**DAY-5**

**33.** Construct a C program to simulate the optimal paging technique of memory management

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#define MAX\_FRAMES 3 // Maximum number of frames in memory

#define MAX\_PAGES 20 // Maximum number of pages referenced

int main() {

int reference\_string[MAX\_PAGES] = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5}; // Reference string

int num\_pages = 12; // Number of pages in the reference string

int frames[MAX\_FRAMES]; // Array to store the frames in memory

int page\_faults = 0; // Counter for page faults

int i, j, k;

// Initialize frames as empty (-1 indicates no page loaded)

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

frames[i] = -1;

printf("Reference String: ");

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

printf("%d ", reference\_string[i]);

printf("\n");

// Simulating the optimal page replacement algorithm

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

int page = reference\_string[i];

int page\_found = 0;

// Check if page is already in frames

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

if (frames[j] == page) {

page\_found = 1;

break;

}

}

// If page is not in frames, replace a page using optimal replacement strategy

if (!page\_found) {

page\_faults++;

// Find the page that will not be used for the longest time in the future

int farthest = i;

int replace\_page = -1;

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

int current\_page = frames[j];

int found\_future = 0;

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

if (current\_page == reference\_string[k]) {

found\_future = 1;

if (k > farthest) {

farthest = k;

replace\_page = j;

}

break;

}

}

if (!found\_future) {

replace\_page = j;

break;

}

}

// Replace the selected page

frames[replace\_page] = page;

}

// Print current frames

printf("Frames: ");

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

if (frames[j] == -1)

printf("- ");

else

printf("%d ", frames[j]);

}

printf("\n");

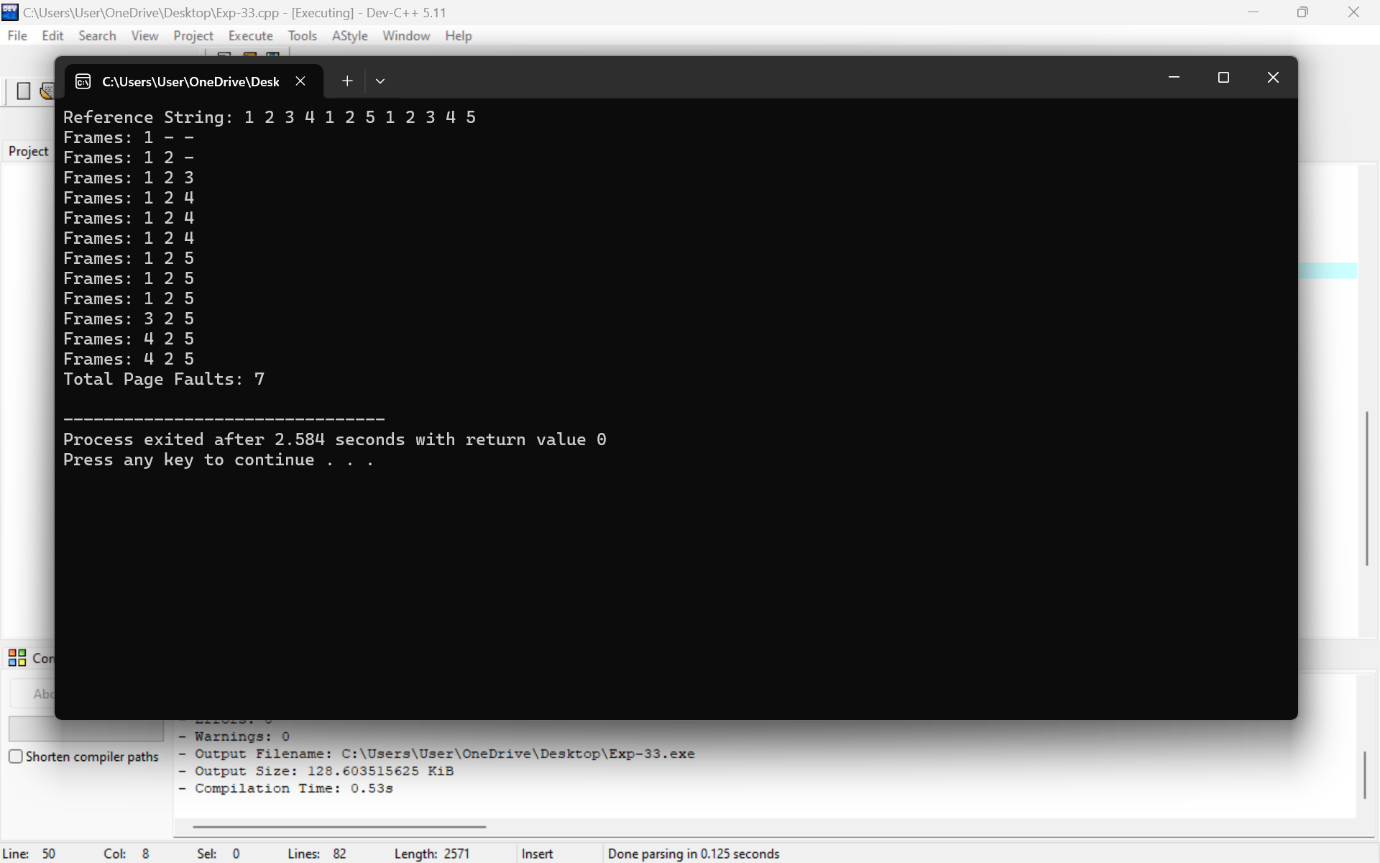
}

printf("Total Page Faults: %d\n", page\_faults);

return 0;

