Phase 2 Progress Report: Architectural Implementation and Algorithmic Analysis of the Web-Based CPU Scheduling Simulation

1. Executive Summary

This document constitutes the formal Phase 2 Progress Report for the "Web-Based CPU Scheduling Algorithms Simulation" project, undertaken by the undergraduate engineering team at WISE University for the Operating Systems course. The project aims to construct a high-fidelity, cross-platform simulation environment capable of visualizing the complex temporal dynamics of process scheduling within a multitasking operating system.

Phase 2, designated as the "Implementation Phase," marks the critical transition from theoretical system design to concrete software engineering. This report provides an exhaustive technical analysis of the work completed, focusing specifically on the successful codification of the First-Come, First-Served (FCFS) scheduling algorithm, the establishment of the system's core web-based architecture, and the deployment of a responsive, mobile-first Graphical User Interface (GUI).

The analysis herein demonstrates that the team has successfully established the Process Control Block (PCB) data structures required to emulate kernel-level process management. Furthermore, the report details the mathematical logic used to derive critical performance metrics—Turnaround Time (TAT), Waiting Time (WT), and Response Time (RT)—and validates these implementations against standard operating system theory as defined by Silberschatz et al..[1] Technical challenges related to asynchronous state management in JavaScript and responsive visualization scaling are thoroughly documented, along with the engineering solutions devised to resolve them. The report concludes that the foundational systems are now robust enough to support the complex, preemptive algorithms scheduled for Phase 3.

2. Introduction and Project Scope

2.1 Theoretical Context: The Scheduler's Role

In modern multiprogrammed operating systems, the CPU scheduler serves as the arbiter of computational resources. As described in seminal texts like *Operating System Concepts* (10th Edition), the objective of multiprogramming is to maximize CPU utilization by keeping some process running at all times.[1] The scheduler must select processes from the ready queue and

allocate the CPU core to them based on specific algorithms that balance competing system objectives, such as maximizing throughput, minimizing latency, or ensuring fairness.

For students of computer science, mastering these concepts requires moving beyond static Gantt charts. It necessitates an understanding of the dynamic interplay between process arrival times, burst durations, and the scheduler's decision-making logic. The "Web-Based CPU Scheduling Algorithms Simulation" aims to fill this pedagogical gap by providing an interactive sandbox where these variables can be manipulated in real-time.

2.2 Phase 2 Deliverables

Phase 2 focuses specifically on the following technical milestones:

1. **Core Implementation:** Developing the underlying software architecture to handle input/output (I/O) operations and process state management.
2. **Algorithm Deployment:** Fully implementing the FCFS scheduling algorithm logic within the `FCFS.js` module.[1]
3. **UI/UX Realization:** Transitioning design prototypes into functional, responsive code using HTML5, CSS3, and Vanilla JavaScript.[1]
4. **Metric Calculation:** Automating the derivation of start times, end times, and efficiency metrics for simulated processes.[1]

2.3 Team Configuration

The project is executed by a collaborative team responsible for distinct layers of the application stack:

- **Joud Kayyali (3230601030):** Lead Architect focusing on the algorithmic logic and mathematical verification.
- **Ibraheem Abdullah (3230605046):** Systems Integrator managing the DOM manipulation and event handling.
- **alnader ahmad (3230601029):** UI/UX Engineer responsible for the mobile-first CSS architecture and visualization layout.[1]

3. Theoretical Framework: Mapping OS Concepts to Data Structures

To build a valid simulation, it is imperative to accurately map kernel-level concepts to application-level data structures. This section analyzes how the project emulates the theoretical constructs of an operating system.

3.1 The Process Control Block (PCB)

In a real operating system kernel, each process is represented by a Process Control Block (PCB). As defined by Silberschatz, the PCB serves as the repository for any information that

may vary from process to process, including the Process State, Program Counter, CPU Registers, and Scheduling Information.[1]

In the simulation environment, we define a synthetic PCB object in JavaScript that encapsulates these attributes. This abstraction allows the simulation to mimic the behavior of a kernel scheduler without requiring low-level system access.

Table 1: Mapping Kernel PCB Attributes to Simulation Objects

Kernel PCB Attribute	Simulation Property	Description & Functionality
Process ID (PID)	`pid` (String)	A unique identifier used to track the process through the ready queue.[1]
Accounting Info	`arrivalTime` (Number)	The simulation timestamp (t) at which the process enters the Ready Queue.[1]
CPU Scheduling Info	`burstTime` (Number)	The predefined duration of CPU time required for the process to complete its task.[1]
Priority	`priority` (Number)	An integer value representing process urgency (utilized in Phase 3 for Priority Scheduling).
State/Timing	`startTime` / `endTime`	Computed properties representing the exact simulation timestamps when the process transitions to "Running" and "Terminated" states.[1]

This object-oriented approach ensures that the data flow within the application mirrors the context-switching mechanism of an OS: when the scheduler selects a process, it loads these attributes to perform calculations, simulating the loading of a PCB into the CPU.

3.2 Scheduling Metrics and Mathematical Definitions

The simulation evaluates algorithm performance using standard quantitative metrics. The Phase 2 implementation rigorously adheres to the following definitions to ensure academic accuracy:

- **Turnaround Time (TAT)**: The interval from the time of submission of a process to the time of completion.1 In the simulation code, this is calculated as:

$$TAT = EndTime - ArrivalTime$$

This metric measures the throughput of the system from the user's perspective.1

- **Waiting Time (WT)**: The sum of the periods spent waiting in the ready queue.1 For the non-preemptive FCFS algorithm implemented in Phase 2, a process waits only once (before it starts). Thus:

$$WT = StartTime - ArrivalTime$$

Alternatively, it is derived as $WT = TAT - BurstTime$. Minimizing WT is a primary optimization goal for schedulers.

- **Response Time (RT)**: The time from the submission of a request until the first response is produced.1 In non-preemptive environments like FCFS, the Response Time is identical to the Waiting Time:

$$WT = StartTime - ArrivalTime$$

The system architecture distinguishes these two metrics to accommodate future preemptive algorithms (like Round Robin) where RT and WT will diverge.

4. System Architecture

The "Web-Based CPU Scheduling Algorithms Simulation" is built upon a modular, client-side architecture. This design choice ensures high performance and zero latency in calculation.

4.1 Technology Stack Selection

The team deliberately chose a "Vanilla" web stack (HTML5, CSS3, JavaScript ES6+) without external frameworks. This decision aligns with the "Mobile-First" philosophy and the requirement for zero external dependencies.[1]

- **HTML5:** Provides the semantic skeleton (`<section>`, `<article>`, `<table>`) ensuring accessibility.
- **CSS3:** Powers the visual layer using Grid and Flexbox for fluid layouts.
- **JavaScript (ES6+):** Functions as the simulation engine, handling the "Kernel" logic—process management and scheduling calculations.

4.2 The "Mobile-First" Design Philosophy

A central pillar of the Phase 2 implementation was the adoption of a Mobile-First design philosophy. The UI was conceptualized primarily for mobile web browsers before being scaled up for desktop viewports.[1]

Architectural Implications:

1. **Vertical Stacking:** On narrow viewports, the Input Configuration Panel and the Results Table stack vertically.
2. **Touch Targets:** Input fields for `Arrival Time` and `Burst Time` are sized to be thumb-friendly.
3. **Gantt Chart Scrolling:** A responsive container with `overflow-x: scroll` allows the timeline to extend horizontally beyond the viewport while maintaining the correct aspect ratio of time blocks.[1]

4.3 Modular File Structure

The project code enforces separation of concerns, simulating the separation between Kernel Space (Logic) and User Space (UI).

- **`index.html`:** The Presentation Layer defining the DOM structure.
- **`style.css`:** The Styling Layer enforcing the color palette and layout logic.[1]
- **`FCFS.js`:** The Algorithm Layer. This module contains the pure logic for the FCFS scheduler. It accepts raw data and returns solved data, independent of the DOM.[1]
- **`script.js`:** The Controller Layer. This acts as the bridge (or System Call Interface) between the HTML UI and the Algorithm Layer, handling event listeners and input validation.[1]

5. Implementation: The First-Come, First-Served (FCFS) Algorithm

The First-Come, First-Served (FCFS) algorithm is the simplest CPU scheduling algorithm. In this scheme, the process that requests the CPU first is allocated the CPU first. It is a non-preemptive algorithm.[1]

5.1 Algorithmic Logic and Control Flow

The implementation logic resides in `FCFS.js`. The function `FCFS(processes)` accepts an array of PCB objects and returns an array of "solved" objects.

Step-by-Step Execution Logic:

1. **Input Sanitation and Sorting:**
   The system sorts the ready queue strictly by Arrival Time ($O(N \log N)$), simulating the FIFO nature of the FCFS queue.1
   JavaScript
   ```
   const sorted = [...processes].sort((a, b) => a.arrivalTime - b.arrivalTime);
   ```

2. **Clock Initialization:**
   A variable `currentTime` acts as the global system clock, initialized to 0.1
3. **Sequential Processing (The Dispatcher Loop):**
   The system iterates through the sorted process list. For each process P_i:
   - **Idle Time Handling:** If `currentTime` < $P_i.arrivalTime$, the CPU must idle. The logic updates `currentTime = p.arrivalTime` to account for this gap.[1]
   - **Execution Simulation:**
     - $StartTime = currentTime$
     - $EndTime = StartTime + BurstTime$
   - **Metric Derivation:**
     - $Waiting\ Time = StartTime - ArrivalTime$
     - $Turnaround\ Time = EndTime - ArrivalTime$
   - State Update:
     The `currentTime` is updated to the `endTime` of the current process, advancing the system clock for $P_{\{i+1\}}$.

5.2 Handling the "Convoy Effect"

One of the theoretical disadvantages of FCFS is the "Convoy Effect," where all other processes wait for one big process to get off the CPU.[1] The simulation accurately models this behavior. By sorting strictly by arrival time, if a CPU-bound process with a large burst time arrives first, all subsequent processes suffer increased Waiting Time. This implementation allows users to empirically observe this phenomenon by manipulating input values.

6. User Interface Engineering and Experience Design

6.1 Color Theory and Visual Hierarchy

The interface employs a dark-themed, "Modern Professional" color palette designed to minimize eye strain and maximize data readability.[1]

Table 2: UI Color Palette Definition

Color Code	Role	Usage Context
#222831	Background	Main application canvas
#393E46	Containers	Data cards, Input forms
#00ADB5	Primary Action	Buttons, Active Data

		highlights
#EEEEEE	Typography	Labels, Instructions

6.2 Input Structure and Validation

The system allows dynamic process entry. The `script.js` controller implements validation checks to prevent negative numbers or null inputs, ensuring input stability before execution.[1]

6.3 Visualization: The Results Table

Upon execution, the system dynamically generates a Results Table. This output component is read-only and displays the computed metrics (TAT, WT, RT). The UI updates in real-time via JavaScript DOM manipulation, providing an "App-like" feel.[1]

7. Implementation Challenges and Engineering Solutions

7.1 Challenge: Asynchronous State vs. Synchronous Logic

Problem: A bug encountered early in Phase 2 involved the UI state not matching the internal data array when users deleted rows.
Solution: The team implemented a "Just-In-Time" data extraction strategy. The system reads data directly from the DOM at the moment the "Run" button is clicked, ensuring the simulation processes exactly what is visible to the user.1

7.2 Challenge: Responsive Gantt Chart Scaling

Problem: Displaying a long timeline on mobile screens caused rendering issues where time blocks became indistinguishable.
Solution: A CSS-based scrolling solution was adopted. The chart container uses `overflow-x: scroll`, allowing the internal canvas to expand horizontally based on total burst time. This preserves visual fidelity on small devices.1

8. Phase 3 Roadmap

With the FCFS algorithm and core architecture complete, Phase 3 will focus on:

1. **Remaining Algorithms:** Implementation of Shortest Job First (SJF), Round Robin (RR), and Priority Scheduling.
2. **Gantt Chart Rendering:** Finalizing the graphical drawing function to visualize the `startTime` and `endTime` data generated in Phase 2.[1]
3. **Comparative Analysis:** Developing a dashboard to compare Average TAT and WT across all algorithms simultaneously.

9. Project Resources & Repository

To ensure transparency and version control throughout the development of the software lifecycle, we have established the following resources:

- Source Code Repository (GitHub): (https://github.com/JoudN2001/OS-Scheduler-Simulation-Project)
- UI/UX Design Prototype (Figma): (https://www.figma.com/design/vTsK3KUjGWgyldyTu8tXic/OS-Project?node-id=0-1&t=xtRwq3WYiEmvd6iz-1)

10. References

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

[2] J. Kayyali, I. Abdullah, and A. Ahmad, "OS-Scheduler-Simulation-Project," GitHub, 2025. [Online]. Available:(https://github.com/JoudN2001/OS-Scheduler-Simulation-Project).