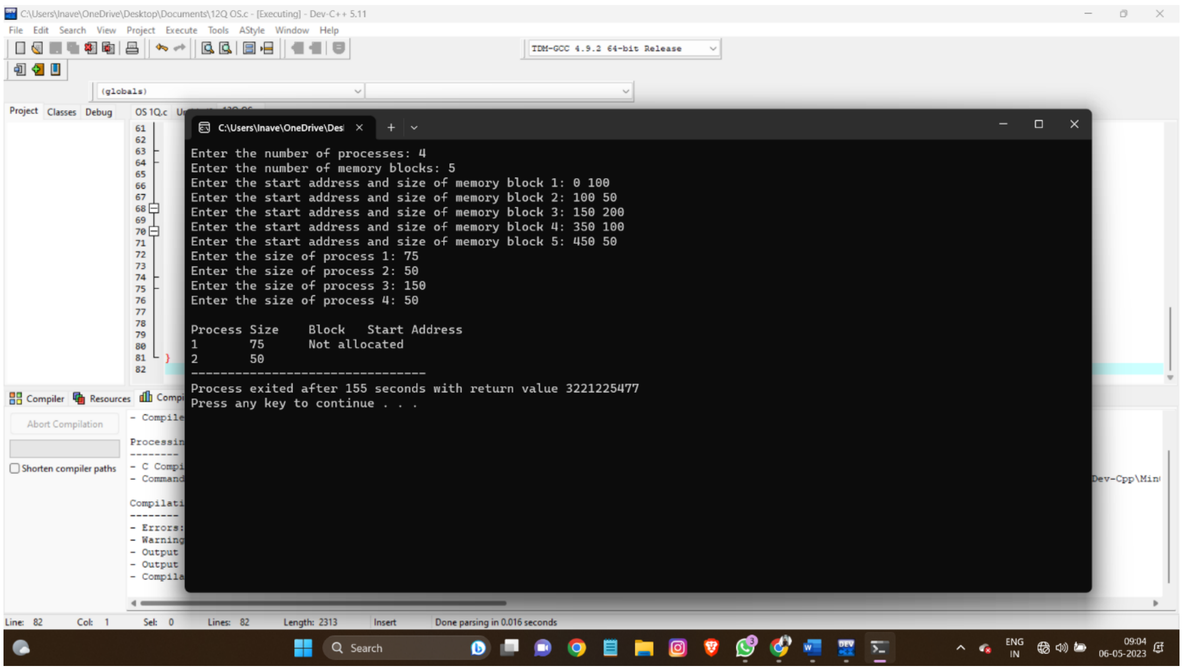
12. Write a C program to implement the best-fit algorithm and allocate the memory block to each process.

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



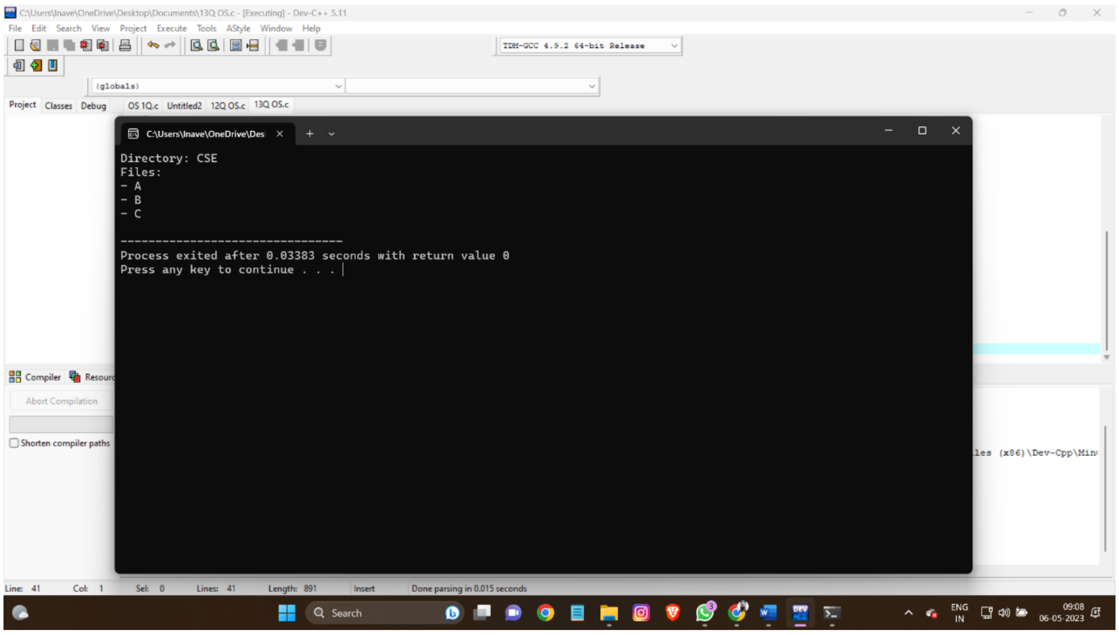
```
Enter the number of processes: 4
Enter the number of memory blocks: 5
Enter the start address and size of memory block 1: 0 100
Enter the start address and size of memory block 2: 100 50
Enter the start address and size of memory block 3: 150 200
Enter the start address and size of memory block 4: 350 100
Enter the start address and size of memory block 5: 450 50
Enter the size of process 1: 75
Enter the size of process 2: 50
Enter the size of process 3: 150
Enter the size of process 4: 50

Process Size    Block   Start Address
1       75      Not allocated
2       50
--------------------------------
Process exited after 155 seconds with return value 3221225477
Press any key to continue . . .
```

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:

```c
#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:



```
Directory: CSE
Files:
- A
- B
- C

--------------------------------
Process exited after 0.03383 seconds with return value 0
Press any key to continue . . .
```

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.

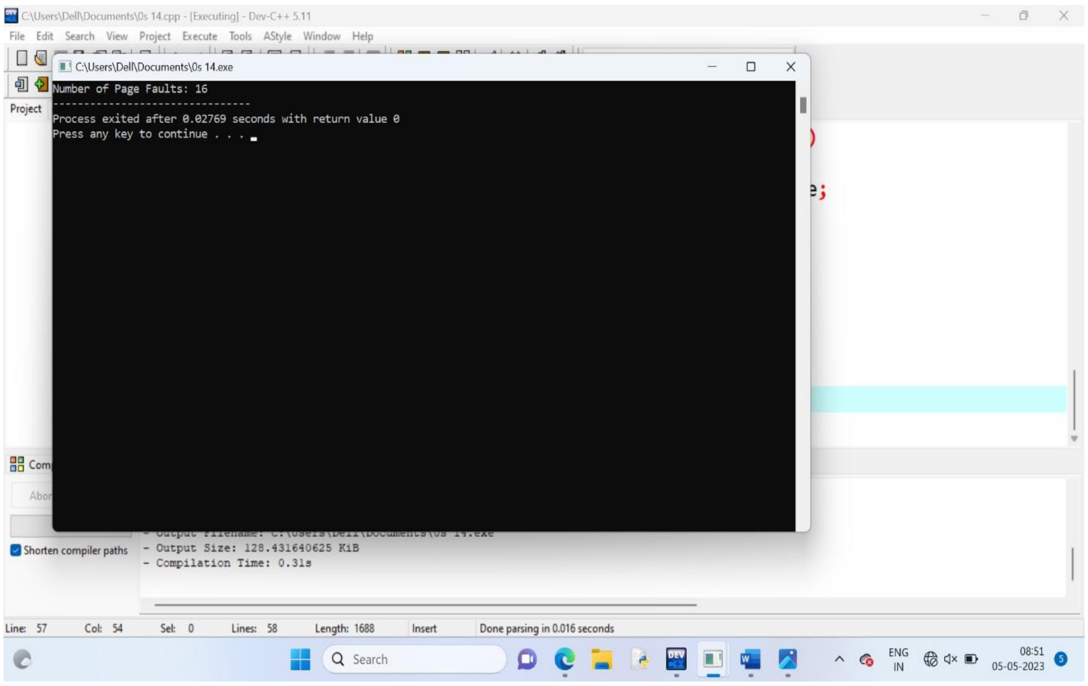
PROGRAM

