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

**Program**:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_MEM\_SIZE 1024 // Maximum memory size

struct Process {

int pid;

int size;

int allocated;

};

struct MemoryBlock {

int start;

int size;

int allocated;

};

int main() {

int n, m, i, j;

struct Process \*processes;

struct MemoryBlock \*memory\_blocks;

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

scanf("%d", &n);

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

scanf("%d", &m);

processes = (struct Process\*)malloc(n \* sizeof(struct Process));

memory\_blocks = (struct MemoryBlock\*)malloc(m \* sizeof(struct MemoryBlock));

// Initialize memory blocks

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

printf("Enter the start address and size of memory block %d: ", i+1);

scanf("%d %d", &memory\_blocks[i].start, &memory\_blocks[i].size);

memory\_blocks[i].allocated = 0;

}

// Initialize processes

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

printf("Enter the size of process %d: ", i+1);

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

processes[i].pid = i+1;

processes[i].allocated = 0;

}

// Best-fit algorithm

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

int best\_fit = -1;

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

if(memory\_blocks[j].allocated == 0 && memory\_blocks[j].size >= processes[i].size) {

if(best\_fit == -1 || memory\_blocks[j].size < memory\_blocks[best\_fit].size) {

best\_fit = j;

}

}

}

if(best\_fit != -1) {

memory\_blocks[best\_fit].allocated = 1;

memory\_blocks[best\_fit].size -= processes[i].size;

processes[i].allocated = memory\_blocks[best\_fit].start;

}

}

// Print allocation

printf("\nProcess\tSize\tBlock\tStart Address\n");

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

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

if(processes[i].allocated != 0) {

printf("%d\t%d\n", processes[i].allocated, memory\_blocks[processes[i].allocated-1].start);

} else {

printf("Not allocated\n");

}

}

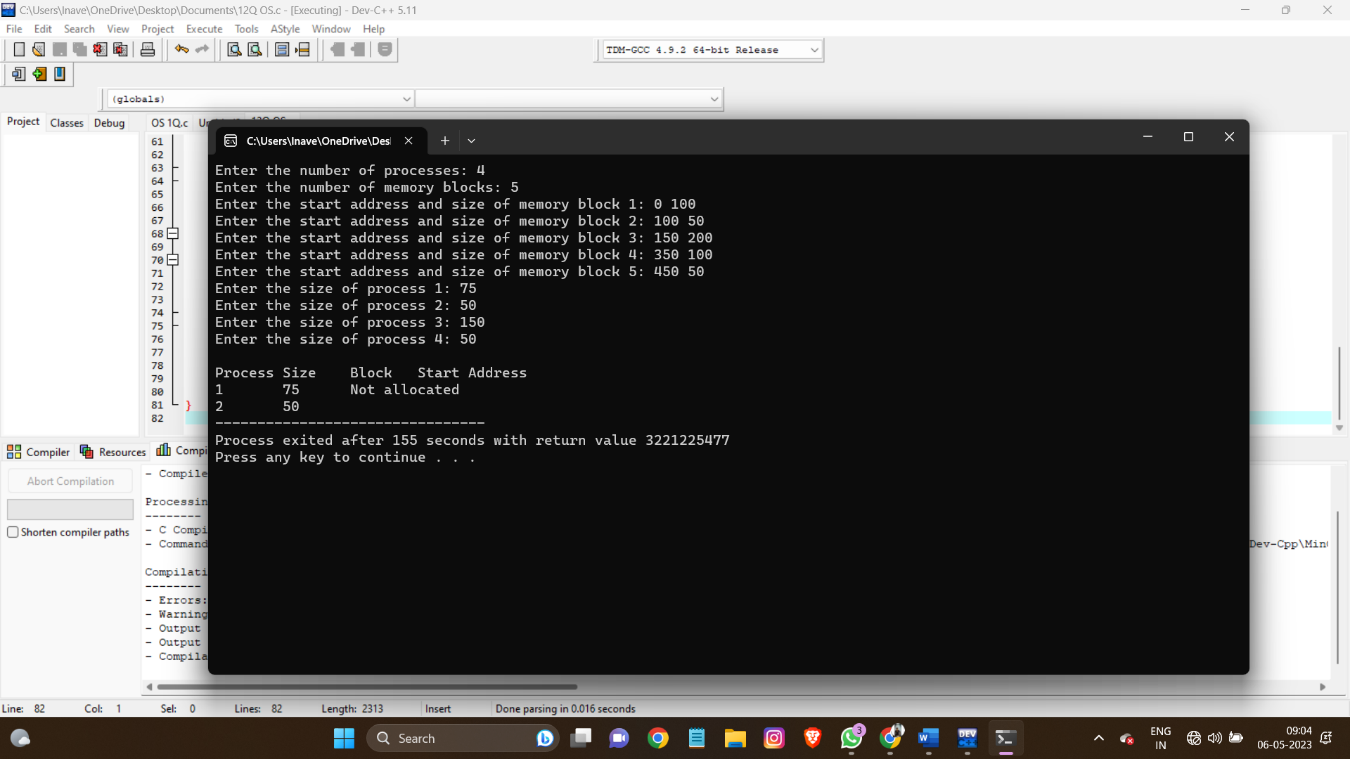
free(processes);

