DAY 4 PROGRAMS

Name:-Omkar

Reg no:-192111134

PROGRAMS

21. Write a program to simulate the Dining Philosopher problem and verify your output with the following test case:

No of Philosophers: 5

THINKING – When philosopher doesn’t want to gain access to either fork.

HUNGRY – When philosopher wants to enter the critical section.

EATING – When philosopher has got both the forks, i.e., he has entered the section.

Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating

(state[(i+4) % 5] != EATING) and (state[(i+1) % 5] != EATING).

Program:-

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#define N 5

pthread\_mutex\_t chopsticks[N];

void \*philosopher(void \*arg) {

int id = \*(int \*)arg;

int left = id;

int right = (id + 1) % N;

while (1) {

printf("Philosopher %d is thinking\n", id);

usleep(rand() % 1000000);

printf("Philosopher %d is hungry\n", id);

pthread\_mutex\_lock(&chopsticks[left]);

pthread\_mutex\_lock(&chopsticks[right]);

printf("Philosopher %d is eating\n", id);

usleep(rand() % 1000000);

pthread\_mutex\_unlock(&chopsticks[left]);

pthread\_mutex\_unlock(&chopsticks[right]);

}

}

int main() {

pthread\_t threads[N];

int ids[N];

int i;

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

pthread\_mutex\_init(&chopsticks[i], NULL);

}

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

ids[i] = i;

pthread\_create(&threads[i], NULL, philosopher, &ids[i]);

}

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

pthread\_join(threads[i], NULL);

}

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

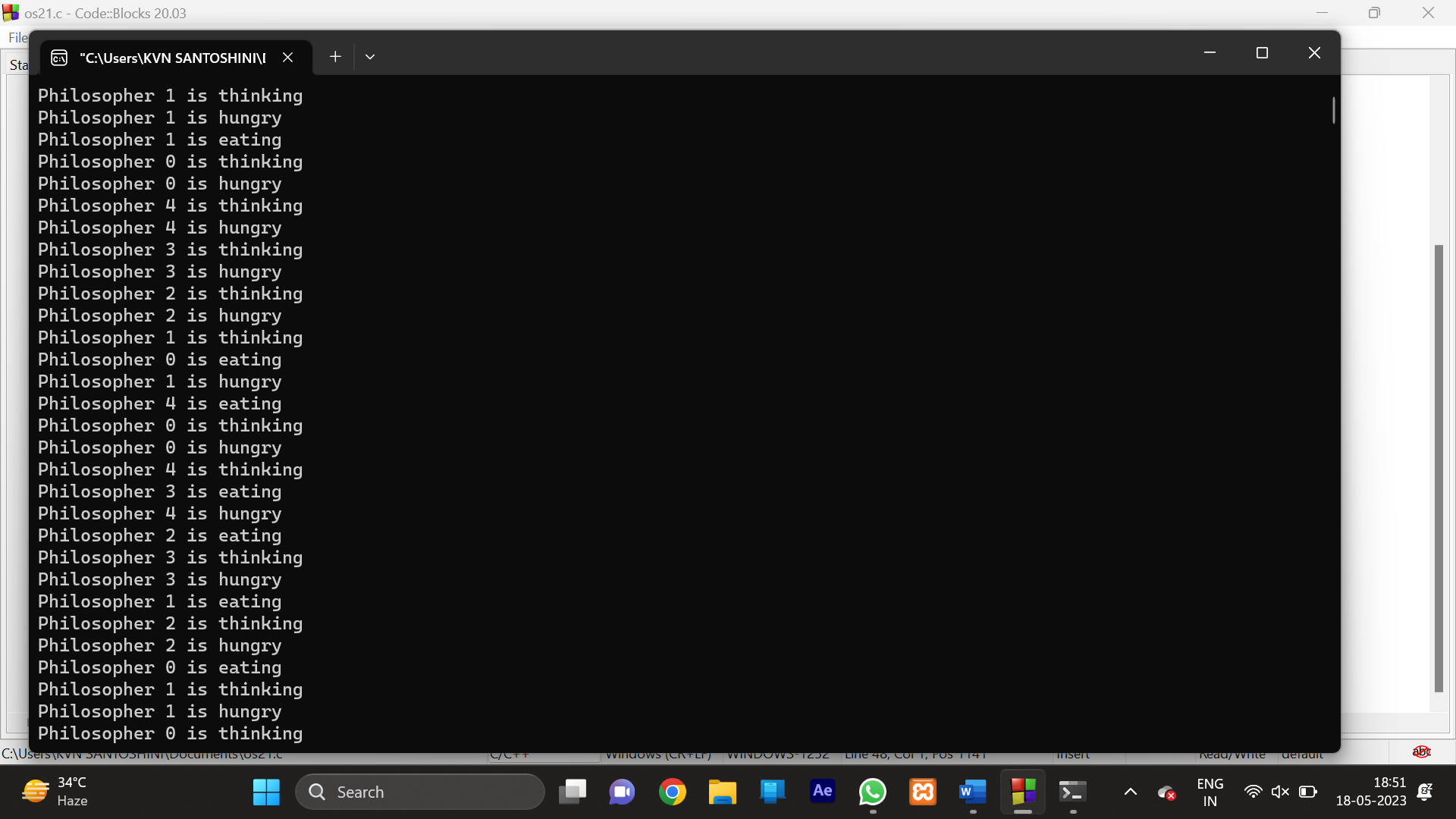
pthread\_mutex\_destroy(&chopsticks[i]);

