BEng Thesis

Analysis of MrsP Protocol in RTEMS Operating System

Orientação científica: Cláudio Maia

Ricardo Miguel Gomes

CISTER-TR-191210

Analysis of MrsP Protocol in RTEMS Operating System

Ricardo Miguel Gomes

CISTER Research Centre

Polytechnic Institute of Porto (ISEP P.Porto)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail:

https://www.cister-labs.pt

Abstract

The technological world is becoming increasingly familiarized with the usage of multiprocessing platforms on several types of devices. The capabilities provided by multiprocessor systems allow for the increase of the overall system 19s performance, enabling the execution of several tasks in a concurrent and parallel way. While this behaviour is beneficial for most systems, it has to be handled with care in the real-time systems domain. Real-time systems are well known for their challenging requirements, the most important of them being the need to guarantee that operations are completed within a given deadline. Thus, several efforts are being made in order to be possible to execute them in multiprocessor environments. However, there are several barriers that make the transition difficult, notably at the level of resource sharing mechanisms and process scheduling on a system.One of such efforts is the added support brought by the inclusion of several scheduling and task synchronization mechanisms to RTEMS (Real-time Executive for Multiprocessor Systems), an open source real-time operating system widely used by the space industry worldwide.RTEMS supports two multiprocessor shared resource protocols, OMIP (O(m) Independence-Preserving Protocol) and MrsP (Multiprocessor Resource Sharing Protocol). From these two, MrsP is considered rather promising, largely covering the requirements imposed by real-time multiprocessing systems. This protocol was successfully implemented in RTEMS but recently it was subject to some refinements in order to cover some flaws on its initial proposal.In the project presented in this report, the MrsP protocol is analysed from a theoretical point of view and this view is compared against the implementation of the protocol on RTEMS. The goal is to evaluate the correctness of the implementation, according to the proposed requirements. Additionally, a testbed is developed to test the protocol in its integrity in order to serve as a regression test platform and in some cases, where it makes sense, comparisons with the OMIP protocol are established.

DEPARTAMENTO DE ENGENHARIA
INFORMÁTICA

Analysis of MrsP Protocol in RTEMS Operating System

CISTER

2018 / 2019

1161078 Ricardo Gomes

isep Instituto Superior de Engenharia do Porto

Analysis of MrsP Protocol in RTEMS Operating System

CISTER

2018 / 2019

1161078 Ricardo Gomes

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Licenciatura em Engenharia Informática

Setembro de 2019

Orientador ISEP: **Cláudio Roberto Ribeiro Maia**

«To my family, for making this possible»

Acknowledgements

I want to thank my family, girlfriend and my closest friends, that always supported me through my academic course, during the good and bad moments.

I also must thank my supervisor, Cláudio Maia, for the invitation to this project and for being always available to keep up with me during this internship. Joel Pinto, a researcher on CISTER, that also helped me understanding some concepts related to this project and finally all the CISTER members that received me here as best as possible and have always been nice to me.

Abstract

The technological world is becoming increasingly familiarized with the usage of multiprocessing platforms on several types of devices. The capabilities provided by multiprocessor systems allow for the increase of the overall system's performance, enabling the execution of several tasks in a concurrent and parallel way. While this behaviour is beneficial for most systems, it has to be handled with care in the real-time systems domain. Real-time systems are well known for their challenging requirements, the most important of them being the need to guarantee that operations are completed within a given deadline. Thus, several efforts are being made in order to be possible to execute them in multiprocessor environments. However, there are several barriers that make the transition difficult, notably at the level of resource sharing mechanisms and process scheduling on a system.

One of such efforts is the added support brought by the inclusion of several scheduling and task synchronization mechanisms to RTEMS (Real-time Executive for Multiprocessor Systems), an open source real-time operating system widely used by the space industry worldwide.

RTEMS supports two multiprocessor shared resource protocols, OMIP (O(m) Independence-Preserving Protocol) and MrsP (Multiprocessor Resource Sharing Protocol). From these two, MrsP is considered rather promising, largely covering the requirements imposed by real-time multiprocessing systems. This protocol was successfully implemented in RTEMS but recently it was subject to some refinements in order to cover some flaws on its initial proposal.

In the project presented in this report, the MrsP protocol is analysed from a theoretical point of view and this view is compared against the implementation of the protocol on RTEMS. The goal is to evaluate the correctness of the implementation, according to the proposed requirements. Additionally, a testbed is developed to test the protocol in its integrity in order to serve as a regression test platform and in some cases, where it makes sense, comparisons with the OMIP protocol are established.

Key words (Theme): Real-Time Systems, Synchronization Mechanisms, Resource Sharing Protocols, Testbed.

Key words (Technologies): RTEMS, MrsP, SMP, semaphores.

Resumo

O mundo tecnológico começa a familiarizar-se com o uso de plataformas multiprocessador nos mais diversos tipos de dispositivos. Estas capacidades, proporcionadas por sistemas multiprocessador, permitem um aumento geral da performance dos sistemas, possibilitando a execução de diversas tarefas de forma concorrente e paralela. Apesar deste comportamento ser benéfico para a maioria dos mesmos, no domínio dos sistemas em tempo real, estas propriedades devem ser manipuladas com cuidado. Os sistemas em tempo real são conhecidos por apresentarem requisitos bastante desafiantes, sendo o mais importante a necessidade de garantia de que as suas operações são concluídas tendo em conta um tempo limite para a realização das mesmas. Sendo assim, têm sido feitos grandes esforços para que seja possível a sua execução em ambientes multiprocessador. No entanto, existem diversas barreiras que dificultam esta transição, essencialmente a nível de mecanismos de partilha de recursos e no escalonamento dos processos do sistema.

É considerado um desses esforços o suporte adicional trazido através da inclusão de vários mecanismos de sincronização e escalonamento de processos no RTEMS (Real-time Executive for Multiprocessor Systems), um sistema operativo em tempo real utilizado mundialmente pela indústria aeroespacial.

O RTEMS suporta dois protocolos multiprocessador de gestão de recursos partilhados, OMIP (O(m) Independence-Preserving Protocol) e MRSP (Multiprocessor Resource Sharing Protocol). MrsP é considerado bastante promissor, cobrindo amplamente os requisitos impostos por sistemas multiprocessador em tempo real. Considera-se também este protocolo como implementado com sucesso no RTEMS, no entanto, recentemente, foi sujeito a alguns aprimoramentos, de forma a cobrir certas falhas na sua proposta inicial.

No projeto aqui apresentado, o protocolo MrsP foi analisado do ponto de vista teórico e, com base nessa análise, foram estabelecidas comparações com a sua implementação. O objetivo deste trabalho é avaliar a implementação do protocolo, de acordo com os requisitos propostos para o mesmo. Para além disso, uma plataforma de testes foi desenvolvida de forma a verificar a integridade do protocolo, a fim de servir como plataforma de teste de regressão tendo sido estabelecidas comparações com o protocolo OMIP.

Palavras-chave (Tema): Sistemas em tempo real, Mecanismos de Sincronização, Mecanismos de Gestão de recursos partilhados, Testbed.

Palavras-chave (Tecnologias): RTEMS, MrsP, Multiprocessador, Semáforos.

Index

Index of Figures

Index of Tables

Acronyms

CISTER	Centro de Investigação em Sistemas Computacionais Embebidos e de Tempo-Real
CPU	Central Processor Unit
DPCP	Distributed Priority ceiling Protocol
EDF	Earliest Deadline First
ESA	European Space Agency
FMLP	Flexible Multiprocessor Locking Protocol
Litmus	Linux Testbed for Multiprocessor Scheduling in Real-Time Systems
MPCP	Multiprocessor Priority Ceiling Protocol
MrsP	Multiprocessor Resource Sharing Protocol
MSRP	Multiprocessor Stack Resource Policy
OMIP	O(m) Multiprocessor Independence Protocol
OS	Operating System
PCP	Priority Ceiling Protocol
PIP	Priority Inherence Protocol
QEMU	Quick Emulator
RSP	Resource Sharing Protocol
RTEMS	Real-time Executive for Multiprocessor Systems
RTOS	Real-time Operating System
RTS	Real-time system
SMP	Symmetric Multiprocessor
SPEPP	Spinning Processor Executes for Preempted Processor
SRP	Stack Resource Policy

1 Introduction

The project entitled "Analysis of MrsP Protocol in RTEMS Operating System" is described in this report and was performed within the scope of the course named Projeto/Estágio (PESTI) of the Computer Engineering Degree in Instituto Superior de Engenharia do Porto (ISEP). The Research Centre in Real-Time Embedded Computing Systems (CISTER), a research laboratory of ISEP, is the proponent organization of the work described in this report.

Nowadays, multiprocessor systems are commonly found in several electronic devices. For instance, most of the current personal computers and smartphones are great examples of devices that integrate multiprocessors. One of the greatest advantages of multiprocessor systems is that they allow the execution of several applications not only at a concurrent level, as it happened with uniprocessor systems, but also at a parallel level. When a computer system has two or more processors (CPUs), it can do work simultaneously in all processors, i.e., in parallel, which increases the performance of the system. However, in these systems, synchronization, the coordination between several events and processes of a system, becomes even more important and challenging than in a single processor system, especially on stringent domains such as the real-time system domain, where human lives may be at stake. Real-time systems are the focus of this report.

Real-time systems are systems that must react to internal or external events (i.e., events coming from the physical world) in a timely manner, while assuring other non-functional aspects of the system, such as reliability, dependability and security. Otherwise, it is considered that the system fails. These systems are commonly used in areas where there is a high degree of criticality involved, as for instance the automotive and aeronautic industries. Examples of real-time systems are the airbag of a car or the flight control system of an aircraft.

Since real-time applications have demanding requirements in terms of performance, multiprocessor support is a relevant feature that can help in achieving efficient real-time systems. However, there two major concerns regarding multiprocessor support for real-time systems: (1) the scheduling of applications; and (2) support for resource sharing and synchronization protocols.

Synchronization mechanisms are important on systems with only one processor (where concurrency is a key aspect for the correctness of the system) but when dealing with concurrency and parallelism, as in the case of multiprocessors, there might be several tasks[1] that try to access a resource at the same time instant. Thus, this kind of operations must be protected in order to guarantee data consistency and determinist output, otherwise data might become corrupted. This aspect is even more important in real-time systems due to their critical nature.

Over the years, there has been an effort to ensure the correct implementation of scheduling and synchronization mechanisms in real-time operating systems (RTOS) executing on top of multiprocessor architectures. In particular, support for multiprocessor scheduling is already beginning to be considered as successfully adapted. However, research on resource sharing protocols is still an important on-going research topic [1] in order to grant synchronization between several processes of a system that try to manipulate the same resource concurrently and in parallel.

In this report, a shared resource is defined as something, such as a block of memory, a hardware device or a piece of code (as for instance global variables) that may be accessed by several processes of the system in a concurrent/parallel way. Usually, the access to a shared resource must be performed synchronously, that is, in a mutually exclusive manner, in order to maintain the reliability and consistency of the system.

There are several synchronization protocols that have been proposed in the recent years to solve the problem of controlling access to shared resources in multiprocessor systems. For instance, O(m) Independence-preserving Protocol, Distributed Priority Ceiling Protocol [2] and Multiprocessor Resource Sharing Protocol. From these protocols, MrsP [1] is a very promising resource sharing protocol which has been successfully implemented in two RTOSes [3], namely RTEMS and Litmus, and is being continuously studied and enhanced [4] [5].

1.1 Problem Description

Real-time operating systems (RTOS) are operating systems that offer features that allows one to control the deterministic execution of applications as well as other non-functional features such as timeliness[2] or reliability[3]. Consequently, RTOSes offer a different behaviour than general purpose operating systems. While RTOSes must provide a deterministic response and

[1] A task is usually considered to be an entity that can be scheduled by an operating system, that is, a schedulable entity. Composed by a sequence of instructions executed in a concurrent way, in the context of this work, tasks may be programs, processes or threads.
[2] Timeliness is defined as the quality of the system in being correct with respect to the time domain.
[3] Reliability is defined as the quality of a system in being trustworthy and consistent.

guarantee the deadlines of any given task, a general purpose system does not necessarily have to provide guarantees of when a task completes, and therefore is unsuitable for developing real-time systems.

RTEMS is one of the RTOSes that offers support for multiprocessor systems, with several scheduling algorithms and synchronization protocols being supported. In fact, the multiprocessor version is now undergoing through a qualification activity in order to be used in the European Space Agency (ESA) space missions [6].

One of the synchronization protocols supported by RTEMS is the Multiprocessor Resource Sharing Protocol (MrsP) protocol [1]. MrsP is a lock-based protocol that, from a theoretical point of view, covers a wide range of issues imposed by the use of synchronization mechanisms in RTOSes that support symmetric multiprocessor systems, that is, systems that use two or more identical processors that share a common main memory and where the operating system treats all processors in an identical manner.

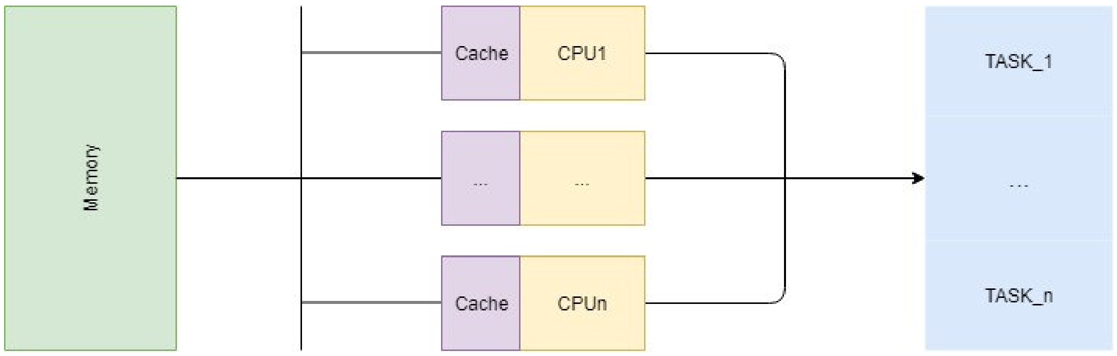


Figure 1 Example of a SMP system

MrsP is a generalized variant of the Priority Ceiling Protocol and the Stack Resource Policy, which are single processor protocols, adapted to multiprocessor real-time systems. The initial implementation of the protocol in RTEMS [3] did not properly cover certain aspects related to its functionality, such as the use of nested resources, i.e., the access to resources within other resources. However, later, the proponents of the initial version of the protocol correctly addressed these aspects, surpassing the flaws previously presented [4].

Thus, the question arises as to whether the current version of the protocol is implemented correctly and if it fulfils all the requirements. This study is important, especially considering that RTEMS for multiprocessors is undergoing qualification by ESA [6]. In this way, this project aims to validate the functionalities of MrsP and assess if the protocol is correctly developed such that it may be safely used as resource sharing protocol on real-time systems.

1.2 Approach

This project requires knowledge in three main areas, namely, real-time systems, real-time operating systems and synchronization protocols. Therefore, the approach to follow needs to cover the main aspects in each of these areas in order to be possible to understand the implementation of synchronization protocols for multiprocessor systems in RTEMS.

In order to approach this project correctly, there is the need to understand the most important properties of real-time systems. For that, a study about the theory of real-time systems will be carried out first. Following this study, the focus will be on the general RTOS concepts, in particular about RTEMS, in order to understand its structure, main functionalities and features and how to handle them.

After the main concepts are understood, a more detailed analysis, using research papers and RTEMS documentation, is carried out concerning single processor synchronization mechanisms, like the priority ceiling protocol[4]. This way, the single processor resource sharing protocols supported by RTEMS have been understood. Additionally, as this work focuses on multiprocessor systems, multiprocessor concepts used by the operating system have to be deepened, in order to properly address the resource sharing protocol covered in this work. Afterwards, the study of the MrsP must be performed, initially in a theoretical way, and then, all the knowledge needs to be mapped into the RTEMS in order to perform the necessary project tasks. These tasks involve the assessment of the MrsP protocol's functionalities [6] so as to test it in order to infer current coverage and timing performance considering several synchronization scenarios.

1.2.1 Working Methodology

The methodology that is applied in this work is based on the Waterfall methodology [8]. Although Agile methods [9], such as SCRUM [10], have been emphasized and widely covered in ISEP's Computer Engineering degree, the nature of this project justifies a sequential work model.

Waterfall is a sequential work methodology, characterized by a set of well-defined and non-repetitive phases. This model's objective is the flow between each phase, reminding a waterfall. Although it may depend on each project, this model is usually composed by six

[4] The Priority Ceiling Protocol is a resource sharing protocol explained in more detail in chapter 2

phases, namely: Requirements, Design, Implementation, Tests, Integration and Maintenance, as described in the following image.

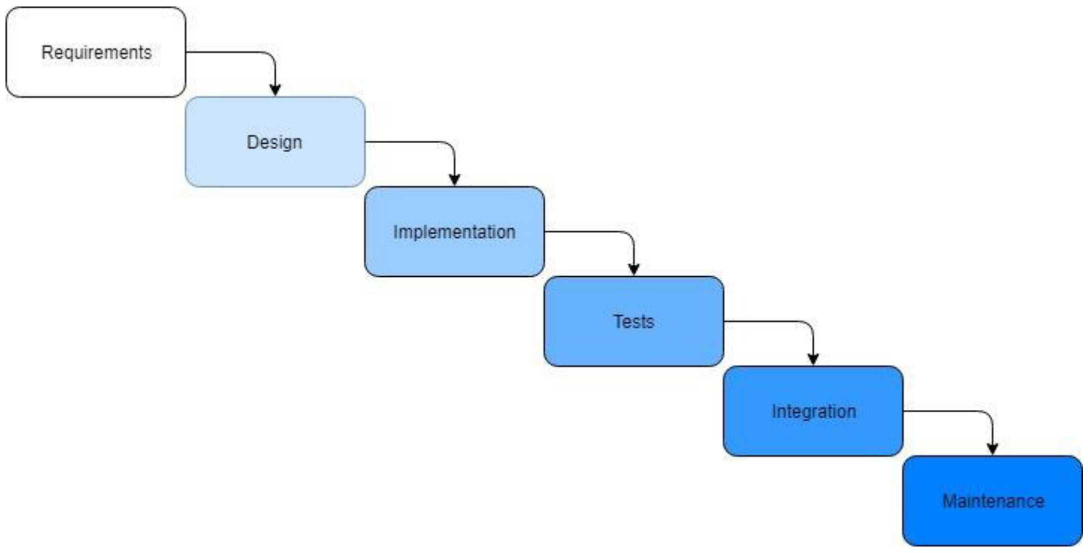


Figure 2 Waterfall Approach Diagram

The Requirement Analysis phase, the first phase of the model, is composed by the elaboration of responsibilities required to satisfy the requests imposed by the stakeholders, defining what must be done by the system. After that, the Design phase, with the requirements already identified, the designer must define how the project is going to be implemented, defining all the components needed by the system to produce the desired product.

The Implementation phase, starts after the design phase is completed, is the phase where the architecture and design of the system is implemented, building the functional system itself, followed by the Test phase, where all the developed functionalities are tested in order to assert if they meet the imposed requirements.

Finally, the Integration phase is responsible for the deployment of the system in the environment where it operates, making it completely operational for the customer, followed by the final stage, named Maintenance, where the development team follows the system during its lifetime, in order to verify if it presents any error and thus repairing it, or, on other hand to produce enhancements on it.

Some of the main advantages of the Waterfall approach, when compared to Scrum, an agile methodology [11], are that in Waterfall the objectives and the working plan are detailed carefully in the beginning of the project and no changes are further envisioned, allowing the team and the project stakeholders to clearly understand what will be delivered in the end of the project.

Scrum methodology is adequate to more dynamic scenarios where new requirements are always dependent on the features that are already implemented in the software under development. For instance, scenarios where during the development phase of the project the requirements of a project may change, removed, or even added.

In this project, all the requirements are well defined and the probability of considerable changes is quite low. Therefore, it is not justified the use of an agile approach, and the Waterfall methodology is perfectly suitable for the project.

1.3 Objectives and Contributions

The main goal of the project is to verify the correctness of the synchronization mechanisms of RTEMS, in particular the study and evaluation of the multiprocessor resource sharing protocol MrsP. In this way, it is expected, in the first place, that an analysis of the theory behind MrsP is performed in order to assert the protocol’s fundamental properties and behaviour. Then, a study of the variation of the protocol in RTEMS must be proceeded, mapping its desirable behaviour, with its implementation in the operating system. Finally, this work proposes to develop a testbed that completely covers all the functionalities of the protocol. The testbed must be developed in RTEMS and shall abide by the specification of the protocol, with the goal of evaluating its implementation in this operating system.

With this testbed all the properties of MrsP may be analysed and evaluated, so that a user or a developer may trust the protocol implementation whenever they want to use it in their own projects. Moreover, the testbed may be used as a regression test platform in order to verify if the protocol still works properly, after changes are made during the software development lifecycle.

The provided analysis, described on this report also helps anyone who wants to explore this synchronization mechanism on RTEMS to understand the way it works, also providing comparisons with the other resource sharing protocol implemented on RTEMS, OMIP. Finally, the final product of this project is intended to be shared with the RTEMS user and developer community.

1.4 Work Plan

In this chapter, the work is divided into tasks and the expected time to accomplish each task is presented. A Gantt chart with the specific planning is presented on the appendixes, referred to the following table.

Table 1 Work Plan

Task	**Work**
RTEMS Study	**30d**
RTEMS Operating System General Study	15d
RTEMS Uniprocessor Tests Study	5d
RTEMS Multiprocessor Concepts	5d
RTEEMS Multiprocessor Tests Study	5d
MrsP Study	**30d**
MrsP Proposal Study	15d
MrsP Implementation Analysis on RTEMS	15d
RTEMS Tests and Analysis	**65d**
Testbed Development	45d
Instrumentation Testing	5d
Analysis of Results	15d
RTEMS Setup on Raspberry Pi2	**15d**
Final Report Development	**150d**

1.5 Report Structure

This report is structured in the following manner:

Chapter 1, introduces the work and contextualizes the problem, also explaining the approach chosen, the work methodology and the contributions.

Chapter 2, presents the state of the art. The main research works related to the work that is described in this reported are presented. Moreover, a description of related technologies is given, including the motivation for selecting the tools adopted in the project.

Chapter 3 introduces the most relevant features about MrsP and OMIP. Those are the protocols used to support this project, since they are the existent resource sharing protocols on RTEMS. It this way, the description of those is separated and more deeply approach on this chapter.

Chapter 4, some of the most relevant aspects about RTEMS, the RTOS that supports this project, are explained. In this way, the reader may be aware of certain fundamental aspects relevant for the work presented on this report.

Chapter 5, the analysis the implementation of MrsP on RTEMS. This chapter explains the way MrsP operates on its most relevant internal operations.

Chapter 6, presents a full testbed asserting MrsP's desired properties, accompanied with a comparison of OMIP under the same circumstances. This comparison is established through the protocol's behaviour analysis and time measures.

Chapter 7, represents an approach to the analysis of the time performances of MrsP and OMIP internal operation in order to assert and justify obtained differences between both protocols.

Chapter 8, the conclusions, presents a summary of the work described on this report and the respective limitations. Finally, it also describes the author's opinion and a synthetization of the obtained results during the project.

2 State Of The Art

Multiprocessor systems have become a common technology over the last few years. However, there are still several challenges that need to be addressed in order to be able to use this type of solutions in a reliable way on real-time systems. In some cases, the generalization of the functionalities of uniprocessor systems into multiprocessor systems is possible, as for instance the case of partitioned scheduling. Due to its importance, several developers and researchers [12], [13] have been focusing on the development of symmetric multiprocessor (SMP) support within the scope of real-time operating systems (RTOS).

One of the biggest issues in the development of real-time multiprocessor systems is the support for synchronization and resource sharing protocols, which are considered crucial components for any RTOS. In these systems, there are several resources that may be accessed by several tasks concurrently at the same time (known as shared resources) and without proper synchronization mechanisms it is impossible to assure a correct behaviour on the access to those resources. Thus, to guarantee the correctness of the operations performed by each task, resource sharing protocols must be used.

Resource Sharing Protocols for uniprocessor systems are well understood and are generally supported by a RTOS, such as the priority ceiling protocol (described in the next section). However, for multiprocessor systems, these protocols are not yet in a mature phase of development, and many of them are still under scrutiny of the community that needs them, either at a scientific level or at an industrial level.

2.1 Resource Sharing in Uniprocessor Systems

In this section of the report, an approach to the single processor resource sharing protocols on which MrsP and OMIP are based is presented. In the first place, the behaviour of those protocols is explained along with an illustrative example. The examples represent how tasks access a shared resource in a single processor system using the protocol under description.

Concerning the properties of each task, there are two that must be defined for better understanding the protocols, the task priority and the task execution time. The task priority is a value (an integer) that represents the order in which the task will execute. Thus, higher priority tasks must execute before lower priority tasks. Throughout this report, tasks with lower priority value are the ones with higher priority. The execution time represents the time a task needs to complete its execution, running in isolation.

2.1.1 Priority Ceiling Protocol

The priority ceiling protocol (PCP) is a uniprocessor locking protocol designed for systems that use fixed-priorities (also known as a fixed-priority system where task priorities do not change throughout the execution of the system) in order to select the executing task. For this protocol a priority is set for the shared resource, known as the ceiling priority. Then, when a task locks the resource, its priority is immediately raised to the ceiling priority. In the end, upon the resource release, i.e., when a task finishes the use of it, the priority of the owner thread is restored to its initial priority.

In order to synthetize the concepts addressed on this chapter, examples for both protocols are presented, starting with the protocol described above.

In this example two tasks are considered with execution time of 5 time units, task A with priority 9, ready to execute at time instant 0 (T0) and task B with a higher priority, 7, ready to execute at time instant 2 (T2). Both tasks execute on a single processor system and both must access the resource R with a defined ceiling priority of 5.

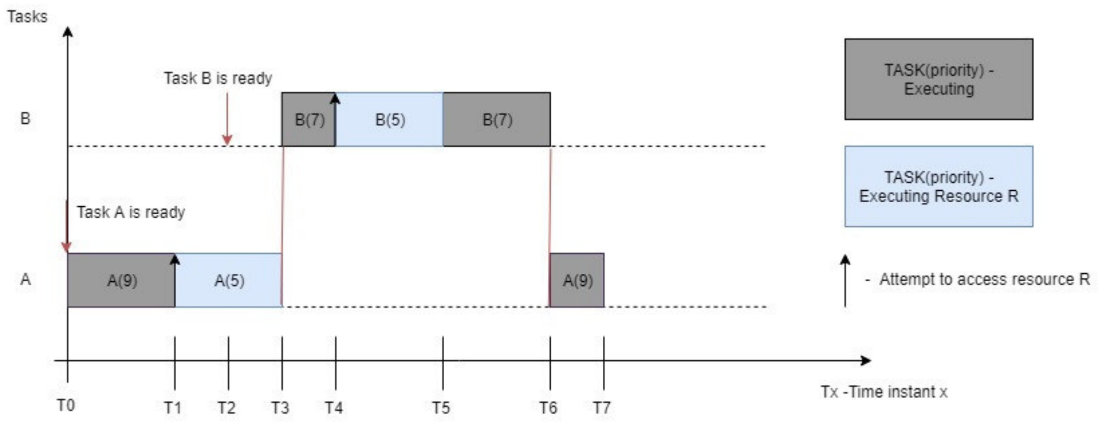


Figure 3 Example of the Priority Ceiling Protocol behaviour

As described above, task A is ready at T0, so it starts running and at T1 it obtains the resource and its priority is raised to 5. At time instant 2, task B is ready but it must wait until task A finishes the execution of the resource since the priority of A is raised to 5. With the execution of R finished, A’s priority changes to the initial priority of 9, which is less than the priority of B. So, B starts executing until it finishes its job, with the priority raised to 5, while it is holding the resource R, from T4 until T5, and task A remains preempted. At the time instant 6, task B’s execution ends and task A is dispatched, executing until T7.

2.1.2 Stack Resource Policy

The stack resource policy (SRP) [14] can be described as a variant of the PCP adapted to systems that work with dynamic priorities, in this case, it means that the priority of a task is equal to the time left to complete the execution. The SRP can also be applied to fixed-priority system, being referred as an emulation of the priority ceiling protocol.

Since this protocol behaves like PCP, when a task obtains a shared resource under the stack resource policy, its current priority must be raised to the ceiling priority of the protocol, during the execution of it. Once the task frees the resource, its priority must be replaced with its original one, the time left for it to execute.

The example referred to SRP, consists of a task A that executes for 8 units of time, a task B that executes for 3 units of time and the resource R that needs 3 units of time to be completely executed. The execution time is the priority of a task that is going to start its execution.

Task B presents a greater priority than task A and the resource R has greater priority than task B. Task A is ready for execution at T0 (instant 0), using the resource R for 2 time units, after starting its execution, and B is only available at T3. The expected behaviour of the system, using SRP is described on the next image.

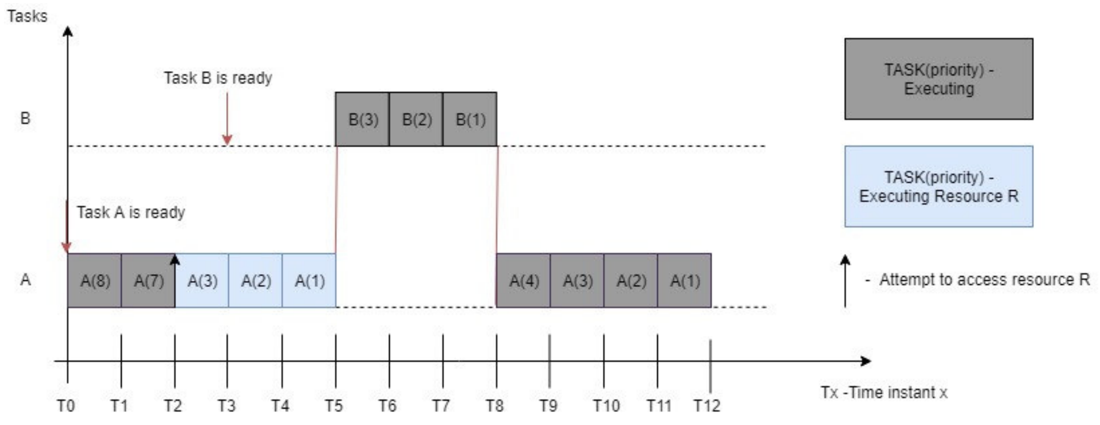


Figure 4 Example of Stack Resource Policy behaviour

Since task B, that has higher priority than A, is only available at the instant T3, 3 time units after A, task A starts its execution earlier, at T0.

At T2, task A obtains the resource, that needs 3 time units to be executed, so the priority of A is raised to 3 and decreased every time a time unit passes, since the priorities are dynamic. Moreover, at T3, task B becomes ready to execute, although, it cannot execute instead of A, because A is executing the resource R, and has its own priority raised to the priority ceiling of the resource, 2, currently higher than the priority of B, 3.

Finally, when A releases the resource, its priority is restored, to 4, the remaining execution time of it. Task B preempts it and starts, because B needs 3 time units to end its execution, while A needs 4, a greater value. In the end, when B finishes its execution, at T8, task A is dispatched and finishes its job, until T12.

2.1.3 Priority Inheritance Protocol

The priority inheritance protocol (PIP) is a single processor locking protocol, the most relevant alternative to PCP, described above. This protocol manages the access to a resource by the order of the priority of the tasks trying to obtain it. The main difference between this protocol and the protocols described above is that, instead of using a ceiling priority, on this case, if a task with greater priority that the owner of the resource tries to access it, its priority is inherited by that same owner, becoming able to execute again.

As in the protocols described above, an example is elaborated, based on the priority ceiling's example, in order to help the reader understanding in the way PIP works.

In this case, there are two tasks, Task A and Task B, with priority levels of 9 and 7, respectively, considering that the higher the level the higher the priority, each one of them with execution time of 5 units. Task A becomes ready to execute at time instance 0 and Task B on time instance 2. Both tasks try to execute resource R, that takes 3 time units to execute.

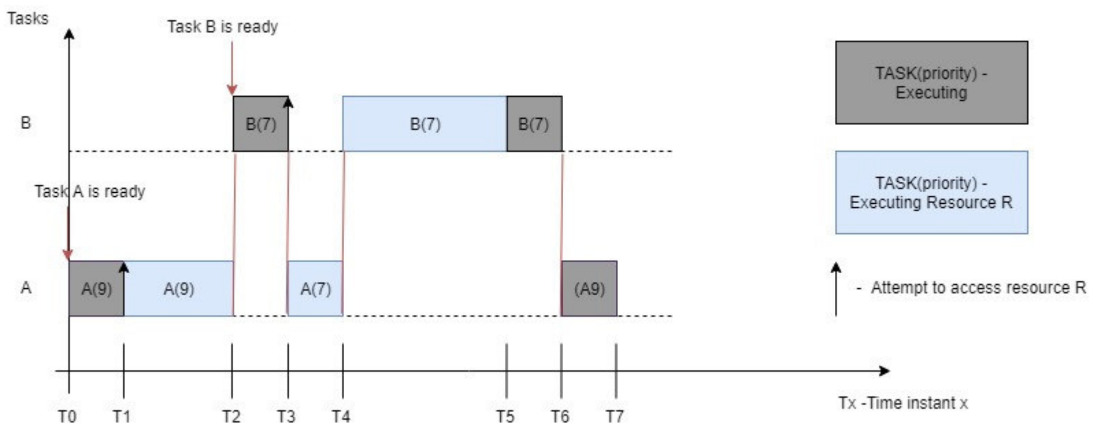


Figure 5 Priority Inheritance Example

At time instance 0 Task A enters the system and starts executing with a priority of 9. Meanwhile, at T1 it obtains resource A and starts continues executing, until T2, when Task B with a greater priority arrives, preempts task A, and starts executing until it also tries to obtain the resource on T3. Since the resource is already held by Task A, in order to let A's execution proceed Task A must inherit Task B's priority until it releases the resource, at T5.

Finally, when Task A releases the resource its priority is changed to the original one, 9. At that instant, the resource becomes free and Task B obtains it and executes until T9. After that, Task A is again able to execute since there are no tasks on the system with higher priority, finishing its execution.

2.2 Resource Sharing in Multiprocessor Systems

This section briefly describes and analyses the most important protocols for resource sharing in multiprocessor real-time operating systems. Since MrsP and OMIP are used to support this project, it makes sense to address these protocols in more detail. Therefore, they are described on the next chapter separately.

In order to assist the analysis of the protocols, a table is used to show the fulfilment of a set of important features for the resource sharing protocols under analysis. Any of these features may be filled with "✓" (fulfils the feature) or "X" (does not meet the feature). The features approached in this report were chosen based on the research about shared resources and the chosen resource sharing protocol, MrsP, namely:

- **Static Priority Scheduling support:** A protocol that allows the use of a scheduling mechanism where the task priority will be the same during its entire lifetime;
- **Dynamic Priority Scheduling support:** Contrarily to the Static Priority Scheduling, in Dynamic Priority Scheduling the priority of a given task may change over time, for example, because of the approaching of its deadline. It is also important that exist resource sharing protocols that cover systems using this kind of priority;
- **Global Resources Support:** A resource sharing protocol that supports the existence of global resources allows a certain resource to be accessed by several processors of the system;
- **Preemptive Waiting Mechanism:** In certain mechanisms, when a task is waiting to access a resource it performs a busy wait, which means that it never stops executing, while the resource is being held by another task. This waiting mechanism also must be preemptive, in order to avoid blocking the execution of higher priority tasks;
- **Nested Resources Allowance:** A nested resource is a resource that is accessed inside another shared resource. Depending on the protocol architecture, nested resources may or may not be allowed.
- **Helping Mechanism:** Ability to provide help to a task that has a resource, for example if it is preempted. In this way, the release of the resource will be less time consuming and will prevent the blocking of tasks waiting for access to the resource.

2.2.1 MPCP

The Multiprocessor Priority Ceiling Protocol [15] is an extension of the Priority Ceiling Protocol to multiprocessor systems. Initially, MCPC relied on the use of a single synchronization processor to deal with all the operations related to shared resources. Although, on the same proposal, this protocol was redesigned in order to allow several synchronization processors, by not allowing distribution of a resource to several processors, each resource is always treated in the same synchronization processor.

The initial architecture of the MPCP allows nested resources. however, the changes proposed in its redesign do not allow it anymore, because one resource can only be treated by one synchronization processor, on this protocol.

Table 2 Feature Fulfilment of MPCP

Static Priority Scheduling	✓
Dynamic Priority Scheduling	X
Supports Global Resources	X
Preemptive Waiting Mechanism	X
Nested Resources Allowance	X
Helping Mechanism	X

As we can see on the table above, MPCP does not satisfy most of the considered features.

2.2.2 DPCP

The Distributed Priority Ceiling Protocol [16] is based on the MCPC protocol. Created by the same author, DPCP addresses some issues related to the remote access mechanisms.

DPCP is a suspension based resource sharing protocol for fixed-priority systems. This protocol is considered suspension based, meaning that the task is suspended from execution while the resource remains locked.

Under DPCP, a task executes its own critical sections on its local processor, and it can access critical sections on global shared resources belonging to remote synchronization processors. Global critical sections are only preemptive to other global resources that have a higher priority.

DPCP allows nested resources, with the condition of the number of processors.

Table 3 Feature Fulfilment of DPCP

Static Priority Scheduling	✓
Dynamic Priority Scheduling	X
Supports Global Resources	✓
Preemptive Waiting Mechanism	X
Nested Resources Allowance	✓
Helping Mechanism	X

DPCP, being based on MPCP, focus on some important details of multiprocessor synchronization. However, as it can be seen in the table, it does not address all the features.

2.2.3 MSRP

The Multiprocessor Stack Resource Policy [17] is the version of the SRP for multiprocessor systems using partitioned EDF scheduling. MSRP is a multiprocessor scheduling algorithm with dynamic priorities. This protocol supports global resources, so a resource can be accessed by tasks from different processors.

The SRP is a priority ceiling-based protocol so it needs a priority ceiling for each resource. In case of the MSRP, each resource has a priority ceiling for each processor, that is, a priority level higher than the priorities presented by the set of tasks that take advantage of the resource.

The access to a global resource is done using a FIFO queue and the tasks on the same processor may share the same run time stack, using non-preemptive busy wait, meaning that a task waiting to access a shared resource, must continue executing, preventing any other tasks to use its own processor.

The concept of run time stack is related to a structure used to store relevant information to the tasks executing on the system.

Table 4 Feature Fulfilment of MSRP

Static Priority Scheduling	X
Dynamic Priority Scheduling	✓
Supports Global Resources	✓

Preemptive Waiting Mechanism	X
Nested Resources Allowance	✓
Helping Mechanism	X

In terms of the features covered by MSRP, MSRP covers almost the same features as DPCP, the only difference is that this protocol is designed for dynamic priority systems.

2.2.4 FMLP

The Flexible Multiprocessor Locking Protocol [18] introduces short and long resources, with the concept of Resource Group. Each resource group may contain either long or short resources, without mixing them.

Short resource groups are managed with a non preemptive FIFO queue lock, while the long resources use a semaphore lock. These groups may contain nested resources, allowing the possession of several resources by a single task, and the non-nestable resources are grouped individually.

According to A. burns and A. Wellings [1], group locks reduce the parallelism as a side effect.

Table 5 Feature Fulfilment of FMLP

Static Priority Scheduling	✓
Dynamic Priority Scheduling	✓
Supports Global Resources	✓
Preemptive Waiting Mechanism	X
Nested Resources Allowance	✓
Helping Mechanism	X

From the point of view of the evaluated features, the FMLP proves to be a more complete protocol than the others described previously. However, some very relevant aspects are missing.

2.2.5 SPEPP

Spinning Processor Executes for Preempted Processor [19] , is a great example of a protocol that offers a helping mechanism. In this protocol, the helping mechanism causes a thread waiting to gain access to a shared resource to execute the critical section of the resource, currently held by another task, if the holder of the resource is not able to proceed with its job, performing the remaining operations.

In SPEPP the access to a resource is handled in FIFO order and only the resource operations are made non preemptively.

Table 6 Feature Fulfilment of SPEPP

Static Priority Scheduling	✓
Dynamic Priority Scheduling	✓
Supports Global Resources	✓
Preemptive Waiting Mechanism	X
Nested Resources Allowance	✓
Helping Mechanism	✓

SPEPP introduces the concept of helping mechanism, which is very important on the multiprocessor resource sharing protocol's context. Although it uses non-preemptive wait instead of a preemptive solution, so it does not match all the addressed quality indicators.

2.3 Real-Time Operating Systems

Nowadays, it is common to find RTOSes that already support multiprocessor systems. However, from the plethora of RTOSes out there in the market only RTEMS and Litmus will be presented, since these operating systems were initially used to support the implementation of the chosen protocol, MrsP. Currently there are a few other RTOS that already support multiprocessor systems, namely eCOS [20], Nucleus RTOS [21], VxWorks [22], TI-RTOS [23], QNX [24], PikeOS [25], Litmus [26] and RTEMS [27].

2.3.1 Litmus

Litmus consists of a real-time extension to the Linux kernel [26] [12] [2] and it has been continuously suffering changes due to contributions of a community of researchers focused on this operating system.

The main goals of this RTOS are to provide easy-to-use and easy-to-modify testbed to implement real-time scheduling policies and locking protocols. There are several areas related to RTOS, so it is important to provide abstract systems that allow customization from the developer side, according to his interests. Litmus also provides several functionalities and data structures compatible with the Linux Kernel [28].

This RTOS provides a generic interface, to enable a relatively simple implementation of locking protocols, such as MrsP. Beyond RTEMS, that is described later, Litmus was one of the RTOS used to support the implementation of MrsP. Firstly, a prototype was implemented, together with the RTEMS implementation [3], but later [29], the fully implementation was done, along with a functional analysis of the protocol, comparing it with other alternatives.

2.3.2 RTEMS

The real-time executive for multiprocessor systems [27] is a multiprocessor open source RTOS. RTEMS provides an environment for embedded systems composed of some of the most important features of a real-time operating system, supporting several processor architectures and open standard application programming interfaces (API) such as POSIX. RTEMS is one of the RTOS where MrsP was successfully implemented and thus the one selected in this project.

One of the main objectives of this operating system is to provide a high level of user customization. RTEMS makes no assumptions about the specific hardware of a system, allowing the support of various sorts of system architectures [30].

RTEMS also provides a framework that supports the operations of the executive on multiprocessor. This framework brings a set of capabilities that makes the SMP handling simple to use and to modify, in order to become suitable to many applications according to their requirements, which may be quite diverse depending on the application domain. Some of the most important features of the multiprocessor support are the Scheduler Helping Mechanism that allows task migration and the definition of a task’s affinity within the schedulers, and Clustered Scheduling, concepts described later in this report.

The implementation of MrsP on RTEMS showed some interesting results in terms of the protocol performance [3]. However, it should be emphasized that there is a high overhead associated with some operations performed by the protocol, namely, the treatment of nested resources. It is important to mention that these difficulties were not clearly addressed in the original formulation of the protocol [1] and therefore, the original proponents of the protocol have already proposed improvements to the original version, specifying new features, along with a recent schedulability analysis [31]. Thus, this is a great motive to analyse its implementation in RTEMS in order to find possible bugs in the current implementation so that such issues, if any, may be corrected.

3 RTEMS Resource Sharing Protocols

This chapter describes in detail two protocols designed for multiprocessor real-time systems implemented in RTEMS, namely MrsP and OMIP. Even though the target of this work is MrsP, OMIP protocol is also covered as it can be used to perform comparisons in cases where it makes sense. Consequently, some of the most relevant features of both protocols, its properties and behaviour are described in order to understand how they perform the management of shared resources from a theoretical perspective.

3.1 MrsP

The Multiprocessor Resource Sharing Protocol, MrsP (with the middle letters in lowercase, so that this is not confused with MSRP), is a priority ceiling based protocol proposed with the goal of creating an abstract protocol that can be adapted to "partitioned, semi-partitioned and globally scheduled systems using fixed-priorities, EDF or any other designation" [1].

Partitioned scheduled system are systems where each processor is managed by one scheduler instance, while on semi-partitioned, one scheduler may manage more than one processor and, finally, using global scheduling, the system only presents one scheduler responsible for the management of all the processors.

MrsP addresses a great set of issues brought by the generalization of the resource sharing protocols concepts from single processor to multiprocessor.

3.1.1 Definition of MrsP

According to the initial proposal of MrsP [1], there are rules that the protocol must obey in order to satisfy its intended behaviour.

In the first place, a MrsP resource must have a set of different priority ceilings, one per scheduler instance that uses the resource. When a task tries to obtain a MrsP resource, its own priority must be immediately raised to the ceiling priority on the scheduler instance where it is executing. This rule is a rule inherited by MrsP from SRP and PCP, since priority raises also happen on these single processor locking protocols.

The access to a MrsP resource must be performed in FIFO way, and, while a task is waiting to access the resource, it must remain executing until it is able to execute the resource, performing busy waiting. One of the most important features of this waiting mechanism is that it is preemptive, which means that, while a task is waiting to access a resource, if a higher priority task enters the system on the waiting task home scheduler, it must be able to execute.

Finally, when a task is waiting to hold a resource it must be able to help the task that is currently executing it. Basically, this mechanism acts when the holder of the MrsP resource is not able to execute, due to a preemption. In this case, one of the tasks waiting to access a resource, because of the task needing help, may offer this task the possibility of migrating to their home processor, being able to continue executing the resource. After the helping procedure is performed, all the requests to access the resource must obey the FIFO order.

Beyond this set of properties, since the proposal of MrsP does not clarify and address nested resources very deeply, later, the authors of this protocol, elaborated a second proposal [4] addressing some missing aspects that existed on the protocol, accompanied with some other relevant rules that must also be loomed in this work.

In the first place, MrsP must present a deadlock detention mechanism. Nested resources under this protocol must present a specific order of obtention or else a deadlock may happen. For example, consider a system with two tasks, A and B, and two resources, R and R1. Assume that task A obtains the resource R, and B obtains R1. Then, if A tries to obtain R1, it will wait for B to release it, and if, at the same time, B tries to acquire R, a deadlock will happen. This deadlock detention mechanism exists to predict these situations, not allowing a task to perform operations related to a shared resource that may cause a deadlock to the system.

Since each MrsP resource may have a different priority ceiling for each scheduler of the system, when the helping mechanism is needed, the helping task must execute with the helper’s priority. This priority is the ceiling priority on the remote processor and the helped task priority cannot be changed while it is being helped, or else the progress on the remote processor may be affected by that priority change.

Also about the helping mechanism, although focusing on nested resources, the helping mechanism must be able to function, if the helping task is trying to access anyone of the holder resources and it must be transitive, which means that, even if the helped task is not owner of one of the resources that the helper is trying to allocate, although it is preventing it from continue executing, the help must be also possible. Beyond that, the helping mechanism, under MrsP, must operate successfully when the helper task and the helped task belong to the same scheduler, due to the transitive behaviour of this mechanism.

Concluding, while being helped, a task must be able to be re-dispatched on its own host processor. Lower priority tasks must not execute on the helped task's host processor while it is being executed on a remote one, under the helping mechanism

3.1.2 MrsP's Behaviour

According to the features described above, each scheduler instance behaves like SRP or PCP. The issues imposed by multiprocessing, such as concurrency and parallelism, require a FIFO queue containing the elected threads of each scheduler instance.

The following scheme (retrieved from the RTEMS SMP final report [32]) represents an example of the MrsP's operation on a partitioned scheduling system using dynamic priorities, where each scheduling instance is composed of one processor, and the accesses to the FIFO queue is dealt using SRP as the locking protocol for each partition.

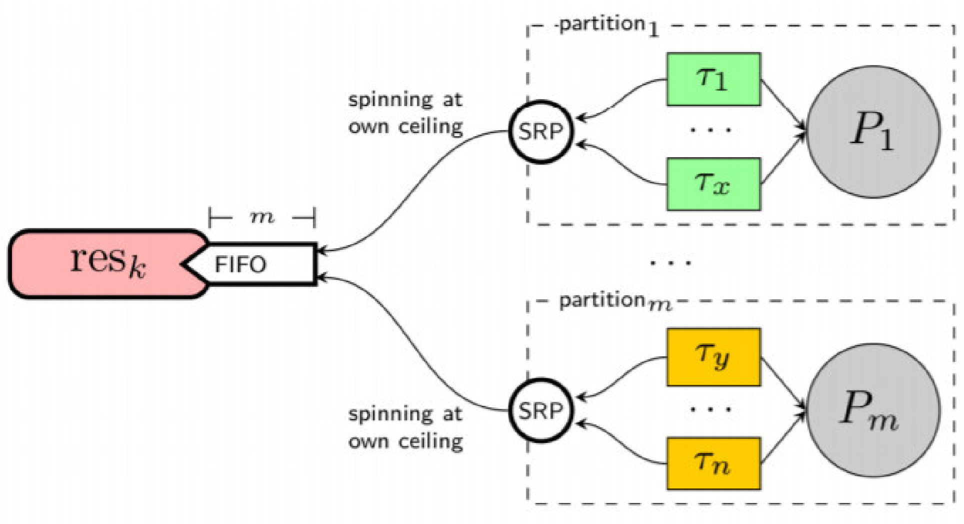


Figure 6 MrsP Schema

Data Source: http://microelectronics.esa.int/gr740/RTEMS-SMP-FinalReport-SpaceBelEmbeddedBrainsUnivPadova.pdf

The environment supporting this diagram is composed by m partitions or scheduler instances, each one of them responsible for managing a processor, represented as *Px*. Each processor executes a set of tasks, represented by *Tx*, attempting to execute their jobs.

Using MrsP, the access to shared resources, represented as *res* in the figure, is managed by a global FIFO queue with a length of *m*, matching to the number of processors of the system. Every time the resource is released, the head of the FIFO queue must be elected to claim the ownership of the resource.

Finally, the access to the global FIFO queue must be managed locally on each partition, according to the SRP rules, in this case. Which means that the higher priority task attempting to access the MrsP resource on each scheduler instance is the one that may access the global queue. Once a task is placed on the queue, it must wait, with its priority raised to the ceiling priority of the scheduler instance where it is running, until it becomes the head of the FIFO queue.

In conclusion, MrsP seems to be the most complete and abstract designed protocol presented in this report, meeting all the addressed features on the state of the art.

