CSE 406 – Lab Report 8: Disk Scheduling using C-SCAN (Circular SCAN Algorithm)

Submitted By:

Sharif Md. Yousuf
ID: 22101128
Section: C-2
4th Year, 1st Semester
Spring 2025

Submitted To:

Atia Rahman Orthi
Lecturer
Department of Computer Science & Engineering
University of Asia Pacific

Date of Submission:
03 October, 2025 (Friday)

1   Problem Statement

In this lab, I was tasked with implementing a disk head scheduling simulator using the C-SCAN (Circular SCAN) algorithm. The challenge was to create a program that, given an initial head position and a list of pending cylinder requests, would serve these requests by moving the disk head in one direction until it reaches the end of the disk, then jumps to the beginning and continues in the same direction - providing more uniform wait time than standard SCAN. My program needed to output the order in which requests are served and calculate the total head movement.

Input

For my implementation, I used:

- A predefined sequence of disk cylinder requests

- An initial head position

- An initial direction of movement

Here's what I worked with:

```
Request sequence: {21, 39, 50, 64, 79, 90, 176}
Initial head position: 50
Initial direction: left
```

Output

My program produces:

- The order in which requests are serviced (C-SCAN sequence)

- Total head movement distance

Here's what my program outputs:

```
Request Order (C-SCAN served):
39 21 176 90 79 64 50
Total Head Movement: 243
```

2   Objective

Through this lab, I aimed to achieve several learning goals:

- Gain a deep understanding of the circular disk scheduling algorithm: C-SCAN (Circular SCAN Algorithm)

- Successfully implement C-SCAN to serve disk requests by maintaining unidirectional movement

- Learn how to calculate total head movement including the circular jump from end to beginning

- Compare C-SCAN characteristics with SCAN, FCFS and SSTF algorithms I've previously studied

- Analyze and appreciate how C-SCAN provides more uniform wait time than SCAN
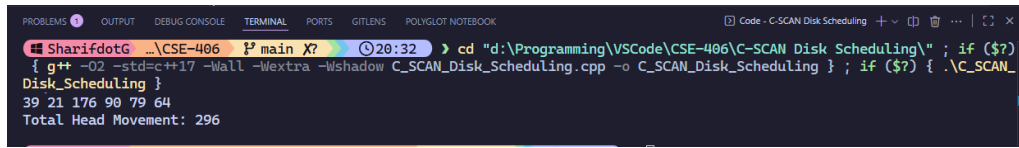
3 Source Code Screenshot


```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<int> request_sequence = {21, 39, 50, 64, 79, 90, 176};
    int head = 50;
    string direction = "left";

    vector<int> left, right;
    int total_movement = 0, current = head;

    for (int request : request_sequence) {
        if (request < head) {
            left.push_back(request);
        } else if (request > head) {
            right.push_back(request);
        }
    }

    sort(left.begin(), left.end(), greater<int>());
    sort(right.begin(), right.end());

    cout << "Request Order (C-SCAN served):" << endl;
    if (direction == "right") {
        for (int request : right) {
            cout << request << " ";
            total_movement += abs(current - request);
            current = request;
        }

        if (!left.empty()) {
            int smallest_left = *min_element(left.begin(), left.end());
            total_movement += abs(current - smallest_left);
            current = smallest_left;
            cout << smallest_left << " ";

            left.erase(find(left.begin(), left.end(), smallest_left));
        }

        reverse(left.begin(), left.end());
        for (int request : left) {
            cout << request << " ";
            total_movement += abs(current - request);
            current = request;
        }
    } else {
        for (int request : left) {
            cout << request << " ";
            total_movement += abs(current - request);
            current = request;
        }

        if (!right.empty()) {
            int largest_right = *max_element(right.begin(), right.end());
            total_movement += abs(current - largest_right);
            current = largest_right;
            cout << largest_right << " ";

            right.erase(find(right.begin(), right.end(), largest_right));
        }

        reverse(right.begin(), right.end());
        for (int request : right) {
            cout << request << " ";
            total_movement += abs(current - request);
            current = request;
        }
    }

    cout << endl << "Total Head Movement: " << total_movement << endl;

    return 0;
}
```

Figure 1: C-SCAN Disk Scheduling Source Code

4 Output Screenshot

Figure 2: C-SCAN Program Output

5 Discussion

Working with C-SCAN (Circular SCAN) has been an enlightening experience that showed me how a simple modification to SCAN can significantly improve fairness in disk scheduling. I discovered that C-SCAN represents an evolution of the SCAN algorithm, addressing its bias toward the middle cylinders. Here's what I learned about its key characteristics:

- **Unidirectional Service:** I found that C-SCAN's strength lies in its unidirectional approach. Unlike SCAN which reverses direction, C-SCAN moves in one direction, services all requests, then jumps back to the beginning and continues in the same direction.

- **Uniform Wait Time:** What impressed me most was how C-SCAN provides more uniform wait time for all requests. By treating the disk as a circular list, it eliminates the advantage that middle-cylinder requests have in standard SCAN.

- **Fairness Guarantee:** I noticed that C-SCAN treats all areas of the disk more fairly. Requests at both ends of the disk have similar maximum wait times, unlike SCAN where requests at the extremes might wait longer.

- **Return Sweep Overhead:** However, I observed that the return sweep from the highest cylinder to the lowest adds overhead that doesn't service any requests, which can increase total seek time.

- **Implementation Complexity:** The algorithm requires sorting requests and managing the circular jump logic, making it slightly more complex than standard SCAN but still straightforward to implement.

My Comparison with Other Algorithms:

- **vs SCAN:** I observed that while C-SCAN may have higher total movement in some cases, it provides much better fairness by eliminating the direction-reversal bias that SCAN exhibits.

- **vs SSTF:** C-SCAN maintains SCAN's advantage over SSTF by completely eliminating starvation, while also improving upon SCAN's fairness characteristics.

- **Real-world Usage:** I learned that C-SCAN is particularly useful in systems where uniform response time is crucial, making it suitable for real-time applications.

Through this implementation, I can see why C-SCAN is considered an improvement over SCAN for certain applications - it provides better fairness and more predictable performance across all disk regions.

6   Conclusion

Completing this C-SCAN disk scheduling implementation has been a valuable experience that taught me about the importance of fairness in algorithm design. Through this project, I've gained a deep appreciation for how a circular approach can eliminate biases present in linear algorithms.

While I observed that C-SCAN may have slightly higher seek times due to the return sweep, I've learned that it offers something more valuable for certain applications: uniform wait times across all disk regions. The elimination of directional bias, combined with the guarantee against starvation, makes C-SCAN an excellent choice for systems requiring predictable response times.

This lab has been invaluable in showing me how incremental improvements to existing algorithms can lead to solutions better suited for specific use cases. C-SCAN demonstrates that sometimes accepting a small overhead can yield significant improvements in fairness and predictability. This understanding will be essential as I continue to explore various disk scheduling strategies and their trade-offs in operating systems.