# Rate Monotonic Scheduling

MOHAMMAD ASHRAFUZZAMAN SIDDIQI
*Electronic Engineering*
*University of Applied Sciences Hamm Lippstadt*
Dortmund, Germany
mohammad-ashrafuzzaman.siddiqi@stud.hshl.de

***Abstract*—This paper presents a comprehensive study of Rate Monotonic Scheduling (RMS), a widely-used fixed-priority algorithm for real-time systems. We explore the theoretical foundations of RMS, provide detailed modeling using UPPAAL, and demonstrate practical implementation in C. The UPPAAL model verifies the correctness of RMS by simulating task execution and preemption, ensuring tasks meet their deadlines. The C implementation validates the algorithm's practical application, showing how tasks with shorter periods preempt those with longer periods. Results confirm the efficacy of RMS in managing real-time tasks, highlighting its importance in systems with stringent timing requirements.**

## I. Introduction

Real-time systems have become an integral part of computing and fundamentally imply a system in which time is a crucial parameter. Its applications range from automotive to aerospace controls. Any system in which deterministic behaviour or a time constant exists is a real-time system [2]. A real-time system works by taking in input and tasks and setting a time constraint for execution; these time constraints can be referred to as deadlines [1].

Real times systems typically exists in two forms:

- **Hard Real time systems** : A hard real-time system is one in which the time constraints are stringent and of the highest priority. Missing a deadline in a hard real-time system is considered a system failure, while meeting the deadline is considered a success for the system's performance, correctness, and safety [3]. A hard real-time system has the highest level of complexity, and the behaviour must be predictable according to the timing requirements set. An example of this can be seen in industrial automation, like in an assembly line where certain actions must take place at certain time intervals to ensure the successful operation of the system. [1].
- **Soft Real time systems**: Soft real-time systems are ones in which missing a deadline is undesirable but not disastrous. The system can handle occasional deadline misses, and performance degradation is acceptable in these cases [4]. It is a system characterized by flexible deadlines but does not imply that deadlines are 100 percent unimportant, it simply means missing a few will result in a reduced quality. An example can be multimedia streaming, where connection loss or buffering can occur for some seconds but the overall system can still function at a reasonable level [4].

Identify applicable funding agency here. If none, delete this.

### *A. Rate Monotonic Scheduling:*

This is a popular, widely used fixed-priority algorithm. It depicts a scheduling system in which priority is assigned to different tasks based on the periodicity of the task; a shorter period leads to a higher priority. The rate of a task is inversely proportional to the period of the task in this system; this means that the higher the rate, the higher its priority [4].

The rate-monotonic scheduling system operates in a system where tasks are assigned at different periods and have static priorities. This algorithm is designed to provide a predictable scheduling framework, ensuring that the most critical timing demands are met, thereby being of use in hard real-time systems where missing a single deadline could present a costly failure [4].

In this paper, we will delve into the foundations and description of rate monotonic scheduling. We will explore its optimally proofs, analyse its runtime and overhead, and provide implementation examples. Furthermore, we will demonstrate how tools like Uppaal and C programming can be utilised to model, verify, and implement RMS in real-world applications [4].

### *B. Example of Rate Monotonic Scheduling (RMS)*

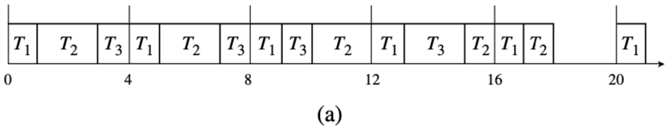


Fig. 1. RMS Example [4]

*1) Task Definitions:* The example consists of three tasks with the following attributes:

- **T1:** Period = 4, Execution Time = 1
- **T2:** Period = 5, Execution Time = 2
- **T3:** Period = 20, Execution Time = 5

*2) Priority Assignment:* In RMS, tasks are assigned priorities inversely proportional to their periods:

- **T1** has the highest priority because it has the shortest period (4 units).
- **T2** has a medium priority with a period of 5 units.
- **T3** has the lowest priority with a period of 20 units.

