



CST232 – Operating Systems


Semester 1, Academic Year

2024/2025

Assignment 2

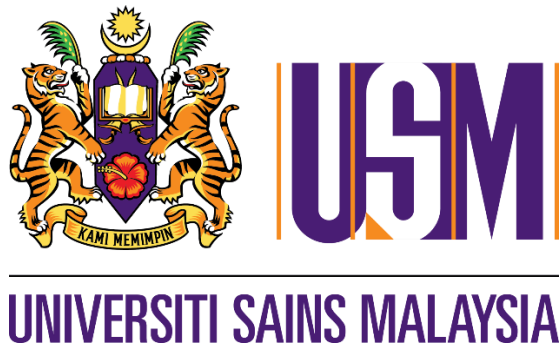
Students' declaration:

We declare that we understand what is meant by plagiarism. This assignment is all our own work and we have acknowledged any use of the published or unpublished works of other people. We hold a copy of this assignment. We can produce if the original is lost or damaged.

Student Name	Signature
Faiq Ahmed Shaikh	
Syeda Laiba Faraz	Laiba
Yan Keer	Yan Keer
Md Fahim Anam Sakib	Fahim

Lecturer has, and may exercise, the right NOT TO MARK this assignment if the above declaration has NOT BEEN SIGNED and if the above declaration is FOUND TO BE FALSE, appropriate action will be taken which would lead to ZERO marks being awarded for this assignment

This **GROUP** assignment shall contribute **15%** of the overall evaluation. For this assignment, each group is required to complete the following tasks in **THREE (3) weeks**.



SCHOOL OF COMPUTER SCIENCES

CST232W Operating Systems

Academic Session 2024/2025, Semester 1

Assignment 2

Prepared By: Group 04

Name	Matrics No.
Faiq Ahmed Shaikh	22304662
Syeda Laiba Faraz	22304917
Yan Keer	22304990
Md Fahim Anam Sakib	165378

Submission date:

9th January 2025

Table of Contents

Task 1	4-8
1.0 Binary semaphore Vs. Counting Semaphore Task 1	
1.1 Introduction	4
1.2 A detail comparison between Binary and Counting Semaphore	5
2.0 Semaphore Code Analysis	
2.1 Types of Semaphore	7
2.2 Purpose of the Semaphore	7
2.3 How the Code Works	8
Task 2	9-23
3.0 Disk Scheduling	
3.1 Introduction	9
3.1.1 Explanation of SCAN	9-10
3.1.2 Explanation of LOOK	11
3.2 Implementation	12-18
1. SCAN	12-15
2. LOOK	16-18
3.3 Performance Comparison	19-22
3.4 Strengths Vs Weakness	23
3.5 Personal Observation and Conclusion	24
4 References	25-26
5 Appendices	27-36

Task 1

1. Binary Semaphore Vs. Counting Semaphore

1.1 Introduction

A semaphore in an Operating System is a synchronisation mechanism that controls how many threads or processes may access shared resources. It helps prevent race conditions, where various processes try to access and modify the same resource simultaneously, leading to unpredictable results. Semaphores are just integer variables that are mostly used to combine two atomic procedures, signal and wait, for process synchronization in order to overcome the critical section problem.

How does a Semaphore work?

A semaphore works as a signalling mechanism to manage access to shared resources in a concurrent environment. It keeps track of resource availability through its value and ensures that only a specific number of processes or threads can access the resource simultaneously. The two atomic operations, **Wait (P)** and **Signal (V)**, are used to decrease or increase the semaphore value.

Semaphore Example:

Let's take an example of a restaurant, having 5 tables only. A semaphore can be used to manage these tables as follows:

1. **Semaphore Initialization:** Set the initial semaphore value to 5 (the number of vacant tables).
2. **Wait (P):** When a customer arrives and requests, a table, the semaphore value is decremented by 1. If the semaphore value is greater than zero, then the customer gets a table. Similarly, if the semaphore value is 0, the customer has to wait until the table is free.
3. **Signal (V):** When a customer finishes and leaves a table, the semaphore value is incremented by one. If there are waiting customers, one of them is notified and gets a table.

Why Semaphore is necessary?

Now that you know what is Semaphore in operating system and how it works, you're probably wondering when you should use it. Semaphore is commonly used when the multiple threads try to access a limited resource. Semaphore can help to ensure that these resources are accessed in a controlled and orderly way, preventing data corruption or inconsistency.

1.2 A detail comparison between Binary Semaphore and Counting Semaphore

Criteria	Binary Semaphore	Counting Semaphore
Definition	A semaphore that has integer values between 0 and 1 is called a binary semaphore.	A semaphore with numerous counter values is called a counting semaphore. There is no limit to the value's range.
Structure Implementation	<pre>typedef struct { int semaphore_variable; } binary semaphore</pre>	<pre>typedef struct{ Int semaphore_variable; Queue list; //A queue to store the list of task } counting_semaphore;</pre>
Representation	0 means that a process or a thread is accessing the critical section, other process should wait for it to exit the critical section. 1 represents the critical section is free.	The value can range from 0 to N, where N is the number of process or thread that must enter the critical section.
Mutual Exclusion	Since only one thread or process may reach the crucial area at a time, the answer is yes, mutual exclusion is guaranteed.	No, as many threads or processes may access the crucial area simultaneously, mutual exclusion is not guaranteed.
Bounded wait	No, because only one process may access the crucial region and there is no restriction on how long the process can stay	Yes, it ensures finite wait since it uses a queue to keep track of all the processes or threads and gives each one

	there, causing another process to starve, it does not ensure bounded wait.	an opportunity to visit the crucial part once. Thus, famine is not a concern.
Starvation	No waiting queue is present then <u>FCFS</u> (first come first serve) is not followed so, starvation is possible and busy wait present.	The waiting queue is present then FCFS (first come first serve) is followed so, no starvation hence no busy wait.
Number of instances	Used only for a single instance of resource type R.it can be used only for 2 processes.	Used for any number of instances of resource of type R.it can be used for any number of processes.

Table 1: Comparison between Binary and Counting Semaphore

2. Code Analysis

2.1 Type of Semaphore:

The semaphore used in the code is a **binary semaphore**.

- a) The semaphore in this code is a **binary semaphore**, meaning it can only have two values: 0 and 1.
- b) In this context:
 - 1 indicates that the resource is available.
 - 0 indicates that the resource is occupied.

2.2 Purpose of the Semaphore

The purpose of this binary semaphore is to **protect the critical section**, ensuring that only one thread or process can access it at a time.

<pre>volatile int semaphore = 1; void P () { while (semaphore == 0); semaphore = 0; } void V () { semaphore = 1; } int main () { printf("Entering critical section...\n"); P(); printf("In critical section!\n"); V(); printf("Exited critical section.\n"); return 0; }</pre>	<pre>// Initialize the semaphore to 1, indicating the resource is available. // Busy wait: check if the resource is occupied. // Occupy the resource by setting semaphore to 0. // Release the resource by setting semaphore to 1. // Indicate the program is attempting to access the critical section. // Wait for the resource and occupy it. // Critical section: execute the protected code. // Release the resource after exiting the critical section. // Indicate the program has finished using the resource.</pre>
--	---

Figure 1: Comment for Each Line of the Code

2.3 How the Code Works

a) **Initialization:**

- a. `volatile int semaphore = 1;` initializes the semaphore to 1, signalling that the resource is initially available.

b) **P() Operation:**

- a. The P() function is the **wait operation**. It checks if the semaphore's value is 1 (indicating the resource is available).
- b. If `semaphore == 1`, the thread sets it to 0, occupying the resource and preventing others from entering the critical section.
- c. If `semaphore == 0`, the thread enters a busy-wait loop (`while (semaphore == 0);`) until the resource is released.

c) **Critical Section:**

- a. Once the thread passes the P() operation, it can safely execute the code in the critical section, as shown by `printf("In critical section!\n");`.

d) **V() Operation:**

- a. The V() function is the **signal operation**. It sets the semaphore back to 1 (`semaphore = 1;`), indicating that the resource is now available.

e) **Main Function:**

- a. The main function demonstrates how P() and V() work together to ensure mutual exclusion for the critical section.

