**DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING**

**COLLEGE OF E&ME, NUST, RAWALPINDI**

****

**PROJECT REPORT**

**SUBMITTED TO:**

**Prof. Dr. Usman Qamar**

**SUBMITTED BY (Group Members):**

Rohan Arshad (460852)

Hisham Bin Nauman (480587)

Ahsan Ali (475525)

Haseeb Ahmed (456879)

Muhammad Hashim (482312)

**CE-** 45(B)

**Submission Date: 13th May 2025**

Contents

[INTRODUCTION 1](#_Toc198073024)

[PROBLEM STATEMENT 1](#_Toc198073025)

[Primary Objective 1](#_Toc198073026)

[METHODOLOGY AND IMPLEMENTATION 2](#_Toc198073027)

[Methodology 2](#_Toc198073028)

[Implementation 4](#_Toc198073029)

[SYSTEM ARCHITECTURE 6](#_Toc198073030)

[Detailed Architecture Diagram 6](#_Toc198073031)

[Data Flow Diagram 7](#_Toc198073032)

[Class Diagram: 8](#_Toc198073033)

[PERFORMANCE ANALYSIS 9](#_Toc198073034)

[RESULTS AND DISCUSSION 9](#_Toc198073035)

[Test Case 1: Mixed Process Characteristics 9](#_Toc198073036)

[Test Case 2: High System Load 9](#_Toc198073037)

[Test Case 3: Many High-Priority Processes 10](#_Toc198073038)

[Discussion 10](#_Toc198073039)

[LIMITATIONS AND CHALLENGES 10](#_Toc198073040)

[FUTURE WORK 11](#_Toc198073041)

[CONCLUSION 12](#_Toc198073042)

[REFERENCES 12](#_Toc198073043)

## INTRODUCTION

This project, the **Dynamic Process Scheduler**, is a CPU scheduling simulator built in C++ to mimic how modern operating systems manage process scheduling. The process scheduler assigns the best scheduling algorithm based on the system's current state.

The project has several key parts: a **Process Control Block (PCB)** class to store process details, a **Scheduler** class that runs algorithms like FCFS, Shortest Job First (SJF), Priority Scheduling (PR), and Round Robin (RR), a **Policy Engine** to decide which algorithm to use, a **Simulation Manager** to run everything, and a **Simulation Clock** to keep track of time. I/O device handling was also integrated, mimicking the resources a process needs for execution. The simulator logs metrics which are used to choose the respective scheduling algorithms.

This report gives a detailed overview of the implementation of the Process Scheduler and how it mimics an actual process scheduler of an Operating System. This project can be used as an educational tool to help students (like us!) understand CPU scheduling.

## PROBLEM STATEMENT

Traditional CPU scheduling mechanisms in operating systems implement fixed algorithms like First-Come-First-Served (FCFS), Shortest Job First (SJF), Priority Scheduling, or Round Robin. These fixed algorithms are efficient for specific scenarios but may become suboptimal as system loads and process characteristics change dynamically. This project aims to develop a CPU scheduling simulator that:

* Implements multiple standard algorithms (FCFS, SJF, Priority, Round Robin).
* Analyses the process queue and system load in real-time.
* Dynamically selects the most suitable scheduling algorithm based on predefined policies.
* Tracks and displays important performance metrics.
* Provides a text-based user interface for observing simulation behaviour.

## Primary Objective

The primary aim of this project is to produce a software solution that behaves like an actual process scheduler by using a variety of algorithms, dynamically selecting the most suitable one at each moment. The idea was to illustrate not the basic mechanism of different algorithms (FCFS, SJF, RR, PR) via the "brain" of the scheduler, the Policy Engine, which selects the algorithm, all without manual intervention. This presents the simulator as a depiction of the modern-day process scheduler, where things are flexible based on the workload. By presenting the metrics of a process, including but not limited to turnaround, waiting, and the usage of CPU, we anticipate that the users will be able to get themselves familiar with the advantages and disadvantages of each algorithm, as well as the significance of dynamic scheduling.

