

**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

**CPU Scheduling in Operating System**

**A CAPSTONE PROJECT REPORT**

*Submitted in the partial fulfillment for the award of the degree of*

**BACHELOR OF ENGINEERING**

**IN**

**Computer Science Engineering**

**Submitted by**

**A. Divyapriya (192210454)**

**Sai Teja Sree K (192210206)**

**S. Grishma Chandu (192210566)**

**Under the Supervision of**

**Dr. Terrance Frederick Fernandez**

**July 2024**

**DECLARATION**

We, **A. Divyapriya ,K. Sai Teja Sree, S. Grishma Chandu** students of **Bachelor of Engineering in Computer Science Engineering**, Department of Computer Science and Engineering, Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work is **CPU Scheduling in Operating System** is the outcome of our own bonafide work and is correct to the best of our knowledge and this work has been undertaken taking care of Engineering Ethics.

A. Divyapriya (192210454)

K. Sai Teja Sree (192210206)

S. Grishma Chandu (192210566)

Date:

Place:

**CERTIFICATE**

This is to certify that the project entitled **CPU Scheduling in Operating System**  submitted by **A. Divyapriya, K. Sai Teja Sree, S. Grishma Chandu** has been carried out under our supervision. The project has been submitted as per the requirements in the current semester of B.E. Computer Science Engineering.

Faculty-in-charge

**Dr. Terrance Frederick Fernandez**

**TABLE OF CONTENTS**

| **S.NO** | **TOPICS** |
| --- | --- |
| 1 | **Abstract** |
| 2 | **Introduction** |
| 3 | **Problem Statement** |
| 4 | **Proposed Design Work**  1. Identifying the key components  2. Functionality  3. Architectural Design |
| 5. | **UI Design**   1. **Layout Design**   Flexible Layout  User friendly  Color selection  **2.Feasible elements used**  Elements positioning  Accessibility  **3.Elements Function.** |
| 6. | **Code:**  Main process  output |
| 7. | **Future trends** |
| 8. | **Conclusion** |

**ABSTRACT**

CPU scheduling is a crucial aspect of operating system (OS) design, responsible for managing how the CPU's time is allocated to various processes. Efficient CPU scheduling aims to maximize CPU utilization and system responsiveness while ensuring fairness among processes. It involves selecting which process in the ready queue should be executed next by the CPU, a decision influenced by various factors such as process priority, execution time, and arrival time. There are several CPU scheduling algorithms, each with its own strengths and weaknesses. Common approaches include First-Come, First-Served (FCFS), Shortest Job Next (SJN), Round Robin (RR), and Priority Scheduling. FCFS is simple but can lead to the "convoy effect," where short processes wait for long processes to complete. SJN optimizes for short processes but may cause starvation for longer tasks. RR provides a time-sharing environment, ensuring that all processes receive attention, but its performance heavily depends on the chosen time quantum. Priority Scheduling assigns importance levels to processes but can lead to low-priority processes being starved. Effective CPU scheduling enhances overall system performance by reducing process waiting time, improving throughput, and ensuring a responsive user experience. It is a fundamental component of OS efficiency, directly impacting how well an operating system can manage multitasking and resource allocation.

**INTRODUCTION**

CPU scheduling is a fundamental function of operating systems, determining the order in which processes access the CPU to optimize performance and efficiency. CPU utilization and system responsiveness while ensuring fairness among processes. It involves selecting which process in the ready queue should be executed next by the CPU, a decision influenced by various factors such as process priority, execution time, and arrival time. It involves various algorithms designed to balance factors such as process priority, execution time, and arrival time. Key algorithms include First-Come, First-Served (FCFS), Shortest Job Next (SJN), Round Robin (RR), and Priority Scheduling, each with unique advantages and challenges. FCFS is straightforward but can cause inefficiencies for short processes, while SJN minimizes average waiting time but risks process starvation. RR ensures time-sharing but is sensitive to the time quantum size, and Priority Scheduling manages process importance but can also lead to starvation. Effective CPU scheduling aims to enhance system throughput, reduce process waiting times, and maintain a responsive user experience. This balancing act is critical for optimizing multitasking and resource management in modern operating systems.

**PROBLEM STATEMENT**

CPU scheduling is a fundamental component of operating systems (OS), crucial for managing the execution of processes and optimizing the use of CPU resources. However, traditional CPU scheduling algorithms face significant challenges that impact overall system performance, responsiveness, and fairness.

**Efficiency:** Many conventional scheduling algorithms, such as First-Come, First-Served (FCFS) and Shortest Job Next (SJN), struggle to efficiently manage processes of varying execution times, leading to issues like the "convoy effect" and process starvation.

**Fairness:** Ensuring that all processes receive equitable CPU time remains a challenge, particularly in Priority Scheduling, where lower-priority processes may be neglected.

**Scalability**: Modern computing environments, characterized by a high volume of processes and diverse workloads, demand scalable scheduling solutions that can adapt to changing conditions and maintain performance.

**Real-Time Processing**: In real-time systems, meeting strict deadlines is essential. Traditional scheduling algorithms often fail to guarantee timely execution of critical tasks, compromising system reliability.

**Performance Overhead:** The computational overhead associated with advanced scheduling algorithms can negatively impact system performance, particularly in resource-constrained environments.

**Adaptability:** Dynamic and unpredictable workloads require adaptive scheduling strategies that can respond to varying process requirements and system states in real-time.

Innovative CPU scheduling techniques that balance efficiency, fairness, and scalability are essential to address these challenges. This study aims to explore and evaluate novel scheduling algorithms, assessing their impact on system performance, responsiveness, and fairness. By investigating these solutions, the study seeks to identify practical approaches for enhancing CPU scheduling in contemporary operating systems.

**PROPOSED DESIGN WORK**

**1. Identifying Key Components**

**Dynamic Priority Adjustment:** Implementing a dynamic priority adjustment mechanism that continually assesses process behavior and system load, adjusting priorities to optimize CPU allocation and prevent starvation.

**Hybrid Scheduling Algorithms:** Combining multiple scheduling algorithms (e.g., FCFS, SJN, RR) into a hybrid approach that adapts based on current system conditions and process characteristics, leveraging the strengths of each method.

**Real-Time Scheduling Enhancements**: Incorporating real-time scheduling techniques to ensure that critical processes meet their deadlines, enhancing the reliability of systems requiring timely execution.

**Load Balancing:** Introducing load balancing strategies that distribute processes across multiple CPUs or cores, improving overall system efficiency and reducing bottlenecks.

**Predictive Scheduling:** Utilizing machine learning models to predict process behavior and optimize scheduling decisions, anticipating future resource needs and adjusting allocations proactively.

**Fairness Mechanisms:** Ensuring fairness by integrating mechanisms that monitor process wait times and adjust scheduling policies to provide equitable CPU access for all processes.

**2. Functionality**

**Dynamic Priority Adjustment:** Continuously monitor and adjust process priorities based on execution patterns, system load, and performance metrics, preventing issues like process starvation and ensuring balanced resource allocation.

**Hybrid Scheduling Algorithms**: Implement an adaptive scheduling system that switches between FCFS, SJN, RR, and other algorithms based on real-time system analysis, optimizing CPU utilization and reducing process wait times.

**Real-Time Scheduling Enhancements**: Incorporate real-time scheduling techniques such as Earliest Deadline First (EDF) to ensure critical tasks meet their deadlines, improving system reliability for time-sensitive applications.

**Load Balancing:** Distribute processes across available CPUs or cores using load balancing algorithms, enhancing system performance by avoiding CPU overloading and ensuring efficient utilization of resources.

**Predictive Scheduling:** Apply machine learning models to predict future process behavior, allowing the scheduler to make informed decisions that anticipate and respond to workload changes.

**Fairness Mechanisms**: Implement fairness mechanisms that dynamically adjust scheduling policies based on process wait times and resource usage, ensuring equitable access to CPU resources for all processes.

**3. Architectural Design**

**Dynamic Priority Adjustment Module:** Design a module that continuously monitors system load and process behavior, adjusting priorities in real-time to optimize CPU scheduling and prevent process starvation.

**Hybrid Scheduling Framework:** Develop a flexible scheduling framework capable of switching between different algorithms (FCFS, SJN, RR) based on real-time analysis of system conditions, leveraging the strengths of each approach.

**Real-Time Scheduling Component:** Integrate a component dedicated to real-time scheduling, employing techniques like EDF to ensure critical tasks meet their deadlines, enhancing system reliability for time-sensitive applications.

**Load Balancing Engine**: Implement a load balancing engine that distributes processes across multiple CPUs or cores, using algorithms to optimize resource utilization and prevent bottlenecks.

**Predictive Scheduling System:** Develop a predictive scheduling system that uses machine learning models to forecast process behavior, enabling proactive scheduling decisions that adapt to changing workloads.

**Fairness Management Module:** Create a fairness management module that monitors process wait times and resource usage, dynamically adjusting scheduling policies to ensure equitable CPU access for all processes, preventing starvation and ensuring balanced resource allocation.

**UI DESIGN**

**1. Layout Design**

**Flexible Layout:**

**Process List Section:**

**Description:** A comprehensive table displaying all processes with attributes such as PID, process name, priority, CPU burst time, arrival time, and status.

**Justification:** Provides an at-a-glance overview of all processes, essential for efficient scheduling.

**Potential Benefits:** Improves user ability to monitor and manage processes.

**Challenges:** Ensuring real-time updates without performance degradation.

**Gantt Chart Section:**

**Description**: A dynamic Gantt chart visualizing the CPU schedule, showing process execution over time.

**Justification:** Helps users understand the scheduling order and CPU utilization visually.

**Potential Benefits:** Enhances comprehension of process scheduling and timing.

**Challenges:** Keeping the chart accurate and responsive in real-time.

**Control Panel Section:**

**Description:** Controls for starting, stopping, and adjusting the scheduling algorithm, such as buttons or dropdown menus to select algorithms like FCFS, SJF, RR, etc.

**Justification:** Allows users to interact with and modify scheduling policies easily.

**Potential Benefits:** Provides flexibility in scheduling policy management.

**Challenges:** Ensuring changes are applied seamlessly without interrupting ongoing processes.

**Statistics Section:**

**Description:** Displays statistics like CPU utilization, throughput, average waiting time, and turnaround time.

**Justification**: Provides critical performance metrics to evaluate scheduling effectiveness.

**Potential Benefits:** Aids in performance analysis and optimization.

**Challenges**: Ensuring metrics are calculated accurately and displayed promptly.

**User Friendly:**

**Intuitive Navigation:**

**Description**: Easy-to-use navigation with clear labels and tooltips for each section.

**Justification:** Enhances user experience and reduces the learning curve.

**Potential Benefits**: Encourages more frequent use and exploration of features**.**

**Challenges**: Balancing simplicity with the availability of advanced features.

**Real-time Updates:**

**Description:** Automatic updates of the process list and Gantt chart to reflect the current state of the CPU scheduler.

**Justification:** Ensures users have the most current information.

**Potential Benefits:** Improves decision-making and response times.

**Challenges**: Minimizing the impact on system performance.

**Interactive Elements:**

**Description**: Draggable processes in the Gantt chart to manually adjust scheduling (for learning and testing purposes).

**Justification:** Provides hands-on experience with scheduling concepts.

**Potential Benefits**: Enhances understanding and engagement.

**Challenges:** Ensuring interactions are smooth and do not disrupt the underlying scheduling logic.

**Color Selection:**

**Process States:**

**Description:** Use distinct colors to represent different process states (e.g., running, waiting, blocked).

**Justification:** Makes it easy to differentiate between process states at a glance.

**Potential Benefits:** Improves clarity and reduces cognitive load.

**Challenges:** Choosing colors that are distinguishable by all users, including those with color vision deficiencies.

**Algorithm Visualization:**

**Description:** Different colors for each scheduling algorithm in the control panel and Gantt chart.

**Justification:** Helps users quickly identify the current scheduling policy in use.

**Potential Benefits:** Enhances visual organization and user understanding.

**Challenges**: Maintaining consistency in color usage across different UI elements.

**Highlighting Critical Information:**

**Description:** Use bright or contrasting colors to highlight critical information such as high-priority processes or alerts.

**Justification:** Ensures important information stands out.

**Potential Benefits:** Draws user attention to urgent tasks and status updates.

**Challenges:** Avoiding overuse of highlighting to prevent visual clutter.

**Implementation and Maintenance Plan**

**Implementation:**

**Phase 1:** Develop the basic layout and integrate real-time process tracking.

**Phase 2**: Implement the Gantt chart visualization and interactive controls.

**Phase 3**: Add detailed statistics and performance metrics.

**Phase 4:** Test for usability and performance, gather user feedback, and make necessary adjustments.

**Maintenance:**

**Regular Updates**: Ensure the software is updated to handle new scheduling algorithms and optimizations.

**User Feedback:** Continuously gather and incorporate user feedback to improve the UI and functionality.

**Performance Monitoring:** Regularly monitor system performance to ensure real-time updates do not degrade overall system efficiency.

By following this design and plan, the UI for CPU scheduling in an OS will be both functional and user-friendly, providing valuable insights and control to the users.

**CODE**

**Main process:**

import java.awt.Color;

import java.util.concurrent.TimeUnit;

import javax.swing.\*;

public class CPUScheduler extends Thread {

    Job[] jobBatch;

    Scheduler policy;

    JTextArea textArea;

    JTextField textField;

    JProgressBar[] pbars;

    JLabel[] burstTimes;

  ComputationThread[]myThreads=newComputationThread[SchedulingGUI.NUM\_OF\_PROCESSES];

    JLabel[] waitingTimes, priorities;

    static int statusSum = 0;

    public CPUScheduler(Job[] jobBatch, Scheduler policy, JTextArea textArea, JTextField textField, JProgressBar[] pbars,

            JLabel[] burstTimes, JLabel[] waitingTimes, JLabel[] priorities) {

        this.jobBatch = jobBatch;

        this.policy = policy;

        this.textArea = textArea;

        this.textField = textField;

        this.pbars = pbars;

        this.burstTimes = burstTimes;

        this.waitingTimes = waitingTimes;

        this.priorities = priorities;

    }

    public void run() {

        if(CalcSimulation.algo.equals("FCFS")) {

            Job arrivedJob;

            for(int i = 0; i < jobBatch.length; i++) {

                arrivedJob = jobBatch[i];

                policy.enqueue(arrivedJob);

            }

            int i = 0;

            while(!policy.isEmpty()) {

                JProgressBar pbar = pbars[i];

                JLabel burstTime = burstTimes[i];

                long arrivalTime = policy.peek().job.arrivalTime;

                //pbar.setBackground(Color.blue);

                try {

                    Thread.sleep(arrivalTime);

                } catch(Exception e) {}

                Job newJob = policy.dequeue();

                int durationInS = (int) TimeUnit.NANOSECONDS.toSeconds(newJob.waitTime);

                waitingTimes[i].setText(String.valueOf(durationInS) + "s");

                myThreads[i] = new ComputationThread(newJob, policy, textArea, textField, pbar, burstTime);

                myThreads[i].t.start();

                i++;

           }

        } else if(CalcSimulation.algo.equals("Round Robin")) {

            for(int i = 0; i < jobBatch.length; i++) {

                jobBatch[i].progressBar = pbars[i];

                jobBatch[i].burstTimeLabel = burstTimes[i];

                jobBatch[i].waitTimeLabel = waitingTimes[i];

                policy.enqueue(jobBatch[i]);

            }

            int i = 0;

            while(!policy.isEmpty()) {

                long arrivalTime = policy.peek().job.arrivalTime;

                try {

                    Thread.sleep(arrivalTime);

                } catch(Exception e) {}

                Job newJob = policy.dequeue();

                int durationInS = (int) TimeUnit.NANOSECONDS.toSeconds(newJob.waitTime);

                newJob.waitTimeLabel.setText(String.valueOf(durationInS) + "s");

                myThreads[0] = new ComputationThread(newJob, policy, textArea, textField,

                        newJob.progressBar, newJob.burstTimeLabel);

                myThreads[0].t.start();

                try {

                    myThreads[0].t.join();

                } catch(Exception ex) {}

            }

            for(int j = 1; j < myThreads.length; j++)

                myThreads[j] = new ComputationThread(null, null, null, null, null, null);

        }

        else if (CalcSimulation.algo.equals("Priority Scheduling")) {

            MaxPriorityQueue mp = new MaxPriorityQueue();

            MaxPriorityQueue tempmp = new MaxPriorityQueue();

            for(int i = 0; i < jobBatch.length; i++) {

                int priority = (new java.util.Random().nextInt(10) + 1);

                JProgressBar pbar = pbars[i];

                JLabel burstTime = burstTimes[i];

                jobBatch[i].progressBar = pbar;

                jobBatch[i].burstTimeLabel = burstTime;

                jobBatch[i].priority = priority;

                jobBatch[i].waitTimeLabel = waitingTimes[i];

                priorities[i].setText(String.valueOf(jobBatch[i].priority));

                mp.insert(jobBatch[i]);

                tempmp.insert(jobBatch[i]);

            }

            while(!tempmp.isEmpty()) {

                policy.enqueue(tempmp.extractMax());

            }

            while(!mp.isEmpty()) {

                long arrivalTime = mp.getMax().arrivalTime;

                try {

                    Thread.sleep(arrivalTime);

                } catch(Exception e) {}

                Job newJob = mp.extractMax();

                policy.dequeue();

                newJob.waitTime = System.nanoTime() - newJob.startTime;

                int durationInS = (int) TimeUnit.NANOSECONDS.toSeconds(newJob.waitTime);

                newJob.waitTimeLabel.setText(String.valueOf(durationInS) + "s");

                ComputationThread cpu = new ComputationThread(newJob, policy, textArea,

                        textField, newJob.progressBar, newJob.burstTimeLabel);

                cpu.t.start();

                try {

                    cpu.t.join();

                } catch(InterruptedException ex) {}

           }

        } else {

            MaxPriorityQueue tempmp = new MaxPriorityQueue();

            for(int i = 0; i < jobBatch.length; i++) {

                int priority = (int) jobBatch[i].burstTime;

                JProgressBar pbar = pbars[i];

                JLabel burstTime = burstTimes[i];

                jobBatch[i].progressBar = pbar;

                jobBatch[i].burstTimeLabel = burstTime;

                jobBatch[i].priority = priority;

                jobBatch[i].waitTimeLabel = waitingTimes[i];

                priorities[i].setText(String.valueOf(jobBatch[i].priority));

                tempmp.insert(jobBatch[i]);

            }

            while(!tempmp.isEmpty()) {

                policy.enqueue(tempmp.extractMax());

            }

            while(!policy.isEmpty()) {

                long arrivalTime = policy.peek().job.arrivalTime;

                try {

                    Thread.sleep(arrivalTime);

                } catch(Exception e) {}

                Job newJob = policy.dequeue();

                newJob.waitTime = System.nanoTime() - newJob.startTime;

                int durationInS = (int) TimeUnit.NANOSECONDS.toSeconds(newJob.waitTime);

                newJob.waitTimeLabel.setText(String.valueOf(durationInS) + "s");

                ComputationThread cpu = new ComputationThread(newJob, policy, textArea,

                        textField, newJob.progressBar, newJob.burstTimeLabel);

                cpu.t.start();

                try {

                    cpu.t.join();

                } catch(InterruptedException ex) {}

           }

        }

        System.out.println("GOT OUT");

        if(!CalcSimulation.algo.equals("Priority Scheduling") &&

                !CalcSimulation.algo.equals("Shortest Job First")) {

            try {

                for(int j = 0; j < myThreads.length; j++)

                    myThreads[j].t.join();

            } catch(InterruptedException ex) {}

        }

        textField.setText("Idle");

        textField.setForeground(Color.red);

        long avgWaitTime = 0;

        long avgTurnaroundTime = 0;

        long totalExecutionTime = 0;

        for(int j = 0; j < jobBatch.length; j++) {

            avgWaitTime += jobBatch[j].waitTime;

            avgTurnaroundTime += jobBatch[j].endTime;

        }

        avgWaitTime /= jobBatch.length;

        avgWaitTime = TimeUnit.NANOSECONDS.toSeconds(avgWaitTime);

        CalcSimulation.avgWaitField.setText(String.valueOf(avgWaitTime) + "s");

avgTurnaroundTime=TimeUnit.NANOSECONDS.toSeconds(avgTurnaroundTime/jobBatch.length);

        CalcSimulation.avgServeField.setText(String.valueOf(avgTurnaroundTime) + "s");

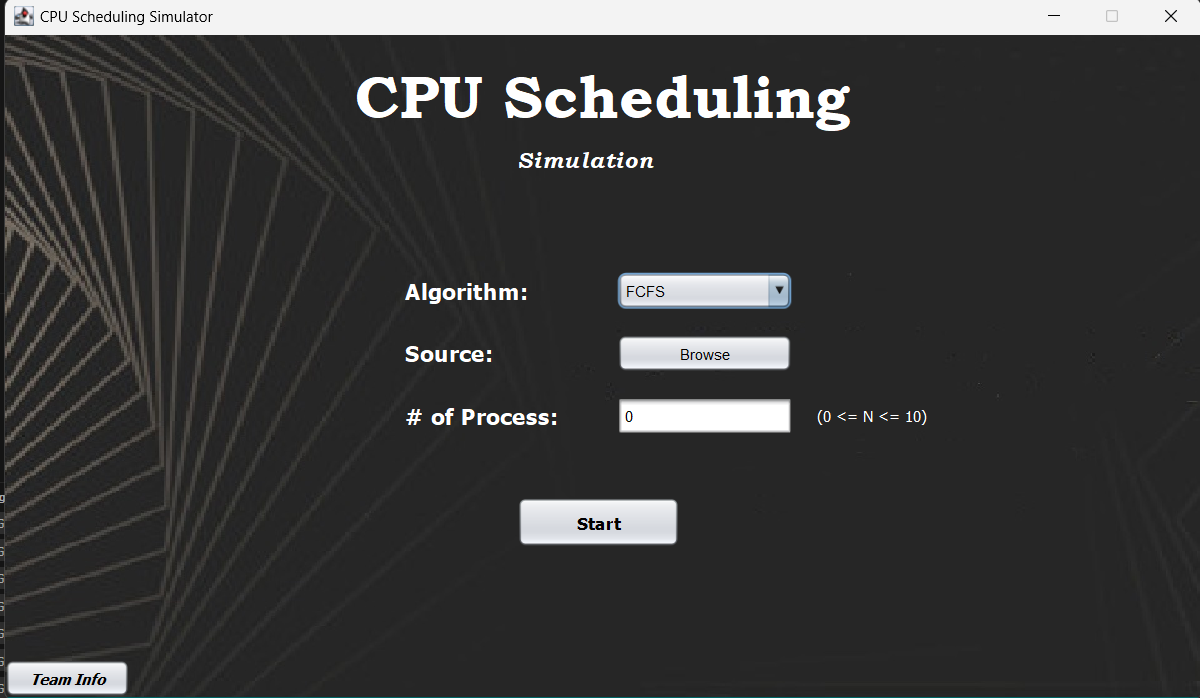
        totalExecutionTime = System.nanoTime() - CalcSimulation.STRTTIME;

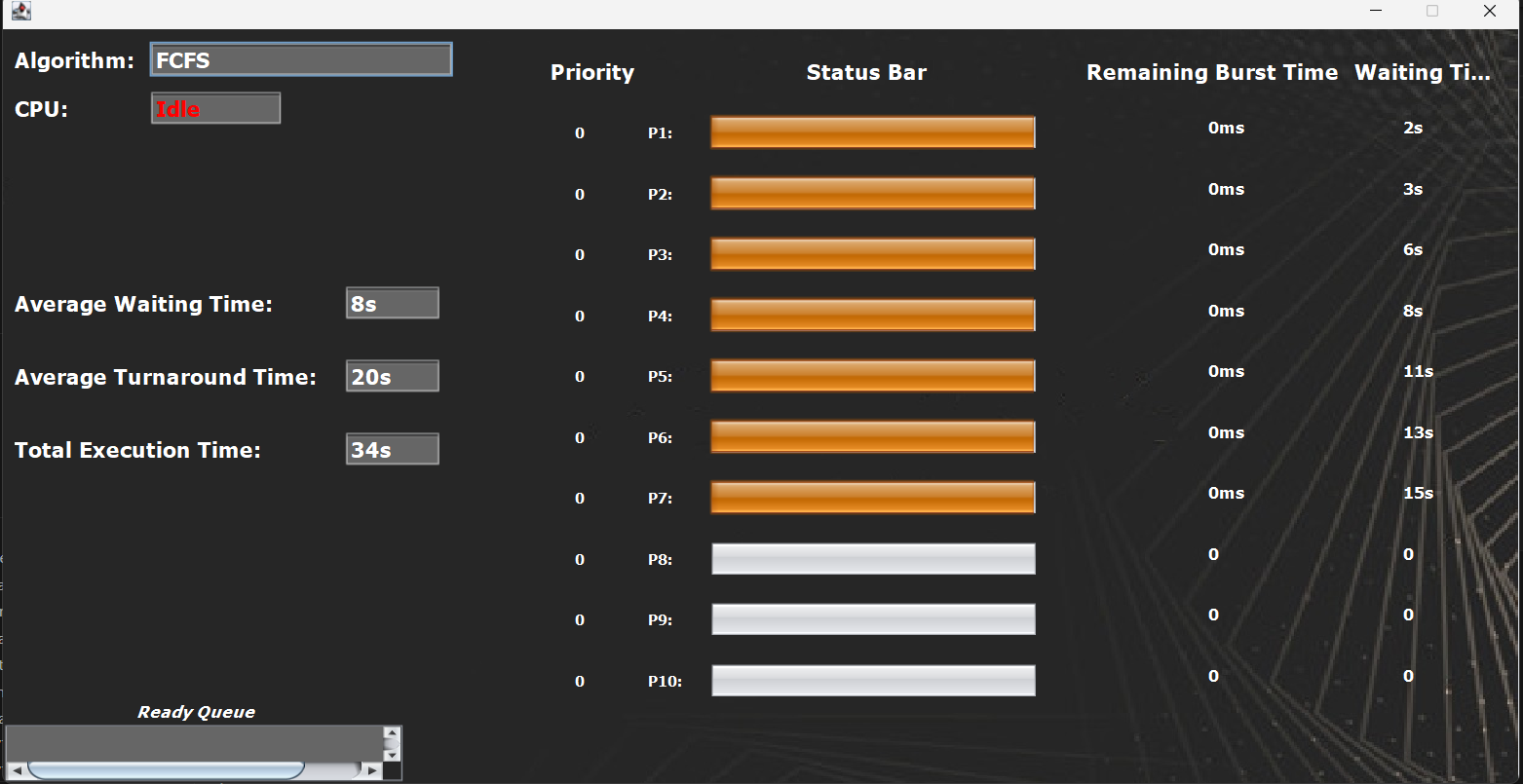
        totalExecutionTime = TimeUnit.NANOSECONDS.toSeconds(totalExecutionTime);

        CalcSimulation.totalExecField.setText(String.valueOf(totalExecutionTime) + "s");

    }

}

**Output:**



**FUTURE TRENDS:**

A**daptive Scheduling:** Modern systems will increasingly use adaptive scheduling algorithms that dynamically adjust based on workload characteristics and system performance metrics.

**Machine Learning Integration**: Machine learning techniques will be integrated into CPU scheduling to predict and manage workloads more effectively. Algorithms will learn from historical data to make intelligent scheduling decisions, reducing latency and improving overall system performance.

**Heterogeneous Architectures**: As systems incorporate diverse processors (e.g., CPUs, GPUs, TPUs), scheduling will need to handle varying types of processing units. Future schedulers will manage workloads across these heterogeneous environments, optimizing for different types of tasks and processing capabilities.

**Energy Efficiency**: With growing concerns about energy consumption, scheduling algorithms will focus on minimizing power usage. Techniques such as dynamic voltage and frequency scaling (DVFS) and energy-aware scheduling will be used to balance performance and power consumption.

**Real-Time Systems**: Advances in real-time scheduling will enhance the ability to guarantee deadlines and manage critical tasks in systems requiring high reliability and precision, such as autonomous vehicles and industrial control systems.

**CONCLUSION**

Implementing CPU scheduling in an operating system using Java provides a robust framework for managing process execution efficiently. Java's object-oriented approach enables modular design, making it easier to implement and modify scheduling algorithms like FCFS, SJF, and Round Robin. The use of Java's concurrent programming capabilities ensures accurate real-time scheduling and effective resource management. By leveraging Java's built-in libraries and features, developers can create a responsive and adaptable scheduling system that supports various process management needs. Overall, Java’s versatility and strong support for concurrent operations make it an excellent choice for developing a reliable and scalable CPU scheduling system, enhancing system performance and ensuring optimal utilization of computing resources.