D2.11

Report on RTOS scheduling Revised

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D2.11
Deliverable name	Report on RTOS scheduling
Date	June 2020
Dissemination level	public
Workpackage and task	WP2 - Task 2.4
Author	Juan Flores Muñoz (eProsima)
Contributors	Francesca Finocchiaro (eProsima), Pablo Garrido Sanchez (eProsima)
Keywords	micro-ROS, robotics, ROS, microcontrollers, scheduling, real-time
Abstract	This document reviews the scheduling algorithms offered by the three RTOSes supported by micro-ROS and presents the scheduling architecture selected for the project.

Contents

1 Summary

This report is the sequel of the deliverable 2.10 (Report on RTOS scheduling-Initial) submitted in December 2018. Based on the previous deliverable, we will analyse the scheduling mechanisms that have been implemented within the micro-ROS project for each of the supported RTOSes, in order to achieve the required functionalities.

2 Acronyms and keywords

Term	Definition
RTOS	Real-Time Operating System
OS	Operating System
DDS	Data Distribution Service
ROS	Robot Operating System
MCU	Microcontroller unit
AL	Abstraction layer

3 Introduction

Schedulers are software entities that manage the execution of processes using different techniques depending on the objectives. Their main task is to decide which process to execute at each time in the processing unit.

This deliverable continues with the work done in the related deliverable D2.10 (Report on RTOS Scheduling - Initial). In the initial version of the deliverable, we proposed a draft idea of the task structure, then we performed an analysis of the scheduling algorithms that are usually available on an RTOS and finally we reported on the scheduling characteristics of the RTOSes supported by micro-ROS. As a conclusion, we drew a picture of which features could be implemented in the mature versions of micro-ROS. With the previously acquired knowledge and months of development, we achieved the results summarized in the present document.

In this deliverable, we will analyse the scheduling mechanisms of the RTOSes supported to date, study the scheduling architecture implemented to understand the decisions behind its implementation and finally comment on the results obtained and suggest possible future improvements.

4 Supported RTOS

In the initial release of this deliverable, our support was mainly focused on NuttX. However, we planned to extend the support to additional RTOSes once satisfactory results would be obtained with the NuttX. Since we were able to successfully achieve full support of Zephyr and FreeRTOS during this last year of development, in this revised version we have extended the scheduling analysis to these RTOSes as well.

We consider being able to support multiple RTOSes to be a remarkable achievement. However, this supposes an increased development complication since the three RTOSes supported present very different approaches in how they manage the system scheduling. We therefore present the scheduling mechanisms offered by the supported RTOSes, highlighting similarities and differences between them.

4.1 NuttX

NuttX offers a powerful set of scheduling techniques. This RTOS is compliant with Linux to a great extent, as it uses the POSIX standard as a development principle.

Following this POSIX standard principle, we can differentiate two entities that are run by the scheduler: tasks and threads.

- **Tasks**: The tasks have their own resources, comprised within a private memory range that only these tasks have access permissions to. Even though each task is isolated, they can interact with each other. Into the task group we can differentiate between two different kinds:
  - **Application tasks**: NuttX implements a concept called application, that is intended to be the RTOS equivalent of a typical OS application.
  - **Kernel task**: The kernel task controls the internal process of the RTOS allowing internal concurrent processes. This task is continuously on idle mode and performs its work only when no other task is requiring computational time.
- **Thread**: A thread is similar to a task, with the main difference that a thread runs inside of a task, sharing its resources. The threads, as well as the tasks, work in parallel but they can collaborate between them. Finally, the task scheduling mechanism is applied with the threads.

To handle both these kinds of entities, the RTOS uses the **FIFO** scheduling algorithm, which is a Fixed priority-based scheduling without preemptions. However, when two different entities have the same priority, NuttX uses **FIFO with Round Robing** scheduling to manage the situation.

4.2 FreeRTOS

FreeRTOS is a less complex RTOS in comparison with the other supported RTOSes. In FreeRTOS, there are two kinds of tasks: standard tasks and idle tasks.

- **Standard task**: This task is created by the user and can be considered as an application on the RTOS. It offers the following options to configure it:
  - Unique task name.
  - Priority task selection.
  - Task stack size.
- **Idle task**: This task is created and executed at the startup of the RTOS. It is the task with lowest priority and it enters into run mode only when there is no other task running. The objective of this task is to coordinate the execution of the RTOS resources and to clean-up those that have been used previously by other tasks.

To handle the task management, FreeRTOS uses a **Priority Based** scheduling and when different tasks have the same priority, it uses the **Round Robin** algorithm.

As a final point, we mention that an interesting add-on is available for FreeRTOS, which is called FreeRTOS+POSIX. This add-on implements a POSIX interface to control drivers, system resources such as timers and, most importantly, the system scheduling mechanism. This add-on has the merit to reduce the gap between the FreeRTOS scheduling management and that of the rest of RTOSes supported by micro-ROS.

4.3 Zephyr

Zephyr is an RTOS supported by the Linux Foundation which adopts by default the Linux POSIX API. The task management is similar to the one implemented by NuttX, with a kernel task that is in charge of managing the whole RTOS and giving the possibilty to create application tasks, which allow to run applications with their private memory stack.

On the other hand, this RTOS implements a wide variety of additional scheduling strategies. By default, it chooses the highest priority task and if multiple tasks with equal priorities are waiting, it chooses the one that has been waiting the longest.

However, Zephyr has the following scheduler algorithms available to use as well:

- Cooperative time slicing: Once a cooperative thread becomes the current thread, it remains the current thread until it performs an action that makes it unready. Consequently, if a cooperative thread performs lengthy computations, it may cause an unacceptable delay in the scheduling of other threads, including those of higher priority and equal priority.
- Preemptive Time Slicing: Once a preemptive thread becomes the current thread, it remains the current thread until a higher priority thread becomes ready, or until the thread performs an action that makes it unready. Consequently, if a preemptive thread performs lengthy computations, it may cause an unacceptable delay in the scheduling of other threads, including those of equal priority. To overcome such problems, a preemptive thread can perform cooperative time slicing (as described above), or the scheduler's time slicing capability can be used to allow other threads of the same priority to execute.

5 Scheduling architecture

In the deliverable 2.10 a multi-thread structure was proposed, based on the following points:

- Communication task: Implements the communication stack to achieve communication between micro-ROS devices.
- Inner communication tasks: Task focused on obtaining sensor measure or to activate an actuator.
- Processing tasks: Runs the micro-ROS logic.
- Synchronization task: Performs the synchronization between the different tasks to achieve a concurrent process, avoiding blocking situations.

- System control tasks: Default RTOS kernel task, which manages the different peripherals and resources of the system. This task is not created by the user.

This study was created at a very early stage of the project which was heavily work in progress without stable results. After this development time, we re-thought the scheduling architecture and we applied a single thread architecture instead of the multi-thread structure.

The reasons to use a single thread architecture are the following:

- **Simplify code development**: Avoiding the multithreading architecture, we avoid possible synchronization problems which can slow the performance of Micro-ROS or even block the entire system.
- **Achieve good performance on single-thread mode.**
- **Easier debugging process.**
- **Allows port to bare-metal environments**: On bare-metal development, it is not possible to use multiples tasks, due to the lack of a scheduler that controls the threads. This could be useful on secure compliant MCUs which in general don't use any RTOS or at most use some very simple RTOS.

6 RTOS implementation

In the initial deliverable, the implementation of an AL which unifies the scheduler control on each supported RTOS was proposed. The reason of such a suggestion was the lack of knowledge of the different RTOSes. After months of development and with better knowledge of how the three supported RTOSes work, we decided to discard the idea of an AL that unifies the usage of the scheduler and task creation.

As it was described above, we're giving support to NuttX, FreeRTOS, and Zephyr. All of them implement the POSIX scheduling management (even though in the FreeRTOS it is necessary to add an extension to achieve it). As a consequence, if we use only the POSIX API, we already have an API that unifies the scheduler management.

Thanks to this, it is not necessary to develop any extra layer which can be a very tedious job, since in this case modification of some of the core features of the RTOSes would be necessary. A positive aspect of using of the POSIX API is that it gives full Linux support without applying any extra effort to perform development for this specific OS.

To conclude, we mention that this choice also solves the problems that could arise with a multi-task architecture in the case of a bare-metal port. Indeed, thanks to the single thread policy, the scheduler manages the whole micro-ROS stack and as a consequence it is equivalent to running on the main loop of a MCU.

7 Conclusion and future steps.

We can conclude that the selected approach is the most desirable. We focused on developing a simple but at the same time efficient code instead of creating an overly complex task architecture,

which could result in heavy synchronization and porting problems. This gives as a result a good and stable micro-ROS behavior and, on the other hand, the simplification of the port process to other RTOSes, which is one of the crucial points of the proposal. Finally, the performance test is still work in progress, but we are confident about the good results.

It is possible that in the future it will be necessary to add some multi-threading add-on due to the executor's requirements, but the core architecture of micro-ROS, which this deliverable is mainly focused on, will continue running on a single thread.