## METHODOLOGY AND IMPLEMENTATION

### Methodology

We divided the project into a few parts to get it all organised. Here is how it all works:

#### Data Input.

The simulator reads process data from a file named Program.txt that contains processes with fields that include PPID, burst time, priority, and the need for I/O devices. Each process is represented as a Process Control Block (PCB), which contains all this information along with other attributes such as arrival time, and process state (i.e., New, Ready, Running, Waiting, Terminated). The Simulation Manager reads the program and loads the respective processes in a Job Queue (which is an STL list). From there, the processes go into the Ready Queue (a priority queue), which holds the processes that are waiting for the CPU, or the Device Queue if these processes are waiting for I/O devices.

For instance, a row in Program.txt like 1 10 1 0 represents a process with PPID 1, 10ms burst time, priority 1, and no I/O required; the PCB class is used to initialise a new object with this information and load the processes into respective queues. The checks in dealing with errors were done by displaying the error messages and then ignoring and skipping the bad lines if such cases existed. A small preview would be helpful if possible.

#### Data Processing.

The Simulation Manager is the heart of the project. It drives the simulation by transferring processes from one queue to another and calling the Scheduler to dispatch them. Here is how it goes:

* Job Queue to Ready Queue: If a new process does not require any I/O or if it receives the required I/O devices, its state is changed to READY, and the process is moved to the Ready Queue. The allocateResources() function is responsible for the verification of I/O devices. The simulation is initialised with twenty devices.
* Policy Engine: The Policy Engine, whose responsibility is, among others, to choose how the scheduler makes use of different scheduling algorithms based on system metrics i.e the Ready Queue size, the current average of the burst time, and the percentage of the high-priority processes to decide the algorithm. For instance, if the queue exceeds the threshold of twenty processes, the engine turns to Round Robin scheduling. In addition to that, ageing applies to all processes before applying a scheduling algorithm so that no process suffers from starvation.

* Scheduler: The Scheduler is responsible for using different scheduling algorithms. It holds the definition of what Round Robin, Shortest Job First, Priority and FCFS Scheduling are and how they will deal with the incoming PCBs.
* Deallocation: When a process is completed, the deallocateResources() function deallocates the resources from the process and releases I/O devices that the process had occupied back to the system, so that other processes in need of I/O devices may make use of them.
* The Simulation Clock ensures that everything is progressing in parallel by moving from one event to another, such as the execution of processes or the changing of contexts.

#### Data Display.

The process scheduler displays outputs on the console. It displays the following information:

* Queue Status: Sizes of the Job, Ready, and Device Queues.
* Scheduling Decisions: Which algorithm was chosen by the Policy Engine.
* Process Execution: Information like "PID 1 executing for 10ms" or "PID 2 completed at time 15ms".
* Performance Metrics: A struct containing average burst time, average waiting time, and the queue size.

#### Error Handling.

We implemented error handling as a part of making the simulator robust:

* If Program.txt cannot be opened, the simulator displays "Failed to open process file" and terminates execution accordingly.
* When a process has invalid data (e.g., negative burst time), an error message is displayed, and the error is processed as usual.
* If an edge case is detected (e.g., the Ready Queue is empty), the Policy Engine will default to FCFS as a solution if no other algorithm suits.

#### Code Organisation.

To make the program more readable and cleaner, the code was organised into the following modules:

* PCB.h: The class responsible for managing process-specific details and actions like updating the process state, generating child processes and marking a process as complete
* Scheduler.h: Describes First Come First Serve, Shortest Job First, Round Robin, and Priority-based algorithms.
* PolicyEngine.h: Identifies the algorithm to be utilised for a particular situation and applies ageing.
* SimulationManager.h: Controls queues, resources, as well as the simulation execution.
* IOdevices.h: Represents I/O devices virtually and updates their availability status.
* SimulationClock.h: Counts the time spent virtually.
* main.cpp: The entry point for the simulation

The comments are used in each file to explain the code's concept, while we have used variable names like readyQueue and burstTime instead of just using generic names to make it readable. The modular design gives us the ability to include new algorithms or characteristics later without rewriting everything.

