A WebAssembly-Based Interactive Simulator for Disk Scheduling Algorithms

Mann Vaswani, Shiven Ahuja
Department of Computer Science and Engineering
University Institute of Engineering and Technology
Chandigarh, India
Email: mannvaswani4@gmail.com, shivenahuja@example.com

***Abstract*—Disk scheduling algorithms are fundamental to operating system design, directly influencing I/O throughput and system responsiveness. Despite their importance, traditional educational approaches rely on static diagrams and manual trace computations, which fail to convey the dynamic behavior of these algorithms. This paper presents a high-performance, interactive, web-based simulator for eight disk scheduling algorithms: First-Come First-Served (FCFS), Shortest Seek Time First (SSTF), SCAN, LOOK, Circular SCAN (C-SCAN), Circular LOOK (C-LOOK), Freeze-SCAN (FSCAN), and N-Step SCAN. The core scheduling algorithms are implemented in C and compiled to WebAssembly (WASM) using the Emscripten toolchain, enabling near-native execution speed within the browser. The frontend, built with Vanilla JavaScript, Vite, and HTML5 Canvas, provides real-time animated visualizations including a linear timeline graph and a circular radar view representing the disk platter. Users can configure parameters such as the request queue, initial head position, scan direction, and batch size (for N-Step SCAN), and interactively scrub through the simulation timeline. Experimental results demonstrate the comparative performance of all eight algorithms across various workloads, and the simulator serves as an effective educational tool for understanding disk scheduling behavior.**

***Index Terms*—Disk Scheduling, WebAssembly, Operating Systems, FCFS, SSTF, SCAN, LOOK, Interactive Simulation**

I. Introduction

Modern operating systems rely on disk scheduling algorithms to determine the order in which disk I/O requests are serviced by the read/write head of a hard disk drive (HDD). The choice of scheduling algorithm directly impacts total seek time, throughput, and fairness among competing processes [1]. As solid-state drives (SSDs) gain prevalence, understanding the mechanics of traditional disk scheduling remains essential for embedded systems, legacy infrastructure, and hybrid storage architectures [2].

Educational resources on disk scheduling predominantly use static diagrams and textbook examples that require students to manually trace algorithm execution on paper [3]. While several simulators exist, most are desktop-based, lack interactivity, or fail to visualize the step-by-step head movement in real time.

This paper presents a web-based interactive simulator that addresses these limitations. The key contributions of this work are:

- Implementation of eight disk scheduling algorithms (FCFS, SSTF, SCAN, LOOK, C-SCAN, C-LOOK, FSCAN, and N-Step SCAN) in the C programming language, compiled to WebAssembly for near-native browser performance.
- A real-time animated visualization system using HTML5 Canvas, featuring both a linear timeline graph and a circular radar view.
- An interactive control interface supporting configurable parameters including head direction (LEFT/RIGHT), batch step size for N-Step SCAN, adjustable simulation speed, and a scrubbable timeline.
- An integrated educational mode providing algorithm descriptions, advantages, and disadvantages for each scheduling policy.

The remainder of this paper is organized as follows: Section II reviews related work. Section III details the system architecture and methodology. Section IV presents experimental results. Section V concludes with a summary and future directions.

II. Related Work

Disk scheduling has been extensively studied since the early days of computing. The foundational algorithms—FCFS, SSTF, SCAN, and C-SCAN—were formalized in the pioneering work on operating system design [1]. Subsequent variants such as LOOK and C-LOOK eliminated unnecessary head travel to disk boundaries [2], while FSCAN and N-Step SCAN introduced queue-freezing and batching mechanisms to prevent starvation under heavy load [3].

Several simulation tools have been developed for educational purposes. Chakraborty et al. [4] developed a Java-based disk scheduling simulator supporting FCFS, SSTF, and SCAN, but lacked support for advanced algorithms and real-time animation. Web-based tools such as those described in [5] offer interactive operating system visualizations but are typically limited in scope and do not leverage modern browser capabilities.

The emergence of WebAssembly (WASM) has enabled high-performance computation within the browser environment [6]. WASM provides a portable, sandboxed execution environment with near-native speed, making it suitable for computationally intensive educational simulations. The Emscripten toolchain [7] facilitates compilation of C/C++ code to WASM, bridging the gap between high-performance system-level code and accessible web-based frontends.

Our work combines the computational rigor of C-based algorithm implementations with the accessibility and interactivity of modern web technologies, supporting a broader set of eight algorithms with full parameter configurability.

III. Methodology

A. System Architecture

The simulator follows a three-tier architecture, illustrated in Fig. 1:

1) **Algorithm Layer (C → WASM):** Core scheduling algorithms implemented in C, compiled to a single WebAssembly module via Emscripten.
2) **Bridge Layer (WASM ↔ JS):** A WASM bridge module and JavaScript loader that marshals data between WASM memory and JavaScript objects.
3) **Presentation Layer (JS + Canvas):** A simulation engine, renderer, and event-driven UI built with Vanilla JavaScript and HTML5 Canvas.

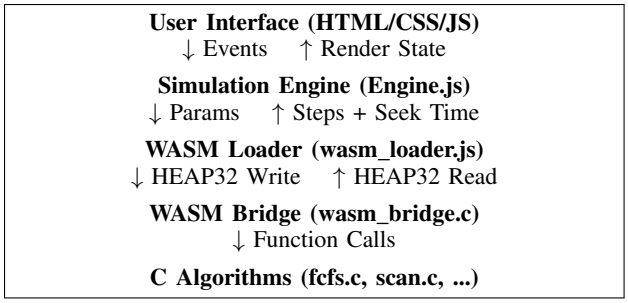

Fig. 1. Three-tier system architecture of the disk scheduling simulator.

B. Algorithm Implementations

All eight algorithms are implemented in C with a uniform interface defined in a shared header file (`scheduler.h`). Each function accepts the initial head position, an array of request track numbers, the number of requests, and algorithm-specific parameters (direction and/or disk size). Each function returns the total seek time as an integer.

1) First-Come First-Served (FCFS): FCFS processes requests in the order they arrive. Given a head position h_0 and a request sequence $R = \{r_1, r_2, \ldots, r_n\}$, the total seek time is:

$$T_{\text{FCFS}} = \sum_{i=1}^{n} |h_{i-1} - r_i| \tag{1}$$

where $h_i = r_i$ after servicing the i-th request.

2) Shortest Seek Time First (SSTF): SSTF selects the pending request closest to the current head position at each step:

$$r_{\text{next}} = \arg \min_{r_j \in R_{\text{pending}}} |h_{\text{current}} - r_j| \tag{2}$$

This greedy approach minimizes local seek time but may cause starvation for distant requests.

3) SCAN (Elevator Algorithm): SCAN moves the head in one direction, servicing all requests along the way, until it reaches the disk boundary, then reverses direction. Given a direction $d \in \{\text{left}, \text{right}\}$ and disk size D:

$$T_{\text{SCAN}} = T_{\text{sweep}_1} + |h_{\text{boundary}} - (D-1)| + T_{\text{sweep}_2} \tag{3}$$

where h_{boundary} is the disk edge (0 or $D-1$ depending on direction).

4) LOOK: LOOK is a variant of SCAN that reverses direction at the last request in the current sweep direction, rather than traveling to the disk boundary:

$$T_{\text{LOOK}} = T_{\text{sweep}_1} + T_{\text{sweep}_2} \tag{4}$$

eliminating the unnecessary seek to the disk edge.

5) Circular SCAN (C-SCAN): C-SCAN services requests in one direction only. Upon reaching the disk boundary, the head jumps to the opposite boundary without servicing, then resumes in the same direction:

$$T_{\text{C-SCAN}} = T_{\text{sweep}} + |h_{\text{end}} - h_{\text{start}}| + T_{\text{return\_sweep}} \tag{5}$$

This provides more uniform wait times compared to SCAN.

6) Circular LOOK (C-LOOK): C-LOOK combines the advantages of LOOK and C-SCAN: it services in one direction, then jumps from the last request to the first request without going to disk boundaries.

7) Freeze-SCAN (FSCAN): FSCAN uses two queues: the active queue is "frozen" and serviced using SCAN, while new arrivals enter a second queue. Once the frozen queue is fully serviced, the queues are swapped [3]. In our implementation, since requests are provided as a static batch, FSCAN behaves identically to SCAN over the complete request set.

8) N-Step SCAN: N-Step SCAN partitions the request queue into sub-queues of size N. Each sub-queue is independently processed using SCAN, with the head position carrying over between batches:

$$T_{\text{N-Step}} = \sum_{k=1}^{\lceil n/N \rceil} T_{\text{SCAN}}(B_k, h_{k-1}, d_k) \tag{6}$$

where B_k is the k-th batch, h_{k-1} is the head position after the previous batch, and d_k alternates between sweeps. This algorithm balances response time and prevents starvation [3].

C. WebAssembly Compilation Pipeline

The C algorithm files are compiled to a single WebAssembly module using Emscripten (`emcc`) with the following configuration:

- **Single-file embedding:** The `SINGLE_FILE=1` flag embeds the WASM binary as Base64 within the JavaScript glue code, enabling seamless ES module imports by Vite.
- **Exported functions:** Algorithm runners (`_run_fcfs`, `_run_scan`, etc.) and memory accessors (`_get_request_buffer`, `_get_steps_buffer`) are explicitly exported.

- **Memory model:** Shared static buffers (`MAX_REQUESTS = 256`, `MAX_STEPS = 512`) are used for zero-copy data transfer between JavaScript and WASM via `HEAP32`.
- **Optimization:** The `-O2` optimization level is applied for production performance.

D. WASM Bridge and JavaScript Loader

The bridge layer consists of two components:

WASM Bridge (`wasm_bridge.c`): Wraps each C algorithm function. For each algorithm, the bridge: (a) calls the algorithm to compute total seek time, (b) reconstructs the ordered sequence of visited tracks into a shared output buffer, and (c) stores the step count and total seek time in global variables accessible from JavaScript.

JavaScript Loader (`wasm_loader.js`): Provides high-level functions (e.g., `calculateSCAN()`) that: (a) write the request array into WASM memory via `HEAP32`, (b) invoke the bridge function, (c) read results back from WASM memory, and (d) return a structured object {`steps, totalSeek`} to the simulation engine.

E. Simulation Engine

The simulation engine (`Engine.js`) manages algorithm selection, playback state, and animation timing. Key features include:

- **Playback control:** Play, pause, stop, and variable speed ($1\times$ to $5\times$).
- **Seek/scrub:** Users can scrub to any point in the simulation timeline using a slider mapped to a normalized progress value $[0, 1]$.
- **Interpolation:** Head position is smoothly interpolated between discrete steps using an ease-in-out quadratic function:

$$f(t) = \begin{cases} 2t^2 & \text{if } t < 0.5 \\ -1 + (4 - 2t)t & \text{otherwise} \end{cases} \tag{7}$$

F. Visualization System

The renderer (`Renderer.js`) provides two synchronized visualizations on HTML5 Canvas:

Linear Timeline Graph: The X-axis represents the step index (time), and the Y-axis represents the track number (0 to $D - 1$). The head path is drawn as a glowing cyan line with magenta markers at each serviced request. A white animated dot indicates the current head position with a floating tooltip displaying the track number.

Circular Radar View: Concentric circles represent track positions. The current head position is displayed as a glowing magenta ring at the corresponding radius. An animated spinning effect simulates disk platter rotation.

Both canvases support dynamic DPI-aware resizing for high-resolution displays.

IV. Results and Discussion

A. Experimental Setup

The simulator was tested with a disk of 200 tracks (0–199). A default request queue of $R = \{98, 183, 37, 122, 14, 124, 65, 67\}$ with an initial head position of $h_0 = 53$ was used. All directional algorithms were tested with the initial direction set to RIGHT.

B. Seek Time Comparison

Table I presents the total seek time for each algorithm under the default configuration.

TABLE I
Total Seek Time Comparison Across Algorithms

Algorithm	Total Seek	Avg. Seek/Step
FCFS	640	80.00
SSTF	236	29.50
SCAN	332	33.20
LOOK	299	37.38
C-SCAN	389	38.90
C-LOOK	322	40.25
FSCAN	332	33.20
N-Step (N=5)	343	34.30

C. Analysis

FCFS exhibits the highest total seek time (640) due to the random order of request servicing, confirming its well-known inefficiency for non-sequential workloads.

SSTF achieves the lowest total seek time (236) by greedily selecting the nearest request. However, as extensively documented [1], this approach risks starving distant requests under sustained load.

SCAN and FSCAN produce identical results (332) under static batch conditions, since FSCAN freezes the entire queue and applies SCAN traversal. The difference between these algorithms manifests only in dynamic arrival scenarios.

LOOK improves upon SCAN (299 vs. 332) by eliminating unnecessary seeks to the disk boundary, confirming its theoretical advantage [2].

C-SCAN (389) has higher seek time than SCAN due to the return sweep to the start of the disk without servicing requests. However, it provides more uniform wait times, which is advantageous in multi-process environments.

N-Step SCAN with $N = 5$ yields a total seek time of 343, falling between SCAN and C-SCAN. The batch partitioning introduces overhead but prevents starvation by limiting the influence of any single batch on subsequent scheduling decisions. The adjustable step size N allows users to explore the trade-off between responsiveness and efficiency.

D. Effect of Direction Parameter

Table II shows the impact of initial head direction on directional algorithms.

The results confirm that the initial direction can significantly affect performance depending on the distribution of requests relative to the head position. For $h_0 = 53$, the LEFT direction

TABLE II
EFFECT OF INITIAL DIRECTION ON TOTAL SEEK TIME

Algorithm	LEFT	RIGHT
SCAN	303	332
LOOK	270	299
C-SCAN	376	389
C-LOOK	319	322
FSCAN	303	332
N-Step (N=5)	330	343

generally results in lower seek times because fewer requests exist below track 53 ($\{37, 14\}$), allowing the head to quickly reverse and service the majority of requests concentrated in the higher track range.

E. Effect of Step Size on N-Step SCAN

Table III demonstrates how varying the batch size N affects N-Step SCAN performance.

TABLE III
N-STEP SCAN: EFFECT OF STEP SIZE (N) ON TOTAL SEEK TIME

Step Size (N)	Total Seek
2	420
3	385
5	343
8	332

As N increases, N-Step SCAN converges to standard SCAN behavior ($N = n$ yields identical results to SCAN). Smaller values of N introduce more boundary-crossing overhead but improve fairness across batches.

V. CONCLUSION

This paper presented a WebAssembly-based interactive simulator for eight disk scheduling algorithms: FCFS, SSTF, SCAN, LOOK, C-SCAN, C-LOOK, FSCAN, and N-Step SCAN. The system architecture combines C-based algorithm implementations compiled to WASM for performance with a modern web frontend for accessibility and interactivity. The simulator provides real-time animated visualizations, configurable parameters including head direction and batch size, and an integrated educational mode.

Experimental results validated the expected performance characteristics of each algorithm and demonstrated the impact of parameters such as initial direction and step size on total seek time. The simulator serves as both an educational tool for students learning operating system concepts and a research platform for exploring scheduling algorithm behavior under various configurations.

Future work includes: (a) dynamic request arrival simulation to differentiate FSCAN from SCAN, (b) support for additional scheduling policies such as SATF (Shortest Access Time First), (c) comparative performance benchmarking across different workload distributions, (d) a side-by-side comparison mode for visualizing multiple algorithms simultaneously, and (e) extending the simulator to model SSD I/O scheduling strategies.

REFERENCES

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ, USA: Wiley, 2018.

[2] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Upper Saddle River, NJ, USA: Pearson, 2015.

[3] W. Stallings, *Operating Systems: Internals and Design Principles*, 9th ed. London, UK: Pearson, 2018.

[4] S. Chakraborty, S. Gupta, and R. Kumar, "A comparative study of disk scheduling algorithms," *Int. J. of Advanced Research in Computer Science and Software Eng.*, vol. 4, no. 3, pp. 236–240, 2014.

[5] S. Robbins, "An interactive web-based simulation for operating systems," in *Proc. ACM SIGCSE Technical Symp. on Computer Science Education*, 2004, pp. 182–186.

[6] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, "Bringing the web up to speed with WebAssembly," in *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2017, pp. 185–200.

[7] A. Zakai, "Emscripten: An LLVM-to-JavaScript compiler," in *Proc. ACM Int. Conf. Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011, pp. 301–312.