Table 7 Feature Fulfilment of MrsP

Static Priority Scheduling	✓
Dynamic Priority Scheduling	✓
Supports Global Resources	✓
Preemptive Waiting Mechanism	✓
Nested Resources Allowance	✓
Helping Mechanism	✓

3.1.3 Task's Logical States Under MrsP

According to the aspects addressed before, it is possible to affirm that, when using MrsP as the resource sharing protocol, a set of different logical states may be considered to represent the different activities performed by the system's tasks [4], according to the proposal of the protocol.

Since these logical states ease the understanding of the behaviour of a system operating under MrsP, in this section, they are briefly described:

- *Executing*: Represents any task of the system executing without needing the access to a resource.
- *Help not needed*: This state represents a task holding a MrsP resource, executing normally on its host processor, without needing any help.
- *Requiring Help*: A task holding a resource that is currently unable to continue executing, as for instance due to preemption.

- *Being Helped*: A task that holds a resource executes on a remote processor which contains a task is waiting to access it.
- *Potential Helper*: A task currently trying to access a resource, held by another task, performing a busy waiting, offering a potential help to the holding task, in case it is needed.
- *Helping*: A task currently waiting to access a resource gives the opportunity of execution on its host processor to another task which holds the same resource, in case the latter task is not able to execute on its host processor.

According to the protocol's proposal and the states explained here, the transitions between them is governed by the following diagram, retrieved from the second paper related to the MrsP's proposal [4].

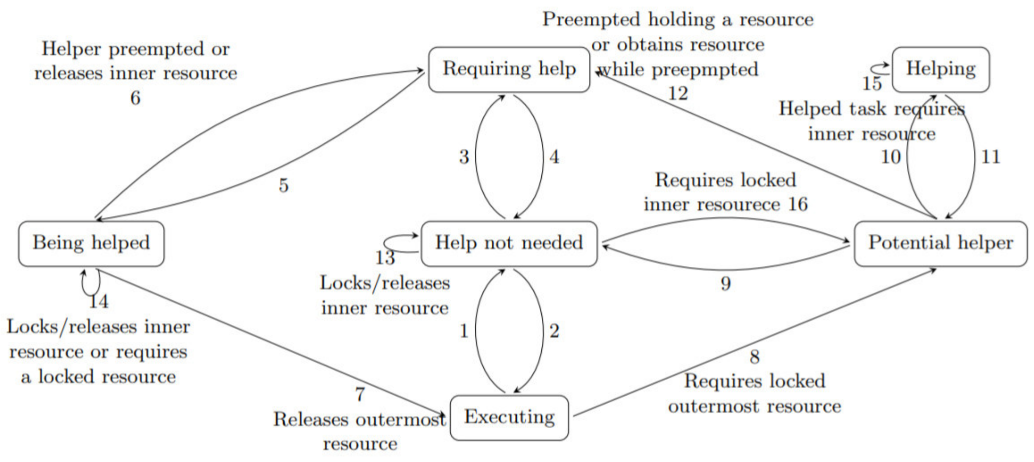

Preempted holding a resource or obtains resource while preepmpted 12
Helper preempted or releases inner resource 6
Requiring help
Helping
15
Helped task requires inner resource 10
11
3
4
5
Requires locked inner resourece 16
Being helped
13
Help not needed
Potential helper
9
14
Locks/releases inner resource
Locks/releases inner resource or requires a locked resource
1
2
8
Requires locked outermost resource
7
Releases outermost resource
Executing

Figure 7 MrsP State Diagram

The first transition, transition 1, represents a task obtaining a resource in a successful way, allocating it and executing it without needing any help and transition, while transition 2 represents the resource release, when the task stops using it, returning to the normal *Executing* state. A task executing a resource may obtain or release nested resources, unless those resources are already held by another task, this task's state must not be altered, as shown in the 13th transition.

While holding and executing a resource, the task may be preempted, transiting to the *requiring help* state, from the *help not needed* state, transition 3. If, meanwhile, the holding task is re-dispatched, the transition 4 may occur, allowing the task to execute again on its host processor.

Transition 5 is related to the helping mechanism. When a remote task is waiting to access a resource held by the task that is currently *Requiring help* accepts the helping request, the holding task transits to the *being helped* state, being allowed to execute the resource on the helper's processor. Transition 6 may occur due to the impossibility of continuing the help, for example due to the helpers' preemption. In this case, the *being helped* task must return to the *Requiring help* state until transition 5 or 4 is possible. At any time, a task that is *being helped* may obtain or release nested resources, transition 14. This operation will not change the task's current state.

When a task that is currently *being helped* finishes the execution of its resources, the helping procedure must stop and the helped task must return to its host processor, to the *executing* state, through the 7th transition, without being related to the resource anymore, unless it tries to lock it again.

A *potential helper* is a task requiring a resource and, in this way, it must wait to obtain it, offering help to the task that is currently executing that resource. This state may be a result of a task trying to obtain one resource that was not related to any resource, by the 8th transition, or it may also happen when a task already holds one or more resources and tries to lock an already held resource in a nested way, transition 16. When the potentially helped task obtains a resource and is allowed to execute on its host processor, the 9th transition may occur and this task must be able to execute the resource. If its processor is not available, this task must become to the *requiring help* state, asking other *potential helpers* related to its resources for help, transition 12.

Finally, while the holding task becomes helped by one of the *potential helpers*, the helper must transit to the *helping* state, transition 10, until it stops helping the holder of the resource, transition 11, becoming again a *potential helper*. While helping another task, if that task obtains one or more nested resources, the helper's state must not be changed as described by the 15th transition.

3.2 OMIP

The O(m) Independence Preserving (OMIP) protocol [2], where *m* denotes the number of processors, is designed for clustered fixed priority scheduling systems. It proposes an optimal resource sharing protocol based on the uniprocessor priority inheritance protocol, avoiding non preemptive segments, with the use of priority increasing.

Independence preserving means that, using this protocol, when a thread locks the critical section of a resource, it will not suffer any delay due to unrelated critical sections accesses.

This feature of OMIP is granted by the helping mechanism consisting on task migrations, which allows a task to execute the resource on a remote processor, as with MrsP.

Some of the main features of OMIP is that this protocol does not require any externally configured data, unlike PCP based protocols that require the setting of a ceiling priority.

3.2.1 Definition of OMIP

In the case of OMIP, as in MrsP, the author of the protocol has defined a set of specific rules that ensure that the protocol's behaviour meets all the requirements imposed on it. These rules are very relevant on the understanding of the protocol on a more practical way and allows the expectation of the results obtained when it is used, also allowing conclusions drawing about the relevance and possibility of comparing it with MrsP.

In first place, OMIP is composed of three queue structures, a global FIFO queue, managing the access to the resource from all the scheduler instances of the system, a local FIFO queue on each scheduler that is responsible for the access to the global queue and, finally, another local queue, managing the tasks access to the local FIFO queue, based on the priority inheritance protocol, managing tasks by their priority level.

In this way, one of the most important rules of OMIP is related to the relation between all the queues and the tasks managing to access the resource. When a task tries to access any OMIP resource, and the local FIFO queue is empty, it must be immediately placed on the global FIFO queue. On the other hand, when this queue is not empty, it means that a task from that scheduler is already on the global queue, so, the task must stay on the local FIFO queue.

Since the local FIFO queue has a limit of tasks, one per processor of the scheduler instance, or cluster, if this queue is already full, when a task tries to access the resource, this task, must be placed on the local priority queue.

Under the OMIP protocol, the next task to access a resource is always the head of the global FIFO queue. If the resource is already held by another one, the waiting tasks must be suspended. Although MrsP uses busy waiting instead of suspension, it is possible to understand that the behaviour is, again, very similar between the protocols.

Therefore, based on the priority inheritance protocol, one of the OMIP features is the existence of migratory priority inheritance, very similar to the helping mechanism of MrsP. Summarizing, when a certain task owns a resource and is able to execute on its own cluster, if there is one task waiting to access the resource and it would be able to execute, the "needing help" task must migrate to the other task's scheduler, inheriting its priority, and being able to proceed with its execution.

Finally, during the release of a resource, the holding task must be dequeued from the global FIFO queue and the local FIFO queue and, if those queues are currently composed by other tasks, the head of the global queue must be authorized to access the resource and the head of the local queue must be placed on the global one. If any task is placed on the local priority queue, its head also must be placed on the local FIFO queue.

3.2.2 OMIP's Behaviour

Currently, OMIP is the only alternative to MrsP in RTEMS. This protocol is based on priority inheritance, while MrsP is based on priority ceiling and there are certain details that distinguish both. Nevertheless, in a certain way, the operations and behaviour presented from both protocols is similar on several features, like, for example, the existence of a helping mechanism composed by task migrations and the function of the queues, even though both protocols present different internal structures. In fact,, OMIP is fit to be compared with MrsP in terms of functionality and behaviour.

The following image presents a summary of the mode of operation of OMIP.

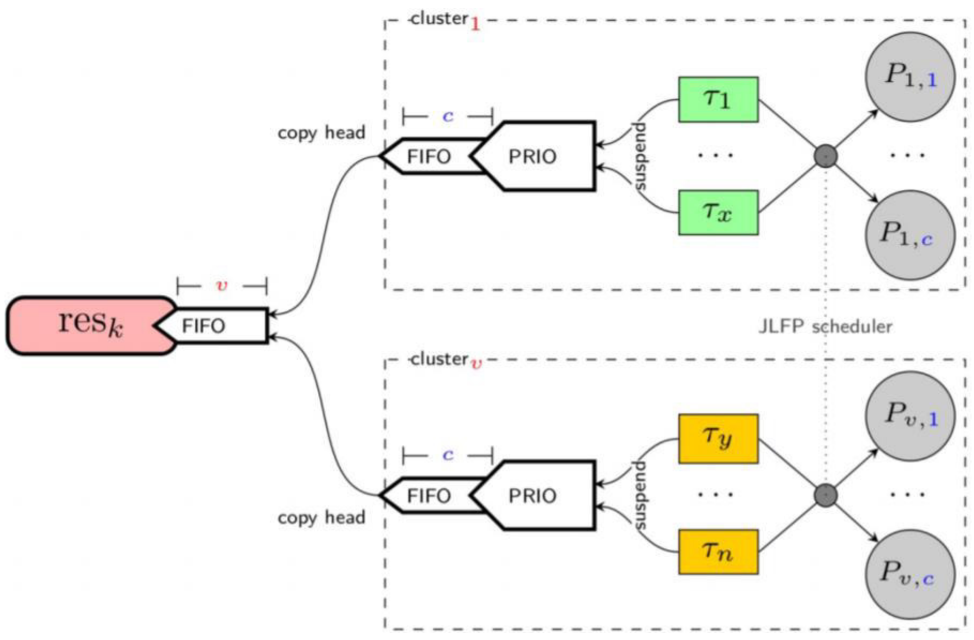


Figure 8 OMIP schema

Data Source: http://microelectronics.esa.int/gr740/RTEMS-SMP-FinalReport-SpaceBelEmbeddedBrainsUnivPadova.pdf

This diagram presents the management of a resource, on a fixed-priority scheduling system, composed by *v* clusters or scheduler instances, each one responsible for the management of *c* processors that are responsible for executing tasks, represented as *Tx*.

According to the properties of OMIP, each scheduler instance of the system manages the access to the resource with a hybrid queue structure. This structure is composed by a FIFO queue, with a length of *c*, and a priority queue, operating according to the priority inheritance protocol, with a dynamic length.

Each task in the system that attempts to access an OMIP shared resource, in case the resource is already held by another task, must be suspended and placed on the local FIFO queue of its host scheduler, unless this queue is already full, in this case, the waiting task must be placed on the scheduler's local priority queue.

Finally, the management of the access to the resource between each cluster of the system is dealt with a global FIFO queue, with a length of *v*, being composed of at most one task per scheduler instance. The tasks that comprise this structure are always the tasks elected by each cluster, this is, each task of OMIP's global queue is the head of each local FIFO queue of the system, and the holder of the resource is the head of this global FIFO queue.

In conclusion, although this protocol presents some relevant differences features, in relation to MrsP, it is possible to assert that there are a lot of aspects in which the behaviour of both protocols is similar, like the helping mechanism, as mentioned before. In this way, not just because both are implemented on RTEMS, it is quite possible to compare them in a safe way.

The table presented next is related to the desirable features meeting of OMIP, in order to compare it with the other resource sharing protocols approached on this chapter and on the state of art, chapter 2.

Table 8 Feature Fulfilment of OMIP

Static Priority Scheduling	✓
Dynamic Priority Scheduling	X
Supports Global Resources	✓
Preemptive Waiting Mechanism	X
Nested Resources Allowance	✓
Helping Mechanism	✓

According to the table, OMIP offers almost all the features under evaluation, meeting the ones that are less common on the described protocols. However, it is only designed for fixed-priority systems and it presents a suspension-based mechanism.

In this way, it can be concluded that OMIP does not meet all the features and is not so abstract as MrsP, in terms of its internal architecture, since it is designed to operate only on fixed-priority systems, while MrsP makes no assumptions about the system specifications.

4 RTEMS

Even though RTEMS was already mentioned and briefly approached previously in chapter 2. In this chapter, several concepts about it are detailed, in order to allow the reader to better understand the project described in this report, taking into account the scope of the work done and all the study performed on this RTOS.

4.1 Overview

RTEMS [27] the Real-time Executive for Multiprocessor Systems is an open source Real-time Operating System, that provides some of the most desirable features for applications implemented in the C programming language related to embedded systems, such as multitasking, multiprocessing, communication and synchronization between software components, dynamic memory allocation and a great level of user level customization. Furthermore, it supports several processor architectures and open standard application programming interfaces like POSIX and Ada.

The following image presents the application architecture of RTEMS. This RTOS provides a connection among the principal layers of real-time systems, presenting, in a certain way, a safeguard role between the system's hardware and the software application, dealing with most of the critical dependencies that may exist between them.

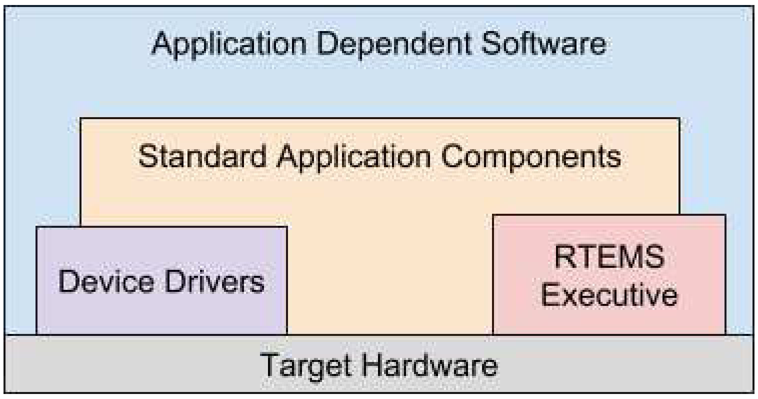

Application Dependent Software
Standard Application Components
Device Drivers
RTEMS Executive
Target Hardware

Figure 9 RTEMS Application Architecture

Data Source: https://docs.rtems.org/branches/master/c-user/overview.html

This operating system provides tools in order to allow the incorporation hardware dependencies on the system, giving the possibility to the developer to create abstract and modular software components that may be reused on several systems.

RTEMS internal architecture is composed by a set of different components providing several services to the system and working in balance to provide numerous desirable properties to the application system. These components presented on RTEMS are called resource managers and each one of them is responsible for providing certain features to the system through functions provided by RTEMS Core. The merging of those components gives rise to the possibility of real-time application development.

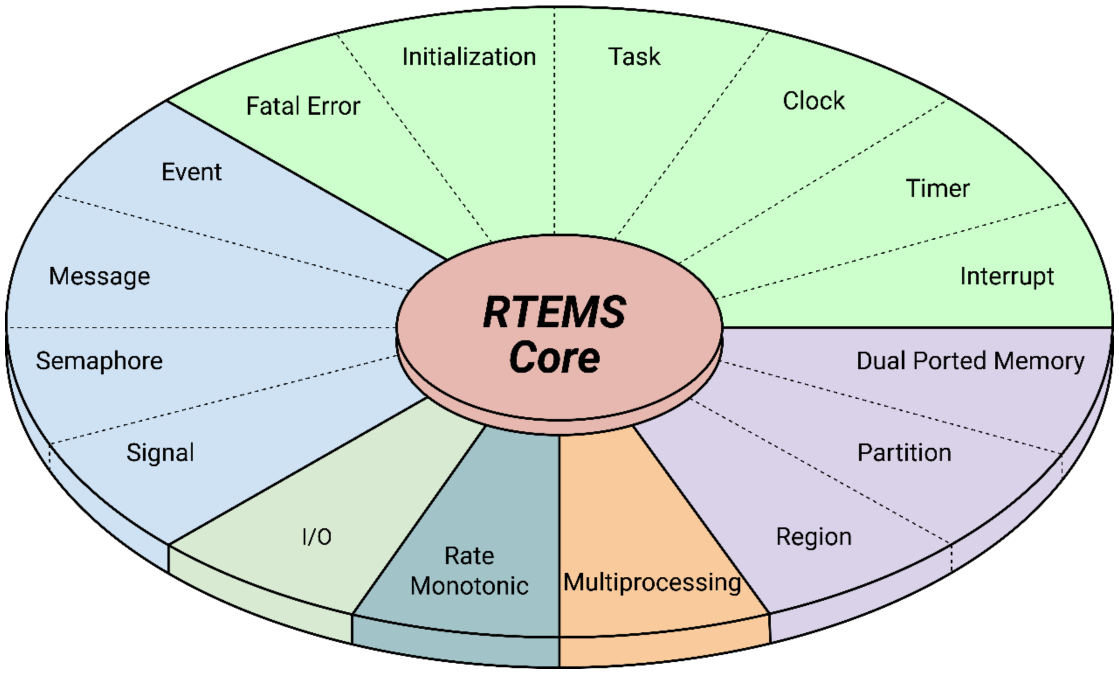


Figure 10 RTEMS Internal Architecture

Data Source: https://docs.rtems.org/branches/master/c-user/overview.html

In this report some of these managers are approached further, due to their relevance to the work described here, namely, the synchronization mechanisms, Semaphores, Signals, Messages and Events.

In this way, this RTOS is designed in order to allow the software developer to customize and extend the system features. RTEMS does not assume the characteristics of the specific hardware of the application. It only undertakes the existence of supported hardware allowing the provision for as many settings as possible and also allowing software portability, which stands for the independency between the real-time application and the machine used to execute it, being possible the change of hardware without jeopardizing the proper performance of the system.

4.2 Scheduling on RTEMS

Generally, the concept of scheduling [34] is related to resource allocation and time to execute a set of processes meeting the system's requirements. Scheduling is a very relevant component of real-time systems, since this kind of systems is constantly dealing with external actions and since all its components are usually dealing with strict time constraints. Without an appropriate scheduling method, the system will not be able to accomplish its minimum necessities.

The RTEMS scheduler manager is responsible to grant a safe and appropriated scheduling mechanisms according to the system's requirements, allowing the use of a set of different scheduling algorithms (such as EDF, RM, etc.), presented further on this chapter, each one with different properties in order to meet the system requirements in an efficient way, being also possible the implementation of a new algorithm by the user.

4.2.1 Task Manager

The concept of task on real-time systems is associated with the its atomic unit. In this way, this manager is particularly relevant for this work. Any application developed in RTEMS is composed by a set of tasks that must perform certain activities in order to achieve the expected operations and results in which the application is responsible for.

In order to able the preservation of the task's context on this operating system, all data related to an existent task of the system, is stored on a structure called Task Control Block (TCB). Each task configured on the system has its own TCB, which is associated with the task during the task's creation and becomes free when the task is deleted.

On RTEMS, a task may be in one of five different states, executing, ready, blocked, dormant, or non-existent. It is possible that the task's state transition occurs several times during the system execution, and these same transitions are ruled by the following diagram.

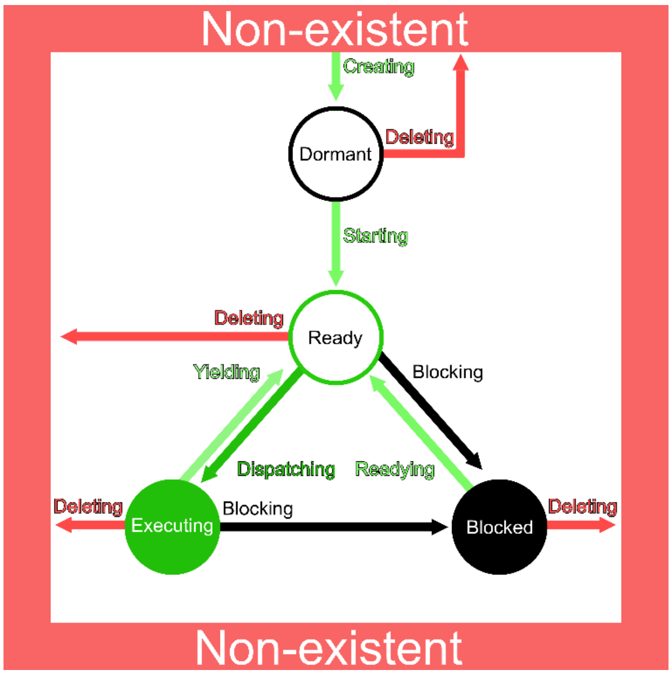


Figure 11 Task State Transitions

Data Source: https://docs.rtems.org/branches/master/c-user/scheduling_concepts.html

A task is considered non-existent if it is not created yet or after being deleted. Tasks may become non-existent from any other state, since they may be deleted in any one of the different states provided on RTEMS.

When a task is created, it becomes dormant, since it already exists, although permission to start running has not yet been given. A task is only able to start executing when it is currently on the ready state. Tasks become ready to execute after being created, when they yield their processor while executing, or after they become blocked.

Finally, a task may also be executing after being dispatched on its host processor, from the ready state, due to the starting of a process or after a preemption or a blocking operation (blocking may occur, for example due to the waiting procedure to access a shared resource, or when it is waiting for I/O).

Beyond the state of a task, RTEMS considers the task's execution mode. This mode may be related to preemption, asynchronous signal routines (ASR) processing, timeslicing and the task's interrupt level.

The set of preemption allowance determines if when the task starts its execution, the processor has the ability to switch that task to another, due, for example, to the task's priority level.

The asynchronous signal processing mode, concept that will be approached further, on the communication mechanisms, defines if the task will or not process the signals that it receives. Timeslicing is referred as the maximum time a task may execute on a processor, if other tasks with the same priority are ready to execute, if timeslicing is disabled on the executing task, a task with the same priority as it, the switch between them must not occur. Finally, the interrupt level of a task defines which interrupts may happen during the task's execution. This level defines the priority of the system's interruptions, generally considered as external stimuli to the system.

Concluding this description, it is important to mention the RTEMS task's priority. The priority of a task clarifies its importance in relation to the processor in which it executes. On RTEMS, 255 levels of priority are supported, from level 1 to level 255. Task with smaller priority levels are the tasks that present the greater priority. The priority of a task must be defined during its creation, although priority changes are possible during the task's lifetime. This level of importance is used by the scheduler to determine which task in the ready state must execute on the processor first.

4.2.2 Symmetric Multiprocessing on RTEMS

As mentioned earlier on this report, one of the most important features of RTEMS is the possibility of multiprocessing support. This operating system supports Symmetric Multiprocessing (SMP) on several processors belonging to the architecture ARMv7-A [35], PowerPC [36], RISC-V [37], or SPARC [38].

The support of SMP on RTEMS is based on Clustered Scheduling using schedulers and synchronization protocols variations, such as MrsP.

In this chapter, most of the relevant SMP features are addressed and clarified.

4.2.2.1 Clustered Scheduling

The scheduler manages the allocation of all the tasks trying to execute on the system's processors. Depending on the system, certain requirements must be accomplished by the scheduler must operate in order to grant the execution of all the system's processes and meet all the desired properties of it. Since real-time systems are characterized by the obligation of meeting several time constraints.

RTEMS presents clustered scheduling in order to deal with SMP systems. The system processors are part into a set of processors groups, called clusters. Each cluster is managed by a scheduler instance, which operates according to a set rules, depending on the chosen scheduling algorithm (a topic that is addressed further in this chapter). When the number of processors of a cluster is one, the cluster is called a partition. In a system where there is only one cluster responsible of managing all the processors with one scheduler instance, it is called global scheduling, since all the tasks belong to the same scheduling environment. This approach to SMP scheduling provides a safe way to isolate different components of the system according to their features and constraints, providing a greater freedom and customization level for the user, according to his interests.

4.2.2.2 Task Affinity

Task Affinity is one of the new RTEMS features on multiprocessing that allows the specification of a set of processors in which a task must execute. On RTEMS only certain scheduling algorithms support task affinity, allowing the developer to establish the sets of processors on which system tasks may perform their operations, inducing the desired behaviour to it.

4.2.2.3 Task Migration

RTEMS provides the possibility, in multiprocessor systems, for a task to migrate between different processors.

This feature may only be possible when the task's scheduler or processor affinity is changed (using the *rtems_task_set_scheduler*, or *rtems_task_set_affinity*). When the task enters on a blocked procedure and is resumed, if the processor where it was executing is already taken by another task, having to migrate to a free processor. Finally, a task migration between scheduler instances can occur induced by the Scheduler Helping Protocol, concept that is detailed next.

