Rate Monotonic Scheduling Implementation

José Vitor Velozo de Carvalho

[1]Computer Science – CESAR School
Avenida, Cais do Apolo, 77, Recife - PE, 50030-22

jvvc@cesar.school

Abstract. *This document describes the implementation of a simulation of rate monotonic scheduling algorithm, developed for Software Infrastructure class*

1. Folder structure

The program's folder structure consists of three files: `func.h`, `func.c` and `rate.c`. Those are combined using `Makefile` to generate a `rate` executable file.

2. Receiving data from files

In order to receive data from a file passed by parameter to the executable, the program first counts the number of lines in the informed file. With this information, `fgets` is utilized to read the total time, using `atoi` to convert the string present in the line to integer.

After this, the program uses the number of tasks in the file to read the rest of the lines using `fscanf`. The lines are separated into a string and two decimals, which are stored directly into `found_tasks` - a `Task` struct array - in the same index as the count. Those values are stored into the Task struct's `task_name`, `period` and `time_unit` (CPU Burst) fields. It also initializes all the other fields of the struct.

2.1. The Task struct fields

The task struct can be found at `func.h` and contains `period`, `original_period`, `time_unit`, `original_time_unit`, `task_name`, `completed_count`, `lost_count`.

The `period` and `original_period` fields represent the deadlines of each task's instance. The original time period will be utilized to increment period to the next deadline whenever its period expires. Similarly, there are `time_unit` and `original_time_unit`, which represent the CPU Burst for each task. For example, whenever the burst of a task hits zero, it means that the task was finished. When the period of a task is updated, original time unit's value is copied to time unit. The `task_name` field contains the task's name, and `completed_count` and `lost_count` are counters for finished tasks and lost deadlines, respectively.

2.2. Ordering tasks by priority

Since priority is inversely proportional to the period, the `found_tasks` array is copied to an array of same size named `ordered_tasks`. This array is then ordered by the value of period of each task, in ascending order by adapting a Bubblesort algorithm developed previously for Data Structures and Algorithms class in second term. It can be found at https://github.com/JVitorCarv/sorting-algorithms/blob/main/BubbleSort.c, in my personal GitHub account.

3. Execution by Rate - Implemented

In order to print the Execution by Rate section to the file, the program uses 3 integer variables and 1 Task struct outside of the loop. There is a `count` variable which is incremented and compared with the total time on every iteration, a `task_count` which stores how much time a task has executed continuously, and an `idle_count` variable which stores how much time the CPU has spent idling. The Task struct, called previous, is initialized with default values for every field, is used to compare the last iteration Task with the current. This will become clearer in the following subsections.

3.1. Updating periods

The first thing that the execution verifies is whether a new task has arrived or not. If the period of the task is lower than the actual time (count), the period is updated using original time period, and its burst time is updated. If the remaining burst time is higher than zero, that means that the task was lost and its lost count field is incremented.

3.2. Idle CPU

When trying to find which task will be executed next, the program tries to find a task with burst time (`time_unit`) higher than zero. For this, a found integer value, which will serve as a flag, is used. If none is found, then this integer remains zero. Because of this, the program will check this flag and increment `idle_count` for as long as it is idle. If the CPU has been idle, the program also checks whether the next cycle is the last in order to print the idle count message. This message is also printed after a new task takes the CPU after idle time. This will be explored in the next subsection.

3.3. Executing a task

As previously mentioned, in order to find out if there is a task to be executed, the program iterates over the ordered array and stops, breaking the loop as soon as it finds a task with burst time higher than zero. Notice that since the array is ordered, this task will always be highest priority task available.

Before continuing, the program checks whether `idle_count` is higher than zero, in order to print how much time the CPU was idle to the file.

After this, the task's burst time will be decremented, and various verifications follow. The first one is if a hold has occurred. For this, the program calls `print_if_hold`. In order to do this, this function compares the previous task's name to the current's, as well as if the previous task was finished or not (burst time zero or not). If these conditions pass, it means that a hold has occurred, and the hold message is printed to the execution. Lastly, the task count's value is set to zero, in order to count the time of execution for the current task.

The next verification is if the task is lost. If the current task's name is the same as the previous', the current burst time is higher than the previous' task burst and if the task count is higher than zero. These are all necessary in order to guarantee that the previous task was a previous instance or the current. When those conditions pass, the lost message is printed to the file and task count is set to zero, for the same reason as the previously described verification.

After those verifications, the task count value is incremented. This is necessary in order to guarantee that the increment will go to the correct task.

The next verification only checks if the task's burst is zero. If that is true, then the task has finished execution, its message will be printed to the file, its completed count is incremented, and task count will be set to zero.

Lastly, the program takes into consideration count and total time. This is used to check whether the next cycle is the last. It also checks whether the burst time is higher than zero or not. If both conditions hold, this means that the task will be killed by the program, and so it prints to the file. But notice that the killed count is not set here. It will be set later, in the Killed section.

Then, the previous task will be set to the current task and found will be set to one, in order to sinalize that a task was found (the CPU was not idle).

4. Lost Deadlines - Implemented

This section iterates through the ordered array and prints its task name and lost count.

5. Complete Execution- Implemented

Iterates through the ordered array and prints its task name and completed count.

6. Killed - Implemented

This section iterates the array and verifies whether there is a remaining burst time for the given task or its next instance is at the same time of the total time of execution. If either passes, the process name will be printed along with 1. Else, it will be printed with 0.

7. Error Handling - Implemented

First, before executing the rest of the program, the program checks the `argc` value. If this value is higher than 2, the program prints to the terminal informing that only one file can be provided as parameter. If this value is lower than two, then the program prints to the terminal asking that at least one file is provided.

Next, it checks if the file can not be found, and prints to the terminal informing that the file was not found.

8. Conclusion

This program was tested using the provided test in the instructions file at Google Classroom, but 7 more cases were taken into consideration. Those files were not included in the submission form, in order to avoid spam, but are available at the private GitHub repository used for this implementation. At the end of the current term of 2022.2 at CESAR School, this repository will be publicly available at https://github.com/JVitorCarv/rate-monotonic.