# DISK SCHEDULING

MINI PROJECT REPORT

By

Arav Goel [RA2211026010349]
Pratham Shrivastav [RA2211026010366]
Himanshu Bhadani [RA2211026010368]

Under the guidance of

**Dr. J.J JAYAKANTH**

*In partial fulfilment for the Course*

of

**21CSC202J-Operating Systems**

in **COMPUTATIONAL INTELLIGENCE**

**FACULTY OF ENGINEERING AND TECHNOLOGY**

**SCHOOL OF COMPUTING**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**KATTANKULATHUR**

**NOVEMBER 2023**

# SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

**(Under Section 3 of UGC Act, 1956)**

## BONAFIDE CERTIFICATE

Certified that this minor project report for the course **21CSC202J-OPERATING SYSTEMS** entitled in "**DISK SCHEDULING**" is the bonafide work of **Arav Goel (RA2211026010349), Pratham Shrivastav (RA2211026010366)** and **Himanshu Bhadani (RA2211026010368)** who carried out the work under my supervision.

| | |
|---|---|
| **PROJECT GUIDE** | **HEAD OF THE DEPARTMENT** |
| **Dr. J.J JAYAKANTH** | Dr Annie Uthra |
| **Assistant Professor** | **Professor & Head** |
| **Department of Computational Intelligence** | **Department of Computational Intelligence** |
| SRM Institute of Science and Technology | SRM Institute of Science and Technology |
| Kattankulathur | Kattankulathur |

## ABSTRACT

This project delves into the intricacies of disk scheduling within an operating system, a critical component for optimizing data retrieval and storage efficiency on modern computer systems. The primary focus is on the implementation and evaluation of various disk scheduling algorithms, including but not limited to FCFS (First-Come-First-Serve), SSTF (Shortest Seek Time First), SCAN, C-SCAN, LOOK, and C-LOOK. Through comprehensive simulations and performance analyses, we assess the strengths and weaknesses of each algorithm in terms of throughput, latency, and overall system responsiveness. The project aims to provide valuable insights into the trade-offs inherent in disk scheduling strategies, aiding in the design and optimization of operating systems for enhanced disk I/O performance. This report presents our methodology, findings, and conclusions, offering a valuable resource for future research and development in the field of operating systems and storage management.

## ACKNOWLEDGEMENT

We express our heartfelt thanks to our honorable **Vice Chancellor Dr. C. MUTHAMIZHCHELVAN**, for being the beacon in all our endeavors. We would like to express my warmth of gratitude to our **Registrar Dr. S. Ponnusamy,** for his encouragement.

We express our profound gratitude to our **Dean (College of Engineering and Technology) Dr. T. V.Gopal,** for bringing out novelty in all executions.

We would like to express my heartfelt thanks to Chairperson, School of Computing **Dr. Revathi Venkataraman,** for imparting confidence to complete my course project

We wish to express my sincere thanks to **Course Audit Professors Dr. Vadivu. G , Professor, Department of Data Science and Business Systems and Dr. Sasikala. E Professor, Department of Data Science and Business Systems** and **Course Coordinators** for their constant encouragement and support.

We are highly thankful to our my Course project Faculty **Dr. J.J JAYAKANTH, Assistant Professor , CINTEL,** for his/her assistance, timely suggestion and guidance throughout the duration of this course project.

We extend my gratitude to our **HOD Dr. Annie Uthra, Professor & Head , CINTEL** and my Departmental colleagues for their Support.

Finally, we thank our parents and friends near and dear ones who directly and indirectly contributed to the successful completion of our project. Above all, I thank the almighty for showering his blessings on me to complete my Course project.

## TABLE OF CONTENTS

# 1. INTRODUCTION

In the ever-evolving landscape of computing, the efficient utilization of storage resources remains a paramount concern. Disk scheduling, as a critical aspect of operating system design, plays a pivotal role in orchestrating the sequence in which data requests are serviced from the disk. As data access patterns and workloads vary, the choice of a suitable disk scheduling algorithm becomes instrumental in determining system performance. This project undertakes a comprehensive exploration of disk scheduling algorithms, acknowledging the nuanced challenges posed by real-world scenarios. By investigating and implementing multiple algorithms, we aim to not only understand their individual behaviors but also to compare their efficacy under diverse conditions. The ultimate goal is to contribute to the development of operating systems that can intelligently and adaptively manage disk I/O operations, fostering improved overall system responsiveness and user experience. Through this endeavor, we strive to make a meaningful contribution to the field of operating system optimization and resource management.

## 1.1 Motivation:

The motivation for undertaking this project stems from the ever-growing importance of efficient data management in contemporary computing environments. As data volumes continue to escalate and diverse applications place varied demands on disk resources, the need for optimized disk scheduling algorithms becomes increasingly apparent. Inefficient disk scheduling can lead to performance bottlenecks, degraded system responsiveness, and compromised user experience. By delving into this crucial aspect of operating system functionality, we aim to address these challenges and contribute to the development of more intelligent and adaptive systems. This project seeks to empower operating system designers with insights into the strengths and weaknesses of different disk scheduling strategies, fostering the creation of systems that can dynamically adapt to changing workloads, thereby enhancing overall system efficiency and responsiveness.

### 1.2 Objective:

The objective is to study and improve the performance of specific disk scheduling algorithms, including C-SCAN, C-LOOK, FCFS, and SSTF. We aim to reduce seek times and enhance data access efficiency.

Minimize Seek Time - Reduce the time it takes for the disk's read/write head to reach the requested data location, optimizing I/O efficiency.