free(memory\_blocks);

return 0;

}

**OUTPUT**

****

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>

#include <string.h>

#define MAX\_FILES 100 // Maximum number of files

#define MAX\_FILENAME\_LENGTH 20 // Maximum length of filename

struct File {

char name[MAX\_FILENAME\_LENGTH];

};

struct Directory {

char name[MAX\_FILENAME\_LENGTH];

struct File files[MAX\_FILES];

int num\_files;

};

int main() {

struct Directory dir;

int i;

// Create directory

strcpy(dir.name, "CSE");

dir.num\_files = 0;

// Add files to directory

strcpy(dir.files[0].name, "A");

strcpy(dir.files[1].name, "B");

strcpy(dir.files[2].name, "C");

dir.num\_files = 3;

// Print directory contents

printf("Directory: %s\n", dir.name);

printf("Files:\n");

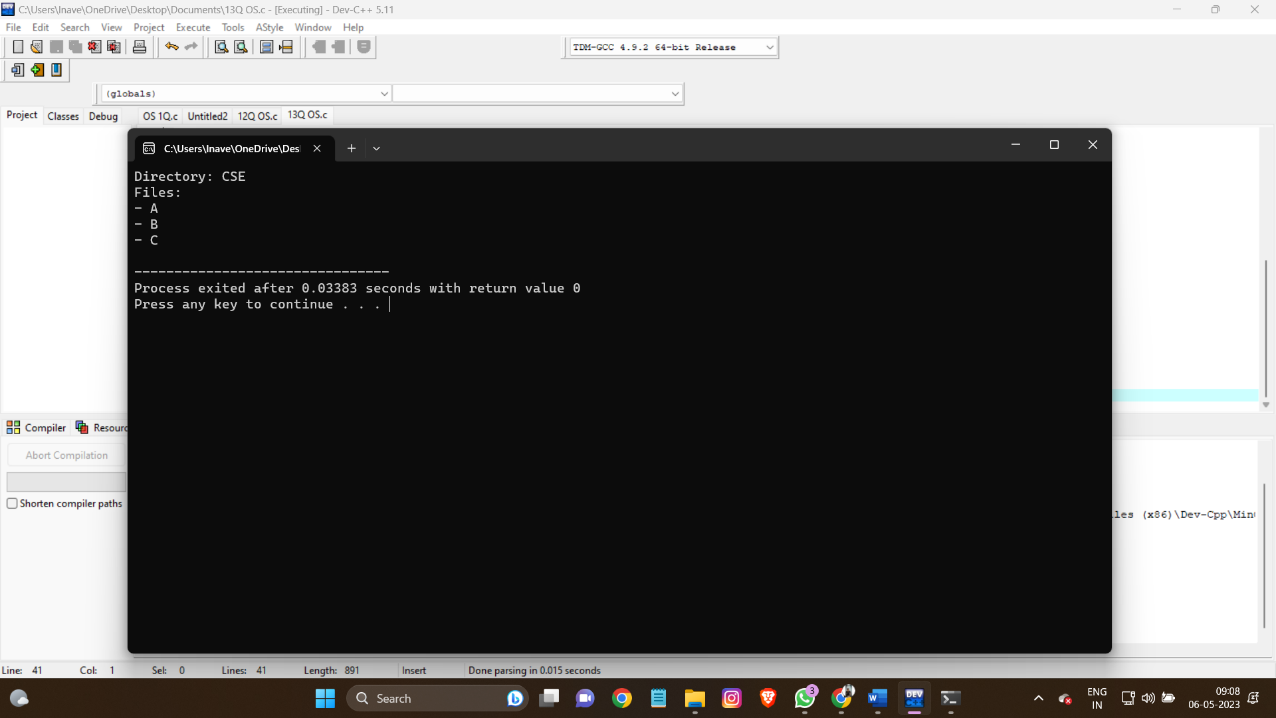
for(i = 0; i < dir.num\_files; i++) {

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

}

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>

#include <stdbool.h>

#define MAX\_PAGE\_FRAMES 100 // Maximum number of page frames

int find\_page(int page\_frames[], int page\_frame\_count, int page) {

int i;

for(i = 0; i < page\_frame\_count; i++) {

if(page\_frames[i] == page) {

return i;

}

}

return -1;

}

int find\_replace\_page(int page\_frames[], int page\_frame\_count, int pages[], int page\_count, int current\_index) {

int i, j, max\_future\_index, max\_future\_page;

bool future\_use[MAX\_PAGE\_FRAMES] = {false};

for(i = current\_index + 1; i < page\_count; i++) {