*3) RMS Schedule:* Figure 1 shows the RMS schedule for these tasks over a 20-unit time frame. The schedule follows these rules:

- **T1** executes at times 0, 4, 8, 12, and 16 because it has the highest priority.
- **T2** is scheduled at times 1-3 but is preempted by **T1** at times 4 and 16.
- **T3** runs when neither **T1** nor **T2** are running, which happens at times 3-4, 5-7, 9-12, and 13-15.

*4) Analysis:* RMS ensures that higher priority tasks (those with shorter periods) are always executed first. This scheduling method guarantees that tasks meet their deadlines.

## II. Mathematical Basics of Rate Monotonic Scheduling (RMS)

### A. Necessary Mathematical Basics

Rate Monotonic Scheduling (RMS) relies on several key mathematical concepts to ensure that tasks are scheduled correctly and meet their deadlines. These concepts include:

- Tasks: A task is a unit of work that must be scheduled and executed by the system. Each task $\tau_i$ has specific attributes such as period $T_i$, execution time $C_i$, and deadline $D_i$. In RMS, the deadline is typically equal to the period ($D_i = T_i$) [4].
- Periods: The period $T_i$ of a task is the interval between consecutive releases of that task. For instance, if a task has a period of 4 units, it will be released every 4 units of time [4].
- Deadlines: The deadline $D_i$ is the latest time by which a task must be completed after it is released. In RMS, deadlines are usually equal to the periods of the tasks [4].
- Preemption: A higher-priority task preempts a lower-priority task if it becomes ready to execute [6]

### B. Formal (Mathematical) Description of the Topic

RMS assigns priorities to tasks based on their periods: the shorter the period, the higher the priority. The key formal concepts include:

*1) Priority Assignment:* Each task $\tau_i$ is assigned a priority inversely proportional to its period $T_i$. This means tasks with shorter periods have higher priorities [4].

*2) Schedulability Test:* The RMS schedulability test is a test developed to determine if a set of tasks can meet their deadlines. The concept of a utilization bound is important to analyze the schedulability of a task. Before explaining the utilization bound and how we then analyze the schedulability of a system, we need to understand what utilization is.

Utilization is a measure of the amount of a processors capacity used by a set of task. Utilization is calculated as the sum of the different execution times of all available tasks divided by the periods it takes for the tasks , where $C_i$ is the execution time of task $i$ and $T_i$ is the period of task i . Utilization is given by:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{1}$$

For $n$ tasks, the utilization bound $U_{RM}$ is given by:

$$U_{RM} = n(2^{1/n} - 1)$$

If the total utilization $U$ of the task set is less than or equal to $U_{RM}$, $U \leq n(2^{1/n} - 1)$, the tasks are schedulable [3]. For large $n$, this bound approaches approximately 0.693 (or 69.3%).

*3) Completion Time Test:* Joseph and Pandya introduced the Completion Time Test, an exact method for verifying the schedulability of tasks under RMS [8]. The Completion Time Test involves calculating the worst-case response time for each task, considering the interference from higher-priority tasks. The worst-case response time $W_i$ for a task $\tau_i$ is calculated iteratively using the formula:

$$W_i^{(k+1)} = C_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{W_i^{(k)}}{T_j} \right\rceil C_j \tag{2}$$

where $hp(i)$ is the set of tasks with higher priority than $\tau_i$.

The iterations continue until the response time stabilizes, meaning $W_i^{(k+1)} = W_i^{(k)}$. A task $\tau_i$ is considered schedulable if its worst-case response time $W_i$ is less than or equal to its deadline $D_i$:

$$W_i \leq D_i \tag{3}$$

### C. Example

Consider three tasks with the following parameters:
- Task 1: $C_1 = 1$, $T_1 = 4$
- Task 2: $C_2 = 2$, $T_2 = 5$
- Task 3: $C_3 = 2$, $T_3 = 20$

To determine if Task 3 is schedulable, we calculate its worst-case response time considering the interference from Tasks 1 and 2.