Minimize Rotational Latency - Minimize the time it takes for the disk to position the requested data under the read/write head, enhancing access speed.

Reduce I/O Wait Times - Decrease process waiting times, improving overall system performance and responsiveness.

Fair Resource Allocation - Ensure equitable access to the disk in multi-user environments, preventing resource monopolization.

Maximize System Performance - Enhance the overall performance of the operating system by reducing I/O bottlenecks and optimizing data access.

### 1.3 Problem Statement:

**Improving Disk Scheduling Efficiency in Operating Systems**

Disk scheduling in modern operating systems is crucial for optimizing data access on storage devices, such as hard disk drives and SSDs. The current disk scheduling mechanisms in use are becoming increasingly inefficient in managing I/O requests, resulting in suboptimal disk performance. Users experience slow data retrieval and longer wait times, impacting the overall system responsiveness.
These algorithms face challenges in efficiently managing I/O requests, leading to suboptimal disk performance and user experience.
Users, administrators, and organizations relying on efficient data access are affected by the limitations of these disk scheduling algorithms.

### 1.4. Challenges:

Navigating the landscape of disk scheduling algorithms presents several challenges that warrant careful consideration in the course of this project. The inherent trade-offs between optimizing for seek time, throughput, and fairness add complexity to algorithm design and evaluation. Additionally, the diverse nature of data access patterns and the dynamic nature of workloads pose challenges in creating a one-size-fits-all solution. Furthermore, the need to strike a balance between simplicity and sophistication in algorithm implementation adds an extra layer of complexity. Real-world conditions such as varying I/O loads and unpredictable access patterns further contribute to the intricate nature of this endeavor. Our project acknowledges and addresses these challenges, aiming to provide not only a thorough evaluation of existing algorithms but also insights that can guide future developments in disk scheduling strategies, ultimately contributing to the evolution of more robust and adaptive operating systems.

## 2. LITERATURE SURVEY

### 1. Operating System Design:

Explore literature that DISKusses the fundamentals of operating system architecture, processes, and resource management. Understanding the broader context of how disk scheduling fits into the overall operating system framework is essential.

### 2. Disk Scheduling Algorithms:

Review seminal papers and studies on various disk scheduling algorithms, including FCFS, SSTF, SCAN, C-SCAN, LOOK, and C-LOOK. Examine their theoretical foundations, performance characteristics, and trade-offs under different scenarios.

### 3. I/O Optimization Techniques:

Investigate literature on techniques aimed at optimizing I/O operations, including caching strategies, buffering mechanisms, and prefetching algorithms. Understanding how these optimizations interact with disk scheduling can provide valuable insights.

### 4. Performance Evaluation Metrics:

Look into literature that DISKusses methodologies for evaluating the performance of disk scheduling algorithms. Metrics such as throughput, latency, and fairness are crucial for assessing the effectiveness of different algorithms.

### 5. Real-world Case Studies:

Explore any case studies or real-world implementations of disk scheduling in practical scenarios. Understanding how these algorithms perform in actual systems and under varying workloads can offer valuable practical insights.

**6. Adaptive Scheduling and Machine Learning Approaches:**

Investigate recent research on adaptive disk scheduling and machine learning-based approaches for optimizing I/O operations. These cutting-edge techniques may provide innovative solutions to the challenges posed by dynamic workloads.

**7. Historical Development of Disk Scheduling:**

Understand the historical evolution of disk scheduling algorithms to gain insights into the motivations behind their design and improvements over time.

# 3. REQUIREMENTS

## 3.1 Software Requirements:

### 3.1.1. Operating System:

The chosen operating system should have support for disk scheduling. Common examples include Windows, Linux, or a custom-built OS.

### 3.1.2. Programming Language:

You'll need a programming language to implement the disk scheduling algorithms. Common languages for this purpose include C, C++, or Python.

### 3.1.3. Disk Scheduling Simulator:

You may choose to use or develop a disk scheduling simulator to test and evaluate different scheduling algorithms. Simulators can help you assess algorithm performance without the need for physical hardware.

### 3.1.4. Data Generation Tools:

You may need tools or scripts to generate I/O requests to simulate workloads and measure algorithm performance.

### 3.1.5. Data Collection and Analysis Tools:

To measure the performance of the disk scheduling algorithms, you may require data collection and analysis tools to track metrics like seek time, throughput, and waiting time.

### 3.1.6. Documentation Tools:

Software for creating documentation and reports to record your findings and conclusions during the implementation and testing phases.

## 3.2 Hardware Requirements:

### 3.2.1. Disk Drive:

You need a physical disk drive, such as a hard disk drive (HDD) or a solid-state drive (SSD), to test and implement disk scheduling algorithms. This drive will be used to simulate and measure the performance of the scheduling algorithms.

### 3.2.2. Computer System:

You need a computer or server to host the operating system where you'll implement and test the disk scheduling algorithms. This system should have the necessary hardware interfaces to connect the disk drive.

### 3.2.3. Disk Controller:

The computer system should have a disk controller or interface (e.g., SATA, SCSI, NVMe) that supports communication with the chosen disk drive.

