Department of Computer Science & Engineering,
University of Asia Pacific.

Course Title : **Operating System Lab**
Course Code : **CSE 406**
Lab Report : **08**

Experiment Name : Scan Disk Scheduling
Submission Date : 12.04.25

Submitted To:
Atia Rahman Orthi
Lecturer,
Department of CSE, UAP

Submitted By:
Debodipto Samadder
Reg: 21201199
Sec: B2

Problem Statement:

Implement the Scan Disk Scheduling Algorithm.

Request Sequence = [0 14 41 53 65 67 98 122 124 183 199]

Head = 53

Solving Process Steps:

1. Request sequence: [0, 14, 41, 53, 65, 67, 98, 122, 124, 183, 199].

2. Head starts at 53.

3. Disk direction: "right".

4. Disk size (max track number): 199.

5. The request sequence is sorted to simulate the correct disk track order.

6. left: All requests less than head → [0, 14, 41].

7. right: All requests greater than or equal to head → [53, 65, 67, 98, 122, 124, 183, 199].

8. Head moves from 53 to all tracks in right.

9. For each move:

10. Append the track to seek_sequence.

11. Calculate and add seek distance to total_seek.

12. If the current head position is not at 199, move to 199 and add the seek distance.

13. Head now services left (in reverse order): [41, 14, 0].

14. Each move adds to seek_sequence and total_seek.

15. Output the seek sequence and total seek operations.

Source Code:

```cpp
#include <iostream>
#include <algorithm>

using namespace std;

int main() {
    int requests[] = {0, 14, 41, 53, 65, 67, 98, 122, 124, 183, 199};
    int n = sizeof(requests) / sizeof(requests[0]);
    int head = 53;
    int total_seek = 0;
    int disk_size = 200;

    sort(requests, requests + n);

    int index;
    for (int i = 0; i < n; i++) {
        if (requests[i] >= head) {
            index = i;
            break;
        }
    }

    cout << "Seek Sequence: ";

    for (int i = index; i < n; i++) {
        cout << requests[i] << " ";
        total_seek += abs(head - requests[i]);
        head = requests[i];
    }

    if (head != disk_size - 1) {
        total_seek += abs(head - (disk_size - 1));
        head = disk_size - 1;
    }
    for (int i = index - 1; i >= 0; i--) {
        cout << requests[i] << " ";
        total_seek += abs(head - requests[i]);
        head = requests[i];
    }

    cout << "\nTotal number of seek operations = " << total_seek << endl;
    return 0;
}
```

Output:

```
Seek Sequence: 53 65 67 98 122 124 183 199 41 14 0
Total number of seek operations = 345

Process returned 0 (0x0)   execution time : 0.284 s
Press any key to continue.
```

Github Link:

https://github.com/debodipto/OS-lab-7/blob/main/OS%20Lab%208.cpp

Discussion:

Advantages:

- **Fair Access for All Requests:** SCAN services requests in both directions, ensuring no request is indefinitely delayed, thus avoiding starvation.
- **Improved Efficiency Over FCFS:** By moving in a systematic sweep, SCAN reduces the total head movement compared to the randomness of FCFS.
- **Predictable and Consistent:** The algorithm provides reliable behavior, making it suitable for systems with a steady or uniform request pattern.
- **Balanced Performance**: It combines fairness with efficiency, striking a good compromise between SSTF's speed and FCFS's simplicity.

Disadvantages:

- **Potential Delay for New Requests:** If a request arrives just after the head has passed, it must wait for the next sweep, increasing response time.
- **Unnecessary End Movement**: The head travels to the end of the disk even if there are no requests there, which can lead to extra seek time.
- **Not Always the Best Fit**: In some cases, alternative algorithms like LOOK or C-LOOK may offer better performance by avoiding full disk sweeps.

Conclusion:

The SCAN Disk Scheduling Algorithm, also known as the "elevator algorithm," processes requests in a sweeping motion from one end of the disk to the other and back. This approach reduces randomness and ensures fairness by servicing all requests along the path. While it introduces some overhead by moving to the physical disk limits, even when unnecessary, it still offers a significant performance boost over simpler methods like FCFS. Overall, SCAN is a dependable and balanced scheduling technique, especially effective in environments with uniformly distributed or predictable request patterns.