Priority Based Task Scheduling Algorithm for an Operating System

Date – 2/22/2025

Sadhana Jarag

# 

**Table of Contents**

[Abstract 2](#_Toc191143766)

[Introduction 3](#_Toc191143767)

[Assumptions 4](#_Toc191143768)

[Non – Functional Requirements 4](#_Toc191143769)

[Methodology 5](#_Toc191143770)

[Abstractions 6](#_Toc191143771)

[Level 1: High-Level Abstraction 6](#_Toc191143772)

[Level 2: Intermediate Abstraction 6](#_Toc191143773)

[Level 3: Low-Level Abstraction 7](#_Toc191143774)

[UML Diagrams 9](#_Toc191143775)

[Activity Diagram : 9](#_Toc191143776)

[Class Diagram 10](#_Toc191143777)

[Sequence Diagram 11](#_Toc191143778)

[Implementation 12](#_Toc191143779)

[Code Execution 12](#_Toc191143780)

[Code Output: 12](#_Toc191143781)

[Conclusion 13](#_Toc191143782)

[References 14](#_Toc191143783)

# Abstract

This project designs and implements a simple priority-based task-scheduling algorithm for an operating system. Using stepwise refinement, the problem is broken into manageable components: task initialization, scheduling, execution, and completion. Implemented in Python with object-oriented principles, the algorithm ensures higher-priority tasks execute first while dynamically updating the task queue. This project demonstrates fundamental concepts of task management, priority scheduling, and system simulation in operating systems.

# Introduction

Task scheduling is a fundamental component of operating systems, responsible for managing the execution of processes or tasks in an efficient and orderly manner. The goal of task scheduling is to optimize resource utilization, ensure fairness, and meet the requirements of different tasks based on their priorities. In this project, we develop a simple task-scheduling algorithm that simulates how an operating system might manage and execute tasks. The algorithm prioritizes tasks based on their priority levels and executes them in a systematic manner, ensuring that higher-priority tasks are completed first.

By using a stepwise refinement approach, the project breaks down the problem into manageable components, such as task initialization, scheduling, execution, and completion. This implementation, done in Python, demonstrates the use of object-oriented programming to model tasks and their attributes, providing a clear and modular structure. The project serves as a foundational example of task scheduling in operating systems, highlighting key concepts such as priority-based scheduling, task management, and system simulation. It also lays the groundwork for future enhancements, such as preemptive scheduling or support for more complex task dependencies.

# Assumptions

For the task-scheduling algorithm project, several assumptions have been made to ensure the project is both practical and achievable. Below is a list of key assumptions taken for this project:

1. **Single CPU System** – Only one task runs at a time.
2. **Priority-Based Scheduling** – Tasks are scheduled based on priority, where a lower number value means a higher priority.
3. **No Preemption** – Tasks cannot be interrupted once they start execution.
4. **Fixed Task Duration** – The duration of each task is fixed and known in advance.
5. **No Task Dependencies** – Tasks are independent of each other, and no task depends on the completion of another.

# Non – Functional Requirements

Task-scheduling algorithm project, here are the top 5 non-functional requirements:

1. **Performance** – The system should schedule and execute tasks efficiently
   1. The scheduler should be able to handle a large number of tasks (e.g., 1,000+ tasks) without significant delays.
   2. Task execution and scheduling should complete within a reasonable time frame (e.g., milliseconds per task).
2. **Scalability** – The system should scale to accommodate an increasing number of tasks
   1. The algorithm should perform well even as the number of tasks grows.
   2. The system should be able to handle dynamic task arrivals (if extended in the future).
3. **Reliability** – The system should consistently execute tasks without error.
   1. The scheduler should not crash or fail during task execution.
   2. Tasks should be executed in the correct order based on priority.
4. **Maintainability** – The code should be easy to maintain and extend.
   1. The code should follow best practices
   2. The system should be easy to modify to support additional features
5. **Usability** – The system should be easy to use and understand.
   1. The output should be clear and informative, showing the status of tasks and the system clock.
   2. The system should provide logs or visualizations (e.g., diagrams) to help users understand the scheduling process.