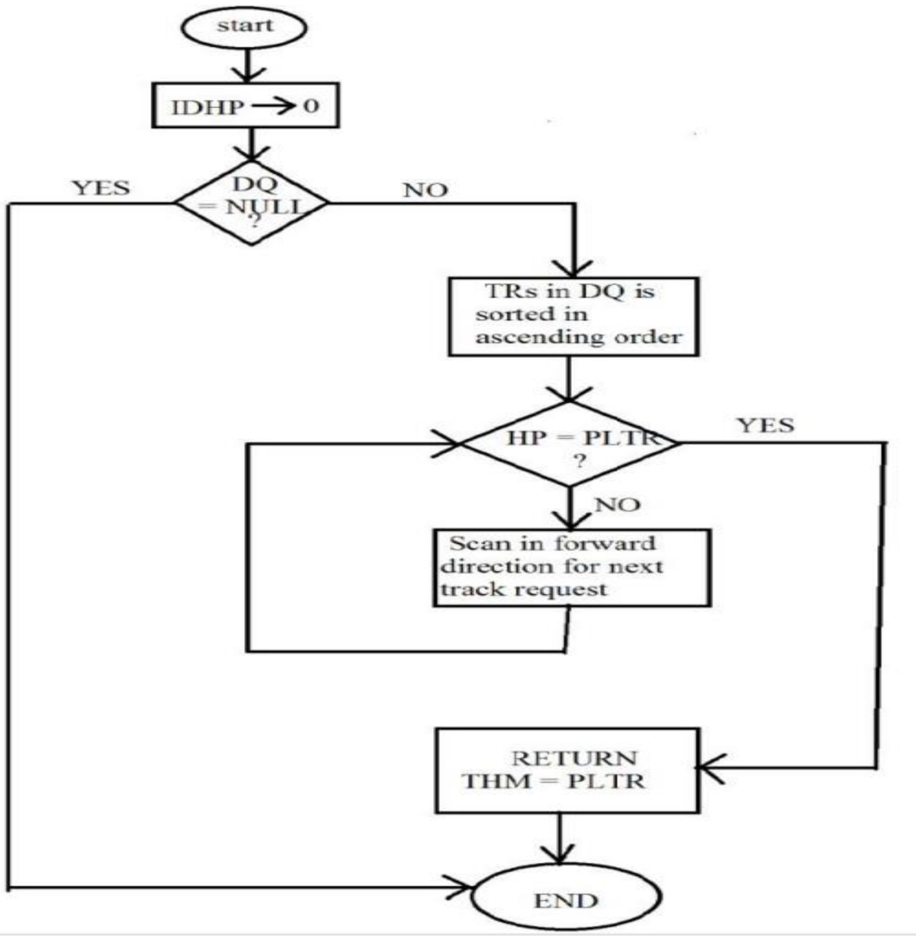
### 3.2.4. Operating System:

You'll require an operating system that supports disk scheduling and provides a platform for implementing and testing the algorithms. This could be a general-purpose operating system or a custom-built environment for simulation and experimentation.

Development Environment: You may need development tools, such as a compiler, debugger, and IDE, for programming and testing the disk scheduling algorithms

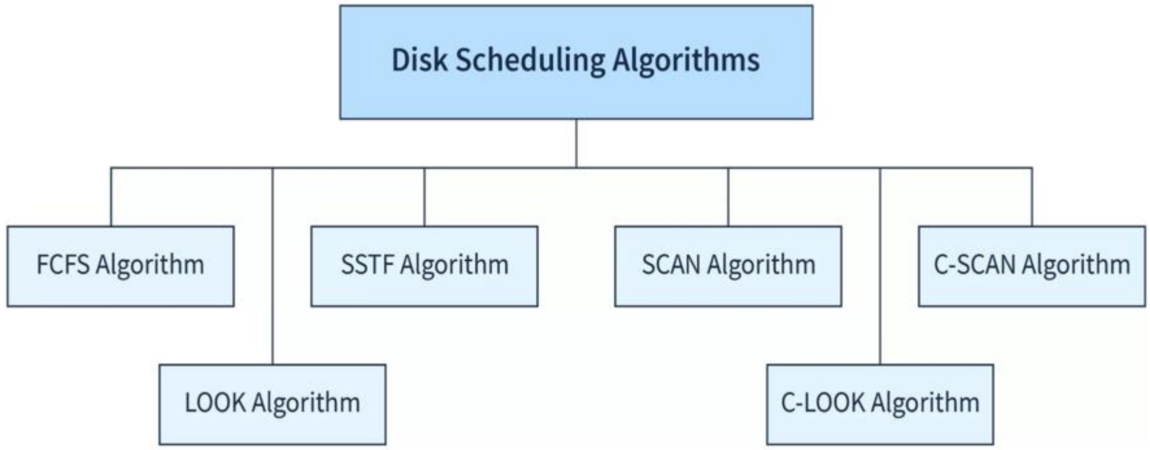
## 4. ARCHITECTURE AND DESIGN

### 4.1 FLOWCHART




**IDHP = Initial Disk Head Position**
**DQ = Disk Queue with requests**
**TR = Track Request**
**HP = Head Position**
**PLTR = Position of Last Track Request**
**THM = Total Head Movements**

## 4.2 DISK SCHEDULING ALGORITHMS




Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by the disk controller. Thus other I/O requests need to wait in the waiting queue and need to be scheduled.

Two or more requests may be far from each other so this can result in greater disk arm movement.

Hard drives are one of the slowest parts of the computer system and thus need to be accessed in an efficient manner.

