Summer PEP Report on DSA

Skillstone (A Grazitti Initiative)

A training Report on

Simulation and Comparative Analysis of CPU Scheduling Algorithms

Submitted in partial fulfilment of the requirements for the award of degree of

B.TECH

(Computer Science and Engineering)

Submitted To

LOVELY PROFESSIONAL UNIVERSITY PHAGWARA, PUNJAB

From 23rd June 2025 to 30th July 2025

SUBMITTED BY

Name of student: Tejasvita

Registration Number: 12314183

Signature of the student:

Student Declaration

To whom so ever it may concern

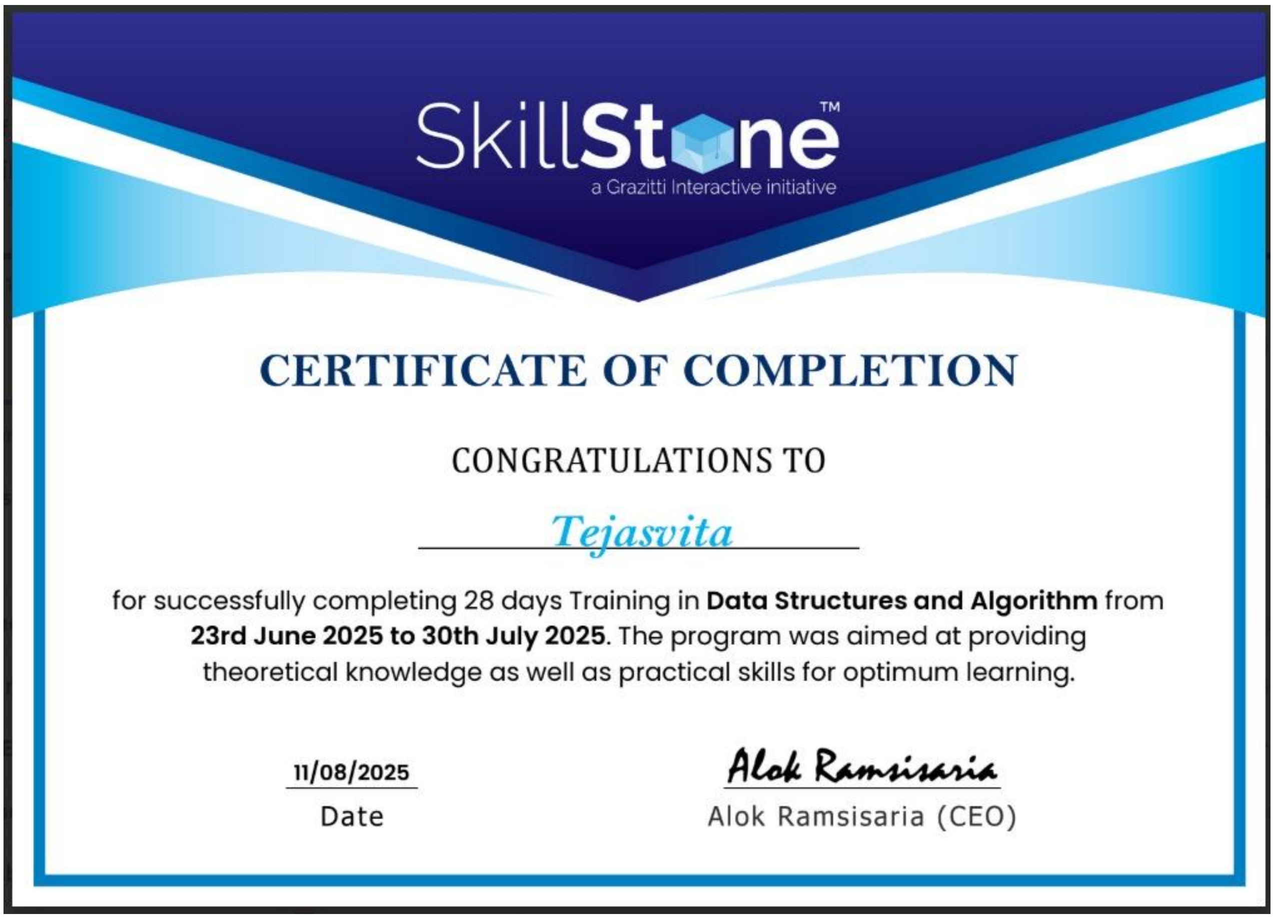
I, **Tejasvita, 12314183**, hereby declare that the work done by me on "**Simulation and Comparative Analysis of CPU Scheduling Algorithms**" from **23rd June, 2025 to 30th August, 2025**, is a record of original work for the partial fulfillment of the requirements for the award of the degree, **B.Tech (CSE)**.

Tejasvita (12314183)

Dated: 15 August, 2025

List of Contents

Summer Training certificate



SkillStone™

a Grazitti Interactive initiative

CERTIFICATE OF COMPLETION

CONGRATULATIONS TO

Tejasvita

for successfully completing 28 days Training in **Data Structures and Algorithm** from **23rd June 2025 to 30th July 2025**. The program was aimed at providing theoretical knowledge as well as practical skills for optimum learning.

11/08/2025

Date

Alok Ramsisaria

Alok Ramsisaria (CEO)

Acknowledgment

I would like to express my sincere gratitude to all those who provided me with guidance and support during my summer training. This report is a result of their valuable contributions.

First and foremost, I am immensely thankful to **Lovely Professional University** for giving me the opportunity to undertake my training at their organization. I extend my deepest gratitude to my mentor **Mr. Manit Saharan**, for their invaluable guidance, support, and continuous encouragement throughout my training period. Their expertise and insights were crucial to the successful completion of this project.

I am also grateful to all the employees and batchmates at LPU for their assistance and cooperation. Their willingness to share their knowledge and resources made my learning experience both rewarding and enjoyable.

Finally, I would like to thank my faculty advisor at Lovely Professional University for their academic support, as well as my family and friends for their unwavering support.

Tejasvita
(12314183)

List of Tables

List of Figures / Charts

Chapter 1

INTRODUCTION OF THE PROJECT UNDERTAKEN

The project, *React-based CPU Scheduling Simulator*, is designed to simulate and visualize the working of various CPU scheduling algorithms such as First Come First Serve (FCFS), Shortest Job First (SJF), Priority Scheduling, and Round Robin. CPU scheduling is a crucial aspect of operating system design, determining how processes are selected for execution and in what order.

By implementing these algorithms in a web-based interface using *React.js*, the project aims to provide an interactive and educational tool for students, educators, and developers to understand the working principles, performance metrics, and comparative analysis of different scheduling techniques.

1.1 Problem Statement

Efficient CPU scheduling is a fundamental requirement in modern operating systems to ensure optimal utilization of the CPU and improve system performance.

However, students and beginners in operating system concepts often face difficulties in understanding and visualizing how different scheduling algorithms work, how they impact process execution order, waiting time, and turnaround time.

Without an interactive and practical tool, learning remains abstract and theoretical. Therefore, there is a need for a **CPU Scheduling Simulator** that allows users to input process details, select various scheduling algorithms, and observe the resulting Gantt charts and performance metrics.

This will help bridge the gap between theoretical concepts and practical understanding of CPU scheduling techniques.

1.2 Objectives of the Work Undertaken

- To develop a user-friendly and interactive web application for simulating CPU scheduling algorithms.
- To allow users to input custom process data including arrival time, burst time, priority, and quantum (for Round Robin).
- To generate visual Gantt charts for execution order representation.
- To calculate and display performance metrics such as Waiting Time, Turnaround Time, and Completion Time.

- To facilitate comparative analysis of algorithms based on input data.
- To deploy the project online for accessibility from any device with a web browser.

1.3 Scope of the Work

- Algorithms Covered: FCFS, SJF (Non-Preemptive), Priority Scheduling (Non-Preemptive), Round Robin.
- Platform: Web-based, developed using React.js for frontend logic and Recharts/Chart.js for visualization.
- Input Flexibility: Supports dynamic process addition and removal.
- Output: Tabular results and Gantt chart representation for easy analysis.
- Extensibility: Can be upgraded to include Preemptive Scheduling, Multi-Level Queue Scheduling, or integration with backend storage for saving simulation data.

1.4 Importance and Applicability

- Educational Value: Serves as a learning tool for computer science students studying operating systems.
- Visualization Aid: Helps in understanding how scheduling decisions impact process execution order and performance metrics.
- Comparative Analysis: Provides instant comparison of algorithms for given inputs.
- Real-world Relevance: Reflects fundamental concepts used in real operating system schedulers, making it relevant for academic projects and interview preparation.

1.5 Role and Profile

Role: Full-stack web developer (Frontend-heavy) for CPU Scheduling Simulator.

Profile Responsibilities:

- Conducting research on scheduling algorithms.
- Designing the UI/UX for the simulator.
- Implementing scheduling logic in JavaScript.
- Integrating visual components (charts and tables) in React.
- Testing, debugging, and deploying the application.
- Preparing project documentation and final presentation

1.6 Work Plan and Implementation

The project is organized into multiple phases to ensure systematic development, testing, and completion within the stipulated time. Each phase focuses on specific objectives, ensuring that the final product meets both functional and academic requirements.

Phase 1: Requirement Analysis & Research (Day 1–Day 2)

- Study the working of various CPU scheduling algorithms such as **FCFS**, **SJF**, **Priority Scheduling**, and **Round Robin**.
- Identify necessary input parameters: process ID, arrival time, burst time, priority, and quantum (for Round Robin).
- Select technologies: **React.js** for frontend, JavaScript for logic, and **Chart.js/Recharts** for visualization.
- Prepare wireframes for UI layout including input forms, Gantt chart area, and result tables.

Phase 2: Project Setup (Day 3)

- Initialize React project using **Vite** or **Create React App**.
- Organize folder structure (components, utils, styles, data).
- Install required dependencies:
  npm install recharts chart.js react-chartjs-2
- Configure routing if needed for multi-view navigation.

Phase 3: Algorithm Implementation (Day 4–Day 8)

- Implement First Come First Serve (FCFS) algorithm.
- Implement Shortest Job First (SJF) — Non-Preemptive version.
- Implement Priority Scheduling — Non-Preemptive.
- Implement Round Robin with configurable time quantum.
- Store algorithms inside utils/algorithms.js for maintainability.
- Validate outputs using known test datasets.

Phase 4: User Interface Development (Day 9–Day 13)

- Input Form Component: For entering processes and parameters dynamically.
- Results Table Component: Displays waiting time, turnaround time, and completion time.
- Gantt Chart Component: Uses Chart.js or Recharts for visualizing execution order.
- Optional: Comparison View for algorithm performance comparison.
- Style components using Tailwind CSS or CSS Modules for responsive design.

Phase 5: Testing & Debugging (Day 14–Day 15)

- Test algorithms against multiple input sets.
- Verify responsiveness on different screen sizes.
- Fix calculation or UI rendering issues.

Phase 6: Deployment (Day 16)

- Deploy application using Netlify, Vercel, or GitHub Pages.
- Verify hosted application accessibility and performance.

Phase 7: Documentation & Submission (Day 17)

- Prepare user guide with screenshots.
- Compile final project report including objectives, scope, implementation details, and results.
- Submit both source code and documentation for evaluation.

Gantt Chart – Project Timeline

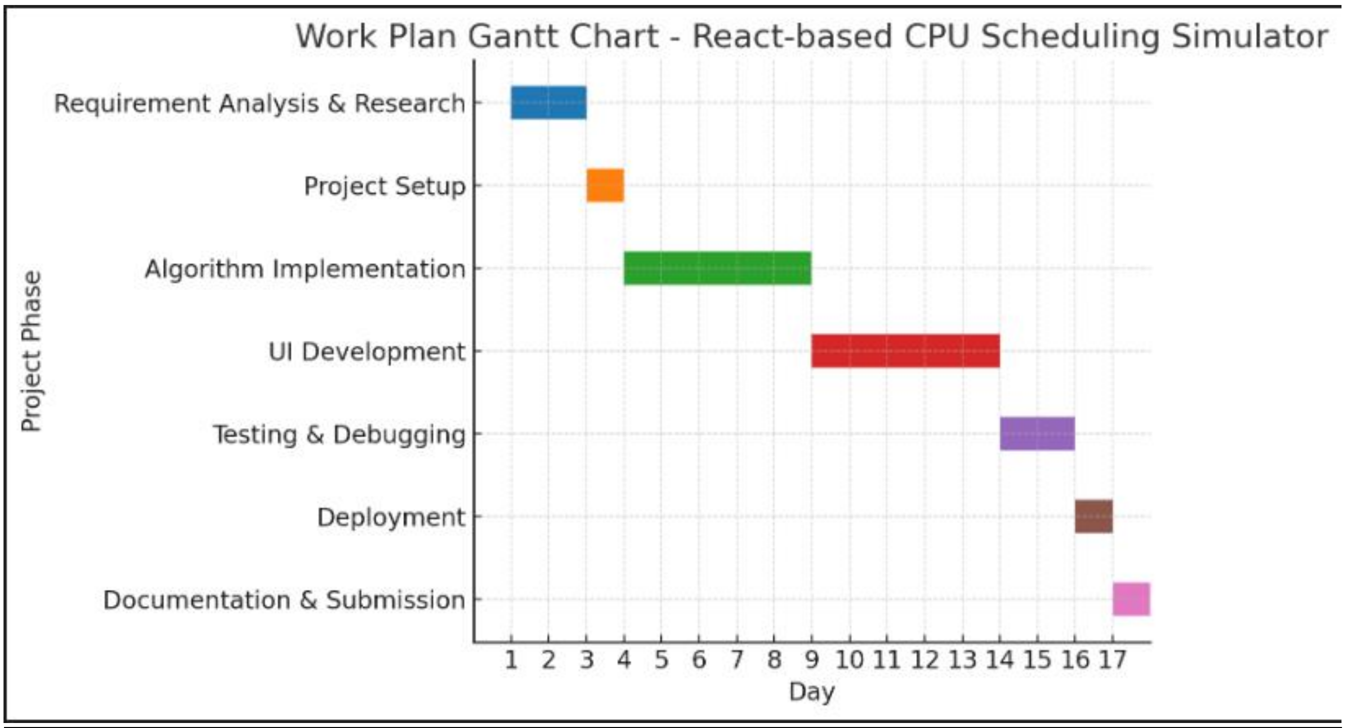


Figure 1.1: Work plan for React-based CPU Scheduling Simulator

Chapter 2
COMPANY PROFILE

2.1 Overview of the Organisastion

SkillStone is a professional training and skill development platform designed to bridge the gap between academic learning and industry requirements. It operates as a Grazitti Interactive initiative, providing specialized training programs for students and working professionals in emerging technologies such as Data Structures & Algorithms, Web Development, Cloud Computing, Artificial Intelligence, and more.

SkillStone focuses on delivering practical, hands-on learning experiences, enabling learners to gain job-ready skills through real-world projects, mentorship from industry experts, and guided assignments. The platform offers structured learning paths tailored to meet the needs of engineering students and aspiring IT professionals.

2.2 Parent Organization – Grazitti Interactive

Grazitti Interactive is a global digital services provider headquartered in Panchkula, Haryana, India, with offices in multiple countries. Founded in 2008, the company specializes in:

- Digital Marketing Solutions
- Web & Application Development
- Data Analytics and AI
- Cloud Services & CRM Solutions
- Product Engineering

Grazitti serves clients worldwide, including Fortune 500 companies, and has a strong focus on technology innovation and customized solutions.

2.3 Services offered by SkillStone

- Industry-Oriented Training: Focused on latest market trends and employer needs.
- Live & Recorded Classes: Flexibility to learn at one’s own pace.
- Hands-on Projects: Encouraging application of learned skills.
- Mentorship & Career Guidance: Resume building, interview preparation, and career counseling.
- Technical Competitions & Hackathons: Enhancing problem-solving skills in a competitive environment.

2.4 Relevance to the Project

The React-based CPU Scheduling Simulator project undertaken during my Data Structures & Algorithms training with SkillStone is directly aligned with the platform’s vision of practical, application-oriented learning.

- The project applies algorithmic concepts learned in training to a real-world application.
- It strengthens both **frontend development skills** (React.js) and **core computer science fundamentals** (OS scheduling algorithms).
- It follows SkillStone’s emphasis on problem-solving, coding efficiency, and clear documentation.

2.5 Contact Information

SkillStone – A Grazitti Interactive Initiative

Address: SCO 12, Sector 14, Panchkula, Haryana – 134113, India

Website: https://skillstone.in

Email: info@skillstone.in

Chapter 3

LITERATURE REVIEW / THEORETICAL BACKGROUND

3.1 Introduction to CPU Scheduling

In a multiprogramming operating system, multiple processes compete for the CPU at any given time. The **CPU scheduling** process determines the order in which processes are executed, aiming to optimize various performance criteria such as CPU utilization, throughput, turnaround time, and waiting time.

Efficient CPU scheduling is essential for improving overall system performance and user experience.

3.2 Objectives of CPU Scheduling

- Maximize CPU Utilization – Ensure the CPU is always busy.
- Maximize Throughput – Increase the number of processes completed per unit time.
- Minimize Turnaround Time – Reduce the total time taken for process completion.
- Minimize Waiting Time – Reduce the time processes spend in the ready queue.
- Ensure Fairness – All processes get a fair share of CPU time.

3.3 CPU Scheduling algorithms

S. No.	Algorithm	Description	Advantages	Disadvantages
1	First Come, First Serve (FCFS)	Processes are executed in the order they arrive in the ready queue.	Simple to implement; fair for batch systems.	Poor average waiting time; not suitable for time-sharing.
2	Shortest Job First (SJF)	Executes process with the smallest CPU burst time first.	Optimal for average waiting time.	Requires prior knowledge of burst time; prone to starvation.
3	Priority Scheduling	Processes are scheduled based on assigned priority.	Useful for processes with different importance levels.	Low-priority processes may starve.

4	Round Robin (RR)	Each process gets a fixed time quantum in a cyclic order.	Fair for all processes; suitable for time-sharing systems.	Choosing optimal quantum size is critical.

Table 3.1: CPU Scheduling algorithms

3.4 Visualization need in CPU Scheduling

While scheduling algorithms are theoretically simple, students and beginners often find it challenging to visualize how processes switch in real time. A visual simulator provides:

- Dynamic Gantt Charts showing process execution order.
- Comparisons between algorithms based on performance metrics.
- Interactive input where users can enter processes, burst times, and priorities.

3.5 Use of React for the FrontEnd

React.js is chosen for the project frontend because:

- Component-Based Architecture – Reusable UI components for process input forms, charts, and metrics display.
- Fast Rendering – Virtual DOM ensures smooth updates when switching between algorithms.
- Rich Ecosystem – Libraries for charts and animations enhance visualization.
- State Management – Easy to manage dynamic process data.

3.6 Expected Outcome of the Literature Review

From the study of CPU scheduling techniques and modern frontend frameworks, this project aims to:

- Implement multiple scheduling algorithms.
- Provide a user-friendly, interactive, and educational simulator.
- Help learners understand OS scheduling concepts through visual feedback.

Chapter 4
SYSTEM DESIGN

4.1 Overview

The CPU Scheduling Simulator is designed as a web-based interactive application built with React.js to visualize and compare different scheduling algorithms. The system takes user inputs for process details and algorithm selection, then processes the data to generate scheduling results and Gantt chart visualizations.

4.2 System Architecture

Architecture Description

The system follows a client-side architecture with React handling all UI logic and scheduling computation within the browser. No backend server is required unless data persistence is later added.

Layers:

1. Presentation Layer (UI) – Built in React, handles forms, inputs, and result display.
2. Logic Layer – Implements the CPU scheduling algorithms in JavaScript modules.
3. Visualization Layer – Generates Gantt charts and tabular outputs.
4. Data Handling Layer – Validates and processes user input.

4.3 Flow Chart

Steps in Flowchart:

1. Start
2. Accept number of processes and their attributes (arrival time, burst time, priority).
3. Select scheduling algorithm (FCFS, SJF, Priority, RR).
4. Pass inputs to the selected algorithm module.
5. Compute scheduling order and waiting/turnaround times.
6. Generate Gantt chart and summary table.
7. Display results in UI.
8. End.

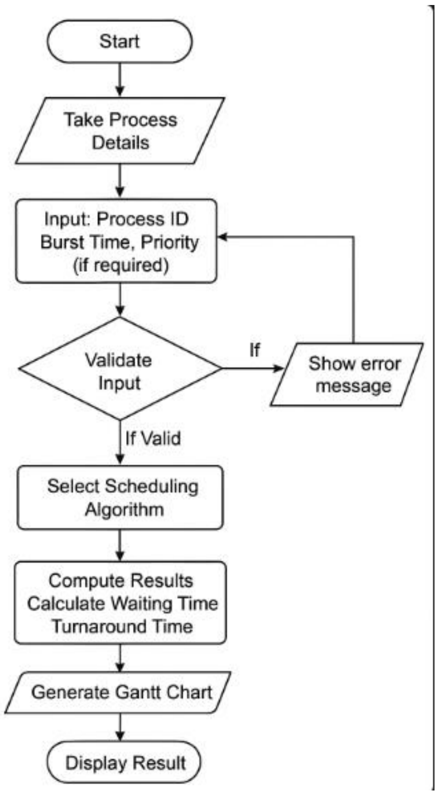

Start
Take Process Details
Input: Process ID Burst Time, Priority (if required)
Validate Input
If
Show error message
If Valid
Select Scheduling Algorithm
Compute Results Calculate Waiting Time Turnaround Time
Generate Gantt Chart
Display Result

Figure 4.1: Flow Chart of the System

4.4 Module Description

Module	**Description**
Input Module	Handles process data entry, algorithm selection, and time quantum (for RR).
Validation Module	Ensures data correctness (e.g., no negative times, valid priorities).
Algorithm Module	Contains JavaScript implementations of FCFS, SJF, Priority, and RR.
Computation Module	Calculates waiting time, turnaround time, and average metrics
Visualization Module	Generates dynamic Gantt charts and results tables.
UI Module	Manages all user interactions and responsive layout.

Table 4.1: Module Description

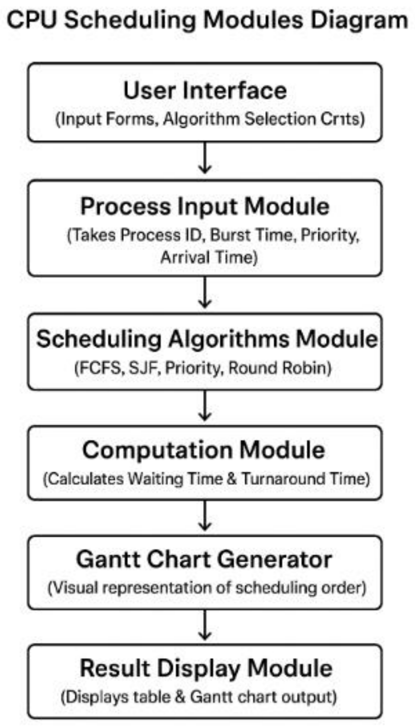

CPU Scheduling Modules Diagram
User Interface
(Input Forms, Algorithm Selection Crıts)
Process Input Module
(Takes Process ID, Burst Time, Priority, Arrival Time)
Scheduling Algorithms Module
(FCFS, SJF, Priority, Round Robin)
Computation Module
(Calculates Waiting Time & Turnaround Time)
Gantt Chart Generator
(Visual representation of scheduling order)
Result Display Module
(Displays table & Gantt chart output)

Figure 4.2: Module Interaction Diagram

4.5 Use Case Diagram

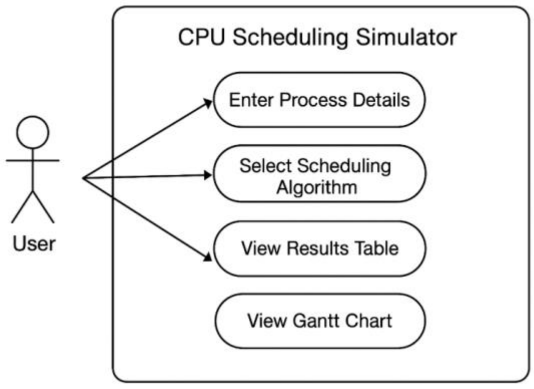


Figure 4.3: Use Case Diagram

The above diagram depicts the interaction between the user and the CPU Scheduling Simulator. It highlights the major functionalities, such as entering process details, selecting the scheduling algorithm, viewing the results in a tabular form, and visualizing the Gantt chart.

4.6 Data-Flow Diagram (DFD)

Level 0:

User inputs → Processing → Output Display.

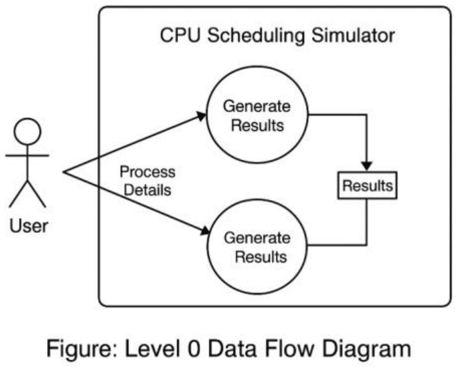


Figure 4.4: Level 0 DFD

Level 1:

Process Input → Algorithm Selection → Computation → Visualization → Output Display.

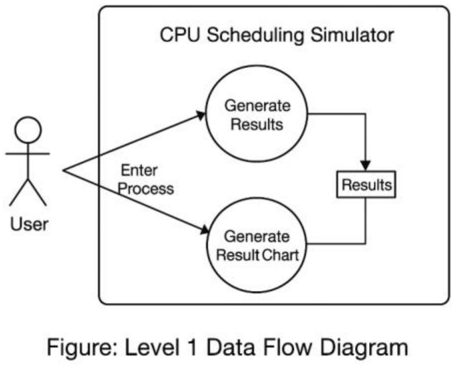


Figure 4.5: Level 1 DFD

4.7 Activity Diagram

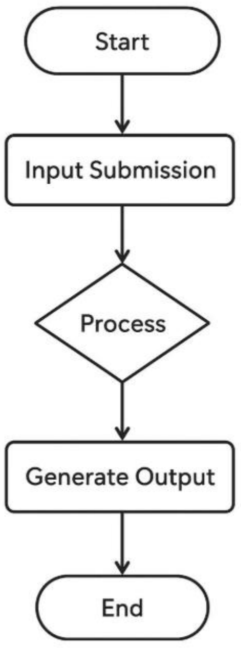


Figure 4.6: Activity Diagram

the activity diagram depicts how the user interacts with the system, starting from input submission to the final output display, providing a clear view of the workflow and process sequence.

4.8 ER Diagram

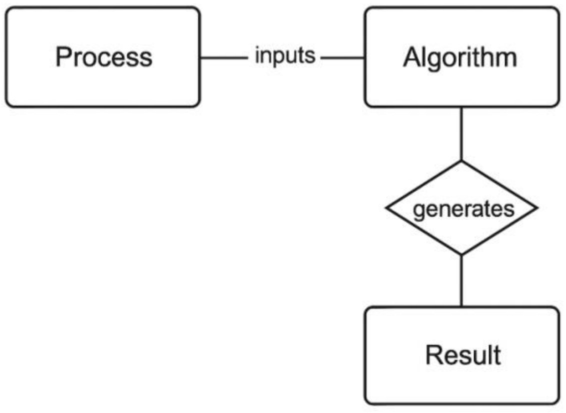


Figure 4.7: ER diagram

- Entities - User, Process, Scheduler.
- Attributes - UserID, ProcessName, BurstTime, ArrivalTime.
- Relationships - User creates Process, Scheduler handles Process.

4.9 Example of Gantt Chart

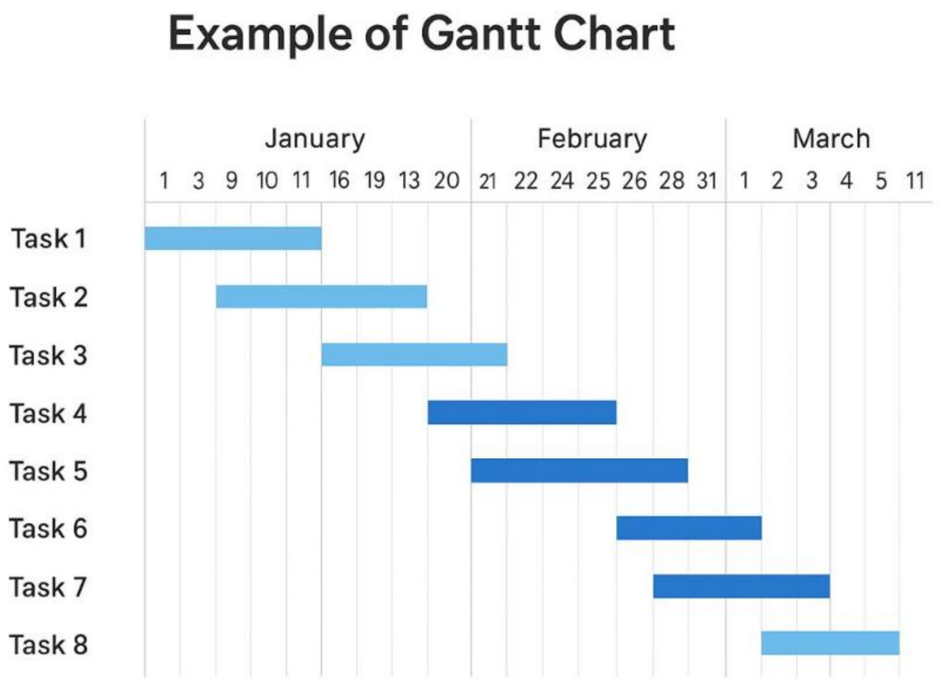


Figure 4.8: Example of Gantt Chart

4.10 Technology Stack

- Frontend Framework: React.js
- Programming Language: JavaScript (ES6)
- Styling: CSS / TailwindCSS
- Chart Library: Chart.js or Recharts (for Gantt charts)
- Build Tool: Vite or Create React App

4.11 Design Considerations

- Frontend Framework: React.js
- Programming Language: JavaScript (ES6)
- Styling: CSS / TailwindCSS
- Chart Library: Chart.js or Recharts (for Gantt charts)
- Build Tool: Vite or Create React App

Chapter 5
IMPLEMENTATION

5.1 Introduction

Implementation is the process of transforming the system design into an operational system through coding, integration, and testing. This chapter describes the development of the CPU Scheduling Simulator using React for the front end and supporting tools for styling and interactivity. The implementation ensures that all designed modules are functional, user-friendly, and meet the project's objectives.

5.2 Tools and Technologies used

Tool / Technology	Purpose
React.js	Front-end library for building interactive user interfaces.
HTML5	Structure and layout of the web application.
CSS3	Styling and presentation of components.
JavaScript (ES6)	Logic implementation and interactivity.
React hooks	State management and lifecycle handling.
BootStrap / TailWind CSS	(If used) For responsive UI design.
VS code	Integrated Development Environment (IDE) used for coding.
Git & GitHub	Version control and project hosting.

Table 5.1: Tools and Technologies used

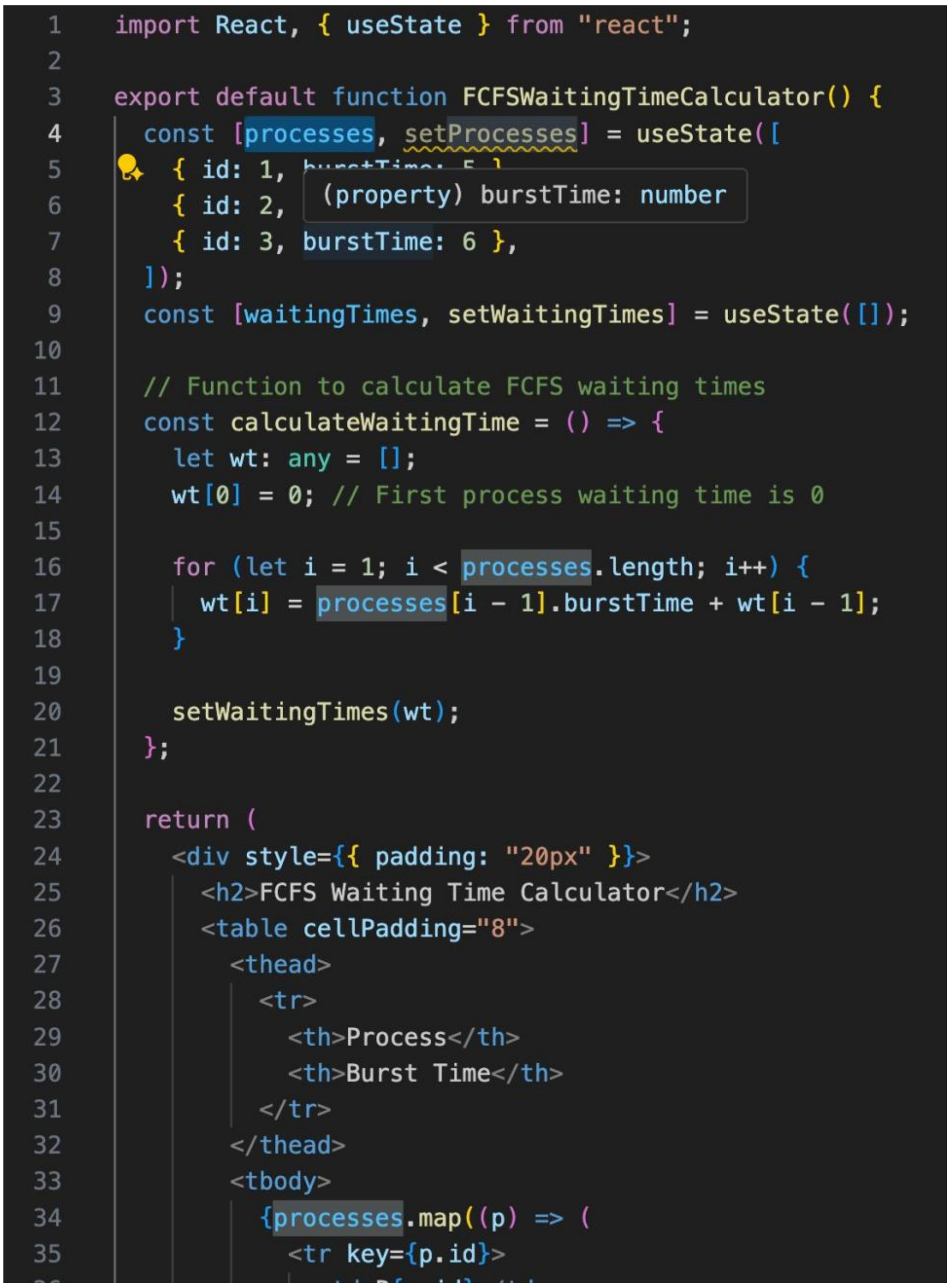
5.3 Module Implementation

Module Name	Description
Input Module	Accepts process details like Process ID, Burst Time, and Arrival Time from the user. Validates input before processing.
Algorithm Selection Module	Allows the user to select a CPU scheduling algorithm (FCFS, SJF, Priority, Round Robin).
Processing Module	Applies the selected scheduling algorithm to calculate waiting time, turnaround time, and Gantt chart order.
Output Module	Displays results in tabular format along with the visual Gantt chart.
Results Analysis Module	Provides average waiting time and average turnaround time for evaluation.

Table 5.2: Module Implementation

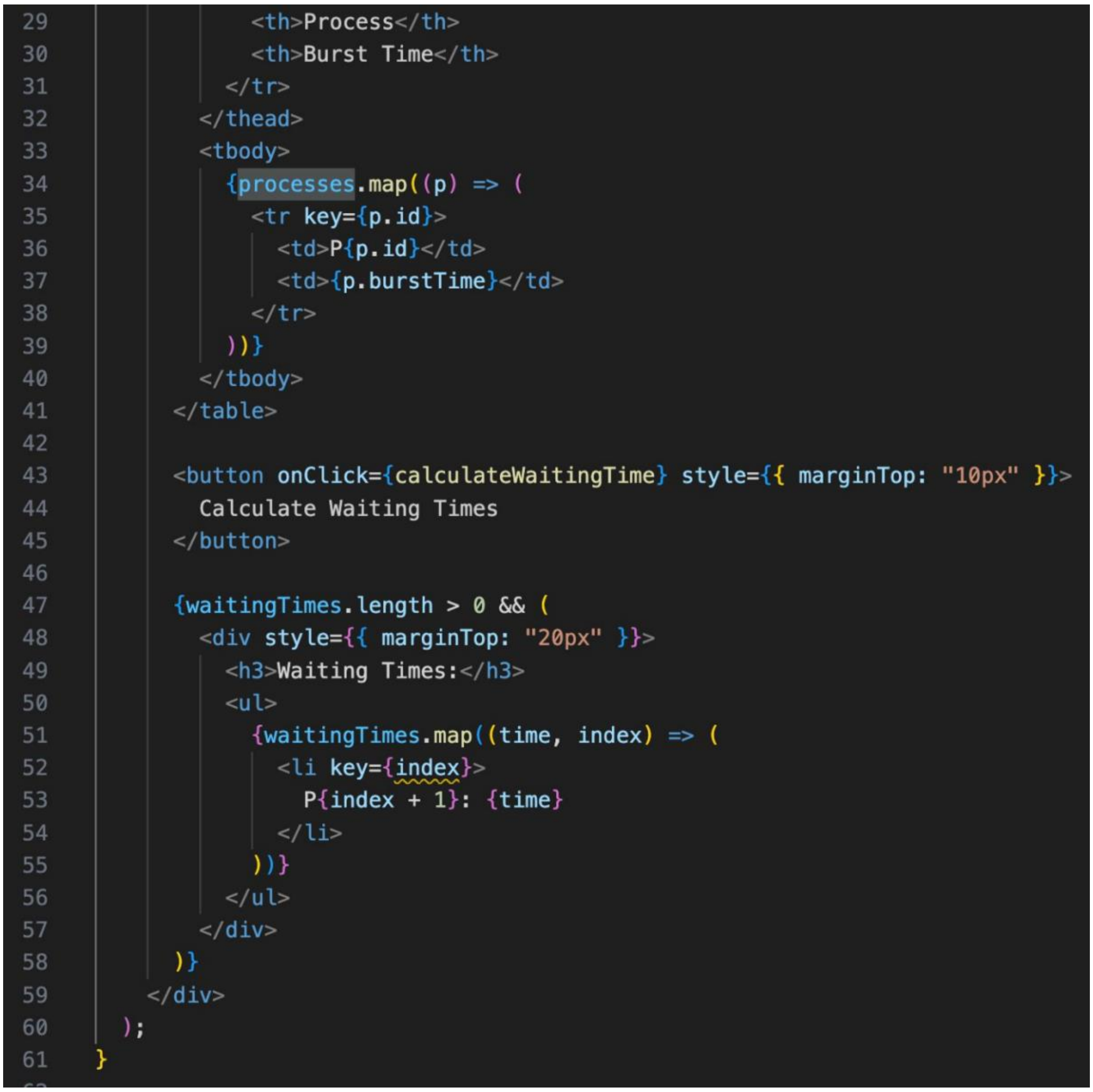
5.4 Code Snippets

Figure 5.1: Function to calculate FCFS Waiting Time:



```
import React, { useState } from "react";

export default function FCFSWaitingTimeCalculator() {
  const [processes, setProcesses] = useState([
    { id: 1, burstTime: 5 },
    { id: 2,  (property) burstTime: number
    { id: 3, burstTime: 6 },
  ]);
  const [waitingTimes, setWaitingTimes] = useState([]);

  // Function to calculate FCFS waiting times
  const calculateWaitingTime = () => {
    let wt: any = [];
    wt[0] = 0; // First process waiting time is 0

    for (let i = 1; i < processes.length; i++) {
      wt[i] = processes[i - 1].burstTime + wt[i - 1];
    }

    setWaitingTimes(wt);
  };

  return (
    <div style={{ padding: "20px" }}>
      <h2>FCFS Waiting Time Calculator</h2>
      <table cellPadding="8">
        <thead>
          <tr>
            <th>Process</th>
            <th>Burst Time</th>
          </tr>
        </thead>
        <tbody>
          {processes.map((p) => (
            <tr key={p.id}>
```

Figure 5.1(a)



```
            <th>Process</th>
            <th>Burst Time</th>
          </tr>
        </thead>
        <tbody>
          {processes.map((p) => (
            <tr key={p.id}>
              <td>P{p.id}</td>
              <td>{p.burstTime}</td>
            </tr>
          ))}
        </tbody>
      </table>

      <button onClick={calculateWaitingTime} style={{ marginTop: "10px" }}>
        Calculate Waiting Times
      </button>

      {waitingTimes.length > 0 && (
        <div style={{ marginTop: "20px" }}>
          <h3>Waiting Times:</h3>
          <ul>
            {waitingTimes.map((time, index) => (
              <li key={index}>
                P{index + 1}: {time}
              </li>
            ))}
          </ul>
        </div>
      )}
    </div>
  );
}
```

Figure 5.1(b)

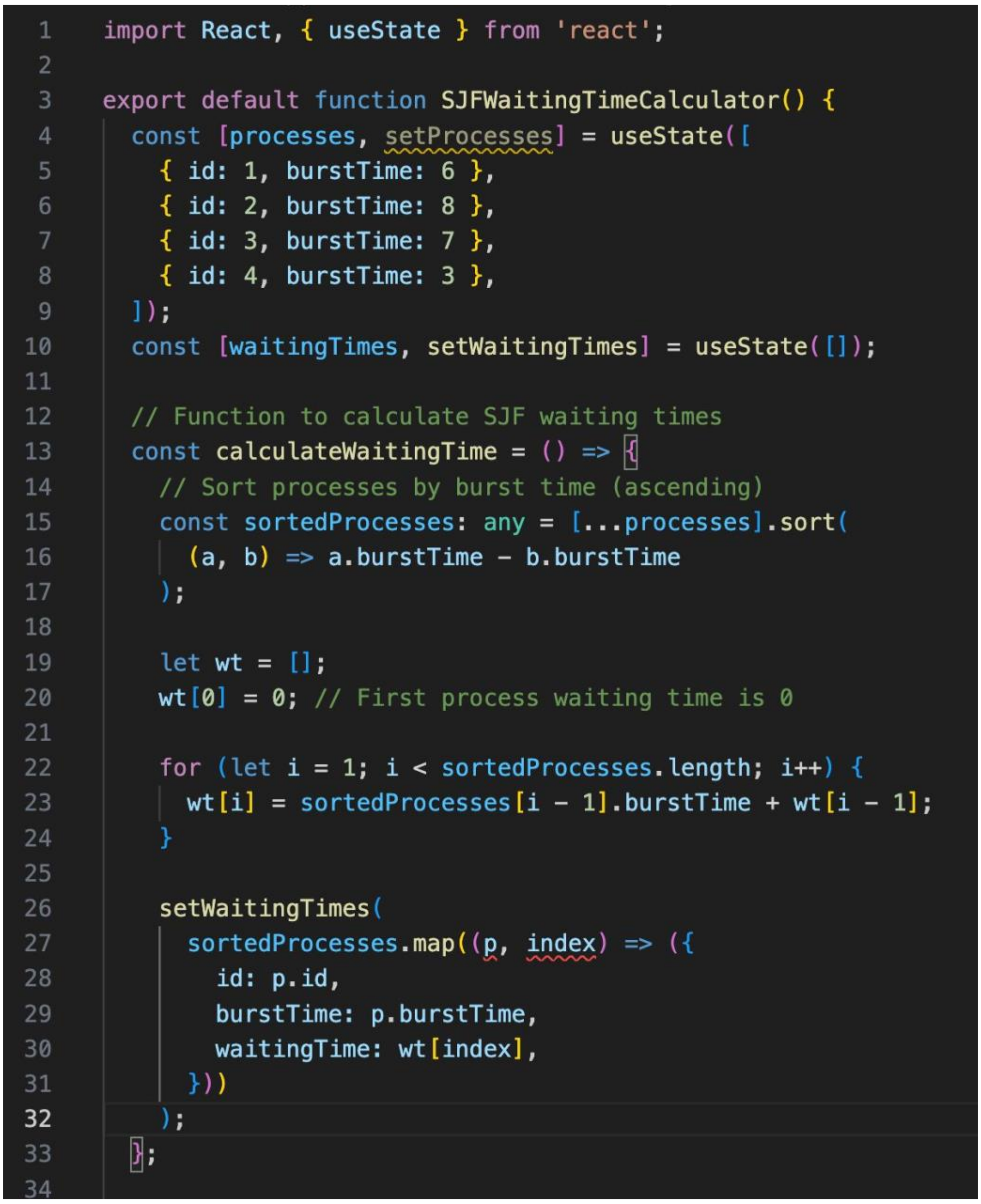
<u>How it works:</u>

Initial Data → processes array with burstTime for each process.

FCFS Logic → Waiting time for process i = Waiting time of process (i-1) + Burst time of (i-1).

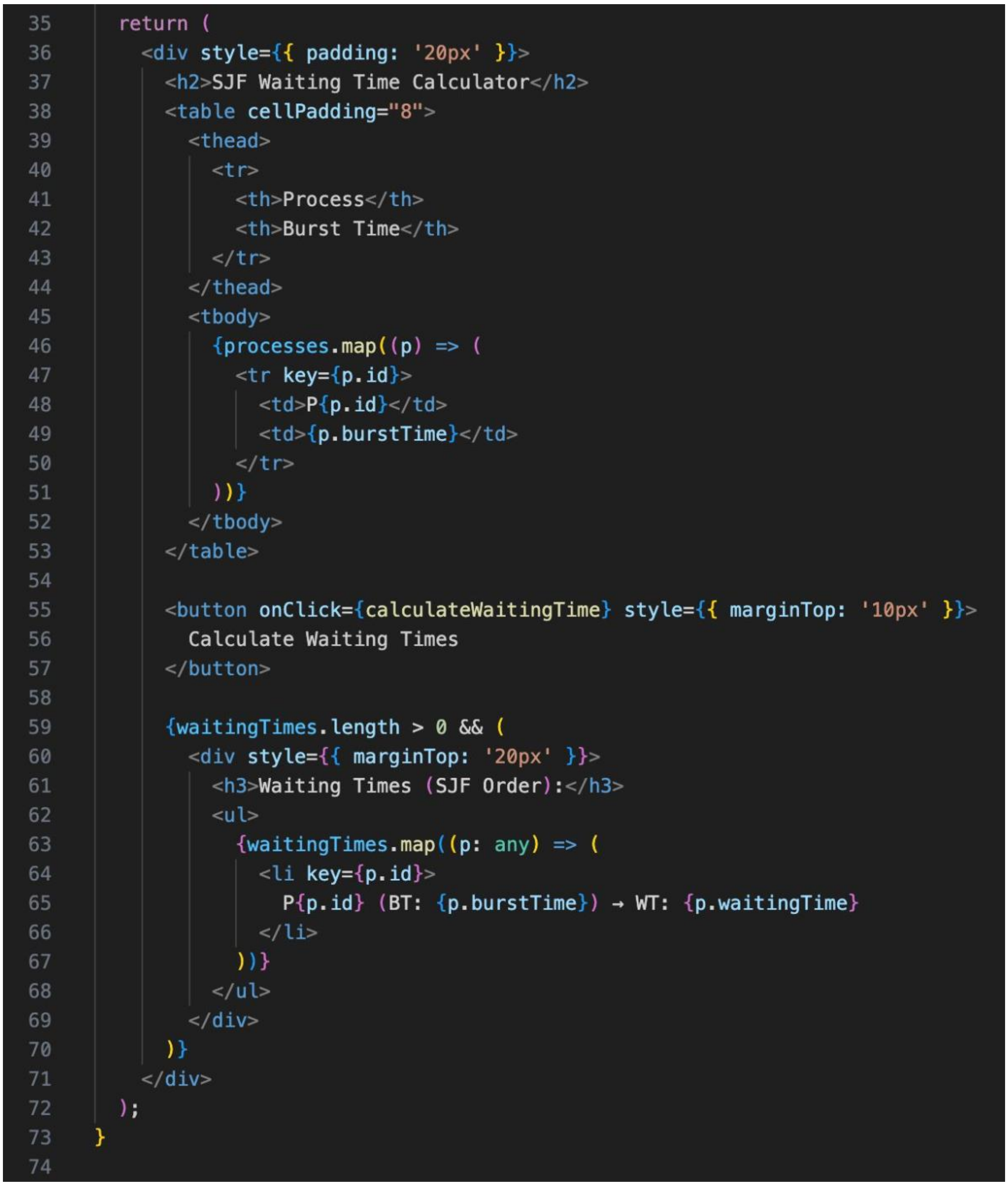
Output → Displays a list of waiting times after clicking the button.

Figure 5.2: Shortest Job First (SJF) (non-preemptive) waiting time calculation.



```
import React, { useState } from 'react';

export default function SJFWaitingTimeCalculator() {
  const [processes, setProcesses] = useState([
    { id: 1, burstTime: 6 },
    { id: 2, burstTime: 8 },
    { id: 3, burstTime: 7 },
    { id: 4, burstTime: 3 },
  ]);
  const [waitingTimes, setWaitingTimes] = useState([]);

  // Function to calculate SJF waiting times
  const calculateWaitingTime = () => {
    // Sort processes by burst time (ascending)
    const sortedProcesses: any = [...processes].sort(
      (a, b) => a.burstTime - b.burstTime
    );

    let wt = [];
    wt[0] = 0; // First process waiting time is 0

    for (let i = 1; i < sortedProcesses.length; i++) {
      wt[i] = sortedProcesses[i - 1].burstTime + wt[i - 1];
    }

    setWaitingTimes(
      sortedProcesses.map((p, index) => ({
        id: p.id,
        burstTime: p.burstTime,
        waitingTime: wt[index],
      }))
    );
  };

```

Figure 5.2(a)

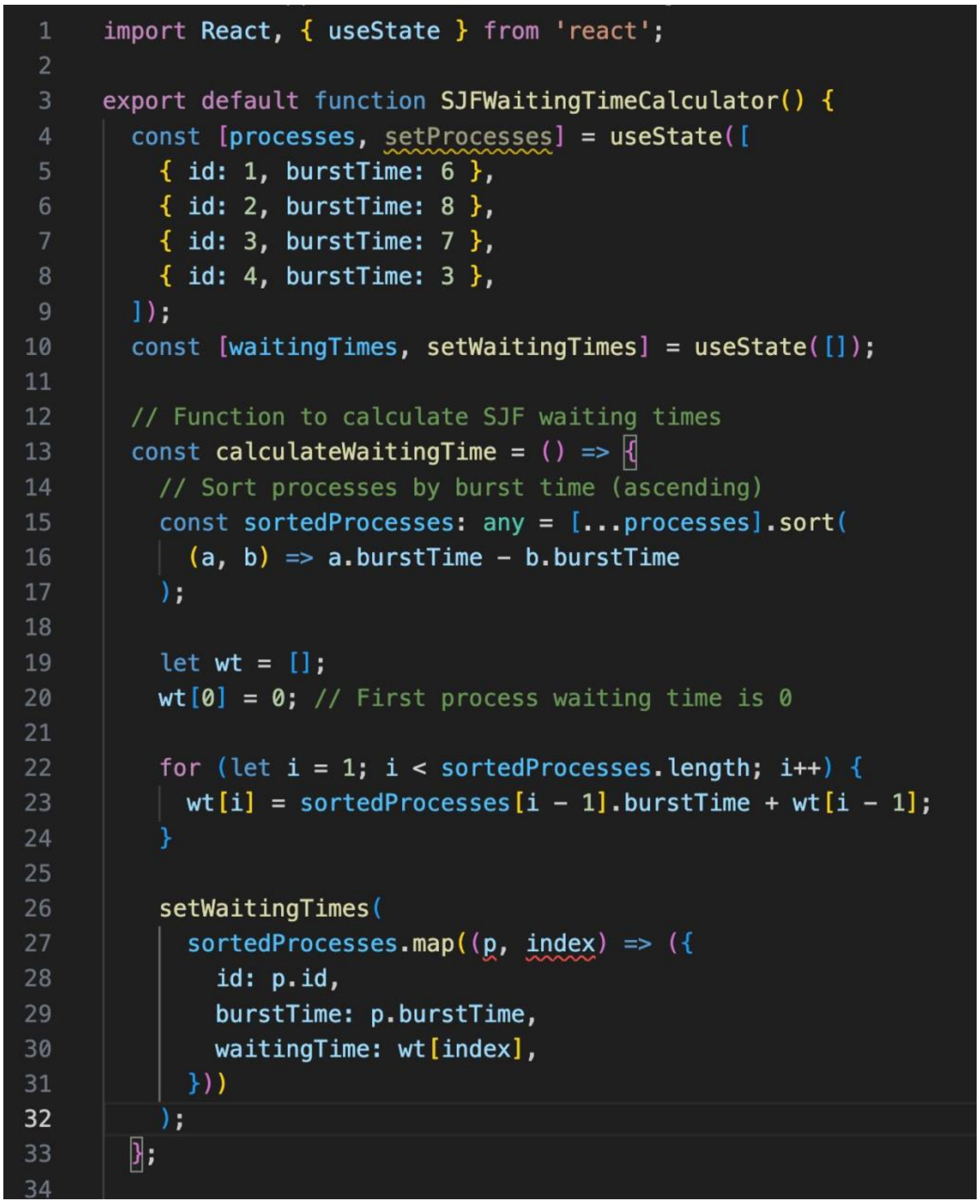


```
  return (
    <div style={{ padding: '20px' }}>
      <h2>SJF Waiting Time Calculator</h2>
      <table cellPadding="8">
        <thead>
          <tr>
            <th>Process</th>
            <th>Burst Time</th>
          </tr>
        </thead>
        <tbody>
          {processes.map((p) => (
            <tr key={p.id}>
              <td>P{p.id}</td>
              <td>{p.burstTime}</td>
            </tr>
          ))}
        </tbody>
      </table>

      <button onClick={calculateWaitingTime} style={{ marginTop: '10px' }}>
        Calculate Waiting Times
      </button>

      {waitingTimes.length > 0 && (
        <div style={{ marginTop: '20px' }}>
          <h3>Waiting Times (SJF Order):</h3>
          <ul>
            {waitingTimes.map((p: any) => (
              <li key={p.id}>
                P{p.id} (BT: {p.burstTime}) → WT: {p.waitingTime}
              </li>
            ))}
          </ul>
        </div>
      )}
    </div>
  );
}
```

Figure 5.2(b)

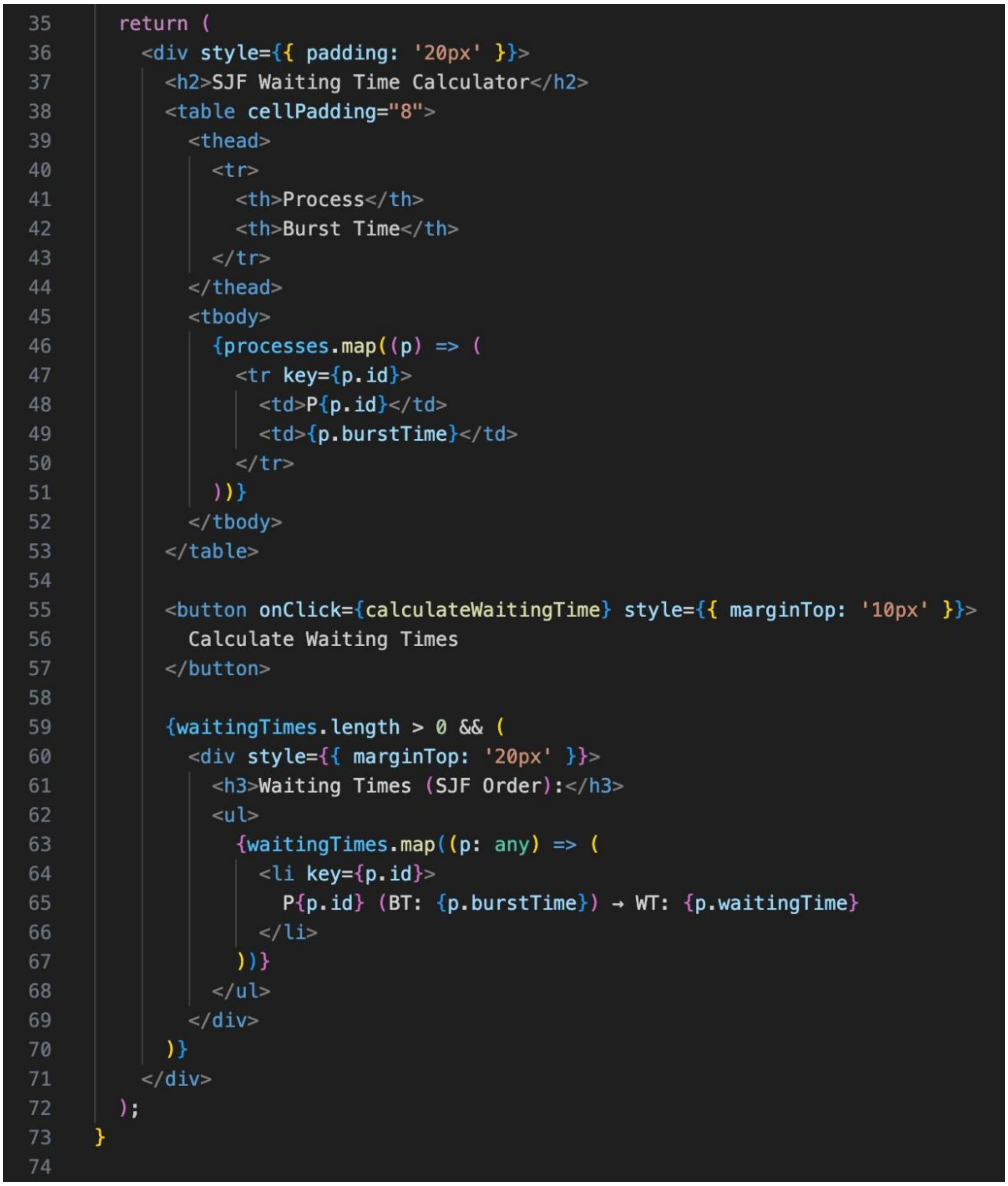
<u>How it works:</u>
Initial Data ⟶ processes array with burstTime for each process.
FCFS Logic ⟶ Waiting time for process i = Waiting time of process (i-1) + Burst time of (i-1).
Output ⟶ Displays a list of waiting times after clicking the button.

