DAY 5 PROGRAMS

NAME:-OMKAR

REG.NO:-192111134

31. Write a C program to simulate CSCAN 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 SIZE 1000

int abs\_diff(int a, int b) {

return abs(a - b);

}

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

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

}

void cscan(int requests[], int num\_requests, int head, int max\_cylinder) {

int total\_head\_movement = 0;

int i, current, previous;

qsort(requests, num\_requests, sizeof(int), compare);

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

if (requests[i] >= head) {

current = i;

break;

}

}

printf("C-SCAN Path: ");

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

printf("%d ", requests[i]);

total\_head\_movement += abs\_diff(head, requests[i]);

head = requests[i];

}

if (current != 0) {

printf("%d ", max\_cylinder);

total\_head\_movement += abs\_diff(head, max\_cylinder);

head = 0;

}

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

printf("%d ", requests[i]);

total\_head\_movement += abs\_diff(head, requests[i]);

head = requests[i];

}

printf("\n");

printf("Total Head Movement: %d\n", total\_head\_movement);

printf("Average Head Movement: %.2f\n", (float)total\_head\_movement / num\_requests);

}

int main() {

int requests[SIZE];

int num\_requests, i, head, max\_cylinder;

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

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

printf("Enter the requests (cylinder numbers):\n");

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

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

}

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

scanf("%d", &head);

printf("Enter the maximum cylinder size: ");

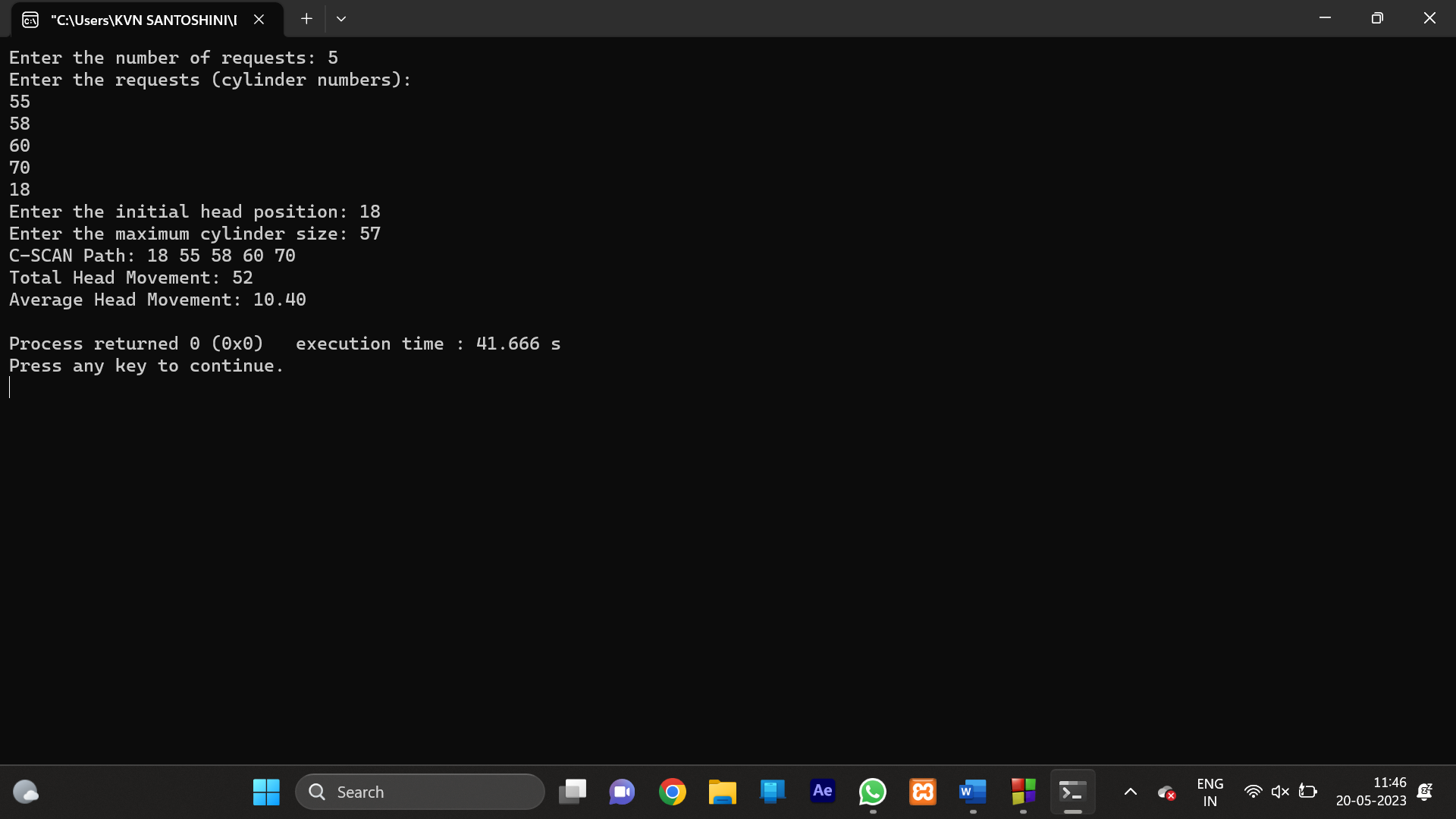
scanf("%d", &max\_cylinder);

cscan(requests, num\_requests, head, max\_cylinder);

return 0;

}

OUTPUT:-



32. 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:9

Track position:55 58 60 70 18 90 150 160 184

Program:-

#include <stdio.h>

#include <stdlib.h>

#define SIZE 1000

int abs\_diff(int a, int b) {

return abs(a - b);

}

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

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

}

void scan(int requests[], int num\_requests, int head, int max\_cylinder) {

int total\_head\_movement = 0;

int i, current, previous;

qsort(requests, num\_requests, sizeof(int), compare);

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

if (requests[i] >= head) {

current = i;

break;

}

}

printf("SCAN Path: ");

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

printf("%d ", requests[i]);

total\_head\_movement += abs\_diff(head, requests[i]);

head = requests[i];