## 5. IMPLEMENTATION – CODE SNIPPET

```python
from tkinter import *
import turtle
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from copy import copy
#---------------------------------------------------------------------------
def FCFS(Request, Start):
    Sum = 0                         #initialize to 0
    position = Start                #set current position = start
    Order = []                      # creates empty list of name Order
    Order.append(Start)             #adds Start to end of list Order
    for i in Request:               # i is the current element in the list(first loop i = 95)
        Sum += abs(i-position)      # sum = sum + (distance of current position from next position)
        position = i                # set position new position (i)
        Order.append(i)             # Add i to the end of the list Order
    return Order, Sum
#---------------------------------------------------------------------------
def SSTF(Request,Start):
    templist = copy(Request)
    position = Start
    highest = max(templist)
    mindiff=abs(Start-highest)
    j=highest
    templist.sort()
    Order = []
    Order.append(Start)
    Sum = 0
    while len(templist) > 0:
        for i in templist:
                diff= abs(position-i)
                if diff<mindiff:
                    mindiff=diff
                    j=i
        Sum+= abs(position-j)
        position = j
        templist.remove(j)
```

```python
            Order.append(j)
            mindiff=abs(position-highest)
            j=highest
    return Order, Sum
#----------------------------------------------------------------------------
def SCAN(Request, Start):
    n = len(Request)
    Order = []
    Request_tmp=copy(Request)
    Request_tmp.sort()
    if Start != 0 and Start < Request_tmp[n-1]:
        Request_tmp.append (0)
    p = len(Request_tmp)

    i = Start - 1
    Order.append(Start)
    while i >= 0:
        for j in range(0,p):
            if(Request_tmp[j] == i):
                Order.append(i)
        i -= 1



    k = Start + 1
    while k < 200:
        for l in range(0,n):
            if(Request[l] == k):
                Order.append(k)
        k += 1

    Sum = 0
    for p in range(0,len(Order) - 1):
        Sum += abs(Order[p] - Order[p+1])
    return Order, Sum
```

```python
#----------------------------------------------------------------------
def CSCAN(Request, Start):
    n = len(Request)
    Order = []
    Request_tmp=copy(Request)
    Request_tmp.sort()
    if Start != 0 and Start < Request_tmp[n-1]:
        Request_tmp.append (0)
    p = len(Request_tmp)

    i = Start - 1
    Order.append(Start)
    while i >= 0:
        for j in range(0,p):
            if(Request_tmp[j] == i):
                Order.append(i)
        i -= 1

    k = 199
    while k > Start:
        if(k == 199):
            Order.append(k)
        for l in range(0,n):
            if(Request[l] == k):
                Order.append(k)
        k -= 1

    Sum = 0
    SortedReq = copy(Order)
    SortedReq.sort()
    for p in range(0,len(Order) - 1):
        if (Order[p] != SortedReq[0]):
            Sum += abs(Order[p] - Order[p+1])
    return Order, Sum
#----------------------------------------------------------------------
def LOOK(Request, Start):
    n = len(Request)                        # Number of Requests
```

```python
    Order = []
    i = Start - 1
    Order.append(Start)
    while i > 0:                            # Diskhead moving outward from start
        for j in range(0,n):                    #position
            if(Request[j] == i):            # Request found
                Order.append(i)             # Request executed
        i -= 1

    k = Start + 1
    while k < 200:                          # Diskhead moving inward from
        for l in range(0,n):                    #previous position
            if(Request[l] == k):            # Request found
                Order.append(k)             # Request executed
        k += 1

    Sum = 0
    for p in range(0,len(Order) - 1):
        Sum += abs(Order[p] - Order[p+1])   # Calculates total movement
    return Order, Sum
#-------------------------------------------------------------------------------
def CLOOK(Request, Start):
    n = len(Request)                            # Number of requests
    Order = []
    i = Start - 1
    Order.append(Start)
    while i > 0:                                # Diskhead moving outward from
        for j in range(0,n):                        #start position
            if(Request[j] == i):                # Request found
                Order.append(i)                 # Request executed
        i -= 1

    k = 199
    while k > Start:                            # Diskhead moving inward from
        for l in range(0,n):                        #highest request position
            if(Request[l] == k):                # Request found
```