```
#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:



```
C:\Users\Dell\Documents\0s 14.exe
Number of Page Faults: 16
--------------------------------
Process exited after 0.02769 seconds with return value 0
Press any key to continue . . .
```

```
- Output Size: 128.431640625 KiB
- Compilation Time: 0.31s
```

C:\Users\Dell\Documents\0s 14.cpp - [Executing] - Dev-C++ 5.11
File Edit Search View Project Execute Tools AStyle Window Help
Project
Shorten compiler paths
Line: 57 Col: 54 Sel: 0 Lines: 58 Length: 1688 Insert Done parsing in 0.016 seconds
Search
ENG IN
08:51 05-05-2023

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.

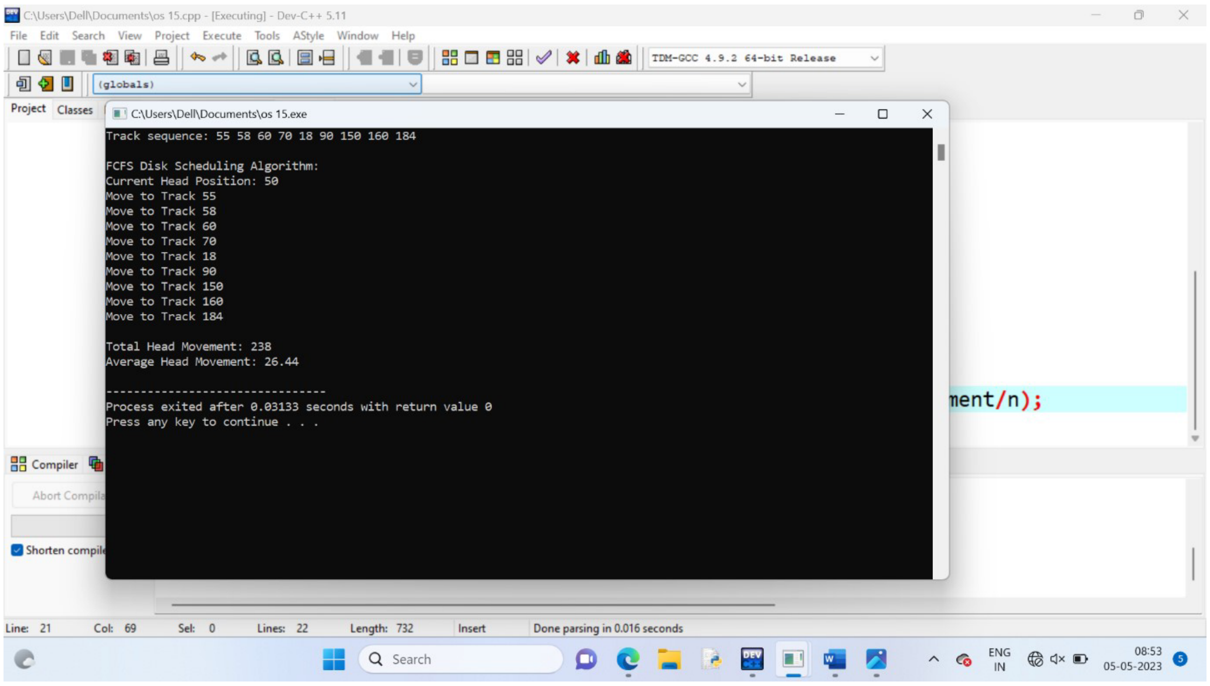
No of tracks:9; Track position:55 58 60 70 18 90 150 160 184