}

if (current != 0) {

printf("%d ", max\_cylinder);

total\_head\_movement += abs\_diff(head, max\_cylinder);

head = max\_cylinder;

}

for (i = current - 1; i >= 0; i--) {

printf("%d ", requests[i]);

total\_head\_movement += abs\_diff(head, requests[i]);

head = requests[i];

}

printf("\n");

printf("Total Head Movement: %d\n", total\_head\_movement);

printf("Average Head Movement: %.2f\n", (float)total\_head\_movement / num\_requests);

}

int main() {

int requests[SIZE];

int num\_requests, i, head, max\_cylinder;

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

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

printf("Enter the requests (cylinder numbers):\n");

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

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

}

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

scanf("%d", &head);

printf("Enter the maximum cylinder size: ");

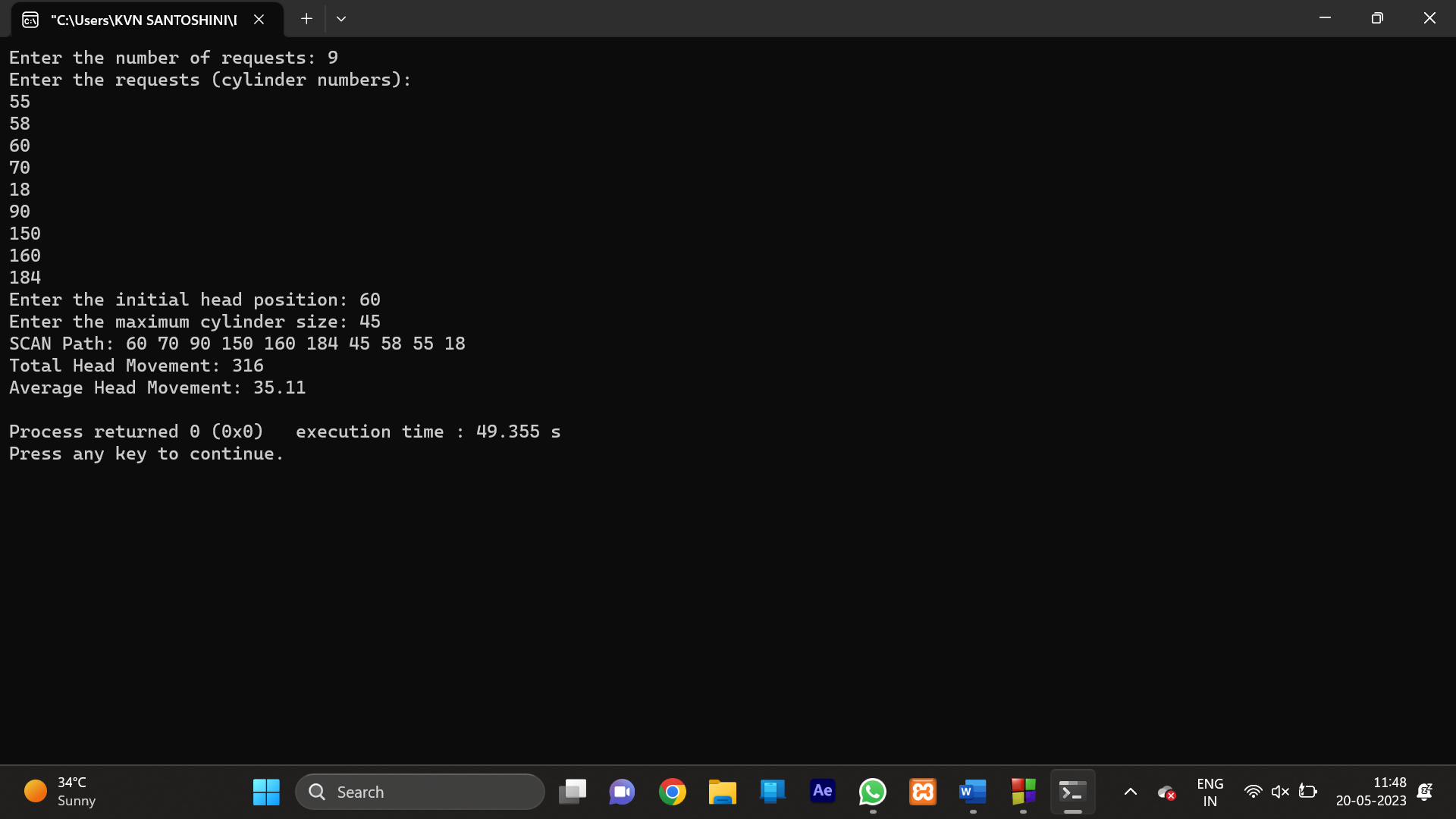
scanf("%d", &max\_cylinder);

scan(requests, num\_requests, head, max\_cylinder);

return 0;

}

Output:-



33. Write a C program using the wait system call to synchronize the parent process and child process. In the parent process print the prime numbers. In the child process generate the Fibonacci series.

Program:-

#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>

int is\_prime(int number) {

if (number <= 1)

return 0;

for (int i = 2; i \* i <= number; i++) {

if (number % i == 0)

return 0;

}

return 1;

}

void generate\_fibonacci(int n) {

int first = 0, second = 1;

printf("Fibonacci Series: %d %d ", first, second);

for (int i = 3; i <= n; i++) {

int next = first + second;

printf("%d ", next);

first = second;

second = next;

}

printf("\n");

}

int main() {

pid\_t pid;

int n;

printf("Enter a positive integer: ");

scanf("%d", &n);

pid = fork();

if (pid < 0) {

fprintf(stderr, "Fork failed\n");

return 1;

} else if (pid == 0) {

generate\_fibonacci(n);

} else {

wait(NULL);

printf("Prime Numbers: ");

for (int i = 2; i <= n; i++) {

if (is\_prime(i))

printf("%d ", i);