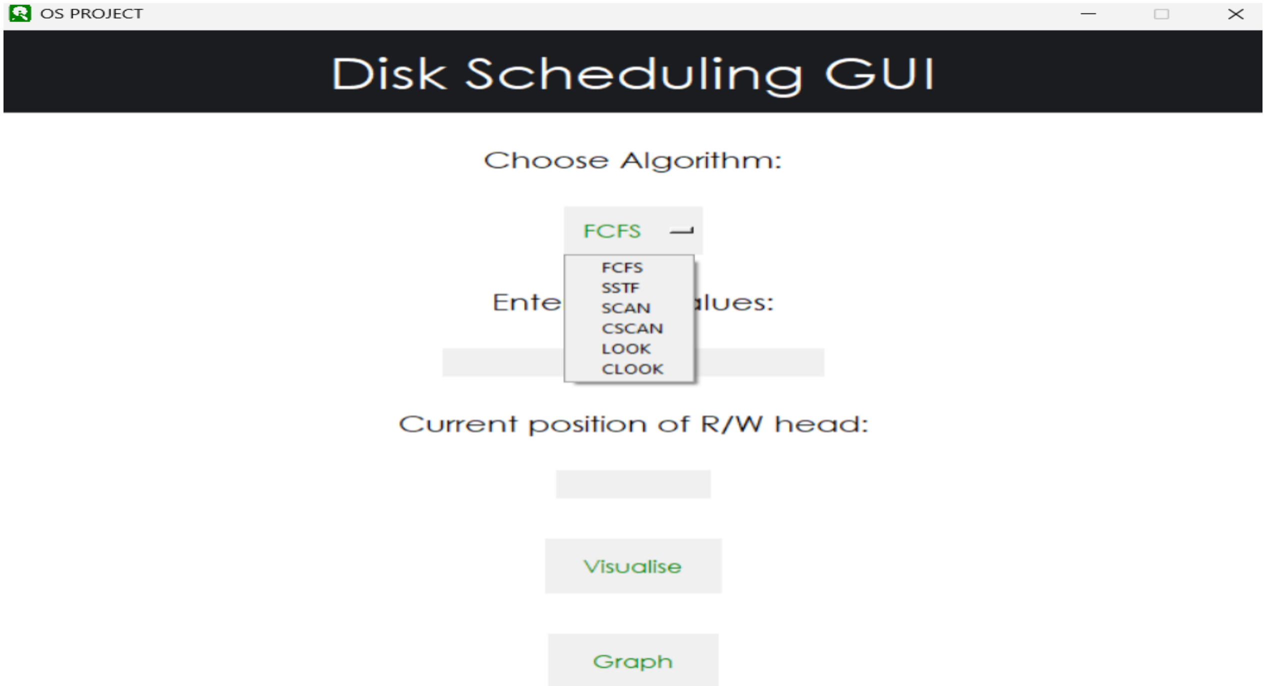
```python
                Order.append(k)                 # Request executed
        k -= 1

    Sum = 0
    SortedReq = copy(Order)                     # Creates copy of job order
    SortedReq.sort()                            # Sorts job order from lowest to
    for p in range(0,len(Order) - 1):               #highest
        if (Order[p] != SortedReq[0]):          # Excludes the circular movement
            Sum += abs(Order[p] - Order[p+1])   # Calculates total movement
    return Order, Sum

def graphy(request_arr, start):
    plot_list=[]
    algos=["FCFS","SSTF","SCAN","CSCAN","LOOK","CLOOK"]
    dummy=0
    request=list(request_arr.split(" "))
    request=[int(i) for i in request]
    _,dummy=FCFS(request,start)
    plot_list.append(dummy)
    _,dummy=SSTF(request,start)
    plot_list.append(dummy)
    _,dummy=SCAN(request,start)
    plot_list.append(dummy)
    _,dummy=CSCAN(request,start)
    plot_list.append(dummy)
    _,dummy=LOOK(request,start)
    plot_list.append(dummy)
    _,dummy=CLOOK(request,start)
    plot_list.append(dummy)
    fig = plt.figure()
    fig.suptitle('Total Head Movement Graph', fontsize=14)
    plt.bar(algos, plot_list)
    plt.show()


#------------------------------------------------------------------------------
```

