
return 0;

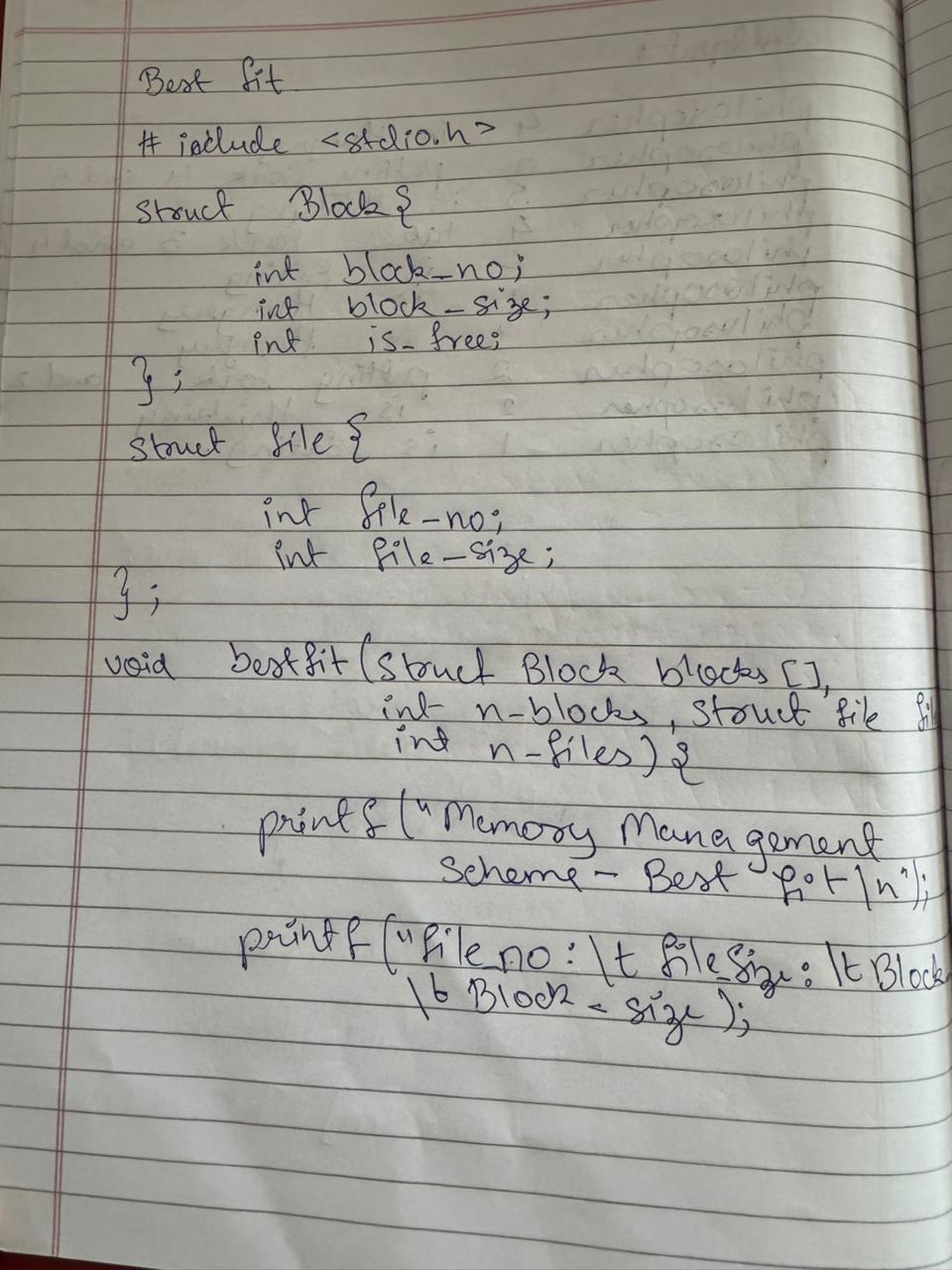
} }

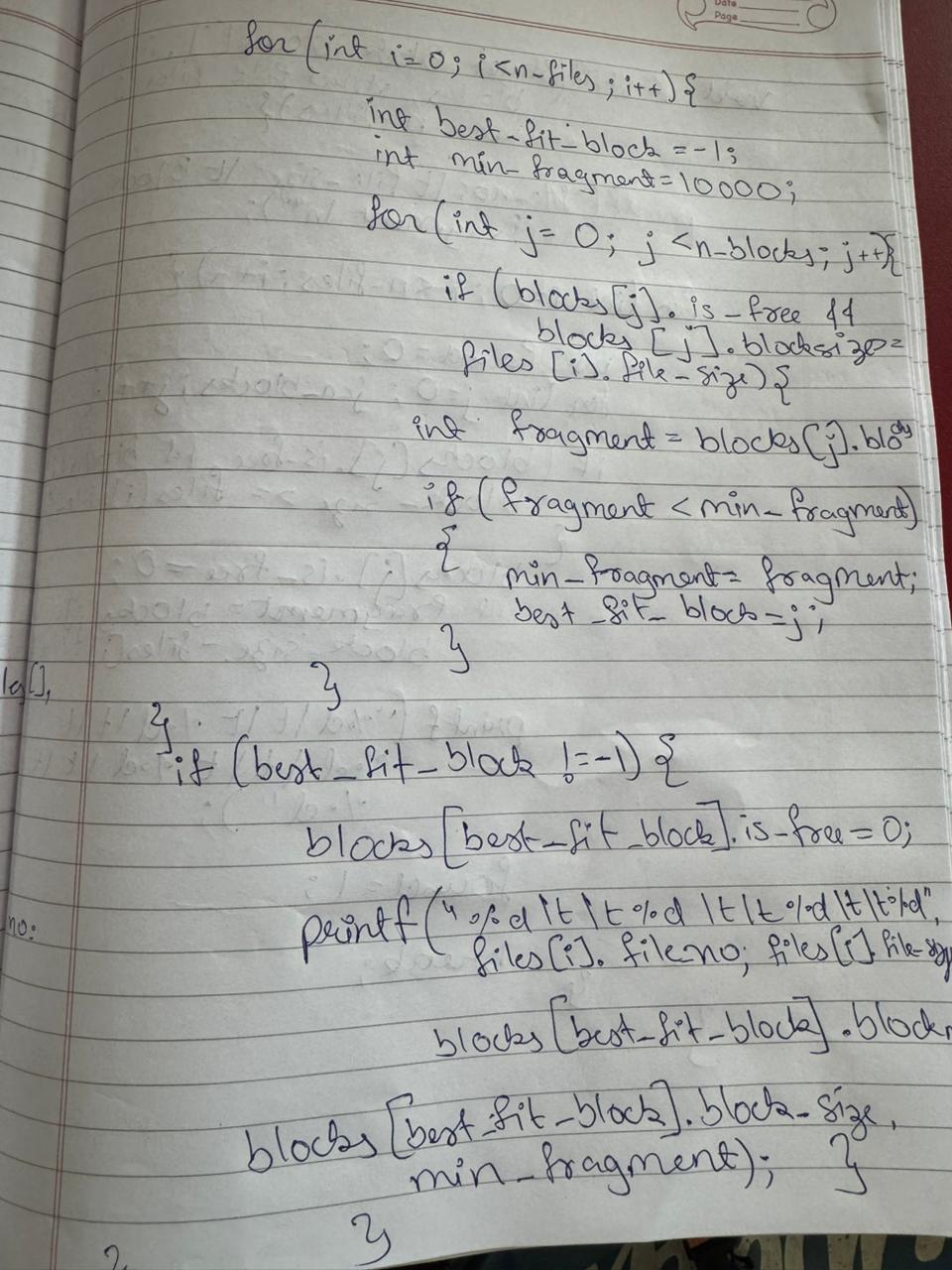
