CSE335s
Operating
Systems

Task Scheduler Simulator

Presented to:
Dr. Sahar M. Mahmoud Haggag

CSE335s - Operating Systems
CPU Task Scheduler Simulator Project

Name	ID	Section
Abdelrahman Salah-Eldin El-Sayed	2100454	3
Ahmed Haitham Ismail	2101629	1
Ahmed Karam Abdel-Hamid	2101767	2
Marwan Ahmed Hassen	2100902	1
Mohamed Khaled Elsayed	2100675	2
Mohamed Abdelbaky Tony	2101710	3
Youssef Hany Elreweny	2101449	3

Overview

This project presents a task scheduler simulator designed to emulate the behavior of various CPU scheduling algorithms. The program is implemented in C++ using the Qt framework, combining strong backend performance with an interactive GUI. It is built as a multithreaded application—not only to enhance responsiveness and concurrency, but also to demonstrate key operating system concepts discussed in this course.

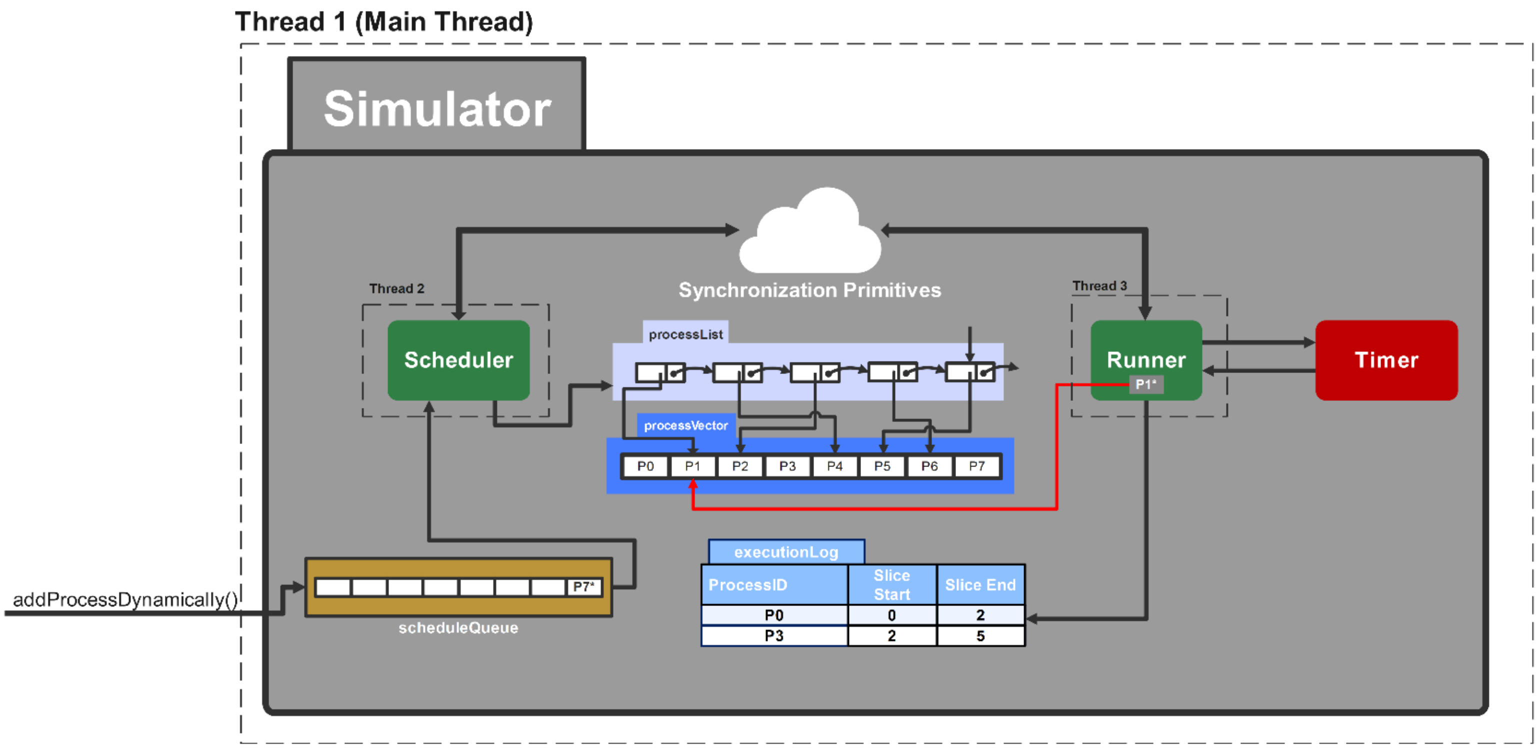
The codebase follows a modular architecture, which enabled seamless GUI integration and supported a clean, collaborative workflow among team members. The simulator supports a range of classical scheduling strategies, including First-Come-First-Served (FCFS), Shortest Job First (SJF), Priority Scheduling, and Round Robin. It provides real-time visualization through a live Gantt chart, continuous process status updates, and performance statistics such as average waiting and turnaround times.