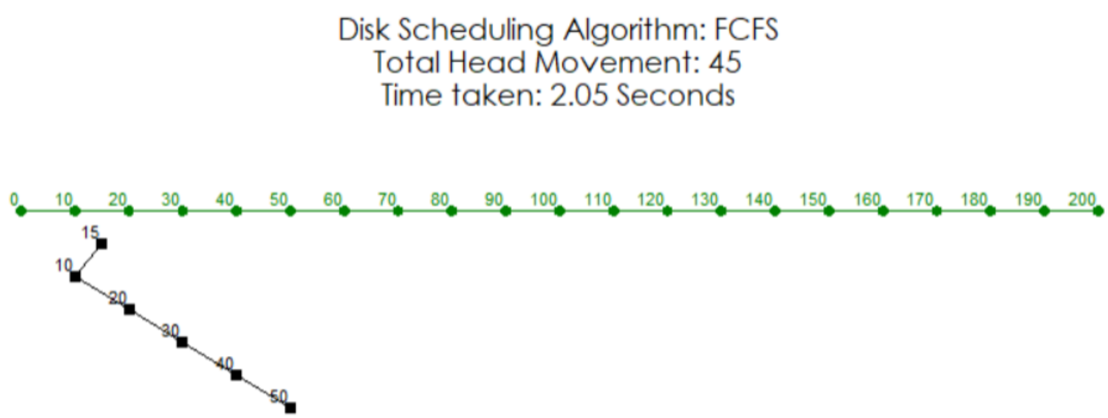
```python
def Visualise(option, request_arr, start):
    request=list(request_arr.split(" "))
    request=[int(i) for i in request]
    if option == "FCFS":                    # Select and run algorithm
        Order, Sum = FCFS(request, start)
    elif option =="SSTF":
        Order, Sum = SSTF(request, start)
    elif option =="SCAN":
        Order, Sum = SCAN(request, start)
    elif option =="CSCAN":
        Order, Sum = CSCAN(request, start)
    elif option =="LOOK":
        Order, Sum = LOOK(request, start)
    elif option =="CLOOK":
        Order, Sum = CLOOK(request, start)

    import time
    turtle.clearscreen()
    t0 = time.time()
    Disk = turtle.Screen()
    Disk.title(option)
    Disk.bgcolor("white")
    Disk.setworldcoordinates(-5, -20, 210, 10)  # Set turtle window boundaries
    head = turtle.Turtle()
    head.shape("square")
    head.color("black")
    head.turtlesize(.3, .3, 1)
    head.speed(2)
    head.pensize(0)

    head2 = turtle.Turtle()
    head2.shape("circle")
    head2.color("green")
    head2.turtlesize(.3, .3, 1)
    head2.speed(4)
    head2.pensize(0)
```

```
    n = len(Order)
    y = -1
    y2=0
    temp_order=[int(i*10) for i in range(0,21)]
    for i in range(0,len(temp_order)):
        head2.goto(temp_order[i], y2)
        head2.stamp()
        head2.write(temp_order[i], False, align="right")


    for i in range(0, n):
        if i == 0:      # No drawing while the diskhead reaches start position
            head.penup()
            head.goto(Order[i], y)
            head.pendown()
            head.stamp()
            head.write(Order[i], False, align="right")
        else:           # Diskhead draws its path to each request
            head.goto(Order[i], y-1)
            head.stamp()
            head.write(Order[i], False, align="right")
            y -= 1
    head.hideturtle()
    head.speed(0)
    head.penup()
    head.goto(100, 5)
    t1 = time.time()


    message1 = "Disk Scheduling Algorithm: " + option
    message2 = "Total Head Movement: " + str(Sum)
    start = "\033[1m"
    end = "\033[0;0m"
    head.write(message1, False, align="center",font=("Century Gothic", 14) )    # Display algorithm used
    head.goto(100,4)
    head.write(message2, False, align="center",font=("Century Gothic", 14))    # Display total movement
```

```python
    head.goto(100,3)
    head.write("Time taken: "+str(round(t1-t0, 2))+" Seconds", False, align="center",font=("Century Gothic", 14))
    head.pendown
    Disk.exitonclick()
#------------------------------------------------------------------------------
def Main():

    #List of colours
    mainbg="White"
    HeaderBG="Black"
    TextCol="Green"
    eleBG="#f0f0f0"
    #List of Messages
    algo="Choose Algorithm:"
    vals="Enter your values:"
    current="Current position of R/W head:"

    Menu = Tk()
    Menu.title("OS PROJECT")
    Menu.overrideredirect(False)
    Menu.iconbitmap("icon.ico")
    Menu.geometry("811x700+0+0")
    Menu.resizable(False, False)
    Menu.configure(bg='white')
    user_inp=Text(Menu,font=("Century Gothic", 16),width=20,height=1,bg=eleBG,fg=TextCol,bd=0)
    user_inp.grid(row=7, column=0)
    user_inp.config(highlightbackground = "Red",highlightcolor="Red")
    title=Label(Menu, text="Disk Scheduling GUI",anchor=CENTER, bd=12,padx=200, bg="#1A1C20", fg=mainbg, font=("Century Gothic",30), pady=2).grid(row=0)



    # List of options in dropdown menu
    optionlist = ('FCFS', 'SSTF', 'SCAN', 'CSCAN', 'LOOK', 'CLOOK')
    Option = StringVar()
    Start = IntVar()
    Option.set("FCFS")
    L1 = Label(Menu, text = algo,font=("Century Gothic", 16),bg=mainbg,fg=HeaderBG,pady=30)            # Label 1
    L1.grid(row=2,column=0)
    OM = OptionMenu(Menu, Option, *optionlist)              # Dropdown menu
    OM.grid(row=3, column=0)
    OM.configure(bd = 0,bg=eleBG,fg=TextCol,highlightthickness = 0,padx=12,pady=12,font=("Century Gothic", 12))
    L2 = Label(Menu, text = current,font=("Century Gothic", 16),bg=mainbg,fg=HeaderBG,pady=30)          # Label 2
    L2.grid(row=12, column=0)
    E1 = Entry(Menu, textvariable = Start, bd = 0,fg=TextCol, width = 8,bg=eleBG,justify=CENTER,font=("Century Gothic", 16))   # Textbox
    E1.grid(row=13, column=0,)
    L1 = Label(Menu, text = "",font=("Century Gothic", 16),bg=mainbg,fg=mainbg,pady=5)            # Label 1
    L1.grid(row=14,column=0)
    B1 = Button(Menu,borderwidth=0,padx=20,pady=10,bg=eleBG,fg=TextCol, text = "Visualise",\
    command = lambda:Visualise(Option.get(), user_inp.get(1.0,END), Start.get()),font=("Century Gothic", 12))  # Button
    B1.grid(row=15, column=0)
    L1 = Label(Menu, text = "",font=("Century Gothic", 16),bg=mainbg,fg=mainbg,pady=5)            # Label 1
    L1.grid(row=16,column=0)
    B2 = Button(Menu,borderwidth=0,padx=20,pady=10,bg=eleBG,fg=TextCol, text = " Graph ",\
    command = lambda:graphy(user_inp.get(1.0,END), Start.get()),font=("Century Gothic", 12))  # Button
    B2.grid(row=17, column=0)
    L1 = Label(Menu, text = vals,font=("Century Gothic", 16),bg="white",fg=HeaderBG,pady=30)            # Label 1
    L1.grid(row=5,column=0)
    Menu.mainloop()
Main()
```