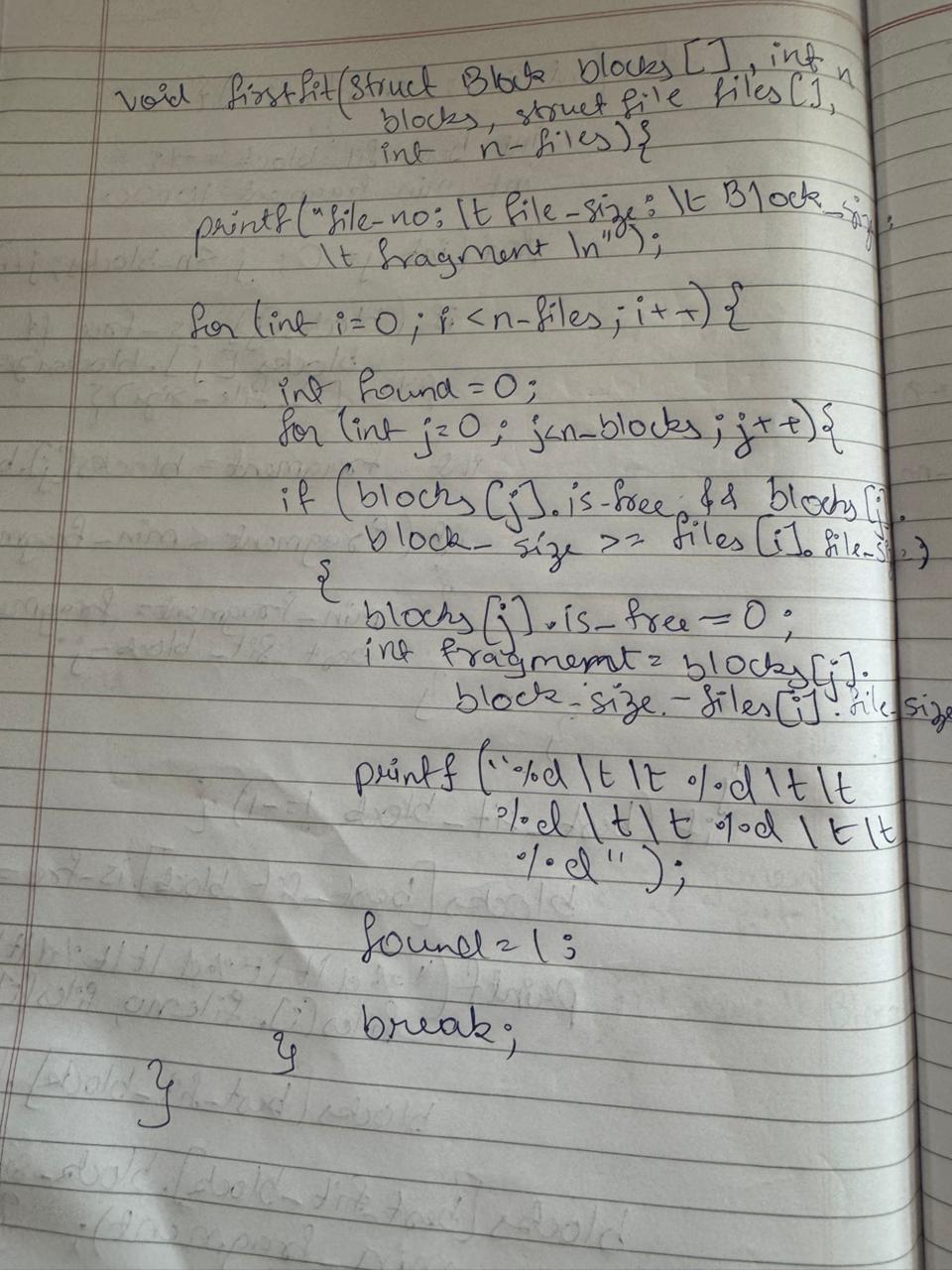
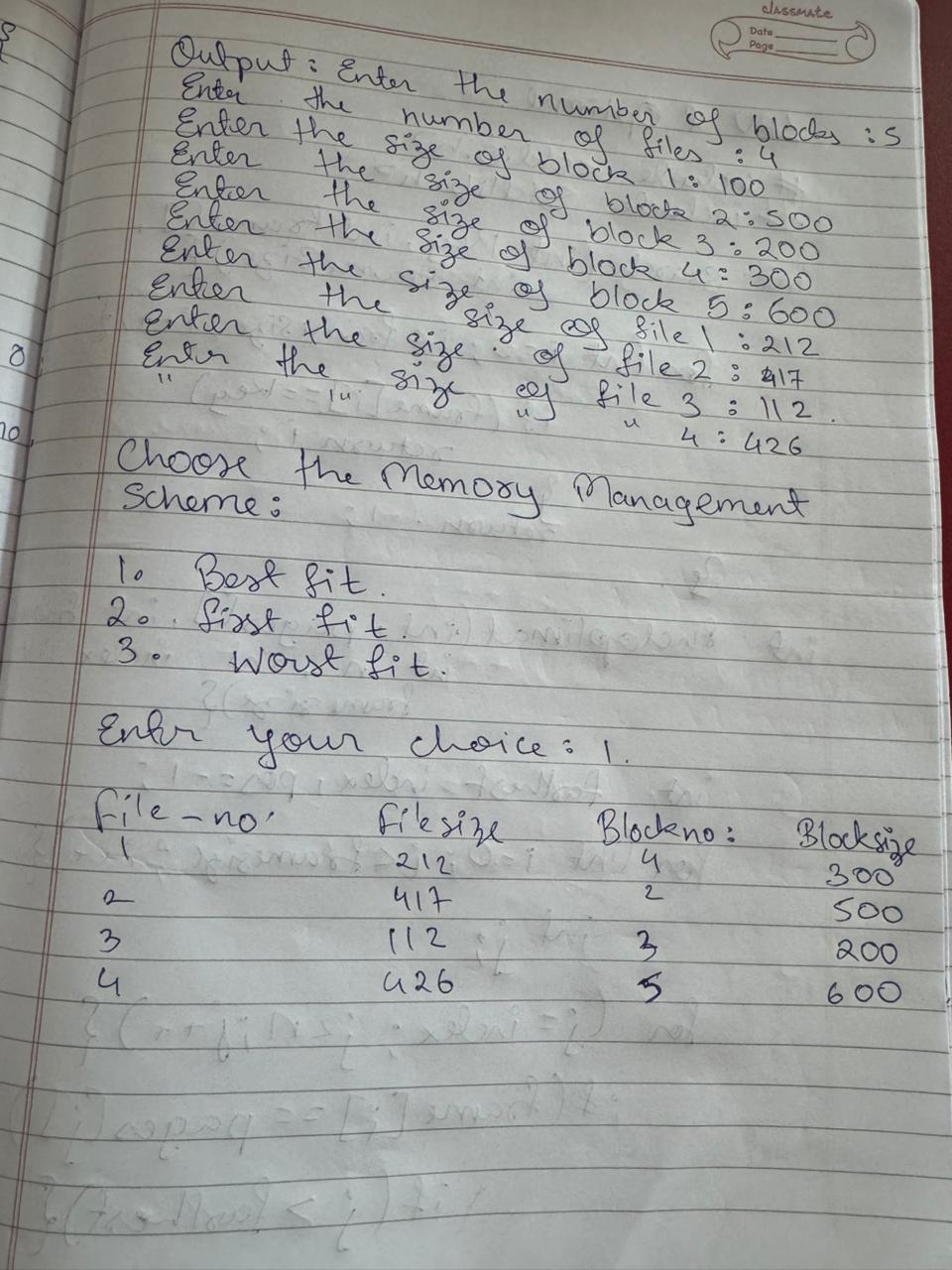
return 0;

}

Output:





Program 10

Question

Write a C program to simulate page replacement algorithms a) FIFO

1. LRU
2. Optimal

=> LRU & Optimal

Code

#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);

return 0;

}

Output:

