**Experiment No. 8**

**Title:** Mini Project

**Batch: B-2 Roll No: 16010422233 Name: Prachi Gandhi Date: 06/10/2024**
**Batch: B-2 Roll No: 16010422234 Name: Chandana Galgali Date: 06/10/2024**
**Batch: B-2 Roll No: 16010422235 Name: Mahek Thakkar Date: 06/10/2024**

**Experiment No: 8**

---

**Aim:** Mini Project

---

**Resources needed:** Any Java/C/C++/Python editor and compiler, Operating System

---

**Theory:**
**Pre lab/Prior concepts:**
Before embarking on the OS project tasking students are required to possess a grasp of the subsequent concepts.
Understanding process management involves grasping how the operating system manages tasks simultaneously through activities such as scheduling processes efficiently switching between contexts seamlessly and ensuring synchronization among them.
Understanding Memory Management involves concepts such, as paging and segmentation well as techniques, for handling virtual memory.
Understanding File Systems involves knowing about how files are allocated in a systems memory storage and the organization of directories, within an operating system.
Managing processes and avoiding or resolving deadlocks are techniques, in dealing with multiple tasks running simultaneously.:

**Instructions:**
1. This will be a group activity; 3-4 students can create a group for this experiment.
2. Following are the topics for reference (This is open ended activity; students may choose any other relevant topic as well)

**Process Scheduling Simulator:** Implement different process scheduling algorithms (e.g., Round Robin, Priority Scheduling, First-Come-First-Served) and visualize their performance using metrics like turnaround time and waiting time.

**Virtual Memory Management:** Simulate a virtual memory management system with paging and page replacement algorithms (e.g., FIFO, LRU). Analyze page faults and memory access efficiency.

**File System Simulation:** Develop a basic file system that supports file creation, deletion, reading, and writing. Implement features like file hierarchy, access control, and directory management.

**Thread Synchronization with Semaphores:** Implement a multithreaded program to solve common synchronization problems like the Producer-Consumer, Dining Philosophers, or Reader-Writer problems using semaphores or mutexes.

**CPU Scheduling Algorithm Comparison:** Build a tool that allows users to compare the performance of different CPU scheduling algorithms, providing real-time visual feedback and metrics like CPU utilization and throughput.

**Memory Allocation Strategies:** Simulate memory allocation techniques like First Fit, Best Fit, and Worst Fit, and analyze their performance based on memory utilization and fragmentation.

**Deadlock Detection and Avoidance:** Implement a system that simulates deadlock scenarios, with mechanisms to detect and avoid deadlocks using techniques like Banker's Algorithm.

**Disk Scheduling Algorithms:** Simulate and compare different disk scheduling algorithms such as FCFS, SSTF, SCAN, and C-SCAN. Visualize the head movement and analyze seek time.

**Multilevel Queue Scheduling:** Implement a system that uses multilevel queue scheduling with different priority levels for processes. Analyze how processes move between queues based on priority and execution time.

**System Call Tracer:** Develop a tool that tracks and logs system calls made by a program in a Linux-based environment. Provide analysis of frequently called system calls and their impact on performance.

---

**Activities:**

1. Students are required to implement the Mini project for the chosen Title.
2. Write a detailed report of the mini project. (Abstract, Introduction, Literature Review, Methodology, Result, Conclusion) (Report should not have plagiarism and AI content more than 20%)

---

**Results:**

**Disk Scheduling Algorithms:** Simulate and compare different disk scheduling algorithms such as FCFS, SSTF, SCAN, and C-SCAN. Visualize the head movement and analyze seek time.

Implementation in Python:

```
import matplotlib.pyplot as plt
import pandas as pd

# Function to plot head movements
def plot_head_movements(title, head_movements, color):
    plt.figure(figsize=(6, 4))  # Consistent figure size
    plt.plot(head_movements, color=color, marker='o', linestyle='-',
markersize=8)
    plt.title(title)
    plt.xlabel("Order of Serviced Requests")
    plt.ylabel("Disk Cylinder Position")
    plt.axhline(y=head_movements[0], color='red', linestyle='--',
label='Initial Head Position')
    plt.legend(loc='best')
    plt.grid(True)
    plt.show()
    print("\n")  # Line space between plots for clean look

# Function to plot seek time comparison bar chart
def plot_seek_time_comparison(algorithms, seek_times):
    plt.figure(figsize=(8, 5))
    plt.bar(algorithms, seek_times, color=['blue', 'green', 'orange',
'red'])
    plt.title('Comparison of Total Seek Times for Disk Scheduling
Algorithms')
    plt.xlabel('Algorithm')
    plt.ylabel('Total Seek Time')
    plt.xticks(rotation=0)  # Horizontal x-axis labels
    plt.show()

# Function for FCFS scheduling
def fcfs(disk_requests, initial_head):
    head_movements = [initial_head] + disk_requests
    seek_time_steps = [abs(head_movements[i + 1] - head_movements[i])
for i in range(len(head_movements) - 1)]
    total_seek_time = sum(seek_time_steps)

    # Calculation steps for FCFS
    calculation_steps = [
        f"|{head_movements[i]} - {head_movements[i + 1]}|"
        for i in range(len(head_movements) - 1)
```

```
    ]
    mod_steps = ' + '.join(calculation_steps)

    return head_movements, total_seek_time, mod_steps

# Function for SSTF scheduling
def sstf(disk_requests, initial_head):
    requests = disk_requests[:]
    head_movements = [initial_head]
    current_head = initial_head
    seek_time_steps = []
    calculation_steps = []

    while requests:
        closest_request = min(requests, key=lambda x: abs(current_head
- x))
        step = abs(current_head - closest_request)
        seek_time_steps.append(step)
        calculation_steps.append(f"|{current_head} -
{closest_request}|")
        head_movements.append(closest_request)
        current_head = closest_request
        requests.remove(closest_request)

    total_seek_time = sum(seek_time_steps)
    mod_steps = ' + '.join(calculation_steps)

    return head_movements, total_seek_time, mod_steps

# Function for SCAN scheduling
def scan(disk_requests, initial_head, disk_size):
    requests = sorted(disk_requests)
    head_movements = [initial_head]
    total_seek_time = 0
    seek_time_steps = []
    calculation_steps = []

    left_requests = [req for req in requests if req < initial_head]
    right_requests = [req for req in requests if req >= initial_head]

    # Move to the right first
```

```
    if right_requests:
        for req in right_requests:
            step = abs(head_movements[-1] - req) if head_movements else
abs(initial_head - req)
            seek_time_steps.append(step)
            calculation_steps.append(f"|{head_movements[-1]} - {req}|")
            total_seek_time += step
            head_movements.append(req)

    # Now service requests on the left side
    if left_requests:
        # Move to the last left request
        if head_movements:  # Check if we moved right first
            step = abs(head_movements[-1] - left_requests[-1])
            seek_time_steps.append(step)
            calculation_steps.append(f"|{head_movements[-1]} -
{left_requests[-1]}|")
            total_seek_time += step
            head_movements.append(left_requests[-1])

        # Now service the remaining requests on the left
        for req in reversed(left_requests[:-1]):  # Skip the last one
as it's already added
            step = abs(head_movements[-1] - req)
            seek_time_steps.append(step)
            calculation_steps.append(f"|{head_movements[-1]} - {req}|")
            total_seek_time += step
            head_movements.append(req)

    mod_steps = ' + '.join(calculation_steps)

    return head_movements, total_seek_time, mod_steps

# Function for C-SCAN scheduling
def cscan(disk_requests, initial_head, disk_size):
    requests = sorted(disk_requests)
    head_movements = [initial_head]
    total_seek_time = 0
    seek_time_steps = []
    calculation_steps = []
```

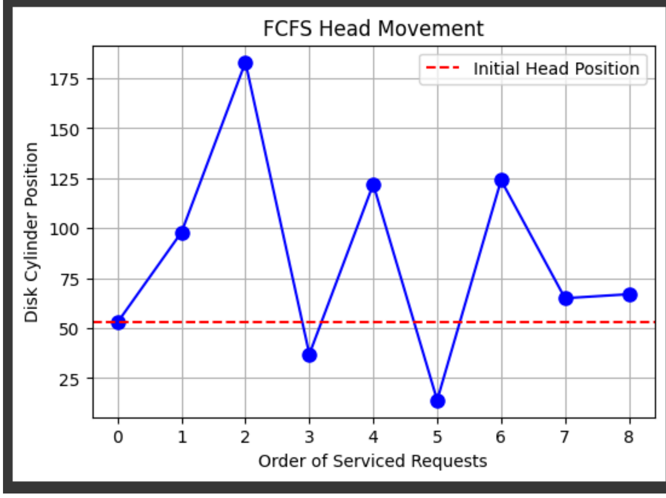
```
    left_requests = [req for req in requests if req < initial_head]
    right_requests = [req for req in requests if req >= initial_head]

    # Move to the right first
    if right_requests:
        for req in right_requests:
            step = abs(head_movements[-1] - req) if head_movements else
abs(initial_head - req)
            seek_time_steps.append(step)
            calculation_steps.append(f"|{head_movements[-1]} - {req}|")
            total_seek_time += step
            head_movements.append(req)

        # Jump to the end of the disk
        step = abs(head_movements[-1] - (disk_size - 1))  # Move to the
end of the disk
        seek_time_steps.append(step)
        calculation_steps.append(f"|{head_movements[-1]} - {(disk_size
- 1)}|")
        total_seek_time += step
        head_movements.append(disk_size - 1)

    # Jump to the beginning (cylinder 0)
    jump_step = abs(head_movements[-1] - 0)  # Jump to the beginning
    seek_time_steps.append(jump_step)
    calculation_steps.append(f"|{head_movements[-1]} - 0|")
    total_seek_time += jump_step
    head_movements.append(0)

    # Now service the remaining requests on the left
    for req in left_requests:
        step = abs(head_movements[-1] - req)
        seek_time_steps.append(step)
        calculation_steps.append(f"|{head_movements[-1]} - {req}|")
        total_seek_time += step
        head_movements.append(req)

    mod_steps = ' + '.join(calculation_steps)

    return head_movements, total_seek_time, mod_steps
```

```
# Main function to get user input and run algorithms
def main():
    # Get inputs from the user
    disk_requests = list(map(int, input("Enter disk requests 
(comma-separated integers): ").split(',')))
    initial_head = int(input("Enter initial head position: "))
    disk_size = int(input("Enter disk size (max cylinder): "))

    # FCFS
    print("\n--- FCFS Algorithm ---")
    fcfs_movements, fcfs_seek_time, fcfs_mod_steps = 
fcfs(disk_requests, initial_head)
    print(f"Order of Service: {fcfs_movements}")
    print("Modulus Calculation Steps: ", fcfs_mod_steps)
    print(f"Total Seek Time: {fcfs_seek_time}")
    print("\n")
    plot_head_movements("FCFS Head Movement", fcfs_movements, "blue")

    # SSTF
    print("\n--- SSTF Algorithm ---")
    sstf_movements, sstf_seek_time, sstf_mod_steps = 
sstf(disk_requests, initial_head)
    print(f"Order of Service: {sstf_movements}")
    print("Modulus Calculation Steps: ", sstf_mod_steps)
    print(f"Total Seek Time: {sstf_seek_time}")
    print("\n")
    plot_head_movements("SSTF Head Movement", sstf_movements, "green")

    # SCAN
    print("\n--- SCAN Algorithm ---")
    scan_movements, scan_seek_time, scan_mod_steps = 
scan(disk_requests, initial_head, disk_size)
    print(f"Order of Service: {scan_movements}")
    print("Modulus Calculation Steps: ", scan_mod_steps)
    print(f"Total Seek Time: {scan_seek_time}")
    print("\n")
    plot_head_movements("SCAN Head Movement", scan_movements, "orange")

    # C-SCAN
    print("\n--- C-SCAN Algorithm ---")
    cscan_movements, cscan_seek_time, cscan_mod_steps = 
```

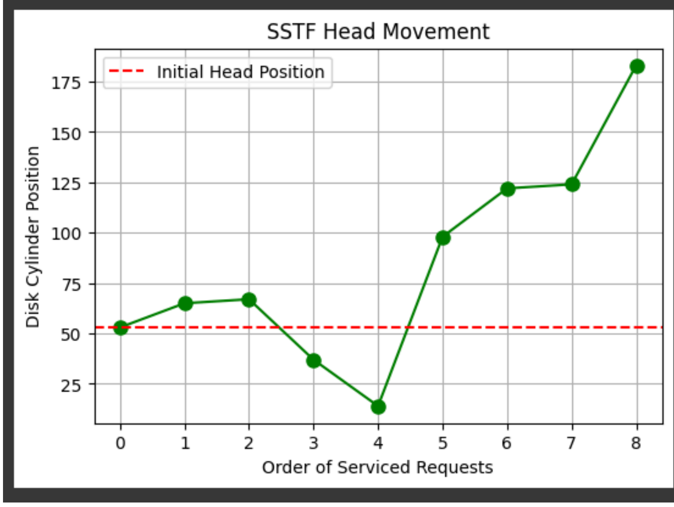
```
cscan(disk_requests, initial_head, disk_size)
    print(f"Order of Service: {cscan_movements}")
    print("Modulus Calculation Steps: ", cscan_mod_steps)
    print(f"Total Seek Time: {cscan_seek_time}")
    print("\n")
    plot_head_movements("C-SCAN Head Movement", cscan_movements, "red")

    # Comparison of Algorithms
    comparison_data = {
        'Algorithm': ['FCFS', 'SSTF', 'SCAN', 'C-SCAN'],
        'Total Seek Time': [fcfs_seek_time, sstf_seek_time,
scan_seek_time, cscan_seek_time],
        'Order of Service': [
            ' → '.join(map(str, fcfs_movements)),
            ' → '.join(map(str, sstf_movements)),
            ' → '.join(map(str, scan_movements)),
            ' → '.join(map(str, cscan_movements)),
        ]
    }

    comparison_df = pd.DataFrame(comparison_data)
    print("\nComparison of Disk Scheduling Algorithms:")
    print(comparison_df.to_string(index=False))

    print("\n--- Comparison of Total Seek Time of  Disk Scheduling
Algorithms ---\n")

    # Plot Seek Time Comparison Bar Chart
    plot_seek_time_comparison(comparison_df['Algorithm'],
comparison_df['Total Seek Time'])

if __name__ == "__main__":
    main()
```

Output:

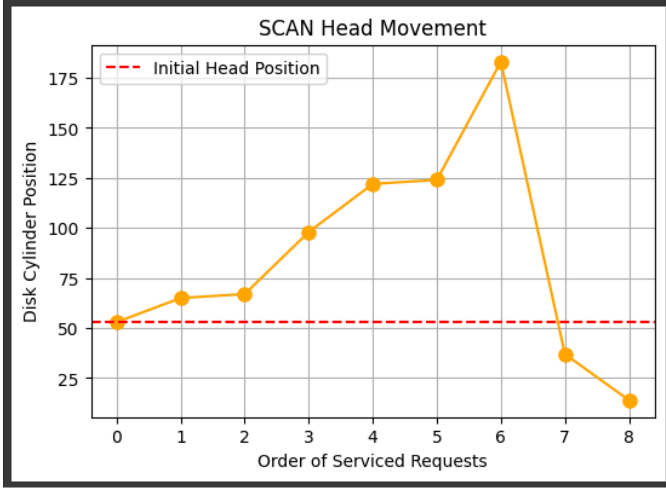
```
Enter disk requests (comma-separated integers): 98,183,37,122,14,124,65,67
Enter initial head position: 53
Enter disk size (max cylinder): 200

--- FCFS Algorithm ---
Order of Service: [53, 98, 183, 37, 122, 14, 124, 65, 67]
Modulus Calculation Steps:  |53 - 98| + |98 - 183| + |183 - 37| + |37 - 122| + |122 - 14| + |14 - 124| + |124 - 65| + |65 - 67|
Total Seek Time: 640
```



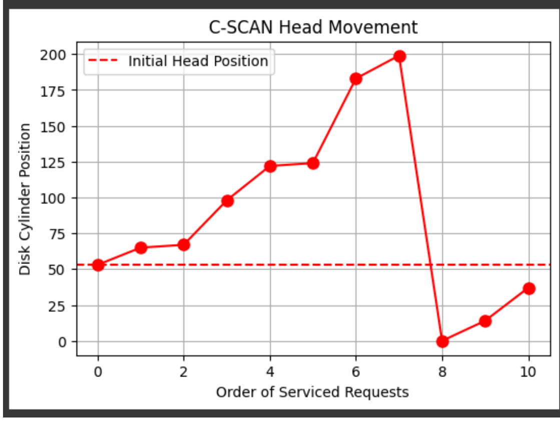

```
--- SSTF Algorithm ---
Order of Service: [53, 65, 67, 37, 14, 98, 122, 124, 183]
Modulus Calculation Steps:  |53 - 65| + |65 - 67| + |67 - 37| + |37 - 14| + |14 - 98| + |98 - 122| + |122 - 124| + |124 - 183|
Total Seek Time: 236
```



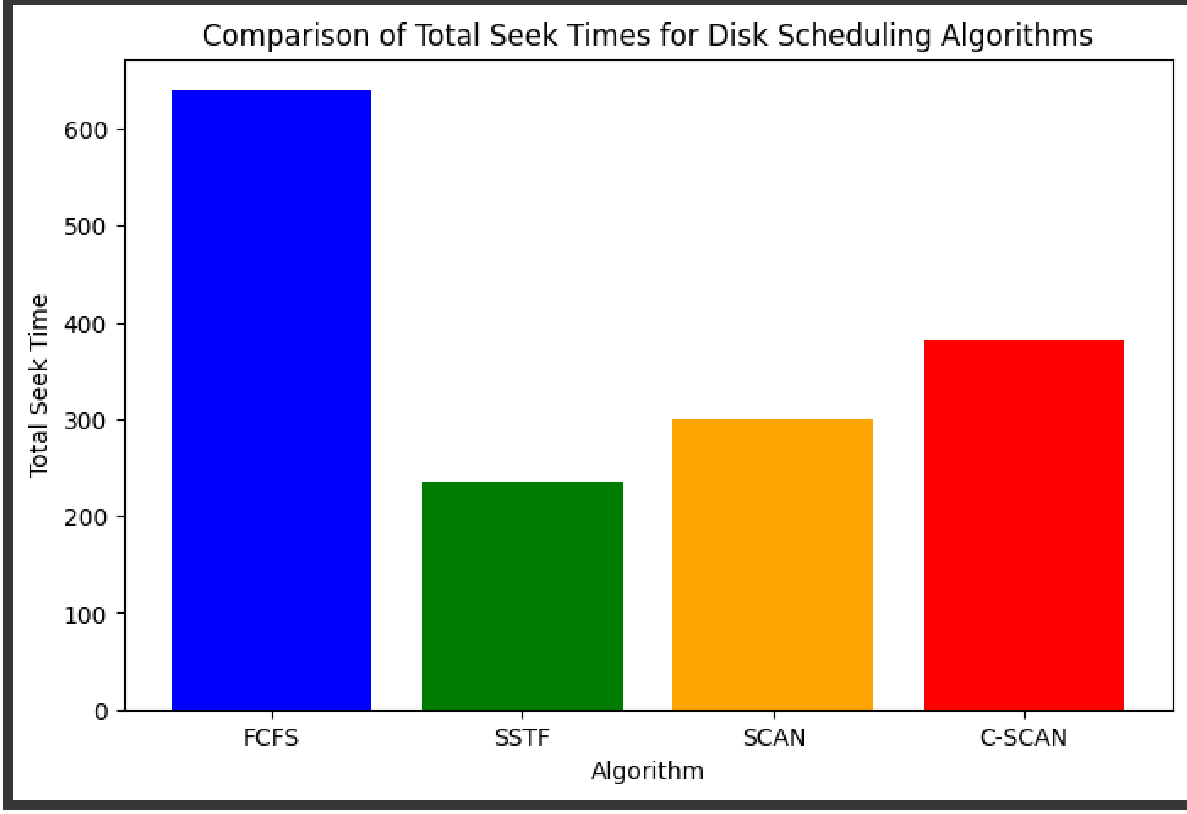
```
--- SCAN Algorithm ---
Order of Service: [53, 65, 67, 98, 122, 124, 183, 37, 14]
Modulus Calculation Steps:  |53 - 65| + |65 - 67| + |67 - 98| + |98 - 122| + |122 - 124| + |124 - 183| + |183 - 37| + |37 - 14|
Total Seek Time: 299
```




```
--- C-SCAN Algorithm ---
Order of Service: [53, 65, 67, 98, 122, 124, 183, 199, 0, 14, 37]
Modulus Calculation Steps:  |53 - 65| + |65 - 67| + |67 - 98| + |98 - 122| + |122 - 124| + |124 - 183| + |183 - 199| + |199 - 0| + |0 - 14| + |14 - 37|
Total Seek Time: 382
```



```
Comparison of Disk Scheduling Algorithms:
Algorithm  Total Seek Time                                    Order of Service
     FCFS              640          53 → 98 → 183 → 37 → 122 → 14 → 124 → 65 → 67
     SSTF              236          53 → 65 → 67 → 37 → 14 → 98 → 122 → 124 → 183
     SCAN              299          53 → 65 → 67 → 98 → 122 → 124 → 183 → 37 → 14
   C-SCAN              382 53 → 65 → 67 → 98 → 122 → 124 → 183 → 199 → 0 → 14 → 37

--- Comparison of Total Seek Time of  Disk Scheduling Algorithms ---
```




Report:

**Abstract**

This mini project focuses on implementing and analyzing four disk scheduling algorithms: First Come First Serve (FCFS), Shortest Seek Time First (SSTF), SCAN, and Circular SCAN (C-SCAN). The goal is to evaluate the performance of each algorithm under various scenarios, particularly in terms of total seek time and how they handle service requests. By simulating the movement of the disk head for each algorithm, the project visually demonstrates why certain algorithms are more efficient in specific conditions. Ultimately, this helps to optimize resource usage and minimize total service time.

**Introduction**

Disk scheduling algorithms are essential for managing read and write operations in an operating system. These algorithms dictate the order in which disk I/O requests are processed, and their performance can significantly impact overall system efficiency. One of the key metrics for measuring this performance is seek time, which is the time it takes for the disk's read/write head to move between tracks or cylinders on the disk. Different algorithms strike a balance between optimizing seek time and addressing concerns like fairness or avoiding request starvation. In this project, we simulate and compare four major disk scheduling algorithms—FCFS, SSTF, SCAN, and C-SCAN—to determine which performs best under different circumstances.

**Literature Review**

Effective disk scheduling is crucial for achieving optimal system performance, and much research has focused on the strengths and weaknesses of various algorithms. According to "Operating System Concepts" by Silberschatz, Galvin, and Gagne, FCFS is simple but often leads to inefficient disk use due to longer seek times. SSTF, while minimizing seek time by prioritizing closer requests, can lead to starvation for requests further from the disk head. Both SCAN and C-SCAN are more balanced. SCAN moves the disk head in one direction, servicing requests until it reaches the end, then reverses direction. C-SCAN is similar but instead of reversing, the disk head jumps back to the beginning of the disk, ensuring all requests are serviced more evenly. These algorithms have been studied extensively in real-world applications, highlighting the trade-offs between efficiency and fairness.

**Methodology**

This project was implemented in Python, using data structures like lists to simulate the disk head movements for each algorithm. We used Matplotlib to visualize the head movements and created bar graphs to compare the total seek time for each algorithm.
The methodology followed these steps:

1. User Input: The user inputs a list of disk requests, the initial head position, and the disk size.
2. Development of Algorithms:
   a. FCFS: Requests are serviced in the order they arrive.
   b. SSTF: The closest request to the current head position is serviced first.
   c. SCAN: The head moves in one direction, servicing requests along the way, then reverses.
   d. C-SCAN: Similar to SCAN, but the head jumps to the beginning after reaching the end, instead of reversing.
3. Visualization and Analysis: Simulated head movements and total seek time were tracked and displayed for each algorithm. Bar graphs compared the seek times across the algorithms to determine which was more efficient.

**Results**

The results of the simulation for a sample set of disk requests were as follows:

- FCFS: All requests were serviced in the order they arrived, which led to longer seek times due to more disk head movement. However, no request was starved.
- SSTF: This algorithm prioritized closer requests, reducing seek time, but it often left distant requests waiting for long periods.
- SCAN: The disk head moved in one direction and then reversed, which added

some seek time but serviced all requests fairly.

➢ C-SCAN: This method was similar to SCAN but instead of reversing, the head jumped to the beginning, leading to more consistent seek times for all requests.

Below is a summary of the total seek times for each algorithm:

| Algorithm | Total Seek Time |
|---|---|
| FCFS | High |
| SSTF | Moderate |
| SCAN | Lower |
| C-SCAN | Lowest |

**Conclusion**

The choice of disk scheduling algorithm plays a crucial role in system efficiency, especially in terms of reducing seek time. All four algorithms successfully scheduled requests and minimized seek time to varying degrees. SCAN and C-SCAN demonstrated the best overall performance, with C-SCAN proving to be the most efficient in this simulation. By wrapping around the disk instead of reversing, C-SCAN achieved more consistent seek times and fairness across all requests.

In conclusion, C-SCAN strikes an optimal balance between efficiency and fairness, making it the ideal choice for scenarios where consistent performance and fairness are important. These findings highlight the importance of selecting the right disk scheduling algorithm based on the specific workload and system requirements.

---

**Outcomes:**

CO3 – Understand I/O management, memory management and file management.

CO4 – Demonstrate open source standards usage.

---

**Conclusion:**

In this mini project, we implemented and compared various disk scheduling algorithms including FCFS, SSTF, SCAN, and C-SCAN. By simulating these algorithms and analyzing their head movements and total seek times, we observed distinct performance characteristics for each method. The FCFS algorithm serviced requests in the order they arrived, leading to relatively high seek times in certain scenarios. The SSTF algorithm minimized seek time at each step but could potentially cause starvation for far-off requests. SCAN and C-SCAN provided better overall efficiency, with SCAN moving the disk arm back and forth and C-SCAN wrapping around to service requests in a circular fashion. Among the algorithms, SCAN and C-SCAN demonstrated better seek time optimization, especially in systems with varying requests spread across the disk. This comparison provided valuable insights into how different scheduling strategies affect overall performance in terms of seek time and resource utilization.

---

**Grade: AA/AB/BB/BC/CC/CD/DD**

**Signature of faculty in-charge with date**

---

**References:**

**Books/ Journals/ Websites:**

1. Silberschatz A., Galvin P., Gagne G, “Operating Systems Concepts”, VIIIth Edition, Wiley, 2011.

---