Multidisciplinary Project

Luca Romanò, Noah Santarossa

September 12, 2024

Contents

1 Introduction

The goal of this project is to add two new functionalities to the existing Schedsim, which simulates various task scheduling techniques. Our work involves implementing a data structure to support the insertion of new tasks and the extension of the total time in an ongoing execution. Additionally, we are developing a graphical user interface (GUI) that supports all available operations and provides a visual representation of the existing tasks.

2 Schedsim

The algorithm implements the following scheduling algorithms: FIFO, SJF, HRRN, SRTF, RR. These are divided into Non-Preemptive (FIFO, SJF, HRRN) and Preemptive (SRTF, RR) categories. In Non-Preemptive scheduling, once a task begins its execution, it will not be interrupted until it finishes. In Preemptive scheduling, tasks can start execution, pause to allow another task to execute, and then resume. Additionally, the algorithm supports tasks that can be either periodic or sporadic, with or without deadlines.

2.1 Algorithm

The algorithm processes each time instant, from the start to the end of the simulation period, and determines which task should be executed at each instant. To make this decision, the algorithm maintains the following lists:

- Arrival Events: Tasks that are ready to start their execution. These tasks enter this list when their activation time is greater than or equal to the current time;
- Finish Events: Tasks that are completing their execution;
- Deadline Events: Tasks that have missed their deadline;
- Start Events: All tasks that are eligible to be executed;

2.1.1 Non Preemptive

In Non-Preemptive algorithms, for each time instant, if no task is currently executing, the events in the Arrival Events list are sorted according to the chosen scheduling algorithm. The algorithm then selects the first task in the list to be fully executed without interruption.

2.1.2 Preemptive

In Preemptive algorithms, tasks can be started, interrupted, and then resumed. In SRTF, at each time instant, the task within the Start Events list that has the shortest remaining execution time is selected for execution. In RR, at each quantum time instant, the task to be executed is switched in a cyclic manner, following a repeated round of the tasks in the Start Events list.

3 Adding functionalities

The scope of the project is to add the following functionalities:

- Adding Tasks and Time: Support the insertion of tasks after the execution has started, without the need to recalculate everything, and provide the ability to extend the simulation's end time;
- Graphical Interface: Implement a GUI that allows users to interact with the scheduling algorithm and provides a visual representation of the tasks being executed throughout the simulation;

Here we present the web software architecture scheme, showcasing how the user interface connects and interacts with various system components. This diagram highlights the seamless communication between the frontend and backend,

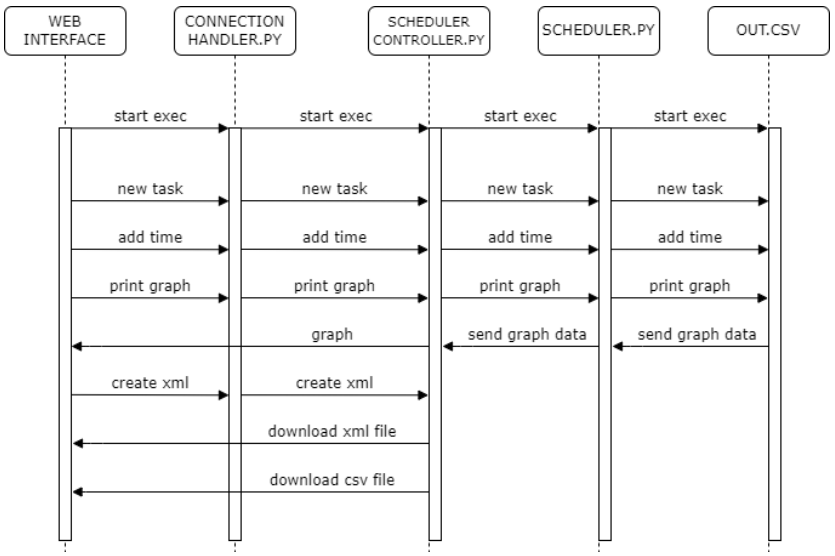


Figure 1: The web application architecture scheme

3.1 Adding Task and Time

To support the addition of a new task after the execution has started, various solutions have been considered. For periodic tasks, it is impossible to add a task without recalculating everything from the beginning, as the start time of each periodic task is tied to the start time of the scheduling. Therefore, the optimizations described below focus on the addition of sporadic tasks.

3.1.1 Basic Solution

The basic solution involves saving a copy of the four lists (*Arrival Events*, *Finish Events*, *Deadline Events*, *Start Events*) at each time instant. When a new task is added, the algorithm would return to the state corresponding to the start time of the new task and continue the execution from that point. However, this solution is not optimal because saving a copy of the four lists at each time instant is computationally expensive. Therefore, a more efficient solution is needed-one that does not require saving the state of the lists at every time instant, but only at certain intervals.

3.1.2 Balanced Binary Search Tree Solution

An alternative solution is to use a Balanced Binary Search Tree (BBST) to save a copy of the four lists only at time instants when a new task begins execution. With this data structure, fewer copies of the lists are needed, and when a new task is added, the time to restart the execution can be found in $O(\log(\text{total time}))$. However, this solution is less effective when tasks have short durations, or when the quantum value in Round Robin scheduling is low, leading to frequent updates of the BBST and, consequently, too many copies of the lists.

Example:

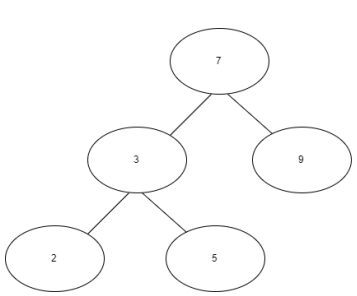
Time = 10

We suppose that at the following time instants {2, 3, 5, 7, 9}, the tasks in "execution" change, and the tree is updated before continuing execution.

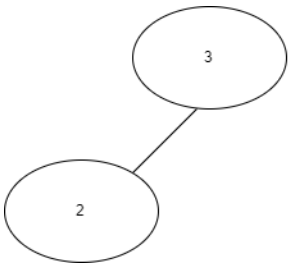
Now, consider a new task that starts at time 4. The system finds the largest element in the BBST (Balanced Binary Search Tree) that is smaller than or equal to 4, which is 3.

The system updates the BBST structure, restores the state from time 3, and resumes the simulation from that point until the end.

If we consider round-robin execution, the tree size remains constant at one, as the tree is updated with each time slice.



(a) Example of data structure



(b) Example of new task starts at 4

Figure 2: Example of BBST structure

3.1.3 SQRT Decomposition Solution

Taking the previous solutions into consideration, we opted for a data structure that saves a copy of the four lists at regular intervals. We chose an approach based on the *Square Root Decomposition Algorithm*. In this method, a copy of the four lists is saved every $\sqrt{\text{time duration}}$. When a new sporadic task arrives, the algorithm uses the most recent saved state prior to the task's start time and then simulates the execution from that point to the end.

Example:

Time = 10

Size = $\sqrt{10} \approx 3$ (rounded to the nearest integer)

eventslist = {events[0], events[3], events[6], events[9]}, where each *events* list contains four items

At time 4, a new task is introduced.

The most recent saved state before time 4 is at time 3.

The system restores the state at time 3 and resumes the simulation from there until the end.

(a) Example of eventlist

(b) Example after add at time 4

Figure 3: Example of SQRT Decomposition structure

3.1.4 Implementation

When the algorithm starts executing, the interval size is calculated as the square root of the total execution time of the simulation, determining how often the states of the lists will be saved. The algorithm then simulates the tasks from the input file, saving the state of the four lists starting at time 0 and at each interval determined by the size. For example, if the total duration is 10, the size will be 3, and the positions that will be saved are 0, 3, 6, and 9. When a new task arrives, if it is sporadic, the time instant at which the execution will be restarted is calculated. This time is the latest saved time instant that is smaller than or equal to the start time of the new task. The execution is then restarted from that time instant and proceeds as before. If additional time is added to the total duration, the list containing the arrival events needs to be updated. The new arrival events are calculated similarly to the initial ones and saved from the previous end time up to the new end time, which is equal to the previous end time plus the added time.

3.1.5 Test the correctness of the implementetion

To test the correctness of the implementation, we created a script that generates two input files. The second file contains a subset of the information from the first one, with a shorter time duration and fewer tasks. The missing tasks in the second file are then added, and the total time is extended to match that of the first file. This approach is used to verify the correct functioning of the method for

adding tasks and time. After execution, the two output files are compared, and our implementation consistently passed the test, producing identical files each time.

3.2 Graphical Interface

The goal of this project is to create a new user interface (UI) that allows for easy access to and utilization of the newly implemented functionalities. We have developed a web-based UI that can be launched using the `ConnectionHandler.py` script.

3.2.1 Running the Application

To run the application, follow these steps:

1. Open the `Schedsim3` folder on your device;
2. Navigate to the `visualizer` directory using your terminal or command prompt;
3. Compile and run the `Connection_Handler.py` file using Python. This will start the server that hosts the web interface;
4. Open a web browser and go to the URL `http://127.0.0.1:5001/` to access the user interface;
5. Interact with the "Schedsim3" GUI to utilize the scheduling features and visualize task execution;

The homepage of the UI looks like this:

Figure 4: The Schedsim Interface

We have created several buttons linked to the key features of Schedsim

3.2.2 Start Button

This button allows you to execute the code using the `execute` function of the `Scheduler` class. The function utilizes the `self.scheduler` instance to start and terminate the execution. If `self.scheduler` is not present, the function throws an exception and quits.

3.2.3 New Task Button

This button allows you to create a new task using the `new_task` function of the `Scheduler` class, which requires several parameters defined in the `Task` class. There are two types of tasks:

- **Sporadic**: Requires the parameters `(real_time, task_type, task_id, activation, deadline, wcet)`;
- **Periodic**: Requires the parameters `(real_time, task_type, task_id, period, deadline, wcet)`;

There are several controls

- **Task ID Check:** Verify that `task_id` is a positive integer. This ensures that only valid tasks are processed;
- **WCET Check:** Confirm that `wcet` (Worst-Case Execution Time) is a positive integer. This check ensures that the execution time provided is valid and non-zero;
- **Period and Activation Check:** For periodic tasks, ensure `period` is a positive integer. For sporadic tasks, ensure `activation` is non-negative. This prevents invalid or nonsensical timing values from being used;

- **Deadline Check:** Ensure `deadline` is a positive integer. This ensures that deadlines are set correctly and are valid;
- **WCET vs Period Check:** For periodic tasks, ensure `wcet` is less than or equal to `period`. This ensures that the execution time fits within the period allocated for the task;
- **Deadline vs WCET Check:** Ensure `deadline` is greater than `wcet`. This guarantees that the task has enough time to complete before its deadline;

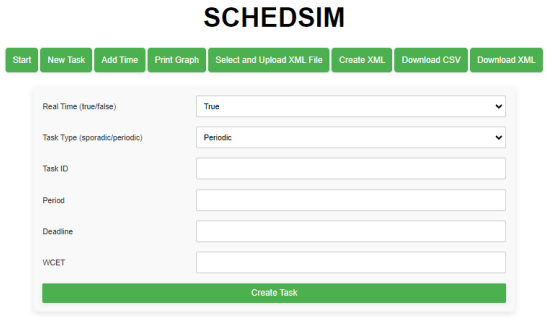


Figure 5: The interface of the New Task button

3.2.4 Add Time Button

This button allows you to use the `add_time` function of the `Scheduler` class. This function requires a numerical input, which adds the specified amount of time to the `scheduler.end` parameter. When this function is activated, the scheduler implicitly calls the `start` function.

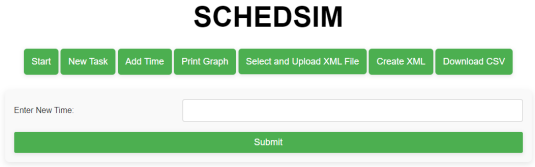
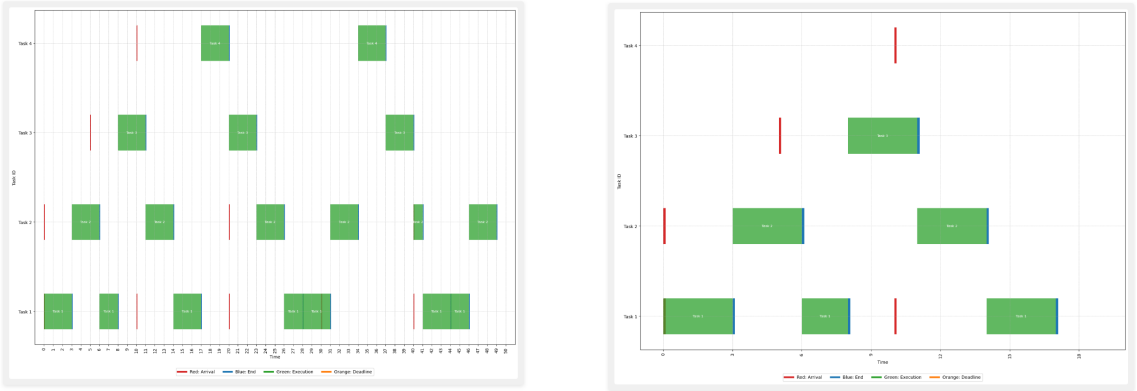


Figure 6: The interface of the Add Time button

3.2.5 Print Graph Button

This button allows you to use the `create_graph` function, which automatically prints the graph with default parameters: fraction set to one, and start and end set to the respective parameters of the `scheduler` instance. An additional functionality allows you to define the following parameters (`start, end, fraction`) to print the graph for a specific interval (usually the most significant). The fraction is a number in the range (1,5), while the start and end are allowed only in the range [`scheduler.start, scheduler.end`]. The graph displays four types of data:

- *Red* for the **arrival** of tasks;
- *Blue* for the **completion** of tasks;
- *Green* for the **execution** of tasks;
- *Orange* for the **deadline** of tasks;



(a) Example of a default graph

(b) Example of a formatted graph

Figure 7: Example of graph with "example_rr.xml"

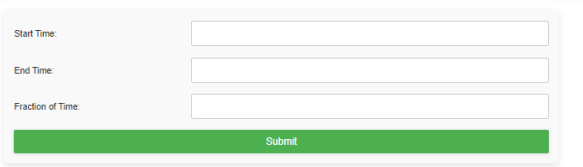


Figure 8: The interface of the Print Graph button

3.2.6 Select and Upload XML File Button

This button allows you to select and upload an XML file from your device to the temporary directory of the project. Controls are in place to prevent uploading files that are not in XML format.

3.2.7 Create XML Button

This button allows you to create a new XML file based on the input parameters (`start`, `end`, `scheduler algorithm`). The `RR` algorithm also requires an additional parameter, `quantum`, which defines the time to assign to each task during each cycle.

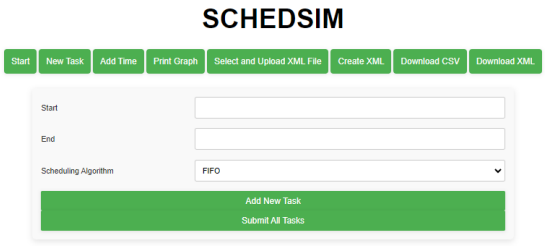


Figure 9: The interface of the Create XML button

Through the dynamic `Add New Task` button, you can add as many tasks as needed, following the same concept as the `New Task` functionality. The `Submit All Tasks` button finalizes the creation of the XML file, which you can then download using the `Download XML` button. To verify the validity and relationships among the tasks, we use the checks detailed in Section 3.2.3.

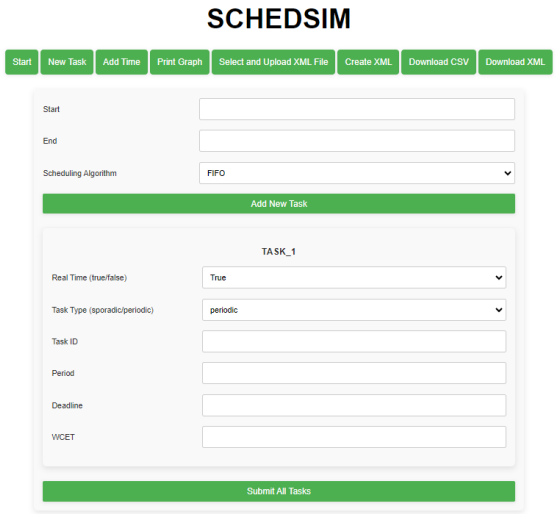


Figure 10: The interface of Create XML with an example task

3.2.8 Download CSV Button

This button allows you to download the output of the execution in CSV format. You can only download the file if you have executed the code beforehand.

timestamp	task	job	processor	type_of_event	extra_data
0	1	1	0	A	0
0	2	1	0	A	0
0	1	1	0	S	0
3	1	1	0	F	0
3	2	1	0	S	0
5	3	0	0	A	0
6	2	1	0	F	0
6	1	1	0	S	0
8	1	1	0	F	0
8	3	0	0	S	0
10	1	2	0	A	0
10	4	0	0	A	0
11	3	0	0	F	0
11	2	1	0	S	0
14	2	1	0	F	0
14	1	2	0	S	0
17	1	2	0	F	0
17	4	0	0	S	0
20	1	3	0	A	0
20	2	2	0	A	0
20	4	0	0	F	0
20	3	0	0	S	0

Figure 11: Example of csv file with "example_rr.xml"