4.2.2.3.1 Scheduler Helping Protocol

The scheduler provides a helping protocol, in order to make the helping mechanism possible on the resource sharing protocols on RTEMS, MrsP and OMIP.

On RTEMS SMP, each task from the system is assigned to its host scheduler instance, defined during the task's creation. Since on both SMP locking protocols mentioned here, a task may temporarily have access to other scheduler instances due to the helping protocol, in case of preemption of the task holding the resource, this feature becomes quite relevant for those protocols.

In order to support the helping mechanism, the RTEMS scheduler needs three indispensable operations, "ask a scheduler node for help", "reconsider the help request of a scheduler node" and “withdraw a scheduler node”, operation that currently are present on all the SMP schedulers of this operating system.

The “ask for help” operation happens when a task requires help, carried during the thread dispatch procedure, when the scheduler determines which task executes next. After that, when this request is recognized, it is registered on the other scheduler instances, unless the ones that, that are defined to have no affinity with the needing help task. Each one of these requests receives a response, indicating if the help is possible or not and the procedure stops if one request is accepted, or if no one is accepted and, in this case, the need for help is registered on the scheduler’s context.

The “reconsider the help request” and the “scheduler node withdrawal” operations are used in order to restore the scheduler context, in order to remove old helping requests and then to remove scheduler node no longer needed, once used to help the task, when the helping mechanism is no longer possible or needed on that case.

4.2.3 Symmetric Multiprocessor Schedulers

RTEMS default scheduling algorithms are all priority based, which means that the scheduler instance executes the highest priority task in that instance’s queue. This operating system supports multiprocessor and uniprocessor scheduling algorithms, although since this work focus on SMP synchronization mechanisms, only the SMP scheduling algorithms are mentioned here.

This section briefly explains the most relevant properties of the RTEMS SMP scheduling algorithms.

4.2.3.1 Earliest Deadline First SMP Scheduler

The Earliest Deadline First (EDF) [39] is a scheduling policy based on dynamic priorities at the job-level in which the priority criteria is based on the deadline property of each job, that is, the shorter the deadline of the job, the higher the priority.

On RTEMS, this scheduler consists on a fixed-priority scheduler using the task’s implicit deadline as priority level, the maximum execution time the task should take to finish.

Tasks without deadline are called background tasks, less prioritized than the tasks with an active deadline and if there are no tasks with active deadline on the system, this scheduler behaves exactly like a fixed-priority scheduler.

4.2.3.2 Deterministic Priority SMP Scheduler

The Deterministic Priority SMP Scheduler consists on a scheduler using fixed-priorities. The operation of this scheduler consists of a chain table, where each entry level corresponds to a priority level.

On RTEMS, the concept of chain is related to a doubly linked list of nodes attached to a main node responsible for the control of that structure. The control node is the first and the last member of the chain [40].

4.2.3.3 Simple Priority Scheduler

The scheduling approach, using this type of scheduler, is based on a single chain with the set of nodes sorted by priority for all the ready tasks.

4.2.3.4 Arbitrary Processor Affinity Priority SMP Scheduler

The Arbitrary Processor Affinity Priority SMP Scheduler is very similar to the Deterministic Priority SMP Scheduler, since it is also based on a table of Chains, where each chain corresponds to a priority level. The main difference between both schedulers is that the Arbitrary Processor Affinity Priority SMP Scheduler allows a task to define affinity to processor, letting them to execute only on certain processors, according to the user defined specifications.

4.3 Synchronization and Communication on RTEMS

RTEMS presents a set of five managers that may be considered responsible for synchronization mechanisms. Although the scope of the project is focused on SMP semaphores, namely SMP Resource Sharing Protocols, all the managers and their most relevant operations are briefly summarized in this section of the report.

4.3.1 Barrier Manager

A Barrier is a synchronization mechanism usually used during in the initialization of an application. This tool operates, in a certain way, like a gate, where every task that tries to pass the barrier becomes blocked until the barrier is released, and then all those blocked tasks may proceed executing normally.

On RTEMS, there are two types of barriers, automatic and manual barriers.

In case of the manual barriers, all the tasks that are waiting to the barrier release have to stay blocked until the controller task releases it, unleashing the waiting ones. With automatic barriers, all the tasks that are waiting for the barrier release, will have to stay blocked until the limit of tasks waiting is achieved.

4.3.2 Message Manager

On RTEMS, a message is considered as a vehicle for the transmission of a variable amount of information in order to support communication and synchronization between a set of tasks. This amount of data can be defined by the user.

A set of messages is transmitted among the intervenient using a Message Queue. This structure usually works as a FIFO queue, although some messages, the urgent ones, may be prioritized behaving on a Last In First Out (LIFO) way. The message queue may also be used in order to grant synchronization between tasks, since the receiver must wait for the message sent by the emitter task.

A message may be sent, in the same queue, to several tasks. RTEMS presents a mechanism that copies the sent message and, in that way, the message is repeated to several peers.

4.3.3 Event Manager

An Event consist on a highly efficient communication method between tasks. The events are used on RTEMS to allow a task to send an important message to another without making great efforts, in a computational light way. An event is basically composed by a set of flags and each one of them has a specific meaning. The set of flags available to send via events is composed by thirty-two elements and each event may send one or more flags at a time.

The events must have a specific destination and provide synchronization between tasks. The receiver is not able to proceed with its own execution while waiting for the reception of it. When a task is waiting to receive an event, it may be waiting for a specific set of flags or may be waiting for any random event to be received.

Finally, a task may be also waiting for at least one of the desired flags, or else, waiting for all the flags received from the event.

4.3.4 Signal Manager

On RTEMS, signals are similar mechanisms when compared to events. The great difference between them is that signals work in an asynchronous way. A signal is composed by a set of flags that are sent to a specific receiver, as in the case of events, although, signals operate asynchronously.

The asynchronous communication between tasks, using signals, is established using Asynchronous Signal Routines (ASR). Contrarily to events, with signals the ASR does not interfere on the state of the receiving task, which means that, signals may be used to establish communication without having to synchronize the sender with the receiver.

4.3.5 Semaphore Manager

The Semaphore Manager is responsible for providing synchronization between tasks and resource's mutual exclusion, presenting a diverse set of semaphore types that may be used according to the application requirements.

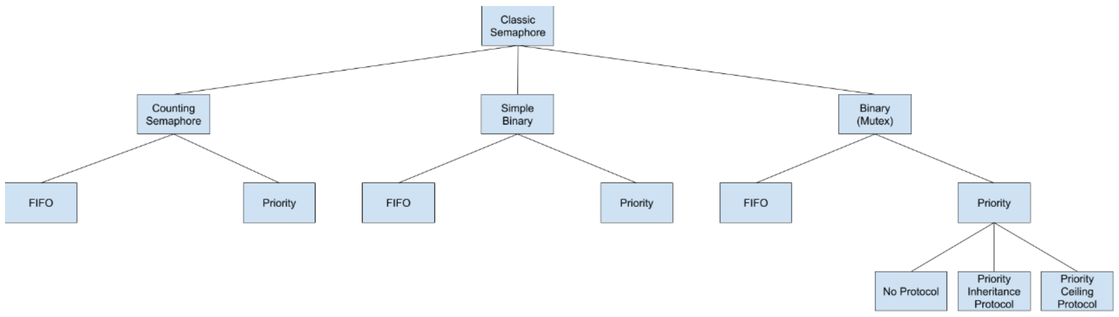


Figure 12 RTEMS Semaphores

Data Source: https://docs.rtems.org/branches/master/c-user/ semaphore_manager.html

This mechanism may be considered as a locking mechanism that protects a critical section, usually a shared resource, accessed by various tasks continuously in a concurrent way or, on the other hand, preventing an entity from making progress. Sometimes these sections can be severely corrupted if the system does not guarantee proper safety of them, so mutual exclusion is mandatory on those resources in order that they may not be manipulated by more than one task at a time. There are binary, simple binary and counting semaphores on RTEMS.

Simple binary semaphores, on RTEMS are only used to grant synchronization between tasks, without protecting a specific resource, for example, if it is necessary that one task performs a given operation before another task performs its own. Specifically, the simple binary semaphore may be used to block the second task and, when the first task finishes its activities it may unblock the other, letting it execute normally. That is, the task that tries to obtain the semaphore may not be the same one releasing. On the other hand, in order to protect shared resources, the task that access the semaphore must be the one releasing it, else mutual exclusion is not granted.

The other types of semaphores, the mutex, which is also binary, and the counting semaphores, may be used to protect a given resource. The main difference between those, is that the mutex is binary, only allowing one task at a time to access the critical section, while the counting semaphores may allow more than one task to pass through the semaphore at the same time, generally in cases where the critical section is allowed to have more than one task accessing it, for example, if the critical section is composed by ten resources, there is no problem that ten tasks access it, since they are not going to manipulating the same source, granting its safety.

Finally, a RTEMS semaphore may manage the access to a resource according to a FIFO order, which means that the first task to try to obtain the semaphore is also the one to have permission to have access to it, or, this access management may be dealt according to the priority of the tasks accessing the resource. The task with higher priority is the next to access the resource.

MrsP is considered a mutex, or binary semaphore, since only one task at a time may obtain it, granting mutual exclusion of resources. Thus, being considered a resource sharing protocol. Tasks access order to MrsP semaphores are dealt according to their priorities.

5 Implementation of MrsP on RTEMS

As explained before, MrsP is a multiprocessor resource sharing protocol that was considered successfully implemented on RTEMS, constituting one of the two possible resource sharing protocols for SMP environments for this operating system.

Since this project is focused on MrsP and not just on the evaluation of its implementation on RTEMS, its implementation was analysed in order to verify if the behaviour under this operating system is the expected one.

In this chapter, the focus is the implementation of the most relevant operations of a MrsP semaphore in RTEMS, namely, the creation of a semaphore, changing its priority ceiling, the resource release, resource obtainment and, finally, the semaphore deletion.

5.1 Create a MrsP Semaphore

As previously mentioned, MrsP is a priority ceiling-based resource sharing protocol, that manages the access to a global resource using a FIFO queue. In this way, the structure that comprises a MrsP semaphore must be composed of a global FIFO queue to manage the access to the resource and a set of priority ceiling values used by each scheduler instance of the system, these are responsible for managing the access to the FIFO queue locally.

In this way, the creation of a semaphore is responsible for the creation of the entire structure responsible for the management of a MrsP semaphore.

The semaphore creation directive, implemented on the semcreate.c file, is responsible to forward this operation to the specific implementation according to the semaphore fashion and passing the needed arguments to proceed it, as represented on the following sequence diagram[5].

[5] The sequence diagram is a widely used UML dynamic diagram, used often on software development to represent the interaction between certain software components, in order to describe complex operations.

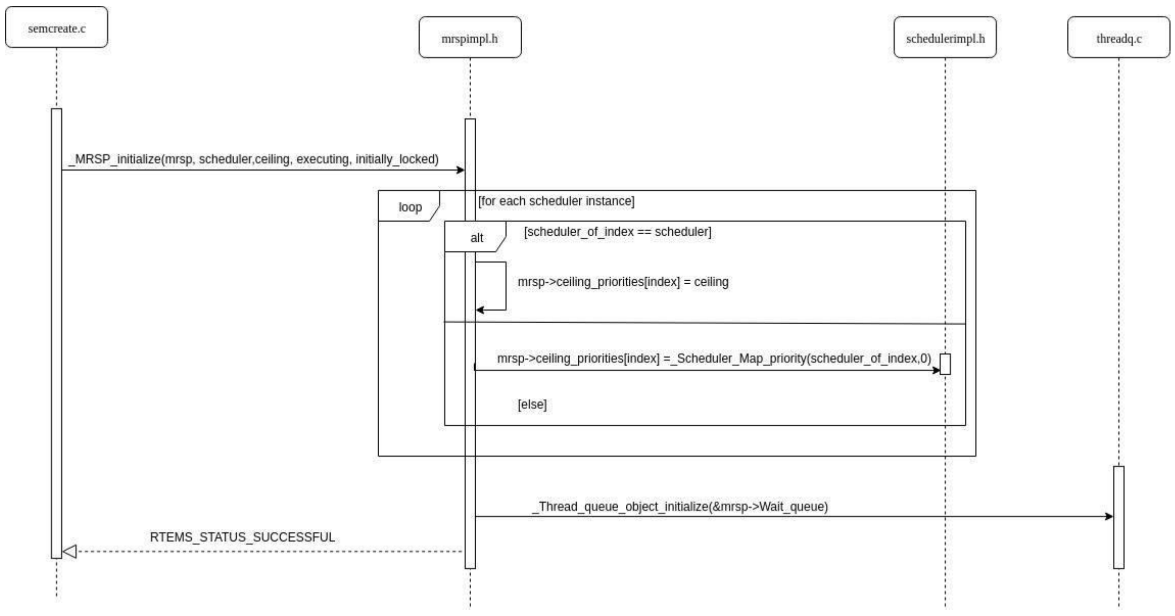

semcreate.c
mrspimpl.h
schedulerimpl.h
threadq.c
_MRSP_initialize(mrsp, scheduler,ceiling, executing, initially_locked)
loop
[for each scheduler instance]
alt
[scheduler_of_index == scheduler]
mrsp->ceiling_priorities[index] = ceiling
mrsp->ceiling_priorities[index] =_Scheduler_Map_priority(scheduler_of_index,0)
[else]
_Thread_queue_object_initialize(&mrsp->Wait_queue)
RTEMS_STATUS_SUCCESSFUL

Figure 13 Creation of a MrsP Semaphore on RTEMS

The arguments passed to the mrspimpl.h file, where the implementation of MrsP is presented, are the following:

- *mrsp*: the MrsP semaphore structure;
- *scheduler*: the scheduler instance of the task that is executing the current operation;
- *executing*: that represents the task performing the semaphore creation;
- *initialy_locked*: a user defined flag used to create a semaphore initially locked or initially unlocked, although this variable is not relevant here, since MrsP semaphores must always be initially unlocked.

In first place, the priority ceiling set is defined for each scheduler instance of the, on the MrsP structure. The variable *ceiling* defines the priority ceiling of the semaphore on the executing task's host scheduler and the priority defined for the other schedulers of the system is 0, the greatest priority of them, using the *Scheduler_Map_priority()* function to map this value the greatest priority of it, that on RTEMS may be user defined. The returned value is stored on the semaphore structure. The priority of a scheduler may be changed using the *rtems_semaphore_set_priority* a process that is described later in this chapter.

Finally, using the *_Thread_queue_Object_initialize()* function, a FIFO queue is created, initializing its control structure adding a reference to it on the MrsP control structure, i.e., *mrsp*. This way, a MrsP semaphore and all the structures that guarantee its correct behaviour are created and initialized.

5.2 Obtaining a MrsP Semaphore

When a task tries to obtain a resource there are two possible operations that may be performed:

1. If the resource is free, the task can claim its ownership, accessing the resource directly;
2. If there is another task already holding the semaphore and executing the resource, the task that attempts to obtain it must wait using a waiting mechanism, according to the semaphore's resource sharing protocol.

Under MrsP's semaphores, when a given task tries to obtain the semaphore, operation dealt on RTEMS with the function *_MRSP_Seize()*, if the semaphore already has an owner, meaning that it is already locked by another task, the attempting task must wait for the ownership of the semaphore, guaranteed by the *_MRSP_Wait_for_ownership()* function. Else, if that task can directly hold the resource, it must claim the semaphore's ownership, operation performed with the *_MRSP_Claim_ownership()* function, present on the MrsP's implementation on RTEMS, as described on the following flow chart[6].

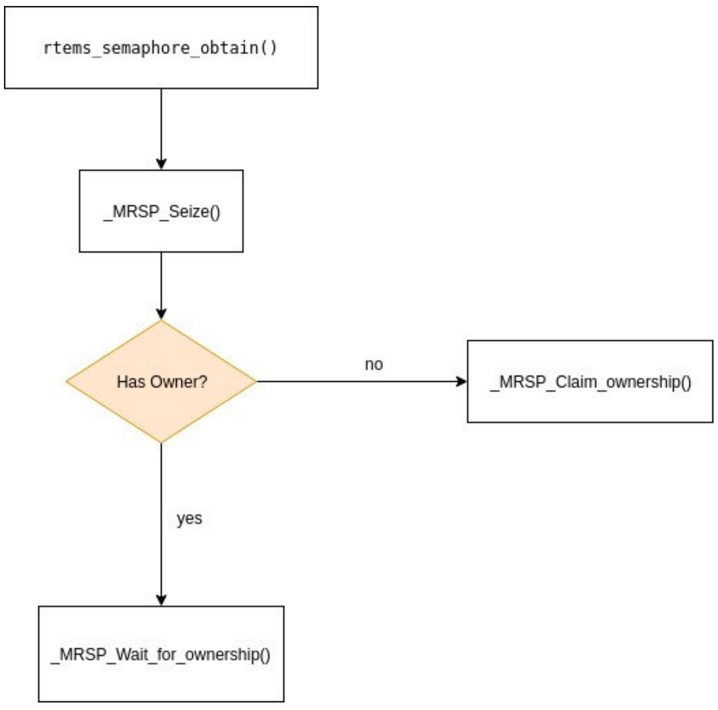


Figure 14 Obtention of a MrsP Semaphore on RTEMS

[6] Flow charts are also commonly used to represents the flow of an algorithm or the behaviour of a given software component, it was used in this chapter to simplify the representation of certain operations of the protocol.

In order to improve reader understanding, the waiting mechanism and the ownership claiming of a semaphore are explained next.

5.2.1 Claim the ownership of a Semaphore

In first place, the obtention operation is approached, according to the diagram present next.

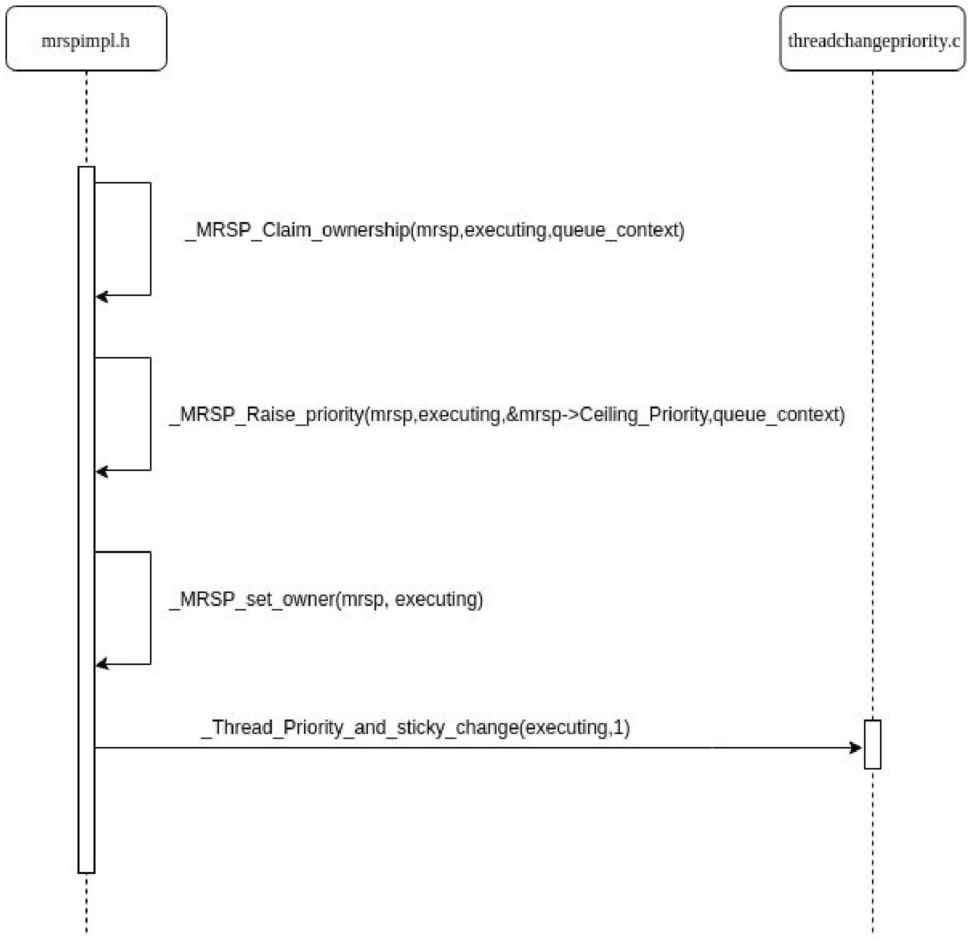

mrspimpl.h
threadchangepriority.c
_MRSP_Claim_ownership(mrsp,executing,queue_context)
_MRSP_Raise_priority(mrsp,executing,&mrsp->Ceiling_Priority,queue_context)
_MRSP_set_owner(mrsp, executing)
_Thread_Priority_and_sticky_change(executing,1)

Figure 15 Claim Ownership of a MrsP Semaphore

During the invocation of the *_MRSP_Seize()* function, when a task is trying to access the MrsP semaphore, as explained above, if the resource is not currently owned by another task of the system, the attempting task must claim the semaphore's ownership, calling the *_MRSP_Claim_ownership()* function.

The arguments passed are presented next:

- *mrsp:* the MrsP semaphore structure, with the waiting queue and the set of ceiling priorities;
- *executing:* the task trying to obtain the resource*;*
- *queue_context:* the context of the waiting queue, used to store relevant information and assist operations related to it.

During this procedure, two main operations are performed:

- The promotion of the task as an owner of the resource;
- Its priority raise to the priority ceiling of the resource.

With the *_MRSP_raise_priority()* function, the priority ceiling of the obtaining task is defined. Therefore, that task is promoted to owner of the resource, through the *_MRSP_set_owner()* function that allows the definition of the owner of the waiting queue's lock.

With the *_Thread_Priority_and_sticky_change()*, the new priority of the task, the ceiling priority, is updated and propagated to the system. This function also increments the sticky level of the task. The sticky level of a task, when incremented to at least the value of 1, grants that when the task is executing in any scheduler instance other than its original one, lower priority tasks may not execute on its host scheduler[7], with the purpose of securing this property. This feature is quite relevant, according to the properties of MrsP, mentioned on chapter 3, where it is defined that under semaphores using MrsP, lower priority tasks must be prevented from executing on the host processor of the holder of the resource.

5.2.2 MrsP's Waiting Mechanism

The implementation of the waiting mechanism of MrsP in RTEMS is rather extensive and complex. Thus, the analysis of this operation is placed in appendix 2.

5.3 Release of a MrsP Semaphore

On MrsP, this operation must remove the owner from the semaphore, restoring its priority and sticky level to the original values and, if the waiting queue has at least one task trying to access the resource, it must also select the head of that same queue as the new owner of the semaphore.

[7] Taking advantage from idle tasks, processes that are not performing any specific job for the system.

The function responsible for the release of a MrsP semaphore on RTEMS is called *_MRSP_surrender()*.

This operation is described on the following diagram.

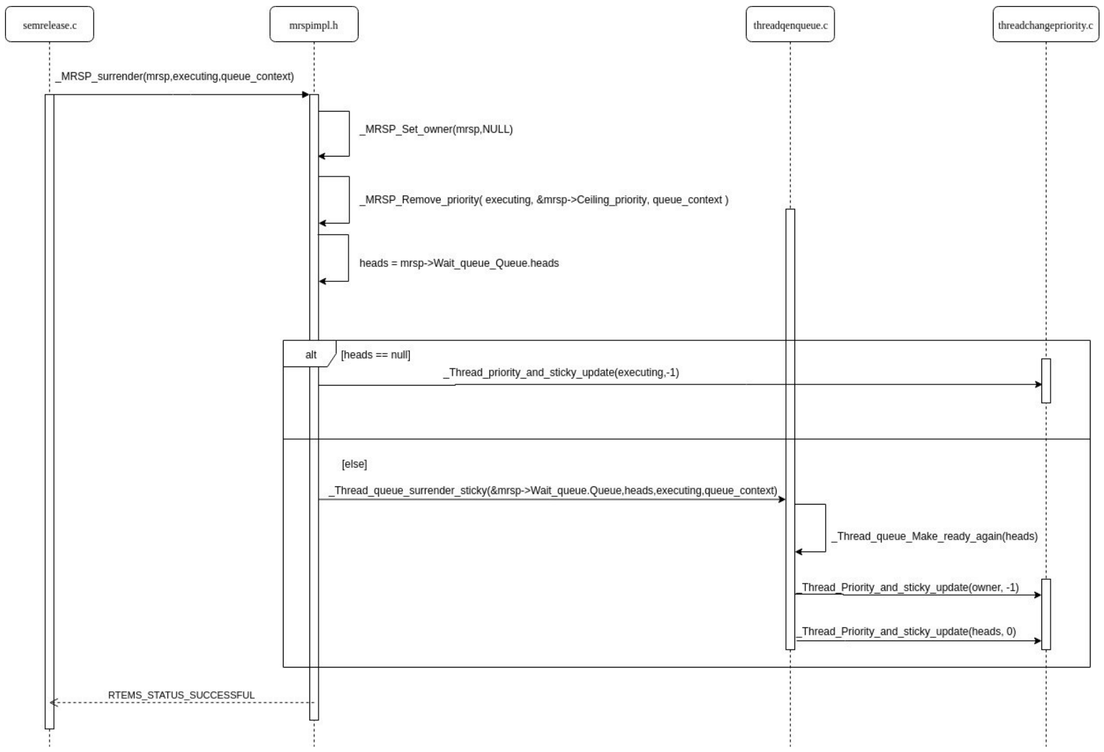

semrelease.c
mrspimpl.h
threadqenqueue.c
threadchangepriority.c
_MRSP_surrender(mrsp,executing,queue_context)
_MRSP_Set_owner(mrsp,NULL)
_MRSP_Remove_priority(executing, &mrsp->Ceiling_priority, queue_context)
heads = mrsp->Wait_queue_Queue.heads
alt
[heads == null]
_Thread_priority_and_sticky_update(executing,-1)
[else]
_Thread_queue_surrender_sticky(&mrsp->Wait_queue.Queue,heads,executing,queue_context)
_Thread_queue_Make_ready_again(heads)
_Thread_Priority_and_sticky_update(owner, -1)
_Thread_Priority_and_sticky_update(heads, 0)
RTEMS_STATUS_SUCCESSFUL

Figure 16 Release of a MrsP Semaphore on RTEMS

The arguments passed through the function are the following:

- mrsp: the main MrsP semaphore structure;
- executing: the task performing the operations;
- *queue_context*: the context of the waiting FIFO queue, responsible for the helping and registration of queue related operations.