# Methodology

1. **Stepwise Refinement:** Break down the problem into smaller, manageable steps (e.g., initialize system, load tasks, schedule tasks, execute tasks).
2. **Object-Oriented Programming (OOP):** Use classes (e.g., Task) to model system components, making the code modular and reusable.
3. **Iterative Development:** Build the system incrementally, test each step, and refine it based on feedback.
4. **Functional Decomposition:** Divide the system into smaller functions (e.g., initialize\_system, schedule\_tasks) for clarity and maintainability.
5. **5. Simulation-Based Approach:** Simulate time and task execution using a system clock and task queue.

# Abstractions

Using a stepwise refinement approach, we will develop a simple task-scheduling algorithm for an operating system. This approach involves breaking down the problem into smaller, more manageable subproblems and refining each subproblem until we reach a level of detail that can be directly implemented in code.

## Level 1: High-Level Abstraction

At this level, we define the overall structure of the task-scheduling algorithm without going into implementation details.

**Algorithm:**

1. Initialize the system.
2. Load tasks into the task queue.
3. Schedule tasks based on priority.
4. Execute tasks.
5. Repeat until all tasks are completed.

## Level 2: Intermediate Abstraction

At this level, we refine each step from Level 1 into more detailed sub-steps.

**Algorithm:**

1. **Initialize the system:**
   1. Create a task queue to hold tasks.
   2. Initialize the system clock.
2. **Load tasks into the task queue:**
   1. Read tasks from a file or user input.
   2. Assign priorities to tasks.
   3. Add tasks to the task queue.
3. **Schedule tasks based on priority:**
   1. Sort tasks in the queue by priority.
   2. Select the highest-priority task for execution.
4. **Execute tasks:**
   1. Run the selected task.
   2. Update the task's status (e.g., completed, running, waiting).
   3. Remove completed tasks from the queue.
5. **Repeat until all tasks are completed:**
   1. Check if the task queue is empty.
   2. If not empty, go back to step 3.

## Level 3: Low-Level Abstraction

At this level, we provide detailed steps that can be directly translated into code.

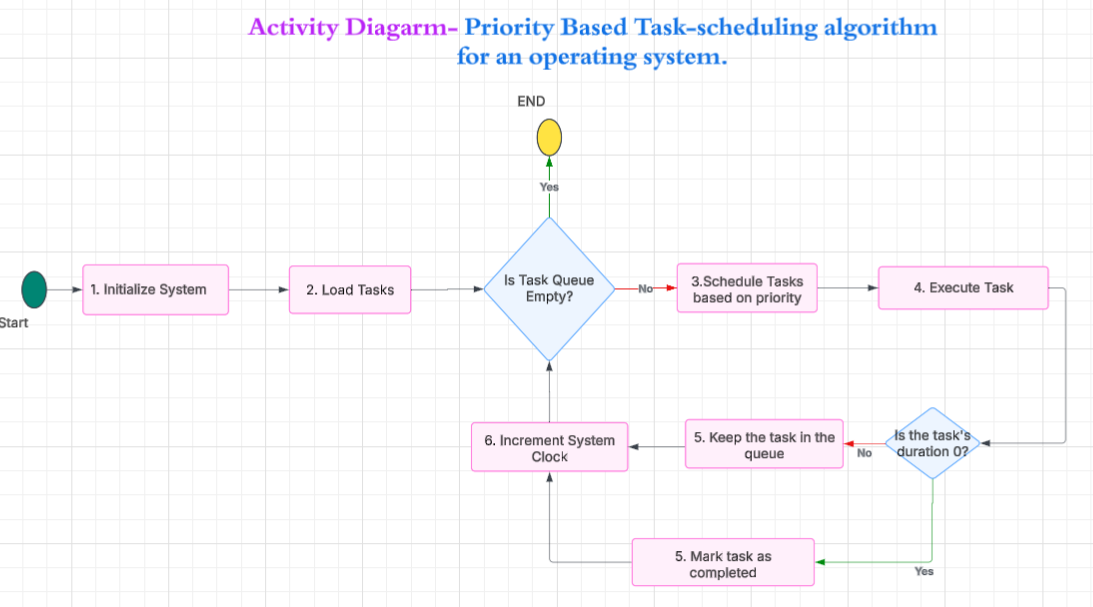
**Algorithm:**

1. **Initialize the system:**
   1. Create an empty list to act as the task queue.
   2. Set the system clock to 0.
2. **Load tasks into the task queue:**
   1. Define a Task class with attributes: id, priority, duration, and status.
   2. Read tasks from a file or user input and create Task objects.
   3. Add tasks to the task queue.
3. **Schedule tasks based on priority:**
   1. Sort the task queue by priority in descending order.
   2. Select the first task in the sorted queue.
4. **Execute tasks:**
   1. Set the selected task's status to "running".
   2. Simulate task execution by decrementing the task's duration.
   3. If the task's duration reaches 0, set its status to "completed" and remove it from the queue.
5. **Repeat until all tasks are completed:**
   1. Check if the task queue is empty.
   2. If not empty, repeat steps 3–5.