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



```
Track sequence: 55 58 60 70 18 90 150 160 184

FCFS Disk Scheduling Algorithm:
Current Head Position: 50
Move to Track 55
Move to Track 58
Move to Track 60
Move to Track 70
Move to Track 18
Move to Track 90
Move to Track 150
Move to Track 160
Move to Track 184

Total Head Movement: 238
Average Head Movement: 26.44

--------------------------------
Process exited after 0.03133 seconds with return value 0
Press any key to continue . . .
```

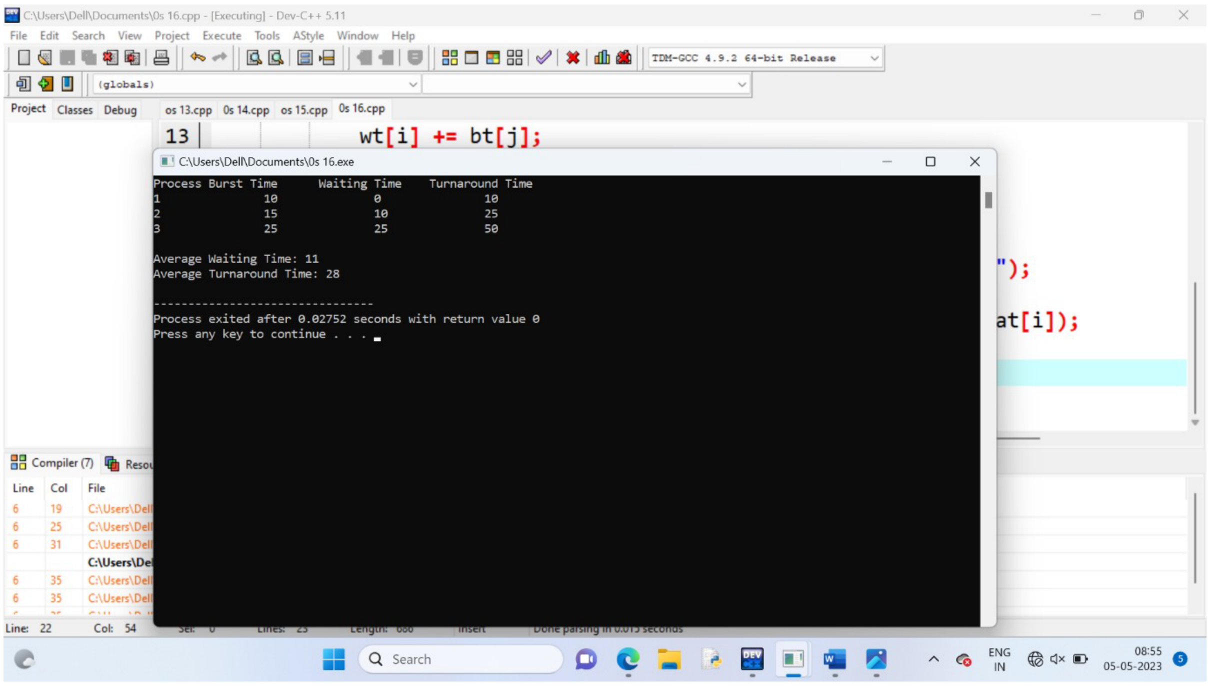
16. Write a program to compute the average waiting time and average turnaround time based on First Come First Serve for the following process with the given CPU burst times, (and the assumption that all jobs arrive at the same time.)

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



```
Process Burst Time      Waiting Time    Turnaround Time
1               10              0               10
2               15              10              25
3               25              25              50

Average Waiting Time: 11
Average Turnaround Time: 28

--------------------------------
Process exited after 0.02752 seconds with return value 0
Press any key to continue . . .
```