In first place, with the *_MRSP_Set_owner()* function, the *executing* task, holder of the semaphore, is removed from the MrsP semaphore, defining the new owner of it as null, which means that the semaphore is not held by any entity. Moreover, the priority of the task that released the semaphore is restored to its original priority, through the *_MRSP_Remove_priority()* function, which uses the queue context (where all the relevant information about the queue operations is stored) in order to proceed with the priority change. In this way, the *executing* task no longer owns the resource.

Finally, in order to assert and possibly make the selection of the next owner of the semaphore, the head of the queue is selected, if any, denoted as *heads* in the diagram represented above.

If the head of the queue is different than null, it means that there are other tasks trying to access the resource. So, it is necessary to proceed with the promotion of the head of the queue to the owner of the MrsP semaphore. This operation is possible with the *_Thread_queue_surrender_sticky()* function, implemented on the RTEMS thread queue. In this way, the head of the queue is promoted to the owner, with the *_Thread_queue_Make_ready_again()* function, which also propagates the updated priority and sticky level of the new and previous owner of the resource to the system, with the function *_Thread_Priority_and_sticky_change()*, decrementing the sticky level in one unit, and without changing the sticky level of the new owner, since during the waiting mechanism it was already incremented, on the obtain operation.

On the other hand, if the queue is empty, the only operation that is required is the priority and sticky level propagation of the previous owner of the resource, again using the *_Thread_Priority_and_sticky_change()* function.

5.4 Set the Priority Ceiling of a MrsP Semaphore

Since one of the fundamental properties of MrsP is the possibility of defining several priority ceilings values for a MrsP semaphore, one per scheduler instance, the RTEMS operation to set the priority of a semaphore becomes very relevant.

This operation allows a task to define a priority ceiling level for a specific scheduler instance of the system, as described on the following diagram.

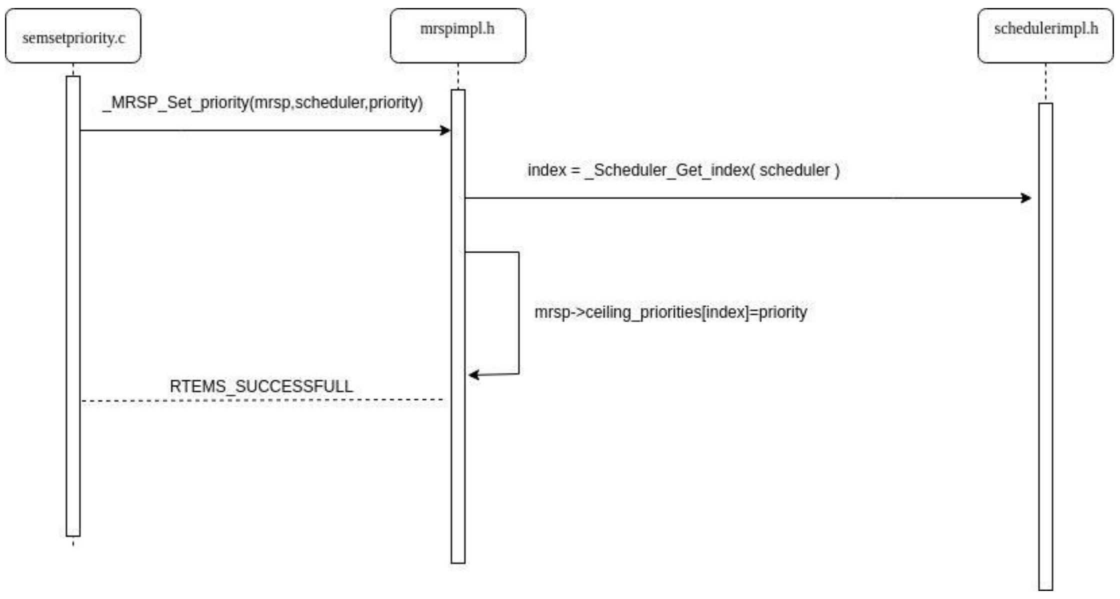

semsetpriority.c
mrspimpl.h
schedulerimpl.h
_MRSP_Set_priority(mrsp,scheduler,priority)
index = _Scheduler_Get_index(scheduler)
mrsp->ceiling_priorities[index]=priority
RTEMS_SUCCESSFULL

Figure 17 Priority Change on a MrsP Semaphore on RTEMS

On the implementation of MrsP, presented on the mrspimpl.h file, the function responsible for the definition of the priority ceiling of a semaphore is called using the *rtems_semaphore_set_priority*, implemented on semsetpriority.c, which is responsible of choosing the correct operation according the semaphore fashion, in this case MrsP.

In order to perform this operation, the following arguments are necessary:

- *mrsp*: the MrsP main structure;
- *scheduler*: the scheduler instance where the priority ceiling is being changed;
- *priority*: the new priority ceiling value.

When compared with other operations, setting the priority ceiling of a MrsP semaphore is quite simple. It consists on the change of the priority ceiling value on the *mrsp* structure on the index of the desired scheduler instance. The scheduler and the priority ceiling are user defined and, with the *_Scheduler_Get_index()* function, present on the implementation of the scheduler, the index of the scheduler that must suffer the priority ceiling change may be returned and used to set the priority ceilings of the MrsP semaphore control structure.

5.5 Delete a MrsP Semaphore

This operation of deleting a semaphore consists on the deletion of all the structures related to the semaphore, like the main structure of a MrsP semaphore, *mrsp*, and also the FIFO queue related to that same semaphore.

The first operation performed when a task tries to delete a MrsP semaphore consists on a validation that a semaphore is not currently held by any task of the system. In this way, using the function *_MRSP_can_destroy()*, the system consults the owner of the MrsP semaphore, searching on the FIFO waiting queue, with the function *_MRSP_get_owner()* and if the resource is currently being used, if the owner is different than null, the resource may not be destroyed, being returned the flag *RTEMS_RESOURCE_IN_USE,* otherwise, if the owner is null, then the semaphore may be deleted, returning the flag *RTEMS_STATUS_SUCCESSFUL*.

The process of deletion of a MrsP semaphore is performed using the *_MRSP_destroy()* function and the arguments of this function are the following:

- *mrsp*: the MrsP semaphore general structure;
- context: the global FIFO queue context.

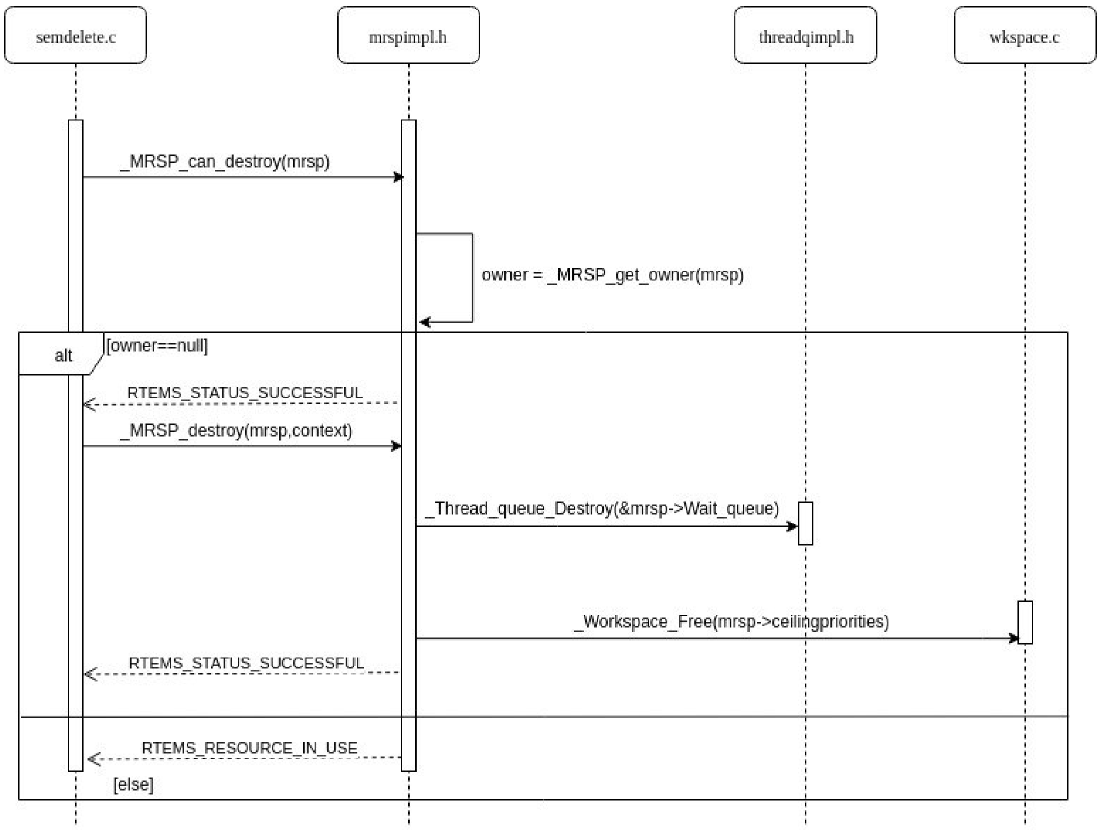


Figure 18 Deletion of a MrsP Semaphore on RTEMS

In this case, the queue context is only used to provide security and low level synchronization to the system, not being considered very relevant to this operation, since it does not interfere directly with the protocol's behaviour.

The FIFO queue is deleted using the function *_Thread_queue_Destroy()* and the argument passed is the queue used by the semaphore. Finally, the function *_Workspace_free()* is also used to free the allocated memory where the set of ceiling priorities used by the semaphore is stored. This way all the components of the semaphore are deleted and all the control blocks become free from it.

6 MrsP Testbed on RTEMS

In order to evaluate the implementation of MrsP protocol in RTEMS and its properties, several tests were developed. The developed tests cover functional aspects and timing aspects and in some cases the performance of MrsP is compared against OMIP.

Due to the amount of code produced for each of the tests developed in the context of the testbed, it was decided to present an example of a test case[8] in Appendix 3 in order to make it easier for the reader to read the report and to understand the structure of a test sample in RTEMS.

6.1 Test Objectives

All the tests were carried out using MrsP as a resource sharing protocol, and in some relevant cases, the tests are adapted to OMIP, so that a comparison between the two protocols is possible, not only at the level of desirable properties, but also at the level of temporal performance.

For the most relevant internal operations of those protocols, such as obtaining or releasing a mutex, the timing performance of the RTEMS's multiprocessor locking protocols is measured. The same is applied to the uniprocessor protocols of RTEMS, PIP and PCP, in order to analyse the induced overhead that exists due to the adaptation to multiprocessor systems.

6.2 Simulation conditions

In order to proceed with the analysis of the temporal behaviour of the protocols, each test was executed 500 times and, with the obtained results, all the analyses were performed using R [7], a programming language oriented to the analysis and visualization of data.

All the obtained data, about the temporal performance of the protocols, is represented using boxplots and the mean execution time of each test case is also mentioned.

A boxplot is a chart widely used to represent information in statistical studies, representing data according to the following image.

[8] The test chosen was Test Case 8, since it is the simplest and most understandable test developed from the testbed presented here.

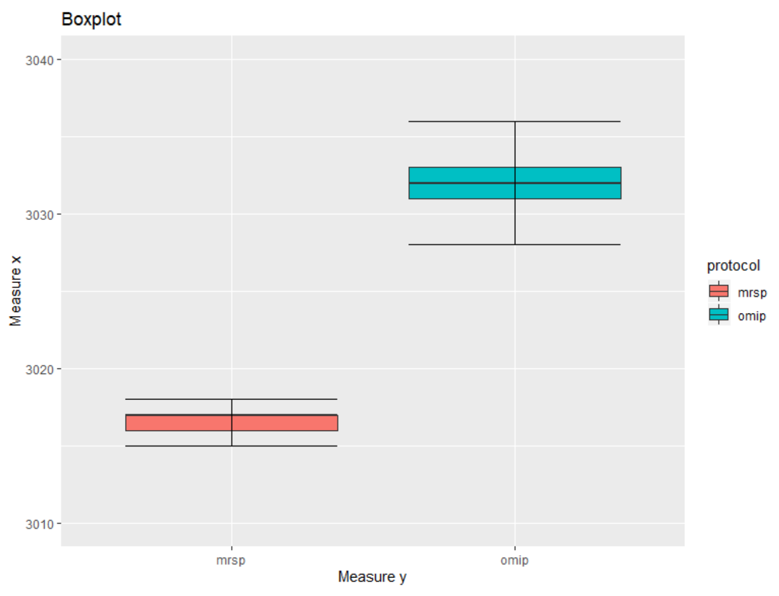


Figure 19 Example of a Boxplot

This graph represents the distribution of the data between the minimum, first quartile, the median, third quartile and the maximum. The region between the first and third quartile is called interquartile range (IQR) and it represents always 50% of all the obtained data, the other 50% are placed out of that range. On this study, this graph is used since it is great to represent how data varies as a function of the different executions of the tests. This is important to take into account on real-time systems, since in the worst-case scenario the response-time of the task must be predictable. For example, a certain operation may present a faster mean execution time then an alternative one, however, if the alternative presents less dispersion it may be considered better since it is more deterministic, considering the differences obtained.

The timing tests presented in this chapter are performed using an emulator, Qemu [41].

When using an emulator there is always some overhead induced by its use when compared to the use of a real machine. However, this is not considered a problem since all results are collected in the exact same environment.

6.3 RTEMS Concepts

In this section, some important concepts are explained in order to help the reader to understand better the testbed presented here.

In first place it is important to explain the concept of INIT task, largely used on this chapter. The INIT task, on RTEMS, abbreviation from initialization tasks, are tasks defined on the RTEMS application configuration and automatically initialized when the application is executed. These tasks are usually used in the creation and initialization of the necessary resources and other application tasks, although they present the same characteristics of the other tasks, such us a priority and identifier, and are equally scheduled by the system according to its general rules.

To support the reader in understanding the behaviour of the system in some of the test cases, the explanation of the test is followed by a diagram describing the flow of the various tasks, resources and processors present in the system. All such diagrams have the same graphical elements, representing the different operations that may be performed during each of the tests, thus contextualizing them. These elements are described in the following figure.

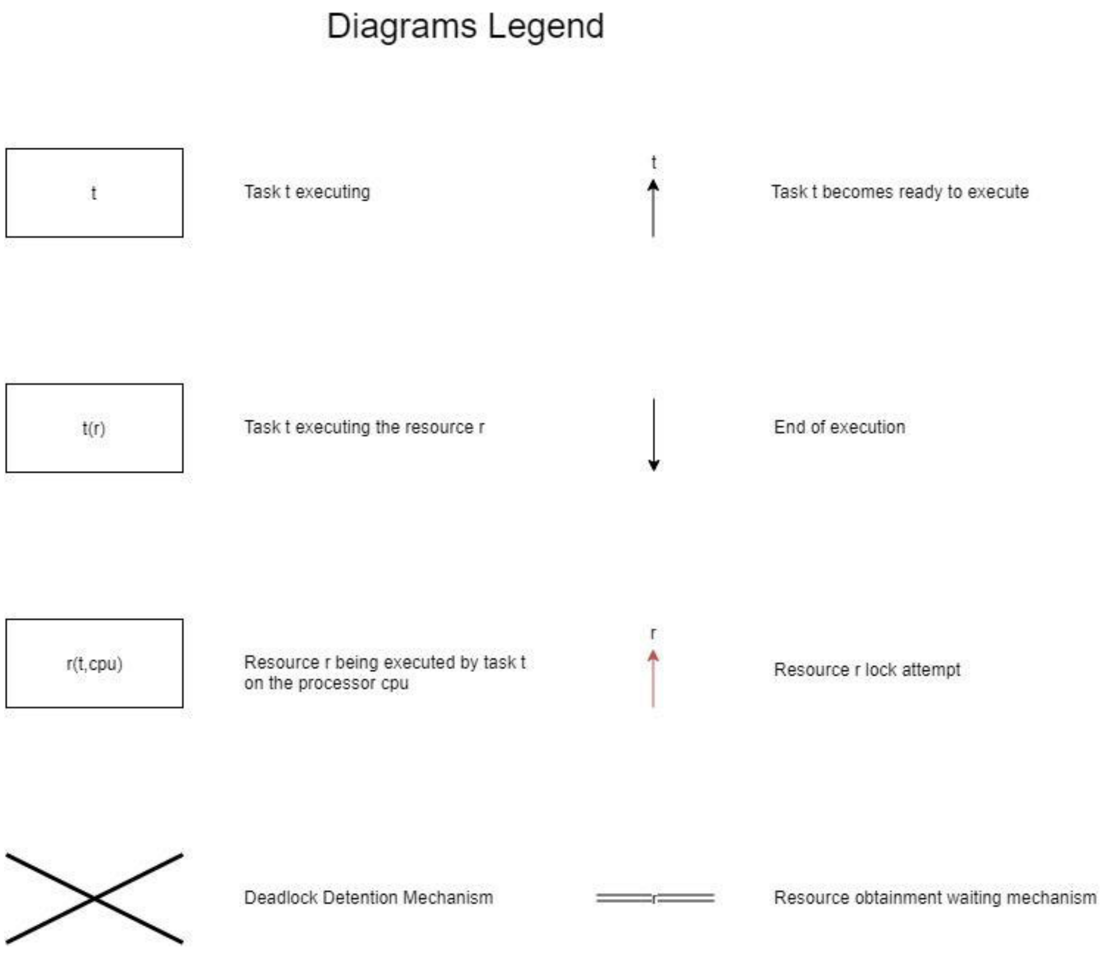

Diagrams Legend
t
Task t executing
t
Task t becomes ready to execute
t(r)
Task t executing the resource r
End of execution
r(t,cpu)
Resource r being executed by task t on the processor cpu
r
Resource r lock attempt
Deadlock Detention Mechanism
r
Resource obtainment waiting mechanism

Figure 20 Test Cases Diagrams Legend

6.4 Test Case 1

This test covers two of the desirable properties of MrsP. In first place, it asserts if a MrsP resource is assigned a set of ceiling priorities, one per scheduler instance. Secondly, it verifies if a task's priority is immediately raised to the scheduler's ceiling priority when it tries to obtain a resource on its host processor.

6.4.1 Test Description

In this test, one MrsP semaphore is created. In order to define multiple ceiling priorities, the test must run on a partitioned scheduling environment where each processor is assigned a scheduler instance and a ceiling priority.

Table 9 Test Case 1 Scheduler Assignments

SCHEDULER INSTANCE	PRIORITY CEILING	PROCESSOR
SCHED_A	2	CPU1
SCHED_B	3	CPU2

In order to assert the second property, related to the priority change induced to a task that tries to obtain a MrsP resource, two tasks are used, INIT and TASK0, and each of them must have an affinity set to each scheduler instance of the system.

Table 10 Test Case 1 Tasks Properties

TASK NAME	PRIORITY	SCHEDULER INSTANCE
INIT	4	SCHED_A
TASK0	6	SCHED_B

6.4.2 Test Behaviour

The INIT task creates a resource and defines the set of priority ceilings for each scheduler. After that, the validation of the resource creation and the priority ceilings of the resources for each of the processors is also performed. Then, TASK0 is created and released onto CPU2, while INIT is running on CPU1. Both of them try to obtain the resource and then, their priority is verified when each task tries to obtain the resource. The priority is immediately raised to the priority ceiling assigned to the task's processor.

6.4.3 Results

This test passes successfully when the priority ceilings are correctly defined for each one of the processors and, when trying to lock the resource, the priority of the task is successfully raised to one of the priority ceilings defined offline according to the host processor.

For the verification of the priority of the tasks on this test, the Task Manager's directive *rtems_task_set_priority* was used.

6.4.4 MrsP and OMIP Comparison

Since OMIP is based on priority inheritance instead of priority ceiling, in this test case, it is not possible to choose OMIP as an alternative, its adaptation would be insignificant as the sample only tests properties related to priority ceiling. In this way, no comparisons between MrsP and OMIP were made.

6.5 Test Case 2

This sample verifies if the access to a MrsP Semaphore is dealt in FIFO (First In First Out) order.

6.5.1 Test Description

This test consists of a set of four tasks trying to access the same MrsP resource in a system composed of three processors, CPU0, CPU1 and CPU2, as defined in the following table. This test uses global scheduling. To ensure synchronization and the desired order of execution of each task, active waiting is used and idle time is induced in the various tasks during their execution, which ensures that in all the test instances, each task always runs on the same processor.

Table 11 Test Case 2 Tasks Properties

TASK	PRIORITY	PROCESSOR
INIT	4	CPU0
TASK0	13	CPU1
TASK1	9	CPU2
TASK2	10	CPU0

6.5.2 Test Behaviour

In this sample, because of the induced active waiting and idle time, the first task trying to obtain the resource is the INIT Task on CPU0. After that, TASK0 tries to obtain the resource on CPU1, then TASK1 on CPU2 and, finally, TASK2 on INIT's host processor, CPU0.

Despite the priority of each task, the order in which the tasks obtain the resource must be the same as the order of arrival (FIFO order), as it can be seen in the following figure.

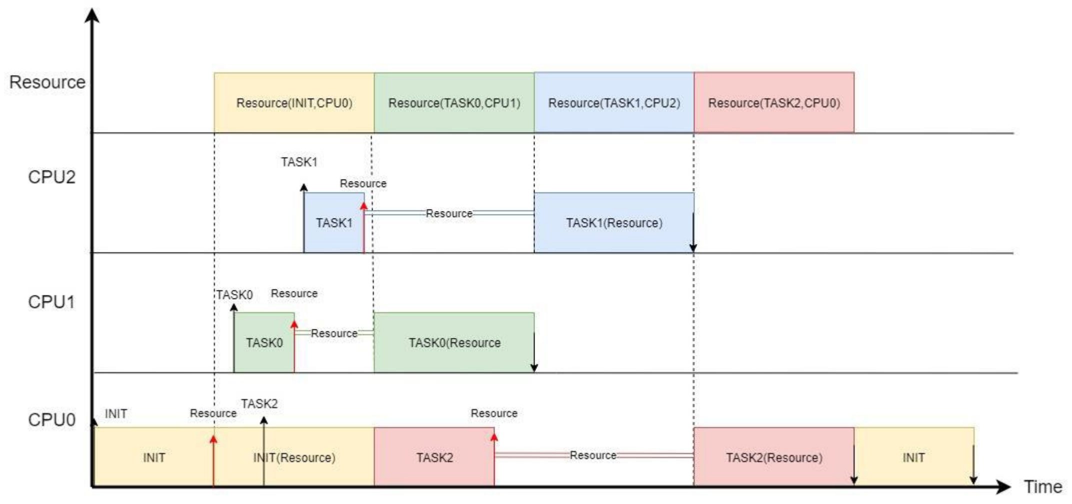


Figure 21 Test Case 2 desired behaviour

6.5.3 Results

This test passes successfully when all the tasks executed their critical sections by their arrival order, thus following a FIFO order.

6.5.4 MrsP and OMIP Comparison

Adapting this test to use OMIP instead of MrsP as the Resource Sharing Protocol, OMIP does not follow the expected FIFO order in this case, thus not granting access to the resource in the desired order. In this test, OMIP deals with the order of obtention of the resource in the same way as the Priority Inheritance Protocol, in which OMIP is based. Hence, the order of obtention of the resource is by the priority of the task, which means that the task with higher priority level is the task that is selected to execute next.

In terms of the execution time of this test, MrsP and OMIP present quite discrepant results, which can be explained by the different behaviour presented by each one of the protocols, as can be seen in the following chart.

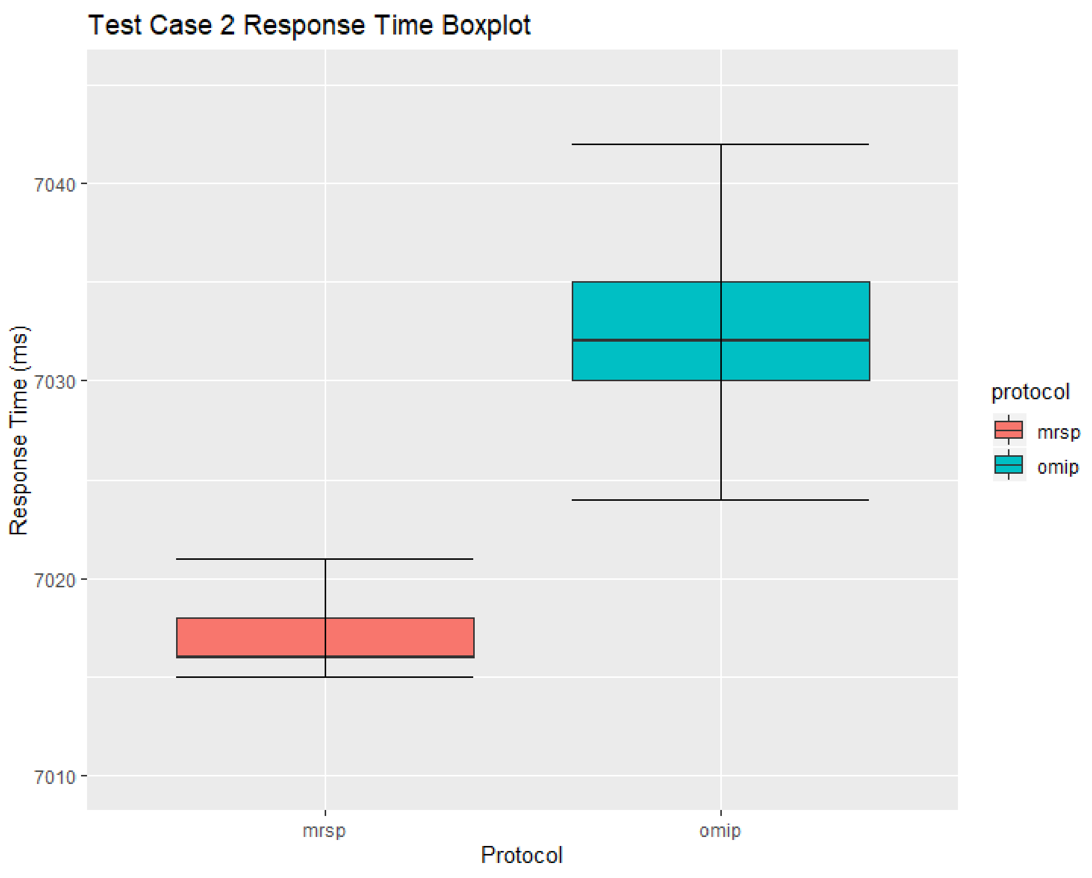


Figure 22 Response Time on Test Case 2

It is easy to see that in this case MrsP presents response times of, on average, 7020 milliseconds, which are lower than OMIP, which on average takes 7035 milliseconds of execution time.

Moreover, it is possible to observe that the execution time, in general, is very long, something that can be explained by the deliberately induced idle time and active waiting in the sample in order to guarantee the synchronization between the different tasks. Therefore, it appears that in this case, considering that the behaviour of the two protocols is not similar, the MrsP protocol not only fulfils the requirements, behaving as expected, but presents better performance than the multiprocessor alternative, OMIP, that presents an average difference of 15 milliseconds in terms of execution time.

6.6 Test Case 3

Test Case 3 tests if the waiting mechanism, when a task is trying to obtain a resource already held by another one, operates in an active way, performing busy waiting, with its original priority immediately raised to the Ceiling Priority.

6.6.1 Test Description

In this test case, three tasks are needed, the INIT task, TASK0 and TASK1, on an environment composed by two processors, CPU0 and CPU1.

As in the previous test, the synchronization between tasks and their assignment to the processor, according to the table below, are granted with busy waiting, induced idle time and to the tasks initial defined priority. Since the INIT task presents greater priority than the other ones and its execution starts during the system's initialization, it is be able to execute on CPU0 initially and neither TASK0 nor TASK1 can preempt it.

Table 12 Test Case 3 Tasks Properties

TASK	PRIORITY	CPU
INIT	4	CPU0
TASK0	13	CPU1
TASK1	13	CPU1

6.6.2 Test Behaviour

In this test, all the tasks try to obtain the same resource, which presents a Priority Ceiling Level of 1. These tasks also follow a specific order of release into the system and, therefore, of obtention of the resource.

It is expected that TASK0 must wait until the INIT task finishes the execution of the resource, actively, with its priority raised to the Ceiling Priority, and, in the same way, TASK1 may only start its execution after TASK0 releases the mutex, proving that the waiting mechanism operates in an active way, because TASK0 does not leave the processor while waiting. This behaviour is summarized on the following diagram.

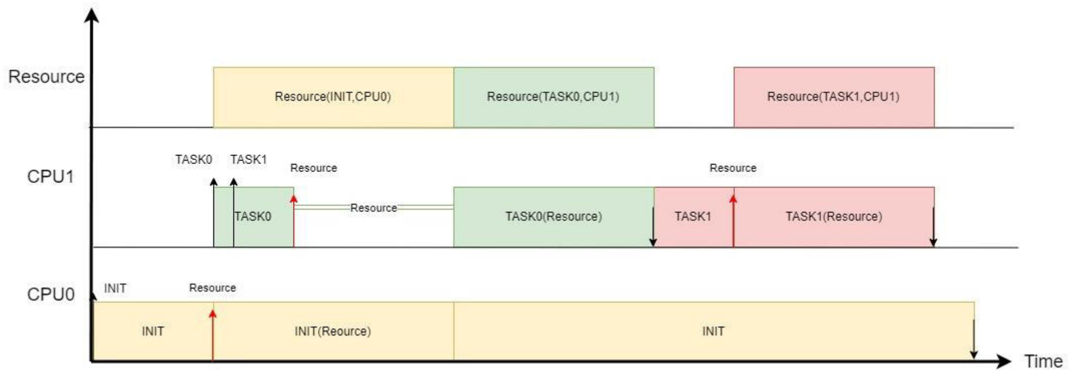


Figure 23 Test Case 3 Desired Behaviour

6.6.3 Results

Since TASK1 only starts its execution after TASK0 releases the resource, this test passed successfully. It means that TASK0, while waiting to access the resource is still executing and, in this way, TASK1 may not be dispatched, since all the CPUs are currently occupied by INIT and TASK0

6.6.4 MrsP and OMIP Comparison

Adapting this test case to use OMIP semaphore, instead of using MrsP as the resource sharing protocol, the results are not the same.

Using OMIP as the resource sharing protocol, the waiting mechanism is suspension based, which means that, when TASK0 tries to obtain the resource, since INIT is already executing it, TASK0 becomes suspended, leaving the processor and giving TASK1 the opportunity to execute. Although, when the INIT task releases the resource, it becomes available to be executed by TASK0, so TASK0 is dispatched to the processor and starts the execution of the resource, which forces the system's scheduler to perform more operations using OMIP than using MrsP, since, using MrsP, TASK0 only leaves the system when it finishes its execution.

In terms of time, in this case, the waiting mechanism of MrsP presents a better performance. The obtained results show that this test took 3017 milliseconds using MrsP, while using OMIP it took 3032 milliseconds to execute.

The obtained execution times are relatively high, since idle time is induced during the test case, which is responsible for a considerable delay.

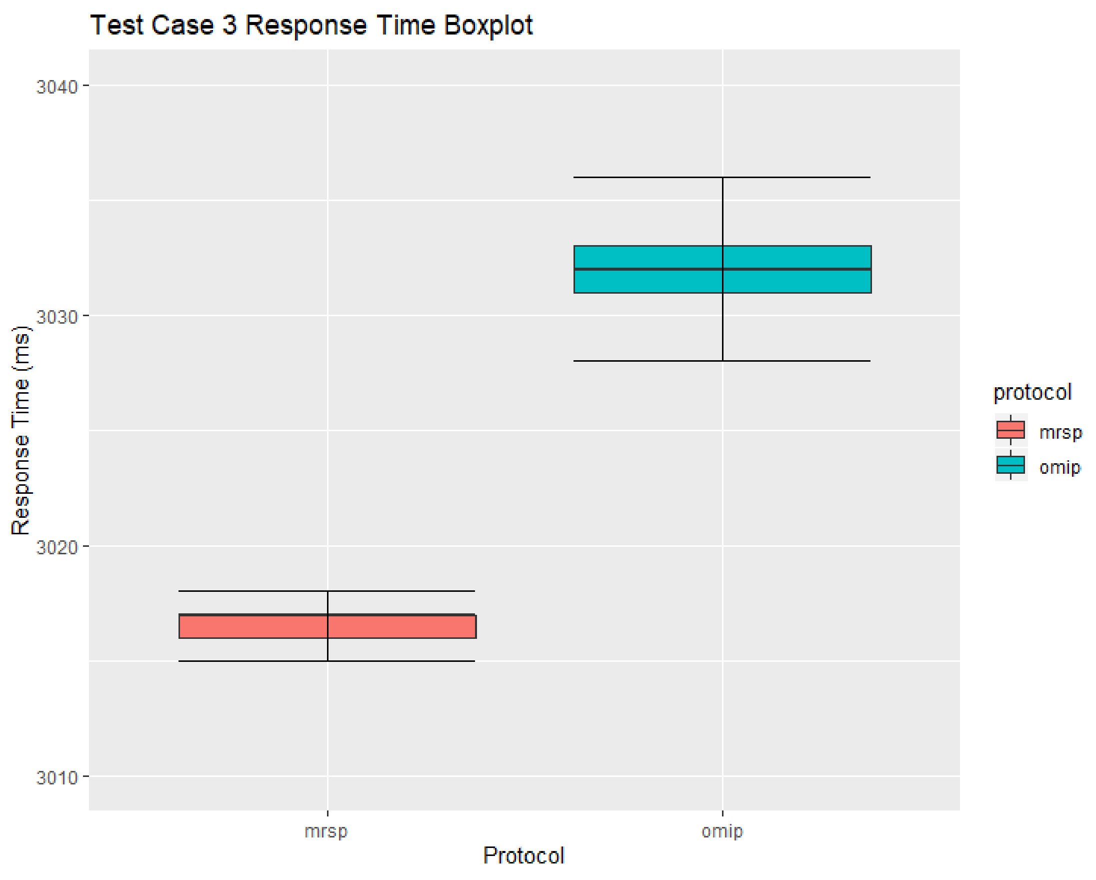


Figure 24 Response Time on Test Case 3

The differences present by each one of the protocols, can be explained by the differences presented by each protocol's waiting mechanism and by the protocol's internal architecture.

The waiting mechanism of MrsP enables a task waiting to access the resource to be able to continue executing on it, while using OMIP, a task is forced to leave the processor due to the suspension mechanism. Naturally, this behaviour induces more operations when using OMIP than using MrsP, which explains the difference between both execution times

Beyond that, as explained on Chapter 3, OMIP waiting queue structures are more complex than MrsP, and, since these structures are actually used to manage the waiting mechanism, it is also a reason for the existence of such relevant timing differences.

6.7 Test Case 4

This test asserts if the waiting mechanism of the protocol is preemptive. On MrsP if a task is waiting to access a resource it must perform busy waiting (which is verified on the previous test case – test case number three).

However, this mechanism must be preemptive, this is, if a task with higher priority than the waiting task arrives to the same scheduler instance, ready to execute, the waiting task must be preempted, in order to guarantee the progress of the higher priority task.

6.7.1 Test Description

Just like test two and test three, this test occurs in a global scheduling environment composed by two processors, CPU0 and CPU1 and the tasks synchronization and assignment to a certain processor is granted by induced idle time, active waiting and, finally, by the task's priority.

Three tasks are created in this test case, the INIT task, TASK0 and TASK1, each one with its own priority, executing in one specific processor, as described in the following table.

Table 13 Test Case 4 Tasks Properties

TASK	PRIORITY	PROCESSOR
INIT	10	CPU0
TASK0	13	CPU1
TASK1	6	CPU1

In this case, two shared resources are used (see Table 14), MrsP used directly on the test to assert the behaviour of the protocol and MrsP1 which is used only to increase the priority of the INIT task (due to the ceiling protocol), ensuring that it runs on CPU0, and that other tasks must execute on CPU1.

Table 14 Test Case 4 Resource Definition

RESOURCE	PRIORITY CEILING
MrsP	8
MrsP1	3

6.7.2 Test Behaviour

In the first place, INIT obtains MrsP and MrsP1, in a nested way, increasing its own priority to the value of 3, higher than all the other tasks of the system. Meanwhile, TASK0 arrives to the system and starts executing on CPU0, trying to obtain the first resource, MrsP. Since this resource is already held by INIT, TASK0's priority must be raised to a value of 8, and the task must perform busy waiting. However, while TASK0 waits to access the resource, TASK1 arrives to the system, with a priority level of 6, and, since it has greater priority than TASK0, which is currently busy waiting on CPU1 with a priority of 8, TASK0 must be preempted, in order to let TASK1's execution proceed.

When TASK1 leaves the system, TASK0 may return to its own execution. This behaviour is depicted in the following image.

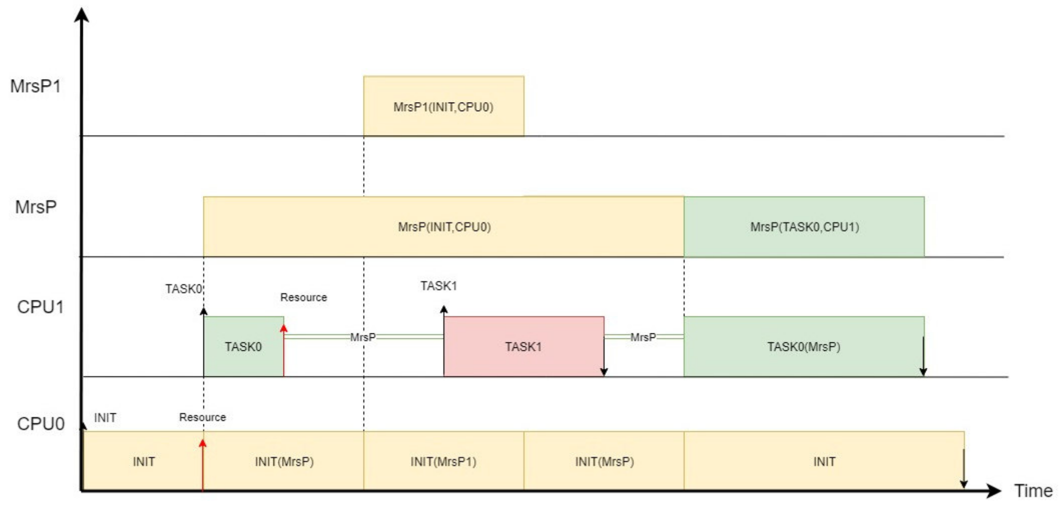


Figure 25 Test Case 4 Desired Behaviour

6.7.3 Results

The test passes successfully using MrsP semaphores.

As expected, when TASK1 enters the system and becomes ready to execute, it preempts TASK0, being able to execute on CPU1, proving that the waiting mechanism is preemptive.

6.7.4 MrsP and OMIP Comparison

In the previous test case, it is shown that the OMIP waiting mechanism is based on suspension, meaning that a task waiting to access a resource is not able to continue executing. However, in this test case, using OMIP semaphores, the test's behaviour is similar as using MrsP. Therefore, a comparison between both protocols is relevant, in order to verify which one presents better performance.

The obtained data is presented in the following chart, showing that MrsP presents faster execution times than OMIP, on this test case. Once again, the attained response times are relatively high, since idle time is highly induced on the system.

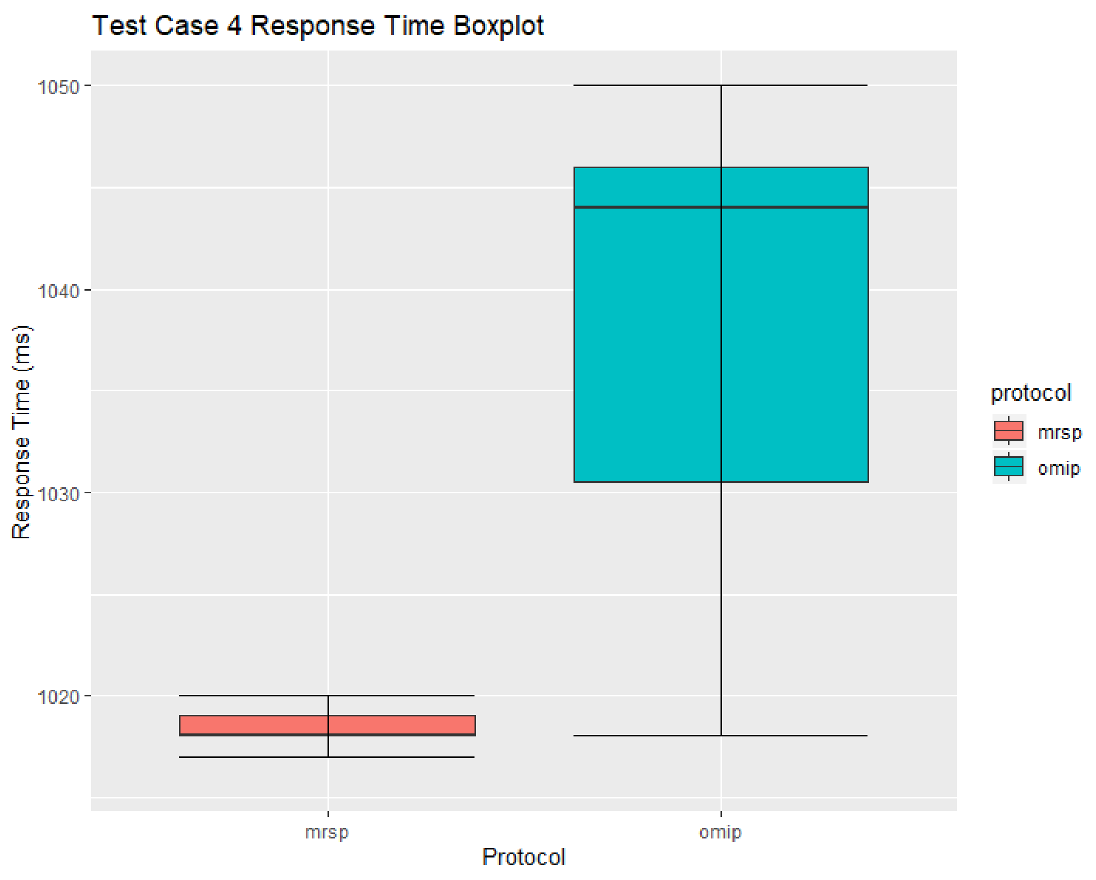


Figure 26 Response Time on Test Case 4

The obtained results show that, on average, MrsP has a faster response time, 1019 milliseconds, while OMIP takes about 1040 milliseconds to complete. But it is possible to see that OMIP presents huge dispersion in terms of execution time. This dispersion may happen since, when the system is executing on global scheduling, OMIP degenerates to the priority inheritance protocol, designed to operate under single processor systems. Therefore, the protocol does not perform in the best way on multiprocessor platforms, which may disturb its performance.

6.8 Test Case 5

This test asserts if the helping mechanism of MrsP may be initiated by a task waiting to own any of the resources held by the task that needs to be helped, ultimately preventing it from making progress. When the helper is no longer depending on the helped task, this procedure must be finished.

6.8.1 Test Description

RTEMS provides a helping mechanism implemented on the scheduler. Both MrsP and OMIP use this functionality to provide help between tasks, allowing a task to migrate from one instance of the scheduler to another, as mentioned on chapter 4.

On this test case partitioned scheduling is used, three scheduler instances are defined, each one associated with a processor, as represented on the following table.

Table 15 Test Case 5 Scheduler Assignments

SCHEDULER INSTANCE	PROCESSOR
SCHED_A	CPU0
SCHED_B	CPU1
SCHED_C	CPU2

In order to assert the property tested here, four tasks are created, the INIT task, TASK0, TASK1 and TASK2, although, INIT does not interfere directly with tests results, since it is only used to perform the creation of tasks and resources, being considered as the main task. The definition of the test's tasks is presented below.

Table 16 Test Case 5 Tasks Properties

TASK	PRIORITY	SCHEDULER INSTANCE
INIT	4	SCHED_A
TASK0	12	SCHED_B
TASK1	3	SCHED_B
TASK2	12	SCHED_C

In addition to the tasks pertaining to this test case, three resources are also needed, MrsP, MrsP1 and MrsP2, all of them presenting the same priority ceiling level for each scheduler instance.

Table 17 Test Case 5 Resource Definition

RESOURCE	CEILING PRIORITY
MRSP	9
MRSP1	7
MRSP2	4

6.8.2 Test Behaviour

In the first place, TASK0 arrives to the system, starting its execution on CPU1 and obtaining all the resources, by a decreasing order of priority ceilings, increasing its priority to 4.

Meanwhile, TASK2 also become ready to execute, on CPU2. On CPU2, Later, it tries to obtain MrsP1, one of the resources held by TASK0, currently executing MrsP2.

Moreover, TASK1 enters the system and must preempt TASK0, since it has greater priority and it is assigned to the same scheduler instance.

According to the property analysed here, TASK2, since it is waiting to access one of the resources held by TASK0, must help it, so, TASK0 must be able to migrate to its host scheduler instance, in order to continue making progress.

When TASK0 finishes the execution of MrsP1, TASK2 is not depending on it anymore and therefore it can start executing, while at the same time TASK0 must return to its original scheduler instance, even if it is still not able to execute, as described on the following image.

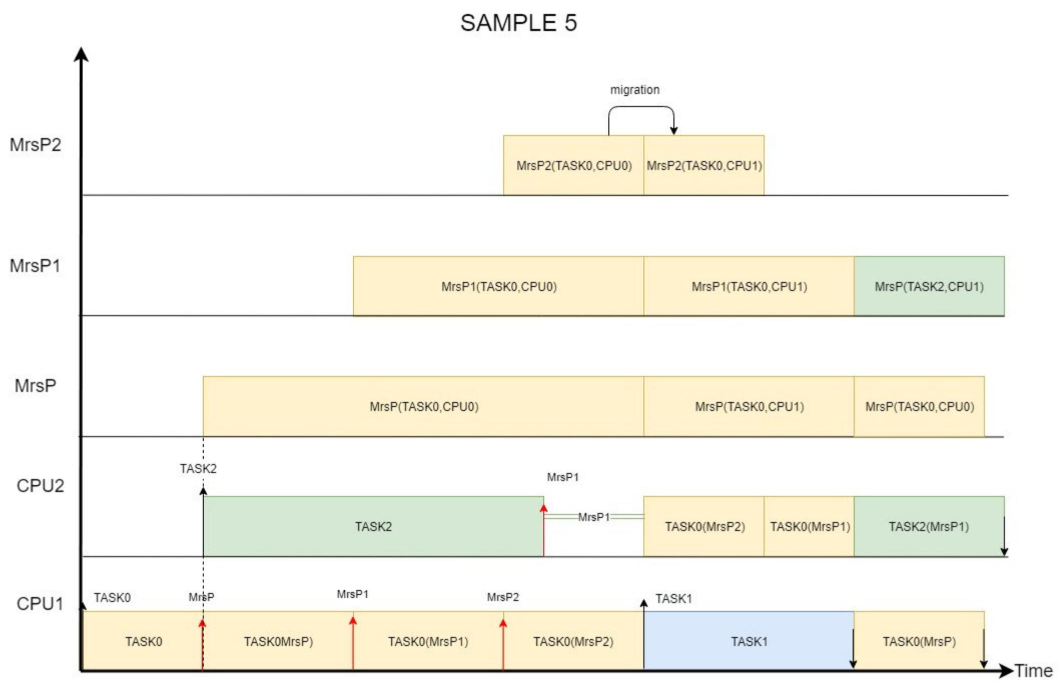


Figure 27 Test Case 5 Desired Behaviour

6.8.3 Results

This test passes successfully, the migration of TASK0 from CPU1 to CPU2 occurs when TASK2 starts waiting to access the desired resource, MrsP1, and when TASK2 is no longer depending on TASK0, the helping mechanism stops and TASK2 is able to execute the resource on its own processor, CPU2, while TASK0 finishes it on CPU1, its host processor, after TASK1 leaves.

6.8.4 MrsP and OMIP Comparison

As expected, the helping procedure presented by OMIP and MrsP in this test case is similar, since the helping mechanism on both protocols is the same, the Scheduler Helping Protocol available on RTEMS.

In this way, it is expected that the protocols do not present relevant differences, in terms of timing performance, as presented on the following chart.

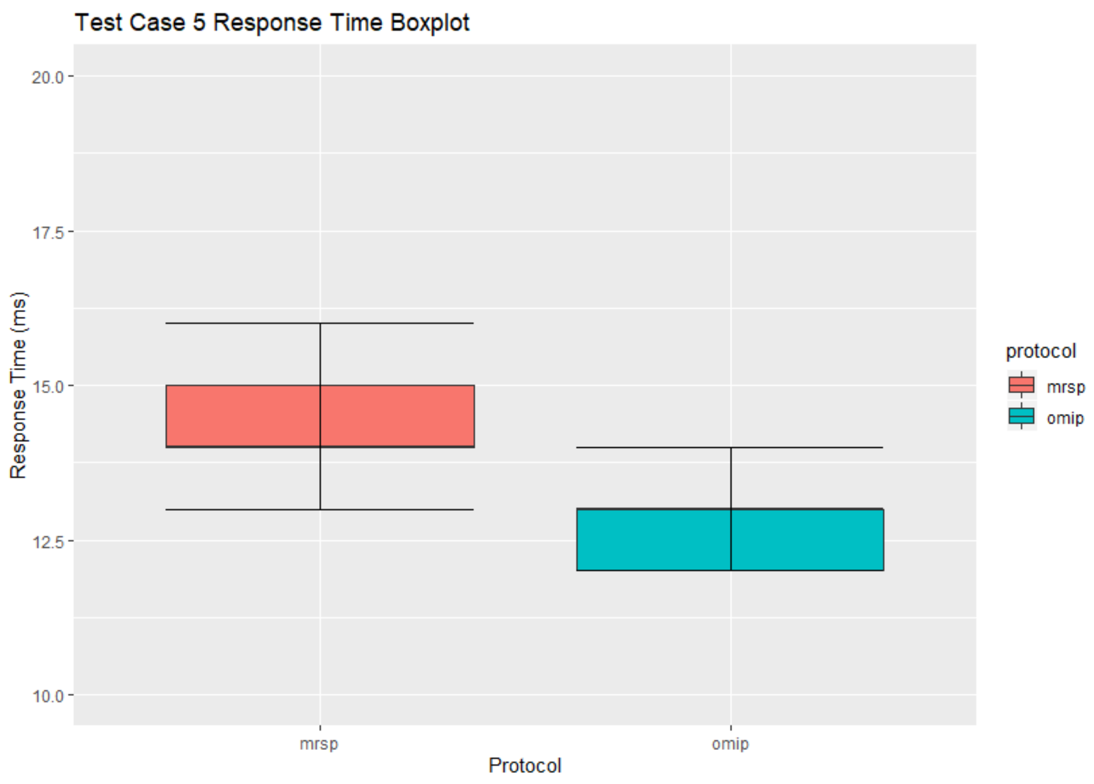


Figure 28 Response Time on Test Case 5

The data presented in the graph show that the response value amplitudes of MrsP and OMIP are intersected, OMIP presents better timing performance in this use case, 12.97 milliseconds, while MrsP takes 14.47 milliseconds. These results raise suspicions about the time required for protocols to perform their internal operations. Since in this sample the protocols present similar behaviours, the fact that MrsP took on average longer execution time could happen because it takes longer to perform its internal operations, namely obtaining and releasing the resource.

6.9 Test Case 6

This test checks if a task, when being helped, executes on the remote processor with the priority of the helping task. Since the Priority Ceiling, under MrsP, may be different for each processor, it means that the resource may have different importance depending on the processor, so the helped task shall not maintain its own priority when executing on the helper's host processor.

6.9.1 Test Description

On Test Case 6, three processors are needed, CPU0, CPU1 and CPU2. This test operates under a partitioned scheduling environment, so scheduler instances are necessary to manage the processors. In this case, each processor is managed by one of these instances, as described in the following table.

Table 18 Test Case 6 Scheduler Assignments

SCHEDULER NAME	PROCESSOR
SCHED_A	CPU0
SCHED_B	CPU1
SCHED_C	CPU2

In terms of the task set presented on this sample, there are five tasks, INIT, which does not interact directly with the assertions made here, it only exists to create and manage all the necessary resources and tasks in the system. Both, the priorities and schedulers for each task are described in the table below.

Table 19 Test Case 6 Tasks Properties

TASK NAME	PRIORITY	SCHEDULER
INIT	11	SCHED_A
TASK0	10	SCHED_B
TASK1	3	SCHED_B
TASK2	8	SCHED_C
TASK3	6	SCHED_C

In the end, only one resource is needed to proceed with the test described here. To make the assertions needed, a set of different ceiling priorities per processor is needed, on CPU1 the priority ceiling is 9, while on the other processors the correspondent value is 5, so the priority changes when the helped task migrates, which is the feature that it is being tested.

6.9.2 Test Behaviour

After the main task, INIT, TASK0 is the first task to enter the system, claiming the MrsP resource in first place, obtaining it successfully, which induces a priority raise to the resource's priority ceiling on its host processor, priority value 9. Slightly later, TASK2 starts executing on CPU2 and also tries to obtain the resource, having its own priority raised to 5, however it has to wait for TASK0 to release the resource.

Meanwhile, TASK1 arrives to the system, on CPU1, with a priority of 3, greater than TASK0's priority, so a preemption must be performed, and TASK1 shall execute instead of TASK0. Because of MrsP helping mechanism, TASK0 migrates to CPU2, where TASK2 is executing, waiting to access the resource, and TASK0 must execute with TASK2's priority.

Finally, in order to validate the property, while TASK0 is executing on a remote processor, TASK3 becomes ready to execute on processor CPU2. TASK3 has a priority value of 6, greater than TASK0's priority on CPU1 but less than TASK0's expected priority on CPU2 while being helped. This way, if TASK0 is preempted in order to let TASK3 execution proceed, the property is not verified, if TASK0 continues to execute normally, it means that it certainly is executing with TASK2's priority, which mean that the property is valid.

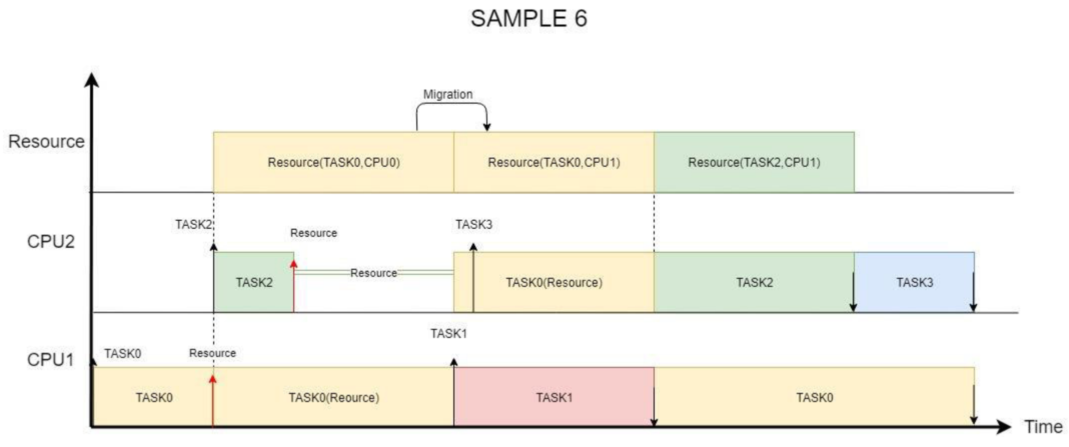


Figure 29 Test Case 6 Desired Behaviour

6.9.3 Results

This test passed successfully. As described before, TASK3's entrance to the system did not interfere with TASK0's execution, which means that TASK0 is executing on CPU2 with TASK2's priority, so the property is verified.

The assertions performed on this test case take advantage of the RTEMS directive *rtems_get_current_processor* that returns the processor where a certain task is currently executing, allowing the validate if the expected migrations were performed.

6.9.4 MrsP and OMIP Comparison

Using OMIP as an alternative to MrsP, as Resource Sharing Protocol, the observed behaviour is the same. When a migration occurs, the helped task inherits the priority of the helper task. The results of the execution time obtained for each protocol are described here. This use case presents slightly better results when using OMIP, with an average value of 13.99 milliseconds, while using MrsP the average value obtained is 15.03 milliseconds, almost one millisecond of difference between the two protocols.

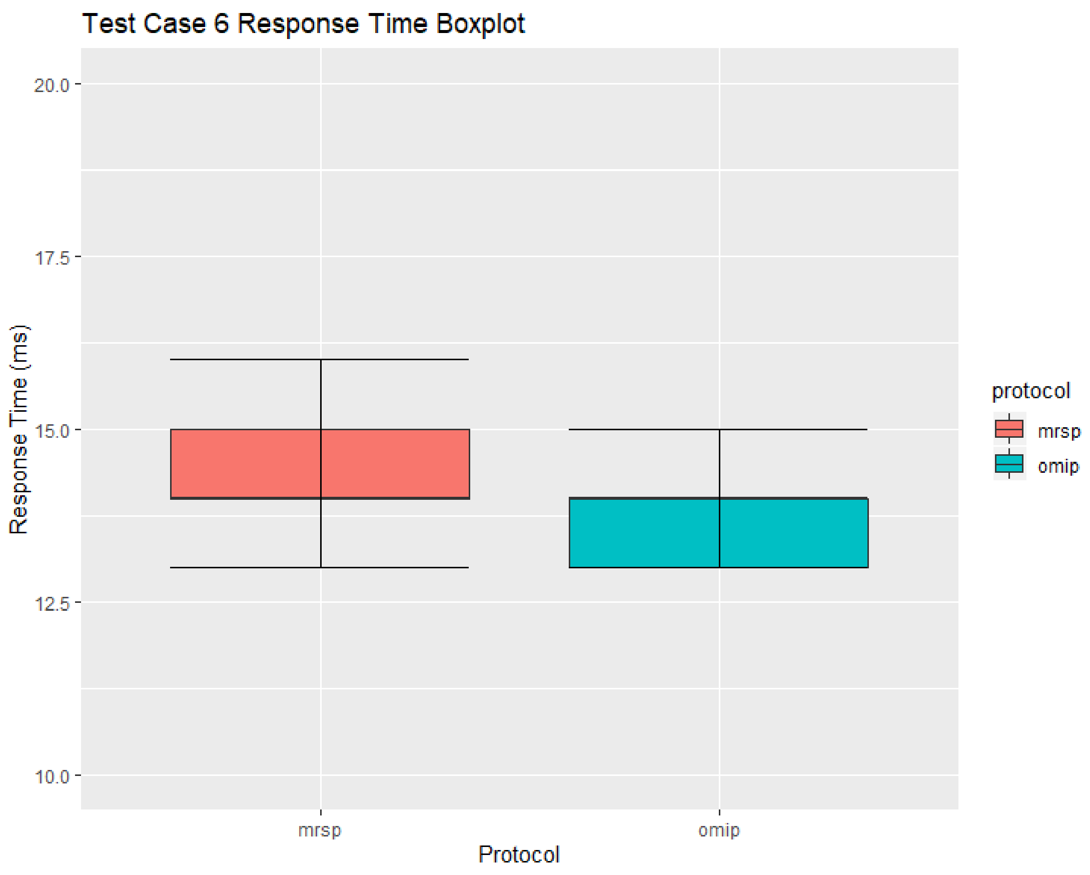


Figure 30 Test Case 6 Response Time

In fact, it can be observed that there are slightly differences between both protocols. In most of the obtained results MrsP and OMIP's results fit on the same range of values.

In general, both protocols present similar execution times, although, it can be observed that OMIP presents a better worst-case scenario than MrsP.

Concluding, since the behaviour of the system, using both protocols is similar, there are some suspicions that the internal operations of MrsP and OMIP take different execution times, namely obtaining and releasing a resource.

6.10 Test Case 7

This test case validates if a task that is being helped, executing on the helper’s scheduler instance, may only execute with the helper’s priority, without having its priority changed at any moment during the helping procedure.

6.10.1 Test Description

On this test, the environment is composed by three processors, CPU0, CPU1 and CPU2. The execution of tasks among them is dealt in partitioned scheduling, where each scheduler instance is assigned to one processor, as described on the following table.

Active waiting and induced idle time is used in order to guarantee the desired flow and synchronization between every tasks on the system.

Table 20 Test Case 7 Scheduler Assignments

SCHEDULER INSTANCE	PROCESSOR
SHCED_A	CPU0
SHCED_B	CPU1
SHCED_C	CPU2

The set of tasks needed to produce the desired assertions is composed by the INIT task, used only in the creation and initialization of tasks and resources, not interfering with the test directly, and another four tasks, defined according to the next table.

Table 21 Test Case 7 Tasks Properties

TASK NAME	SCHEDULER	PRIORITY
INIT	SCHED_A	11
TASK0	SCHED_B	12
TASK1	SCHED_B	3
TASK2	SCHED_C	12
TASK3	SCHED_C	8

Finally, two resources are used in this test case, MrsP and MrsP1, both with the same priority for all the scheduler instances.

Table 22 Test Case 7 Resource Definition

RESOURCE	CEILING PRIORITY
MrsP	9
MrsP1	7

6.10.2 Test Behaviour

In first place, TASK0 arrives to the system on CPU1 and obtains the resource MrsP, having its priority raised to 9. Therefore, TASK2 on CPU2 also tries to obtain the same resource being forced to wait for its release.

Meanwhile, TASK1 becomes ready to execute as well, on CPU1 and it preempts TASK0, since it has greater priority. This way, TASK0 must migrate to TASK2's CPU (using the helping mechanism) and continue executing with a priority of 9, the current priority of TASK2, since a task, while being helped under MrsP semaphores must execute with the helper's priority.

Later, while being helped, TASK0 obtains the resource MrsP1, in a nested way. Normally, this nested access would result on a priority raise to the priority ceiling of the resource, 7. Although since TASK0 is being helped by TASK2, its priority must be immutable, remaining 9.

Finally, in order to assert the property, TASK3 is released to the system, on CPU2, where TASK0 is being helped, with a priority of 8, between the priority Ceiling of MrsP1, which is 7, and the expected priority of TASK0, 9, since while being helped, the task's priority must not be changed.

It is expected that the TASK3 is able to execute on CPU2, preempting TASK0, since the priority of TASK0 must remain 9, as represented on the following diagram.

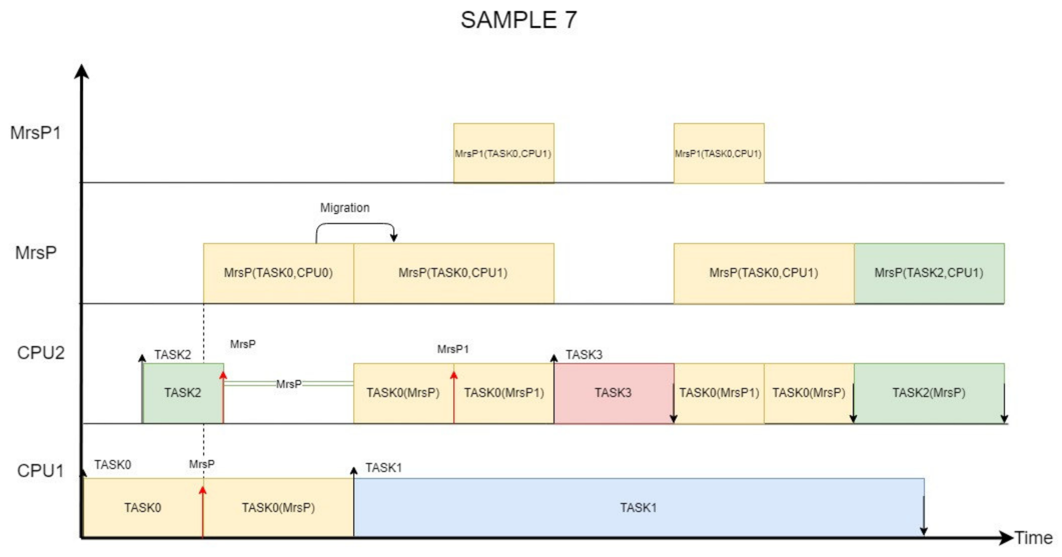


Figure 31 Test Case 7 Desired Behaviour

6.10.3 Results

This test passed successfully. TASK0's priority remains always the same, when executing on the remote scheduler instance, while being helped by TASK2. Therefore, when TASK3 enters the system it is able to execute, being the task with greater priority on that scheduler instance.

6.10.4 MrsP and OMIP Comparison

Using OMIP semaphores, this test case presents the same behaviour as using MrsP semaphores.

In terms of time, both protocols present the same average execution time of 2054 milliseconds, a relatively high response time due to the amount of idle time induced to force synchronization.

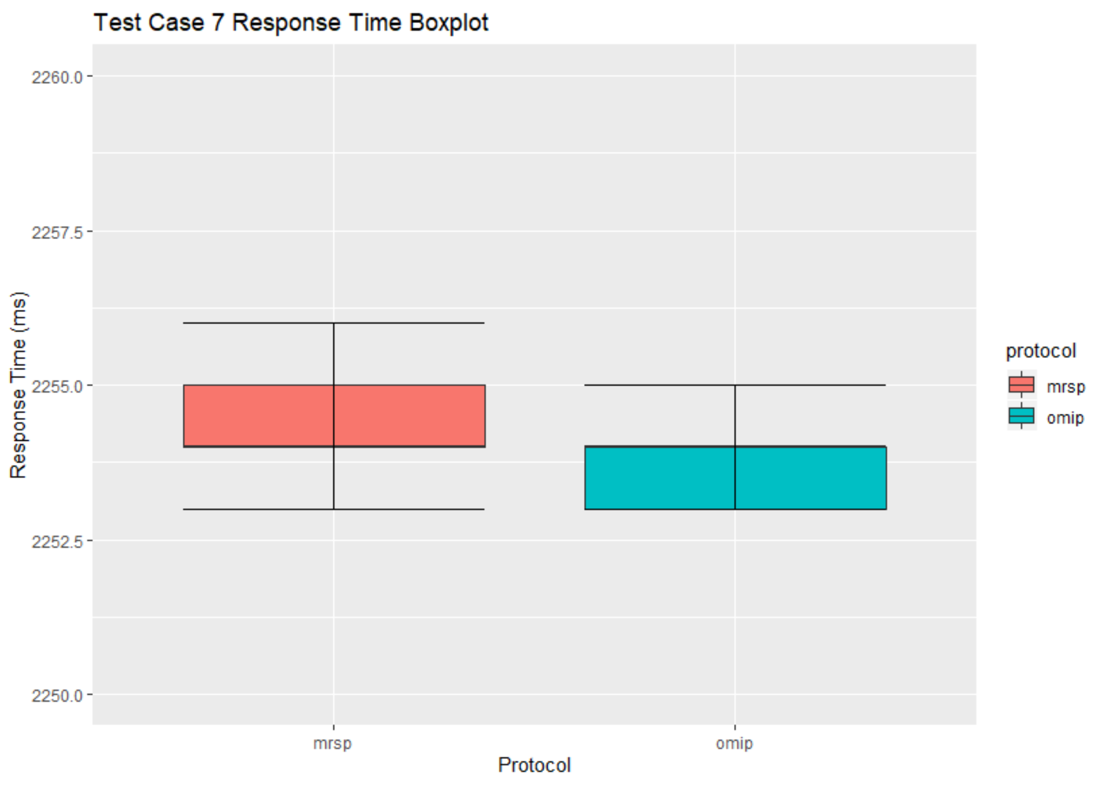


Figure 32 Response Time on Test Case 7

Based on the results obtained in this graph, although the average execution time of the protocols is the same, a larger range of results obtained in the MrsP can be verified and, moreover, it presents a worse scenario. Thus, it can be concluded that using MrsP as a resource sharing protocol in this use case will more likely take more time than using the OMIP protocol, once again alerting the possibility of different execution time between the MrsP and OMIP's internal operations.

6.11 Test Case 8

This test verifies if the helping mechanism can be used by a task that executes on the same host processor. This property is also tested on test case 12.

6.11.1 Test Description

If using MrsP semaphores a task could obtain two resources, in a nested way, and another task with greater priority entered the system, preempting the first one, and also obtaining one of those two resources, the higher priority task should help the holder of the resource, in order to allow its release and proceed executing. Nevertheless, a task is not allowed to obtain a resource with a priority ceiling smaller than its original priority.

In this way, a helping procedure may not be proceeded when the helper belongs to the same scheduler instance of the helped task, trying to obtain of the resource held by it.

In this case, there is the INIT task, with a priority level of 4 and a MrsP resource, which has a priority ceiling of 11, less than INIT's priority.

6.11.2 Test Behaviour

Initially, INIT tries to obtain the MrsP resource and it is expected that it cannot obtain the resource, since it presents lower priority ceiling than the task obtaining it.

This proves that the helping mechanism cannot work under the same processor, if the helper is trying to obtain one of the resources directly held by the helped task, because.

For this to happen, the helper would have to present a higher priority compared to the helped task's one. So, it would have also to present higher priority than the ceiling priority resource, the current priority of the helper.

Although, as mentioned before, because of other desirable properties of RTEMS that allow the helping mechanism on the helped task processor, this subject is addressed again forward on Test Case 12.

6.11.3 Results

As expected, INIT is not allowed to obtain the resource, since the Priority Ceiling of it is less than the priority of the obtaining task.

6.11.4 MrsP and OMIP Comparison

Since This Test is directly related to the Priority Ceiling and OMIP is based on priority inheritance, it is not possible to proceed with comparisons between the two protocols for this test case.

6.12 Test Case 9

Test case 9 validates if, while the holder of a resource is being helped, executing on a remote scheduler instance, lower priority tasks are prevented from making progress on its home scheduler instance, remaining free for its return after being helped.

6.12.1 Test Description

In this test, the system is composed by three processors, CPU0, CPU1 and CPU2. Each processor is handled by one scheduler instance, under a partitioned scheduling fashion, according to the following table.

Table 23 Test Case 9 Scheduler Assignments

SCHEDULER NAME	PROCESSOR
SCHED_A	CPU
SCHED_B	CPU0
SCHED_C	CPU1

The set of tasks of the test is composed by the INIT task, only used on the creation and management of other tasks and shared resources and another four tasks used to assert the property. The definition of the system's tasks is described below, on the table.

Table 24 Test Case 9 Tasks Properties

TASK NAME	SCHEDULER	PRIORITY
INIT	SCHED_A	4
TASK0	SCHED_B	10
TASK1	SCHED_B	7
TASK2	SCHED_B	15
TASK3	SCHED_C	10

Finally, one shared resource is needed on this test. This resource is defined with a priority ceiling level of 9, for all the processors of the system

6.12.2 Test Behaviour

In first place, TASK0 starts executing on its host processor, CPU1, obtaining the resource.

Later, TASK3 also arrives to the system on CPU2, and attempts to obtain the resource, waiting for its turn to claim its ownership, since it is held by TASK0.

Meanwhile, TASK1 and TASK2 enter the system on CPU1, one with greater priority than TASK0 and the other with less priority, respectively.

In this way, it is expected that TASK1 is dispatched on CPU1, preempting TASK0 and making it unable to proceed with its own execution.

Since MrsP presents a helping mechanism and TASK3, executing on CPU2, is attempting to access the resource held by TASK0. TASK0 must be able to migrate to TASK3's home scheduler instance, being able to continue executing.

Finally, TASK1 finishes its execution, while TASK0 is still executing the resource on TASK3's processor, and, since TASK2, ready to execute on CPU1, presents less priority than TASK0, that belongs to the same scheduler instance, it is expected that it may only be able to execute after TASK0 finishes its own execution, making CPU1 free to be used.

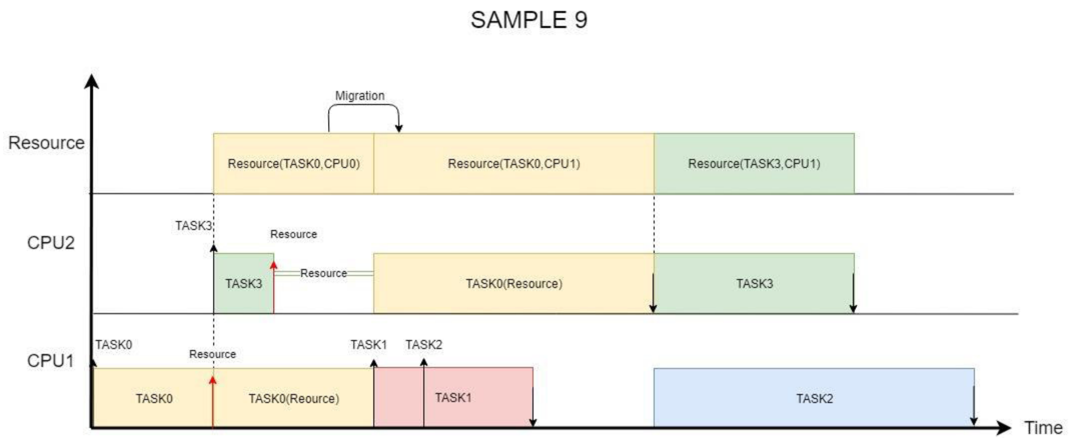


Figure 33 Test Case 9 Desired Behaviour

6.12.3 Results

This test passed successfully. After TASK1 leaves the system, TASK2 is prevented from making progress, until TASK0 finishes its own execution.

