Analysis Of Disk Scheduling Algorithms

Group Assignment Report

EC 6110: Operating Systems

11th Aug 2022

By Group 2:

Cumaranathunga P. I. A (2018/E/022)

Keshan M. W. M. (2018/E/062)

Maduranga W.P.N. (2018/E/073)

Piyasamara G. L. D. S. (2018/E/093)

Rodrigo S.M. (2018/E/102)

Sachintha D. C. (2018/E/104)

Table of Contents

1. Introduction

Disk scheduling is a policy of operating system to decide which I/O request is going to be satisfied foremost. Some of the important terms in disk scheduling,

- **Seek Time**: time taken in locating the disk arm to a specified track where the read/write request will be satisfied
- **Rotational Latency**: time taken by the desired sector to rotate itself to the position from where it can access the R/W heads
- **Transfer Time**: time taken to transfer the data
- **Disk Access Time** = Rotational Latency + Seek Time + Transfer Time
- **Disk Response Time**: average of time spent by each request waiting for the IO operation

Disk scheduling algorithms applied to decide which I/O request is going to be satisfied first. The goal of disk scheduling algorithms is to maximize the throughput and minimize the response time.

Main task of this assignment was finding the best disk scheduling algorithm from the below list.

- First Come First Served (FCFS)
- Shortest Seek Time First (SSTF)
- SCAN
- C-SCAN
- C-LOOK

We created a user interface for give input and take output on a proper way to achieve our task. Furthermore, we describe each output that we were tested using our UI. And also, here we attached a comparison of disk scheduling algorithms in this report.

2. Goal of Disk Scheduling Algorithm

- Fairness
- High throughout
- Minimal traveling head time

3. Working Procedure of Disk Scheduling Algorithms

1. First Come First Served (FCFS)
   - services the IO requests in the order in which they arrive

2. Shortest Seek Time First (SSTF)
   - Selects the disk I/O request which requires the least disk arm movement from its current position regardless of the direction.

3. SCAN / Elevator
   - The disk arm moves into a particular direction till the end, satisfying all the requests coming in its path, and then it turns back and moves in the reverse direction satisfying requests coming in its path.

4. C-SCAN
   - disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, then goes to the other end of the disk and starts servicing the requests from there.

5. C-LOOK
   - the arm of the disk moves outwards servicing requests until it reaches the highest request cylinder, then it jumps to the lowest request cylinder without servicing any request then it again start moving outwards servicing the remaining requests.

4. Codes for algorithms

a. Codes for each algorithm

i. FCFS

```
export default function FCFS(arr, head) {
    let size = arr.length;
    let seek_count = 0;
    let distance, cur_track;

    for (let i = 0; i < size; i++) {
        cur_track = arr[i];
```

```
        // Calculate absolute distance
        distance = Math.abs(cur_track - head);

        // Increase the total count
        seek_count += distance;

        // Accessed track is now new head
        head = cur_track;
    }

    return { seek_count, arr };
}
```

ii. SSTF

```
export default function SSTF(queue, head) {

    var n = queue.length;
    head = 50;

    if (n <= 0) {
        return;
    }
    let seek_time = 0.0;
    var minimum = 0.0;
    //This are storing the information of seek sequence
    var skeek = Array(n + 1).fill(0);
    //Create 2d array which is used to store distance and visited
status
    var auxiliary = Array(n).fill(0).map(() => new Array(n).fill(0));
    // Loop controlling variable
    var i = 0;
    var j = 0;
    var location = 0;
    for (i = 0; i < n; ++i) {
        // set initial distance
        auxiliary[i][0] = 0;
        // set the visiting element status
        auxiliary[i][1] = 0;
    }
    for (i = 0; i < n; i++) {
        skeek[i] = head;
        // Find distance using head value
        for (j = 0; j < n; ++j) {
            auxiliary[j][0] = queue[j] - head;
            if (auxiliary[j][0] < 0) {
```

```
                auxiliary[j][0] = -auxiliary[j][0];
            }
        }
        minimum = Number.MAX_VALUE;
        location = -1;
        //Find the minimum element location that is not visited
        for (j = 0; j < n; ++j) {
            if (auxiliary[j][1] === 0 && auxiliary[j][0] <= minimum)
{
                // Get the new minimum distance element location
                location = j;
                minimum = auxiliary[j][0];
            }
        }
        // Update the visited status of new get element
        auxiliary[location][1] = 1;
        // Update head data into current track value
        head = queue[location];
        // Add current distance into seek
        seek_time += auxiliary[location][0];
    }
    if (head !== 0) {
        // Add last skee info
        skeek[n] = head;
    }
    // process.stdout.write("\n Seek Sequence : ");
    //Display given queue elements

    let finalArr = [];

    for (i = 1; i <= n; i++) {
        finalArr.push(skeek[i]);
        // process.stdout.write(" " + skeek[i] + "");
    }
    //Display result
    // console.log("\n Total Seek Time : " + seek_time);
    // console.log("\n Arr : " + finalArr);

    let seek_count = seek_time;
    let arr = finalArr;

    return { seek_count, arr };
}
```