1. **Initial Estimate**:

$$W_3^{(0)} = C_3 = 2$$

2. **First Iteration**:

$$W_3^{(1)} = C_3 + \left\lceil \frac{W_3^{(0)}}{T_1} \right\rceil C_1 + \left\lceil \frac{W_3^{(0)}}{T_2} \right\rceil C_2$$

$$W_3^{(1)} = 2 + \left\lceil \frac{2}{4} \right\rceil \cdot 1 + \left\lceil \frac{2}{5} \right\rceil \cdot 2 = 2 + 1 + 2 = 5$$

3. **Second Iteration**:

$$W_3^{(2)} = C_3 + \left\lceil \frac{W_3^{(1)}}{T_1} \right\rceil C_1 + \left\lceil \frac{W_3^{(1)}}{T_2} \right\rceil C_2$$

$$W_3^{(2)} = 2 + \left\lceil \frac{5}{4} \right\rceil \cdot 1 + \left\lceil \frac{5}{5} \right\rceil \cdot 2 = 2 + 2 + 2 = 6$$

4. **Third Iteration**:

$$W_3^{(3)} = C_3 + \left\lceil \frac{W_3^{(2)}}{T_1} \right\rceil C_1 + \left\lceil \frac{W_3^{(2)}}{T_2} \right\rceil C_2$$

$$W_3^{(3)} = 2 + \left\lceil \frac{6}{4} \right\rceil \cdot 1 + \left\lceil \frac{6}{5} \right\rceil \cdot 2 = 2 + 2 + 4 = 8$$

Since $W_3^{(3)} = 8$, and assuming the deadline $D_3 \geq 8$, Task 3 is schedulable. If $D_3 < 8$, Task 3 is not schedulable.

## III. UPPAAL Implementation of Rate Monotonic Scheduling (RMS)

### A. Overview

Rate Monotonic Scheduling (RMS) is a fixed-priority algorithm where tasks with shorter periods are given higher priority. To illustrate RMS using UPPAAL, we define the task periods ($p1$, $p2$) and computation times ($c1$, $c2$). In this model, $p1$ and $p2$ represent the periods for Task 1 and Task 2, respectively, while $c1$ and $c2$ represent their execution times.

UPPAAL is a tool for modeling, simulation, and verification of real-time systems modeled as timed automata. Our objective is to demonstrate how RMS can be visualized and verified using timed automata within the UPPAAL framework. By defining the system in UPPAAL, we can simulate the tasks' execution and preemption to ensure they adhere to the RMS scheduling policy.

### B. Task and Execution Time Definition

In our UPPAAL model:

- **Task 1:** Period = 4, Computation time = 2
- **Task 2:** Period = 10, Computation time = 5

We aim to validate that the tasks meet their deadlines under the RMS algorithm. This involves verifying that higher-priority tasks preempt lower-priority ones as necessary.

### C. Modeling RMS in UPPAAL

The UPPAAL model consists of several automata representing tasks, the scheduler, and a period checker. Each template is designed to manage specific aspects of the RMS algorithm. Below, we discuss each template in detail.

*1) Task 1 Automaton:* The Task 1 automaton models the lifecycle of Task 1 under the RMS algorithm. Initially, Task 1 resides in a waiting state until its period begins, as signaled by the scheduler. Upon receiving the start signal (`start_T1?`), Task 1 transitions to the executing state, initializing its local clock. During execution, Task 1 continuously checks its clock to ensure it runs for the specified computation time ($c1$). If Task 1 completes its execution, it transitions to the completion state, where it signals completion to the scheduler (`T1_done!`) and increments its execution count.

The automaton also handles preemption by higher-priority tasks. If Task 1 is preempted (`preempt_p1?`), it moves to a preempted state, pausing its execution until it can resume. This mechanism ensures Task 1 can efficiently manage its execution within the RMS framework, handling interruptions and continuations as dictated by the scheduler.