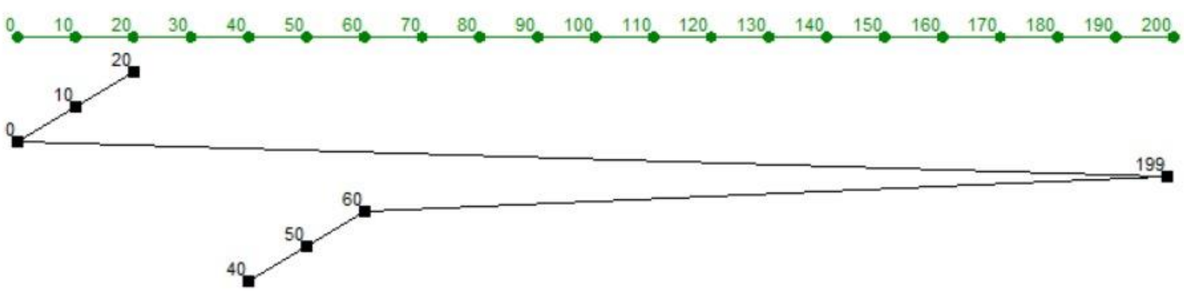
## 6. RESULTS – Screen Shots Of Output

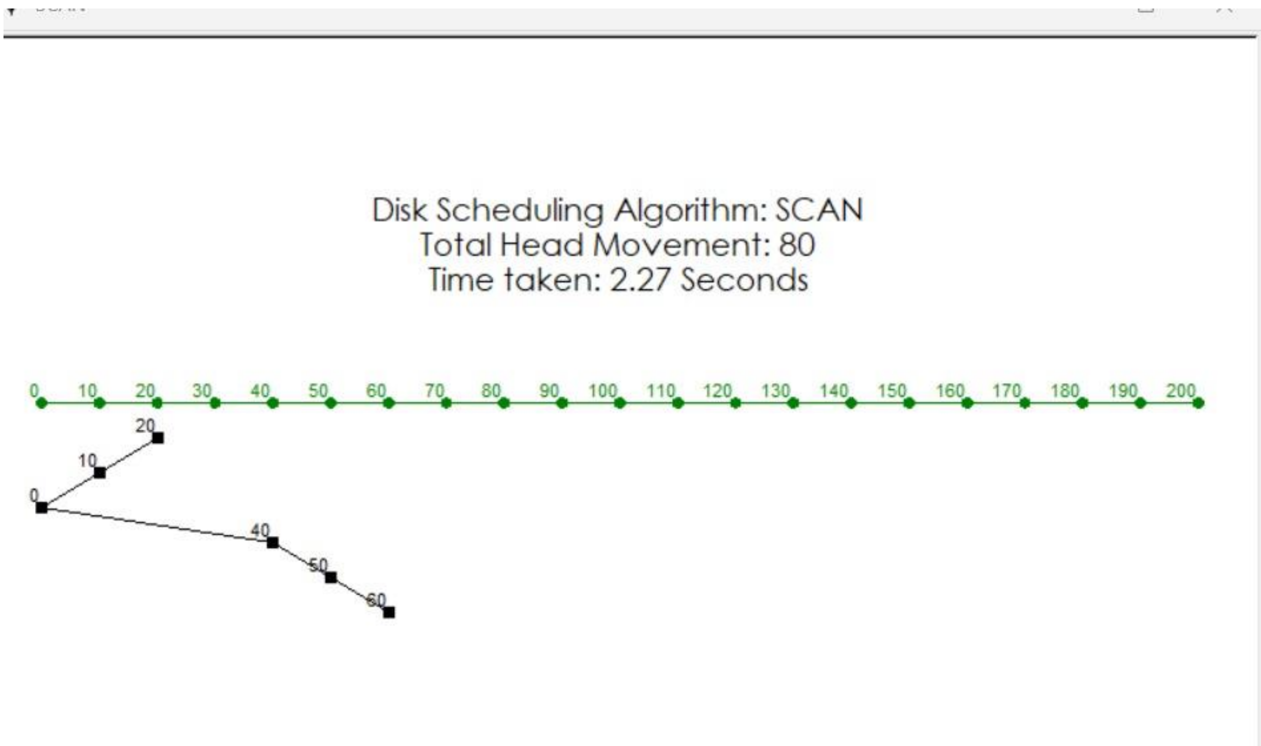


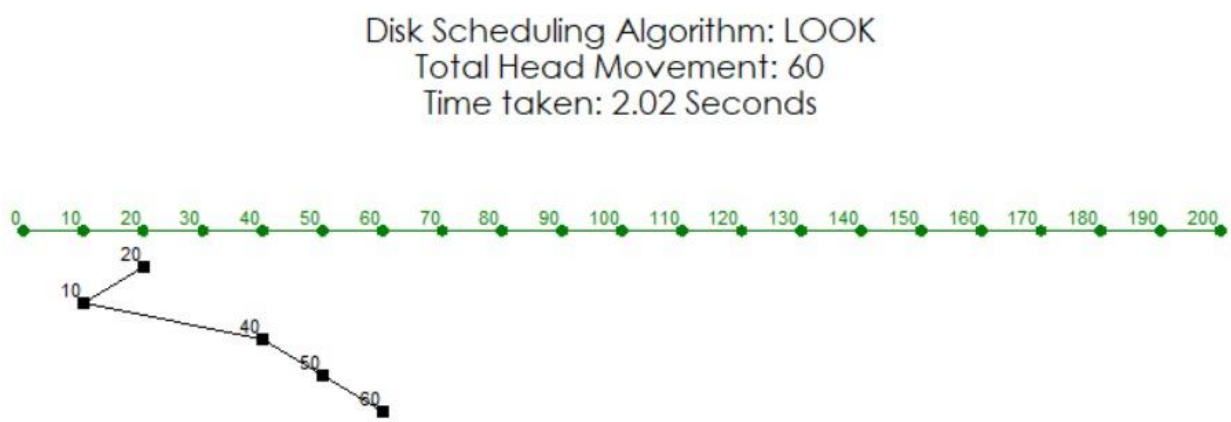



Disk Scheduling Algorithm: CLOOK
Total Head Movement: 30
Time taken: 2.15 Seconds