Task 2

3. Disk Scheduling

3.1 Introduction

Disk Scheduling algorithm is a technique used in operating systems to determine the order in which disk I/O requests are processed. When multiple I/O requests arrive for accessing data stored on a disk, the operating system uses a disk scheduling algorithm to decide the optimal order to service these requests, minimizing seek time and improving overall system performance.

How does Disk Scheduling Algorithm work?

Disk scheduling algorithms are essential for managing data access requests on the hard drive, ensuring efficient system performance. The hard drive can be conceptualized as a library containing numerous books (data blocks), and the disk scheduling algorithm functions as a librarian, systematically retrieving and organizing data to meet the user's requests effectively.

Here's is more about how it works technically:

1. **Seek Time:** This is the time it takes for the disk arm (like a needle on a vinyl record) to move to the right track where the data is located. The algorithm aims to minimize this seek time.
2. **Rotational Latency:** Once the disk arm is on the right track, the disk platter must rotate to bring the desired data under the read/write head. Again, the algorithm tries to minimize this rotational latency.
3. **Transfer Time:** Finally, the data is read from or written to the disk. This transfer time depends on the data's size and speed.
4. **Disk Access Time:** The total time for these three steps is the Disk Access Time, which the metric Disk Scheduling algorithms aim to optimize.

3.1.1 A Brief Explanation of the SCAN Algorithm

The SCAN algorithm, which is also known as "Elevator Algorithm", is a disk scheduling method that moves the disk's read/write head in one direction (either inward or outward across the disk) to service requests. Once it reaches the end of the disk or the last request in that direction, it reverses

and moves in the opposite direction, continuing to service requests. This behavior is similar to an elevator that goes up to the top floor before descending to the bottom floor.

Working Principles of SCAN Algorithm

To understand the working principles of SCAN algorithm, let's assume a disc queue with requests for I/O 'head' is position of the disc head. We will now apply the SCAN algorithm by following these steps below:

- 1. Arrange Request in Ascending Order:**

All I/O requests are sorted in ascending order to ensure the algorithm processes requests systematically.

- 2. Initial Direction of the Disk Head:**

The disk head begins moving in a specific direction (e.g., toward higher track numbers, or "right") from its initial position.

- 3. Processing Requests While Moving:**

- As the head moves, it processes each request it encounters along the way.
- The seek distance is calculated as the absolute difference between the current position of the head and the next request.

- 4. Reaching the End of the Disk:**

The head continues in the chosen direction until it reaches either the end of the disk (block size) or the last request in that direction.

- 5. Reversing Direction:**

Once the head reaches the end of the disk or the last request in the current direction, it reverses its direction.

- 6. Processing Requests in the Opposite Direction:**

The head processes the remaining requests in the reverse direction until all requests have been serviced.

- 7. Calculate Total Head Movement:**

The sum of all head movements (seek distances) gives the **total seek time**, a key metric for evaluating the algorithm's performance.

3.1.2 A Brief Explanation of the LOOK Algorithm

The LOOK algorithm is an optimized version of the SCAN algorithm, designed to reduce the seek time. It operates by scanning the disk to check if there are any more requests in a specific direction, but unlike the SCAN algorithm, it does not traverse all the way to the end of the disk before reversing direction. Instead, the LOOK algorithm reverses direction as soon as it reaches the last request in the current direction. This helps avoid the extra delays associated with SCAN, leading to improved performance (Javatpoint, n.d.).

Working Principle

1. Request Queue Sorting:

Before servicing the disk requests, the requests are sorted based on their track numbers. Sorting is done in ascending or descending order, depending on the current direction of the disk arm (GeeksforGeeks, 2023). They must be arranged in the best order to reduce disk arm movement and process the requests in a single pass.

2. Find Nearest Request:

The disk arm moves in the current direction (either towards the outer edge or the inner edge of the disk) and services all requests in that direction. The arm will continue until it reaches the last request in the current direction (GeeksforGeeks, 2023). At that point, the arm reverses direction and starts moving back, servicing requests in the opposite direction.

3. Adjusting the Movement:

The movement stops when there are no more requests in the current direction. It does not traverse to the end of the disk.

3.2 Implementation

To implement our algorithms, we used the Python programming language to execute, sort, and evaluate the total seek time and the order of requests for both the SCAN and LOOK algorithms,

1. SCAN (Elevator) Algorithm

	SCAN	
Head Position	78	
Total Number of cylinders	200	
Seek request	[82, 170, 43, 140, 24, 16, 190, 150, 36, 39, 148, 7]	
Head direction	Right	Left
Sorted Seek request	[82, 140, 148, 150, 170, 190, 199, 43, 39, 36, 24, 16, 7]	[43, 39, 36, 24, 16, 7, 0, 82, 140, 148, 150, 170, 190]
Total seek time	313	268

```
PS C:\Users\Faiq Ahmed Shaikh> python -u "c:\Users\Faiq Ahmed Shaikh\Desktop\SCAN.py"
SCAN Disk Scheduling Simulation
Enter the initial head position: 78
Enter the total number of cylinders on the disk: 200
Enter the number of I/O requests: 12
Enter the I/O requests:
82
170
43
140
24
16
190
150
36
39
148
7
Enter the initial direction (left or right): left
```

Figure 2: A Sample Input of SCAN Algorithm (Left Direction).

The above picture of sample input of SCAN (Elevator) Disk Scheduling Algorithm shows with an initial head position of 78 on a disk containing 200 cylinders, ranging from 0 to 199. There are 12

(I/O) requests to proceed, located at the cylinder positions queue 82, 170, 43, 140, 24, 16, 190, 150, 36, 39, 148 and 7. The disk head is instructed to start moving in left direction. The SCAN algorithm starts at the initial head position and moves leftward, processing all requests in that direction. After reaching the leftmost end or the lowest request, the head reverses direction and begins processing the remaining requests while moving rightward.

```
PS C:\Users\Faiq Ahmed Shaikh> python -u "c:\Users\Faiq Ahmed Shaikh\Downloads\demo.py"
PS C:\Users\Faiq Ahmed Shaikh> python -u "c:\Users\Faiq Ahmed Shaikh\Desktop\SCAN.py"
SCAN Disk Scheduling Simulation
Enter the initial head position: 78
Enter the total number of cylinders on the disk: 200
Enter the number of I/O requests: 12
Enter the I/O requests:
82
170
43
140
24
16
190
150
36
39
148
7
Enter the initial direction (left or right): right
```

Figure 3: A Sample Input of SCAN Algorithm (Right Direction).

The above picture of sample input of SCAN (Elevator) Disk Scheduling Algorithm demonstrates a scenario with an initial head position of 78 on a disk containing 200 cylinders, ranging from 0 to 199. There are 12 (I/O) requests to proceed, located at the cylinder positions queue 82, 170, 43, 140, 24, 16, 190, 150, 36, 39, 148 and 7. The disk head is instructed to start moving in right direction. The SCAN algorithm begins at the initial head position and moves rightward, servicing all requests in that direction. After reaching the rightmost request or the edge of the disk, the head reverses direction and services the remaining requests while moving leftward.

```
SCAN (Elevator) Disk Scheduling:
Seek Sequence Order: [43, 39, 36, 24, 16, 7, 0, 82, 140, 148, 150, 170, 190]
Total Seek Count: 268 NS
```

Figure 4: A Sample Output of SCAN Algorithm (Left Direction).

The output is how result would look when we run SCAN Disk Scheduling Algorithm, we used inputs we mentioned earlier. The **seek sequence** is displayed as [43, 39, 36, 24, 16, 7, 0, 82, 140, 148, 150, 170, 190], which indicates the order in which the disk head serviced the I/O requests. The algorithm starts by moving left to the lowest numbered request (and the edge of the disk, 0), then reverses direction and processes the remaining requests to the right. The total seek count is displayed as 268 NS. This metric demonstrates the efficiency of the SCAN algorithm in minimizing seek time by systematically processing requests in a sweeping motion.

```
SCAN (Elevator) Disk Scheduling:  
Seek Sequence Order: [82, 140, 148, 150, 170, 190, 199, 43, 39, 36, 24, 16, 7]  
Total Seek Count: 313 NS  
PS C:\Users\Faiq Ahmed Shaikh>
```

Figure 5: A Sample Output of SCAN Algorithm (Right Direction).

The sample output of the SCAN (Elevator) Algorithm (Right Direction) displays the sequence order in which the disk head services the requests is [82, 140, 148, 150, 170, 190, 199, 43, 39, 36, 24, 16 and 7]. This indicates that the scan algorithm starts at the initial head position then moves right to process all requests, continues to the rightmost edge of the disk, and then reverses direction to process remaining requests on the left. The total seek count is 313 NS, which represents the total distance the disk head travelled to service all the requests.

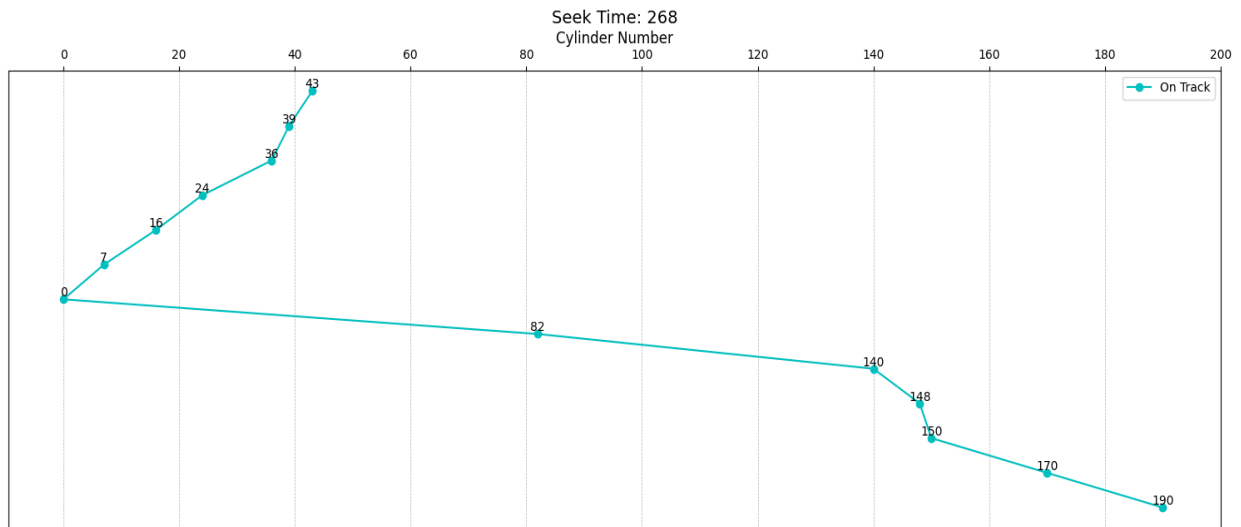


Figure 6: Graph Implementation of SCAN (Left Direction).

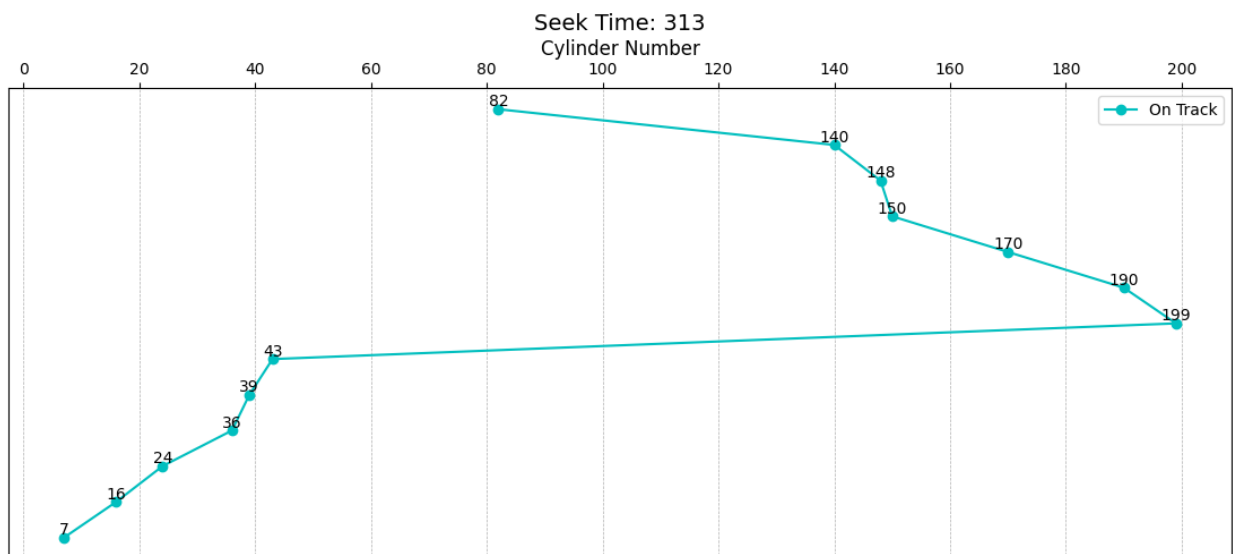


Figure 7: Graph Implementation of SCAN (Right Direction)

2. LOOK Algorithm

	LOOK	
Head Position	78	
Total Number of cylinders	200	
Seek request	[84, 34, 64, 24, 87, 56, 76, 166, 131, 174, 4, 3, 7, 19, 198, 157, 94]	
Head direction	Right	Left
Sorted Seek request	[84, 87, 94, 131, 157, 166, 174, 198, 76, 64, 56, 34, 24, 19, 7, 4, 3]	[76, 64, 56, 34, 24, 19, 7, 4, 3, 84, 87, 94, 131, 157, 166, 174, 198]
Total seek time	315	270

```
• → workspace git:(master) /bin/python3 /project/workspace/main.py
Enter the disk's initial head position: 78
Enter the total number of cylinders on the disk: 200
Enter the I/O requests (space-separated cylinder numbers): 84 34 64 24 87 56 76 166 131 174 4 3 7 19 198 157 94
Enter the initial direction of head movement ('left' for smaller, 'right' for larger track numbers): left
```

Figure 8: Sample Input for LOOK Algorithm (Left Direction)

As shown in Figure 9 an example input for the LOOK disk scheduling algorithm has an initial head position of 78 on a disk containing 200 cylinders, ranging from 0 to 199. With the following I/O requests of 84, 34, 64, 24, 87, 56, 76, 166, 131, 174, 4, 3, 7, 19, 198, 157 and 94 queued in. The disk head is set to move initially in the left direction.

```
Order of requests processed: [76, 64, 56, 34, 24, 19, 7, 4, 3, 84, 87, 94, 131, 157, 166, 174, 198]
Total seek time: 270
```

Figure 9: Sample Output for LOOK Algorithm (Left Direction)

The algorithm starts at the initial head position (78) and moves inward, starting by servicing the nearest request relative to the initial position. The seek order [76, 64, 56, 34, 24, 19, 7, 4, 3, 84, 87, 94, 131, 157, 166, 174, 198], represents the order in which the disk head services the I/O requests. Upon reaching the last request in the inward direction, the head reverses without traversing to the innermost physical end of the disk and continues processing the remaining requests while moving outward. The total seek time is 270 ms.


```

→ workspace git:(master) /bin/python3 /project/workspace/main.py
Enter the disk's initial head position: 78
Enter the total number of cylinders on the disk: 200
Enter the I/O requests (space-separated cylinder numbers): 84 34 64 24 87 56 76 166 131 174 4 3 7 19 198 157 94
Enter the initial direction of head movement ('left' for smaller, 'right' for larger track numbers): right

```

Figure 10: Sample Input for LOOK Algorithm (Right Direction)

```

Order of requests processed: [84, 87, 94, 131, 157, 166, 174, 198, 76, 64, 56, 34, 24, 19, 7, 4, 3]
Total seek time: 315

```

Figure 11: Sample Output for LOOK Algorithm (Right Direction)

The LOOK algorithm for the right direction works similarly to the left direction. Figure 12 shows the seek sequence for the rightward movement is [84, 87, 94, 131, 157, 166, 174, 198, 76, 64, 56, 34, 24, 19, 7, 4, 3]. The total seek time is 315 ms.

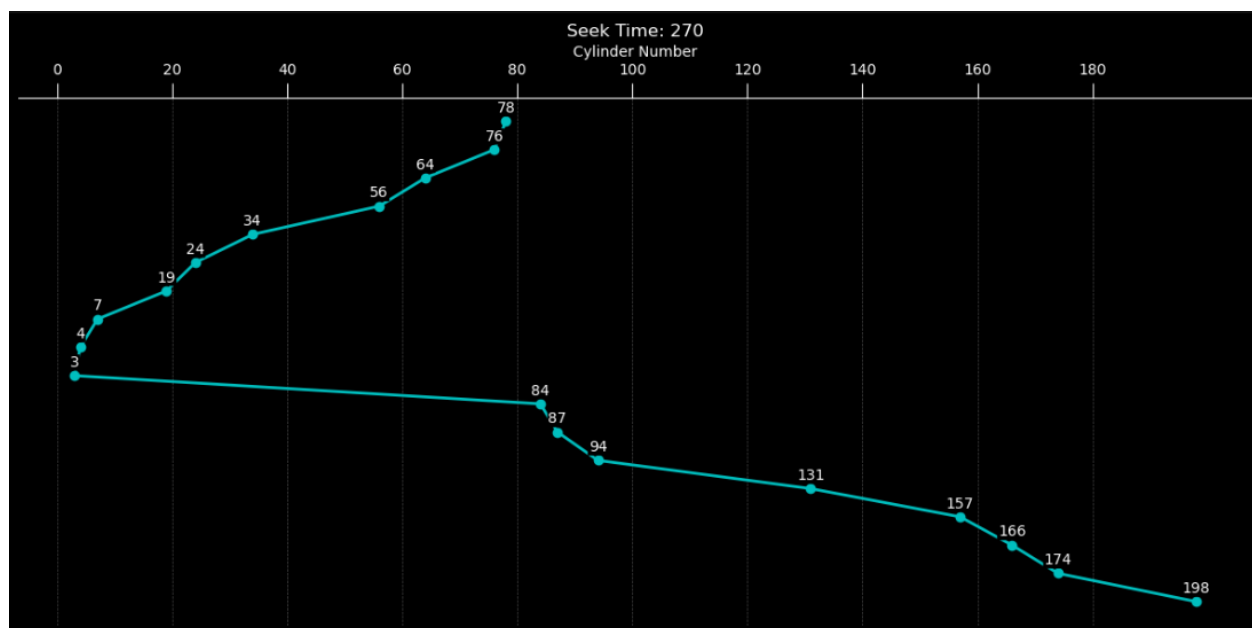


Figure 12: Disk head movement plot for the LOOK algorithm (Left Direction)

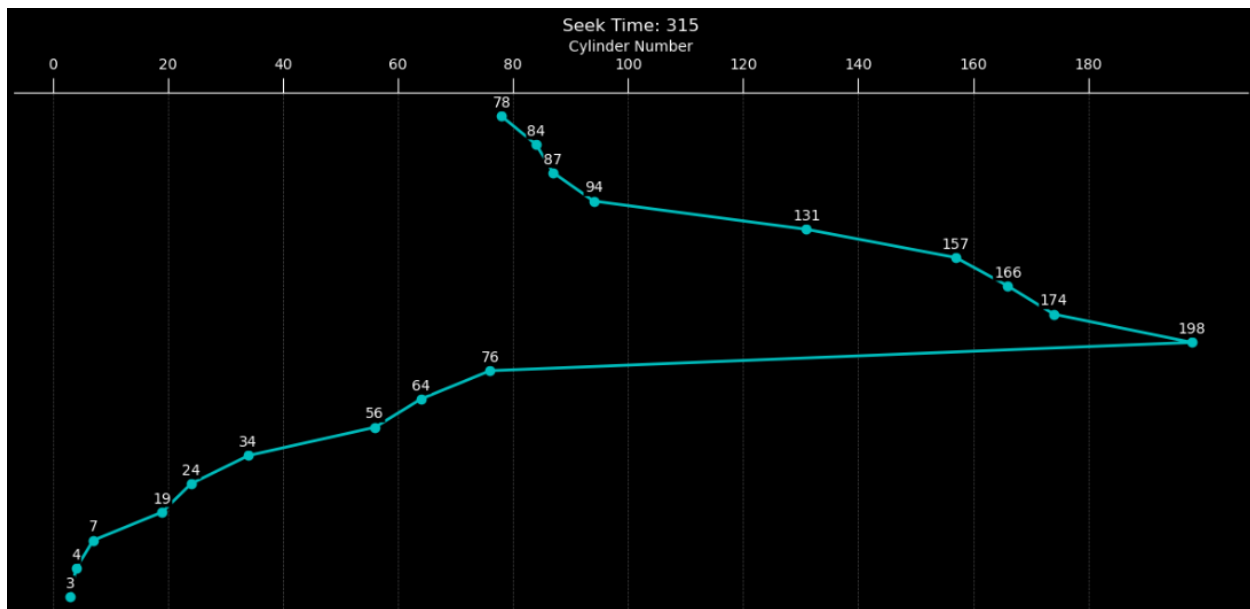


Figure 13: Disk head movement plot for the LOOK algorithm (Right Direction)

The plot shows how the disk head moves from one track to another as per the specified sequence, providing a clear understanding of the disk head movement. The cylinder numbers are annotated above each marker on the plot, and the total seek time is displayed in the title.