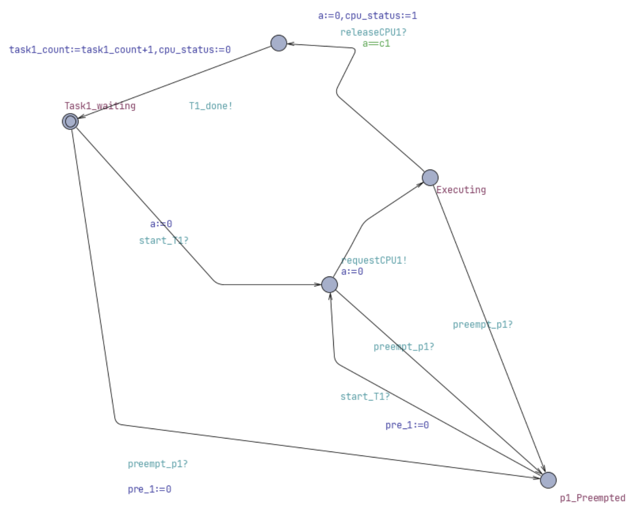


Fig. 2. Task 1 Automaton

*2) Task 2 Automaton:* Similarly, the Task 2 automaton represents the lifecycle of Task 2. It starts in a waiting state and transitions to execution upon receiving the start signal (`start_T2?`). Task 2's execution is monitored by its local clock, ensuring it runs for the specified computation time ($c2$). After completing its execution, Task 2 signals its completion to the scheduler (`T2_done!`) and increments its execution count.

Task 2 can also be preempted by higher-priority tasks. When preempted (`preempt_p2?`), it transitions to a preempted state, pausing its execution. This ensures that Task 2 respects the priorities set by the RMS algorithm, allowing higher-priority tasks to run when needed.

*3) Scheduler Automaton:* The scheduler automaton is central to the RMS implementation, managing CPU allocation based on task priorities. In the idle state, the scheduler awaits requests from tasks to start execution. Upon receiving a request from Task 1 (`requestCPU1?`), the scheduler transitions to the Task1_Executing state, granting CPU access to Task 1. Similarly, requests from Task 2 (`requestCPU2?`) transition the scheduler to the Task2_Executing state.

The scheduler ensures that tasks are preempted appropriately. If a higher-priority task needs to run, the scheduler signals preemption (`preempt_p1?` or `preempt_p2?`), causing the current task to move to a preempted state. Once a task completes its execution, the scheduler receives a release signal (`releaseCPU1!` or `releaseCPU2!`) and transitions back to the idle state, ready to allocate the CPU to the next task in line.

*4) Period Checker Automaton:* The period checker automaton ensures tasks start execution at the correct times, adhering to their defined periods. It monitors the system clock and

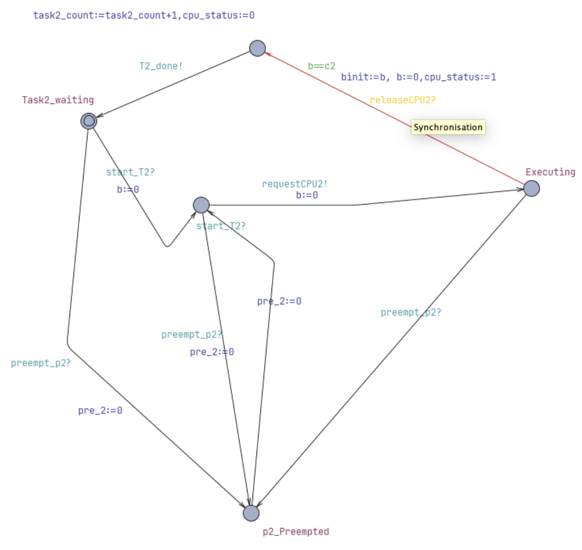


Fig. 3. Task 2 Automaton

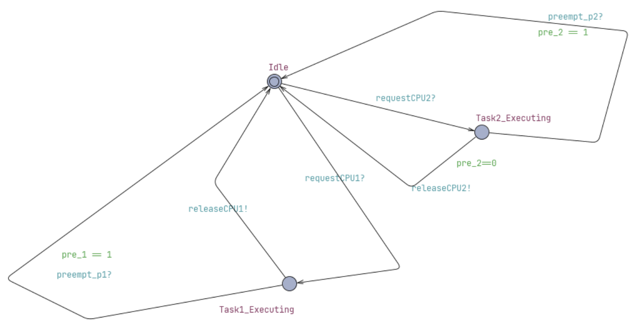


Fig. 4. Scheduler Automaton

checks if tasks need to be started or preempted based on their periods.

The period checker automaton starts by evaluating which task should run based on their priorities. When Task 1 completes (`T1_done?`), the period checker initiates Task 2 (`start_T2!`), if Task 2 is the next task in line according to the RMS rules. Similarly, if Task 2 completes (`T2_done?`), it checks if Task 1 should be restarted.

Preemption is managed by continuously checking if a higher-priority task should preempt a currently running lower-priority task. If Task 1's period ends while Task 2 is executing, the period checker signals preemption (`preempt_p2!`), ensuring Task 1 can resume execution as it holds a higher priority.

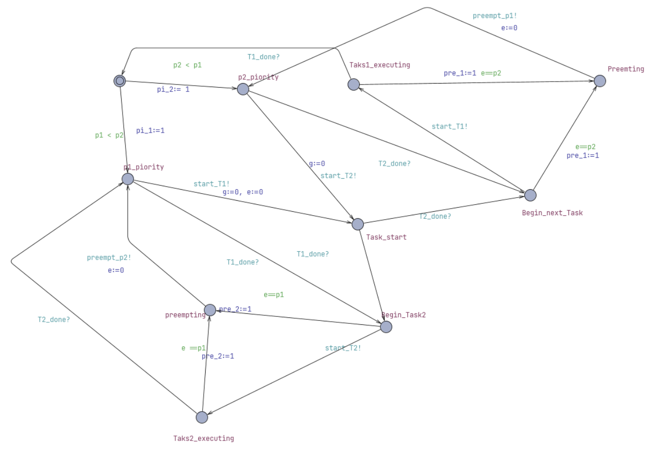


Fig. 5. Period Checker Automaton

## D. Conclusion

The integration of these automata within the UPPAAL model ensures the correct implementation of the RMS algorithm. The scheduler coordinates CPU allocation based on task priorities, while individual task automata manage their execution and preemption states. The period checker ensures tasks start at the correct times and handles priority checks, making preemption decisions when necessary.

Guards and synchronization channels play a crucial role in maintaining system integrity:

- Guards: Conditions such as `a == c1` and `b == c2` ensure that tasks only transition to completion states when their execution times are met.
- Synchronization: Channels like `requestCPU1`, `releaseCPU1`, `preempt_p1`, and `T1_done` coordinate the interactions between tasks and the scheduler, ensuring seamless task execution and preemption.

By simulating this system, we can observe the RMS algorithm in action, validating that tasks meet their deadlines and higher-priority tasks preempt lower-priority ones as expected. This model provides a robust framework for verifying real-time systems using RMS.

## E. Analysis

After running a symbolic simulation of our UPPAAL model for an extended period, the results in 6 demonstrate that the Rate Monotonic Scheduling (RMS) algorithm functions as intended. The system correctly prioritizes tasks based on their periods, ensuring that tasks with shorter periods preempt those with longer periods when necessary.

The key observations from the simulation include:

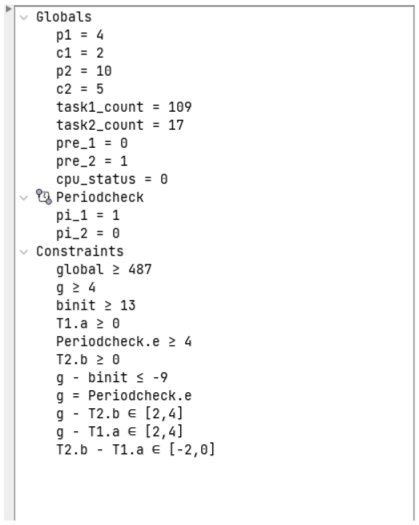
- **Task Execution and Preemption**: Task 1, with the shortest period (p1 = 4), consistently preempts Task 2 (p2 = 10), validating the RMS principle of higher priority for shorter periods.

- **Deadline Adherence**: All tasks meet their deadlines as specified by their period and computation time constraints, indicating the scheduler's effectiveness.
- **Task Count**: The task count variables show that Task 1 has been executed 109 times, while Task 2 has been executed 17 times. This indicates that Task 1 is prioritized by the system, which aligns with the RMS priority rule that tasks with shorter periods are given higher priority.

The final system behavior shows:

- Task 1 runs for its designated computation time, preempts Task 2 appropriately, and completes its cycles within its period.
- Task 2 is preempted as expected but still completes its cycles within its respective period.

The simulation verifies that our model adheres to RMS rules, effectively demonstrating the algorithm's implementation and the coordination between tasks and the scheduler.



```
Globals
    p1 = 4
    c1 = 2
    p2 = 10
    c2 = 5
    task1_count = 109
    task2_count = 17
    pre_1 = 0
    pre_2 = 1
    cpu_status = 0
Periodcheck
    pi_1 = 1
    pi_2 = 0
Constraints
    global ≥ 487
    g ≥ 4
    binit ≥ 13
    T1.a ≥ 0
    Periodcheck.e ≥ 4
    T2.b ≥ 0
    g - binit ≤ -9
    g = Periodcheck.e
    g - T2.b ∈ [2,4]
    g - T1.a ∈ [2,4]
    T2.b - T1.a ∈ [-2,0]
```

Fig. 6. Overview of the UPPAAL Model Simulation

## IV. Implementation in C Code

In this section, we describe the implementation of the Rate Monotonic Scheduling (RMS) algorithm using the C programming language. The RMS algorithm is designed to schedule tasks based on their periodicity, ensuring that tasks with shorter periods (higher priority) preempt tasks with longer periods. Below, we detail the important concepts and code structure used to achieve this.

### A. Approach

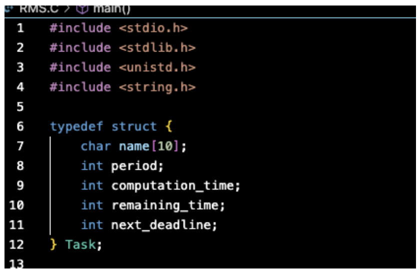
The RMS algorithm is implemented by creating a set of tasks, each defined with a specific period, computation time, and priority. The main elements of the implementation include:

- **Task Structure**: Each task is represented by a structure containing its name, period, computation time, remaining time, and next deadline.
- **Simulation Loop**: A loop that runs for the specified total simulation time, managing task execution and preemption based on their priorities.
- **Priority Selection**: At each time unit, the highest priority task that is ready to run is selected for execution.
- **Task Execution and Preemption**: The selected task runs for one time unit. If a higher-priority task becomes ready, it preempts the current task.
- **Idle State Management**: The system checks if no task is ready to run and enters an idle state, while continually checking for task readiness.
- **Task Readiness Re-evaluation**: Tasks are reset with their initial computation times and deadlines at the start of each period.

### B. C Code Implementation

The C code implementation of the RMS algorithm is divided into several key sections:

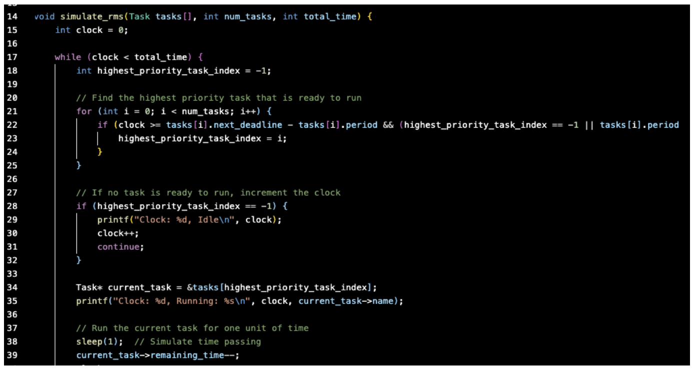
*1) Setup and Initialization:* This section includes the definition of the 'Task' structure and initialization of the tasks. Each task is initialized with its respective period, computation time, remaining time, and next deadline.



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

typedef struct {
    char name[10];
    int period;
    int computation_time;
    int remaining_time;
    int next_deadline;
} Task;
```

Fig. 7. Setup and Initialization. Preview of code,full code found here.

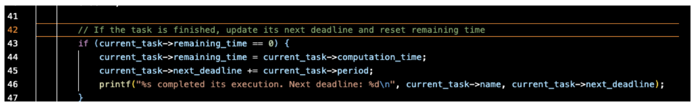
*2) Main Simulation Loop:* The main simulation loop runs for the specified total simulation time. It selects the highest priority task that is ready to run, executes it for one time unit, and manages task preemption and readiness re-evaluation.



```c
void simulate_rms(Task tasks[], int num_tasks, int total_time) {
    int clock = 0;

    while (clock < total_time) {
        int highest_priority_task_index = -1;

        // Find the highest priority task that is ready to run
        for (int i = 0; i < num_tasks; i++) {
            if (clock >= tasks[i].next_deadline - tasks[i].period && (highest_priority_task_index == -1 || tasks[i].period
                highest_priority_task_index = i;
            }
        }

        // If no task is ready to run, increment the clock
        if (highest_priority_task_index == -1) {
            printf("Clock: %d, Idle\n", clock);
            clock++;
            continue;
        }

        Task* current_task = &tasks[highest_priority_task_index];
        printf("Clock: %d, Running: %s\n", clock, current_task->name);

        // Run the current task for one unit of time
        sleep(1);  // Simulate time passing
        current_task->remaining_time--;
```

Fig. 8. Main Simulation Loop. Preview of code,full code found here.

*3) Task Execution and Preemption:* This section handles the execution of the selected task for one time unit. It decrements the task's remaining time and updates the clock. If a higher-priority task becomes ready, it preempts the current task.



```c
    // If the task is finished, update its next deadline and reset remaining time
    if (current_task->remaining_time == 0) {
        current_task->remaining_time = current_task->computation_time;
        current_task->next_deadline += current_task->period;
        printf("%s completed its execution. Next deadline: %d\n", current_task->name, current_task->next_deadline);
    }
```

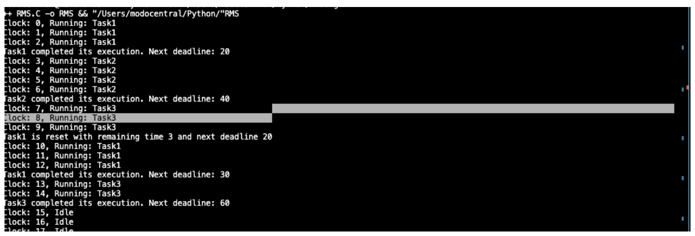
Fig. 9. Task Execution and Preemption. Preview of code,full code found here.

*4) Idle State Management:* The system enters an idle state if no task is ready to run. It continually checks for task readiness during this state.

*5) Task Readiness Re-evaluation:* At the start of each period, tasks are reset with their initial computation times and deadlines. This ensures that tasks are ready to run at their specified intervals.

### *C. Results of Running the Code*

The results of running the RMS algorithm are depicted in Figure 10, which shows the scheduling of tasks over a period of 60 units of time. The processor is shown to execute tasks according to their priorities and periods, ensuring that the highest priority tasks are executed first.