}

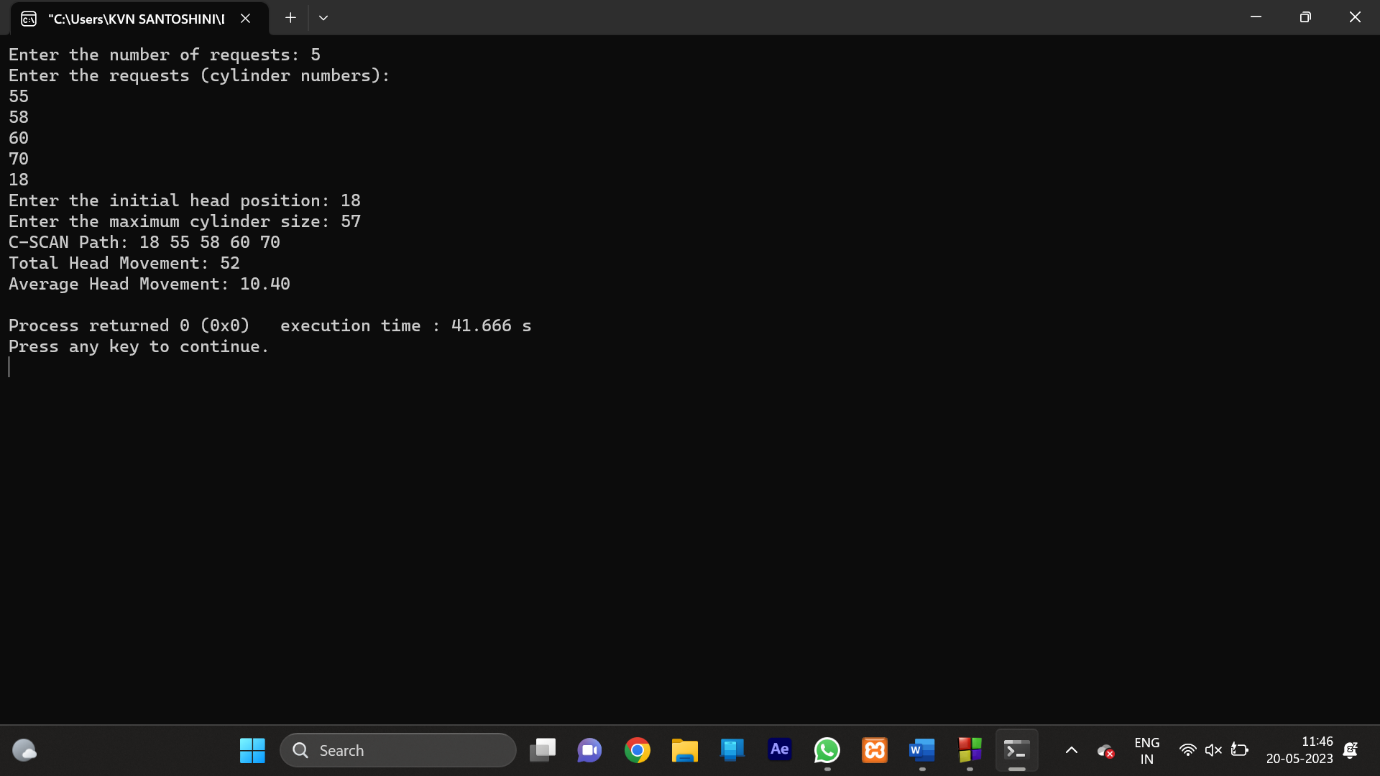
printf("\n");

}

return 0;

}

Output:-



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

Test Case:

Memory partitions: 40 KB, 10 KB. 30 KB, .60 KB (in order),

Show the outcome for the test case with the worst-fit algorithms to place 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\_PROCESSES 100

void worstFit(int blocks[], int num\_blocks, int processes[], int num\_processes) {

int allocation[MAX\_PROCESSES] = {0};

int i, j;

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

int worst\_block\_index = -1;

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

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

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

worst\_block\_index = j;

}

}

}

if (worst\_block\_index != -1) {

allocation[i] = worst\_block\_index;

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

}

}

printf("\nProcess No.\tProcess Size\tBlock No.\n");

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

printf("%d\t\t%d\t\t", i + 1, processes[i]);

if (allocation[i] != 0) {

printf("%d\n", allocation[i] + 1);

} else {

printf("Not Allocated\n");

}

}

}

int main() {

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

int num\_blocks, num\_processes, i;

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

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

printf("Enter the sizes of memory blocks:\n");

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

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

}

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

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

printf("Enter the sizes of processes:\n");