3.2 Performance Comparison

In this performance comparison between the LOOK and SCAN disk scheduling algorithms, we will run a program using the same input values for both algorithms to observe and compare their seek times.

For instance, consider the disk size of 200 cylinders, the initial head position set at 80, and the disk request sequence [54, 125, 158, 100, 153, 115, 51, 49, 188, 22, 176, 4, 26, 185, 91]. The program will compute the total seek time for both the LOOK and SCAN algorithms, allowing us to compare how each algorithm handles disk requests.

After running the program for both algorithms, we got the following results based on the total seek time of the SCAN and LOOK Algorithms respectively we got 314ms (Right), 268ms (Left), 292ms (Right) and 260ms (Left).

	SCAN		LOOK	
Head Position	80			
Total No. of cylinders	200			
Seek request	[54, 125, 158, 100, 153, 115, 51, 49, 188, 22, 176, 4, 26, 185, 91]			
Head direction	Right	Left	Right	Left
Sorted Seek request	[91, 100, 115, 125, 153, 158, 176, 185, 188, 199, 54, 51, 49, 26, 22, 4]	[54, 51, 49, 26, 22, 4, 0, 91, 100, 115, 125, 153, 158, 176, 185, 188]	[91, 100, 115, 125, 153, 158, 176, 185, 188, 54, 51, 49, 26, 22, 4]	[54, 51, 49, 26, 22, 4, 91, 100, 115, 125, 153, 158, 176, 185, 188]
Total seek time	314 ms	268 ms	292 ms	260 ms

Table 2: Input and Comparison Table of SCAN and LOOK

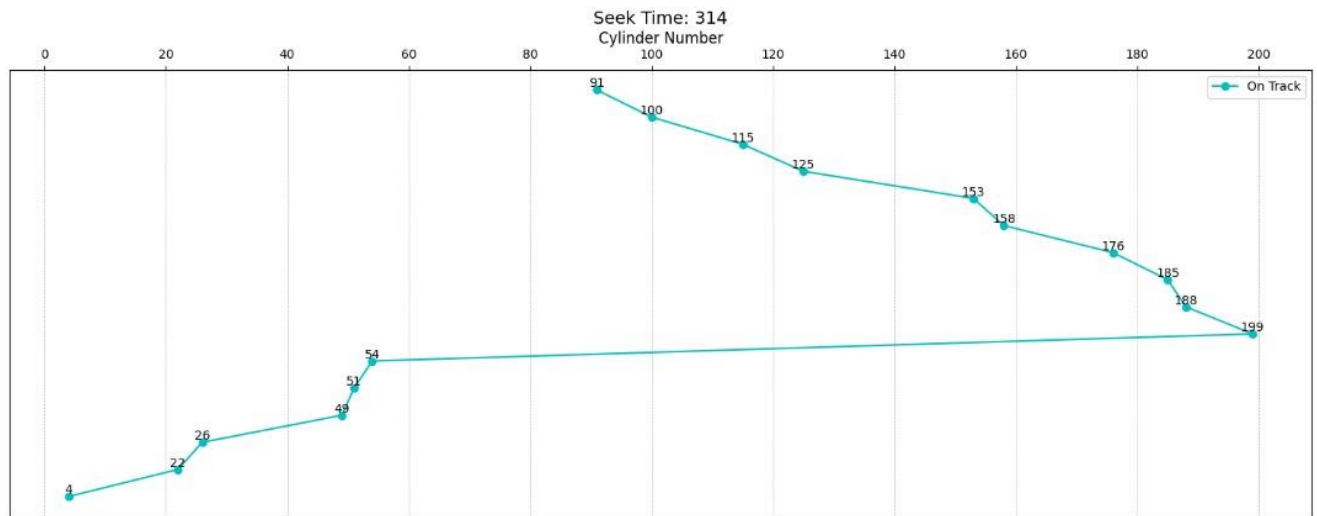


Figure 14: Disk head movement plot for the SCAN Comparison (Right Direction)

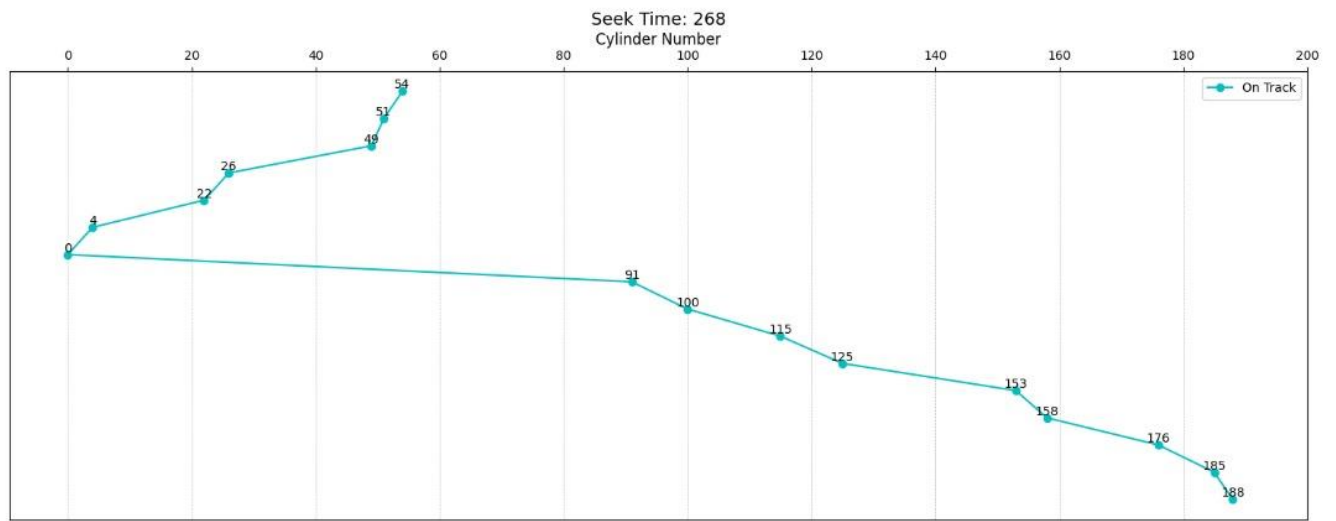


Figure 15: Disk head movement plot for the SCAN Comparison (Left Direction)