iii. SCAN

```
let size = 8;
let disk_size = 199;

export default function SCAN(arr, head, direction) {
    let seek_count = 0;
    let distance, cur_track;
    let left = [],
        right = [];
    let seek_sequence = [];

    // appending end values
    // which has to be visited
    // before reversing the direction
    if (direction === "left") left.push(0);
    else if (direction === "right") right.push(disk_size - 1);

    for (let i = 0; i < size; i++) {
        if (arr[i] < head) left.push(arr[i]);
        if (arr[i] > head) right.push(arr[i]);
    }

    // sorting left and right vectors
    left.sort(function (a, b) {
        return a - b;
    });
    right.sort(function (a, b) {
        return a - b;
    });

    // run the while loop two times.
    // one by one scanning right
    // and left of the head
    let run = 2;
    while (run-- > 0) {
        if (direction === "left") {
            for (let i = left.length - 1; i >= 0; i--) {
                cur_track = left[i];

                // appending current track to seek sequence
                seek_sequence.push(cur_track);

                // calculate absolute distance
                distance = Math.abs(cur_track - head);

                // increase the total count
                seek_count += distance;
```

```
                // accessed track is now the new head
                head = cur_track;
            }
            direction = "right";
        } else if (direction === "right") {
            for (let i = 0; i < right.length; i++) {
                cur_track = right[i];

                // appending current track to seek sequence
                seek_sequence.push(cur_track);

                // calculate absolute distance
                distance = Math.abs(cur_track - head);

                // increase the total count
                seek_count += distance;

                // accessed track is now new head
                head = cur_track;
            }
            direction = "left";
        }
    }

    arr = seek_sequence;
    console.log(arr);
    console.log(seek_count);

    return { seek_count, arr };

}
```

iv. C-SCAN

```
let size = 8;
let disk_size = 199;

export default function CSCAN(arr, head) {
    let seek_count = 0;
    let distance, cur_track;
    let left = [],
        right = [];
    let seek_sequence = [];

    // appending end values
    // which has to be visited
    // before reversing the direction
    left.push(0);
```

```
    right.push(disk_size - 1);

    // tracks on the left of the
    // head will be serviced when
    // once the head comes back
    // to the beggining (left end).
    for (let i = 0; i < size; i++) {
        if (arr[i] < head) left.push(arr[i]);
        if (arr[i] > head) right.push(arr[i]);
    }

    // sorting left and right vectors
    left.sort(function (a, b) {
        return a - b;
    });
    right.sort(function (a, b) {
        return a - b;
    });

    // first service the requests
    // on the right side of the
    // head.
    for (let i = 0; i < right.length; i++) {
        cur_track = right[i];

        // appending current track to seek sequence
        seek_sequence.push(cur_track);

        // calculate absolute distance
        distance = Math.abs(cur_track - head);

        // increase the total count
        seek_count += distance;

        // accessed track is now new head
        head = cur_track;
    }

    // once reached the right end
    // jump to the beggining.
    head = 0;

    // adding seek count for head returning from 199 to 0
    seek_count += disk_size - 1;

    // Now service the requests again
    // which are left.
    for (let i = 0; i < left.length; i++) {
```

```
        cur_track = left[i];

        // appending current track to seek sequence
        seek_sequence.push(cur_track);

        // calculate absolute distance
        distance = Math.abs(cur_track - head);

        // increase the total count
        seek_count += distance;

        // accessed track is now the new head
        head = cur_track;
    }


    arr = seek_sequence;

    return { seek_count, arr };
}
```

