Rate Monotonic Scheduling

Shehroz Bashir Malik
Hochschule Hamm-Lippstadt
Lippstadt, Germany
Bachelors in Electronic Engineering

***Abstract*—Rate-Monotonic Scheduling (RMS) (a.k.a. Rate Monotonic Algorithm, RMA) method is a preemptive, priority-based scheduler. This paper dives deeper in its ability to meet deadlines of a periodic task set. Bound tests will be explored to determine whether a task can be scheduled by the RMA principle In addition to the accumulative utilization. The algorithm will be implemented in Python code. The limitations and potential modifications will be also be discussed.**

I. Introduction

REAL Time Systems are systems that need to respond to a service request within a decided set of time. These service requests consist of a task and an accompanying constraint. The nature of this constraint is paramount. This constraint or more commonly known as the deadline is the time by which the task needs to be executed. There are two types of deadlines:
1) Hard Deadline: this requires the task to be completed exactly at the time specified or there will be catastrophic consequences [1]
2) Soft Deadline: Missed deadline is undesirable but tolerable.
Real Time applications needs to be predictable (deterministic), accurate and have a high degree of schedulability.This is achieved by Preemptive priority scheduling, Bounded latency, Efficient Interrupt Service Routines and Deterministic Synchronization.
For this paper we will focus on a static-priority scheduling policy known as the Rate Monotonic Algorithm.

II. What is RMA

Real Time Tasks have four particular specifications [2] :

1) Pi: Period, time interval that indicates the frequency of a task
2) Ri: Release Time, time when the task is available for execution
3) Ei: Worst-Case execution time, indicating the demand for processing time
4) Di: Deadline, time by which the task needs to be executed by

Rate Monotonic Scheduling Algorithm assigns priorities to tasks according to their periods. The priority of a task is inversely proportional to its period. Periods are constant which in turn makes RMS a fixed-priority assignment. It is intrinsically preemptive. This means that a currently executing task is preempted by a newly arrived task with a higher priority.

Now consider the following:

The Task X1 has 3 individual jobs J1, J2 and J3

$$X_1 = \begin{cases} J_1 = (3, 20) \\ J_2 = (2, 5) \\ J_3 = (2, 10) \end{cases} \tag{1}$$

According to RMS, Task 2 has the highest priority, Task 1 the lowest and Task 3 in the middle. This is due to the fact Task 1 has the largest period and Task 2 the lowest.
The time period at which the schedule repeats is the LCM of the Task's period also known as the Hyperperiod [3] . Therefore in our case, the LCM is 20. Using this information we can make a chart to visualize the Scheduling Algorithm.

III. Visualizing the Algorithm

Considering X1 from earlier, RMA will prioritize T2 over T3 over T1. Since T2 has the highest priority and repeats every 5 seconds for 2 seconds each, it will be the first task to execute. Following this, the schedule will execute T3, which repeats every 10 seconds for 2 seconds each. Lastly T1 will be assigned.

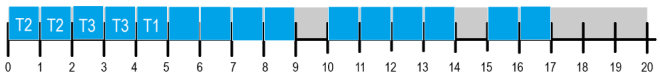

Fig. 1. Tasks executed from 0-5

Although T1 needs to run for 3 seconds, it will only be allowed to execute for one. This is due to the fact that at the 5 second mark, T2 will have a higher priority to run. T3 will not be running in the 5-10 second interval as it only runs every 10 seconds. Thus giving T1 the priority it needs to finally run. Since there is no task left, the system remains idle during the 9-10 second interval.

Since T1 is executed for 3 seconds every 20 seconds, it will no longer be executed in the 10-20 second interval. T2 and T3 will thus be executed as expected. All the tasks have

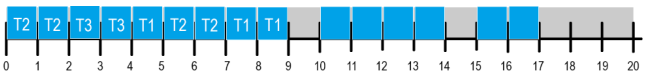

Fig. 2. Tasks executed from 0-10

been executed which allows the system to remain idle from 17-20 seconds.

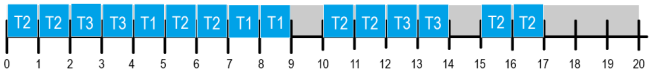

Fig. 3. Tasks executed from 0-20

IV. RMA ANALYSIS

Not all tasks can be scheduled according to RMA. There are plenty of cases where the deadline can not be met. Consider a periodic task with a Period and a Worst-Case Execution time, we can calculate its Processing Demand/ CPU Utilization with the following formula [4] :

$$U = (\frac{e_i}{p_i}) \tag{2}$$

Then calculate the bound using the Bound Test formula:

$$\sum_{i=1}^{k}(\frac{e_i}{p_i}) = \frac{e_i}{p_i} + ... + \frac{e_k}{p_k} \leq k(2^{\frac{1}{k}} - 1) \tag{3}$$

RMA states that a task can be scheduled if the cumulative CPU utilization is less than or equal to the Bound Test.
Consider Job X1 from earlier:

$$X_1 = \begin{cases} J_1 = (3, 20) \\ J_2 = (2, 5) \\ J_3 = (2, 10) \end{cases} \tag{4}$$

The CPU Utilization for Job 1 is

$$U = 3/20 + 2/5 + 2/10 = 0.75 \tag{5}$$

Taking k as 3, the bound will be calculated as follows:

$$3(2^{\frac{1}{3}} - 1) = 0.7798 \tag{6}$$

The Utilization is less than the Bound which implies that this task is schedulable.
A few scheduability tests will be explored [5] in the following sections:

A. Worst Case Bound

We can calculate the bound of the total CPU utilization as K tends to infinity using the following Python Code:

```
from sympy import *
k = symbols('k')
bound = k*((2**(1/k))-1)
l = limit(bound,k,oo)
print(l)
```

The limit comes out to be:

$$\log_e 2 = 0.69$$

. This is known as the Worst Case Bound [4]. This means that a task set with infinite tasks will be successfully scheduled provided that the total utilization is less than or equal to 0.69. In most use cases, certain tasks sets can be scheduled even if the utilization is greater than the worst case bound. This infers the average case behavior is better than the worst case behavior.

Using Stochastic Analysis [6], it has been found that the threshold for scheduability is approximately 88%. Therefore, there is a possibility that a task can be scheduled even if it exceeds the Bound. The Bound is a soft condition not a hard condition.

B. Burchard-Bound

This bound has the capability to increase a processors utilization provided that all the periods in the tasks have values that are close to each other. This was proposed by Burchard et al. in their paper [7]
Consider a Task Set $\tau_1, \tau_2.., \tau_n$

$$S_i = \log_2 T_i = [\log_2 T_i] and i - 1, ..., n \tag{7}$$

and

$$\beta = S_i * max_{1<=i<=n} - S_i * min_{1<=i<=n} \tag{8}$$

There are two conditions to be fulfilled and they are as follows:
When $\beta < 1 - 1/n$:

$$U <= (n-1)(2^{\frac{\beta}{(n-1)}} - 1) + 2^{1-\beta} - 1 \tag{9}$$

When $\beta >= 1 - 1/n$:
$U <= n(2^{\frac{1}{n}} - 1)(10)$
If either one of these conditions is satisfied that the task set can be scheduled on a single processor. Although this raises the least upper bound of the algorithm, it still can not be considered a general scheduability test.

C. Hyperbolic Bound

This paper by Bini and Buttazzo [8] provides another way to check tasks. This one in particular checks whether the tasks in the RMA are feasible. It is based on the Worst Case Bound defined earlier.It manipulates the feasibility condition to find a stricter schedulability test which is a function of the individual task utilizations.
Consider a Task Set $\tau_1, \tau_2.., \tau_n$
where if the following condition holds true, the task set can be scheduled:

$$\prod_{i=1}^{n}(U_i + 1) <= 2 \tag{11}$$

V. Python Implementation

To implement the Rate Monotonic Scheduling Algorithm, I chose to go with Python. The following proved to be helpful [9], [10]

```
import copy

tasks = {
    0: {'Period': 20, 'Execution': 3},
    1: {'Period': 5, 'Execution': 2},
    2: {'Period': 10, 'Execution': 2}
     }
lcm = 20
print("The number of tasks are 3 with
an lcm of 20")
```

The tasks are stored in a nested dictionary. I used the dictionary data structure because of its unique format [11] . Starting with the nested dictionary, Period is the key while 20 is its corresponding value. The outer dictionary's key is 0 and its value is the dictionary that contains the Period and Execution along with their respective values.

```
def PriorityCalculator(TaskOrder_Priority):
lcm = 20
Priority = -1
for i in TaskOrder_Priority.keys():
    if (TaskOrder_Priority[i]["Execution"] != 0)
        if (lcm > TaskOrder_Priority[i]["Period"
        or lcm > tasks[i]["Period"]):
            lcm = tasks[i]["Period"]
            Priority = i
    return Priority

TempTask = copy.deepcopy(tasks)
```

The for-loop goes through the Tasks. The .keys() method allows the for-loop to access the keys of the tasks. The if-condition checks if the priority of execution time of the tasks is not yet zero.Every time the task is executed on a time slot, its execution time will decrease by 1 so if the execution time is zero, it means this particular task can not be executed any further. The nested if-condition compares the LCM with the period of the tasks and if larger it assigns the LCM to the largest Period. Simultaneously, it also assigns the priority which initially was -1.

The deepcopy() is used from the copy library. A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original [12]. The temporary task dictionary is used to calculate the priority .

```
for time in range(lcm):
    priority = PriorityCalculator(TempTask)

    if (priority==-1):
        print("Time Slot %d to Time Slot
        %d will be idle\n" %
        (time,time+1))
    else:
        print("Time Slot %d to Time slot
        %d is Task %d \n" %
        (time, time + 1, priority))

        TempTask[priority]["Execution"] -= 1
    for i in TempTask.keys():
        TempTask[i]["Period"] -= 1
        if (TempTask[i]["Period"] == 0):
            TempTask[i] =
            copy.deepcopy(tasks[i])
```

Now this function runs 20 times and each time it will calculate the priority using the aforementioned function. The rest of the code checks if it is idle and prints out the Time Slot and the fact that it is idle. Whenever a task is executed, it reduces the execution time by 1. Otherwise, it reduces the period by 0. When the period finally reaches zero it copies the temporary task into the main task.

```
The number of tasks are 3 with an lcm of 20
Time Slot 0 to Time slot 1 is Task 1

Time Slot 1 to Time slot 2 is Task 1

Time Slot 2 to Time slot 3 is Task 2

Time Slot 3 to Time slot 4 is Task 2

Time Slot 4 to Time slot 5 is Task 0
```

Fig. 4. Tasks executed from 0-5

```
Time Slot 5 to Time slot 6 is Task 1

Time Slot 6 to Time slot 7 is Task 1

Time Slot 7 to Time slot 8 is Task 0

Time Slot 8 to Time slot 9 is Task 0

Time Slot 9 to Time Slot 10 will be idle

Time Slot 10 to Time slot 11 is Task 1
```

Fig. 5. Tasks executed from 6-11

```
Time Slot 11 to Time slot 12 is Task 1

Time Slot 12 to Time slot 13 is Task 2

Time Slot 13 to Time slot 14 is Task 2

Time Slot 14 to Time Slot 15 will be idle
```

Fig. 6. Tasks executed from 11-15

VI. Delayed Rate Monotonic Algorithm

Mahmoud Naghibzadeh provides an improved version of the RMA in his paper [13]. Since RMA has two states - the ready and running state, where tasks are being preempted and executed according to their periods, he proposes adding an additional state called the delayed state. Task with the lowest period enters the Ready State Queue. All other tasks are sent directly to the Delay State Queue. The priority of the ready state is greater than the delay state. He achieves this by adding a Delay Time to each task and allows tasks to jump to the ready state once their delay time is over. The Delay Time is split in two characteristics: Task Set 1 and all other Tasks.
For T1 (which has the lowest period):

$$DelayTime = p_i - e_i \tag{12}$$

For all other tasks where i is greater than 2

$$DelayTime = \alpha * T_i * Utilization \tag{13}$$

Alpha is a constant with values ranging from 0 to 1. This method has lower overhead and is safer than traditional RMA. DRMA can also be applied to Multiprocessors and is detailed in this paper [14].

VII. Parallel Implementation of RMA

Rajnish Dashora et al. [15] propose a method to run the RMA algorithm in parallel on multiple cores. They employ the use of OpenMP where multiple threads are created. It runs on the basis that all cores can not remain idle provided that there is a process in ready state. These ready state processes must be executed by a core. After priority determination, the tasks are assigned to their individual unique threads thus preventing race conditions. After every unit of time, the task's period and execution time are checked and their priority recalculated. Those with a lower priority are put on hold. Their ParaRMS approach was able to achieve performance improvements in the ranges of 50-80%.

VIII. Conclusion

Rate Monotonic Scheduling is a powerful scheduling algorithm. It is one of the most important ones used in Real Time Systems. It has its drawbacks and can only be implemented in certain use-cases. Identifying these use cases is a tad difficult as the Liu Worst Case Bound is not an all encompassing rule. The Algorithm does have a lot of overhead which can be solved by using Delayed Rate Monotonic Scheduling.

References

[1] K. Ramamritham and J. A. Stankovic, *Advances in real-time systems*. Los Alamitos, Calif IEEE Computer Society Press, 1995. [Online]. Available: https://books.google.de/books?id=KuycBAAAQBAJ

[2] X. Fan, *Real-Time Embedded Systems: Design Principles and Engineering Practices*. Elsevier Science, 2015. [Online]. Available: https://books.google.de/books?id=KuycBAAAQBAJ

[3] G. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, ser. Real-Time Systems Series. Springer US, 2011. [Online]. Available: https://books.google.de/books?id=h6q-e4Q_rzgC

[4] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," vol. 20, no. 1, 1973. [Online]. Available: https://doi.org/10.1145/321738.321743

[5] Y. Li, T. Liu, J. Zhu, X. Wang, M. Duan, and Y. Wang, "Comprehensive study of schedulability tests and optimal design for rate-monotonic scheduling," *Computer Communications*, vol. 173, pp. 107–119, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366421001092

[6] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *[1989] Proceedings. Real-Time Systems Symposium*, 1989, pp. 166–171.

[7] A. Burchard, J. Liebeherr, Y. Oh, and S. Son, "New strategies for assigning real-time tasks to multiprocessor systems," *IEEE Transactions on Computers*, vol. 44, no. 12, pp. 1429–1442, 1995.

[8] E. Bini, G. Buttazzo, and G. Buttazzo, "Rate monotonic analysis: the hyperbolic bound," *IEEE Transactions on Computers*, vol. 52, no. 7, pp. 933–942, 2003.

[9] prasannjeet, "Rate-monotonic-scheduling-algorithm: Rms implementation in java. without considering resource sharing and precedence." [Online]. Available: https://github.com/prasannjeet/Rate-Monotonic-Scheduling-Algorithm

[10] sampreets3, "Scheduler-rm: Simulation of the behaviour of a preemptive fixed priority rate monotonic scheduler in c++." [Online]. Available: https://github.com/sampreets3/scheduler-RM

[11] "Python dictionaries." [Online]. Available: https://www.w3schools.com/python/python$_dictionaries.asp$

[12] "Copy, shallow and deep copy operations - python 3.10.4 documentation." [Online]. Available: https://docs.python.org/3/library/copy.html

[13] M. Naghibzadeh, "A modified version of rate-monotonic scheduling algorithm and its' efficiency assessment," in *Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. (WORDS 2002)*, Jan 2002, pp. 289–294.

[14] S. Senobary and M. Naghibzadeh, "Delayed rate monotonic with semi-partitioned technique adapted to multiprocessors," in *2014 6th International Conference on Computer Science and Information Technology (CSIT)*, 2014, pp. 82–89.

[15] R. Dashora, H. P. Bajaj, A. Dube, and N. M, "Pararms algorithm: A parallel implementation of rate monotonic scheduling algorithm using openmp," in *2014 International Conference on Advances in Electrical Engineering (ICAEE)*, 2014, pp. 1–6.