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

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

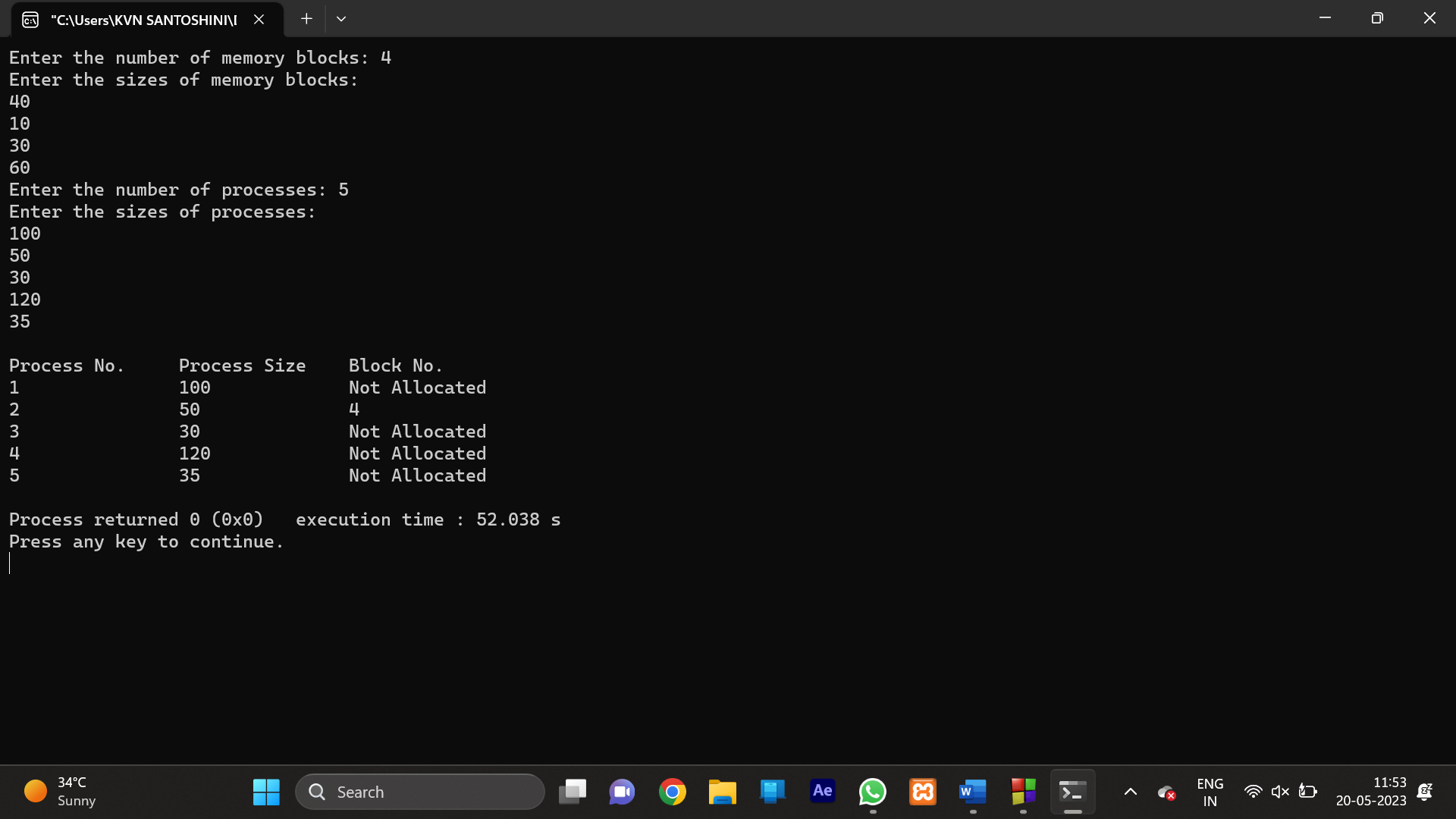
}

worstFit(blocks, num\_blocks, processes, num\_processes);

return 0;

}

Output:-



35. Write a C program to simulate the sequential file allocation in a very simple file system with a disk of 16 blocks, each block is of 1 KB size and first 8 blocks (0 to 7) are allocated to the “iNodes” and can’t be used by the file system. Blocks available for allocation are from block 8 to block 16. Minimum file size is 1 KB. 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.

Test Case: If there are not enough contiguous blocks available for a file, the program must exit ()

Program:-

#include <stdio.h>

#include <stdbool.h>

#define NUM\_BLOCKS 16

#define INODE\_BLOCKS 8

#define MIN\_FILE\_SIZE 1

#define BLOCK\_SIZE 1024

typedef struct {

int startBlock;

int numBlocks;

} File;

typedef struct {

File files[NUM\_BLOCKS - INODE\_BLOCKS];

bool freeBlocks[NUM\_BLOCKS];

} FileSystem;

void initializeFileSystem(FileSystem\* fs) {

for (int i = 0; i < NUM\_BLOCKS; i++) {

fs->freeBlocks[i] = true;

}

}

void createFile(FileSystem\* fs, int fileSize) {

if (fileSize < MIN\_FILE\_SIZE) {

printf("File size should be at least 1 KB.\n");

return;

}

int requiredBlocks = (fileSize + BLOCK\_SIZE - 1) / BLOCK\_SIZE;

int startBlock = -1;

int numBlocks = 0;

for (int i = INODE\_BLOCKS; i < NUM\_BLOCKS; i++) {

if (fs->freeBlocks[i]) {

if (startBlock == -1) {

startBlock = i;

}

numBlocks++;

if (numBlocks == requiredBlocks) {

break;

}

} else {

startBlock = -1;

numBlocks = 0;

}

}

if (numBlocks == requiredBlocks) {

File newFile;

newFile.startBlock = startBlock;

newFile.numBlocks = numBlocks;

fs->files[startBlock - INODE\_BLOCKS] = newFile;

for (int i = startBlock; i < startBlock + numBlocks; i++) {

fs->freeBlocks[i] = false;

}

printf("File created successfully. Start block: %d, Number of blocks: %d\n", startBlock, numBlocks);

} else {

printf("Not enough contiguous blocks available for the file.\n");

}

}

void displayFileSystem(FileSystem\* fs) {

printf("File System Status:\n");

printf("Allocated Files:\n");

for (int i = 0; i < NUM\_BLOCKS - INODE\_BLOCKS; i++) {

File file = fs->files[i];

if (file.numBlocks > 0) {

printf("File %d: Start block: %d, Number of blocks: %d\n", i, file.startBlock, file.numBlocks);

}

}

printf("Free Blocks:\n");

for (int i = INODE\_BLOCKS; i < NUM\_BLOCKS; i++) {

if (fs->freeBlocks[i]) {

printf("Block %d: Free\n", i);

} else {

printf("Block %d: Allocated\n", i);

}

}

}

int main() {

FileSystem fs;

initializeFileSystem(&fs);

int fileSize;

printf("Enter the file size in KB: ");

scanf("%d", &fileSize);