for(j = 0; j < page\_frame\_count; j++) {

if(pages[i] == page\_frames[j]) {

future\_use[j] = true;

break;

}

}

}

max\_future\_index = -1;

max\_future\_page = -1;

for(i = 0; i < page\_frame\_count; i++) {

if(!future\_use[i]) {

if(max\_future\_index == -1 || max\_future\_index < find\_page(pages, page\_count, page\_frames[i])) {

max\_future\_index = find\_page(pages, page\_count, page\_frames[i]);

max\_future\_page = i;

}

}

}

return max\_future\_page;

}

int main() {

int page\_frames[MAX\_PAGE\_FRAMES], pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};

int page\_frame\_count = 3, page\_count = 20, page\_faults = 0, i, j, k, index;

bool page\_fault;

// Initialize page frames to -1

for(i = 0; i < page\_frame\_count; i++) {

page\_frames[i] = -1;

}

// Loop through each page

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

page\_fault = true;

// Check if page is already in a page frame

index = find\_page(page\_frames, page\_frame\_count, pages[i]);

if(index == -1) {

// Find a page to replace

index = find\_replace\_page(page\_frames, page\_frame\_count, pages, page\_count, i);

// Replace page

page\_frames[index] = pages[i];

page\_faults++;

} else {

// Page hit

page\_fault = false;

}

// Print page frames after each page

printf("Page %d: ", pages[i]);

for(j = 0; j < page\_frame\_count; j++) {

if(page\_frames[j] == -1) {

printf("- ");

} else {

printf("%d ", page\_frames[j]);

}

}

if(page\_fault) {

printf("FAULT\n");

} else {

printf("\n");

}

}

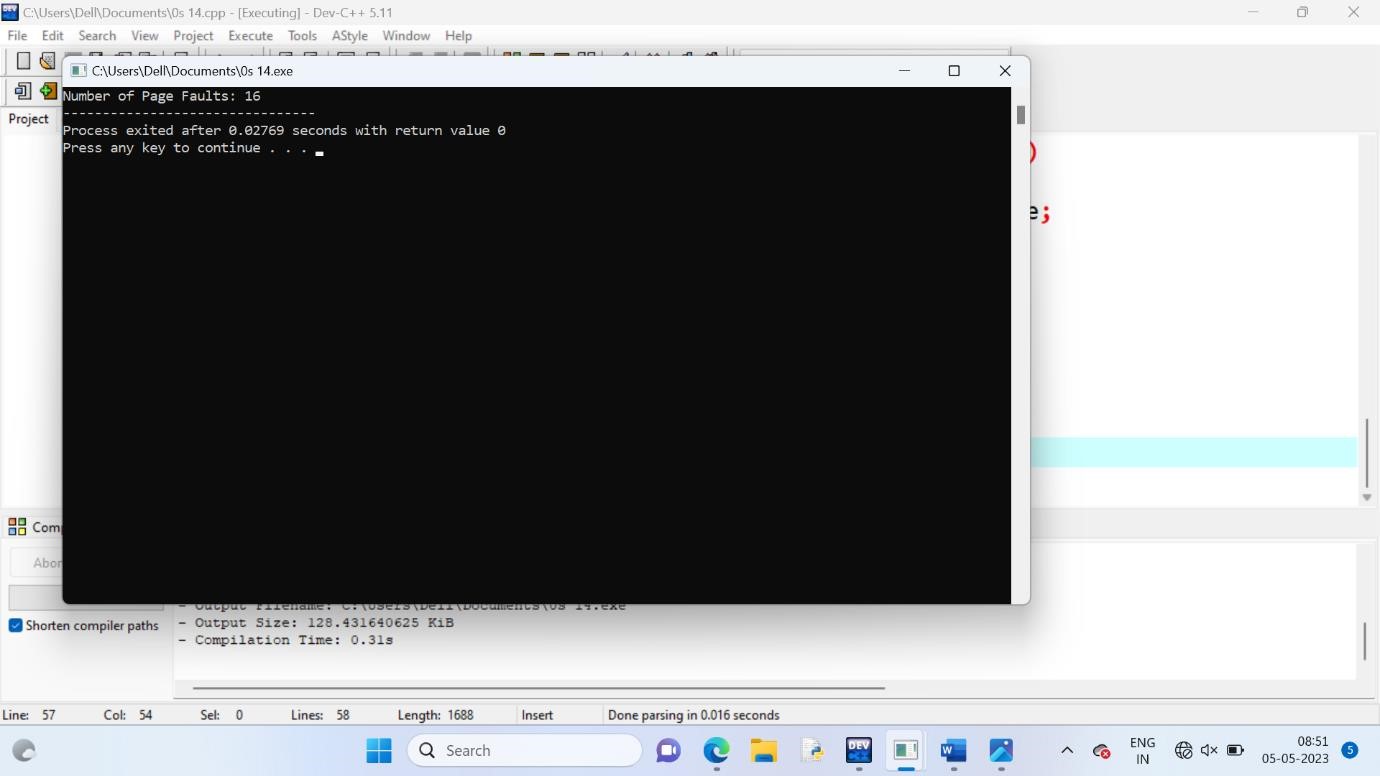
// Print total number of page faults

printf("Total page faults: %d\n", 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>

int main() {

int n = 9;

int tracks[] = {55, 58, 60, 70, 18, 90, 150, 160, 184};

int head\_pos = 50; // starting position of the head

int total\_head\_movement = 0;

printf("FCFS disk scheduling algorithm\n");

printf("Initial head position: %d\n", head\_pos);

printf("Track sequence: ");

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

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

total\_head\_movement += abs(head\_pos - tracks[i]);

head\_pos = tracks[i];

}

printf("\n");

float avg\_head\_movement = (float)total\_head\_movement / n;

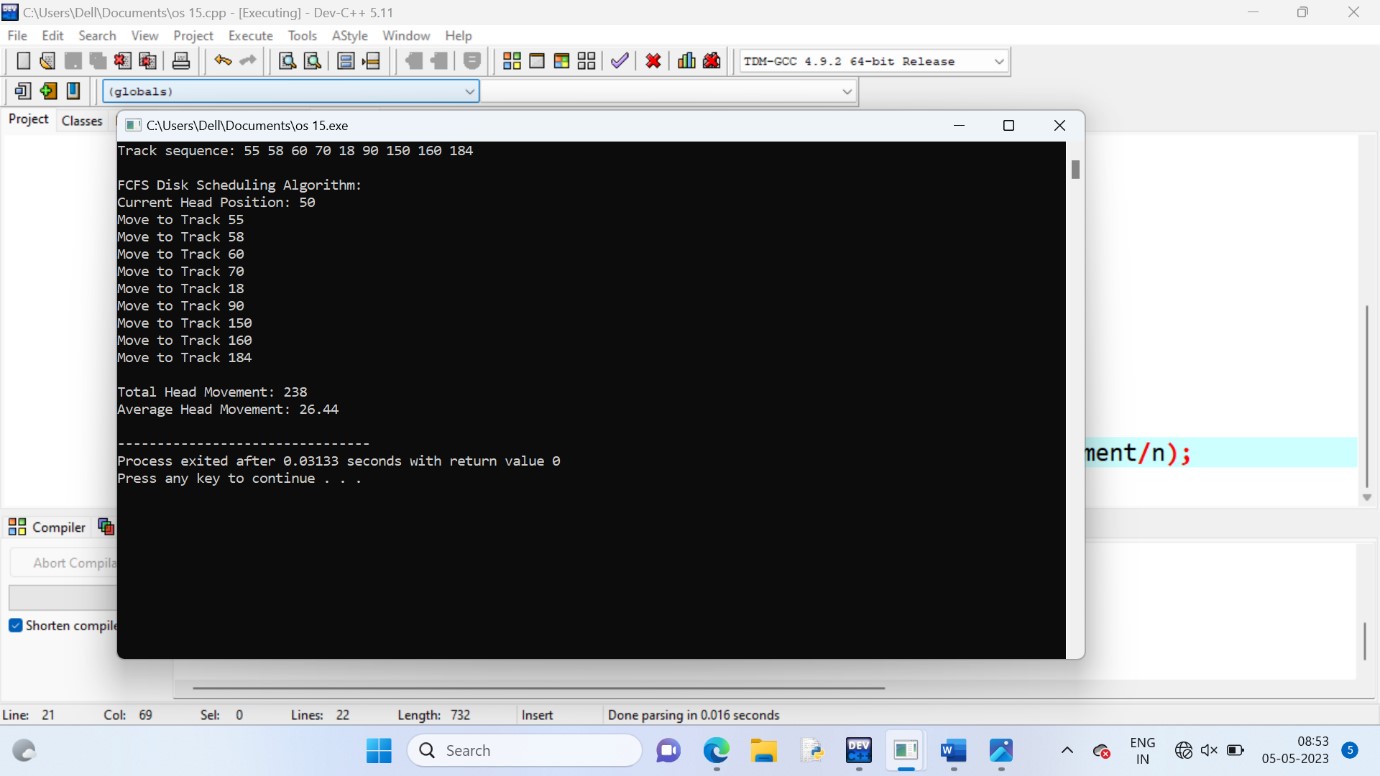
printf("Total head movement: %d\n", total\_head\_movement);

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.)

**PROGRAM**

#include <stdio.h>

int main() {

int n = 3; // number of processes

int burst\_times[3] = {10, 15, 25}; // burst times of the processes

int arrival\_time = 0; // arrival time of all processes

int waiting\_time[n], turnaround\_time[n]; // arrays to store waiting and turnaround times

float avg\_waiting\_time = 0, avg\_turnaround\_time = 0; // variables to store the averages

waiting\_time[0] = 0; // waiting time of the first process is 0

// calculate waiting and turnaround times for the remaining processes

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

waiting\_time[i] = waiting\_time[i - 1] + burst\_times[i - 1]; // waiting time is the sum of the previous burst times

}

// calculate turnaround times and the averages

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

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

avg\_waiting\_time += waiting\_time[i];

avg\_turnaround\_time += turnaround\_time[i];

}

avg\_waiting\_time /= n;

avg\_turnaround\_time /= n;

// print the results

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

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

printf("P%d\t%d\t\t%d\t\t%d\n", i + 1, burst\_times[i], waiting\_time[i], turnaround\_time[i]);

}

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

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

return 0;

}

**OUTPUT**

