DAY 3 PROGRAMS

Name:-OMKAR

Reg.No:-192111134

13. Write a C program to implement single-level directory system. In which all the files are placed in one directory and there are no sub directories.

Test Case: Create one directory with the name of CSE and Add 3 files(A,B,C) in to that directory

Program:-

#include <stdio.h>

#include <stdlib.h>

int main() {

system("mkdir CSE");

FILE \*fileA = fopen("CSE/A", "w");

FILE \*fileB = fopen("CSE/B", "w");

FILE \*fileC = fopen("CSE/C", "w");

if (fileA == NULL || fileB == NULL || fileC == NULL) {

printf("Error: Unable to create files\n");

return 1;

}

fprintf(fileA, "This is file A\n");

fprintf(fileB, "This is file B\n");

fprintf(fileC, "This is file C\n");

fclose(fileA);

fclose(fileB);

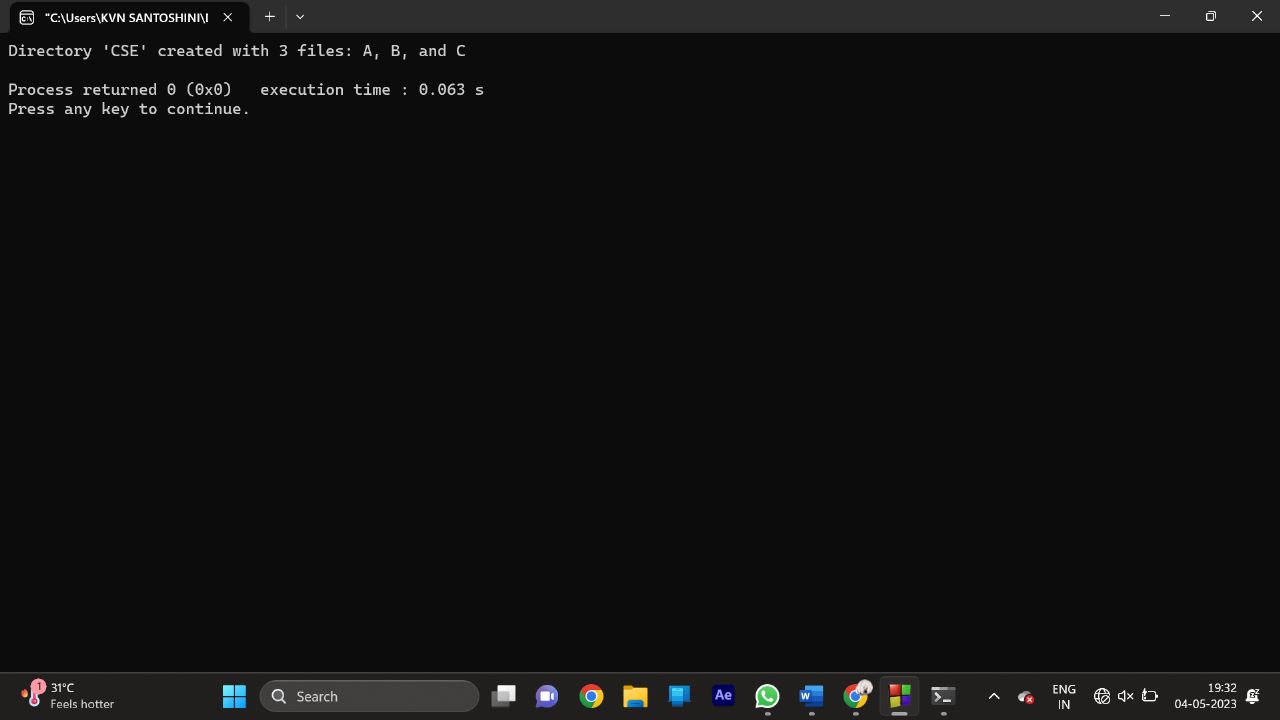
fclose(fileC);

printf("Directory 'CSE' created with 3 files: A, B, and C\n");

return 0;

}

Output:-



14. Write a C program to illustrate the page replacement method where the page which is not in demand for the longest future time is replaced by the new page and determine the number of page faults for the following

test case: No. of page frames: 3; Page reference sequence 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0 and 1.

Program:-

#include <stdio.h>

#define MAX\_FRAMES 3

int main()

{

int frames[MAX\_FRAMES], pages[MAX\_FRAMES], page\_faults = 0;

int page\_reference[] = {7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1};

int num\_pages = sizeof(page\_reference)/sizeof(page\_reference[0]);

int i, j, k, max\_future\_distance, page\_to\_replace;

for(i = 0; i < MAX\_FRAMES; i++)

{

frames[i] = -1;

pages[i] = -1;

}

for(i = 0; i < num\_pages; i++)

{

int page\_found = 0;

int page = page\_reference[i];

for(j = 0; j < MAX\_FRAMES; j++)

{

if(frames[j] == page)

{

page\_found = 1;

break;

}

}

if(page\_found == 0)

{

for(j = 0; j < MAX\_FRAMES; j++)

{

int page\_exists = 0;

int future\_distance = 0;

for(k = i + 1; k < num\_pages; k++)

{

if(frames[j] == page\_reference[k])

{

page\_exists = 1;

future\_distance = k - i;

break;

}

}

if(page\_exists == 0)

{

page\_faults++;

frames[j] = page;

break;

}

if(future\_distance > max\_future\_distance)

{

max\_future\_distance = future\_distance;

page\_to\_replace = j;

}

}

page\_faults++;

frames[page\_to\_replace] = page;

}

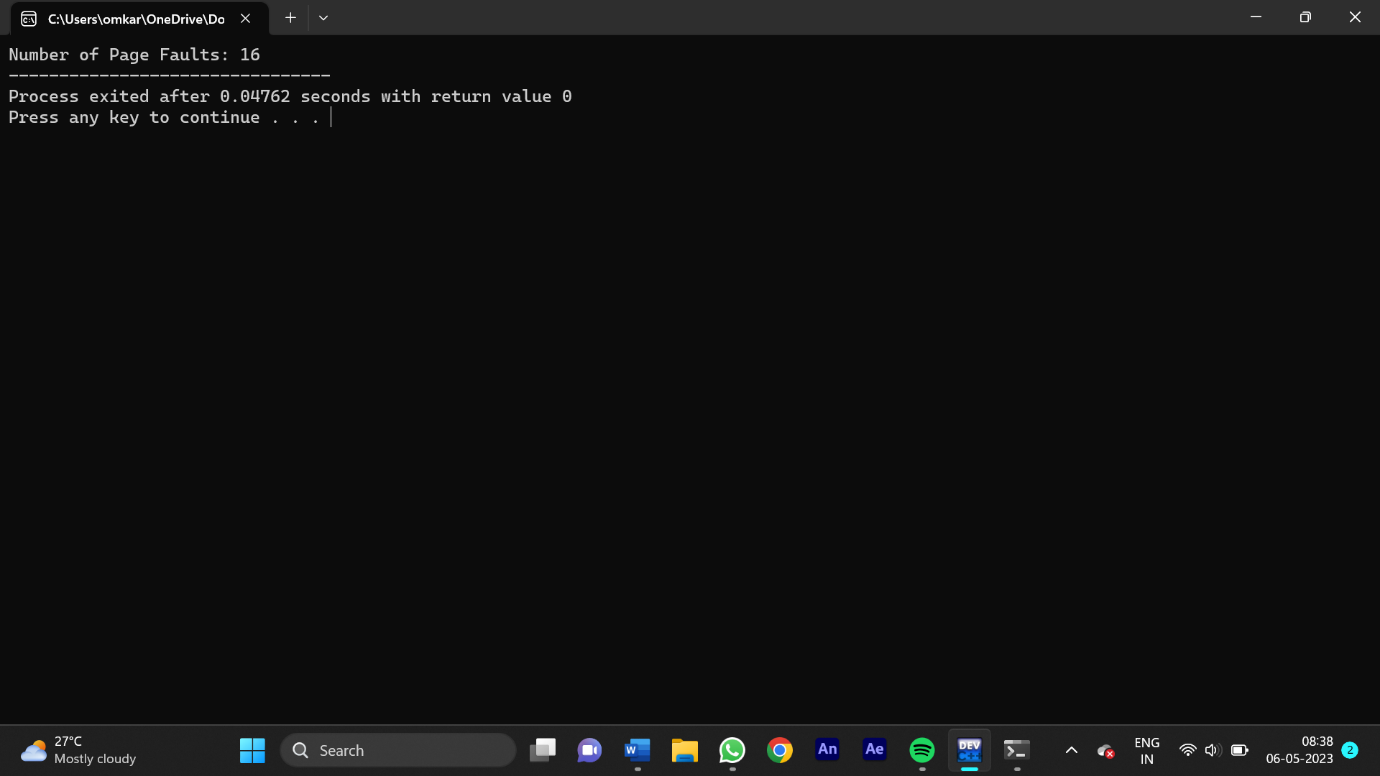
}

printf("Number of Page Faults: %d", page\_faults);

return 0;

}

OUTPUT:-



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

No of tracks:9; Track position:55 58 60 70 18 90 150 160 184

Program:-

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#define MAX\_REQUESTS 1000

int main() {

int requests[MAX\_REQUESTS];

int num\_requests;

int head\_pos, total\_head\_movement;

float avg\_head\_movement;

printf("Enter the number of disk requests: ");

scanf("%d", &num\_requests);

printf("Enter the disk requests:\n");

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

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

}

printf("Enter the initial head position: ");

scanf("%d", &head\_pos);

total\_head\_movement = abs(head\_pos - requests[0]);

for (int i = 1; i < num\_requests; i++) {

total\_head\_movement += abs(requests[i] - requests[i-1]);

}

avg\_head\_movement = (float)total\_head\_movement / num\_requests;

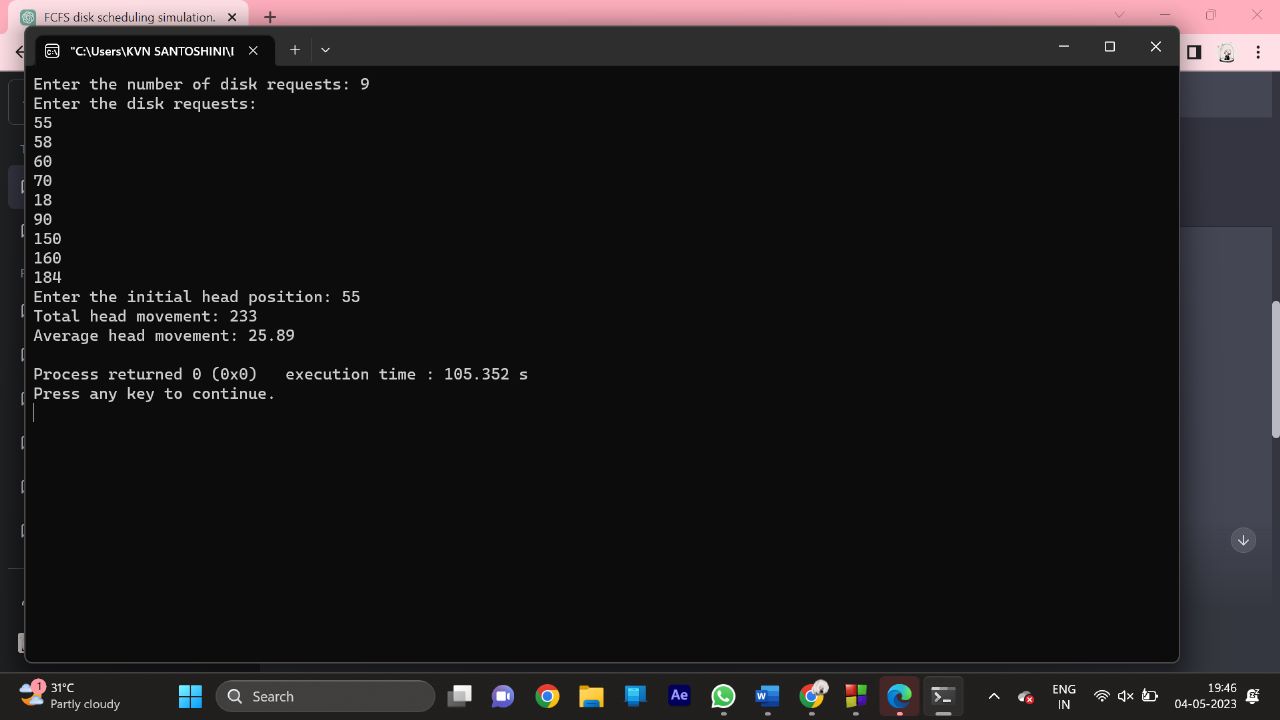
printf("Total head movement: %d\n", total\_head\_movement);

printf("Average head movement: %.2f\n", avg\_head\_movement);

return 0;

}

Output:-



16. Write a program to compute the average waiting time and average turnaround time based on First Come First Serve for the following process with the given CPU burst times, (and the assumption that all jobs arrive at the same time.)

Process Burst Time

P1 10

P2 15

P3 25

Program:-

#include <stdio.h>

#define MAX\_PROCESSES 100

int main() {

int burst\_times[MAX\_PROCESSES];

int num\_processes;

int waiting\_times[MAX\_PROCESSES];

int turnaround\_times[MAX\_PROCESSES];

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

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

scanf("%d", &num\_processes);

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

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

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

}

waiting\_times[0] = 0;

turnaround\_times[0] = burst\_times[0];

for (int i = 1; i < num\_processes; i++) {

waiting\_times[i] = waiting\_times[i-1] + burst\_times[i-1];

turnaround\_times[i] = waiting\_times[i] + burst\_times[i];

}

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

total\_waiting\_time += waiting\_times[i];

total\_turnaround\_time += turnaround\_times[i];

}

float avg\_waiting\_time = (float)total\_waiting\_time / num\_processes;

float avg\_turnaround\_time = (float)total\_turnaround\_time / num\_processes;

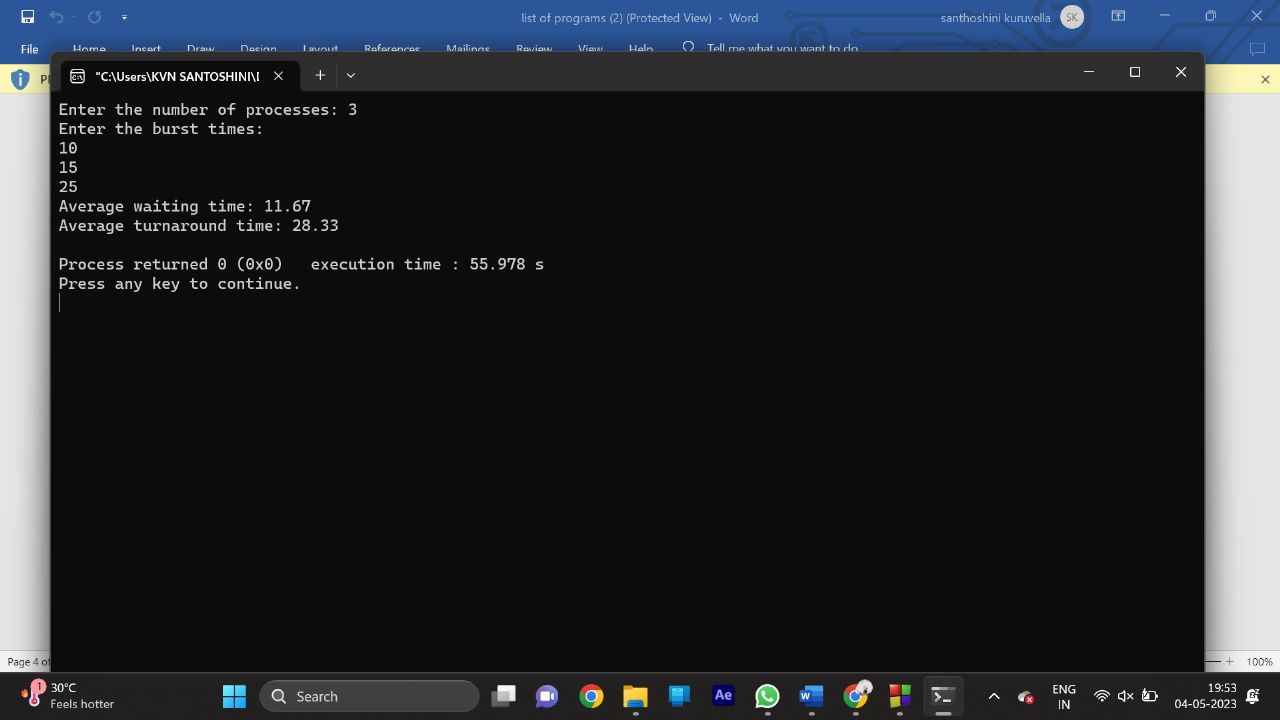
printf("Average waiting time: %.2f\n", avg\_waiting\_time);

printf("Average turnaround time: %.2f\n", avg\_turnaround\_time);

return 0;

}

Output:-



17. Write a program to compute the average waiting time and average turnaround time based on Round Robin scheduling for the following process with the given CPU burst times and quantum time slots 4 ms, ( and the assumption that all jobs arrive at the same time.)

Process Burst Time

P1 24

P2 3

P3 3

Program:-

#include <stdio.h>

#define MAX\_PROCESSES 100

int main() {

int burst\_times[MAX\_PROCESSES];

int num\_processes;

int waiting\_times[MAX\_PROCESSES];

int turnaround\_times[MAX\_PROCESSES];

int remaining\_burst\_times[MAX\_PROCESSES];

int quantum = 4;

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

int completed\_processes = 0;

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

scanf("%d", &num\_processes);

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

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

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

remaining\_burst\_times[i] = burst\_times[i];

}

while (completed\_processes < num\_processes) {

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

if (remaining\_burst\_times[i] > 0) {

if (remaining\_burst\_times[i] <= quantum) {

time += remaining\_burst\_times[i];

waiting\_times[i] = time - burst\_times[i];

turnaround\_times[i] = time;

remaining\_burst\_times[i] = 0;

completed\_processes++;

} else {

time += quantum;

remaining\_burst\_times[i] -= quantum;

}

}

}

}

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

total\_waiting\_time += waiting\_times[i];

total\_turnaround\_time += turnaround\_times[i];

}

float avg\_waiting\_time = (float)total\_waiting\_time / num\_processes;

float avg\_turnaround\_time = (float)total\_turnaround\_time / num\_processes;

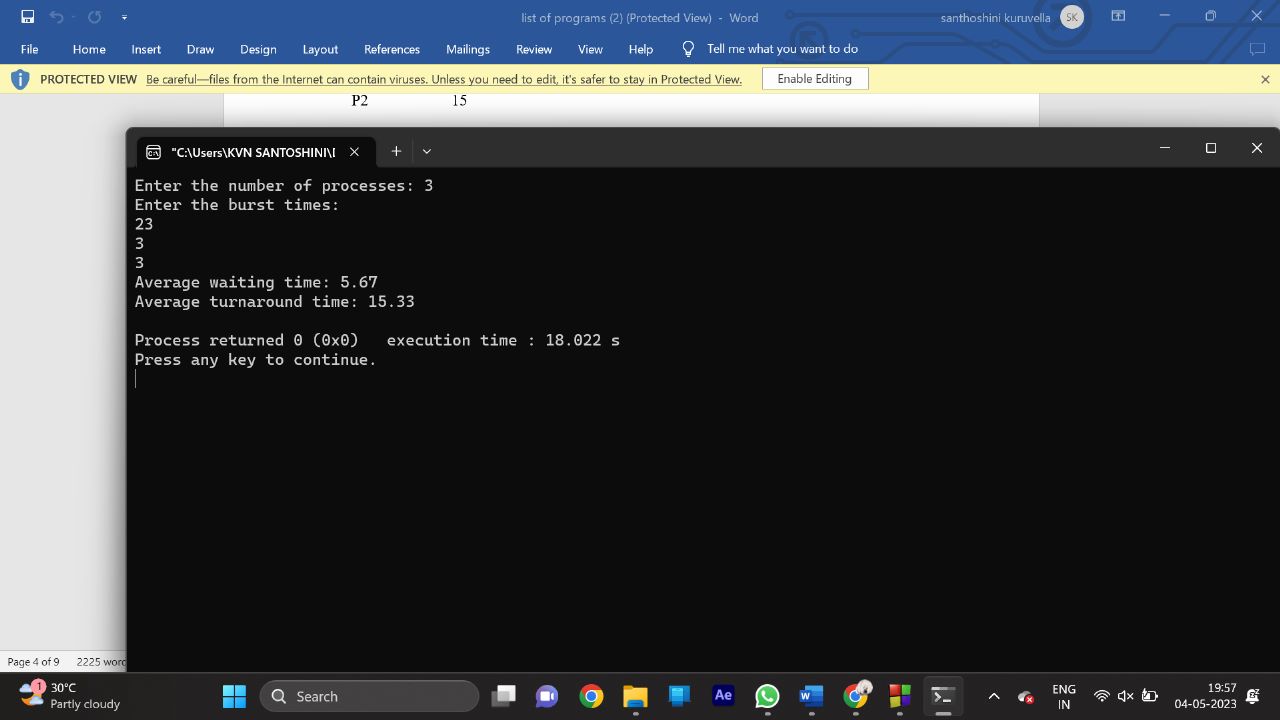
printf("Average waiting time: %.2f\n", avg\_waiting\_time);

printf("Average turnaround time: %.2f\n", avg\_turnaround\_time);

return 0;

}

Output:-



18. Write a program for solving the producer consumer problem with the following scenario: The producer should produce data only when the buffer is not full. Data can only be consumed by the consumer if and only if the memory buffer is not empty.

Test Case:

Buffer Size: 3

Consume an item in the beginning and show that the buffer is EMPTY

Produce 4 items and show that the buffer is FULL

Program:-

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define BUFFER\_SIZE 10

int buffer[BUFFER\_SIZE];

int producer\_index = 0, consumer\_index = 0;

sem\_t empty;

sem\_t full;

pthread\_mutex\_t mutex;

void \*producer(void \*arg) {

int data = 1;

while (1) {

sleep(1);

printf("Producer produced data: %d\n", data);

sem\_wait(&empty);

pthread\_mutex\_lock(&mutex);

buffer[producer\_index] = data;

producer\_index = (producer\_index + 1) % BUFFER\_SIZE;

pthread\_mutex\_unlock(&mutex);

sem\_post(&full);

data++;

}

}

void \*consumer(void \*arg) {

while (1) {

sem\_wait(&full);

pthread\_mutex\_lock(&mutex);

int data = buffer[consumer\_index];

printf("Consumer consumed data: %d\n", data);

consumer\_index = (consumer\_index + 1) % BUFFER\_SIZE;

pthread\_mutex\_unlock(&mutex);

sem\_post(&empty);

sleep(2);

}

}

int main() {

sem\_init(&empty, 0, BUFFER\_SIZE);

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

pthread\_mutex\_init(&mutex, NULL);

pthread\_t producer\_thread, consumer\_thread;

pthread\_create(&producer\_thread, NULL, producer, NULL);

pthread\_create(&consumer\_thread, NULL, consumer, NULL);

pthread\_join(producer\_thread, NULL);

pthread\_join(consumer\_thread, NULL);

sem\_destroy(&empty);