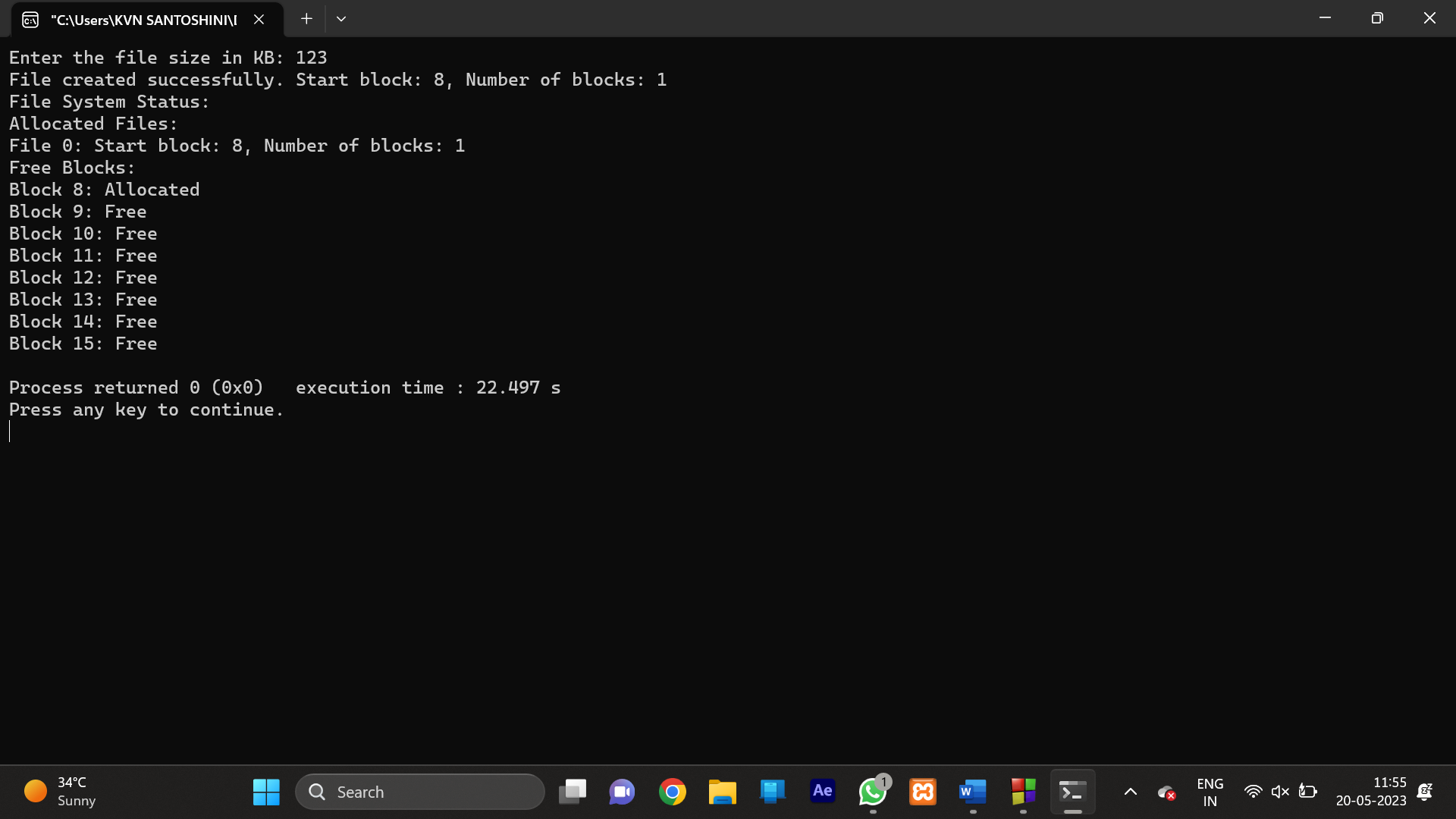
createFile(&fs, fileSize);

displayFileSystem(&fs);

return 0;

}

Output:-



36. Write a C program to simulate SSTF disk scheduling algorithm and execute your program and find the average head movement with the following test case:

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

Program:-

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <math.h>

int findShortestSeekTime(int \*requests, int n, int head) {

int shortestSeekTime = INT\_MAX;

int index = -1;

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

if (abs(requests[i] - head) < shortestSeekTime) {

shortestSeekTime = abs(requests[i] - head);

index = i;

}

}

return index;

}

int calculateHeadMovement(int \*requests, int n, int initialHead) {

int head = initialHead;

int totalHeadMovement = 0;

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

int shortestSeekTimeIndex = findShortestSeekTime(requests, n, head);

int shortestSeekTime = abs(requests[shortestSeekTimeIndex] - head);

totalHeadMovement += shortestSeekTime;

head = requests[shortestSeekTimeIndex];

requests[shortestSeekTimeIndex] = -1;

}

return totalHeadMovement;

}

int main() {

int n, initialHead;

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

scanf("%d", &n);

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

printf("Enter the disk requests: ");

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

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

}

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

scanf("%d", &initialHead);

int totalHeadMovement = calculateHeadMovement(requests, n, initialHead);

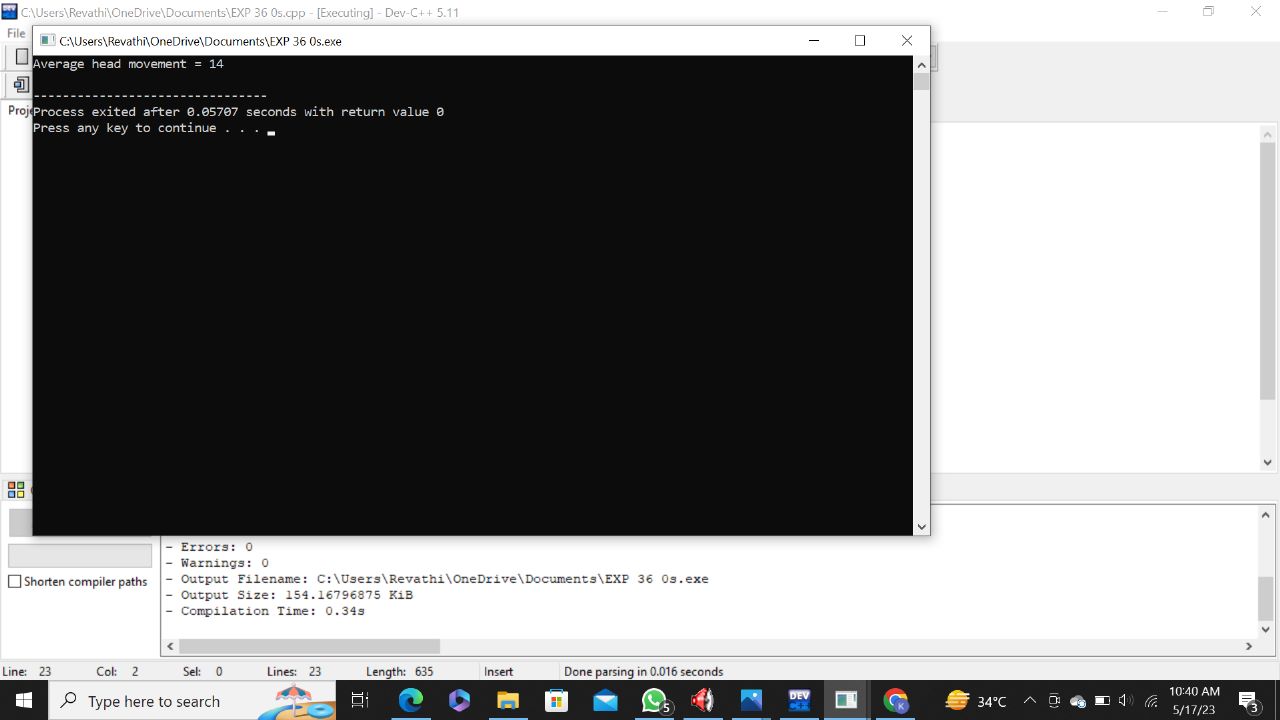
printf("Average head movement: %.2f\n", (float)totalHeadMovement / n);

free(requests);

return 0;

}

Output:-



37. Write a C program to illustrate the Optimal method of page replacement and determine the number of page faults for the following test case:

No of page frames: 3; Page reference sequence: 4, 1, 2, 4, 3, 2, 1 and 5.

Program:-

#include <stdio.h>

#include <stdlib.h>

int findOptimalPage(int pages[], int n, int memory[], int m, int currentIndex) {

int optimalPage = -1;

int farthestIndex = currentIndex;

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

int j;

for (j = currentIndex; j < n; j++) {

if (memory[i] == pages[j]) {

if (j > farthestIndex) {

farthestIndex = j;

optimalPage = i;

}

break;

}

}

if (j == n)

return i;

}

return (optimalPage == -1) ? 0 : optimalPage;

}

int pageFaultsOptimal(int pages[], int n, int m) {

int pageFaults = 0;

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

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

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

memory[i] = -1;

nextUse[i] = -1;

}

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

int j;

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

if (memory[j] == pages[i]) {

break;

}

}

if (j == m) {

int optimalPage = findOptimalPage(pages, n, memory, m, i + 1);

memory[optimalPage] = pages[i];

nextUse[optimalPage] = i + 1;

pageFaults++;

}

}

free(memory);

free(nextUse);

return pageFaults;

}

int main() {

int n, m;

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

scanf("%d", &n);

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

printf("Enter the page sequence: ");

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

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

}

printf("Enter the number of page frames: ");

scanf("%d", &m);

int pageFaults = pageFaultsOptimal(pages, n, m);

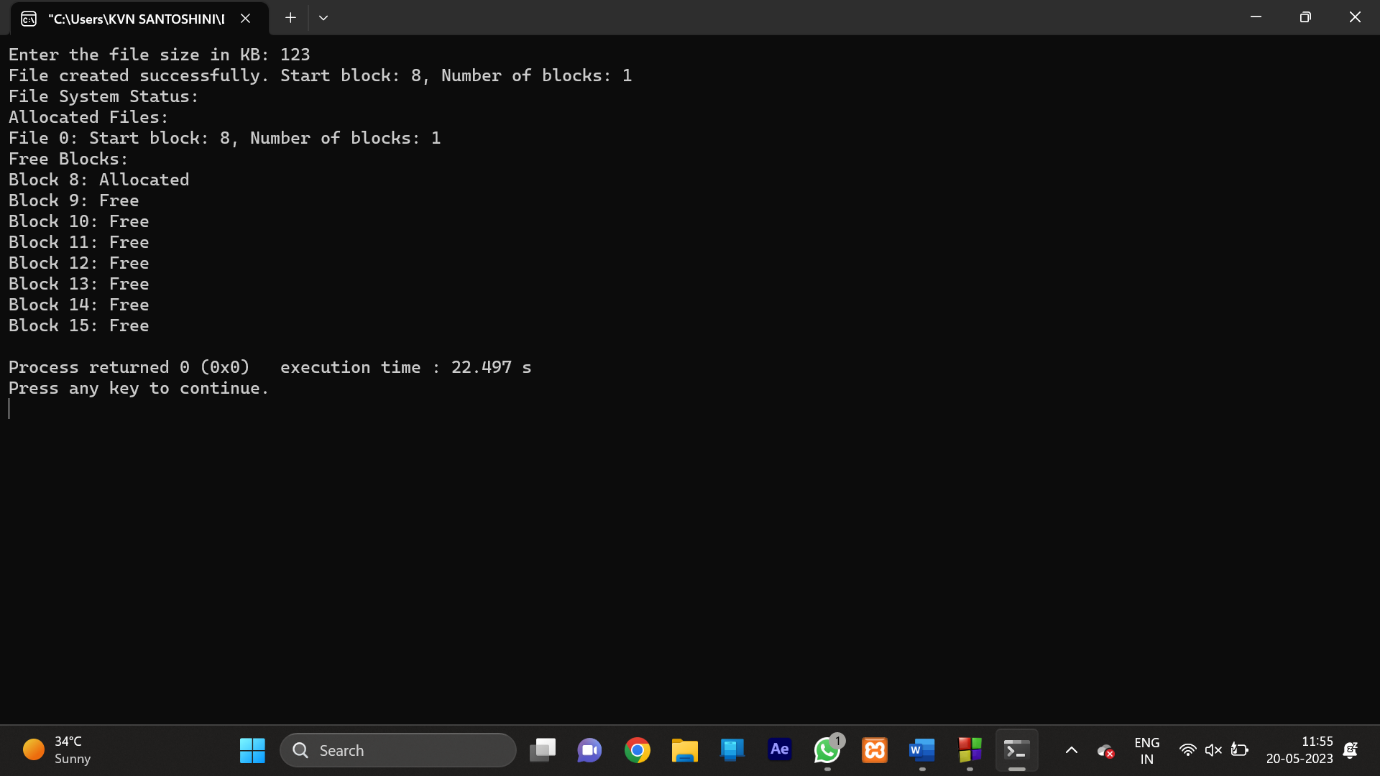
printf("Number of page faults: %d\n", pageFaults);

free(pages);

return 0;