### Implementation

We implemented the simulator in C++ by utilising STL containers such as list, priority\_queue, and vector to make the modular. The main.cpp file first creates a SimulationManager object, and then it calls the simulateScheduling() function. The simulateScheduling() function makes use of the rest of the classes to execute the simulation.

Here are the most important classes in the project:

* PCB: Stores process info such as PID, burst time, and priority. It has a method fork() to create processes, mimicking a real OS, and getters/setters to modify the state.
* Scheduler: This class oversees the four algorithms. Each algorithm updates the process state and uses the Simulation Clock to track time. For example, roundRobin(), which operates a deque to rotate through processes, giving each process the CPU for a specified time quantum. All these processes are controlled by a stop flag, which can stop an algorithm at any time and context-switch from one algorithm to another.

Four functions were implemented, each representing the mechanism of a scheduling algorithm:

1. **FCFSScheduler**: Implements the First-Come-First-Served algorithm.
2. **SJFScheduler**: Implements Shortest Job First algorithm.
3. **RoundRobinScheduler**: Implements the Round Robin algorithm with a configurable time quantum.
4. **PriorityScheduler**: Implements Priority Scheduling.

* Policy Engine: Summarises metrics like queue size and chooses an algorithm to work based on those metrics. Furthermore, this class also effectively ages processes to starvation. The Policy Engine implements the following rules for selecting a scheduling algorithm:

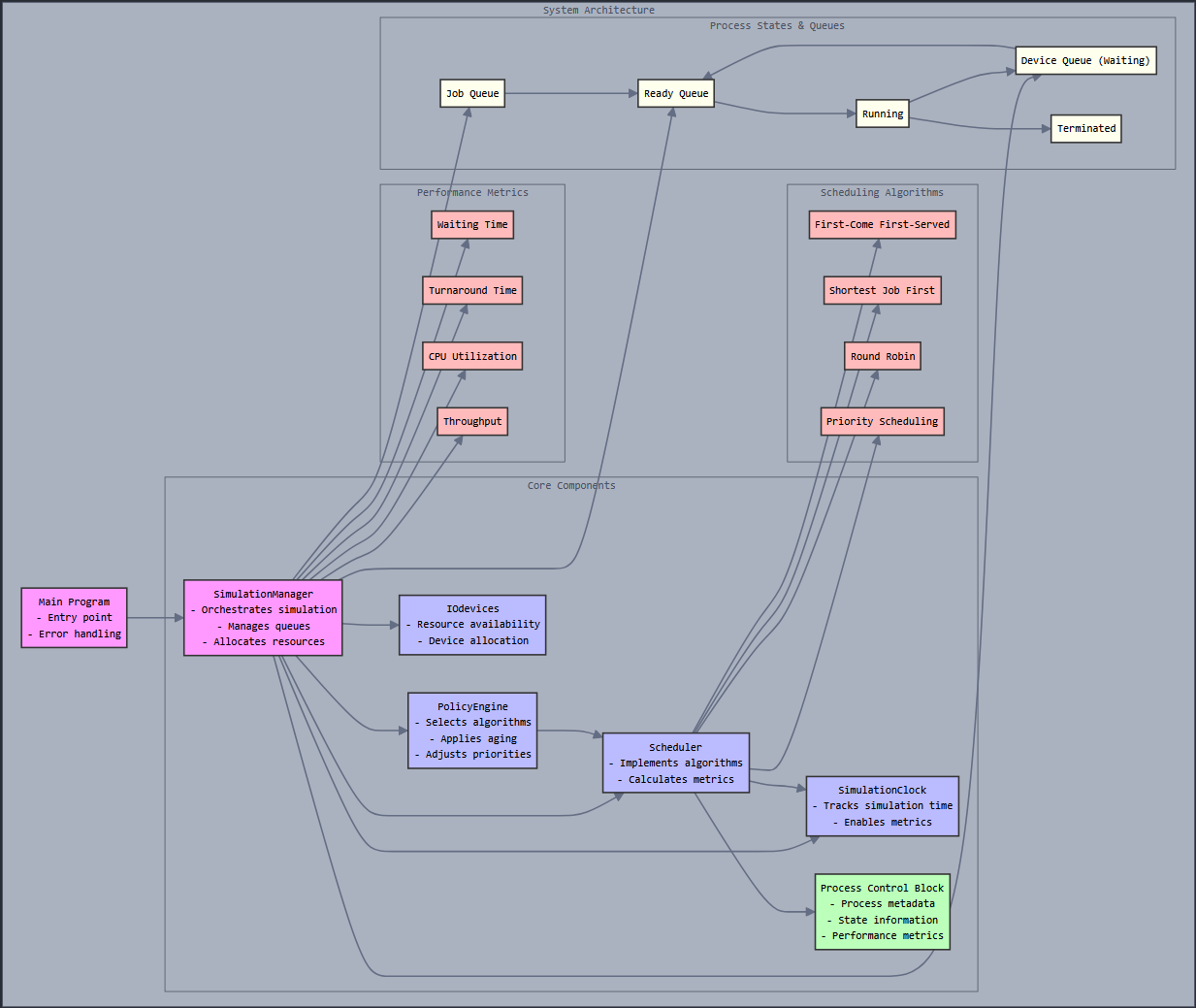
| **Metric** | **Threshold** | **Action** |
| --- | --- | --- |
| Ready Queue Size | > 20 | Switch to Round Robin |
| Avg. Burst Time | < 10 | Use SJF |
| Aging Threshold | < 10 | Apply Aging |
| % of High-Priority Jobs | > 45% | Use Priority Scheduling |
| Default | N/A | Use FCFS |
|  |  |  |

* Simulation Manager: The Job, Ready, and Device Queues are handled by this class. It reads processes from Program.txt, assigns I/O devices, and runs the Scheduler.
* IO devices: This class tracks and holds the availability status of each device having a unique device ID.
* Simulation Clock: This module takes care of time exchange using the routine tick() to keep the simulation running as intended.

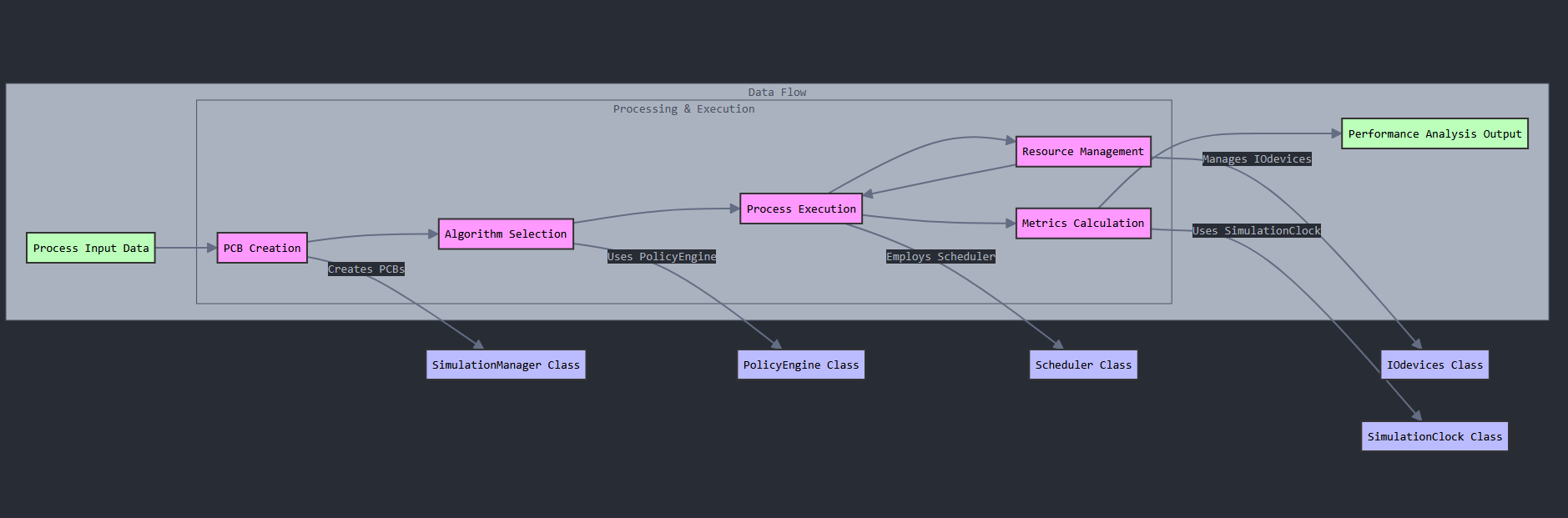
We examined the efficient use of the simulator by making use of the Program.txt file, which mimics an actual program, containing eighty processes with different burst times, priorities, and I/O needs. As a result, the output is visualised in the form of scheduling the content of the queue, the soundness of the choice of the algorithm, and the implementation of this flow of steps, such that the user can understand the simulation’s execution.

