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

Course Title : **Operating System Lab**
Course Code : **CSE 406**
Lab Report : **09**

Experiment Name : C 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 C Scan Disk Scheduling Algorithm.

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

Head = 53

Solving Process Steps:

1. Input Initialization
2. Request queue: [0, 14, 41, 53, 65, 67, 98, 122, 124, 183, 199]
3. Head start position: 53
4. Disk size: 200 (valid track numbers range from 0 to 199)
5. Sort the Request Queue
6. Sorted list: [0, 14, 41, 53, 65, 67, 98, 122, 124, 183, 199]
7. Right of head (≥ 53): [53, 65, 67, 98, 122, 124, 183, 199]
8. Left of head (< 53): [0, 14, 41]
9. Service Right Side
10. Move from head = 53 to:
11. 53 → 65 → 67 → 98 → 122 → 124 → 183 → 199
12. Add absolute differences to seek_count
13. Jump to Start (C-SCAN Behavior)
14. If not already at the end (199), go to 199
15. From 199, jump to 0 (circular move)
16. Add seek distances to seek_count
17. Service Left Side
18. From 0, go to: 0 → 14 → 41
19. Continue updating the seek_count and sequence
20. Output the seek sequence and total seek operations.

Source Code:

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

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;
    }

    total_seek += head;
    head = 0;
    for (int i = 0; i < index; 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 0 14 41
Total number of seek operations = 386

Process returned 0 (0x0)   execution time : 0.968 s
Press any key to continue.
```

Github Link:

https://github.com/debodipto/OS-lab-7/blob/main/OS%20Lab-9.cpp

Discussion:

Advantages:

- **Equal Treatment of Requests:** C-SCAN provides uniform wait time by servicing requests in one direction only and jumping back to the beginning, ensuring fairness and predictability.

- **Avoids Starvation:** By not revisiting earlier tracks in the same pass, it ensures no request is indefinitely delayed.

- **Efficient for Heavy Loads**: Ideal for high-throughput systems with a constant stream of requests spread across the disk.

- **Consistent Performance:** Maintains a more uniform response time compared to SCAN by treating the disk as a circular buffer.

Disadvantages:

- **Extra Seek for the Jump:** After reaching the end, the head jumps back to the start without servicing any requests, adding to the total seek time.

- **Not Ideal for Light Loads:** In cases where requests are clustered or sparse, the long jump may result in unnecessary movement.

- **Wastes Movement if End is Empty:** If there are no requests near the disk's end, the travel up to and the jump back from the end could be inefficient.

Conclusion:

The C-SCAN (Circular SCAN) Disk Scheduling Algorithm optimizes disk performance by moving the head in one direction only and jumping back to the beginning without servicing requests on the return path. This design ensures uniform wait times and avoids starvation, making it suitable for systems with consistent or distributed workloads. Although the circular movement introduces additional seek during the jump, C-SCAN delivers a more balanced and predictable performance compared to traditional SCAN, especially under heavy or continuous disk access scenarios.