Architecture

There are THREE main objects/players in the project:

1. The Runner, which simulates the CPU.
2. The Scheduler, which is responsible for scheduling the processes according to some algorithm.
3. The Simulator. It is the monolith object encompassing both the runner and the scheduler and contains the synchronization primitives and other state variables. It is the object that deals with the GUI.

Each of the previous entities run on a separate thread and they communicate via shared data structures and synchronize via the aforementioned primitives. The multithreaded nature of the program models the separation of concern between the players.



ProcessID	Slice Start	Slice End
P0	0	2
P3	2	5

Figure 1 - Project Architecture

An external user of the simulator (such as the GUI) will only deal with the simulator object via public methods and member variables.

The C++ backend signals updates to the GUI via the Slots & Signals feature. Signals and Slots are GUI event-handling made easy by Qt's meta object compiler (MOC). To take advantage of this feature, the simulator class must be constructed as a meta object.

How The Scheduler Works

We made the following observations:

1. A scheduling algorithm determines the relative order of execution between processes in the system.
2. Except in the case of Round-Robin, unless a new process enters the system, the relative order of execution between processes does NOT change.
3. The CPU only cares about what it should execute now.
4. In the case of preemptive scheduling, the newly arrived processes should be compared with the currently-executing process.

These observations led to the following design decisions:

1. When a new process enters into the simulation object, it is placed permanently in an array of process objects: `processVector`. Any other object would access said process via pointers.
2. We can represent this relative order of execution via a linked list. We chose to make a doubly linked list for convenience. This linked list is called the `processList.` Note that this linked list doesn't hold the actual process object, but rather pointers to them. The `processList` is thread-safe.
3. The scheduler object get involved when, and only when, a new process arrives. It is tasked with inserting the new process into the appropriate position inside the linked list.
4. The process-to-be-executed-now is the `processList` head process. When the runner wants to execute this process, it pops the process from the front of the list and starts executing it.
5. If the scheduler is preemptive, it asks the runner to put the process at hand (if it has any) back to the `processList` to ensure that it is compared with the newly arrived process. Otherwise, the scheduler inserts the new process without interrupting the runner.
6. The runner object is responsible for maintaining the `executionLog`, which is a vector of entries representing time slices (`PID, slice_start, slice_end`). This comes very handy when printing the Gantt chart (and for debugging, of course).
7. In the case of round-robin scheduling, the preemption logic due to expiry of time quantum is local to the runner object.

More on design decisions:

Why use a linked list?

The use of a linked list means that the only difference between the different schedulers is in the way they compare processes (i.e.: by arrival time? remaining burst time? priority?). Other than that, the schedulers do pretty much the same thing: insert the process inside the correct position in the list. This made the code much cleaner, shorter, and more readable.

One might object: but insertions to a linked list run in $O(n)$! Why not use a min-heap? (which supports insertions in $O(\log(n))$) The answer: the number of processes to be simulated by our program probably isn't large enough to justify the complexity introduced by using different underlying data structures. That being said, if this was the real thing, we probably would have to use different/hybrid data structures to get better performance, but unless the user of this simulator intends to insert tens of thousands of processes, this design is just fine.

How do you calculate the in-simulation time?

Initially, we relied on the system clock and C++'s `<chrono>` utilities. This worked fine when the time scale was large enough, but produced inconsistent/incorrect behavior when we tried to speed up the simulation or tried running challenging test cases (e.g.: simultaneous arrival, arrival at the end of another processes). Stress-testing showed that the system clock (`std::chrono::system_clock`) wasn't accurate enough to rely on.

In addition, the sleep provided by `std::this_thread::sleep_for()` only provided guarantees on the minimum sleep time, not maximum. This means that a thread intended to sleep a 100ms could sleep a 115ms, which is disastrous.

Ironically, the operating system itself on which the program was running (Windows) itself had limitations. The finest level of granularity you can get without resorting to OS-specific libraries is 15.6ms, and even then, it wouldn't be accurate enough to rely on.

The solution: use an in-simulation timer that represents time as experienced by the runner. The runner may sleep a 120 milliseconds, but still add just 100 milliseconds to the simulation time. The result of this is a consistent and repeatable behavior across multiple runs of the same stress-testing input. The user may not feel those extra 20 milliseconds, but over multiple time slices, they are more than enough to cause failure in the previous approach.

Was multithreading necessary?

No. Of course, the GUI and the backend had to run on different threads, but adding more threads in the backend wasn't a must. However, it was a good practice of concurrency and synchronization concepts such as mutexes, atomics, and conditional variables.

GUI and the Qt Framework

Qt's signals and systems made GUI programming lightwork. We used Qt's Creator Studio to build the UI using drag-and-drop Qt widgets. The GUI prompts the user for needed inputs only (e.g.: no need to ask for priority in a FCFS simulation). The GUI is minimal, but it works. There are other implementation details and intricacies, but they are not worth going over.

Screenshots

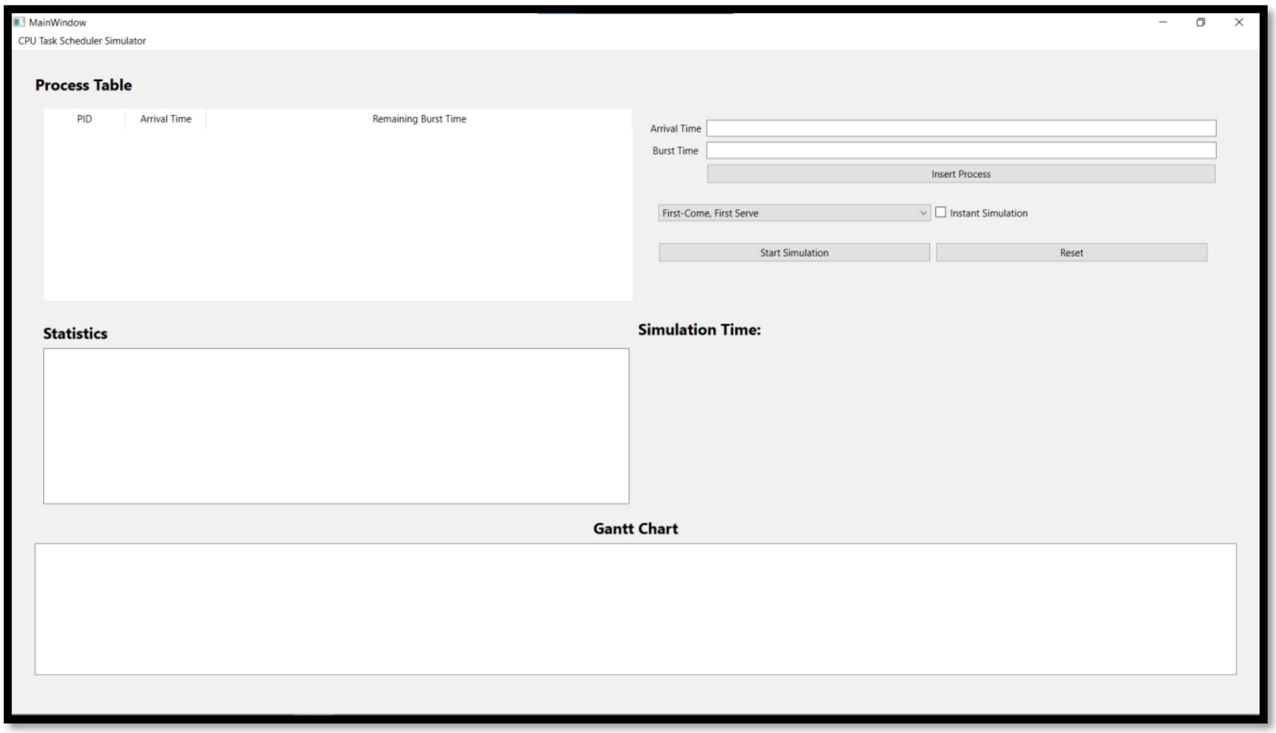
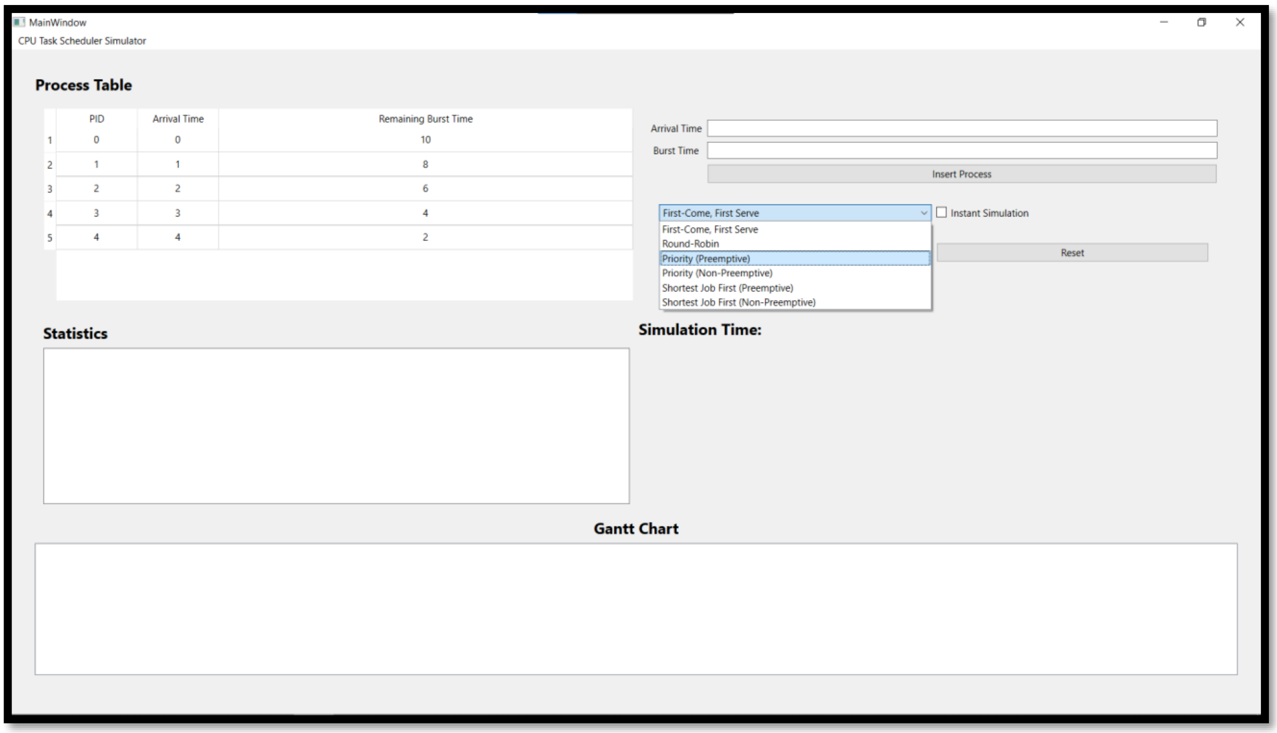


Figure 2 - Simulator UI


MainWindow
CPU Task Scheduler Simulator
Insert Process
First-Come, First Serve
Round-Robin
Priority (Preemptive)
Priority (Non-Preemptive)
Shortest Job First (Preemptive)
Shortest Job First (Non-Preemptive)
Instant Simulation
Reset
Gantt Chart

Process Table

	PID	Arrival Time	Remaining Burst Time
1	0	0	10
2	1	1	8
3	2	2	6
4	3	3	4
5	4	4	2

Arrival Time

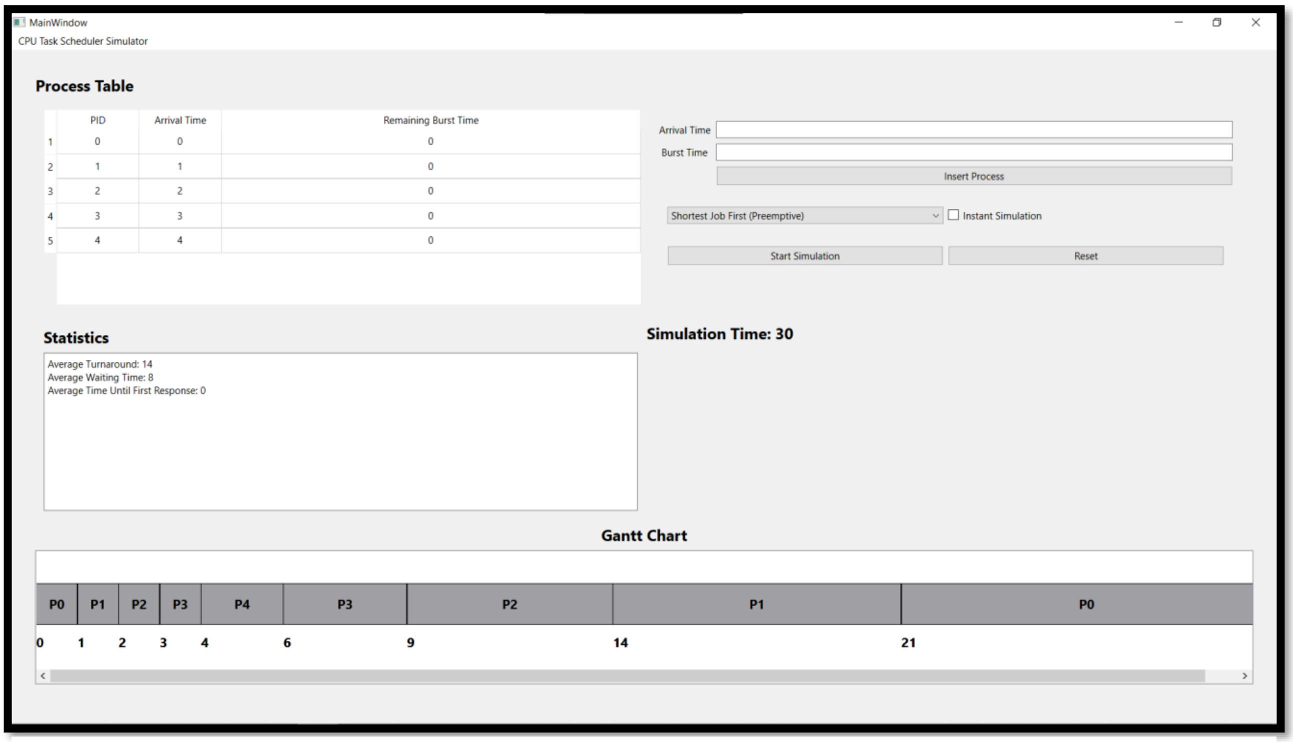
Burst Time

First-Come, First Serve

Statistics

Simulation Time:

Figure 3 - User Input



Process Table

	PID	Arrival Time	Remaining Burst Time
1	0	0	0
2	1	1	0
3	2	2	0
4	3	3	0
5	4	4	0

Arrival Time

Burst Time

Shortest Job First (Preemptive)

Statistics

```
Average Turnaround: 14
Average Waiting Time: 8
Average Time Until First Response: 0
```

Simulation Time: 30

Figure 4 - Simulation Finished

Links

- GitHub Repository
- Scheduler Simulator Installer
  (Download 1)
  (Download 2)