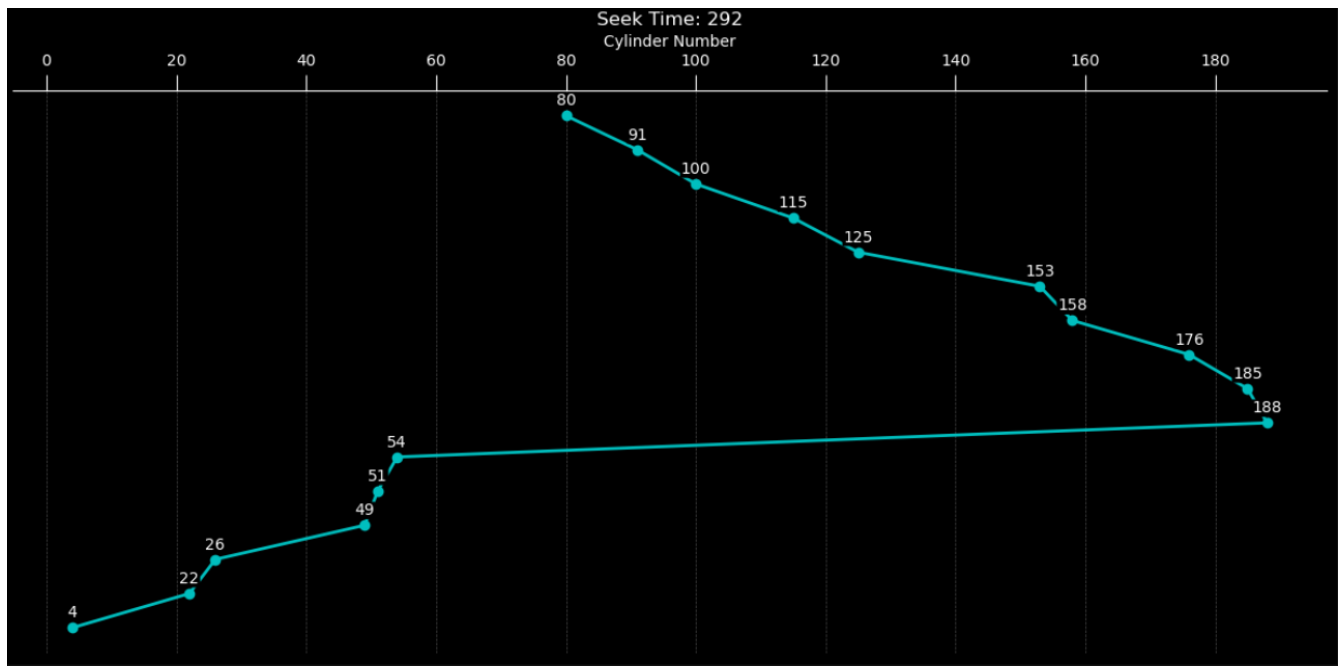


Figure 16: Disk head movement plot for the LOOK Comparison (Right Direction)

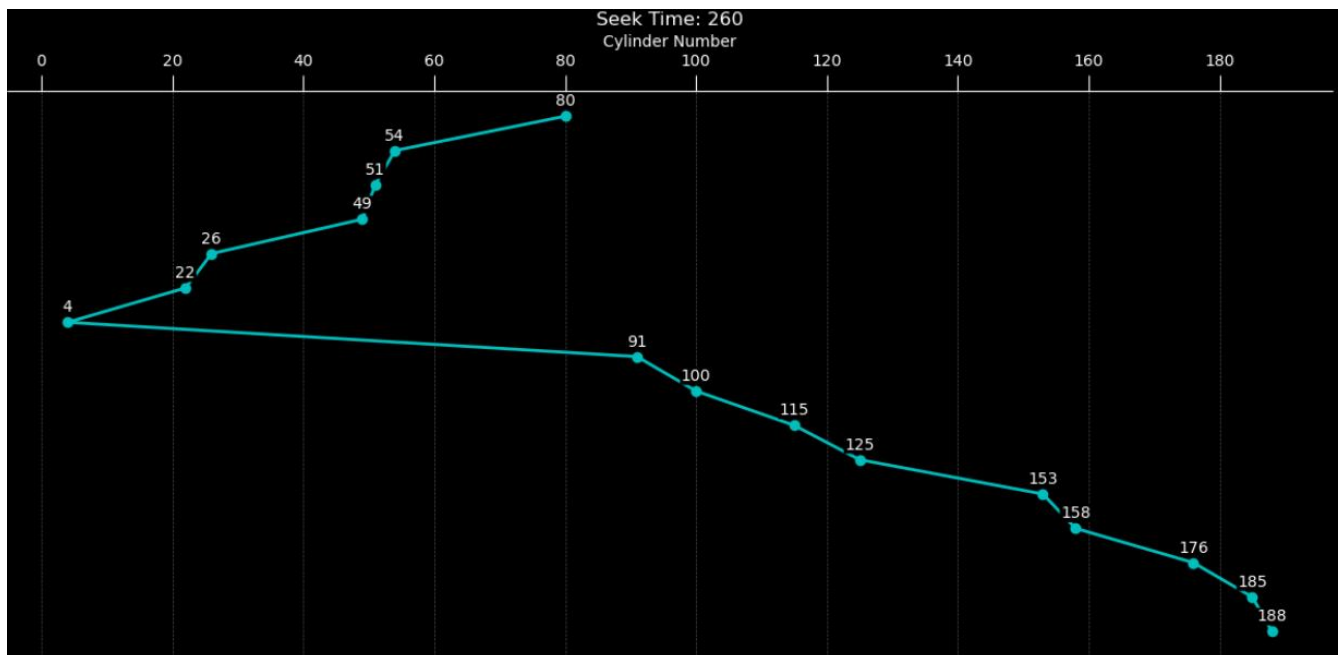


Figure 17: Disk head movement plot for the LOOK Comparison (Left Direction)

By referring to the implementation of each algorithm which are SCAN and LOOK using Python, we can analyse their performance based on total seek time. From the results of our implementation, it is clear that the total seek time varies depending on the algorithms and the distribution of random requests.

After comparing the SCAN and LOOK algorithms, we can observe the following key points. When the head direction is set to right, the SCAN algorithm results in the highest seek time of 314 ms, as the head travels to the furthest cylinder (199) before reversing. In contrast, the LOOK algorithm, when moving right, avoids traveling all the way to the end, reducing the seek time to 292 ms.

When the head direction is set to left, the seek times decrease for both algorithms. SCAN achieves a seek time of 268 ms, as it moves towards the lower end of the disk, after servicing the last request at 4, it continues its traversal to the end disk (cylinder 0). LOOK, however, achieves a total seek time of 260 ms, just 8 ms short of SCAN. It performs the best in this direction, with the lowest seek time because, it stops at the last request (4) without continuing to the extreme of 0.

Overall, the LOOK algorithm outperforms SCAN in all cases. Thus, demonstrating that the LOOK algorithm is more efficient, as it avoids unnecessary traversal to the end of the disk, thus reducing seek time significantly.

In summary, SCAN and LOOK algorithms offer trade-offs in terms of simplicity, fairness, and seek-time efficiency. While SCAN ensures simplicity and fairness by uniformly servicing all requests, LOOK is suitable for workloads where minimizing latency is critical. Further experimentation and parameter tuning may be required to fully optimize performance in real-world scenarios

3.4 Strengths vs Weaknesses

	SCAN	LOOK
Strengths	<p>Systematic and Predictable: The SCAN algorithm follows a clear, sweeping motion like an elevator, making it easy to implement.</p> <p>Fair Request Servicing: It ensures that the requests are serviced without starvation, as the head moves back and forth across the disk.</p> <p>Handles Heavy Workloads: It efficiently handles a high volume of requests by systematically servicing them in order.</p>	<p>Reduced seek time: The LOOK algorithm minimizes seek time and unnecessary movement by moving the arm only to the last request in each direction.</p> <p>Efficient: The LOOK algorithm enhances performance by only going to the tracks with requests, which is useful in systems with copious disk operations.</p> <p>Fairness: The LOOK algorithm warrants fairness by preventing starvation, making sure that all requests are eventually serviced. (Naukri, n.d.)</p>
Weaknesses	<p>Edge Bias: Requests near the edges of the disk may experience longer waiting times.</p> <p>Unnecessary Movement to Edges: The head continues to the disk's physical edge even if there are no requests there, which wastes time and reduces efficiency.</p> <p>Higher Seek Time Compared to LOOK: Compared to other algorithms, which only go as far as the farthest request, SCAN can have a longer seek time because it always goes to the edge.</p>	<p>Variable Response Time: The time taken to service a request can vary significantly, especially if a request is just missed by the disk head in its current sweep.</p> <p>Not Optimal for Sparse Requests: In cases where disk requests are infrequent or widely distributed, LOOK may not offer significant improvements over simpler strategies.</p> <p>Non-uniform waiting times: The LOOK algorithm can have non-uniform waiting times. (Naukri, n.d.)</p>

Table 3: Strengths and Weaknesses of SCAN and LOOK

3.5 Personal Observation and Conclusion

After playing around with the values, like the initial head position it was observed that the seek time decreases significantly when the starting point is close to the majority of the requests. The distribution of the requests also plays a critical role. When requests are clustered, the LOOK algorithm performs efficiently, resulting in significantly reduced seek times. In contrast, a more scattered distribution and a distribution with many points near either ends of the disk tends to increase the total seek time, demonstrating the importance of the data's structure in optimizing disk scheduling.

