**Objectives:**

The primary objective of this program is to simulate and compare different disk scheduling algorithms, namely First-Come-First-Served (FCFS), Shortest Seek Time First (SSTF), SCAN, C-SCAN, LOOK, and C-LOOK. Each algorithm’s efficiency is assessed by calculating the total head movement required to service a given sequence of disk page requests starting from an initial head position. By implementing these algorithms, the program aims to highlight their respective strengths and weaknesses in optimizing disk access. Users can input various disk request sequences and observe how each algorithm performs under identical conditions. This will provide valuable insights into which algorithm is most effective for minimizing head movement in practical scenarios.

**Suppose the head of a moving- head disk with 200 tracks, numbered 0 to 199 is currently serving request at tracks 143 and has finished a request at track 125. The queue it requests is kept in the FIFO order 86, 147, 91, 177, 94, 150, 102, 175, 130. Write a program to calculate the total head movement using following algorithms.**

**• FCFS**

**• SSTF**

**• SCAN**

**• C-SCAN**

**• LOOK**

**• C-LOOK**

**Code:**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main() {

int no\_of\_frames, no\_of\_pages;

// Read number of frames

cout << "Enter number of frames: ";

cin >> no\_of\_frames;

// Read number of pages

cout << "Enter number of pages: ";

cin >> no\_of\_pages;

vector<int> ref\_string(no\_of\_pages);

// Read page reference string

cout << "Enter page reference string: ";

for (int i = 0; i < no\_of\_pages; ++i) {

cin >> ref\_string[i];

}

vector<int> frames(no\_of\_frames, -1); // Initialize frames with -1 to indicate empty

int faults = 0;

cout<<"\n================================================================================\n";

for (int i = 0; i < no\_of\_frames; ++i)

cout << "frame[" << i << "]\t";

cout << "page fault";

cout<<"\n================================================================================\n";

for (int i = 0; i < no\_of\_pages; ++i) {

bool hit = false;

// Check if the page is already in the frames (hit)

for (int j = 0; j < no\_of\_frames; ++j) {

if (frames[j] == ref\_string[i]) {

hit = true;

break;

}

}

if (!hit) {

// Page fault occurs

// Check if there's any empty frame

bool inserted = false;

for (int j = 0; j < no\_of\_frames; ++j) {

if (frames[j] == -1) {

frames[j] = ref\_string[i]; // Insert the page in the empty frame

++faults;

inserted = true;

break;

}

}

// If no empty frame is found, apply optimal replacement

if (!inserted) {

// Determine which page to replace (optimal page replacement strategy)

int replace\_index = -1;

int farthest = -1;

for (int j = 0; j < no\_of\_frames; ++j) {

int future\_index = -1;

// Search for the next occurrence of the page in the future

for (int k = i + 1; k < no\_of\_pages; ++k) {

if (frames[j] == ref\_string[k]) {

future\_index = k;

break;

}

}

// If the page is never used again, replace it

if (future\_index == -1) {

replace\_index = j;

break;

}

// Track the page that is used farthest in the future

if (future\_index > farthest) {

farthest = future\_index;

replace\_index = j;

}

}

// Replace the page in the frames

frames[replace\_index] = ref\_string[i];

++faults;

}

}

// Print the frames

cout << "\n";

for (int j = 0; j < no\_of\_frames; ++j) {

cout << frames[j] << "\t\t";

}

if (!hit)

cout << " F"; // Mark page fault

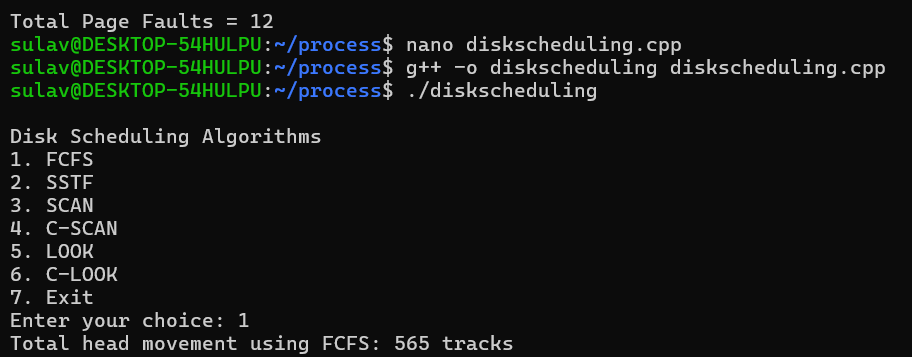
}

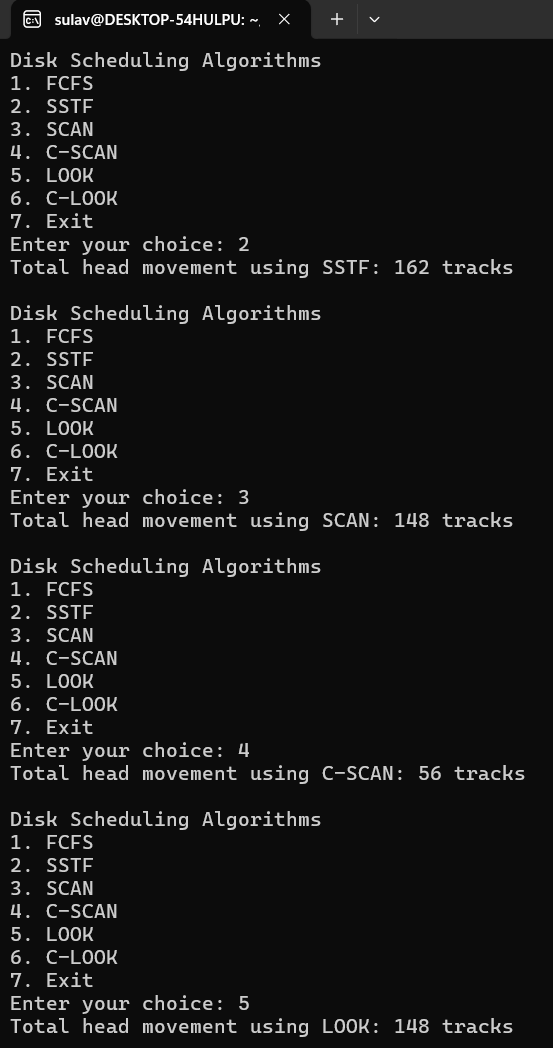
cout << "\n================================================================================\n";

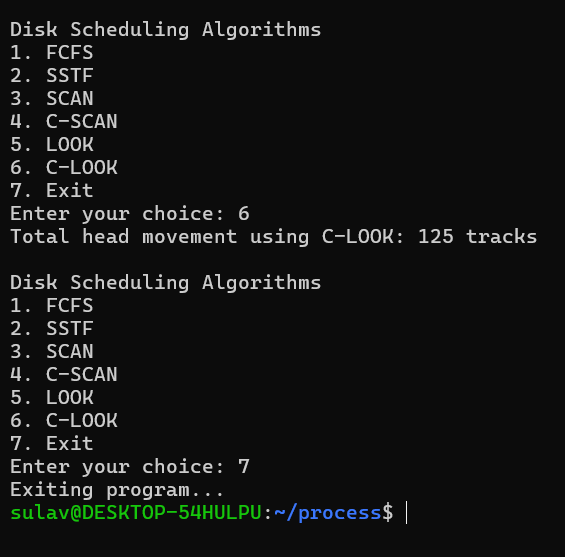
cout << "\n\nTotal Page Faults = " << faults << endl;

return 0;

}

**Output:**

****

****

**Conclusion:**

The comparison of different disk scheduling algorithms reveals distinct performance characteristics based on the total head movement required. Algorithms such as FCFS provide a straightforward approach but may result in higher head movement, while SSTF minimizes movement by serving the closest request first. SCAN and C-SCAN offer efficient access patterns by sweeping through the disk, with C-SCAN improving upon SCAN by avoiding the return sweep. LOOK and C-LOOK enhance performance further by focusing only on requests in the current direction of movement. Overall, understanding these algorithms helps users select the most appropriate disk scheduling strategy for specific applications, thereby optimizing disk performance and efficiency.