```
+ RMS.C -o RMS && "/Users/modocentral/Python/"RMS
Clock: 0, Running: Task1
Clock: 1, Running: Task1
Clock: 2, Running: Task1
Task1 completed its execution. Next deadline: 20
Clock: 3, Running: Task2
Clock: 4, Running: Task2
Clock: 5, Running: Task2
Clock: 6, Running: Task2
Task2 completed its execution. Next deadline: 40
Clock: 7, Running: Task3
Clock: 8, Running: Task3
Clock: 9, Running: Task3
Task1 is reset with remaining time 3 and next deadline 20
Clock: 10, Running: Task1
Clock: 11, Running: Task1
Clock: 12, Running: Task1
Task1 completed its execution. Next deadline: 30
Clock: 13, Running: Task3
Clock: 14, Running: Task3
Task3 completed its execution. Next deadline: 60
Clock: 15, Idle
Clock: 16, Idle
Clock: 17, Idle
```

Fig. 10. Result of Running the C Code. Preview of code,full code found here.

Key observations from the simulation include:

- Task 1, with the highest priority, runs every 10 units for 3 units of time.
- Task 2, with medium priority, runs every 20 units for 4 units of time.
- Task 3, with the lowest priority, runs every 30 units for 5 units of time.
- The system correctly preempts lower-priority tasks when a higher-priority task is ready to run.

The result confirms that the RMS algorithm successfully schedules tasks based on their periods and priorities, aligning with the theoretical expectations of the RMS scheduling policy.

### *D. Conclusion*

The implementation of the Rate Monotonic Scheduling (RMS) algorithm in C demonstrates the feasibility and effectiveness of the algorithm in scheduling periodic tasks based on their priority and period. The simulation results validate that tasks are scheduled correctly according to the RMS principles, ensuring that tasks with shorter periods (higher priority) preempt tasks with longer periods when necessary.

The implementation highlights the importance of handling task preemption and readiness re-evaluation, ensuring that the system operates efficiently without unnecessary idle times. The successful execution of the C code provides a practical example of how RMS can be applied in real-time systems.

## V. Conclusion

This paper examined the implementation and analysis of Rate Monotonic Scheduling (RMS) using UPPAAL for modeling and C programming for practical execution. Our study confirms the efficacy of RMS in scheduling periodic tasks based on priority and period.

### *A. Key Findings*

- **UPPAAL Implementation:**
  - The UPPAAL model accurately simulates RMS, including automata for tasks, a scheduler, and a period checker.
  - Simulations verified that higher-priority tasks preempt lower-priority ones, ensuring tasks meet their deadlines.
  - Observations showed Task 1, with the shortest period, consistently preempted Task 2, demonstrating RMS principles.
- **C Code Implementation:**
  - The C code replicated RMS logic, prioritizing tasks with shorter periods.
  - Results confirmed theoretical expectations: Task 1 ran every 10 units for 3 units, Task 2 every 20 units for 4 units, and Task 3 every 30 units for 5 units.
  - The system handled task preemption effectively, avoiding unnecessary idle times and ensuring efficient CPU utilization.

## References

[1] A. Tanenbaum, “Modern operating systems,” Pearson Education, Inc., 2009.

[2] A. Silberschatz, J. L. Peterson, and P. B. Galvin, “Operating system concepts,” Addison-Wesley Longman Publishing Co., Inc., 1991.

[3] G. C. Buttazzo, “Hard real-time computing systems: predictable scheduling algorithms and applications,” Springer Science & Business Media, 2011

[4] J. W. S. Liu, “Real-Time Systems,” Prentice Hall, 2000.

[5] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[6] C. L. Liu and J. W. Layland, “Rate Monotonic Analysis for Real-Time Systems,” Carnegie Mellon University, Software Engineering Institute, Technical Report, 1989. Presents the foundational principles of RMS and its mathematical analysis.

[7] University of Idaho, “Introduction to Rate Monotonic Scheduling,” 2021. Provides an introductory overview of RMS with examples and basic concepts. Available at: https://www.webpages.uidaho.edu/~jimc/Teaching/CS445/lectures/rms.pdf

[8] M. Joseph and P. Pandya, “Finding Response Times in a Real-Time System,” The Computer Journal, vol. 29, no. 5, pp. 390-395, 1986.

.