Whereas, the SCAN algorithm performs well in scenarios where disk requests are evenly distributed and when edge traversal is not a concern. It ensures fairness by servicing all requests systematically as it sweeps through the disk in one direction before reversing. However, its tendency to move to the edge of the disk, even if there are no requests, which can result in longer seek times compared to LOOK.

In conclusion, SCAN provides predictable and systematic request servicing, making it more suitable for evenly distributed workloads. On the other hand, LOOK provides efficiency, making it ideal for scenarios where efficiency and quick response are critical. The choice between these algorithms depends on the specific requirements of the system, such as the need for reduced seek time or balanced servicing of all requests. Both algorithms have their strengths and are better suited to different types of workloads.

References

Shiksha. (n.d.). *Semaphore in Operating System: What is it and why is it needed?* Retrieved October 13, 2023, from <https://www.shiksha.com/online-courses/articles/semaphore-in-operating-system/#Why-Semaphore-needed>

GeeksforGeeks. (n.d.). *Difference between Counting and Binary Semaphores*. Retrieved October 3, 2024, from <https://www.geeksforgeeks.org/difference-between-counting-and-binary-semaphores/>

Workat.Tech. (n.d.). *Disk Scheduling Algorithms in Operating System*. Retrieved October 7, 2024, from <https://workat.tech/core-cs/tutorial/disk-scheduling-algorithms-in-operating-system-os-ope5ahnn6mhh>

Naukri.com. (n.d.). *SCAN Disk Scheduling Algorithm*. Retrieved October 7, 2024, from <https://www.naukri.com/code360/library/scan-disk-scheduling-algorithm>

GeeksforGeeks. (2015, October 13). *Disk Scheduling Algorithms*. Retrieved from <https://www.geeksforgeeks.org/disk-scheduling-algorithms/>

Naukri.com. (n.d.). *LOOK Disk Scheduling*. Retrieved from <https://www.naukri.com/code360/library/look-disk-scheduling>

GeeksforGeeks. (2023, September 6). *LOOK disk scheduling algorithm*. GeeksforGeeks. <https://www.geeksforgeeks.org/look-disk-scheduling-algorithm/>

Javatpoint. (n.d.). *SSTF vs LOOK Disk Scheduling Algorithm*. Retrieved from <https://www.javatpoint.com/sstf-vs-look-disk-scheduling-algorithm>

GeeksforGeeks. (n.d.). *Difference Between SCAN and LOOK Disk Scheduling Algorithms*. Retrieved from <https://www.geeksforgeeks.org/difference-between-scan-and-look-disk-scheduling-algorithms/>

Wahaj, S. (2023, August 25). *Disk Scheduling: LOOK and CLOOK Algorithms*. Techkluster. Retrieved from <https://techkluster.com/algorithm/disk-scheduling-look-clook/>

YouTube. (n.d.). *Disk Scheduling: LOOK and CLOOK Algorithm*. Retrieved from https://www.youtube.com/watch?v=Q2qcqX_hvR0

Teorey, T. J., & Pinkerton, T. B. (1971). A comparative analysis of disk scheduling policies. *Proceedings of the 3rd Symposium on Operating Systems Principles*, 114–121. Digital Systems Lab., Stanford University, Stanford, Calif. <https://dl.acm.org/doi/pdf/10.1145/361268.361278>

Teresco, T. (n.d.). *Files and Disk Scheduling*. Retrieved from https://courses.teresco.org/cs432_f02/lectures/17-files/17-files.html

5. Appendices

LOOK Algorithm Code

```
import matplotlib.pyplot as plt

# Sort requests
requests = sorted(requests)

# Split requests into left and right of the initial head position
left = [req for req in requests # for left
        if
            req < initial_head]

right = [req for req in requests #for right
         if
            req >= initial_head]

#order of requests
if initial_direction == 'right':
    order = right + left[::-1] #request to right processed and reverse (-1)
else: # initial_direction == 'left'
    order = left[::-1] + right # request to left and reverse

# Calculate total seek time
seek_time = 0
current_position = initial_head

positions = [initial_head] # List to Track position

#calculate total time
for req in order:
```

```

        seek_time += abs(req - current_position) #dist b/w urrent and req (abs used for absolute
value)

        current_position = req #update new pos
        positions.append(req)

    return order, seek_time, positions

# graph function with 2 parameters
def plot_disk_movement(positions, seek_time, save_path=None): #positions list of cylinder
positions visited by the disk head during the algorithm execution
    # total seek time for the disk head during the process

    plt.figure(figsize=(12, 6), facecolor='#000000') # creates a figure with a specific size for the
plot in width and length respectively
    #plt.figure(facecolor='#000000')

    # the track number, the order disk head visits each track, circular markers, color of the line,
solid line and label
    plt.plot(positions, range(len(positions)-1, -1, -1), marker='o', color='c', linestyle='-'
',linewidth=2, label='On Track')

    # Annotate each position
    #Loops through each position and its index.
    for i, pos in enumerate(positions):
        #dds a text annotation at each position
        plt.text(pos, len(positions) - i - 1 + 0.2, f'{pos}', fontsize=10 ## Add an offset (+ 0.2) to
move the annotation above
            , ha='center', va='bottom', color='white',
            bbox=dict(
                facecolor='black',
                boxstyle='round,pad=0.001'))

```

```

# sets the title and labels
plt.title(f'Seek Time: {seek_time}', color='white')
plt.xlabel('Cylinder Number', color='white')
# shows grid lines on both axes
plt.grid(True, which='both', linestyle='--', linewidth=0.5, color='white', alpha=0.3) # Added
alpha for transparency

# Gets the current axes
ax = plt.gca()

#Customizes the x-axis tick ( Hides the x-axis ticks and bottom line and moves to the top)
ax.tick_params(axis='x',length=10,      which='both',  labelbottom=False,  bottom=False,
top=True, labeltop=True, color='white')

# Moves the x-axis label to the top
ax.xaxis.set_label_position('top')

#Hides the y-axis
plt.gca().axes.get_yaxis().set_visible(False)

ax.set_facecolor("Black")

ax.spines['top'].set_color('white')
# Set x-axis ticks in increments of 20
max_pos = max(positions) # Finds the maximum cylinder number visited by the disk head
ticks = range(0, max_pos + 1, 20) #Creates a range for the x-axis ticks from 0 to the maximum
track request number, with a step size of 20. every 20 tracks.
plt.xticks(ticks, color='white')

#displays the legend, adjusts layout and shows final plot

```

```

plt.legend()
plt.tight_layout()
plt.show()

# Inputs
initial_head = int(input("Enter the disk's initial head position: "))
total_cylinders = int(input("Enter the total number of cylinders on the disk: "))
requests = list(map(int, input("Enter the I/O requests (space-separated cylinder numbers): ").split())) #convert them into a list of integers
initial_direction = input("Enter the initial direction of head movement ('left' for smaller, 'right' for larger track numbers): ").strip().lower() #turns any input to lowercase

# Input Validation
if initial_head < 0 or initial_head >= total_cylinders:
    print("Error: Initial head position must be within the range of 0 to total_cylinders - 1.")
elif any(req < 0 or req >= total_cylinders for req in requests):
    print("Error: All requests must be within the range of 0 to total_cylinders - 1.")
elif initial_direction not in ['left', 'right']:
    print("Error: Initial direction must be either 'left' or 'right'.")
#If all inputs are valid
else:
    #function calls; runs algorithm if inputs valid
    order, seek_time, positions = look_disk_scheduling(initial_head, total_cylinders, requests, initial_direction)
    # Output results
    print("-----")
    print("Order of requests processed:", order)
    print("Total seek time:", seek_time)

    # Plot the disk head movement
    plot_disk_movement(positions, seek_time)

```

SCAN (Elevator) Algorithm Code

```
import matplotlib.pyplot as plt

#SCAN (ELEVATOR)ALGORITHM PYTHON CODE

def SCAN_disk_scheduling(requests, head, disk_size, direction):

    seek_count = 0
    current_track = 0

    # Separate requests into left and right based on the head position
    left = []
    right = []
    for request in requests:
        if request < head:
            left.append(request)
        elif request > head:
            right.append(request)

    # Sort the requests in both directions
    left.sort()
    right.sort()

    # Adding edge points(0 and 199(disk_size - 1)to simulate full SCAN
    if direction == "left":
        left.insert(0, 0) # Include 0 if moving left
    elif direction == "right":
        right.append(disk_size - 1) # Include disk_size - 1 if moving right

    seek_sequence = []
```

```

run = 2 # Run twice to cover both directions
while run:
    if direction == "left":
        # Process requests to the left
        for i in range(len(left) - 1, -1, -1):
            current_track = left[i]
            seek_sequence.append(current_track)
            seek_count += abs(current_track - head)
            head = current_track
        direction = "right" # Switch direction
    elif direction == "right":
        # Process requests to the right
        for i in range(len(right)):
            current_track = right[i]
            seek_sequence.append(current_track)
            seek_count += abs(current_track - head)
            head = current_track
        direction = "left" # Switch direction

    run -= 1

return seek_sequence, seek_count

# graph function with 2 parameters
def plot_disk_movement(positions, seek_time): #positions list of cylinder positions visited by
the disk head during the algorithm execution
    # total seek time for the disk head during the process

    plt.figure(figsize=(15, 6)) # creates a figure with a specific size for the plot in width and
length respectively

```



```
# the track number, the order disk head visits each track, circular markers, color of the line,
solid line and label
```

```
plt.plot(positions, range(len(positions)-1, -1, -1), marker='o', color='c', linestyle='-',
label='On Track')
```

```
# Annotate each position
```

```
#Loops through each position and its index.
```

```
for i, pos in enumerate(positions):
```

```
    plt.text(pos, len(positions) - i - 1, f'{pos}', fontsize=10, ha='center', va='bottom')
```

```
# sets the title and labels
```

```
plt.title(f'Seek Time: {seek_time}', fontsize=14)
```

```
plt.xlabel('Cylinder Number', fontsize=12)
```

```
plt.ylabel('Movement Order', fontsize=12)
```

```
# shows grid lines on both axes
```

```
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
```

```
# Move x-axis labels to the top
```

```
ax = plt.gca()
```

```
ax.tick_params(axis='x', which='both', labelbottom=False, bottom=False, top=True,
labeltop=True)
```

```
ax.xaxis.set_label_position('top')
```

```
# Hide the y-axis
```

```
plt.gca().axes.get_yaxis().set_visible(False)
```

```
# Set x-axis ticks in increments of 5
```

```

max_pos = max(positions) # Finds the maximum cylinder number visited by the disk head
ticks = range(0, max_pos + 20, 20) #Creates a range for the x-axis ticks from 0 to the
maximum track request number, with a step size of 20. every 20 tracks.

plt.xticks(ticks)

#displays the legend, adjusts layout and shows final plot
plt.legend()
plt.tight_layout()
plt.show()

def main():
    print("SCAN Disk Scheduling Simulation")

    # Input:Initial head position
    head = int(input("Enter the initial head position: "))
    if head < 0:
        print("Error: Head position must be a non-negative integer.")
        return

    # Input:Total number of cylinders on the disk
    disk_size = int(input("Enter the total number of cylinders on the disk: "))
    if disk_size <= 0:
        print("Error: Disk size must be a positive integer.")
        return

    # Input: Number of I/O requests
    n = int(input("Enter the number of I/O requests: "))
    if n <= 0:
        print("Error: Number of requests must be a positive integer.")
        return

```

```

# Input:I/O requests
requests = []
print("Enter the I/O requests: ")
for i in range(n):
    req = int(input())
    if req < 0 or req >= disk_size:
        print(f'Error: Request {req} is invalid. It must be in the range [0, {disk_size - 1}].')
        return
    requests.append(req)

# Input:Initial direction
direction = input("Enter the initial direction (left or right): ").lower()
if direction not in ["left", "right"]:
    print("Error: Invalid direction. Please enter 'left' or 'right'.")
    return

# SCAN(Elevator) Algorithm Output
scan_sequence, scan_seek_count = SCAN_disk_scheduling(requests, head, disk_size,
direction)

print("\nSCAN (Elevator) Disk Scheduling:")
print("Seek Sequence Order:", scan_sequence)
print("Total Seek Count:", scan_seek_count, "NS")
plot_disk_movement(scan_sequence, scan_seek_count)

if __name__ == "__main__":
    main() ## Run main() only if the script is executed directly

```

Table of Figures and Tables

Table	<i>Page</i>
Table 1: Comparison between Binary and Counting Semaphore	5-6
Table 2: Input and Comparison Table of SCAN and LOOK	19
Table 3: Strengths and Weaknesses of SCAN and LOOK	23
Figure	<i>Page</i>
Figure 1: Comment for each line of the code	7
Figure 2: A Sample Input of SCAN Algorithm (Left Direction).	12
Figure 3: A Sample Input of SCAN Algorithm (Right Direction).	13
Figure 4: A Sample Output of SCAN Algorithm (Left Direction).	14
Figure 5: A Sample Output of SCAN Algorithm (Right Direction).	14
Figure 6: Graph Implementation of SCAN (Left Direction).	15
Figure 7: Graph Implementation of SCAN (Right Direction).	15
Figure 8: Sample Input for LOOK Algorithm (Left Direction)	16
Figure 9: Sample Output for LOOK Algorithm (Left Direction)	16
Figure 10: Sample Input for LOOK Algorithm (Right Direction)	17
Figure 11: Sample Output for LOOK Algorithm (Right Direction)	17
Figure 12: Disk head movement plot for the LOOK algorithm (Left Direction)	17
Figure 13: Disk head movement plot for the LOOK algorithm (Right Direction)	18
Figure 14: Disk head movement plot for the SCAN Comparison (Right Direction)	20
Figure 15: Disk head movement plot for the SCAN Comparison (Left Direction)	20
Figure 16: Disk head movement plot for the LOOK Comparison (Right Direction)	21
Figure 17: Disk head movement plot for the LOOK Comparison (Left Direction)	21