3.2.9 Download XML Button

This button allows you to download the created XML file. You can only download the file if it was previously created using the `Create XML` button.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<simulation>
        <time start="0" end="50" />     <!-- Simulation time -->
    <software>
        <tasks>
                <!-- A task can be periodic or sporadic. The deadline parameter is relevant only for
                     real-time tasks -->
            <task real-time="true" type="periodic" id="1" period="10" deadline="50"  wcet="5" />
            <task real-time="true" type="periodic" id="2" period="20" deadline="30" wcet="10" />
            <task real-time="true" type="sporadic" id="3" activation="5" deadline="50"  wcet="20" />
            <task real-time="false" type="sporadic" id="4" activation="10" wcet="15" />
        </tasks>
        <scheduler algorithm="RR" quantum="3" />        <!-- add any relevant numbers, such as quantum time for RR -->
    </software>
    <hardware>
        <cpus>
                <pe id="0" speed="1" /> <!-- Speed represents the multiplicator for the wcet. For instance
                                            A task with wcet=10 running of a core with 1.25 speed it takes
                                             8 time units to execute -->
        </cpus>
    </hardware>
</simulation>
```

Figure 12: Example of xml file "example_rr.xml"

3.3 Support file

For use well the GUI we have develop many support file

3.3.1 SchedulerController

This class manages the interaction between the frontend and backend using an instance of a scheduler. Its functions are:

- **Initialization**: Sets up the scheduler, temporary directories, and initializes start and end times;
- **load_xml_file**: Loads an XML file to initialize the scheduler with simulation parameters;
- **execute_scheduler**: Runs the scheduler and finalizes its execution;
- **create_task**: Adds a new task to the scheduler, supporting both sporadic and periodic tasks;
- **add_new_time**: Extends the simulation time by a specified amount.
- **print_graph**: Generates and saves a graph representing task execution over a specified time range;
- **create_xml**: Creates an XML file representing the simulation configuration, including tasks, scheduler, and hardware;

3.3.2 Visualizer

The `create_graph` method produces a timeline visualization of task events. It uses Pandas to handle and process numerical data from a CSV file, including managing missing values and converting data types. The method filters the data according to `start_time` and `end_time`, and utilizes Matplotlib to generate a graphical representation. The graph illustrates task arrivals, executions, completions, and deadlines with distinct colors. If the data is missing or empty, an empty graph is generated. The resulting plot is saved as the image file "`out.png`".

3.3.3 Style

The `style.css` file defines the visual presentation of the web page elements. It includes rules for layout, color schemes, fonts, and other stylistic aspects to ensure a cohesive and aesthetically pleasing user interface.

3.3.4 Script

The `script.js` file contains JavaScript code that handles various interactions and dynamic behaviors on the web page. It manages event listeners for user actions such as button clicks and form submissions, communicates with the server via AJAX, and updates the user interface based on server responses. This script is essential for enabling interactive features and ensuring smooth functionality.

3.3.5 Dynamic_form

The `dynamic_form.js` file manages the dynamic behavior of the form used for creating XML files. It adjusts the visibility of form fields based on user selections (e.g., task type or scheduling algorithm) to present only relevant options. This script ensures that the form adapts in realtime to user input, improving usability and streamlining the process of configuring XML files.

4 Conclusion

With our project, users can now easily interact with the Schedsim Simulator through the web interface we developed, allowing them to visualize a graphical representation of the results. Additionally, the system supports the addition of new tasks and the extension of the total duration without requiring a full recalculation.

References

[1] HEAPLab, Schedsim GitHub Repository, https://github.com/HEAPLab/schedsim, 28 April 2022.

[2] Noah Santarossa e Luca Romanò, Schedsim3 GitHub Repository, https://github.com/NoahSantarossa/schedsim3, 5 September 2024.

[3] GeeksforGeeks, Binary Search Tree (BST) Data Structure, https://www.geeksforgeeks.org/binary-search-tree-data-structure/, Accessed September 2024.

[4] GeeksforGeeks, Square Root (Sqrt) Decomposition Algorithm, https://www.geeksforgeeks.org/square-root-sqrt-decomposition-algorithm/, Accessed September 2024.

[5] GeeksforGeeks, Python Lists, https://www.geeksforgeeks.org/python-lists/, Accessed September 2024.

[6] GeeksforGeeks, Process Schedulers in Operating System, https://www.geeksforgeeks.org/process-schedulers-in-operating-system/, Accessed September 2024.