}

OUTPUT



**34.** Consider a file system where the records of the file are stored one after another both physically and logically. A record of the file can only be accessed by reading all the previous records. Design a C program to simulate the file allocation strategy.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_RECORDS 100 // Maximum number of records

#define MAX\_LENGTH 100 // Maximum length of each record

struct File {

char records[MAX\_RECORDS][MAX\_LENGTH];

int num\_records;

};

// Function to initialize the file

void initializeFile(struct File \*file) {

file->num\_records = 0;

}

// Function to add a record to the file

void addRecord(struct File \*file, const char \*record) {

if (file->num\_records < MAX\_RECORDS) {

strcpy(file->records[file->num\_records], record);

file->num\_records++;

printf("Record added successfully.\n");

} else {

printf("File is full. Cannot add more records.\n");

}

}

// Function to display all records in the file

void displayFile(struct File \*file) {

printf("Records in the file:\n");

for (int i = 0; i < file->num\_records; ++i) {

printf("%d: %s\n", i + 1, file->records[i]);

}

}

int main() {

struct File file;

initializeFile(&file);

int choice;

char record[MAX\_LENGTH];

do {

printf("\nFile Allocation Strategy Simulation\n");

printf("1. Add a record\n");

printf("2. Display all records\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter record to add: ");

getchar(); // Clearing the input buffer

fgets(record, MAX\_LENGTH, stdin);

record[strcspn(record, "\n")] = '\0'; // Remove trailing newline character

addRecord(&file, record);

break;

case 2:

displayFile(&file);

break;

case 3:

printf("Exiting the program.\n");

break;

default:

printf("Invalid choice. Please enter again.\n");

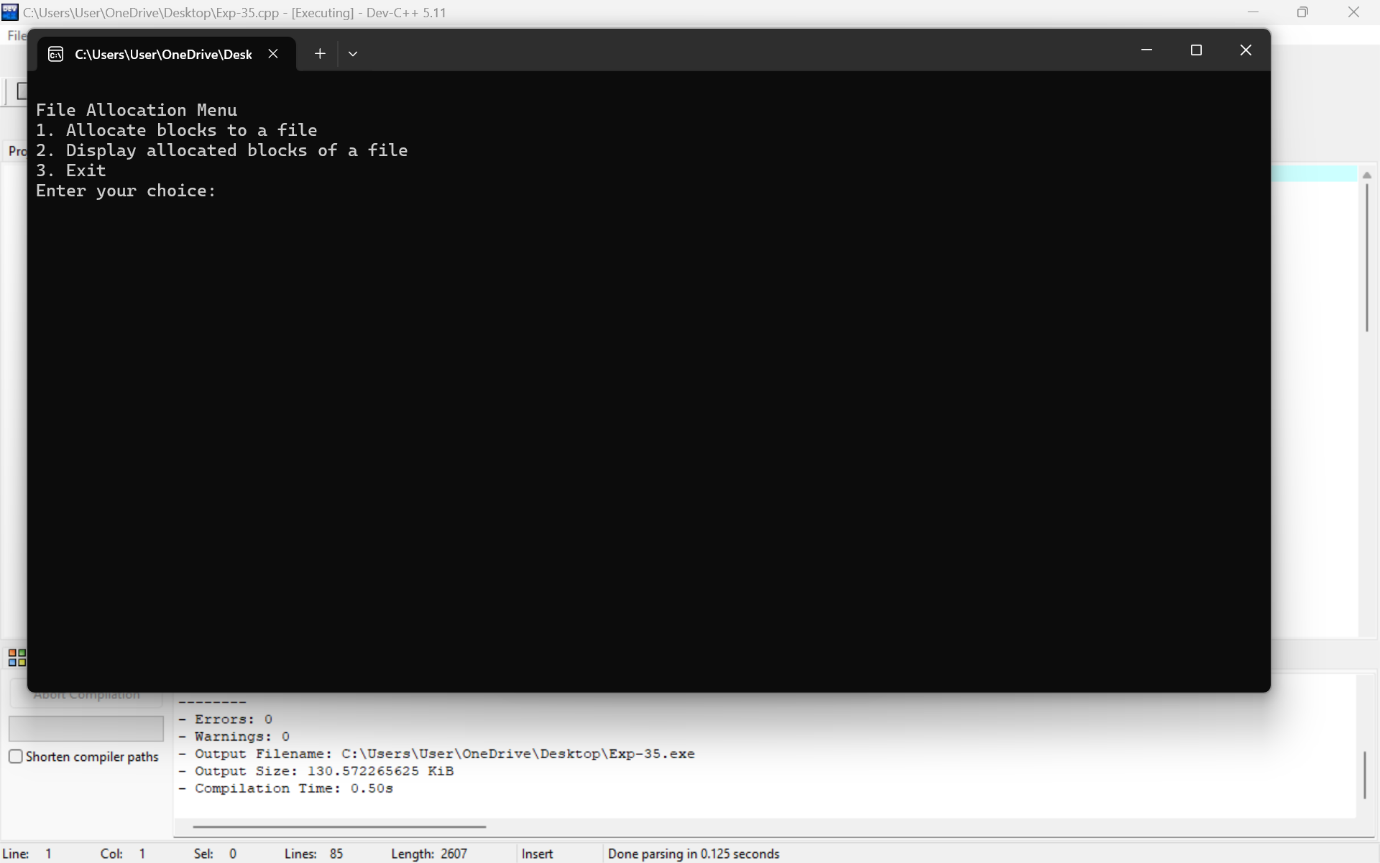
}

} while (choice != 3);

return 0;

}

OUTPUT



35. Consider a file system that brings all the file pointers together into an index block. The ith entry in the index block points to the ith block of the file. Design a C program to simulate the file allocation strategy.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#define MAX\_BLOCKS 100

#define MAX\_FILES 10

// Structure to represent a file

struct File {

int indexBlock[MAX\_BLOCKS]; // Array to hold index block entries

int numBlocks; // Number of blocks allocated to the file

};

// Function to initialize a file with empty index blocks

void initializeFile(struct File \*file) {

file->numBlocks = 0;

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

file->indexBlock[i] = -1; // Initialize all index block entries to -1 (indicating empty)

}

}

// Function to allocate blocks to a file

void allocateBlocks(struct File \*file, int numAllocated) {

if (file->numBlocks + numAllocated > MAX\_BLOCKS) {

printf("Error: File size exceeds maximum block limit.\n");

return;

}

// Allocate blocks by updating index block entries

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

file->indexBlock[file->numBlocks + i] = file->numBlocks + i;

}

file->numBlocks += numAllocated;

}

// Function to display allocated blocks of a file

void displayFileBlocks(struct File \*file) {

printf("Allocated blocks for the file:\n");

for (int i = 0; i < file->numBlocks; ++i) {

printf("%d ", file->indexBlock[i]);

}

printf("\n");

}

int main() {

struct File files[MAX\_FILES];

int choice, fileIndex, numBlocks;

// Initialize all files

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

initializeFile(&files[i]);

}

do {

printf("\nFile Allocation Menu\n");

printf("1. Allocate blocks to a file\n");

printf("2. Display allocated blocks of a file\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter file index (0 to %d): ", MAX\_FILES - 1);

scanf("%d", &fileIndex);

printf("Enter number of blocks to allocate: ");

scanf("%d", &numBlocks);

allocateBlocks(&files[fileIndex], numBlocks);

break;

case 2:

printf("Enter file index (0 to %d): ", MAX\_FILES - 1);

scanf("%d", &fileIndex);

displayFileBlocks(&files[fileIndex]);

break;

case 3:

printf("Exiting program.\n");

break;

default:

printf("Invalid choice. Please enter again.\n");

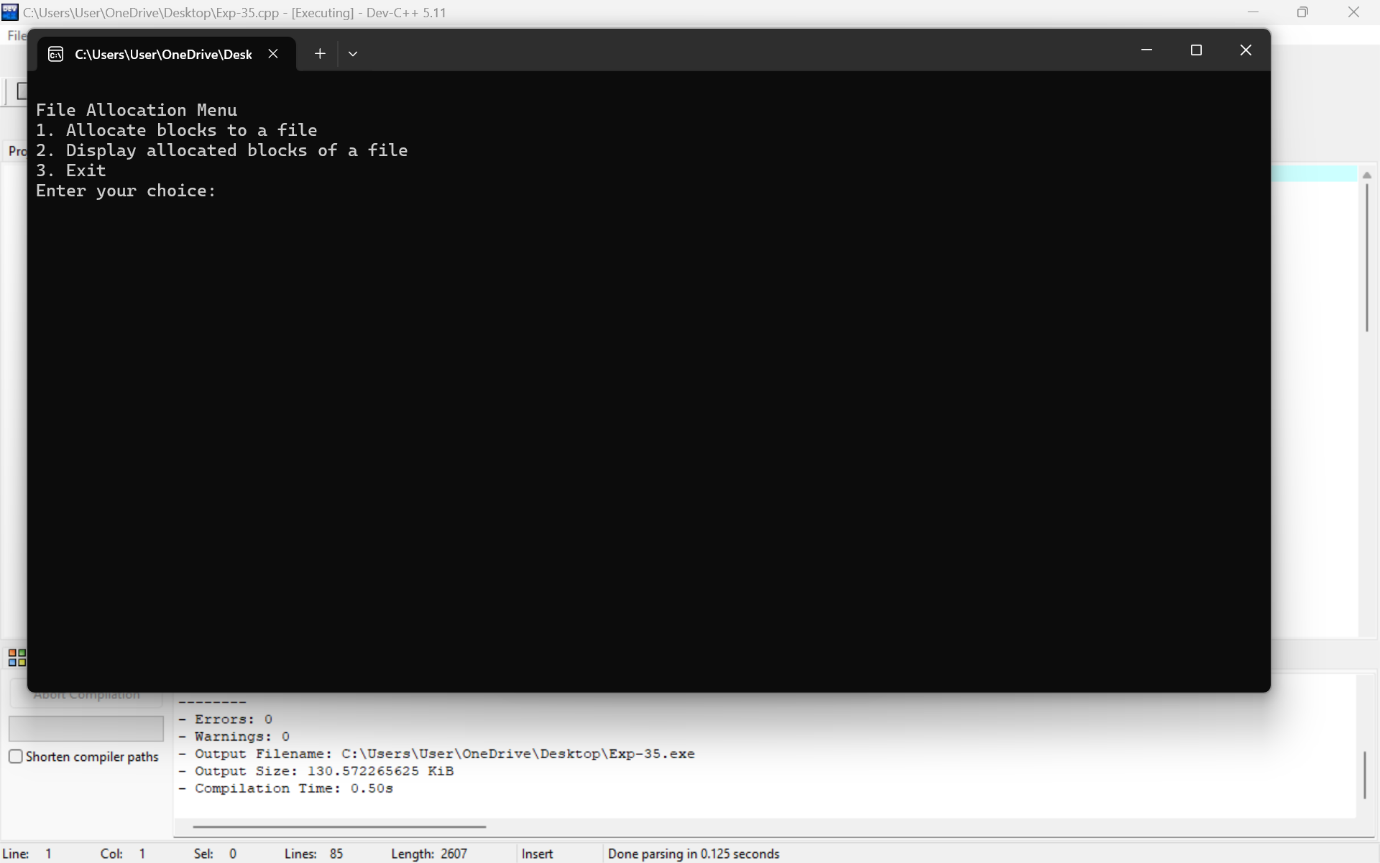
}

} while (choice != 3);

return 0;

}

OUTPUT



36. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block. Design a C program to simulate the file allocation strategy.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#define MAX\_BLOCKS 100

#define MAX\_FILES 10

// Structure to represent a block on disk

struct DiskBlock {

int data; // Data stored in the block (for simplicity)

struct DiskBlock \*next; // Pointer to the next block in the file

};

// Structure to represent a file

struct File {

struct DiskBlock \*firstBlock; // Pointer to the first block of the file

struct DiskBlock \*lastBlock; // Pointer to the last block of the file

};

// Function to initialize a file

void initializeFile(struct File \*file) {

file->firstBlock = NULL;

file->lastBlock = NULL;

}

// Function to add a block to the end of the file

void addBlock(struct File \*file, struct DiskBlock \*block) {

if (file->firstBlock == NULL) {

file->firstBlock = block;

file->lastBlock = block;

block->next = NULL;

} else {

file->lastBlock->next = block;

file->lastBlock = block;

block->next = NULL;

}

}

// Function to display blocks of a file

void displayFileBlocks(struct File \*file) {

printf("Blocks of the file:\n");

struct DiskBlock \*current = file->firstBlock;

while (current != NULL) {

printf("%p -> ", (void \*)current);

current = current->next;

}

printf("NULL\n");

}

int main() {

struct File files[MAX\_FILES];

struct DiskBlock diskBlocks[MAX\_BLOCKS];

int choice, fileIndex;

int blockIndex = 0;

// Initialize all files

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

initializeFile(&files[i]);

}

do {

printf("\nLinked Allocation Menu\n");

printf("1. Allocate blocks to a file\n");

printf("2. Display blocks of a file\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

if (blockIndex >= MAX\_BLOCKS) {

printf("Error: Maximum number of blocks reached.\n");

break;

}

printf("Enter file index (0 to %d): ", MAX\_FILES - 1);

scanf("%d", &fileIndex);

// Allocate a new block

struct DiskBlock \*newBlock = &diskBlocks[blockIndex++];

printf("Allocating block %p to file %d.\n", (void \*)newBlock, fileIndex);

addBlock(&files[fileIndex], newBlock);

break;

case 2:

printf("Enter file index (0 to %d): ", MAX\_FILES - 1);

scanf("%d", &fileIndex);

displayFileBlocks(&files[fileIndex]);

break;

case 3:

printf("Exiting program.\n");

break;

default:

printf("Invalid choice. Please enter again.\n");

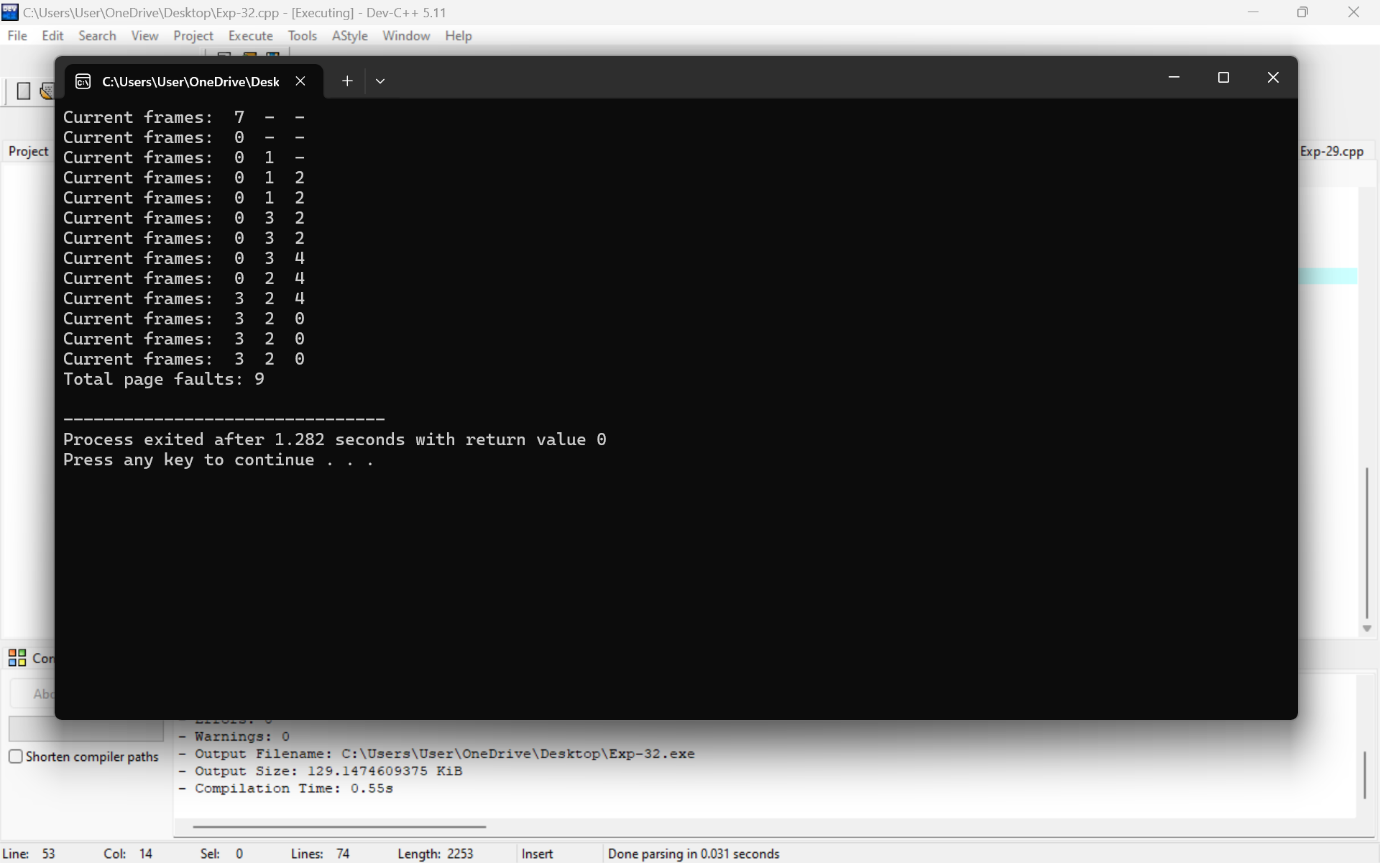
}

} while (choice != 3);

return 0;

}

OUTPUT



37. Construct a C program to simulate the First Come First Served disk scheduling algorithm.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#define MAX\_REQUESTS 100 // Maximum number of disk requests

// Function to simulate FCFS disk scheduling algorithm

void fcfs(int requests[], int num\_requests, int initial\_position) {

int current\_position = initial\_position;

int total\_head\_movement = 0;

printf("FCFS Disk Scheduling:\n");

printf("Initial Head Position: %d\n", initial\_position);

printf("Sequence of head movement:\n");

// Process each request in the order they are received

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

int current\_request = requests[i];

printf("Move from %d to %d\n", current\_position, current\_request);

total\_head\_movement += abs(current\_request - current\_position);

current\_position = current\_request;

}

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

}

int main() {

int requests[MAX\_REQUESTS];

int num\_requests;

int initial\_position;

// Input number of requests

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

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

// Input disk requests

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

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

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

}

// Input initial head position

printf("Enter initial head position: ");

scanf("%d", &initial\_position);

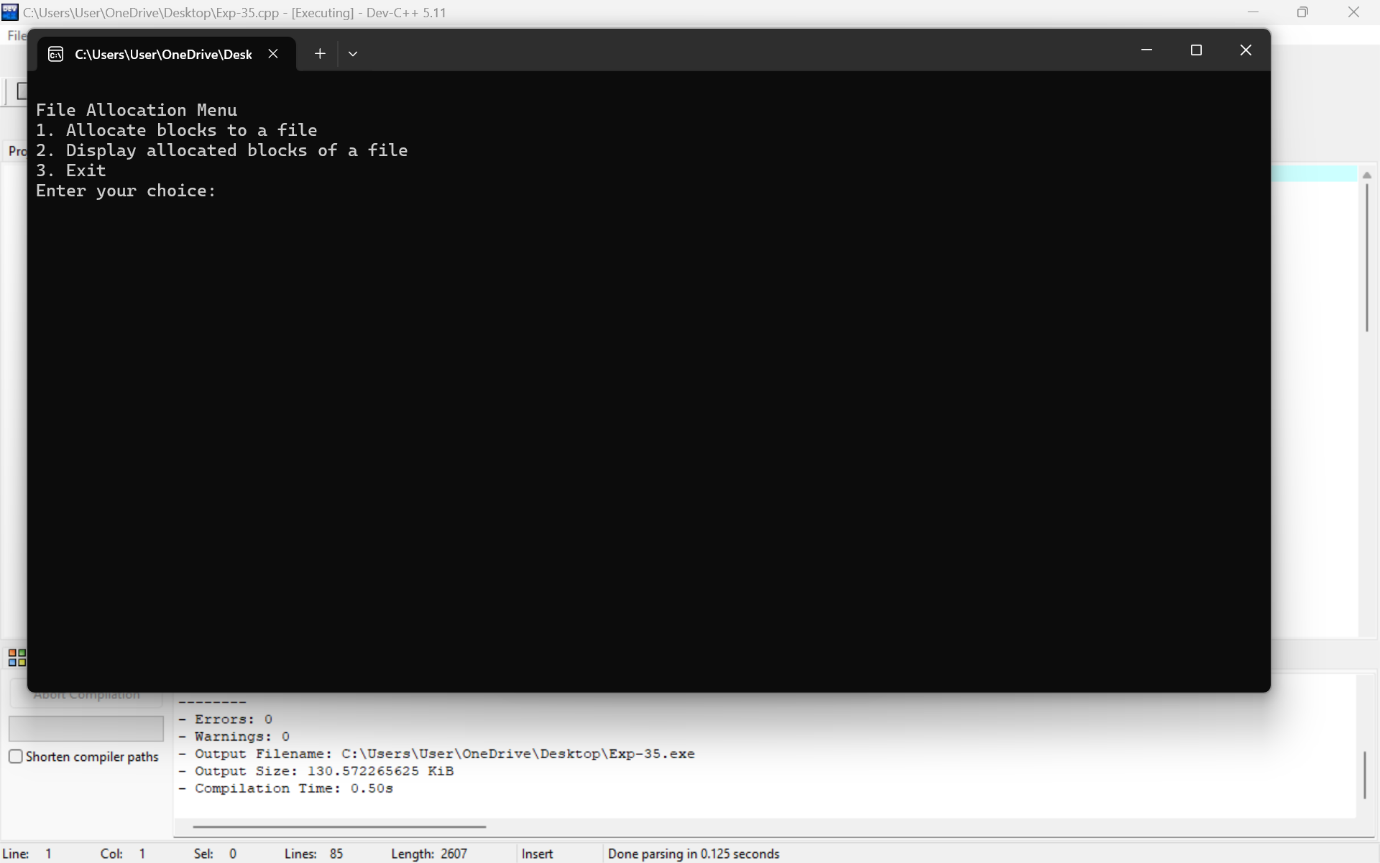
// Perform FCFS scheduling

fcfs(requests, num\_requests, initial\_position);

return 0;

}

OUTPUT



38. Design a C program to simulate SCAN disk scheduling algorithm.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

// Function to sort the request array in ascending order

void sort\_requests(int \*requests, int num\_requests) {

int i, j, temp;

for (i = 0; i < num\_requests - 1; i++) {

for (j = 0; j < num\_requests - i - 1; j++) {

if (requests[j] > requests[j + 1]) {

temp = requests[j];

requests[j] = requests[j + 1];

requests[j + 1] = temp;

}

}

}

}

// Function to simulate the SCAN disk scheduling algorithm

void scan\_disk(int \*requests, int num\_requests, int head\_position, int direction, int disk\_size) {

int total\_movement = 0;

int current\_position = head\_position;

// Sort requests

sort\_requests(requests, num\_requests);

// Find index of current head position in sorted array

int i;

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

if (requests[i] >= head\_position) {

break;

}

}

// If direction is 0, simulate right scan

if (direction == 0) {

printf("Sequence of head movements: %d -> ", current\_position);

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

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

total\_movement += abs(requests[i] - current\_position);

current\_position = requests[i];

}

printf("%d -> %d\n", disk\_size - 1, 0);

total\_movement += disk\_size - 1 - current\_position;

current\_position = 0;

for (i = 0; i < num\_requests && requests[i] < head\_position; i++) {

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

total\_movement += abs(requests[i] - current\_position);

current\_position = requests[i];

}

} else {

// If direction is 1, simulate left scan

printf("Sequence of head movements: %d -> ", current\_position);

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

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

total\_movement += abs(requests[i] - current\_position);

current\_position = requests[i];

}

printf("0 -> ");

total\_movement += current\_position;

current\_position = 0;

for (i = num\_requests - 1; i >= 0 && requests[i] > head\_position; i--) {

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

total\_movement += abs(requests[i] - current\_position);

current\_position = requests[i];

}

}

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

}

int main() {

int requests[] = {98, 183, 37, 122, 14, 124, 65, 67}; // Example request array

int num\_requests = sizeof(requests) / sizeof(requests[0]);

int head\_position = 53; // Starting head position

int direction = 0; // 0 for right, 1 for left

int disk\_size = 200; // Total size of the disk

printf("SCAN Disk Scheduling Algorithm Simulation\n");

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

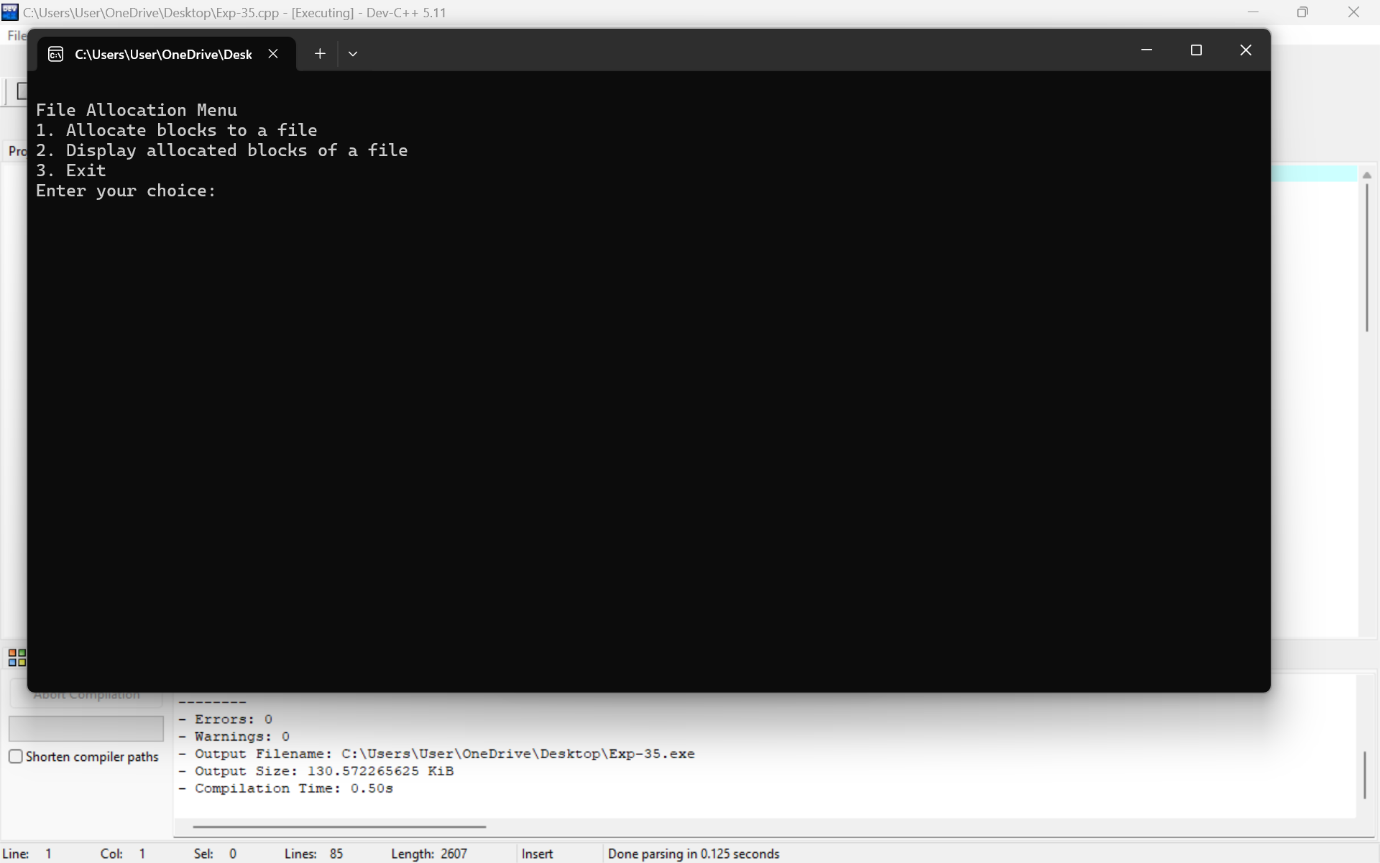
printf("Disk size: %d\n", disk\_size);

scan\_disk(requests, num\_requests, head\_position, direction, disk\_size);

return 0;

}

OUTPUT



39. Develop a C program to simulate C-SCAN disk scheduling algorithm.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#define MAX\_REQUESTS 100

// Function declarations

void cscan(int requests[], int n, int start, int max);

int main() {

int requests[MAX\_REQUESTS];

int n, start, max;

// Input number of requests

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

scanf("%d", &n);

// Input requests

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

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

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

}

// Input starting position of disk head

printf("Enter the starting position of disk head: ");

scanf("%d", &start);

// Input maximum disk limit

printf("Enter the maximum disk limit: ");

scanf("%d", &max);

// Perform C-SCAN scheduling

cscan(requests, n, start, max);

return 0;

}

// Function to simulate C-SCAN disk scheduling algorithm

void cscan(int requests[], int n, int start, int max) {

int total\_seek\_time = 0;

int current\_position = start;

int direction = 1; // 1 for moving forward, -1 for moving backward

// Sort requests to ensure they are in ascending order

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (requests[j] > requests[j + 1]) {

int temp = requests[j];

requests[j] = requests[j + 1];

requests[j + 1] = temp;

}

}

}

// Finding the position of current request

int current\_request = 0;

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

if (requests[i] >= start) {

current\_request = i;

break;

}

}

// Perform C-SCAN scheduling

printf("Sequence of disk accesses:\n");

// Move to the end of the disk

while (current\_request < n) {

printf("%d ", requests[current\_request]);

total\_seek\_time += abs(current\_position - requests[current\_request]);

current\_position = requests[current\_request];

current\_request++;

}

// Move to the beginning of the disk

total\_seek\_time += abs(current\_position - max);

current\_position = max;

printf("%d ", max);

// Move back to the start of the requests

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

total\_seek\_time += abs(current\_position - requests[i]);

current\_position = requests[i];

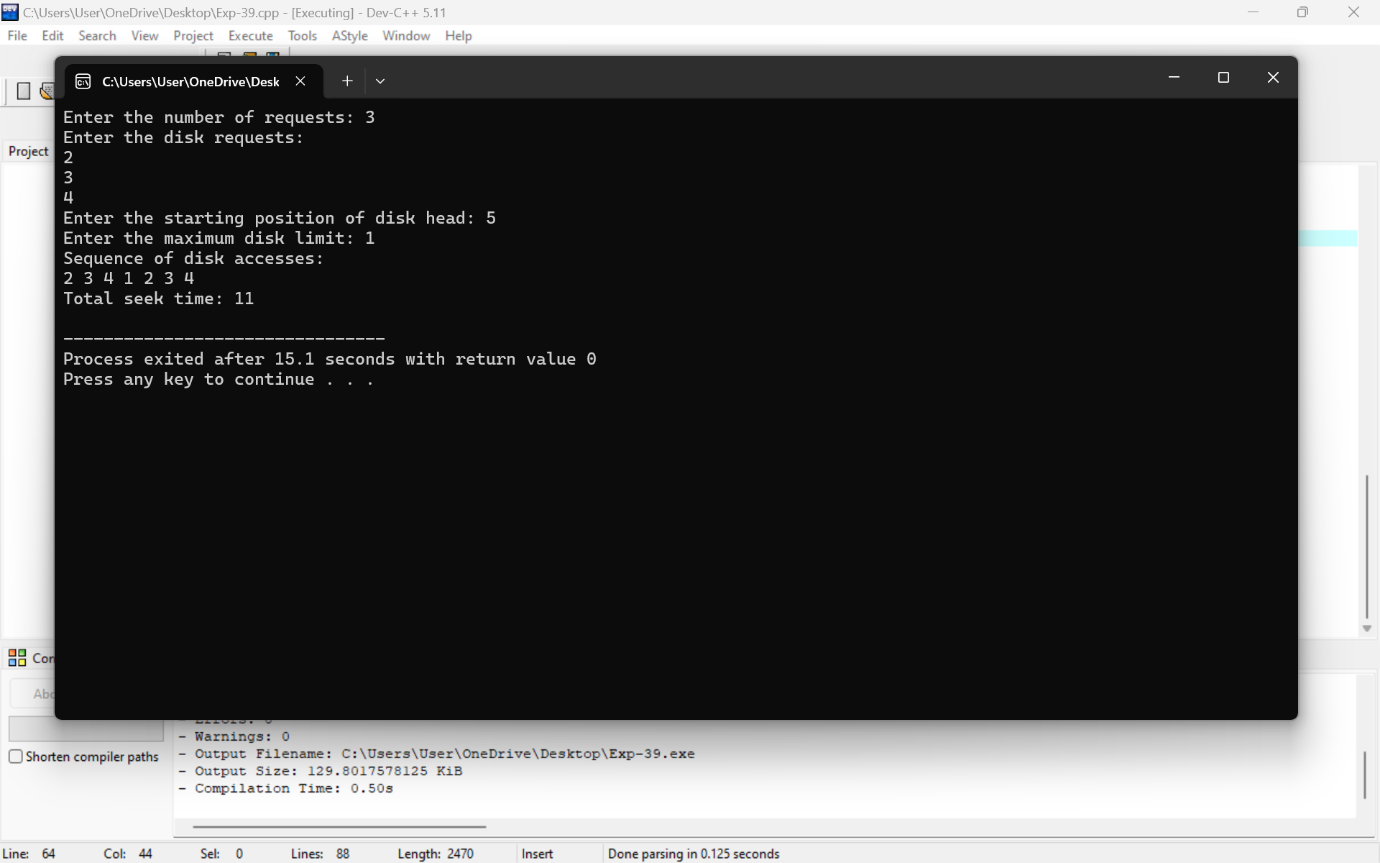
printf("%d ", requests[i]);

}

printf("\nTotal seek time: %d\n", total\_seek\_time);

}

OUTPUT



40. Illustrate the various File Access Permission and different types users in Linux.

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/stat.h>

#include <fcntl.h>

int main() {

char filename[] = "testfile.txt";

int fd;

// Create a file with read and write permissions for owner, read-only for others

fd = open(filename, O\_WRONLY | O\_CREAT | O\_TRUNC, S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH);

if (fd == -1) {

perror("Failed to create file");

return 1;

}

printf("File '%s' created successfully.\n", filename);

close(fd);

// Change file permissions to read-only for owner, group, and others

if (chmod(filename, S\_IRUSR | S\_IRGRP | S\_IROTH) == -1) {

perror("Failed to change file permissions");

return 1;

}

printf("File permissions changed successfully.\n");

// Check current user's effective UID and GID

printf("Current user's effective UID: %d\n", geteuid());

printf("Current user's effective GID: %d\n", getegid());

// Attempt to open the file for writing

fd = open(filename, O\_WRONLY);

if (fd == -1) {

perror("Failed to open file for writing");

return 1;

}

// Write to the file

if (write(fd, "Hello, World!\n", 14) != 14) {

perror("Write error");

return 1;

}

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

close(fd);

return 0;

}

OUTPUT

