Politecnico di Torino

Project Report

A new RealTime Scheduling library for FreeRTOS

Rollo Francesco s328739

01GYKUV Computer architectures and operating systems

February 2024

Contents

1 Introduction **2**

2 Running FreeRTOS on QEMU **2**

3 A FreeRTOS Demo **2**

4 RealTimeScheduler: a library for Real-Time Scheduling **3**
4.1 Periodic Tasks - RMS & EDF 4
4.2 Aperiodic Tasks - Polling Server 4
4.3 Feasibility Test and Worst-Case-Response-Time Analysis 4
4.4 Execution Time Statistics 5

5 Evaluation **5**
5.1 Base scheduling 5
5.2 RMS 6
5.3 EDF 6
5.4 RMS Polling Server 7
5.5 RMS with WCRT 8

1 Introduction

This report provides an overview of the work conducted and the results obtained throughout the project, aimed at achieving the following main objectives:

- Define and set up the target hardware within a QEMU emulating environment for executing the FreeRTOS kernel
- Demonstrate the functionality of the operating system by highlighting practical examples that better illustrate its capabilities
- Implement an efficient library enabling the support for Real Time scheduling
- Evaluate the performance achieved in the previous point

All of these points have been successfully accomplished with great results as shown later on in the document.

2 Running FreeRTOS on QEMU

For this project and subsequent testing and implementation, we have decided to emulate the AN385 platform on the **MPS2** board using QEMU. The MPS2 board is a **Cortex-M3** based SoC.

A detailed tutorial for downloading, setting up, and executing the FreeRTOS kernel on the MPS2 emulated machine, as well as building the entire project code, can be found in the README file on the main page of the GitLab repository for this project.

3 A FreeRTOS Demo

To gain proficiency with the FreeRTOS kernel, we primarily focused on the usage of **Task Management API** and **Queue Management API** highlighting how the Task Scheduler and queues influence the execution of our first demo.

The main concept behind the demo is to emulate a communication between a router (**server**) and multiple hosts (**clients**). The clients contact a fictitious DHCP server to request an IP assignment. Once all hosts are configured with valid IP addresses, an endless loop of fictitious ICMP echo requests begins from the clients towards the server. The server and the clients communicate by means of two queues: `xQueueHostname` for the demanding of the address by the client and `xQueueIP` for sending out the assigned address by the server.

For this purpose these two roles are separated in two isolated files and two data structures are defined:

- the server contains a **linked list** (ADT I class) keeping track of the IP address and name for each host
- the client basic information are saved in a **general structure** which is part of an array of various clients structures

To implement the IP assignation phase, three tasks are created: `SendClientTask`, `ReceiveClientTask` and `ServerAssign`. Each task is given a suitable priority (higher value = higher priority):

- the client Task that requests an IP has the highest priority (7)
- the client Task that receives the new IP has a low priority (5)
- the server Task that sends out the configuration parameters will have a medium priority (6)

The function `AssignClientIP()` initializes the two queues and the IP List owned by the server, assigns a new IP address to the Server by storing it in the IP List and then creates the `ServerAssign` task. The function `ReceiveClientIP()` initializes the general structure of the clients containing their hostname and their IP (initially empty) and then creates for each client two tasks: `SendClientTask` for requesting the IP address and `ReceiveClientTask` for storing the IP address received. All the tasks are deleted at the end of the following process.

Each `SendClientTask` adds to the `xQueueHostname` its hostname as identifier for the request. Then the `ServerAssign` task activates, extracts from the queue the hostname of each Client until the queue is empty and generates its IP address. A new entry containing the new client is stored in the IP List and then the task adds to the `xQueueIP` the IP address of each client. At the end the `ReceiveClientTask` of each client extracts from the

`xQueueIP` the IP address assigned and updates its entry in the general structure with the new address received.

Since we are working on a single-core CPU, emulating both the ICMP echo request and response would be impractical. Therefore, we've chosen to focus solely on simulating the client behavior. We've created five periodic `Ping` tasks with fixed frequencies to alternate between them. The scheduling is determined by a **fixed priority**, which is passed as a parameter along with the client initiating the action when calling the specific `Ping` function.

According to their priorities, each client generates a fictitious echo ICMP request to the server and displays it. The following output code has been optimized out for space reasons.

```
[CLIENT] Sending IP request configuration by client1
[CLIENT] Sending IP request configuration by client2
[CLIENT] Sending IP request configuration by client3
[...]
[SERVER] client1 is asking for an IP
[SERVER] IP 10.0.1.1 assigned to client1 host
[SERVER] client2 is asking for an IP
[SERVER] IP 10.0.1.2 assigned to client2 host
[SERVER] client3 is asking for an IP
[SERVER] IP 10.0.1.3 assigned to client3 host
[...]
[CLIENT] IP 10.0.1.1 received and configured
[CLIENT] IP 10.0.1.2 received and configured
[CLIENT] IP 10.0.1.3 received and configured
[...]
Ping sent from 10.0.1.1
Ping sent from 10.0.1.3
Ping sent from 10.0.1.2
[...]
```

4 RealTimeScheduler: a library for Real-Time Scheduling

A Real Time Operating System (RTOS) is a specialized software system designed to manage tasks and processes in real time systems, where tasks have strict timing constraints. While FreeRTOS offers only a preemptive fixed-priority scheduling algorithm, incorporating new Real-Time scheduling principles would prove beneficial for the development of more complex real-time systems.

To this end, the *RealTimeScheduler* library has been developed to enable advanced functionalities for Real-Time scheduling. The features provided are:

- Support for **static** and **dynamic** priority scheduling by implementing **Rate Monotonic Scheduling (RMS)** and **Earliest Deadline First (EDF)**
- Support for **aperiodic** tasks by implmeneting a **Polling Server** with flexible budget size
- Support for **Feasibility Test** under RMS and EDF scheduling algorithms by implementing both the Standard Test and the Necessary Test based on **Worst Case Response Time (WCRT) Analysis**
- Support for Task's **Execution Time Statistics**

The code architecture is designed with efficient modularity in mind. Therefore, depending on the feature selected, thanks to the use of C directives only the required functions and data structures are initialized.

For enabling the use of the mentioned algorithms we created a custom Task Control Block, following the original FreeRTOS TCB, including additional information for handling both periodic and aperiodic tasks. Since periodic and aperiodic tasks require different information, they have their own TCB. In particular periodic tasks' TCB includes:

- Worst Case Time Execution (WCET) & Worst Case Response Time (WCRT)
- Last Wake Time & Arrival Time
- Period, Deadline & Absolute deadline
- Execution Time

Aperiodic tasks' TCB, instead, includes only the WCET and the Arrival Time.

4.1 Periodic Tasks - RMS & EDF

With the `vPeriodicTaskCreate()` API it's possible to create periodic tasks. All the periodic tasks are then stored inside a linked-list which will be useful later for sorting tasks according to the policies of each scheduling algorithm.

To handle periodic tasks we can decide to use either Rate Monotonic Scheduling or Earliest Deadline First algorithms.
When RMS is enabled, *RealTimeScheduler* prioritizes tasks by assigning higher priority to those with shorter periods before the native scheduler starts.
Instead, when EDF is enabled, *RealTimeScheduler* creates another task called `SchedulerEDF` where a context switch is made to at the release and at the end of each task. Its role is to assign priorities dinamically based on their absolute deadlines.

4.2 Aperiodic Tasks - Polling Server

With the `vAperiodicTaskCreate()` API it's possible to create aperiodic tasks. These tasks are stored within a FIFO queue based on their arrival time and are not treated as normal FreeRTOS tasks. Instead, they are managed by the **Polling Server**, which is a periodic task created by *RealTimeScheduler* when the support for aperiodic tasks is enabled.

The Polling Server has a limited budget size for handling these tasks, which can be modified by the user as needed. The concept behind this implementation is to utilize RMS for managing periodic tasks. Therefore the Polling Server, when activated, is responsible for retrieving the first available aperiodic task from the queue and executing it, if the budget size constraint is satisfied.

4.3 Feasibility Test and Worst-Case-Response-Time Analysis

RealTimeScheduler library provides support also for computing a Feasibility Test given a task set. A feasible scheduling ensures that the timing constraints of all computations are met. Schedulability analysis hinges upon prior knowledge of the WCET for each computation. We know that the **utilization factor U** represents the fraction of CPU time used by the computation and it is defined by the following equation:

$$U_i = \frac{C_i}{T_i}$$

Where C_i is the WCET of the computation i and T_i is the period.

Feasibility of RMS

A set of n perioidc tasks is schedulable with RM if

$$\prod_{i=1}^{n}(U_i + 1) \leq 2$$

If the condition is not valid it does not mean that the algorithm does not work but we can prove that it may still. This feasibility test is also referred as a **sufficient** but not necessary test under RM schedulability.

Feasibility of EDF

A set of n periodic tasks is schedulable with EDF if and only if:

$$\sum_{i=1}^{n} U_i \leq 1$$

This feasibility test is also referred as a **sufficient and necessary** test under EDF schedulability.

Despite this the utilization factor is just a non-exact test to verify if a set of periodic tasks is schedulable with EDF, because if the response is non-negative, it still does not guarantee that all the tasks in the set are schedulable.
This statement has been demonstrated by numerous studies, using the **Worst-Case Response Time** (**WCRT**) analysis approach [JP86].

The worst-case response time of a task is the length of the longest interval from a task's release until its completion. The worst case response time R_i of a task τ_i is given by the smallest R that satisfies:

$$R^0 = C_i$$

$$R^{m+i} = C_i + \sum_{j<i} \left\lceil \frac{R^m}{T_j} \right\rceil C_j$$

The factor $\left\lceil \frac{R^m}{T_j} \right\rceil$ gives the worst-case number of preemptions that an execution of task τ_i suffers from task τ_j in an interval $[0, n)$ with $(m < n)$. The R is the interval between the release time and the end of the execution of the computation and it defines the WCRT. The computation is schedulable when the maximum response time R_i is less than or equal to its deadline ($R_i \leq D_i$).
This approach is not only suitable for Earliest Deadline First (EDF) scheduling but also provides a necessary condition for Rate-Monotonic (RM) schedulability, where in this scenario the deadline is equal to the period.

With this implementation, we offer the ability to conduct not only the standard Feasibility Test under RMS and EDF as seen previously, but also the option to perform worst Response Time analysis within a user program.

4.4 Execution Time Statistics

RealTimeScheduler gives the possibility for the user to run some statistics during the run of the task set. The library exploits the native `vApplicationTickHook()` function to keep track of the execution time of each task. This function runs periodically, once every tick. It operates within the context of the timer ISR, preempting even higher priority tasks. Due to its periodic execution, we can utilize it to poll for the necessary information. Every time this function is entered the execution time of the running task is incremented and stored in its custom TCB.
With `vIdleApplicationTickHook()`, instead, we are able to know if the CPU is IDLE and no tasks are ready to be executed yet.

For every task's execution the following information are printed as output:

- Start Tick Count
- Last Wake Time
- Task's name and priority (Absolute Deadline if EDF is active)
- Real Execution Time and comparison with WCET

```
--------*[IDLE]*--------
--------------------------------------------------------------------------------
Tick Count 104 Task Client3 lastWakeTime 0 Abs deadline 800 Priority 8

[CLIENT] File Transfer Status: 10%
[...]
[CLIENT] File Transfer Status: 90%
[CLIENT] File transfer completed!
[END] Execution time 419 (WCET: 400) - Task Client3
--------------------------------------------------------------------------------
```

5 Evaluation

All the tests below are related to this set of tasks.

Task	Start Time	Priority	Period	WCET
Ping Task	0	9	600	100
WGET	0	8	1200	400
FTP	0	8	1400	300

5.1 Base scheduling

The First demo, `FIXED PRIORITY NO PREEMPTION DEMO` utilizes the basic fixed priority scheduling without preemption. In this case, even with the best configuration, higher priority to the ping and lower priority to the FTP, the scheduling is not feasable. For this reason we activated the preemption.

If we activate preemption we have to insert the priority manually. The overall performance depends on our choice, the worst case happens when we assign the highest priority to FTP, since it has the longest period, and the lowest priority to the Ping. In this case the scheduling is not feasible. On the other hand if we assign the priority based on the period we obtain the same performance as Rate Monotonic as shown later on.

We tested Round Robin Scheduling (TQ = 50 Ticks) with the best configuration based on the Period.

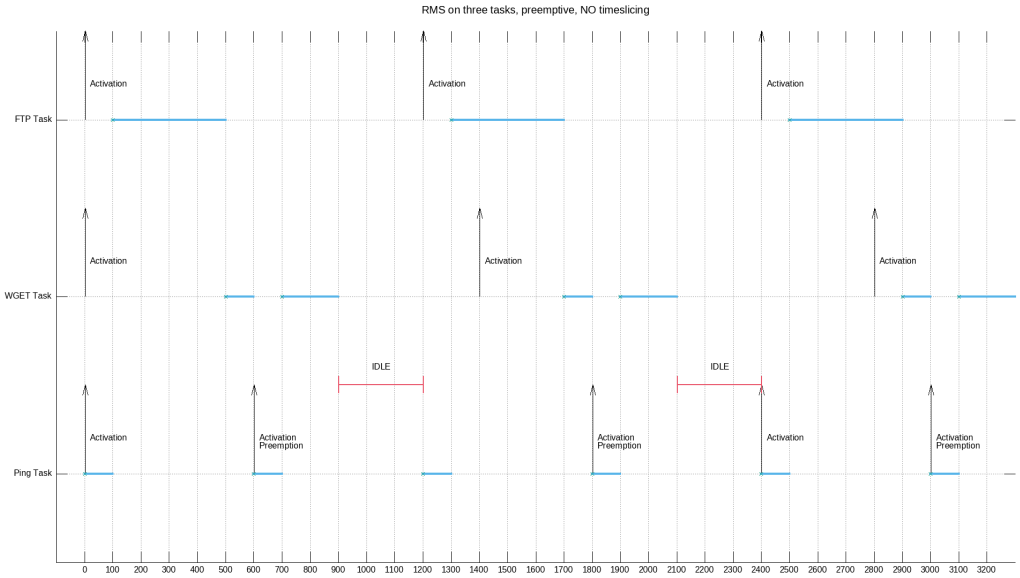
With an average Waiting time of 316 and an average Turn around time of 583 After the implementation of the Real time scheduling, the results of the tests are:

Task	Turnaround	Waiting time
Ping	100	0
WGET	900	500
FTP	750	450

$$\left[TAT_{avg} = \frac{1750}{3} = 583ticks, \quad WT_{avg} = \frac{950}{3} = 316ticks\right]$$

5.2 RMS

Assigning the priority manually may cause unfeasible scheduling so in order to automate the priority assignment we implemented RMS algorithm. The results are:



Task	Turnaround	Waiting time
Ping	100	0
WGET	500	100
FTP	900	600

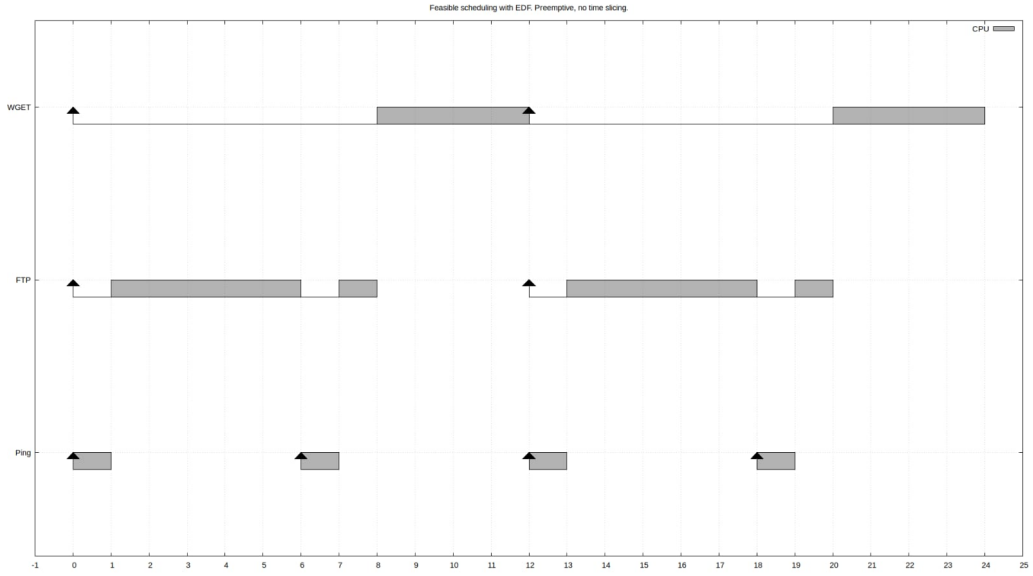
$$\left[TAT_{avg} = \frac{1500}{3} = 500ticks, \quad WT_{avg} = \frac{700}{3} = 233ticks\right]$$

Comparing these results with the Round Robin ones we have an improvement of 14% on average turnaround time and 27% on average waiting time.

5.3 EDF

After stressing RMS algorithm we found that some sets of tasks are not feasible. To solve this issue we implemented EDF scheduling algorithm to exploit its flexibility in handling unpredictable task sets and minimizing IDLE periods. We tried to run the following set of tasks:

Task	Start Time	Deadline	Period	WCET
Ping Task	0	400	600	100
WGET	0	1200	1200	400
FTP	0	800	1200	600

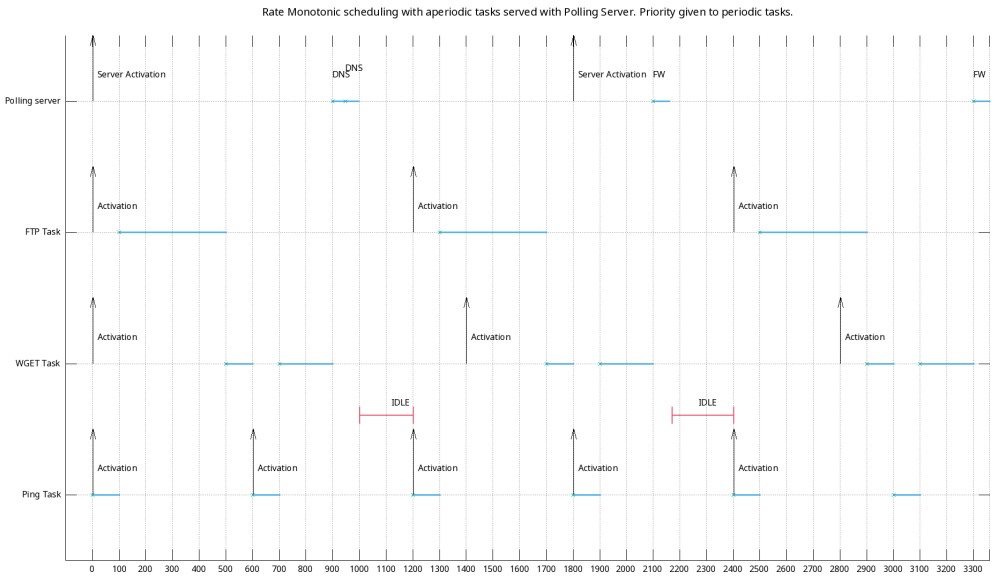


With this particular set of tasks RMS is not feasible while EDF is. This proves that since EDF uses a dynamic priority assignment it is more flexible.

5.4 RMS Polling Server

The polling server is added to handle aperiodic tasks. These are the results with the set displayed below.

Task	Start Time	Deadline	Period	WCET
Ping Task	0	400	600	100
WGET	0	800	1200	400
FTP	0	1200	1400	300
PS	0	1600	1800	100



Task	Turnaround	Waiting time
Ping	100	0
WGET	800	600
FTP	500	200
PS	1000	900

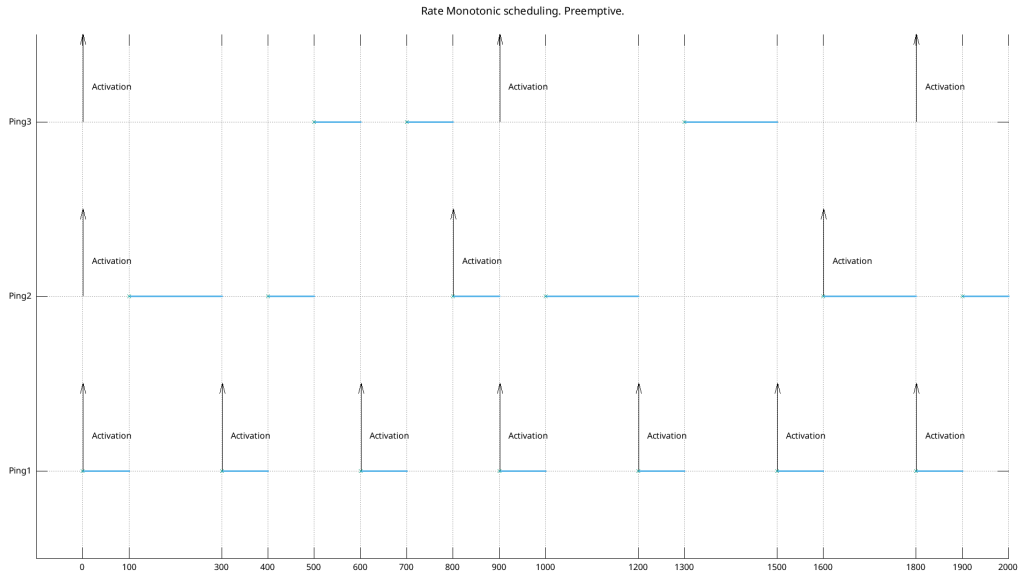
$$\left[TAT_{avg} = \frac{2400}{4} = 600ticks, \quad WT_{avg} = \frac{1700}{4} = 425ticks\right]$$

These results show that the introduction of the polling server ,in the same set of tasks seen before for RMS, lowers the performances of both turnaround time by 17% and waiting time by 46%.

5.5 RMS with WCRT

The feasibility test $U_{max} \leq 2$ for RMS is sufficient but not needed. When the test fails it possible to run WCRT test. To show this we used this set of tasks.

Task	Start Time	Deadline	Period	WCET
Ping Task	0	600	600	100
WGET	0	800	800	300
FTP	0	900	900	200



Even though the feasibility test failed the set is still feasible as proved by the WCRT algorithm.

References

[JP86] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 01 1986.