# UML Diagrams

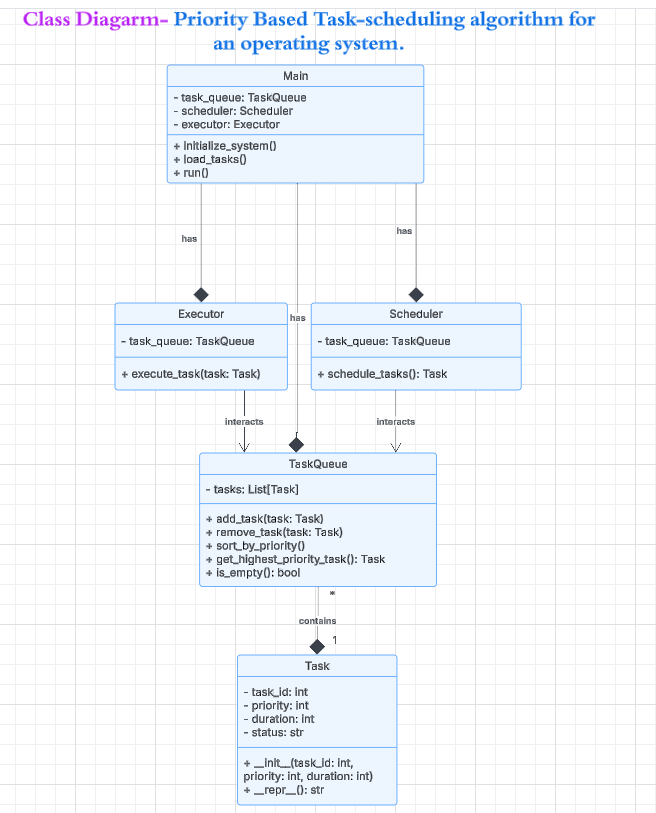
## Activity Diagram:

An activity diagram is a UML diagram that represents the flow of activities or processes in a system. For the task-scheduling algorithm, the activity diagram will illustrate the sequence of steps from initializing the system to completing all tasks.