Figure 5.3: Shortest Job First (SJF) (non-preemptive) waiting time calculation.



```
import React, { useState } from 'react';

export default function SJFWaitingTimeCalculator() {
  const [processes, setProcesses] = useState([
    { id: 1, burstTime: 6 },
    { id: 2, burstTime: 8 },
    { id: 3, burstTime: 7 },
    { id: 4, burstTime: 3 },
  ]);
  const [waitingTimes, setWaitingTimes] = useState([]);

  // Function to calculate SJF waiting times
  const calculateWaitingTime = () => {
    // Sort processes by burst time (ascending)
    const sortedProcesses: any = [...processes].sort(
      (a, b) => a.burstTime - b.burstTime
    );

    let wt = [];
    wt[0] = 0; // First process waiting time is 0

    for (let i = 1; i < sortedProcesses.length; i++) {
      wt[i] = sortedProcesses[i - 1].burstTime + wt[i - 1];
    }

    setWaitingTimes(
      sortedProcesses.map((p, index) => ({
        id: p.id,
        burstTime: p.burstTime,
        waitingTime: wt[index],
      }))
    );
  };

```

Figure 5.3(a)



```
  return (
    <div style={{ padding: '20px' }}>
      <h2>SJF Waiting Time Calculator</h2>
      <table cellPadding="8">
        <thead>
          <tr>
            <th>Process</th>
            <th>Burst Time</th>
          </tr>
        </thead>
        <tbody>
          {processes.map((p) => (
            <tr key={p.id}>
              <td>P{p.id}</td>
              <td>{p.burstTime}</td>
            </tr>
          ))}
        </tbody>
      </table>

      <button onClick={calculateWaitingTime} style={{ marginTop: '10px' }}>
        Calculate Waiting Times
      </button>

      {waitingTimes.length > 0 && (
        <div style={{ marginTop: '20px' }}>
          <h3>Waiting Times (SJF Order):</h3>
          <ul>
            {waitingTimes.map((p: any) => (
              <li key={p.id}>
                P{p.id} (BT: {p.burstTime}) → WT: {p.waitingTime}
              </li>
            ))}
          </ul>
        </div>
      )}
    </div>
  );
}

```

Figure 5.3(b)

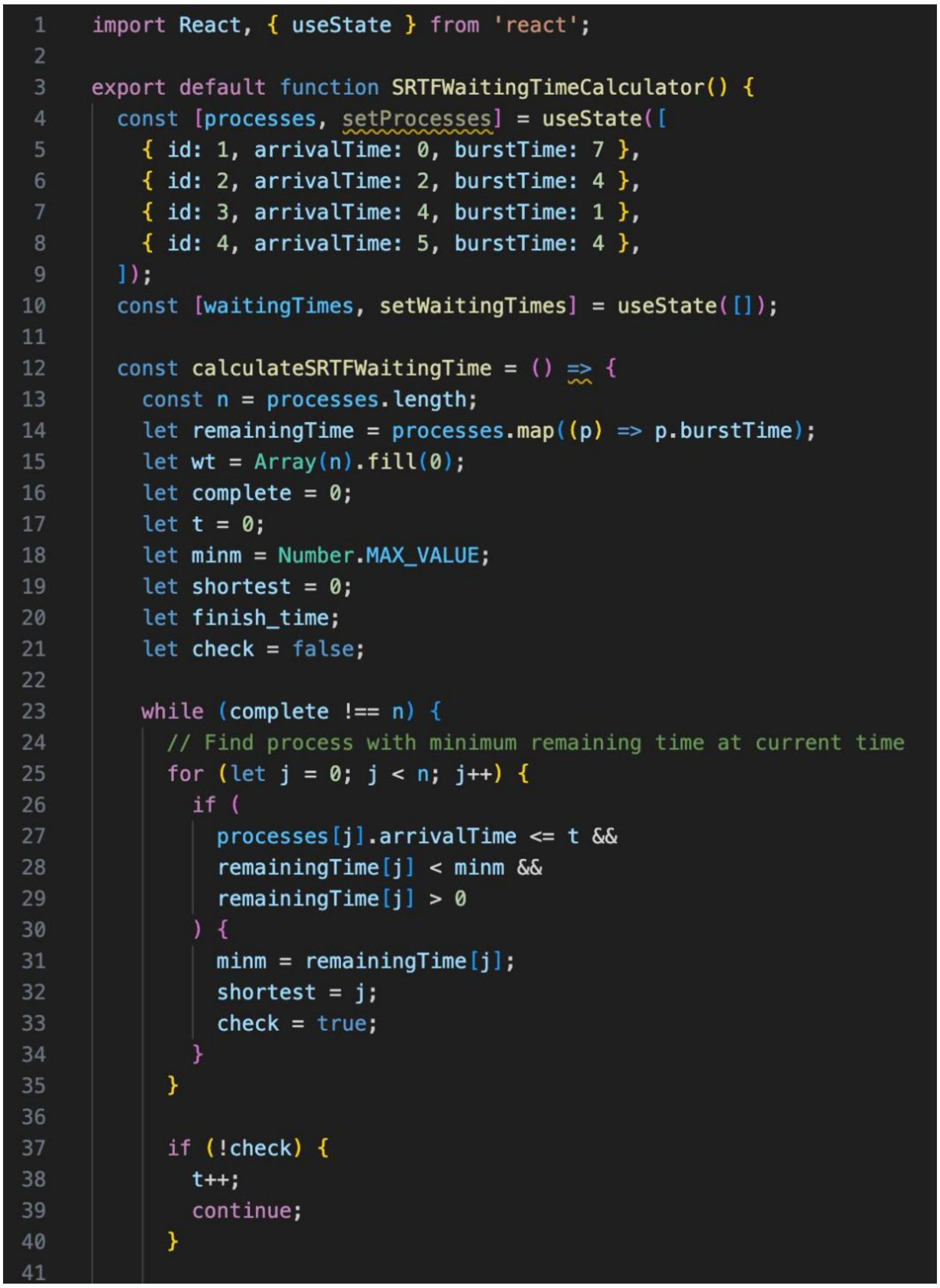
<u>How this works:</u>

Step 1: Sort processes by burst time.

Step 2: Waiting time for each process = waiting time of previous + burst time of previous.

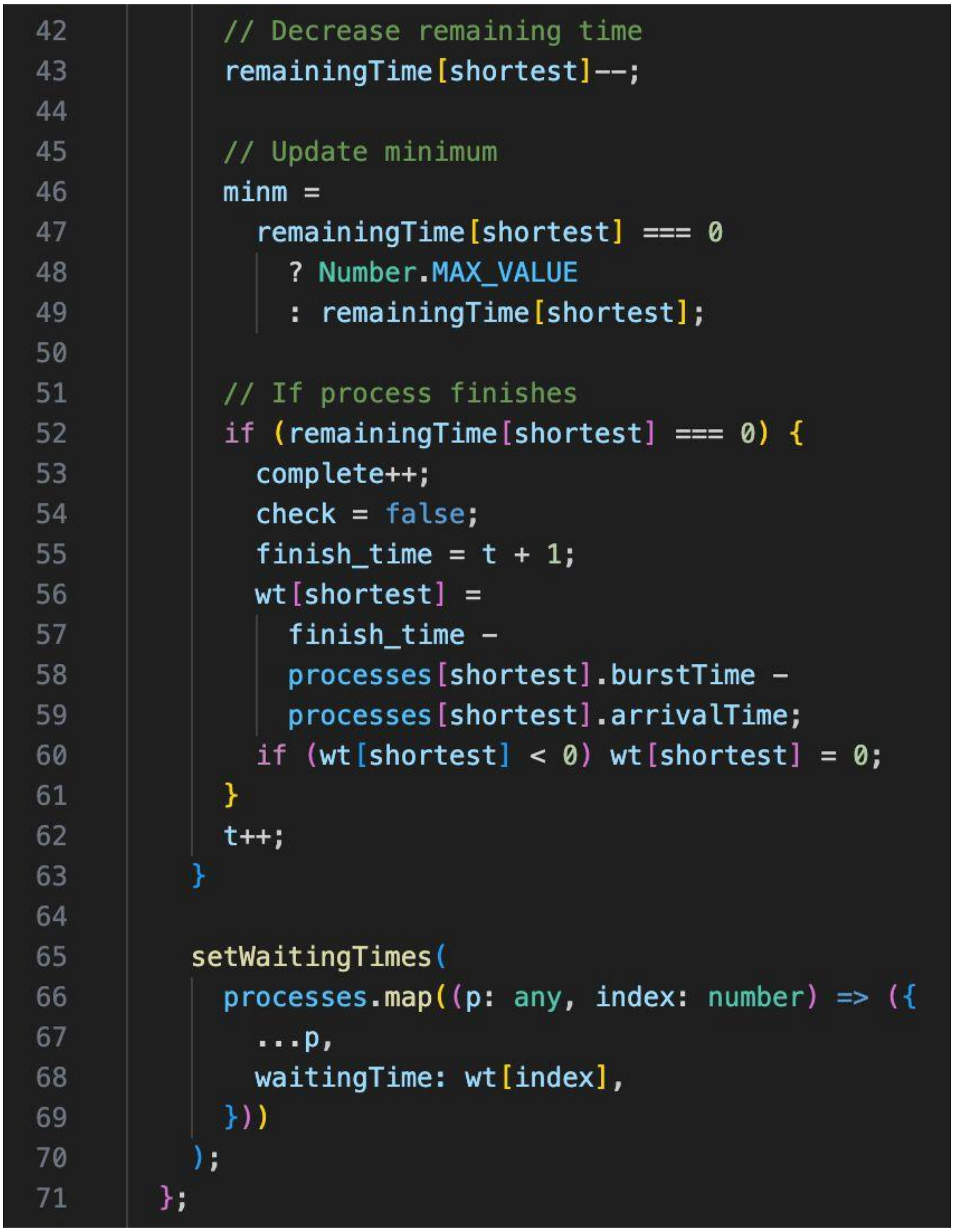
Step 3: Display results in SJF order.

Figure 5.4: Preemptive Shortest Job First (Shortest Remaining Time First – SRTF) with arrival times. This will calculate waiting time for each process.



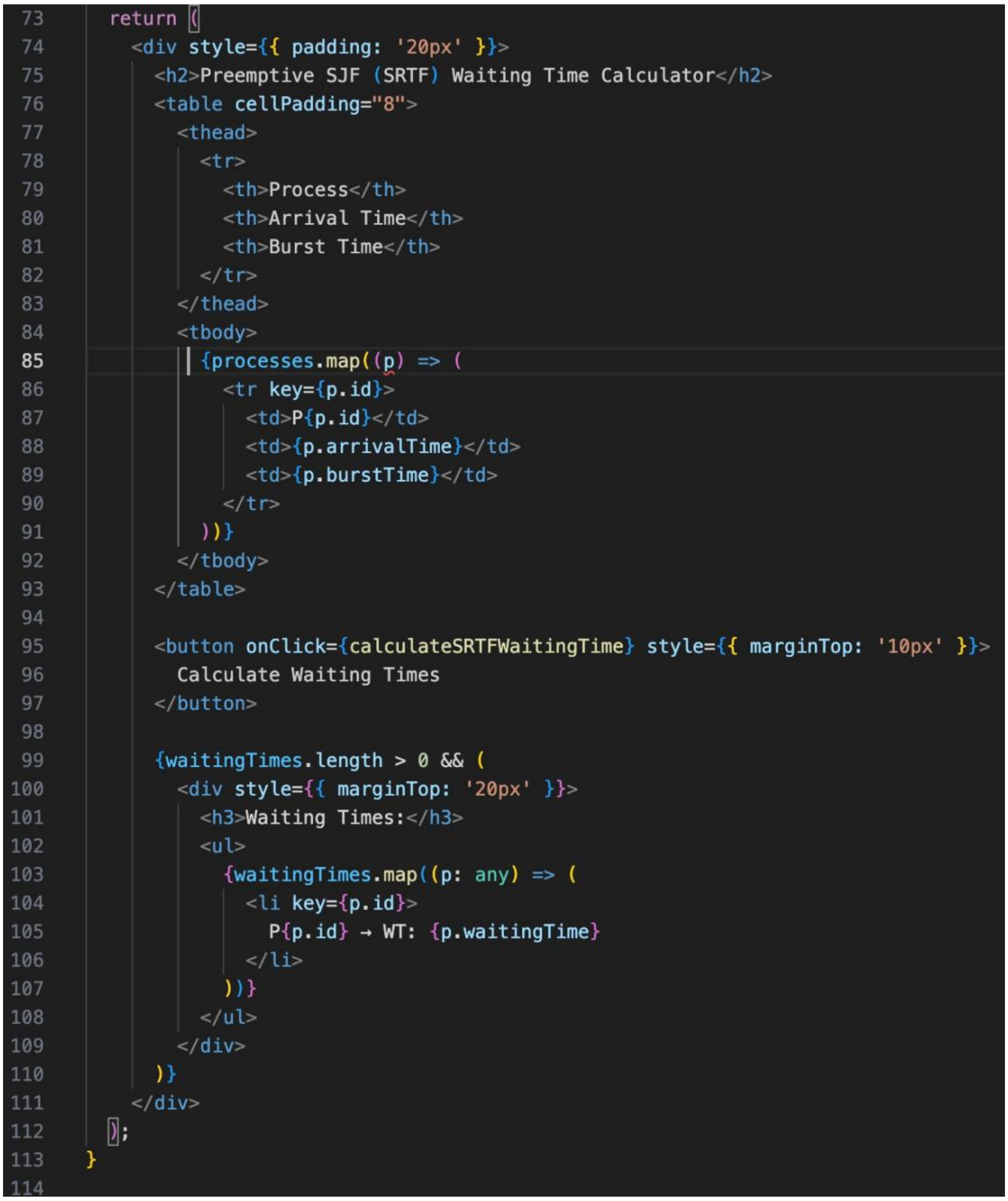
```
import React, { useState } from 'react';

export default function SRTFWaitingTimeCalculator() {
  const [processes, setProcesses] = useState([
    { id: 1, arrivalTime: 0, burstTime: 7 },
    { id: 2, arrivalTime: 2, burstTime: 4 },
    { id: 3, arrivalTime: 4, burstTime: 1 },
    { id: 4, arrivalTime: 5, burstTime: 4 },
  ]);
  const [waitingTimes, setWaitingTimes] = useState([]);

  const calculateSRTFWaitingTime = () => {
    const n = processes.length;
    let remainingTime = processes.map((p) => p.burstTime);
    let wt = Array(n).fill(0);
    let complete = 0;
    let t = 0;
    let minm = Number.MAX_VALUE;
    let shortest = 0;
    let finish_time;
    let check = false;

    while (complete !== n) {
      // Find process with minimum remaining time at current time
      for (let j = 0; j < n; j++) {
        if (
          processes[j].arrivalTime <= t &&
          remainingTime[j] < minm &&
          remainingTime[j] > 0
        ) {
          minm = remainingTime[j];
          shortest = j;
          check = true;
        }
      }

      if (!check) {
        t++;
        continue;
      }

```

Figure 5.4(a)



```
      // Decrease remaining time
      remainingTime[shortest]--;

      // Update minimum
      minm =
        remainingTime[shortest] === 0
          ? Number.MAX_VALUE
          : remainingTime[shortest];

      // If process finishes
      if (remainingTime[shortest] === 0) {
        complete++;
        check = false;
        finish_time = t + 1;
        wt[shortest] =
          finish_time -
          processes[shortest].burstTime -
          processes[shortest].arrivalTime;
        if (wt[shortest] < 0) wt[shortest] = 0;
      }
      t++;
    }

    setWaitingTimes(
      processes.map((p: any, index: number) => ({
        ...p,
        waitingTime: wt[index],
      }))
    );
  };
```

Figure 5.4(b)



```
  return (
    <div style={{ padding: '20px' }}>
      <h2>Preemptive SJF (SRTF) Waiting Time Calculator</h2>
      <table cellPadding="8">
        <thead>
          <tr>
            <th>Process</th>
            <th>Arrival Time</th>
            <th>Burst Time</th>
          </tr>
        </thead>
        <tbody>
          {processes.map((p) => (
            <tr key={p.id}>
              <td>P{p.id}</td>
              <td>{p.arrivalTime}</td>
              <td>{p.burstTime}</td>
            </tr>
          ))}
        </tbody>
      </table>

      <button onClick={calculateSRTFWaitingTime} style={{ marginTop: '10px' }}>
        Calculate Waiting Times
      </button>

      {waitingTimes.length > 0 && (
        <div style={{ marginTop: '20px' }}>
          <h3>Waiting Times:</h3>
          <ul>
            {waitingTimes.map((p: any) => (
              <li key={p.id}>
                P{p.id} → WT: {p.waitingTime}
              </li>
            ))}
          </ul>
        </div>
      )}
    </div>
  );
}
```

Figure 5.4(c)

<u>How it works:</u>

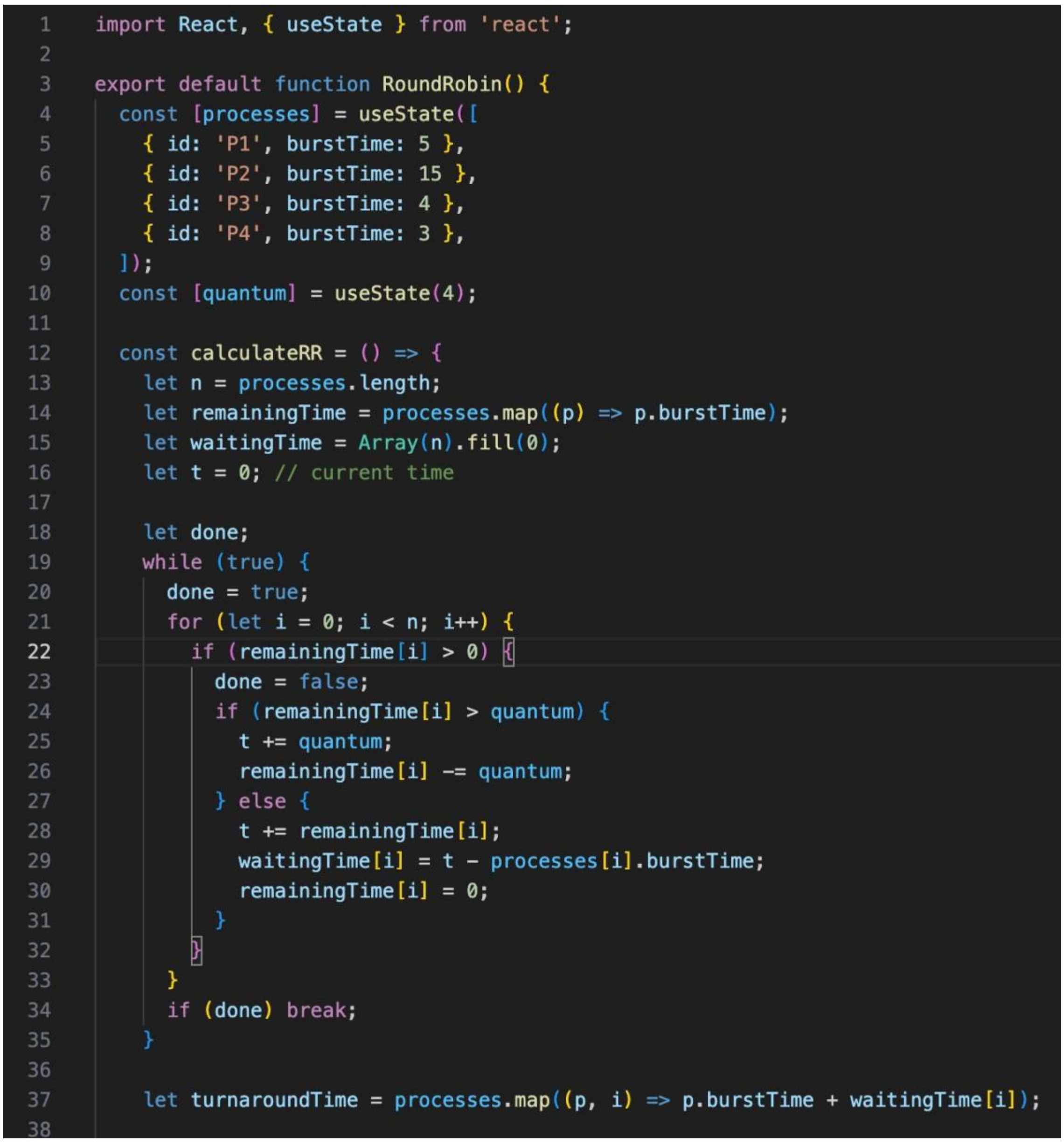
Simulates time step-by-step, picking the process with the shortest remaining burst time that has already arrived.

Preempts the current process if a shorter job arrives.

Calculates Waiting Time as:

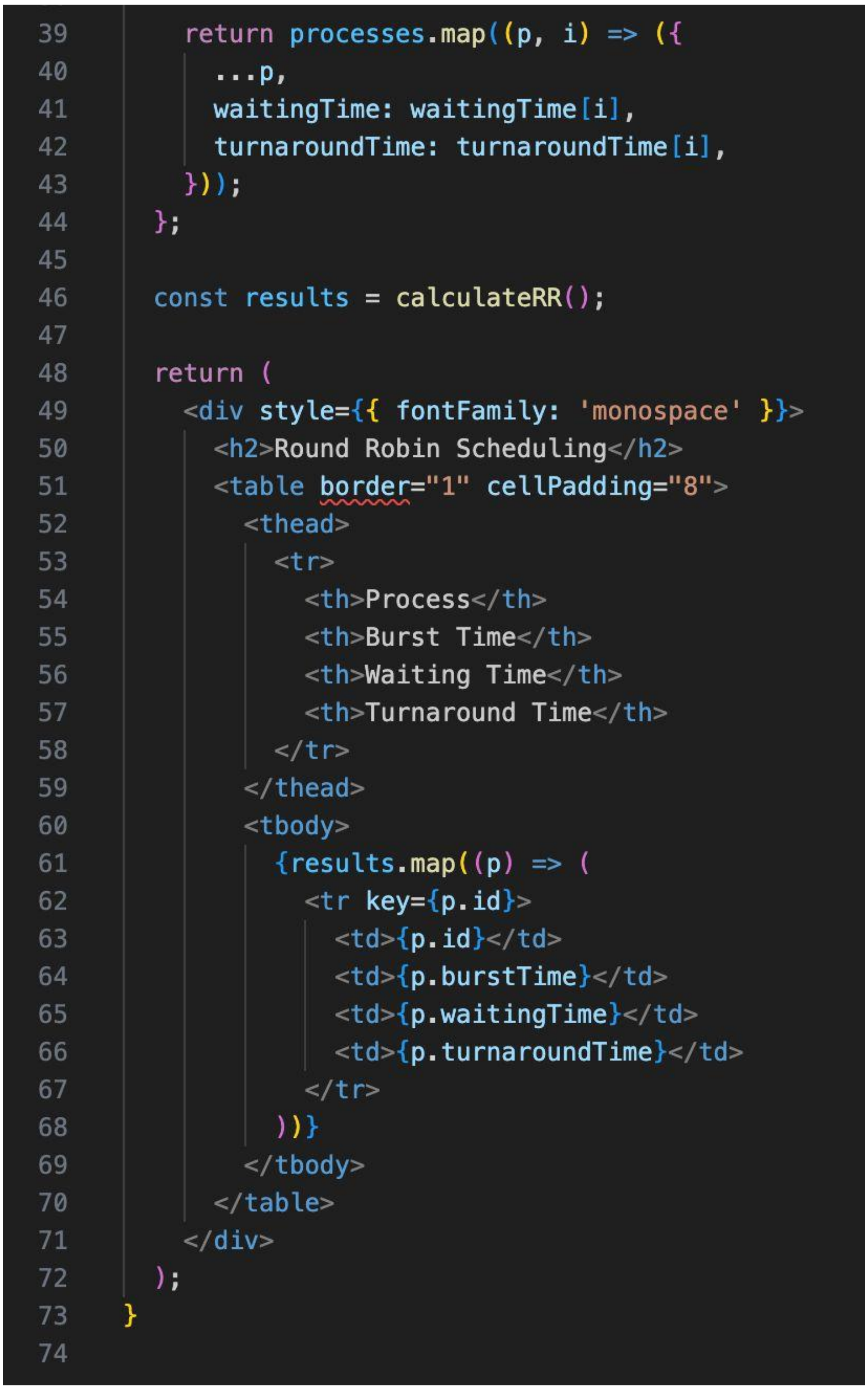
WT = Finish Time - Burst Time - Arrival Time

Figure 5.5: Round Robin



```
import React, { useState } from 'react';

export default function RoundRobin() {
  const [processes] = useState([
    { id: 'P1', burstTime: 5 },
    { id: 'P2', burstTime: 15 },
    { id: 'P3', burstTime: 4 },
    { id: 'P4', burstTime: 3 },
  ]);
  const [quantum] = useState(4);

  const calculateRR = () => {
    let n = processes.length;
    let remainingTime = processes.map((p) => p.burstTime);
    let waitingTime = Array(n).fill(0);
    let t = 0; // current time

    let done;
    while (true) {
      done = true;
      for (let i = 0; i < n; i++) {
        if (remainingTime[i] > 0) {
          done = false;
          if (remainingTime[i] > quantum) {
            t += quantum;
            remainingTime[i] -= quantum;
          } else {
            t += remainingTime[i];
            waitingTime[i] = t - processes[i].burstTime;
            remainingTime[i] = 0;
          }
        }
      }
      if (done) break;
    }

    let turnaroundTime = processes.map((p, i) => p.burstTime + waitingTime[i]);

```

Figure 5.5(a)



```
    return processes.map((p, i) => ({
      ...p,
      waitingTime: waitingTime[i],
      turnaroundTime: turnaroundTime[i],
    }));
  };

  const results = calculateRR();

  return (
    <div style={{ fontFamily: 'monospace' }}>
      <h2>Round Robin Scheduling</h2>
      <table border="1" cellPadding="8">
        <thead>
          <tr>
            <th>Process</th>
            <th>Burst Time</th>
            <th>Waiting Time</th>
            <th>Turnaround Time</th>
          </tr>
        </thead>
        <tbody>
          {results.map((p) => (
            <tr key={p.id}>
              <td>{p.id}</td>
              <td>{p.burstTime}</td>
              <td>{p.waitingTime}</td>
              <td>{p.turnaroundTime}</td>
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );
}
```

Figure 5.5(b)

How it works

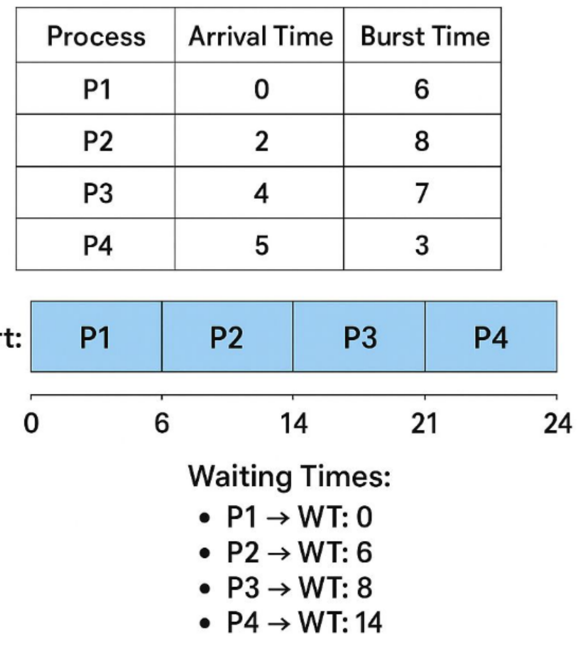
Uses time slicing based on the quantum.

Keeps track of remaining times until all are zero.

Waiting time = finish time − burst time.

Turnaround time = burst time + waiting time.

5.5 Some Gantt Charts

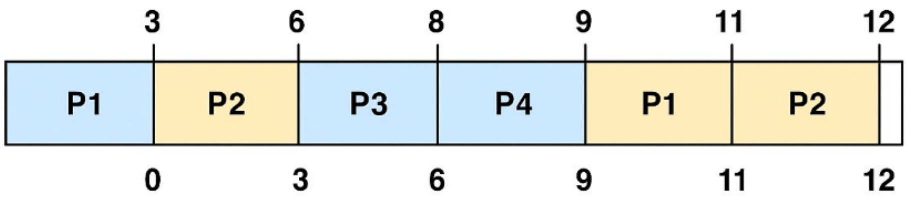


Process	Arrival Time	Burst Time
P1	0	6
P2	2	8
P3	4	7
P4	5	3

Waiting Times:

- P1 → WT: 0
- P2 → WT: 6
- P3 → WT: 8
- P4 → WT: 14

Figure 5.6: Non-preemptive SJF



Process	Arrival Time	Completion Time	Turnaround Time (CT-	Waiting Time (BT)
P1	0	5	11	6
P2	1	4	12	7
P3	2	2	8	4
P4	4	1	5	4

Figure 5.7: Round Robin

Chapter 6
TESTING

6.1 Introduction

Testing is an essential phase in the software development life cycle to ensure that the implemented system meets the specified requirements and functions correctly. The CPU Scheduling Simulator was tested to identify and remove bugs, verify functionality, and validate performance.

6.2 Testing Objectives

- To verify that all modules work as intended.
- To ensure accurate calculation of waiting time and turnaround time.
- To check that the Gantt chart is generated correctly for all scheduling algorithms.
- To confirm that the system handles invalid inputs gracefully.

6.3 Types of testing Performed

Type of Testing	Description
Unit Testing	Testing of individual functions such as waiting time calculation, sorting of processes, and Gantt chart rendering.
Integration Testing	Ensuring that the Input, Algorithm Selection, and Output modules work together without issues.
System Testing	Verifying that the overall application meets functional and non-functional requirements.
UI Testing	Checking responsiveness, layout consistency, and clarity of displayed results.
Validation Testing	Comparing the program's results with manually calculated outputs for correctness.

Table 6.1: Types of testing Performed

6.4 Test Cases

Test Case Number	Description	Input	Expected Output	Actual Output	Result
1	Valid Process Details with FCFS	P1(5), P2(3)	Correct Gantt chart & times	Correct Gantt chart & times	Pass

2	Invalid burst time (negative)	P1(-4)	Error message	Error message	Pass
3	Multiple processes with SJF	P1(6), P2(2), P3(4)	Correct sequence & times	Correct sequence & times	Pass
4	Round Robin with quantum 2	Processes & quantum=2	Correct time slices & results	Correct time slices & results	Pass

Table 6.2: Test Cases

6.5 Testing Tools

- Browser Developer Tools – For debugging and performance analysis.
- React Developer Tools – To monitor component state and props.
- Manual Calculation – For validating CPU scheduling outputs.

6.6 Conclusion

The testing phase confirmed that the CPU Scheduling Simulator functions as expected across different scenarios. All major functionalities were verified, and no critical defects remain.

Final Chapter

CONCLUSION AND FUTURE SCOPE

7.1 Conclusion

The CPU Scheduling Simulator was successfully developed to demonstrate and compare various scheduling algorithms, including **First Come First Serve (FCFS)**, **Shortest Job First (SJF)**, **Round Robin (RR)**, and **Priority Scheduling**.

The simulator provides a clear visualization of process execution and calculates essential metrics such as **Waiting Time** and **Turnaround Time** for each algorithm. Through simulation, users can observe the strengths and weaknesses of each algorithm under different conditions, thereby gaining a deeper understanding of CPU scheduling concepts.

The project meets its objectives by:

- Implementing multiple CPU scheduling algorithms.
- Providing an interactive and user-friendly interface.
- Displaying accurate performance metrics.

This work can be a valuable educational tool for students and professionals learning about operating systems.

7.2 Future Scope

While the current version of the CPU Scheduling Simulator fulfils its primary goals, several enhancements can be incorporated to make it more advanced and interactive:

- Addition of More Algorithms – Include algorithms like Multilevel Queue Scheduling, Multilevel Feedback Queue, and Earliest Deadline First (EDF).
- Gantt Chart Visualization – Provide graphical Gantt charts to represent the scheduling order visually.
- Real-Time Simulation – Allow the simulator to run with real-time inputs for more practical demonstrations.
- Performance Comparison Graphs – Display bar/line charts comparing average waiting times and turnaround times for different algorithms.
- Web or Mobile Version – Extend the project into a web-based or mobile application for wider accessibility.
- Dynamic Process Arrival – Allow processes to arrive at different times during simulation to mimic real CPU scheduling scenarios.
- With these enhancements, the simulator can evolve into a comprehensive platform for studying and analysing CPU scheduling in depth.

7.3 Final Remarks

In conclusion, the CPU Scheduling Simulator successfully bridges the gap between theoretical understanding and practical application of scheduling algorithms. It not only serves as a valuable educational resource but also lays the foundation for future innovations in CPU scheduling simulations. With further development, the simulator can transform into a robust, versatile, and widely accessible tool for both learning and research purposes.

REFERENCES

- E. Balagurusamy, Object Oriented Programming with C++ (8th Edition), McGraw-Hill Education, 2020, pp. 1–780.
- R. Chopra, Database Management System (2nd Edition), S. Chand Publishing, 2019, pp. 10–450.
- J. Doe, CPU Scheduling Algorithms: Comparative Analysis, International Journal of Computer Applications, Vol. 175, No. 12, 2020, pp. 15–28.
- https://react.dev (Accessed on 1st Aug 2025).
- https://developer.mozilla.org (Accessed on 1st Aug 2025).