v. CLOOK

```
// Javascript implementation of the approach

let size = 8;

// Function to perform C-LOOK on the request
// array starting from the given head
export default function CLOOK(arr, head) {
    let seek_count = 0;
    let distance, cur_track;

    let left = [];
    let right = [];
    let seek_sequence = [];

    // Tracks on the left of the
    // head will be serviced when
    // once the head comes back
    // to the beginning (left end)
    for (let i = 0; i < size; i++) {
        if (arr[i] < head) left.push(arr[i]);
        if (arr[i] > head) right.push(arr[i]);
    }

    // Sorting left and right vectors
```

```
    left.sort(function (a, b) {
        return a - b;
    });
    right.sort(function (a, b) {
        return a - b;
    });

    // First service the requests
    // on the right side of the
    // head
    for (let i = 0; i < right.length; i++) {
        cur_track = right[i];

        // Appending current track
        // to seek sequence
        seek_sequence.push(cur_track);

        // Calculate absolute distance
        distance = Math.abs(cur_track - head);

        // Increase the total count
        seek_count += distance;

        // Accessed track is now new head
        head = cur_track;
    }

    // Once reached the right end
    // jump to the last track that
    // is needed to be serviced in
    // left direction
    seek_count += Math.abs(head - left[0]);
    head = left[0];

    // Now service the requests again
    // which are left
    for (let i = 0; i < left.length; i++) {
        cur_track = left[i];

        // Appending current track to
        // seek sequence
        seek_sequence.push(cur_track);

        // Calculate absolute distance
        distance = Math.abs(cur_track - head);

        // Increase the total count
        seek_count += distance;
```

```
        // Accessed track is now the new head
        head = cur_track;
    }


    arr = seek_sequence;

    return { seek_count, arr };
}
```

b. Link

You can download the full project code from this link:

https://github.com/IsuruAkalanka/operating-systems-disk-scheduling.git

5. Explanation of outputs

Example 1:

A disk contains 200 tracks (0-199).

Request queue contains track numbers,

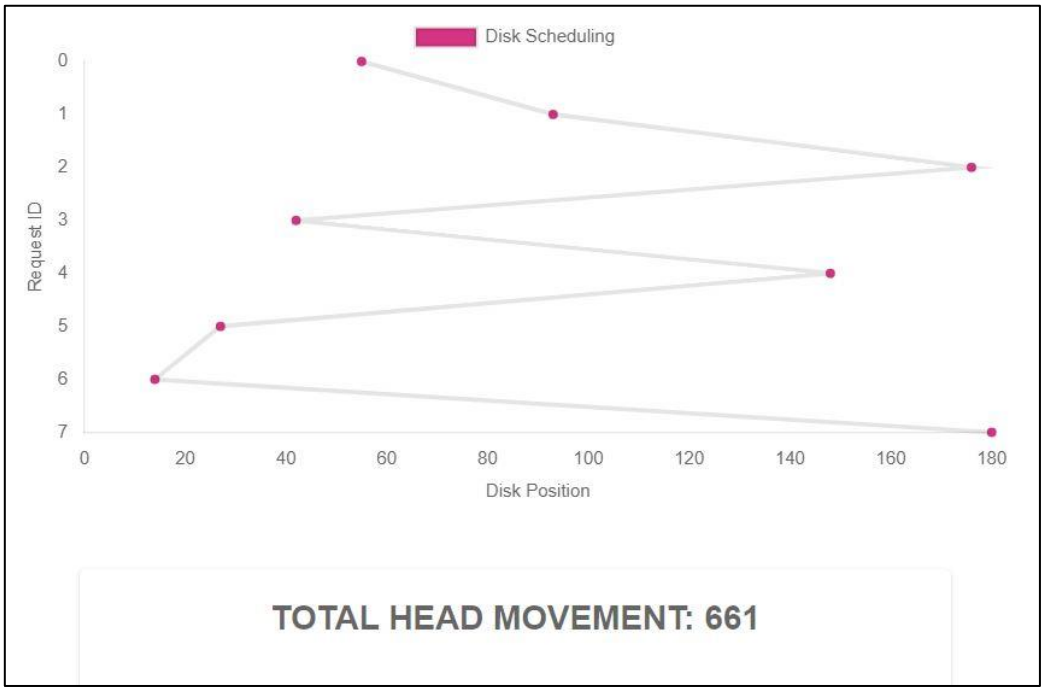
93, 176, 42, 148, 27, 14, 180

Current position read/write head = 55

Calculate total number of track movements of read/write head using,

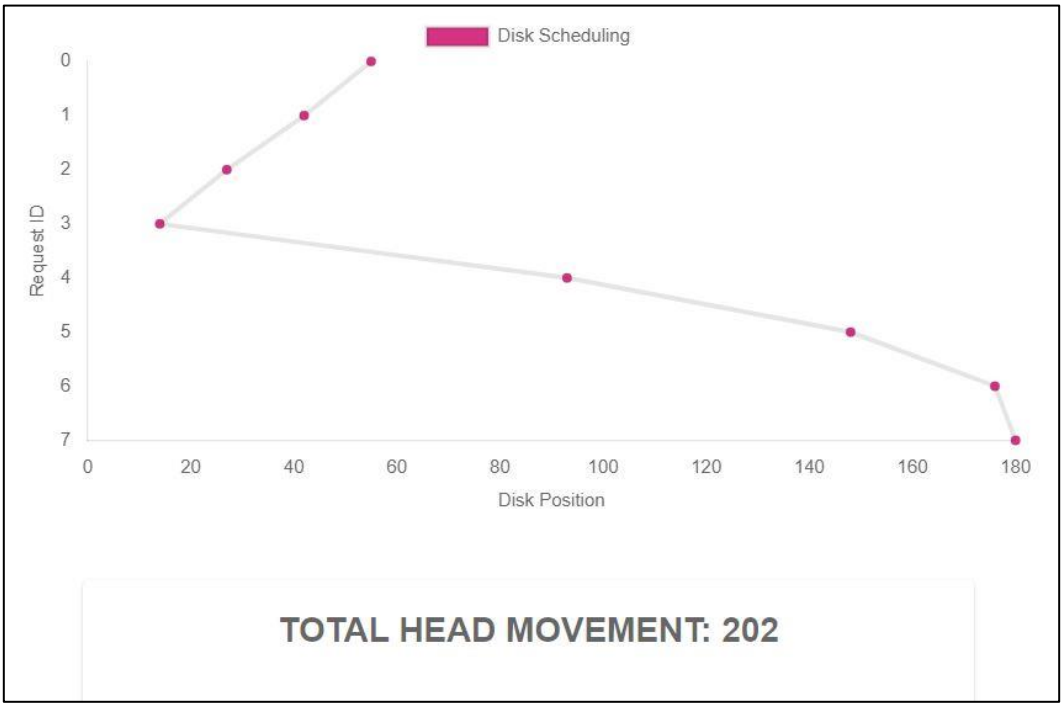
1. First come first served (FCFS)
2. Shortest seek time first (SSTF)
3. SCAN
4. C-SCAN
5. C-LOOK

1. First come first served (FCFS)



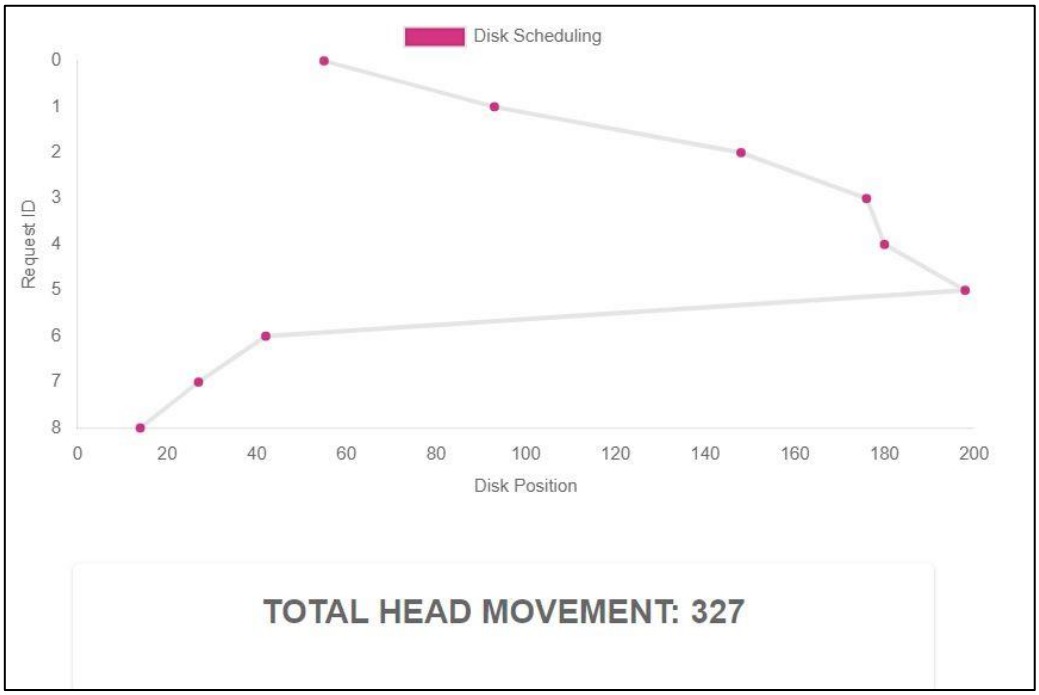
- In first come first served, the requests are addressed in the order they arrived in the disk queue. Track will move forward or backward according to track numbers using FCFS algorithm.
- Here, the arm starts at 55. Then the track moves to 93. Because it has come first. Then move to 176. Likewise arm move to track numbers according to contains track numbers in the request queue.

2. Shortest seek time first (SSTF)



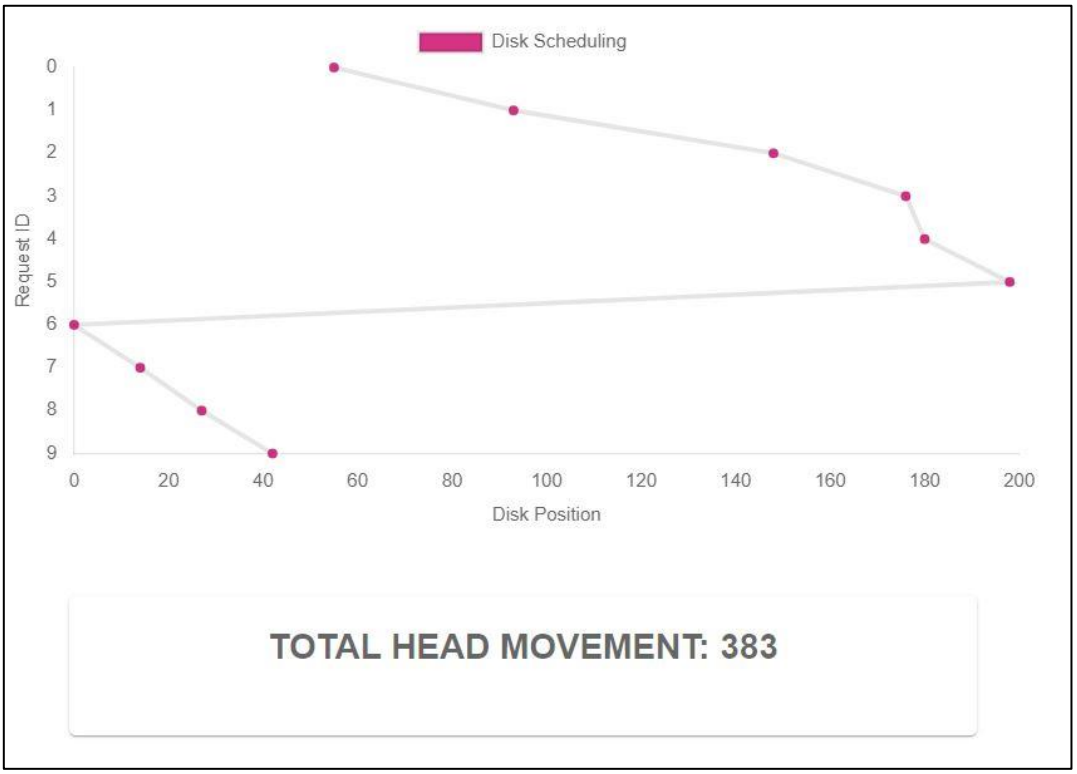
- In SSTF algorithm, request having shortest seek time are executed first. So, the seek time at every request is calculated in advance in queue and schedule according to their seek time. As a result, request near the disk arm will get executed first.
- Here, the arm starts at 55. Then it moves to 42. Because that track number is the nearest track number from 55. Then move into 27. It is the nearest track number from 42 among the remaining track numbers.

3. SCAN



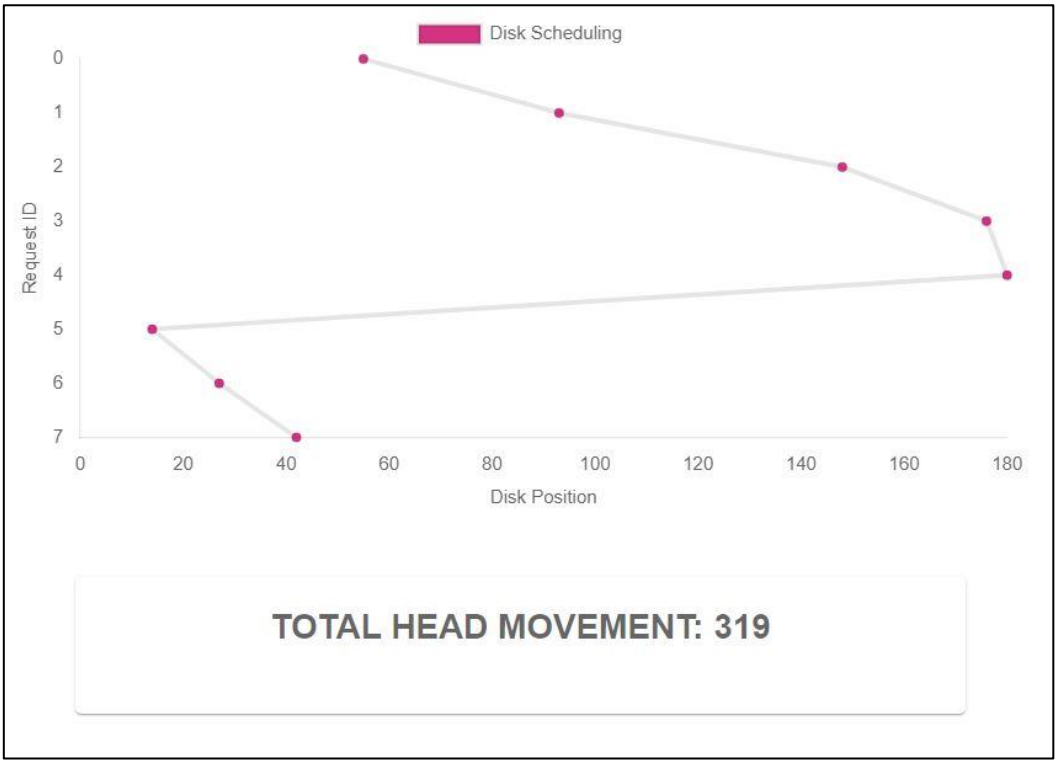
- In SCAN, arm moves in one direction only by satisfying all outstanding requests, until there are no more requests in that direction. Then service direction is reversed.
- Here, the arm starts at 55. Then it moves to one direction. That's why it reaches 93, 148, 176, 180 respectively. After reaching the end of that direction, head reverses its direction and move towards the starting end servicing all the requests in between. So, it reaches 42, 27, 14 respectively.

4. C-SCAN



- The C-SCAN algorithm is the variant of the SCAN algorithm only difference is that when the head reaches the other end, it immediately returns to the beginning of the disk without servicing any requests on a return trip.
- Here also, the arm starts at 55 and it moves to one direction. Then it reaches 93, 148, 176, 180 respectively. After reaching the end of that direction, it immediately returns to the beginning of the disk without servicing any requests on a return trip. After that it reaches 14, 27, 42 respectively.

5. C-LOOK



- In this algorithm, the head services requests only in one direction (either left or right) until all the requests in this direction are not serviced and then jumps back to the farthest request in the other direction and services the remaining requests which gives a better uniform servicing as well as avoids wasting seek time for going till the end of the disk.
- Here, the arm starts at 55. Then it moves to one direction and reaches 93, 148, 176, 180 respectively. After services requests in this direction, it jumps back to the farthest request in the other direction and reaches 14, 27, 42 respectively.

6. Comparison of Disk Scheduling Algorithms

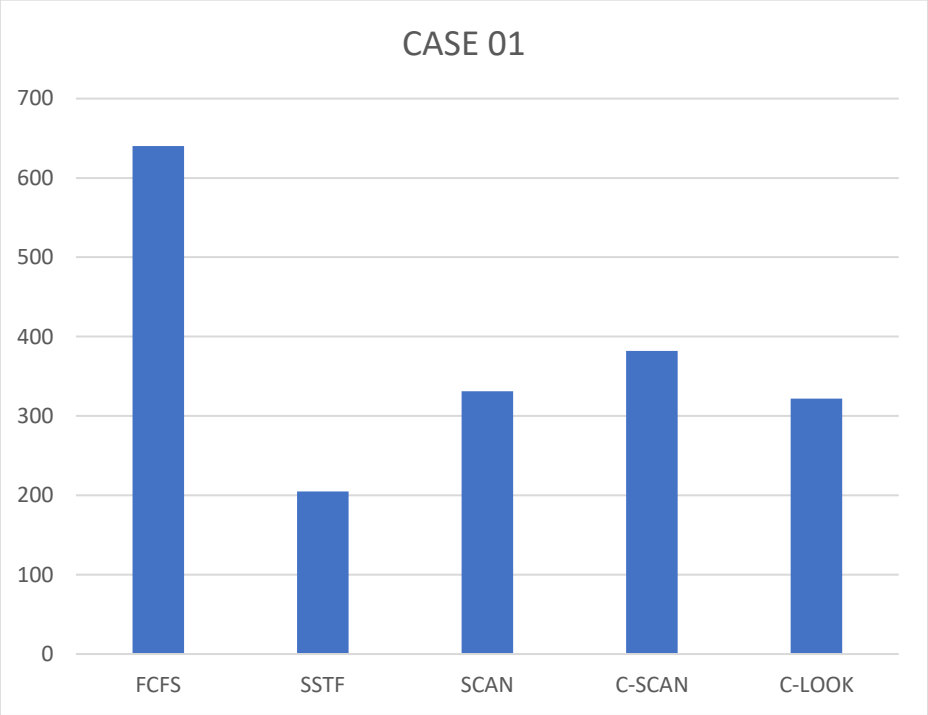
For Comparison we used 3 cases.

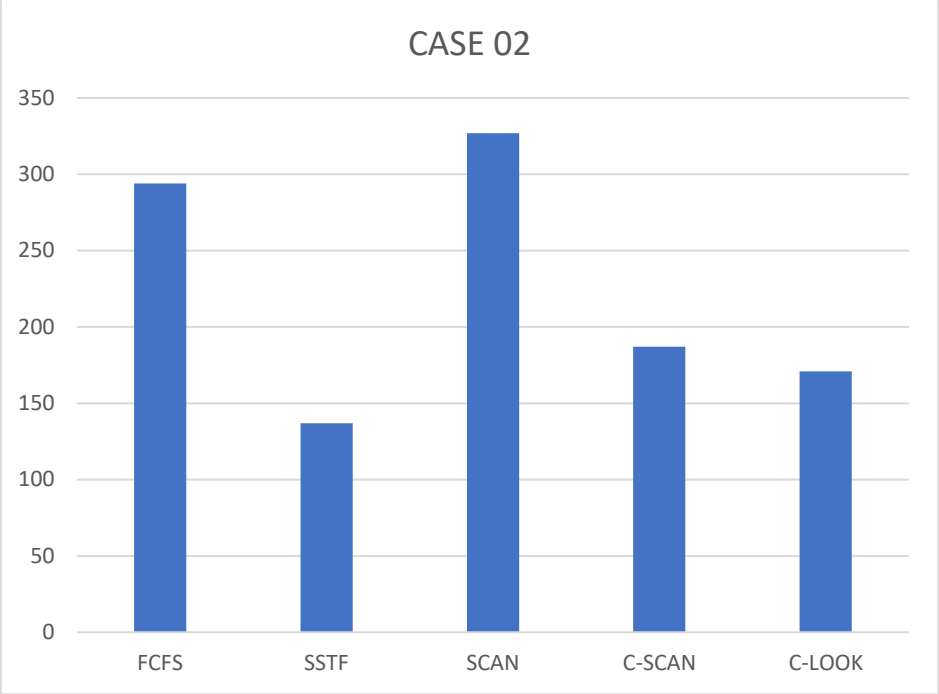
Table 1: Case description

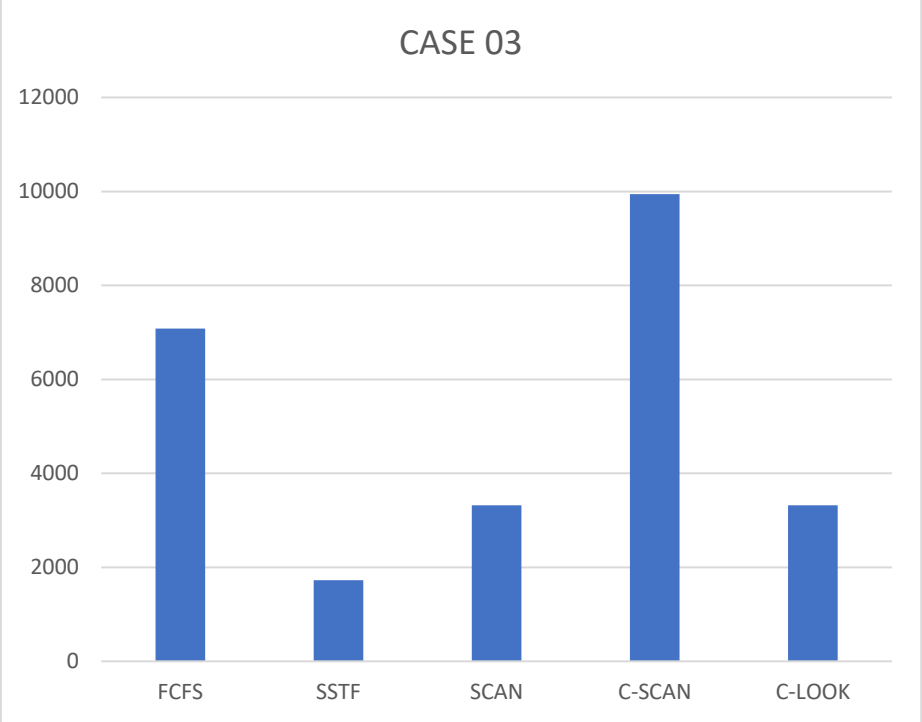
CASE	CYLINDERS SIZE	DISK QUEUE	CURRENTLY AT
01	200	98, 183, 37, 122, 14, 124, 65, 67	53
02	100	33, 72, 47, 8, 99, 74, 52, 75	63
03	5000	86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130	143

Table 2: Outputs for the cases

SCHEDULING ALGORITHM	TOTAL HEAD MOVEMENT		
	CASE 01	CASE 02	CASE 03
FCFS	640	294	7081
SSTF	205	137	1724
SCAN	331	327	3319
C-SCAN	382	187	9941
C-LOOK	322	171	3319







When we consider Total Head Movement, the best algorithm is SSTF, and secondly C-LOOK algorithm.

In FCFS, the requests are addressed in the sequence they come in the disk queue.

The SSTF (Shortest Seek Time First) algorithm prioritizes requests with the shortest seek times for execution. Then, based on the calculated seek time, they are scheduled. As a

result, the request that is closest to the disk arm is executed first. SSTF is unquestionably better compared to FCFS because it reduces average response time and increases system throughput.

In the SCAN algorithm, the disk arm travels in a specific direction and services the requests that come into its path. When it reaches the end of the disk, it reverses its direction and services the requests that come into its path again. Because of this, the requests at the midrange receive more attention, and those coming from behind the disk arm must wait.

In the C-SCAN algorithm, SCAN conditions are circumvented by having the disk arm travel to the opposite end of the disk instead of reversing its direction, where it then begins to check requests. As a result, the disk arm transfers in a circular manner.

The CSCAN disk scheduling algorithm is similar to CLOOK. In the CLOOK, despite working to the end, the disk arm only moves to the last application to be examined in front of the head and then to the last request on the other end. As a result, it also eliminates the extra delay brought on by unnecessary traversal of the disk's last sector.

7. References

1) Shastri, Sourabh & Mansotra, Vibhakar & Sharma, Anand. (2016). A Comparative Analysis of Disk Scheduling Algorithms.
2) https://www.javatpoint.com/os-disk-scheduling
3) https://www.geeksforgeeks.org/disk-scheduling-algorithms/
4) M.R. Mahesh Kumar, B. Renuka Rajendra, An Improved Approach to Maximize the Performance of Disk Scheduling Algorithm by Minimizing the Head Movement and Seek Time Using Sort Mid Current Comparison (SMCC) Algorithm, Procedia Computer Science, Volume 57,2015, Pages 222-231, ISSN 1877-0509, https://doi.org/10.1016/j.procs.2015.07.468.