}

Output:-



38. Consider three processes (process id 0, 1, 2 respectively) with compute time bursts 2, 4 and 8-time units. All processes arrive at time zero. Write a program to compute the average waiting time and average turnaround time based on Shortest Job First Scheduling

Program:-

#include <stdio.h>

void calculateWaitingTime(int processes[], int n, int burstTime[], int waitingTime[]) {

int remainingTime[n];

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

remainingTime[i] = burstTime[i];

}

int completed = 0;

int currentTime = 0;

int minIndex;

int minBurstTime = INT\_MAX;

int shortestJobComplete = 0;

while (completed != n) {

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

if (processes[i] <= currentTime && remainingTime[i] < minBurstTime && remainingTime[i] > 0) {

minIndex = i;

minBurstTime = remainingTime[i];

shortestJobComplete = 1;

}

}

if (shortestJobComplete == 0) {

currentTime++;

continue;

}

remainingTime[minIndex]--;

if (remainingTime[minIndex] == 0) {

completed++;

waitingTime[minIndex] = currentTime + 1 - burstTime[minIndex] - processes[minIndex];

if (waitingTime[minIndex] < 0)

waitingTime[minIndex] = 0;

}

currentTime++;

minBurstTime = INT\_MAX;

shortestJobComplete = 0;

}

}

void calculateTurnaroundTime(int burstTime[], int waitingTime[], int n, int turnaroundTime[]) {

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

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

}

}

void calculateAverageTimes(int processes[], int n, int burstTime[]) {

int waitingTime[n], turnaroundTime[n];

int totalWaitingTime = 0, totalTurnaroundTime = 0;

calculateWaitingTime(processes, n, burstTime, waitingTime);

calculateTurnaroundTime(burstTime, waitingTime, n, turnaroundTime);

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

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

totalWaitingTime += waitingTime[i];

totalTurnaroundTime += turnaroundTime[i];

printf("%d\t%d\t\t%d\t\t%d\n", processes[i], burstTime[i], waitingTime[i], turnaroundTime[i]);

}

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

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

}

int main() {

int n = 3;

int processes[] = {0, 1, 2};

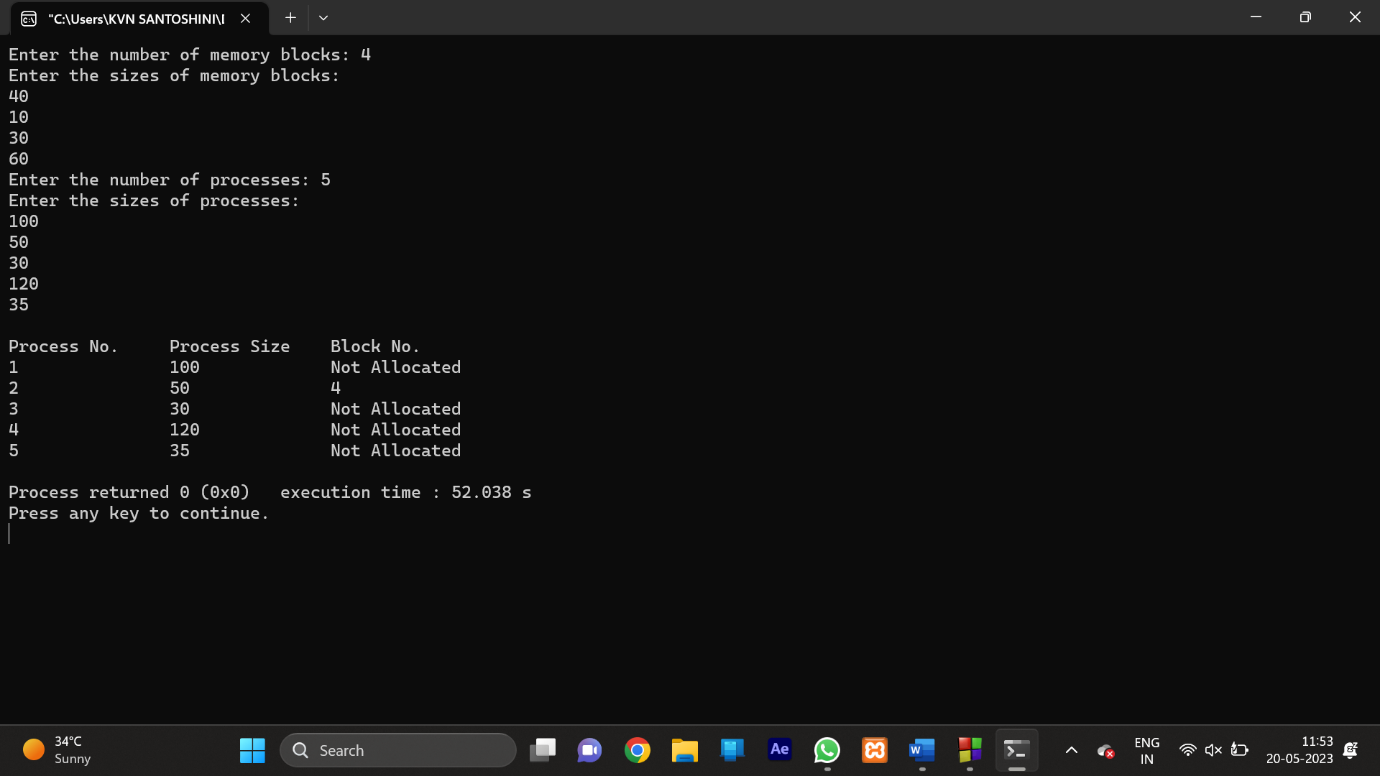
int burstTime[] = {2, 4, 8};

calculateAverageTimes(processes, n, burstTime);

return 0;

}

Output:-



39. Write a C program to simulate LOOK 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>

void lookDiskScheduling(int \*requests, int n, int head, int direction, int \*totalHeadMovement) {

int i, j;

int currentHead = head;

int minIndex = 0, maxIndex = 0;

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

if (requests[i] < requests[minIndex])

minIndex = i;

if (requests[i] > requests[maxIndex])

maxIndex = i;