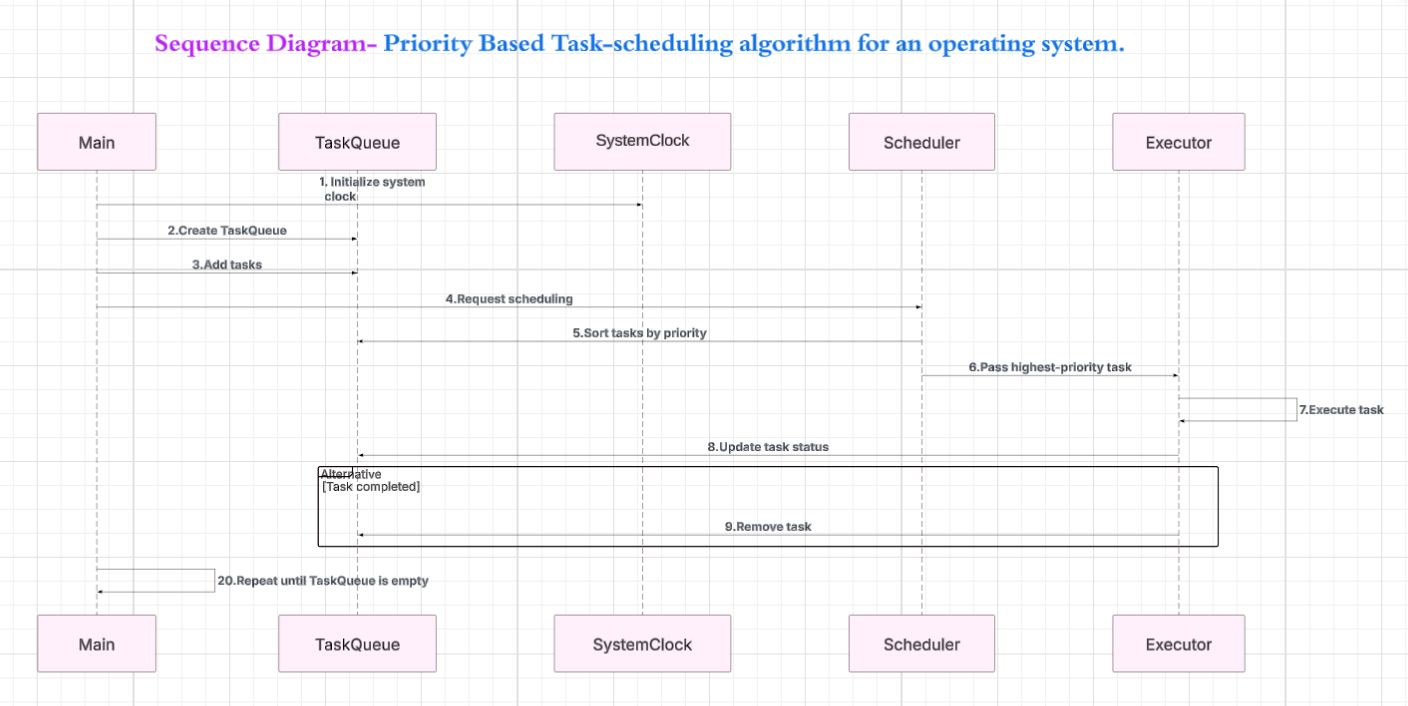
## Class Diagram:

In the class diagram for the task-scheduling algorithm, the Main class has relationships with three other classes: TaskQueue, Scheduler, and Executor. These relationships are composition relationships because the Main class owns and manages the lifecycle of these objects. If the Main class is destroyed, the TaskQueue, Scheduler, and Executor objects are also destroyed.



## Sequence Diagram:

Sequence diagram for the task-scheduling algorithm, we need to visualize the interaction between the different components of the system over time. A sequence diagram focuses on the order of messages or method calls between objects or components.



# Implementation

## Code Execution:

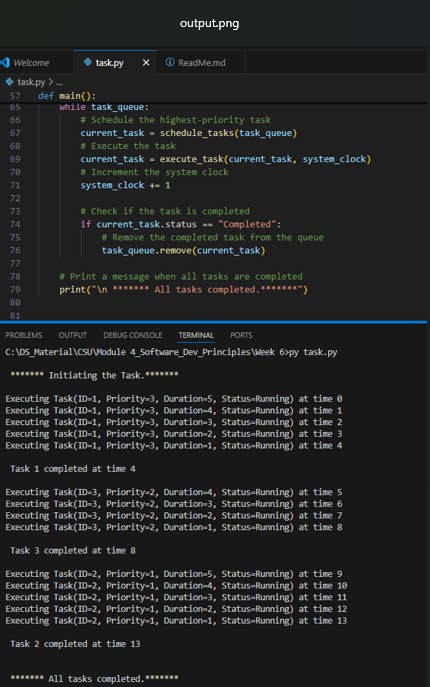
1. The implementation is done in Python. Below is the code.



1. Git Hub Link – <https://github.com/sadhanajarag/Week-5.git>

## Code Output:

The code is executed, and the output is captured as shown below:



# Conclusion

In conclusion, this project successfully demonstrates a simple task-scheduling algorithm using a stepwise refinement and object-oriented approach. By breaking down the problem into manageable components and simulating task execution, the system efficiently schedules and executes tasks based on priority. The use of Python ensures clarity and flexibility, making the implementation easy to understand and extend. This project serves as a foundational example of task scheduling in operating systems, with potential for future enhancements like preemption or multi-level queues.

# References

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.
2. Python Software Foundation. (n.d.). *Python Documentation*. Retrieved from <https://docs.python.org/3/>
3. Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd ed.). Addison-Wesley.
4. Lucid Software Inc. (n.d.). *Lucidchart: Diagramming and Visualization*. Retrieved from <https://www.lucidchart.com/>
5. Kopetz, H. (2011). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer.