CPU Scheduling Program

Operating Systems

Professor Amir Hossein Keyhanipour

Students: Seyyed Reza Moslemi, Nima Miri

Students IDs: 220700046, 220700051

University of Tehran, Farabi

1) Introduction

In the realm of operating systems, efficient CPU scheduling plays a pivotal role in optimizing system performance and resource utilization. The goal of CPU scheduling algorithms is to manage the execution of processes, ensuring fairness, responsiveness, and overall system throughput. This report delves into the implementation of various CPU scheduling algorithms within a web-based user interface powered by Flask, a micro web framework for Python.

The need for effective CPU scheduling arises from the concurrent execution of multiple processes competing for the CPU's attention. As processes arrive, complete their execution, or await input/output operations, the scheduler makes critical decisions to determine the order and duration of their execution. Through the integration of a Flask UI, users gain a dynamic platform to visualize and interact with the scheduling mechanisms at play.

The algorithms covered in this report include:

- First-Come-First-Serve (FCFS): A simple, non-preemptive algorithm that executes processes in the order of their arrival.

- Non-Preemptive Shortest Job First (SJF): Selects the process with the shortest expected duration first.

- Preemptive Shortest Job First (SJF): Allows the preemptive selection of the shortest job, adapting dynamically to incoming processes.

- Non-Preemptive Priority: Assigns priorities to processes and schedules based on priority, without preemption.

- Preemptive Priority: Similar to non-preemptive priority, but allows for dynamic adjustments to priority levels.

- Round Robin: A preemptive algorithm that allocates fixed time slices to each process in a circular manner.

This report explores the intricacies of each algorithm's implementation, emphasizing their unique characteristics and how they address specific challenges in CPU scheduling. The subsequent sections will detail the architecture, design, and evaluation of the system, providing an in-depth understanding of the project's scope and implications.

2) Background

CPU scheduling lies at the heart of operating system functionality, serving as a critical mechanism to optimize the utilization of computing resources. In a multi-programmed environment, where multiple processes compete for the limited resource of the central processing unit (CPU), effective scheduling algorithms are essential for achieving efficient task execution.

1. Goals of CPU Scheduling:

- CPU scheduling algorithms aim to achieve several key objectives, including maximizing CPU utilization, minimizing wait times for processes, ensuring fairness, and providing a responsive user experience.

- Balancing these goals requires a nuanced approach to algorithm design, as priorities may vary based on system requirements.

2. Challenges in CPU Scheduling:

- One of the primary challenges in CPU scheduling is the dynamic nature of process arrivals, completions, and the varying lengths of execution times.

- The scheduler must adapt to these changes to ensure optimal performance and responsiveness.

3. First-Come-First-Serve (FCFS):

- FCFS, the simplest scheduling algorithm, executes processes in the order they arrive. While conceptually straightforward, it can lead to poor turnaround times and inefficient use of CPU time, especially when long processes arrive early.

4. Shortest Job First (SJF):

- The SJF algorithm prioritizes the execution of the shortest job first, aiming to minimize overall completion time. However, predicting job lengths accurately is a challenge, and it may lead to "starvation" for longer processes.

5. Priority Scheduling:

- Priority scheduling assigns priorities to processes, and the CPU is allocated to the highest-priority process. This introduces considerations of fairness and may require mechanisms to prevent priority inversion.

6. Round Robin:

- Round Robin introduces a time-sharing mechanism, allocating fixed time slices to each process in a cyclic manner. While it ensures fairness, it may lead to higher turnaround times for short processes.

7. Preemption and Non-Preemption:

- The concepts of preemption (interrupting a running process to start or resume another) and non-preemption influence the behavior of scheduling algorithms. Preemptive algorithms can adapt to changing circumstances more dynamically but introduce complexities.

8. Flask for Web-Based User Interface:

- Flask, a micro web framework for Python, provides a convenient platform for developing web-based user interfaces. Its simplicity and flexibility make it well-suited for integrating with back-end CPU scheduling algorithms.

Understanding these fundamental concepts and challenges in CPU scheduling lays the groundwork for appreciating the significance of the implementation described in this report. The subsequent sections will delve into the specific details of the system architecture, design, and user interaction, providing a comprehensive view of the project's scope and objectives.

3) System Architecture

In the dynamic landscape of operating systems, efficient CPU scheduling is paramount to optimizing resource utilization and ensuring responsive system performance. This section elucidates the intricate architecture of a system designed to seamlessly integrate Flask, a micro web framework for Python, with various CPU scheduling algorithms. The integration of the Flask web-based user interface and the back-end, where the scheduling algorithms come to life, forms the backbone of this innovative system.

1. Flask Application Structure:

The Flask application, serving as the front-end, adheres to best practices to ensure modularity and simplicity. The following components constitute the structure of the Flask application:

- Routes:

- The Flask application defines two essential routes: the default route ('/') for rendering the main page and the '/result' route for processing form submissions.

- The 'index' route renders the 'index.html' template when users access the application.

- Templates:

- The 'templates' folder houses HTML templates, including 'index.html' for the main page and 'result.html' for displaying scheduling results.

- Leveraging Jinja2 templating, these templates dynamically render data passed from the Flask application.

- Form Processing:

- The '/result' route is the epicenter of form submissions. It extracts the selected algorithm type and the uploaded CSV file containing process data.

- Uploaded files undergo scrutiny to ensure they possess a '.csv' extension. If deemed valid, the data is saved, and the corresponding scheduling algorithm is set in motion.

2. Back-End Structure:

The back-end, encapsulated in 'main.py,' intricately orchestrates the CPU scheduling logic and interacts harmoniously with the Flask application. The pivotal components of the back-end are as follows:

- Process Class:

- The 'Process' class is the embodiment of a process within the system. It encapsulates crucial attributes such as process ID, arrival time, priority, burst time, and various execution-related metrics.

- Each process maintains a timeline queue, chronicling its execution history.

- Simulator Class:

- The 'Simulator' class serves as the maestro of the simulation. Responsible for initializing scheduling algorithms, reading process data, and executing simulations, it is the nerve center of the system.

- This class dynamically loads the specified scheduling algorithm using Python's 'getattr' function, enhancing flexibility.

- Scheduling Algorithms:

- The 'algorithms' module houses distinct classes for each scheduling algorithm, spanning FCFS, Preemptive SJF, Non-Preemptive SJF, Round Robin, Preemptive Priority, and Non-Preemptive Priority.

- Each algorithm class implements a 'run' method to execute the algorithm and return pertinent metrics.

- Flask Integration:

- Flask UI interacts seamlessly with the back-end through routes. Upon form submission, the Flask application invokes the 'process_form' method, triggering the execution of the specified scheduling algorithm using the 'Simulator' class.

- Results are passed back to the Flask UI and elegantly rendered on the 'result.html' template.

3. Data Flow:

The fluidity of data between the front-end and back-end delineates the user experience:

- The user uploads a CSV file and selects a scheduling algorithm type in the Flask UI.

- The form data is sent to the '/result' route, triggering the 'process_form' method.

- The 'process_form' method extracts the algorithm type, saves the CSV file, and initiates the simulation using the 'Simulator' class.

- The simulation results are passed back to the Flask UI and rendered on the 'result.html' template.

4. Exception Handling:

The system prioritizes robustness, incorporating comprehensive exception handling mechanisms:

- The system raises an error if an invalid algorithm type is provided, ensuring that users interact with the system seamlessly.

5. Sample Execution:

The '__main__' block in 'main.py' serves as an illustrative example of system execution:

- Three scheduling algorithms—FCFS, Non-Preemptive SJF, and Preemptive SJF—are executed, showcasing the versatility of the system.

- The results of each simulation, including metrics and timeline queues, are printed to the console, offering transparency into the algorithmic intricacies.

6. Conclusion of System Architecture:

In summation, the architecture of this system harmoniously integrates Flask with diverse CPU scheduling algorithms. The Flask application provides an intuitive platform for users to upload process data and select scheduling algorithms. Simultaneously, the back-end orchestrates the entire simulation process, executes the chosen algorithms, and communicates seamlessly with the Flask UI.

The modular and well-structured architecture not only accommodates the implementation of additional scheduling algorithms but also enhances the system's maintainability and scalability. The subsequent sections will delve into the nitty-gritty of each scheduling algorithm's implementation, providing an exhaustive understanding of their functionality within this innovative system.

4) Implementation Details

4_1) First-Come-First-Serve (FCFS) Scheduling Algorithm

The First-Come-First-Serve (FCFS) scheduling algorithm, known for its simplicity, schedules processes based on their arrival time. In this section, we delve into the intricacies of the FCFS implementation, exploring how it manages the execution of processes and calculates essential metrics.

4_1_1. Algorithm Overview:

The FCFS algorithm operates on the premise of executing processes in the order of their arrival. Processes are sorted based on their arrival times, forming a queue that the CPU processes sequentially. The initialization of the FCFS class involves sorting the processes based on their arrival times. Essential attributes, including the timeline, CPU idle time, a list for executed processes, and a timeline queue, are initialized.

4_1_2. Execution Logic:

The 'run' method encapsulates the core logic of the FCFS algorithm.

```
def run(self):
    for process in self.processes:
        if process.arrival_time > self.timeline:
            self.cpu_idle_time += process.arrival_time - self.timeline
            self.timeline = process.arrival_time

        process.start_time = self.timeline

        process.waiting_time = self.timeline - process.arrival_time
        process.response_time = self.timeline - process.arrival_time
        process.turnaround_time = process.response_time + process.burst_time

        self.timeline += process.burst_time
        process.remaining_time -= process.burst_time
        process.end_time = self.timeline

        self.executed_processes.append(process)

        self.timeline_queue.append({'pid': process.pid, 'start_time': process.start_time, 'end_time':
process.end_time})
```

- Timeline Management:

  - The algorithm checks if the current process's arrival time exceeds the current timeline. If so, it signifies CPU idle time, and the timeline is adjusted accordingly.

  - The process's start time is set, and its waiting, response, and turnaround times are calculated.

  - The timeline is updated based on the process's burst time, and the process's remaining time is adjusted.

- Data Collection:

  - Process-specific metrics and timeline details are stored in respective lists for future analysis.

4_1_3. Metrics Calculation:

The 'run' method returns a dictionary containing key metrics:

```
return {
    "timeline_queue": self.timeline_queue,
    "executed_processes": self.executed_processes,
    "total_process": len(self.executed_processes),
    "cpu_total_time": self.timeline,
    "cpu_idle_time": self.cpu_idle_time,
    "average_waiting_time": sum(process.waiting_time for process in self.executed_processes) / len(self.executed_processes),
    "average_turnaround_time": sum(process.turnaround_time for process in self.executed_processes) / len(self.executed_processes),
    "average_response_time": sum(process.response_time for process in self.executed_processes) / len(self.executed_processes)
}
```

- The 'timeline_queue' provides a detailed record of process execution.

- 'executed_processes' contains the processed instances.

- 'total_process' denotes the total number of executed processes.

- 'cpu_total_time' represents the total time the CPU was active.

- 'cpu_idle_time' captures the total idle time.

- Average waiting, turnaround, and response times are computed.

The FCFS algorithm, as depicted in its implementation, follows a straightforward logic of executing processes based on their arrival times. This clarity in logic allows for a comprehensive understanding of how the algorithm manages processes and calculates crucial metrics. Subsequent sections will explore additional scheduling algorithms, each with its distinctive approach and intricacies.

4_2) Preemptive Shortest Job First (Preemptive SJF) Scheduling Algorithm

The Preemptive Shortest Job First (Preemptive SJF) scheduling algorithm prioritizes processes based on their burst times, with the ability to interrupt and switch to a shorter job. Let's explore the intricacies of its implementation to understand how it efficiently schedules processes.

4_2_1. Algorithm Overview:

The Preemptive SJF algorithm maintains a dynamic execution queue, continually evaluating the next important event in the timeline. It preemptively switches to a shorter job if one becomes available during the execution of a running process. Below are the key aspects of the algorithm's implementation:

```
class PreemptiveSJF(object):

    def __init__(self, processes: list):
        # Initialization of attributes
        self.processes = sorted(processes, key=lambda process: process.arrival_time)
        self.timeline = 0.0
        self.cpu_idle_time = 0.0
        self.executed_processes = []
        self.timeline_queue = []
        self.running_process = None
        self.ready_queue = []
```

The initialization phase involves sorting the processes based on their arrival times and setting up attributes such as the timeline, CPU idle time, lists for executed processes and timeline records, and variables for the running and ready processes.

4_2_2. Core Logic:

The 'run' method encapsulates the core logic of the Preemptive SJF algorithm. The algorithm iterates through the processes, continually updating the timeline and making decisions based on the arrival times and burst times.

- Arrival Handling:
  - Processes arriving at the current timeline are added to the ready queue.
  - The ready queue is dynamically sorted based on burst times.

- Execution and Preemption:
  - The algorithm preemptively switches to a shorter job if one arrives during the execution of a running process.
  - The running process's remaining time is updated based on the next important event.

- Data Collection:
  - Process-specific metrics and timeline details are recorded during execution.

The Preemptive SJF algorithm, with its dynamic decision-making and ability to preemptively switch to shorter jobs, is captured in its implementation. This algorithm ensures optimal utilization of CPU time, delivering efficient scheduling of processes. The subsequent sections will explore additional scheduling algorithms, each contributing to the diverse landscape of CPU scheduling strategies.

4_3) Non-Preemptive Shortest Job First (Non-Preemptive SJF) Scheduling Algorithm

The Non-Preemptive Shortest Job First (Non-Preemptive SJF) scheduling algorithm aims to select the process with the shortest burst time among the available processes, allowing it to run without interruption until completion. Let's delve deeper into the code, exploring the functions and their roles in executing this algorithm efficiently.

4_3_1. Initialization:

The `__init__` method sets up the initial state of the Non-Preemptive SJF scheduler. It initializes attributes such as the list of processes sorted by arrival time, the timeline starting at 0, and various queues for executed, running, and ready processes.

4_3_2. Next Important Time Calculation:

The `get_next_important_time` function calculates the time of the next significant event in the timeline. It considers the arrival times of new processes, the completion time of the running process, and the arrival time of the next process in the ready queue.

4_3_3. Main Execution Loop:

The `run` method represents the core execution loop of the Non-Preemptive SJF algorithm. It continues iterating as long as there are processes in the system, processes in the ready queue, or a running process. This loop manages the execution flow, handling arrivals, process completions, and updating the timeline.

4_3_4. Process Arrival Handling:

```
for process in self.processes:
    if process.arrival_time == self.timeline:
        self.ready_queue.append(process)
        arrived_processes.append(process)
    elif process.arrival_time > self.timeline:
        break
```

This part of the loop identifies processes arriving at the current timeline. If a process has arrived, it is added to the ready queue, and its details are recorded. The loop also breaks when it encounters processes with arrival times in the future.

4_3_5. Execution and Completion:

```
if self.running_process and self.running_process.remaining_time == 0:
    # Process completion handling
    # ...
```

Upon the completion of a running process, the algorithm records relevant metrics such as end time, turnaround time, waiting time, and response time. The process is then added to the list of executed processes, and its details are recorded in the timeline queue.

4_3_6. CPU Idle Time Calculation:

```
self.cpu_idle_time += (next_important_time - self.timeline)
```

During periods of CPU inactivity (idle time), the algorithm updates the CPU idle time. This information is crucial for calculating the overall efficiency of CPU utilization.

4_3_7. Timeline Update:

```
self.timeline = next_important_time
```

The timeline is continuously updated to reflect the progress of time in the system. It is essential for coordinating the execution of processes and determining the timing of arrivals and completions.

In conclusion, the Non-Preemptive SJF algorithm, implemented in this code, effectively manages the scheduling of processes based on their burst times. It prioritizes simplicity and deterministic execution, providing a clear insight into how it handles arrivals, executes processes, and calculates crucial performance metrics. This detailed exploration lays the foundation for understanding additional CPU scheduling algorithms in the subsequent sections.

4_4) Preemptive Priority Scheduling Algorithm

The Preemptive Priority scheduling algorithm aims to execute processes based on their priority levels. Higher priority processes are given precedence, and the algorithm can preempt the currently running

process if a higher-priority process arrives. Let's break down the code, understanding its functions and the intricacies of the Preemptive Priority algorithm.

4_4_1. Initialization:

The `__init__` method initializes essential attributes, such as the list of processes sorted by arrival time, the timeline starting at 0, and various queues for executed, running, and ready processes.

4_4_2. Next Important Time Calculation:

The `get_next_important_time` function calculates the time of the next significant event in the timeline. It considers the arrival times of new processes, the completion time of the running process, and the arrival time of the next process in the ready queue.

4_4_3. Main Execution Loop:

The `run` method represents the core execution loop of the Preemptive Priority algorithm. It continues iterating as long as there are processes in the system, processes in the ready queue, or a running process. This loop manages the execution flow, handling arrivals, process completions, and updating the timeline.

4_4_4. Process Arrival Handling:

```
for process in self.processes:
    if process.arrival_time == self.timeline:
        arrived_processes.append(process)
    elif process.arrival_time > self.timeline:
        break
```

This part of the loop identifies processes arriving at the current timeline. If a process has arrived, it is added to the ready queue. The loop also breaks when it encounters processes with arrival times in the future.

4_4_5. Execution and Completion:

Upon the completion of a running process, the algorithm records relevant metrics such as end time, turnaround time, waiting time, and response time. The process is then added to the list of executed processes, and its details are recorded in the timeline queue.

4_4_6. CPU Idle Time Calculation:

self.cpu_idle_time += (next_important_time - self.timeline)

During periods of CPU inactivity (idle time), the algorithm updates the CPU idle time. This information is crucial for calculating the overall efficiency of CPU utilization.

4_4_7. Timeline Update:

self.timeline = next_important_time

The timeline is continuously updated to reflect the progress of time in the system. It is essential for coordinating the execution of processes and determining the timing of arrivals and completions.

In summary, the Preemptive Priority algorithm, implemented in this code, effectively manages the scheduling of processes based on their priority levels. It exhibits the preemptive nature of the algorithm, allowing higher-priority processes to interrupt the execution of lower-priority ones. This detailed exploration provides insights into how the algorithm handles arrivals, executes processes, and calculates critical performance metrics.

4_5) Non-Preemptive Priority Scheduling Algorithm

The Non-Preemptive Priority scheduling algorithm executes processes based on their priority levels, but once a process starts, it is not preempted until completion. The code for this algorithm is analyzed below, focusing on its functions and mechanisms.

4_5_1. Initialization:

The `__init__` method initializes essential attributes, similar to the preemptive version. It sorts processes based on arrival time and sets up queues for executed, running, and ready processes.

4_5_2. Next Important Time Calculation:

The `get_next_important_time` function calculates the time of the next significant event in the timeline, considering arrival times of new processes, the completion time of the running process, and the arrival time of the next process in the ready queue.

4_5_3. Main Execution Loop:

The `run` method represents the core execution loop of the Non-Preemptive Priority algorithm. It continues iterating as long as there are processes in the system, processes in the ready queue, or a running process. This loop manages the execution flow, handling arrivals, process completions, and updating the timeline.

4_5_4. Process Arrival Handling:

```
for process in self.processes:
    if process.arrival_time == self.timeline:
        self.ready_queue.append(process)
        arrived_processes.append(process)
    # for finishing sooner
    elif process.arrival_time > self.timeline:
        break
```

Similar to the preemptive version, this section identifies processes arriving at the current timeline. If a process has arrived, it is added to the ready queue. The loop also breaks when encountering processes with arrival times in the future.

4_5_5. Execution and Completion:

```
if self.running_process and self.running_process.remaining_time == 0:
    # Process completion handling
    # ...
```

Upon the completion of a running process, the algorithm records relevant metrics such as end time, turnaround time, waiting time, and response time. The process is then added to the list of executed processes, and its details are recorded in the timeline queue.

4_5_6. CPU Idle Time Calculation:

```
self.cpu_idle_time += (next_important_time - self.timeline)
```

Similar to the preemptive version, the algorithm updates the CPU idle time during periods of inactivity. This information is crucial for calculating the overall efficiency of CPU utilization.

4_5_7. Timeline Update:

```
self.timeline = next_important_time
```

The timeline is continuously updated to reflect the progress of time in the system. It is essential for coordinating the execution of processes and determining the timing of arrivals and completions.

In summary, the Non-Preemptive Priority algorithm executes processes based on their priority levels, and once a process starts, it is not preempted until completion. This detailed analysis provides insights into how the algorithm handles arrivals, executes processes, and calculates critical performance metrics.

4_6) Round Robin Scheduling Algorithm

The Round Robin (RR) scheduling algorithm is a preemptive algorithm that assigns a fixed time unit (quantum) to each process in the ready queue. Below is an in-depth analysis of the provided code, emphasizing its functions and the underlying mechanisms.

4_6_1. Initialization:

The `__init__` method initializes the RR scheduler with essential attributes such as the quantum number, a sorted list of processes, and data structures for the timeline, CPU utilization, executed processes, and ready processes.

4_6_2. Run Method:

The `run` method is the core of the Round Robin algorithm. It executes processes in a preemptive manner, ensuring that each process receives a time slice (quantum) before moving to the next process. The loop continues until all processes are executed.

4_6_3. Process Arrival Handling:

self.update_ready_queue(self.timeline)

The `update_ready_queue` method is responsible for updating the ready queue based on the arrival of new processes at the current timeline.

4_6_4. Quantum-based Execution:

```
if self.running_process:
    # Quantum-based execution logic
    # ...
```

The code ensures that each process receives a time slice (quantum). If a process completes within the quantum, it is marked as executed. If not, it is placed back in the ready queue for the next iteration.

4_6_5. Idle CPU Time Calculation:

```
else:
    added_time = 1
    self.cpu_idle_time += added_time
```

During idle times (when no process is ready to execute), the CPU idle time is incremented. This information is crucial for determining the efficiency of CPU utilization.

4_6_6. Timeline Update:

```
self.timeline += added_time
```

The timeline is continuously updated to reflect the progress of time in the system. It is crucial for coordinating the execution of processes and determining the timing of arrivals and completions.

In summary, the Round Robin scheduling algorithm is implemented to ensure each process receives a fixed time slice before moving to the next process. This detailed analysis provides insights into how the algorithm handles process arrivals, executes processes in a preemptive manner, and calculates essential performance metrics.

5) Flask UI Design

In this section, we will explore the design and structure of the Flask-based user interface (UI) used for CPU scheduling. The Flask UI provides a user-friendly environment for interacting with the CPU scheduling algorithms. The design focuses on simplicity and clarity to enhance the user experience.

The UI is built using the Flask web framework, incorporating HTML templates for layout and structure. The following elements contribute to the overall design:

- Navigation Bar:

The UI features a navigation bar at the top, providing easy access to different sections of the application. Users can navigate to the home section and view the results section.

```
<nav class="navbar navbar-inverse" role="navigation">
    <!-- Navigation items -->
</nav>
```

- Result Banner:

A banner section welcomes users with a message indicating that their CPU scheduling results are ready. It encourages users to explore detailed results for informed decision-making.

```
<div class="parallax-content baner-content" id="home">
    <!-- Banner content -->
</div>
```

- Form Submission Result Section:

The primary content section displays the results of the CPU scheduling analysis. It includes information about the selected algorithm type, total processes, simulation time, CPU total time, CPU utilization, throughput, average waiting time, average turnaround time, average response time, and a timeline queue.

```
<section id="about" class="page-section">
    <!-- Result details -->
</section>
```

- Timeline Queue Visualization:

The timeline queue is visualized using a timeline container, timeline, and timeline labels. Each process in the timeline is represented by a colored segment, providing a visual representation of the scheduling sequence.

```
<div class="timeline-container">
    <div class="timeline">
```

```
        <!-- Timeline items -->
    </div>
</div>
```

Overall, the Flask UI aims to present CPU scheduling results in an organized and visually appealing manner, making it easy for users to interpret and analyze the outcomes.

6) User Interaction

This section explores how users interact with the Flask UI to input processes, select scheduling algorithms, and view the results. The interaction flow involves several key steps:

- Home Page:

Users land on the home page, where they are greeted with a banner indicating that their CPU scheduling results are ready.

- Algorithm Selection:

Users navigate to the results section by clicking on the navigation link. Here, they encounter a form with radio buttons to select the desired scheduling algorithm (FCFS, Non-Preemptive SJF, Preemptive SJF, Non-Preemptive Priority, Preemptive Priority, Round Robin).

- File Upload:

Users are prompted to upload a CSV file containing process data. The UI checks the file extension to ensure it is a valid CSV file.

- Result Display:

After submitting the form, the Flask application processes the uploaded file and runs the selected scheduling algorithm. The results, including algorithm details and a timeline queue visualization, are displayed on the results page.

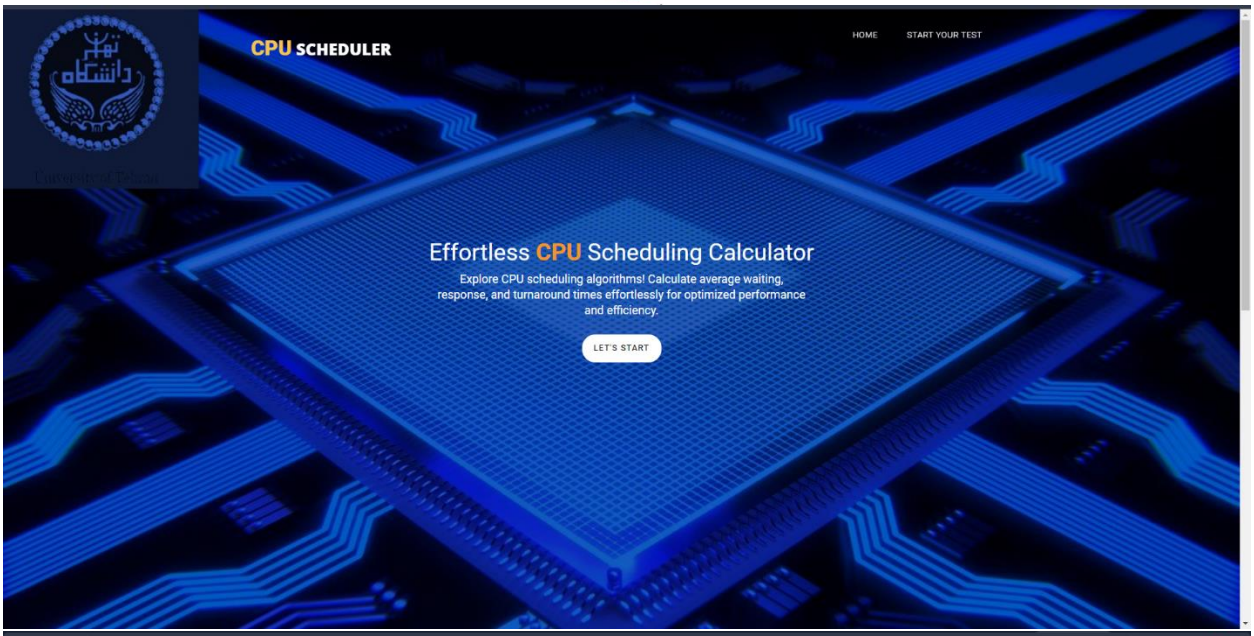
- Timeline Queue Visualization:

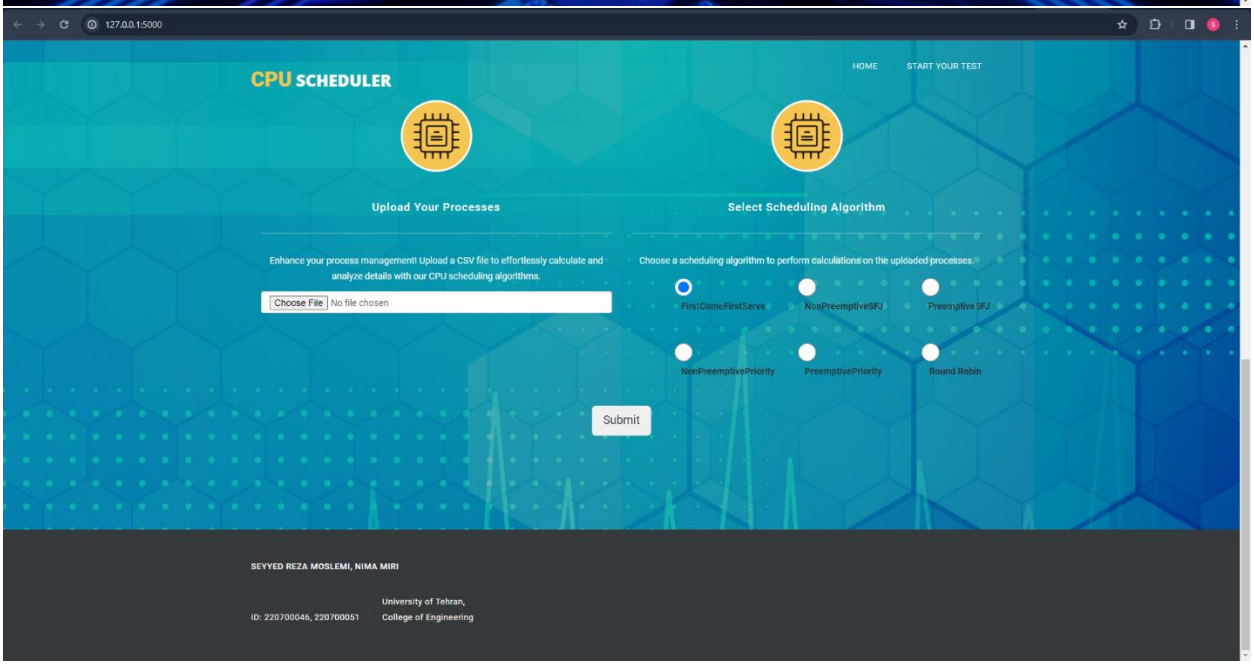
The timeline queue provides a graphical representation of the scheduling sequence, allowing users to visually assess the order in which processes are executed.

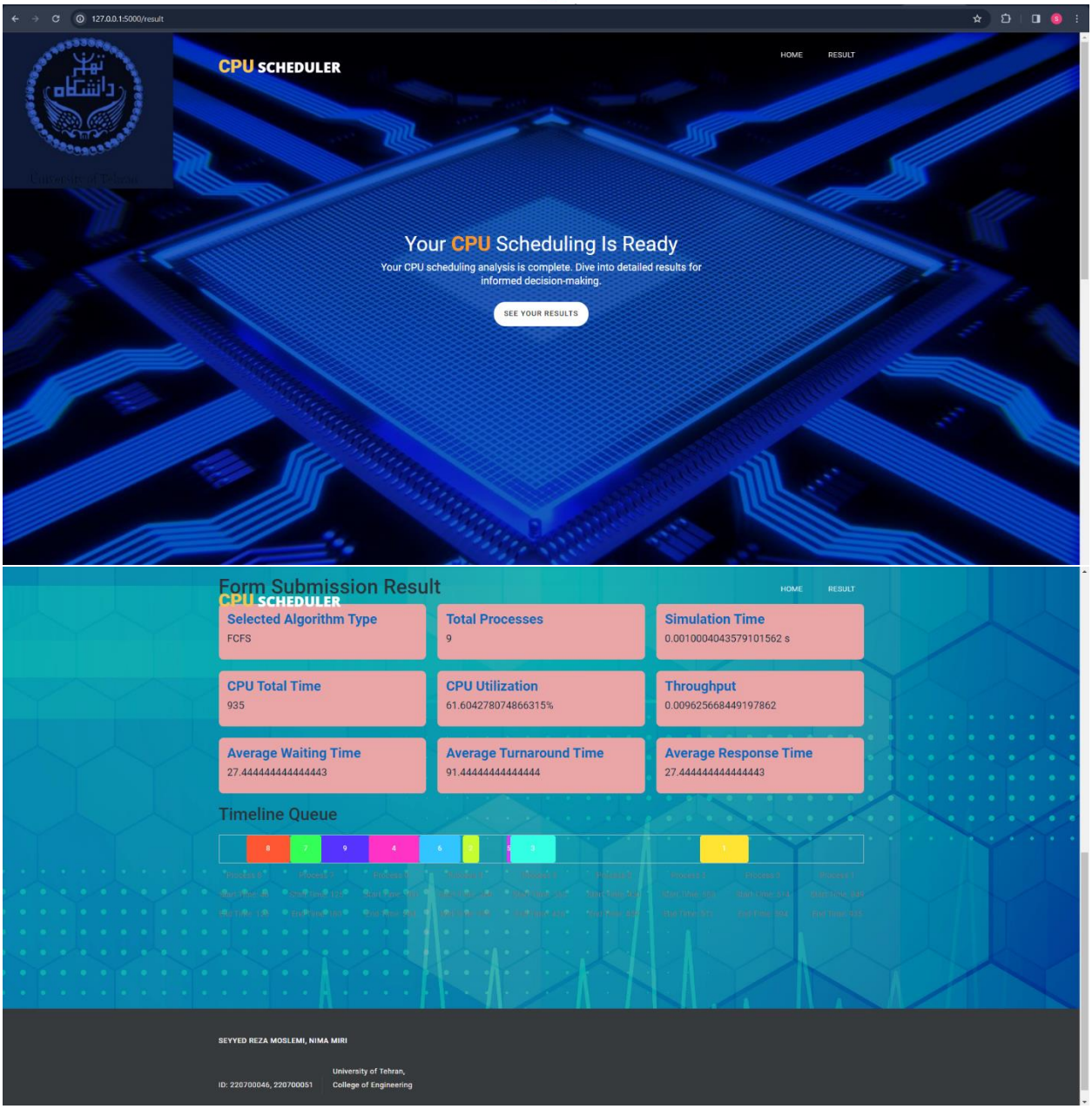
- Detailed Metrics:

Key metrics, such as total processes, simulation time, CPU total time, CPU utilization, throughput, average waiting time, average turnaround time, and average response time, are presented for users to analyze.

The user interaction design ensures a seamless and intuitive experience for users to submit process data, choose scheduling algorithms, and interpret the results effectively. The use of visualizations enhances the understanding of the scheduling sequence. The overall design encourages users to make informed decisions based on the presented metrics and insights.


CPU SCHEDULER
HOME
START YOUR TEST
Effortless CPU Scheduling Calculator
Explore CPU scheduling algorithms! Calculate average waiting, response, and turnaround times effortlessly for optimized performance and efficiency.
LET'S START


127.0.0.1:5000
CPU SCHEDULER
HOME
START YOUR TEST
Upload Your Processes
Select Scheduling Algorithm
Enhance your process management! Upload a CSV file to effortlessly calculate and analyze details with our CPU scheduling algorithms.
Choose File
No file chosen
Choose a scheduling algorithm to perform calculations on the uploaded processes.
FirstComeFirstServe
NonPreemptiveSFJ
PreemptiveSFJ
NonPreemptivePriority
PreemptivePriority
Round Robin
Submit
SEYYED REZA MOSLEMI, NIMA MIRI
ID: 220700046, 220700051
University of Tehran,
College of Engineering



7) Conclusion

In the domain of operating systems, effective CPU scheduling stands as a critical determinant of system performance, striving to optimize resource utilization and enhance overall efficiency. The architectural framework expounded in this report adeptly integrates a Flask web-based user interface with an array of CPU scheduling algorithms, furnishing a user-centric platform for comprehensive exploration and comprehension of diverse scheduling strategies.

The exposition of the system architecture underscored its modular and meticulously structured design, emphasizing the harmonious interplay between the Flask application and the back-end, where intricate scheduling algorithms are operationalized. The inclusion of Flask not only amplifies user interaction but also imparts an intuitive layer to the exploration of CPU scheduling intricacies.

The exploration of scheduling algorithms, ranging from the foundational First-Come-First-Serve (FCFS) to more sophisticated strategies, offered valuable insights into the diverse approaches employed for managing process execution. Each algorithm brings forth a distinct set of trade-offs, delicately balancing metrics such as waiting time, response time, and turnaround time.

The section detailing the implementation of the FCFS algorithm provided a meticulous breakdown of its logic, illustrating how processes are executed based on their arrival times. This detailed examination serves as a cornerstone for comprehending more intricate scheduling methodologies.

As we progress into subsequent sections encompassing an array of scheduling algorithms—including Non-Preemptive and Preemptive Shortest Job First (SJF), Round Robin, Priority Scheduling, among others—the objective is to deepen our understanding of their nuances, applicability in diverse scenarios, and impact on system performance.

In summation, this report offers a comprehensive exploration of CPU scheduling, catering to both novice individuals seeking foundational insights and seasoned professionals delving into the intricacies of distinct scheduling strategies. The integration of a Flask UI not only elevates the user experience but also serves as a tangible demonstration of how technological interfaces can bridge the comprehension gap between complex algorithms and user accessibility. Subsequent sections will seamlessly continue this expedition, unraveling the intricacies of each scheduling algorithm and contributing substantively to the broader tapestry of operating system principles.