}

if (direction == -1) {

for (i = head; i >= requests[minIndex]; i--) {

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

if (requests[j] == i) {

\*totalHeadMovement += abs(currentHead - requests[j]);

currentHead = requests[j];

break;

}

}

}

for (i = requests[minIndex] + 1; i <= requests[maxIndex]; i++) {

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

if (requests[j] == i) {

\*totalHeadMovement += abs(currentHead - requests[j]);

currentHead = requests[j];

break;

}

}

}

}

else if (direction == 1) {

for (i = head; i <= requests[maxIndex]; i++) {

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

if (requests[j] == i) {

\*totalHeadMovement += abs(currentHead - requests[j]);

currentHead = requests[j];

break;

}

}

}

for (i = requests[maxIndex] - 1; i >= requests[minIndex]; i--) {

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

if (requests[j] == i) {

\*totalHeadMovement += abs(currentHead - requests[j]);

currentHead = requests[j];

break;

}

}

}

}

}

int main() {

int n, head, direction;

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

scanf("%d", &n);

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

printf("Enter the disk requests: ");

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

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

}

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

scanf("%d", &head);

printf("Enter the direction (left = -1, right = 1): ");

scanf("%d", &direction);

int totalHeadMovement = 0;

lookDiskScheduling(requests, n, head, direction, &totalHeadMovement);

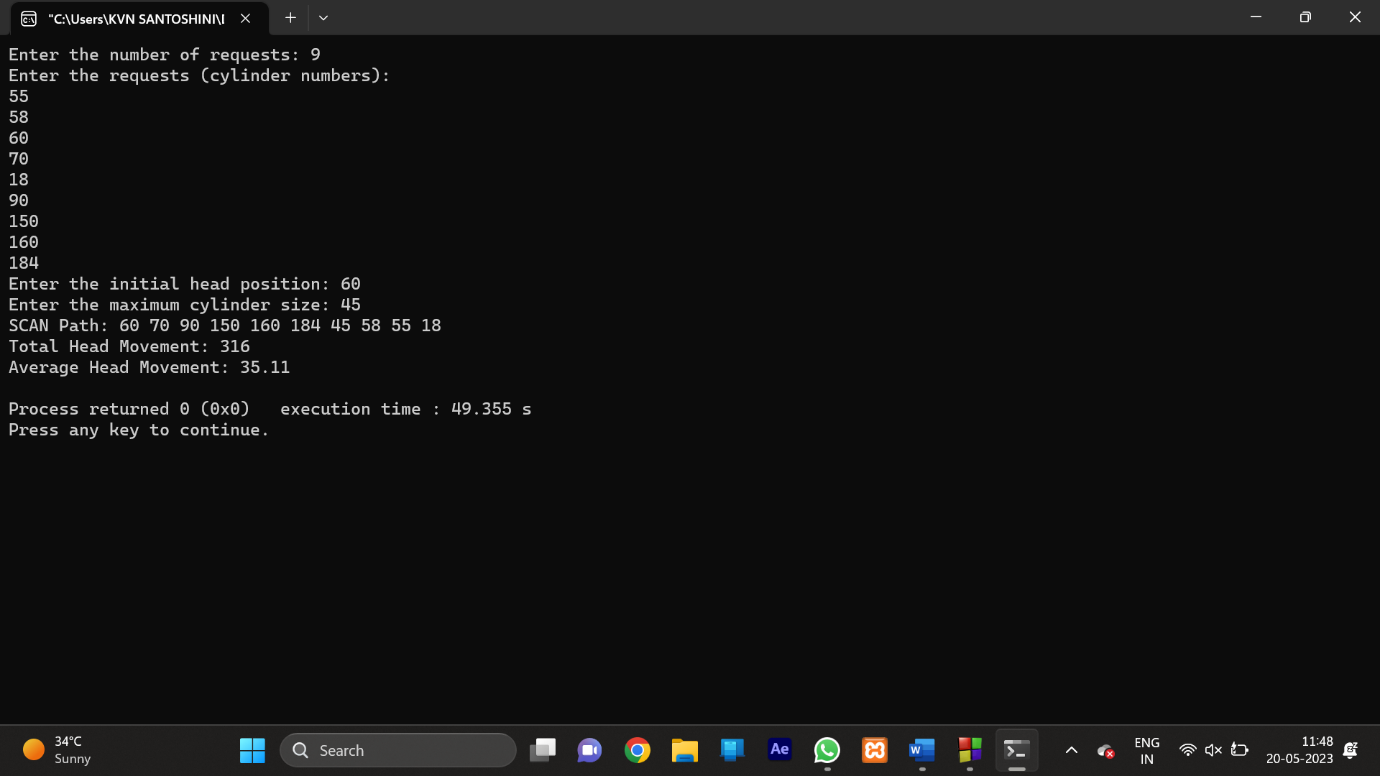
printf("Average head movement: %.2f\n", (float)totalHeadMovement / n);

free(requests);

return 0;

}

Output:-



40. Write a C program to simulate CLOOK 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>

void clookDiskScheduling(int \*requests, int n, int head, int \*totalHeadMovement) {

int i, j;

int currentHead = head;

int minIndex = 0, maxIndex = 0;

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

if (requests[i] < requests[minIndex])

minIndex = i;

if (requests[i] > requests[maxIndex])

maxIndex = i;

}

for (i = head; i <= requests[maxIndex]; i++) {

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

if (requests[j] == i) {

\*totalHeadMovement += abs(currentHead - requests[j]);

currentHead = requests[j];

break;

}

}

}

\*totalHeadMovement += abs(currentHead - 0);

currentHead = 0;

for (i = 0; i <= requests[minIndex]; i++) {

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

if (requests[j] == i) {

\*totalHeadMovement += abs(currentHead - requests[j]);

currentHead = requests[j];

break;

}

}

}

}

int main() {

int n, head;

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

scanf("%d", &n);

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

printf("Enter the disk requests: ");

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

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

}

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

scanf("%d", &head);

int totalHeadMovement = 0;

clookDiskScheduling(requests, n, head, &totalHeadMovement);

printf("Average head movement: %.2f\n", (float)totalHeadMovement / n);

free(requests);

return 0;

}

Output:-