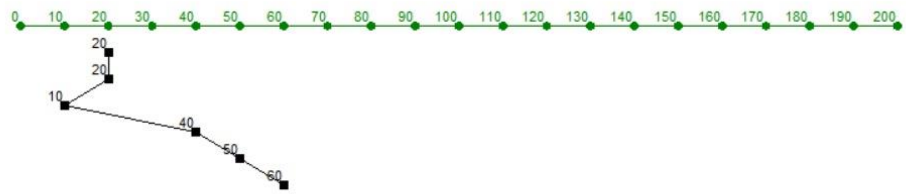

Disk Scheduling Algorithm: CSCAN
Total Head Movement: 179
Time taken: 5.18 Seconds



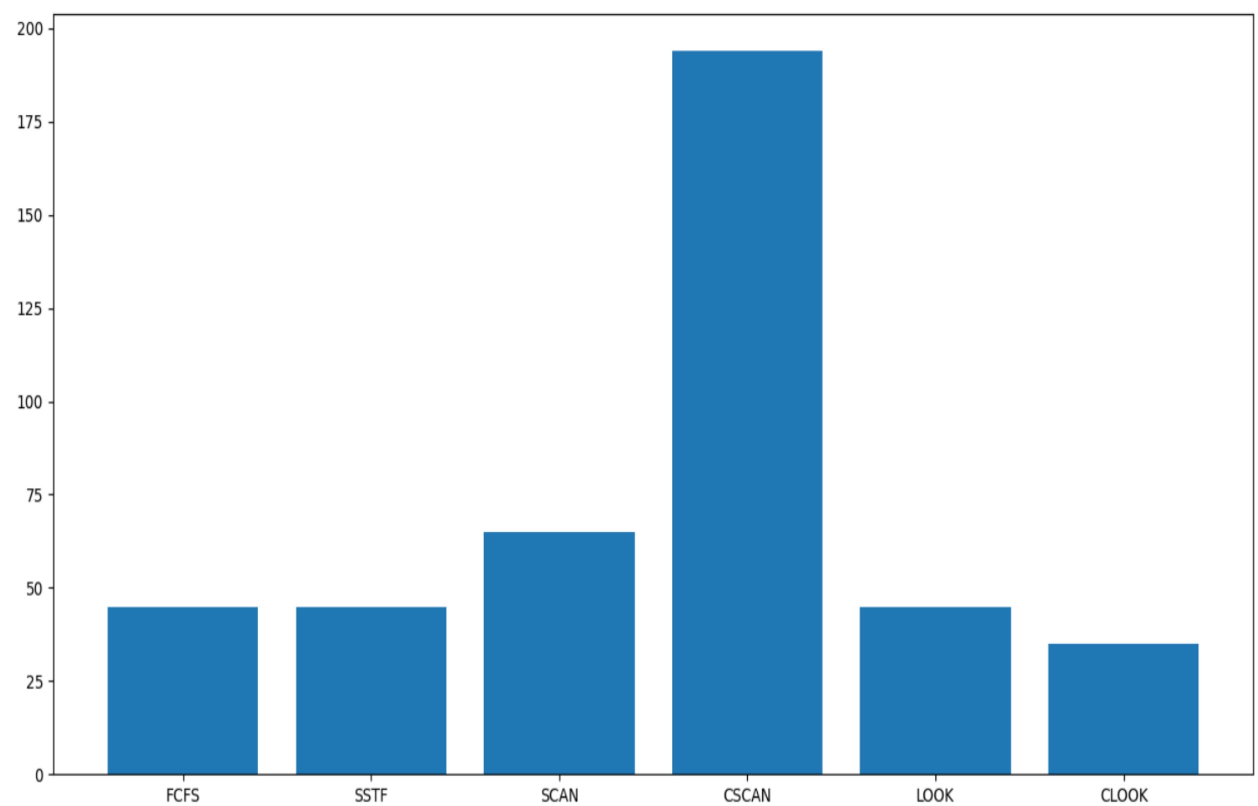

Disk Scheduling Algorithm: SCAN
Total Head Movement: 80
Time taken: 2.27 Seconds
0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200
20
10
0
40
50
60


Disk Scheduling Algorithm: LOOK
Total Head Movement: 60
Time taken: 2.02 Seconds
0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200
20
10
40
50
60

Disk Scheduling Algorithm: SSTF
Total Head Movement: 60
Time taken: 2.29 Seconds




Total Head Movement Graph



## 7. CONCLUSION

Disk scheduling is the unsung hero behind efficient data access in modern computer systems. By managing the order in which data requests are executed, it plays a crucial role in reducing waiting times, optimizing disk arm movement, and ultimately improving system performance. Disk scheduling algorithms, such as FCFS, SSTF, and others, are the linchpin in ensuring smooth and responsive data access for users and organizations, making them a fundamental component of any operating system.

The impact of disk scheduling extends far beyond the realm of computer science. It affects user experience, system productivity, and the competitive advantage of organizations. As technology evolves, and the demand for faster and more reliable data access grows, understanding and implementing efficient disk scheduling become increasingly significant. Whether you're a system administrator, developer, or simply a user of modern technology, the world of disk scheduling underpins a faster, more responsive digital experience.

## 8. REFERENCES


1. Tanenbaum, A. S., & Bos, H. (2014). Modern Operating Systems (4th ed.). Pearson.

2. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th ed.). Wiley.

3. El-Rewini, H., & Lewis, T. G. (2002). Introduction to Parallel Computing. CRC Press.

4. Garg, R. (2006). System Modeling and Simulation: An Introduction. CRC Press.

5. Chen, P. M. (1994). The Evolution of I/O Systems in the VAX Computer System. ACM Transactions on Computer Systems (TOCS), 12(1), 66-92.

6. Denning, P. J. (1968). The Working Set Model for Program Behavior. Communications of the ACM, 11(5), 323-333.