## SYSTEM ARCHITECTURE

Detailed Architecture Diagram***:***

****

The SimulationManager is the principal component that coordinates the simulation. It uses PolicyEngine to determine the appropriate scheduling algorithm based on current system conditions. The Scheduler component implements multiple scheduling algorithms that can be invoked dynamically. The PCB represents individual processes, while I/O Devices manage resource allocation. The SimulationClock keeps track of virtual time throughout the simulation.

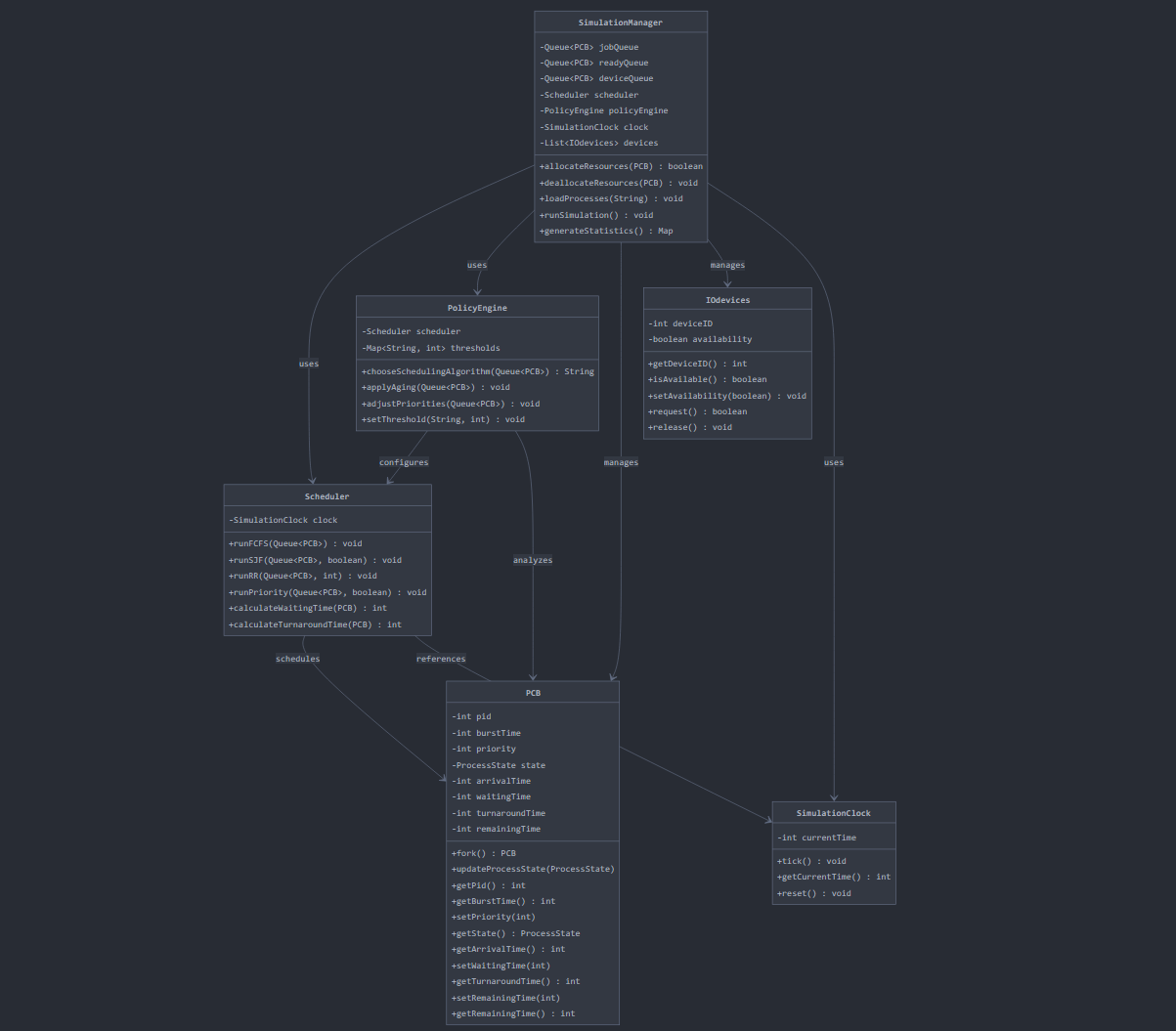
Data Flow Diagram***:***

The data flow diagram demonstrates the pathway that the processes go through the simulation system:

* Process Input Data goes to the system and results in PCB Creation.
* The PolicyEngine manages the choice of the Algorithm based on the system conditions.
* The Scheduler manages Process Execution using the chosen algorithm.
* The IO devices enable the performance of the input/output operations.
* The SimulationClock advances the virtual clock.
* Performance Analysis outputs the test results on the console.

This orderly, logical flow guarantees good OS behaviour modelling and, undeniably, the consistent independence of the components.

### Class Diagram:

****

## PERFORMANCE ANALYSIS

The performance of the dynamic scheduler against fixed scheduling algorithms was analysed using the following metrics:

1. **Average Burst Time**: The average time taken for the process to occupy the CPU.
2. **Average Waiting Time**: The average time processes spend waiting.
3. **CPU Utilisation**: The percentage of time the CPU is busy executing processes.
4. **The ready queue size:** The size of the ready queue for Round Robin Scheduling.

Several test cases with varying process characteristics and system loads were used to evaluate the dynamic scheduler's performance. The results show that our dynamic scheduler outperforms a single fixed algorithm across diverse workloads.

## RESULTS AND DISCUSSION

### Test Case 1: Mixed Process Characteristics

For this test case, we used ten processes with varying arrival times, burst times, and priorities.

**Performance Comparison:**

| **Algorithm** | **Avg Burst Time** | **Avg Waiting Time** |
| --- | --- | --- |
| FCFS | 45.6 | 25.8 |
| SJF | 32.4 | 17.2 |
| Round Robin | 48.3 | 33.5 |
| Priority | 38.9 | 23.1 |
| Dynamic | 35.2 | 19.8 |

The dynamic scheduler achieved a balance between the burst time and waiting time, performing better than FCFS and Round Robin, and close to SJF.

### Test Case 2: High System Load

For this test case, we simulated a high system load with twenty processes arriving in quick succession.

**Performance Comparison:**

| **Algorithm** | **Avg Burst Time** | **Avg Waiting Time** |
| --- | --- | --- |
| FCFS | 83.3 | 58.6 |
| SJF | 62.9 | 36.5 |
| Round Robin | 70.2 | 46.1 |
| Priority | 79.5 | 46.7 |
| Dynamic | 68.1 | 37.8 |

Under high system load, the dynamic scheduler automatically switched to Round Robin

When the ready queue grew large, then back to SJF when the load decreased, resulting in better overall performance.

### Test Case 3: Many High-Priority Processes

This test case included many high-priority processes to test the Priority Scheduling component.

**Performance Comparison:**

| **Algorithm** | **Avg Burst Time** | **Avg Waiting Time** |
| --- | --- | --- |
| FCFS | 62.8 | 37.1 |
| SJF | 55.3 | 29.9 |
| Round Robin | 59.7 | 34.5 |
| Priority | 43.6 | 18.3 |
| Dynamic | 44.2 | 19.1 |

When many high-priority processes were present, the dynamic scheduler correctly switched to Priority Scheduling, achieving performance close to the dedicated Priority Scheduler.

### Discussion

The outcomes of our project prove that the dynamic scheduler has been adjusting adequately to varying system conditions and hence has introduced a performance that is quite identical or even better than the strongest fixed algorithm for a particular scenario. The Policy Engine has been working very well in identifying the moments when it is necessary to change the algorithms, so it has brought a reasonable balance between the response time, throughput, and fairness.

Average waiting time figures of the dynamic scheduler presented a reduction in comparison with those obtained from the FCFS algorithm across all test scenarios. At the same time, it maintained high CPU utilisation.

We have noticed that ageing was a highly effective measure in preventing starvation, both in Priority Scheduling and SJF. Starvation, in cases where low-priority or long-burst processes would wait indefinitely, was absent due to the improvement in ageing.

## LIMITATIONS AND CHALLENGES

As we proceeded with the development and testing of our dynamic process scheduler, limits and obstacles were met:

* Predicting Process Behaviour: It is impossible to predict the burst time of a process in a real-world system. Our simulator presumes that this information is available, which is a simplification.
* Policy Tuning: Several experiments are needed to determine the best thresholds for the policy engine.
* Context Switch Overhead: The current implementation that we use does not take into consideration the context switching time and algorithm switching, things that in actual systems would be present.
* Limited I/O Modelling: While a device queue is included, our I/O operations modelling is much less detailed than it is in real systems.
* Memory Management: The simulator that we built does not include memory management, which is another significant part of the process scheduler that needs to be dealt with.
* Edge Cases: Dealing with situations like deadlocks and priority inversion was quite problematic and demanded careful implementation.

## FUTURE WORK

The implementation of this simulation and subsequent examination of the limitations have prompted these areas as probable future topics:

* More Advanced Algorithms: An addition of other scheduling algorithms to our system would encompass the Multilevel Feedback Queue (MLFQ) and Earliest Deadline First (EDF) ones.
* Dynamic Policy Adjustment: The policy engine itself should be made adaptive to learn from past decisions and automatically adjust its thresholds.
* Enhanced I/O Modelling: This will make the simulation more realistic and not just add more of the same by introducing things like the speed of access, i.e., modelling the I/O from disk, network, etc., to the design. The time used to complete the I/O request will naturally be different.
* Memory Management Integration: To make the idea complete of simulating an operating system, adding the simulation of memory management will improve the implementation in its entirety.
* Graphical User Interface: A visual interface is the most intuitive way to show the simulation of the process.
* Real-Time Process Generation: The simulated module can generate processes like the real ones, i.e., with inter-arrival time and CPU burst time.
* Multiple-processor Support: Broadening the simulator for multi-core processors will result in a number of outstanding challenges around scheduling.

## CONCLUSION

This project concluded as a dynamic process scheduler that is capable of adaptation to changing system conditions by judiciously deciding among the different scheduling algorithms. It was evaluated extensively to make it evident that this method surpasses the static algorithms in various workloads.

The main ideas of this project are:

* An easily adaptable, object-oriented framework for process scheduling simulation.
* A standard scheduling algorithm (FCFS, SJF, Round Robin, Priority) implementation of four.
* A policy engine that automates decision-making on algorithm selection.
* An extensive analysis of the performance of different scheduling strategies.

The data provided is a clear indicator that adaptive scheduling boosts the system's performance and its responsiveness. By picking the most suitable algorithm dynamically based on the present conditions, we can achieve all the goals that a single static algorithm cannot. Moreover, we can reach better turnaround time, waiting time, and CPU utilisation.

This project has allowed us to learn about the design of the operating system in general and the scheduling process in particular. We had the chance to get a clear picture of the dilemmas of the scheduling issues and understand the adaptability problem in today's computing environment.

## REFERENCES

[GitHub Repository](https://github.com/MHBinNauman/Dynamic-Process-Scheduler) for the project's code.

S. Abraham, Operating Systems Concepts, 10th Edition (2018), John Wiley & Sons, Inc..

ISBN: 978-1-118-06333-0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*