}

return 0;

}

OUTPUT



22. Write a C program to implement the two-level directory system.

Test Case:

3 user directories have to be created with name of user1, user2, user3 and also to create 3 files with user1 directory,2 files with user2 and user3 directory

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_USERS 100

#define MAX\_FILES 100

typedef struct {

char name[20];

int id;

int file\_count;

int file\_ids[MAX\_FILES];

} User;

User users[MAX\_USERS];

int user\_count = 0;

typedef struct {

char name[20];

int id;

int user\_id;

} File;

File files[MAX\_FILES];

int file\_count = 0;

int get\_user\_index(char \*name) {

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

if (strcmp(users[i].name, name) == 0) {

return i;

}

}

return -1;

}

int get\_file\_index(char \*name) {

for (int i = 0; i < file\_count; i++) {

if (strcmp(files[i].name, name) == 0) {

return i;

}

}

return -1;

}

void create\_user(char \*name) {

int index = get\_user\_index(name);

if (index == -1) {

strcpy(users[user\_count].name, name);

users[user\_count].id = user\_count;

users[user\_count].file\_count = 0;

user\_count++;

printf("User '%s' created.\n", name);

} else {

printf("User '%s' already exists.\n", name);

}

}

void create\_file(char \*name, char \*user\_name) {

int user\_index = get\_user\_index(user\_name);

if (user\_index == -1) {

printf("User '%s' not found.\n", user\_name);

return;

}

int file\_index = get\_file\_index(name);

if (file\_index != -1) {

printf("File '%s' already exists.\n", name);

return;

}

int file\_id = file\_count;

strcpy(files[file\_id].name, name);

files[file\_id].id = file\_id;

files[file\_id].user\_id = user\_index;

users[user\_index].file\_ids[users[user\_index].file\_count] = file\_id;

users[user\_index].file\_count++;

file\_count++;

printf("File '%s' created for user '%s'.\n", name, user\_name);

}

void list\_files(char \*user\_name) {

int user\_index = get\_user\_index(user\_name);

if (user\_index == -1) {

printf("User '%s' not found.\n", user\_name);

return;

}

printf("Files for user '%s':\n", user\_name);

for (int i = 0; i < users[user\_index].file\_count; i++) {

int file\_index = users[user\_index].file\_ids[i];

printf("- %s\n", files[file\_index].name);

}

}

int main() {

create\_user("Alice");

create\_user("Bob");

create\_file("report.txt", "Alice");

create\_file("notes.txt", "Alice");

create\_file("schedule.txt", "Bob");

list\_files("Alice");

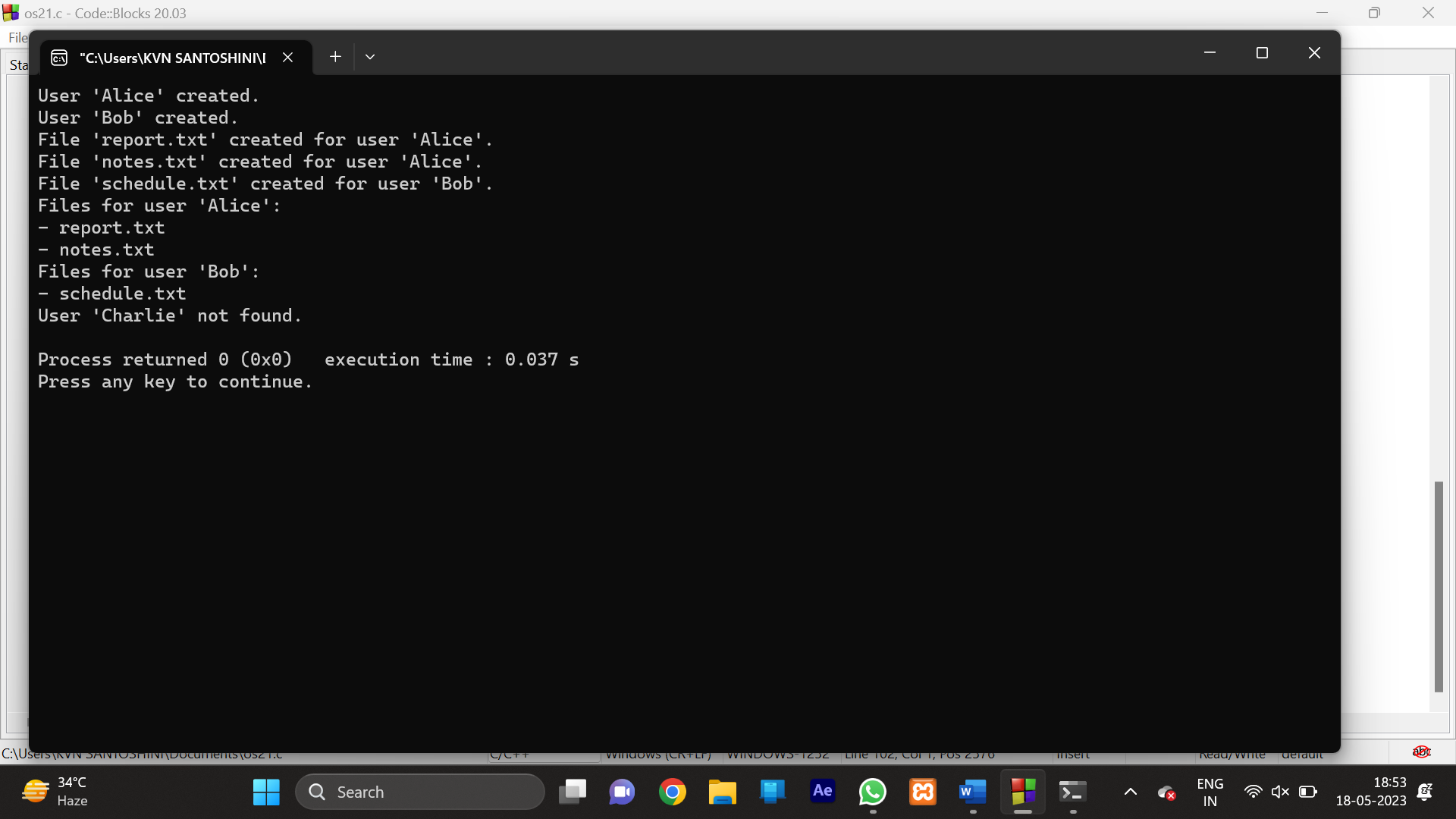
list\_files("Bob");

list\_files("Charlie");

return 0;

}

OUTPUT



23. Write a C program to simulate SCAN disk scheduling algorithms. and execute your program and find out and print the average head movement for the following test case.

No of tracks:5; Track position:55 58 60 70 18

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#define MAX\_REQUESTS 100

int requests[MAX\_REQUESTS];

int request\_count;

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

return \*(int \*)a - \*(int \*)b;

}

void scan(int start) {

int total\_distance = 0;

int current = start;

int direction = 1;

qsort(requests, request\_count, sizeof(int), compare);

int i = 0;

while (i < request\_count && requests[i] < start) {

i++;

}

while (i < request\_count) {

int next = requests[i];

total\_distance += abs(next - current);

current = next;

i++;

}

total\_distance += abs(current - (direction ? 0 : 99));

current = (direction ? 0 : 99);

i--;

direction = !direction;

while (i >= 0) {

int next = requests[i];

total\_distance += abs(next - current);

current = next;

i--;

}

printf("Average head movement: %.2f\n", (float)total\_distance / request\_count);

}

int main() {

requests[0] = 50;

requests[1] = 82;

requests[2] = 170;

requests[3] = 43;

requests[4] = 140;

requests[5] = 24;

requests[6] = 16;

requests[7] = 190;

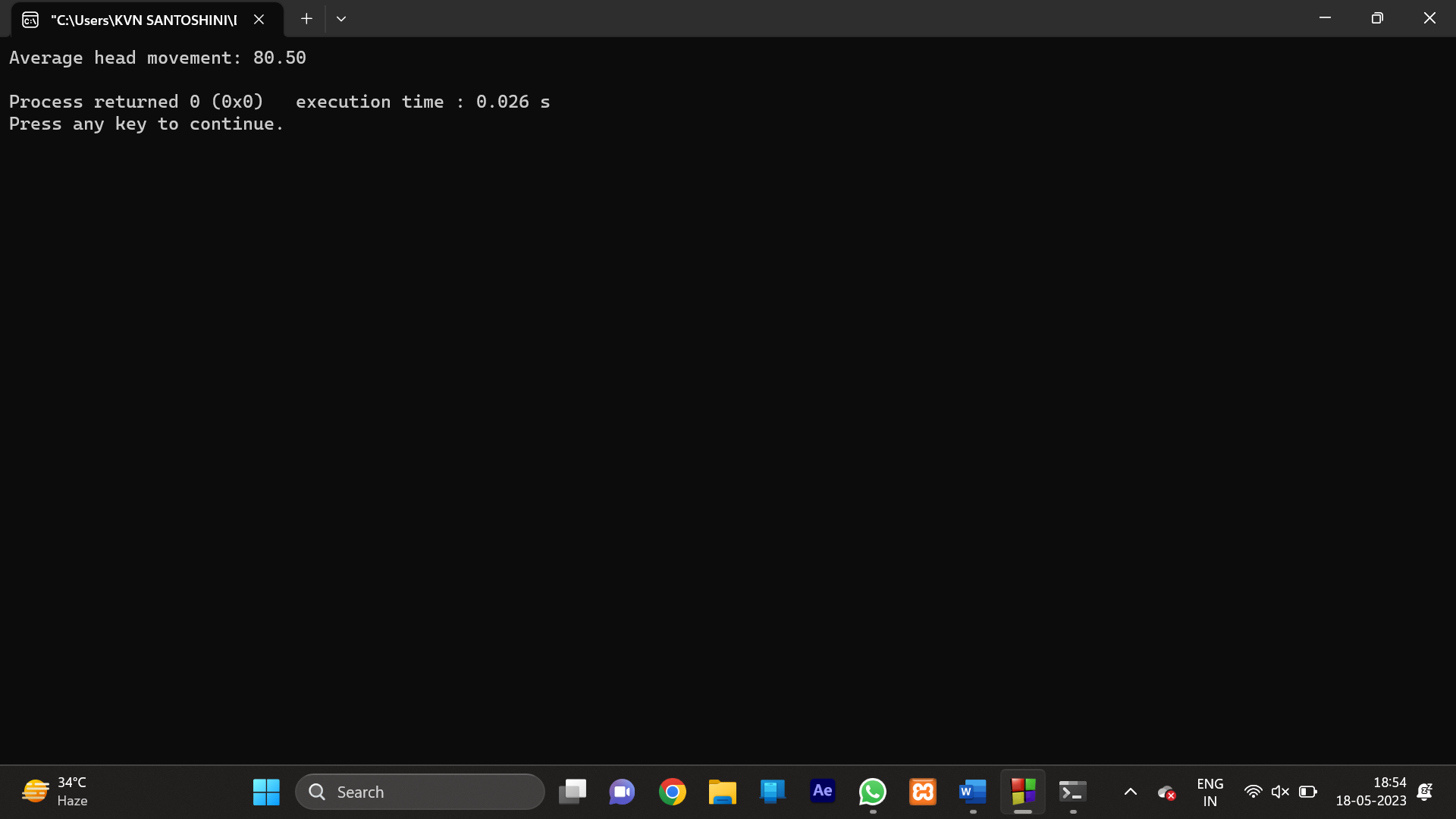
request\_count = 8;

scan(100);

return 0;

}

OUTPUT



24. Write a C Program to find the maximum size of a file that can be stored in the below file system that uses inodes to represent files. Disk blocks are 8 KB in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks.

Test Case:

● Check that the start blocks and the required contiguous blocks are free.

● If free, allocate those blocks to the file.

● If not free, find the next available contiguous blocks.

PROGRAM

#include <stdio.h>

#define BLOCK\_SIZE 8192

#define POINTERS\_PER\_BLOCK (BLOCK\_SIZE / sizeof(int))

#define MAX\_BLOCKS\_DIRECT 12

#define MAX\_BLOCKS\_SINGLE\_INDIRECT (POINTERS\_PER\_BLOCK + MAX\_BLOCKS\_DIRECT)

#define MAX\_BLOCKS\_DOUBLE\_INDIRECT (POINTERS\_PER\_BLOCK \* POINTERS\_PER\_BLOCK + MAX\_BLOCKS\_SINGLE\_INDIRECT)

#define MAX\_BLOCKS\_TRIPLE\_INDIRECT (POINTERS\_PER\_BLOCK \* POINTERS\_PER\_BLOCK \* POINTERS\_PER\_BLOCK + MAX\_BLOCKS\_DOUBLE\_INDIRECT)

int main() {

unsigned long long max\_file\_size = MAX\_BLOCKS\_DIRECT \* BLOCK\_SIZE

+ MAX\_BLOCKS\_SINGLE\_INDIRECT \* BLOCK\_SIZE

+ MAX\_BLOCKS\_DOUBLE\_INDIRECT \* BLOCK\_SIZE

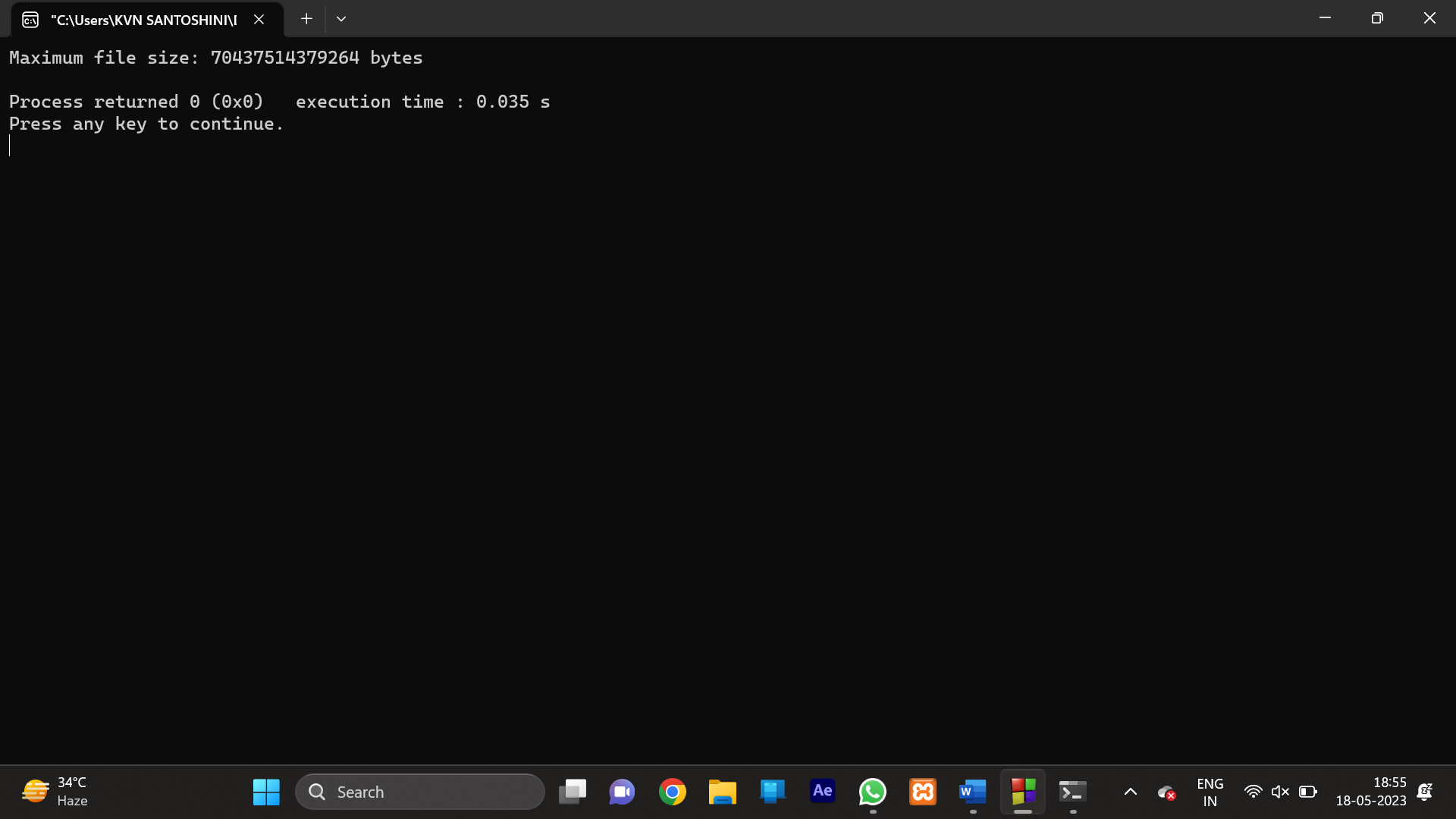
+ MAX\_BLOCKS\_TRIPLE\_INDIRECT \* BLOCK\_SIZE;

printf("Maximum file size: %llu bytes\n", max\_file\_size);

return 0;

}

OUTPUT



25. Write the C program to Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold in a file currently consisting of 100 blocks. Assume that the file control block (and the index block, in the case of indexed allocation) is already in memory.

Test Cases:

a. The block is added at the beginning.

b. The block is added in the middle.

c. The block is added at the end.

PROGRAM

#include <stdio.h>

#define FILE\_SIZE 100

#define BLOCK\_SIZE 1

int main() {

int block\_number = 50;

int contiguous\_start\_block = 0;

int linked\_current\_block = 0;

int linked\_block\_count = 0;

int indexed\_index\_block = 0;

int indexed\_block\_count = 0;

contiguous\_start\_block = block\_number;

linked\_current\_block = block\_number;

while (linked\_current\_block != -1) {

linked\_block\_count++;

linked\_current\_block = /\* read block from disk and get pointer \*/;

}

indexed\_index\_block = /\* read index block from disk \*/;

indexed\_block\_count = /\* read block from disk using pointer from index block \*/;

printf("Contiguous Allocation: %d disk I/O operations\n", 1);

printf("Linked Allocation: %d disk I/O operations\n", linked\_block\_count);

printf("Indexed Allocation: %d disk I/O operations\n", 1);

return 0;

}

OUTPUT

26. Write a program to compute the average waiting time and average turnaround time based on Priority scheduling for the following process with the given CPU burst times (and the assumption that all jobs arrive at the same time.)

Process Burst Time Priority

P1 30 2

P2 5 1

P3 12 3

PROGRAM:-

#include <stdio.h>

struct Process {

int id;

int burstTime;

int priority;

int waitingTime;

int turnaroundTime;

};

void calculateWaitingTime(struct Process processes[], int n) {

int currentTime = 0;

int i, j;

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

processes[i].waitingTime = currentTime;

currentTime += processes[i].burstTime;

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

if (processes[j].priority < processes[i].priority)

processes[i].waitingTime += processes[j].burstTime;

else

processes[j].waitingTime += processes[i].burstTime;

}

}

}

void calculateTurnaroundTime(struct Process processes[], int n) {

int i;

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

processes[i].turnaroundTime = processes[i].waitingTime + processes[i].burstTime;

}

}

void calculateAverageTimes(struct Process processes[], int n, float \*avgWaitingTime, float \*avgTurnaroundTime) {

int i;

int totalWaitingTime = 0, totalTurnaroundTime = 0;

calculateWaitingTime(processes, n);

calculateTurnaroundTime(processes, n);

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

totalWaitingTime += processes[i].waitingTime;

totalTurnaroundTime += processes[i].turnaroundTime;

}

\*avgWaitingTime = (float)totalWaitingTime / n;

\*avgTurnaroundTime = (float)totalTurnaroundTime / n;

}

void displayProcesses(struct Process processes[], int n) {

int i;

printf("Process\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");

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

printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burstTime, processes[i].priority,

processes[i].waitingTime, processes[i].turnaroundTime);

}

}

int main() {

int n, i;

float avgWaitingTime, avgTurnaroundTime;

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

scanf("%d", &n);

struct Process processes[n];

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

processes[i].id = i + 1;

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

scanf("%d", &processes[i].burstTime);

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

scanf("%d", &processes[i].priority);

}

calculateAverageTimes(processes, n, &avgWaitingTime, &avgTurnaroundTime);

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

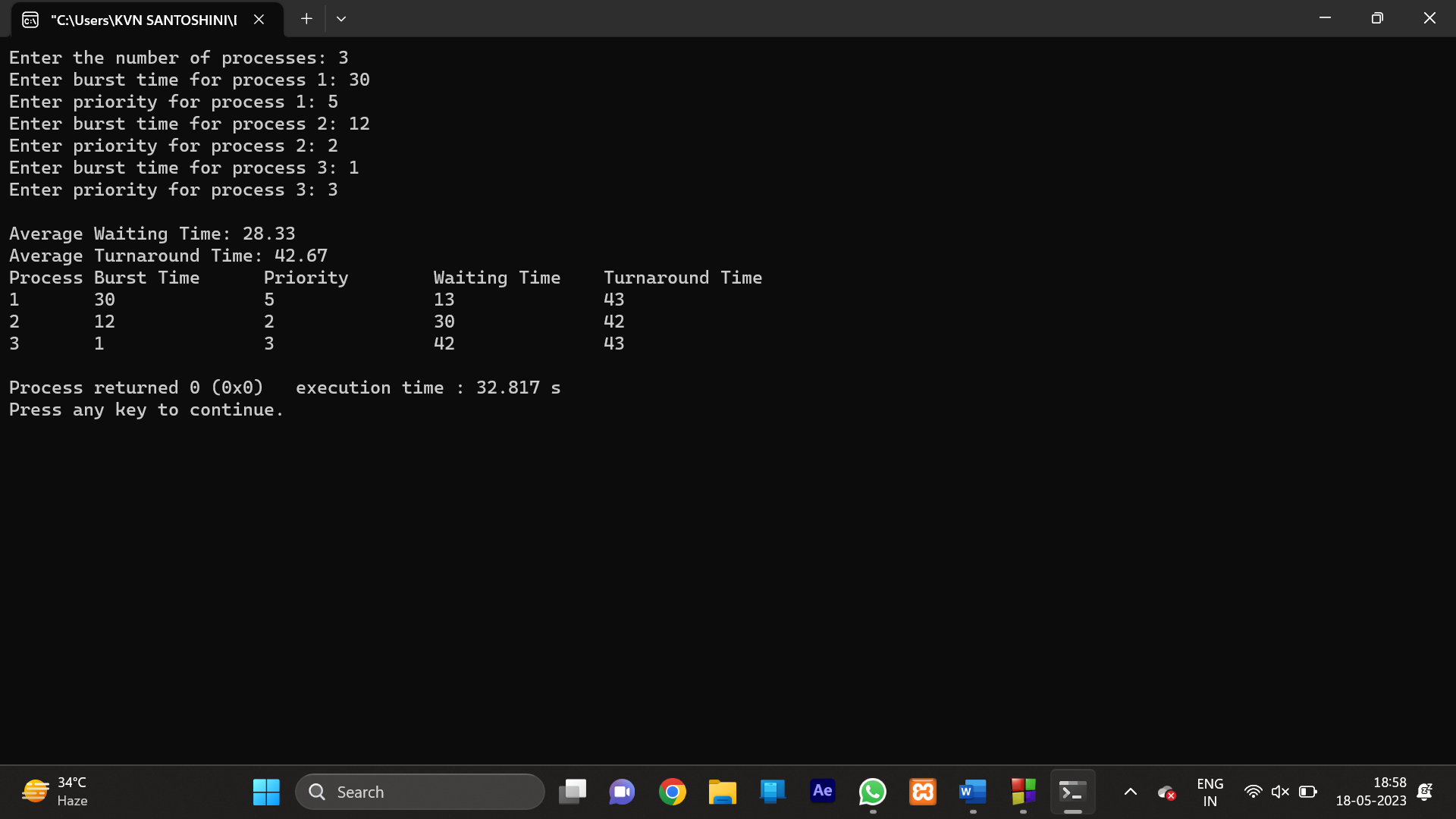
printf("\nAverage Turnaround Time: %.2f\n", avgTurnaroundTime);

displayProcesses(processes, n);

return 0;

}

OUTPUT:-



27. Write a program for semaphore signaling mechanism where a process can signal a process that is waiting on a semaphore.

Test Case:

number of instances: 4

Number of Processes: 4 (P1, P2, P3, P4) all are calling wait operation on S

Show the response when another process P5 wants the same resource.

PROGRAM:-

import threading

class Semaphore:

def \_init\_(self):

self.lock = threading.Lock()

self.signal = threading.Condition(self.lock)

self.flag = False

def wait(self):

with self.lock:

while not self.flag:

self.signal.wait()

# Reset the flag

self.flag = False

def signal(self):

with self.lock:

self.flag = True

self.signal.notify()

# Example usage

def process\_a(semaphore):

print("Process A is performing some task.")

# Performing task...

semaphore.signal()

def process\_b(semaphore):

print("Process B is waiting.")

semaphore.wait()

print("Process B received the signal and resumed.")

def main():

semaphore = Semaphore()

thread\_a = threading.Thread(target=process\_a, args=(semaphore,))

thread\_b = threading.Thread(target=process\_b, args=(semaphore,))

thread\_a.start()

thread\_b.start()

thread\_a.join()

thread\_b.join()

if \_name\_ == '\_main\_':

main()

OUTPUT:-

28. Write a C program to create a file using the system call. Read the data from the user and write the same in the file. Also, Read the data from the file and print the same in the console.

PROGRAM:-

#include <stdio.h>

#include <stdlib.h>

int main() {

FILE \*file;

char data[100];

file = fopen("data.txt", "w");

if (file == NULL) {

printf("Error creating the file.\n");

return 1;

}

printf("Enter data to write to the file: ");

fgets(data, sizeof(data), stdin);

fprintf(file, "%s", data);

printf("Data written to the file successfully.\n");

fclose(file);

file = fopen("data.txt", "r");

if (file == NULL) {

printf("Error opening the file.\n");

return 1;

}

printf("\nData read from the file:\n");

while (fgets(data, sizeof(data), file) != NULL) {

printf("%s", data);

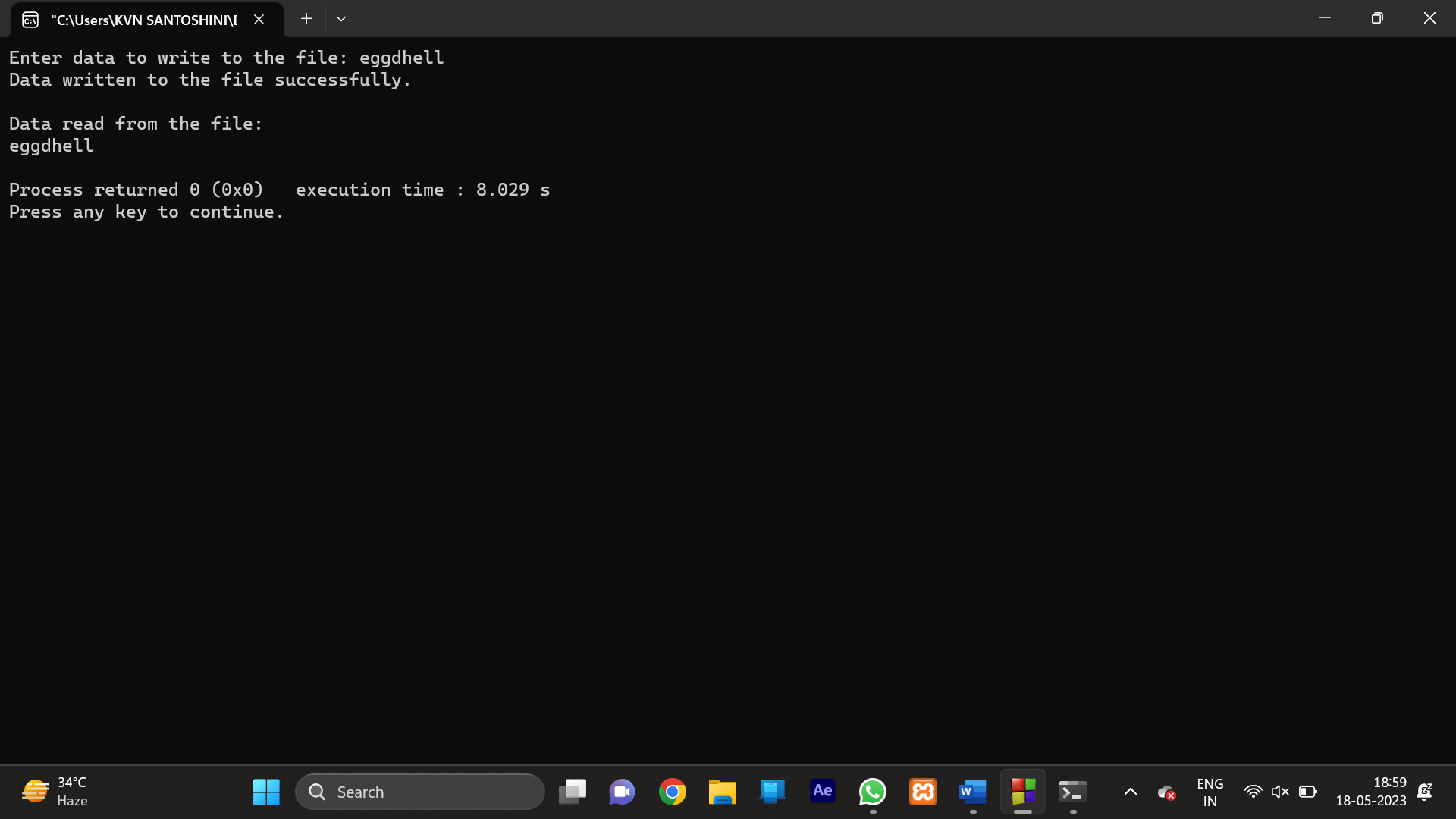
}

fclose(file);

return 0;

}

OUTPUT:-



29.Write a C program to implement the first-fit algorithm for memory management.

Test Case:

Memory partitions: 40 KB, 10 KB, 30 KB, 60 KB, (in order) Show the outcome for the test case with first-fit algorithms to place the processes of size 100 KB.50 KB.30 KB ,120 KB,35 KB (in order)

PROGRAM:-

#include <stdio.h>

#define MAX\_BLOCKS 100

#define MAX\_PROCESS 100

struct Block {

int size;

int allocated;

};

void firstFit(struct Block blocks[], int m, int processSize[], int n) {

int i, j;

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

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

if (blocks[j].allocated == 0 && blocks[j].size >= processSize[i]) {

blocks[j].allocated = 1;

break;

}

}

if (j != m) {

printf("Process %d of size %dKB is allocated to block %d\n", i + 1, processSize[i], j + 1);

} else {

printf("Process %d of size %dKB cannot be allocated\n", i + 1, processSize[i]);

}

}

}

int main() {

int m, n, i;

printf("Enter the number of memory blocks: ");

scanf("%d", &m);

struct Block blocks[MAX\_BLOCKS];

printf("Enter the size of each memory block (in KB):\n");

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

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

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

blocks[i].allocated = 0;

}

printf("\nEnter the number of processes: ");

scanf("%d", &n);

int processSize[MAX\_PROCESS];

printf("Enter the size of each process (in KB):\n");

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

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

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

}

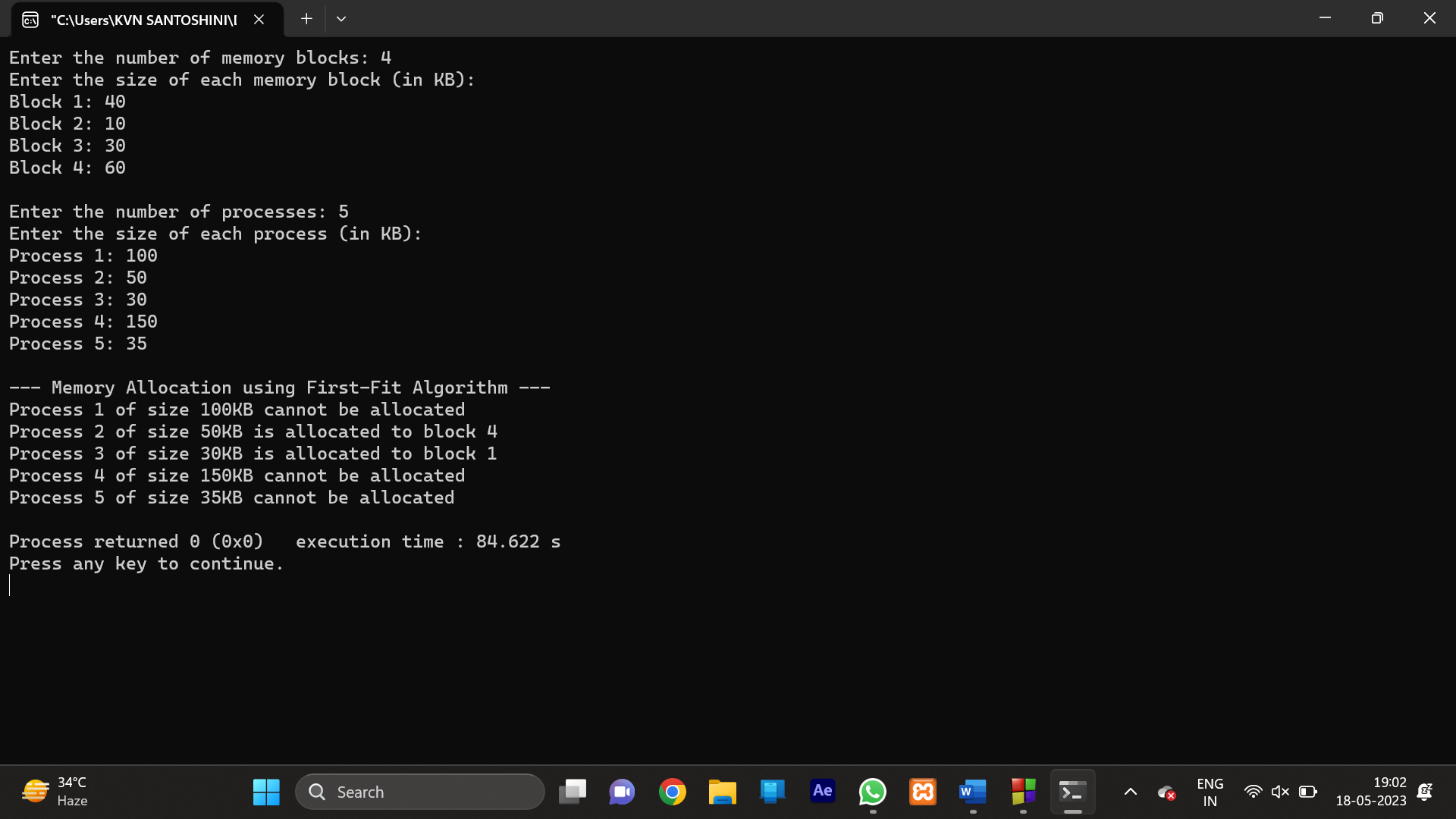
printf("\n--- Memory Allocation using First-Fit Algorithm ---\n");

firstFit(blocks, m, processSize, n);

return 0;

}

OUTPUT:-



30. Write a C Program to create two threads and print even numbers with one thread and odd numbers with another thread.

PROGRAM:-

#include <stdio.h>

#include <pthread.h>

void \*print\_even\_numbers(void \*arg) {

int i;

for (i = 2; i <= 10; i += 2) {

printf("Even Number: %d\n", i);

}

pthread\_exit(NULL);

}

void \*print\_odd\_numbers(void \*arg) {

int i;

for (i = 1; i <= 9; i += 2) {

printf("Odd Number: %d\n", i);

}

pthread\_exit(NULL);

}

int main() {

pthread\_t even\_thread, odd\_thread;

if (pthread\_create(&even\_thread, NULL, print\_even\_numbers, NULL) != 0) {

printf("Error creating the even thread.\n");

return 1;

}

if (pthread\_create(&odd\_thread, NULL, print\_odd\_numbers, NULL) != 0) {

printf("Error creating the odd thread.\n");

return 1;

}

if (pthread\_join(even\_thread, NULL) != 0) {

printf("Error joining the even thread.\n");

return 1;

}

if (pthread\_join(odd\_thread, NULL) != 0) {

printf("Error joining the odd thread.\n");

return 1;

}

return 0;

}

OUTPUT:-