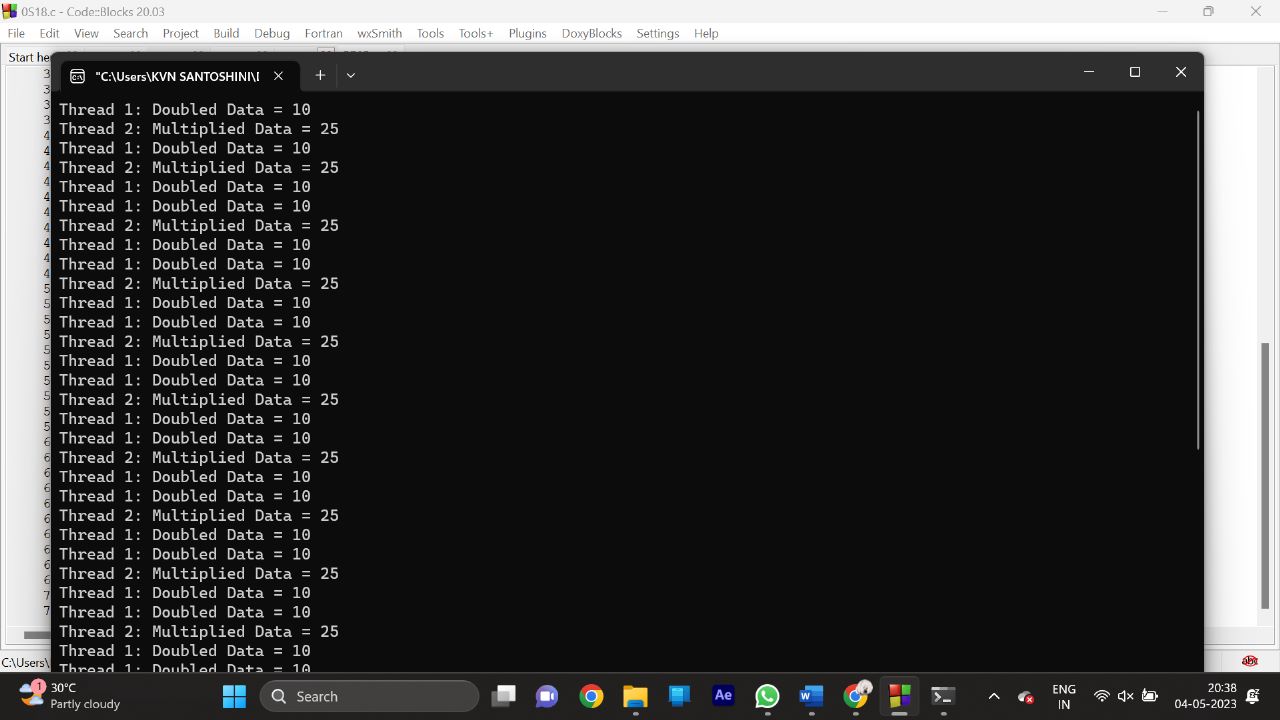
sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

Output:-



19. Write a C program to create two threads to access shared memory which is an integer in a synchronized fashion using semaphore. In the first thread print the doubled the integer data after reading from the shared memory. In the second thread, print the five times of the integer data after reading from the shared memory

Program:-

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

int shared\_data = 5;

sem\_t mutex;

void \*thread1(void \*arg) {

while (1) {

sem\_wait(&mutex);

int data = shared\_data;

data = data \* 2;

printf("Thread 1: Doubled Data = %d\n", data);

sem\_post(&mutex);

sleep(1);

}

}

void \*thread2(void \*arg) {

while (1) {

sem\_wait(&mutex);

int data = shared\_data;

data = data \* 5;

printf("Thread 2: Multiplied Data = %d\n", data);

sem\_post(&mutex);

sleep(2);

}

}

int main() {

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

pthread\_t tid1, tid2;

pthread\_create(&tid1, NULL, thread1, NULL);

pthread\_create(&tid2, NULL, thread2, NULL);

pthread\_join(tid1, NULL);

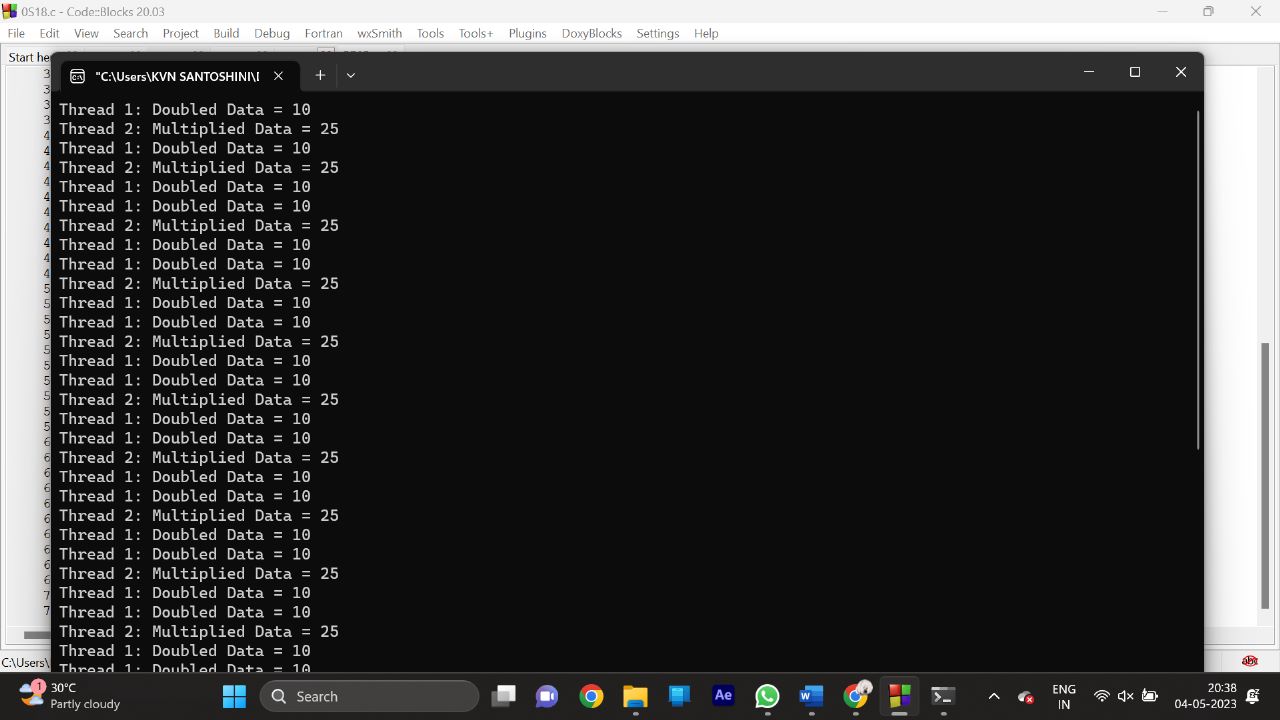
pthread\_join(tid2, NULL);

sem\_destroy(&mutex);

return 0;

}

Output:-



20. Write a C program to implement the worst-fit algorithm and allocate the memory block to each process.

Test Case:

Memory partitions: 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order),

Show the outcome for the test case with the worst-fit algorithms to place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)

Program:-

#include <stdio.h>

#define MAX\_BLOCKS 10

#define MAX\_PROCESSES 10

int main() {

int blocks[MAX\_BLOCKS], processes[MAX\_PROCESSES];

int num\_blocks, num\_processes, i, j;

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

scanf("%d", &num\_blocks);

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

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

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

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

}

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

scanf("%d", &num\_processes);

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

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

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

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

}

int allocation[MAX\_PROCESSES];

for (i = 0; i < MAX\_PROCESSES; i++) {

allocation[i] = -1;

}

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

int worst\_index = -1;

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

if (blocks[j] >= processes[i]) {

if (worst\_index == -1 || blocks[j] > blocks[worst\_index]) {

worst\_index = j;

}

}

}

if (worst\_index != -1) {

allocation[i] = worst\_index;

blocks[worst\_index] -= processes[i];

}

}

printf("\nProcess\tProcess Size\tBlock\tBlock Size\n");

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

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

if (allocation[i] != -1) {

printf("%d\t%d\n", allocation[i], blocks[allocation[i]]);

} else {

printf("Not Allocated\n");

}

}

return 0;

}

Output:-