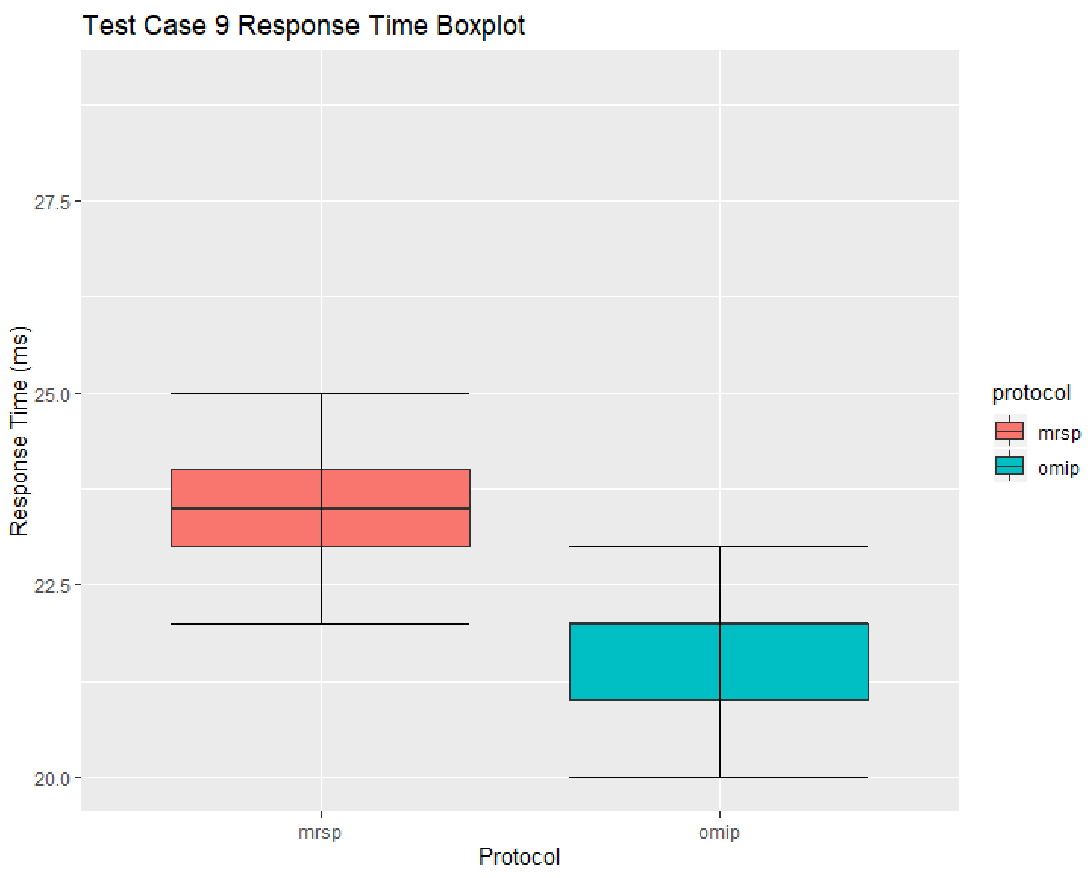
On RTEMS, this mechanism is dealt using idle tasks, defined by the task's sticky level, mentioned on Chapter 4. Those idle tasks take possession of the host processor of a task that migrates to another scheduler instance, due to the helping procedure, during a resource execution, in order to prevent lower priority tasks to execute there, releasing the scheduler as most as possible for the return of the migrated task.

6.12.4 MrsP and OMIP Comparison

OMIP does not present this feature, unlike MrsP. Under OMIP semaphores, when a task is executing on a remote scheduler instance, due to the helping mechanism, the host scheduler instance of that task is completely released to be used by other processes of the system, such as lower priority tasks.

This differences on the behaviour between MrsP and OMIP may not be very considerable when comparing the differences on the execution times presented by the execution of shared resources under them.

Through the results shown in the following graph, it is verified that, on average, the response time obtained from the use of MrsP semaphores is about 23 milliseconds, whereas using OMIP the test instances take about 22 milliseconds to execute.



As suspected on many test cases presented here, the obtained results must be explained by the costs of the protocols internal operations, that seems to be more expensive using MrsP semaphores.

6.13 Test Case 10

This test verifies if the implementation of MrsP on RTEMS presents a Deadlock Prevention Mechanism, since, a MrsP resource must follow a strict irreflexive and partial order[9], preventing potential deadlocks on the system.

[9] The obtention of nested resources must not be dealt between resources depending on each other. This circular dependencies between critical sections originate deadlocks on the system.

6.13.1 Test Description

Two processors are used, CPU0 and CPU1, in a partitioned scheduling environment, each one assigned to a scheduler instance, as shown in the following table.

Table 25 Test Case 10 Scheduler Assignments

SCHEDULER INSTANCE	PROCESSOR
SCHED_A	CPU0
SCHED_B	CPU1

Each scheduler is responsible for the management of one of the two tasks necessary to perform the verification of the property, the INIT task and TASK0.

Table 26 Test Case 10 Tasks Properties

TASK NAME	SCHEDULER	PRIORITY
INIT	SCHED_A	7
TASK0	SCHED_B	7

Finally, two resources are also used, MrsP and MrsP1, each one of them with different Ceiling Priorities for each scheduler, as defined in the following table.

Table 27 Test Case 10 Resource Definition

RESOURCE	SCHED_A PRIORITY CEILING	SCHED_B PRIORITY CEILING
MrsP	5	2
MrsP1	2	5

6.13.2 Test Behaviour

In first place, the INIT task obtains the MrsP resource, while TASK0 obtains MrsP1. Therefore, TASK0 also tries to obtain MrsP, in a nested way, which is expected to successfully happen, waiting for INIT to release it. Then, INIT tries to access the MrsP1 resource, which TASK0 holds. These operations would induce a deadlock, since TASK0 would be waiting for INIT to release the resource, while INIT would be also waiting for TASK0 to release its resource. However, the deadlock prevention mechanism from the protocol avoids this possibility, not allowing INIT to try to obtain MrsP, held by TASK2, that is already waiting for the INIT's one, as presented on the next diagram.

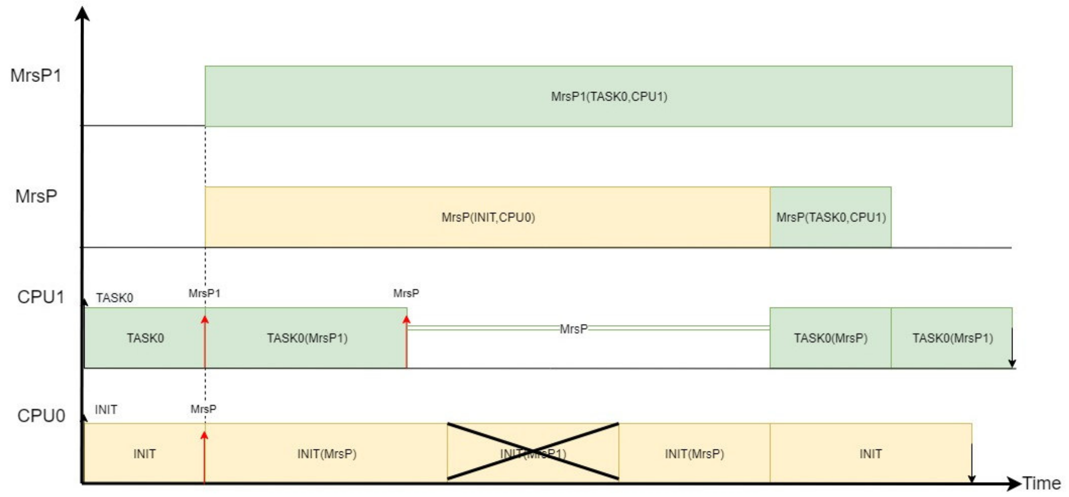


Figure 34 Test Case 10 Desired Behaviour

6.13.3 Results

This test passed successfully, since the deadlock detection mechanism went into action not allowing INIT to attempt the obtention of the resource, since this action would induce a deadlock. So, INIT must proceed is execution quitting the attempt to access the resource.

6.13.4 MrsP and OMIP Comparison

Using OMIP as resource sharing protocol, this test case demonstrated the same behaviour as using MrsP, detecting a potential deadlock and preventing it from happening.

In terms of timing results, for this test it can be considered relevant the measurement of the performance of deadlock prevention mechanism of each protocol. In this way, it is possible to assert which protocol possesses the most efficient mechanism for treating deadlocks.

According to the obtained data, as presented in the figure below, for MrsP the deadlock detention mechanism presents an execution time of 1 millisecond in average. On the other hand, using the OMIP protocol, the deadlock prevention mechanism presents a great data dispersion, in a range of approximately 6 milliseconds, presenting an average value of 3 milliseconds.

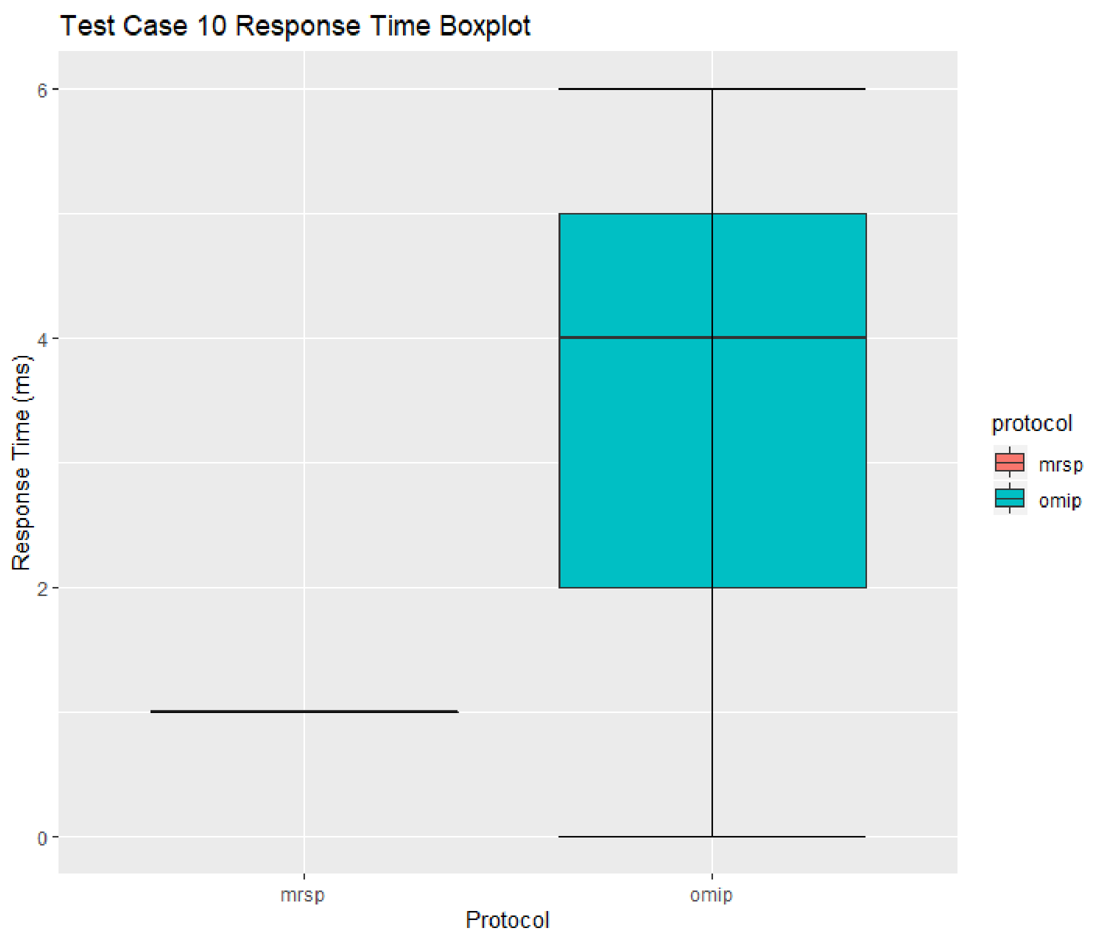


Figure 35 Response Time on Test Case 10

The observed vales may occur due to the differences related to the thread waiting queues on the protocols, that are more complex in case of OMIP, which may delay deadlock related operation, performed during operations associated with the queues, as mentioned on MrsP's implementation mapping, on chapter 4.

6.14 Test Case 11

Test Case 11 verifies if the helping mechanism is transitive[10]. The transitive helping mechanism is directly related to nested resources, for example if the helped task owns a resource that another task is trying to obtain in a nested way, although, since this task is not able to help, the helper must be a task attempting to access one of this last task's resources.

[10] A transitive helping mechanism happens when the helper is not attempting to access one of the helped task resources, however, since the helped task is not able to execute the resource, the helper is also prevented from making progress.

6.14.1 Test Description

This test executes on a partitioned scheduling system, composed by three processors, CPU0, CPU1 and CPU2, following the scheduler assignments presented on the following table.

Table 28 Test Case 11 Scheduler Assignments

SCHEDULER INSTANCE	PROCESSOR
SCHED_A	CPU0
SCHED_B	CPU1
SCHED_C	CPU2

In terms of tasks, this test case is composed by a set of 5 tasks, all of them necessary to perform the assertion of this property of the protocol.

Table 29 Test Case 11 Tasks Properties

TASK NAME	SCHEDULER	PRIORITY
INIT	SCHED_A	11
TASK0	SCHED_B	12
TASK1	SCHED_B	6
TASK2	SCHED_C	8
TASK3	SCHED_C	2

Finally, since this use case involves nested resources, two resources are used, each one with different Priority Ceilings defined on each scheduler.

Table 30 Test Case 11 Resource Definition

SEMAPHORE	SCHED_A PRIORITY	SCHED_B PRIORITY	SCHED_C PRIORITY
MrsP	7	7	4
MrsP1	7	4	7

6.14.2 Test Behaviour

Initially, while INIT is executing on CPU0, TASK0 becomes available and starts executing on its host processor, CPU1, taking ownership of the resource MrsP.

Later, TASK2 enters the system, on CPU2, starting its execution and obtaining the resource MrsP1 and, after that, INIT also tries to access it.

Meanwhile, TASK2 also tries to obtain MrsP, held by TASK0, being placed on the waiting queue, until the resource is released, in order to proceed with its own execution.

Moreover, TASK1 and TASK3 also become ready to execute, on CPU1 and on CPU2, respectively.

Since TASK1 presents higher priority than TASK0, a preemption must occur, allowing the execution of TASK1 and preempting TASK0 from executing. Similarly, TASK3, on CPU2, also has greater priority than TASK2, so, in the same way, a preemption is also performed, in order to let the higher priority task execution proceed.

In this way, TASK0 and TASK2 are unable to operate, waiting for TASK1 and TASK3 to finish their execution. Although, the INIT task is attempting to obtain the resource MrsP1, held by TASK2, which allows the possibility of using RTEMS helping mechanism.

If the INIT task tries to help TASK2 nothing will happen, since TASK2 is waiting for TASK0 to release the resource. In this way, the INIT task must perform a transitive help, allowing the migration of TASK0 to CPU0 and letting it continue executing, as described on the following image.

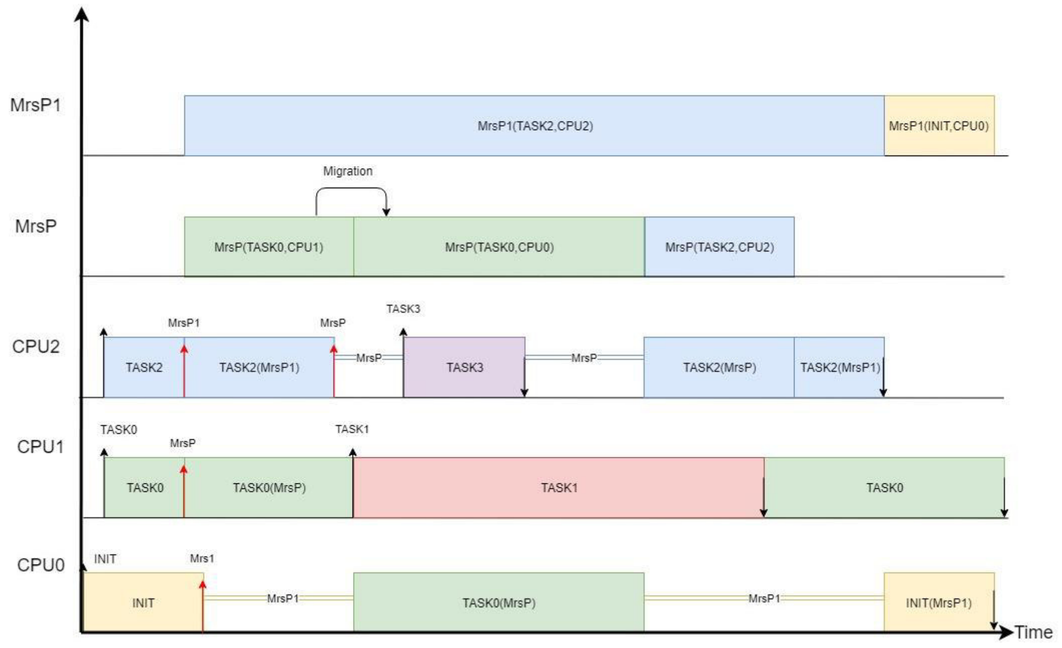


Figure 36 Test Case 11 Desired Behaviour

6.14.3 Results

This property has been successfully verified in this test case.

Although INIT did not attempt to obtain any of TASK0's resources, as it was indirectly waiting for it, it was able to help it by avoiding unnecessary waiting times that would affect system performance, in terms of resource usage.

6.14.4 MrsP and OMIP Comparison

In this sample when opting for OMIP semaphores, the helping mechanism behaves in the same way as using MrsP, presenting transitive help, which can be explained by the fact that the helping mechanism on RTEMS is implemented in the scheduler, being shared among both protocols.

Regarding the timing analysis performed, MrsP presents a mean execution time slightly lower than OMIP, about 87.26 milliseconds and 87.71 milliseconds respectively. Although, analysing the following graph it cannot be affirmed that MrsP presents better results than OMIP.

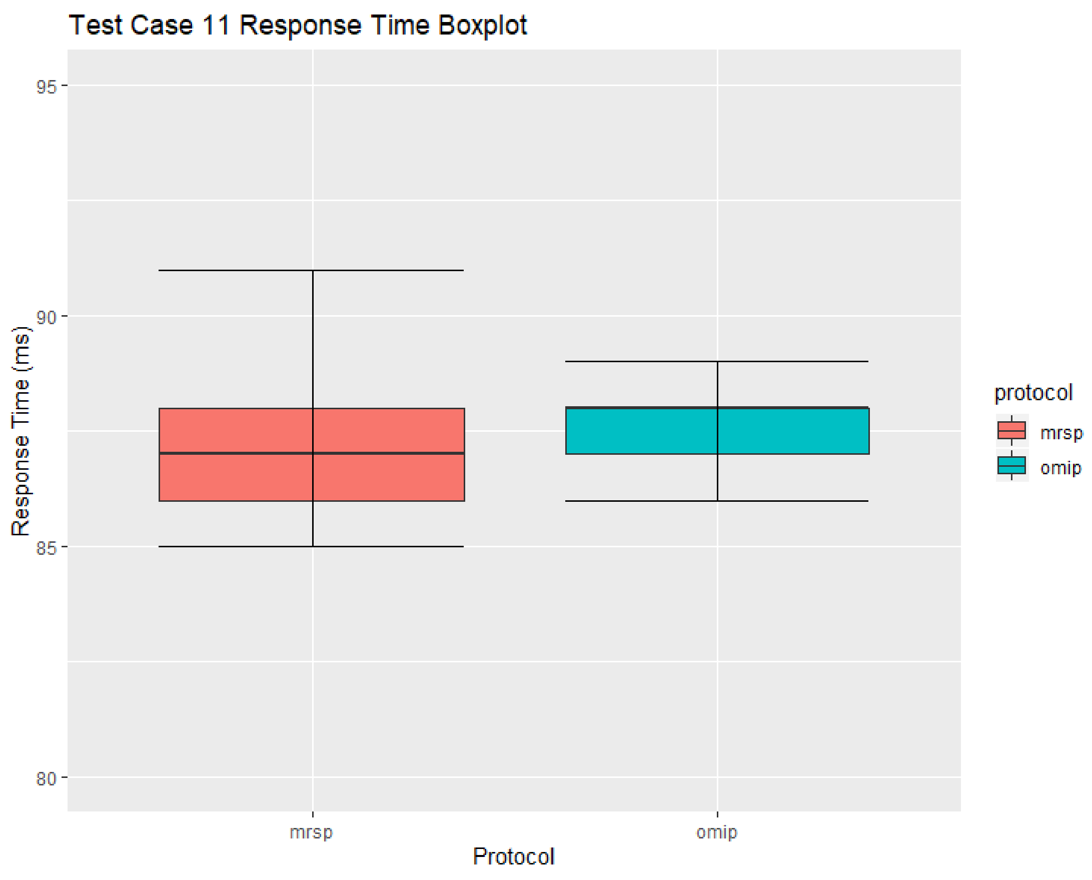


Figure 37 Response Time on Test Case 11

It is possible to verify that using MrsP semaphores, on worst case scenario, the system presents a more considerable higher execution time, while using OMIP as Resource Sharing Protocol, the obtained values are less dispersed, so, although it is possible to obtain better results with MrsP, on this test case, it is not granted.

6.15 Test Case 12

This test is based on Test Case 11, using the transitive helping mechanism, although it verifies if the transitive helping mechanism performs successfully, when the helper belongs to the same host scheduler instance as the helped task.

6.15.1 Test Description

This sample takes advantage of the same resource and tasks of Test Case 11, although the INIT task is only used to create and initialize resources and tasks. So, the definition of the system and its components is the same as the last test case.

6.15.2 Test Behaviour

In first place, TASK0 and TASK2 become ready to execute, on CPU1 and CPU2 respectively. Since the processors are free, both tasks are dispatched and TASK0 obtains MrsP, while TASK2 obtains the other resource, MrsP1 and later it also attempts to access MrsP, having to wait, actively, until TASK0 releases the resource.

Meanwhile, TASK1 enters on CPU1, currently held by TASK0 to execute. Although, because TASK1 presents higher priority than TASK0, a preemption is performed and TASK1 starts executing, attempting to access the MrsP1, currently held and being executed on CPU2, by TASK2, that is attempting to obtain the MrsP resource.

In this case, it is expected that TASK0 migrates to TASK2's scheduler instance, since it is currently unable to make progresses. Although, a higher priority task, TASK3, also arrives to the system on that processor, CPU2, preempting TASK2 and starting its own execution

In this way, TASK1, executing on CPU1, is waiting for TASK2 to release the resource MrsP1 and TASK2, unable to execute on its home scheduler instance, is waiting for TASK0 to release MrsP, unable to execute on CPU1, because of TASK1.

It expected from the helping protocol to operate on the host scheduler of the task needing help, TASK0, so, it should be helped by TASK1, inheriting its current priority and executing on CPU1, in order to rush the progress of the system. Else, the execution of all the tasks related to the resources MrsP and MrsP1, on this test case, are depending on the execution of TASK3, on CPU2, which may cause several delays to the shared resources execution.

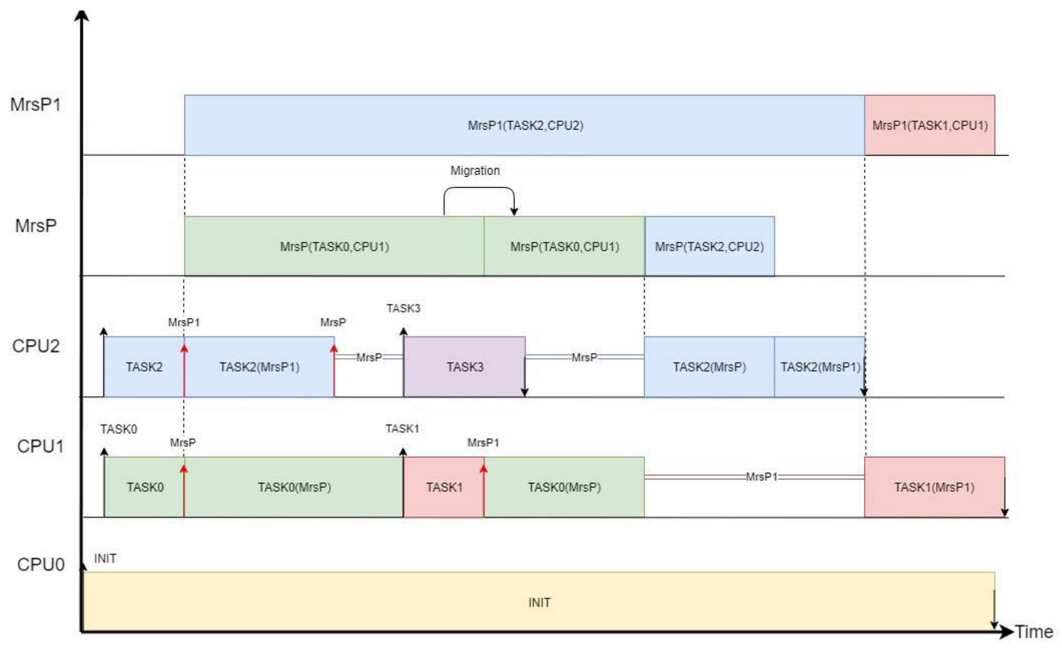


Figure 38 Test Case 12 Desired Behaviour

6.15.3 Results

This property is not successfully granted using MrsP semaphores. On this test case, TASK1 cannot help TASK0, which belongs to the same processor.

Consequently, only when TASK3 is finished, TASK0 may be able to be helped by TASK2, on CPU2, in order to enable the execution of it and then from TASK1.

6.15.4 MrsP and OMIP Comparison

Contrarily to MrsP, OMIP presents a way to overcome this problem. In this test case, OMIP presents a migratory priority inheritance mechanism, which means that, the helped task is able to inherit the priority of the helped, in this case, TASK0 inherits it from TASK1.

In terms of timing analysis, it is expected that using MrsP, since it doesn't guarantee the support of a helping mechanism on the helped task host scheduler, that the execution of the sample is considerably higher than the execution time when using OMIP, as resource sharing protocol.

The results are presented on the following chart.

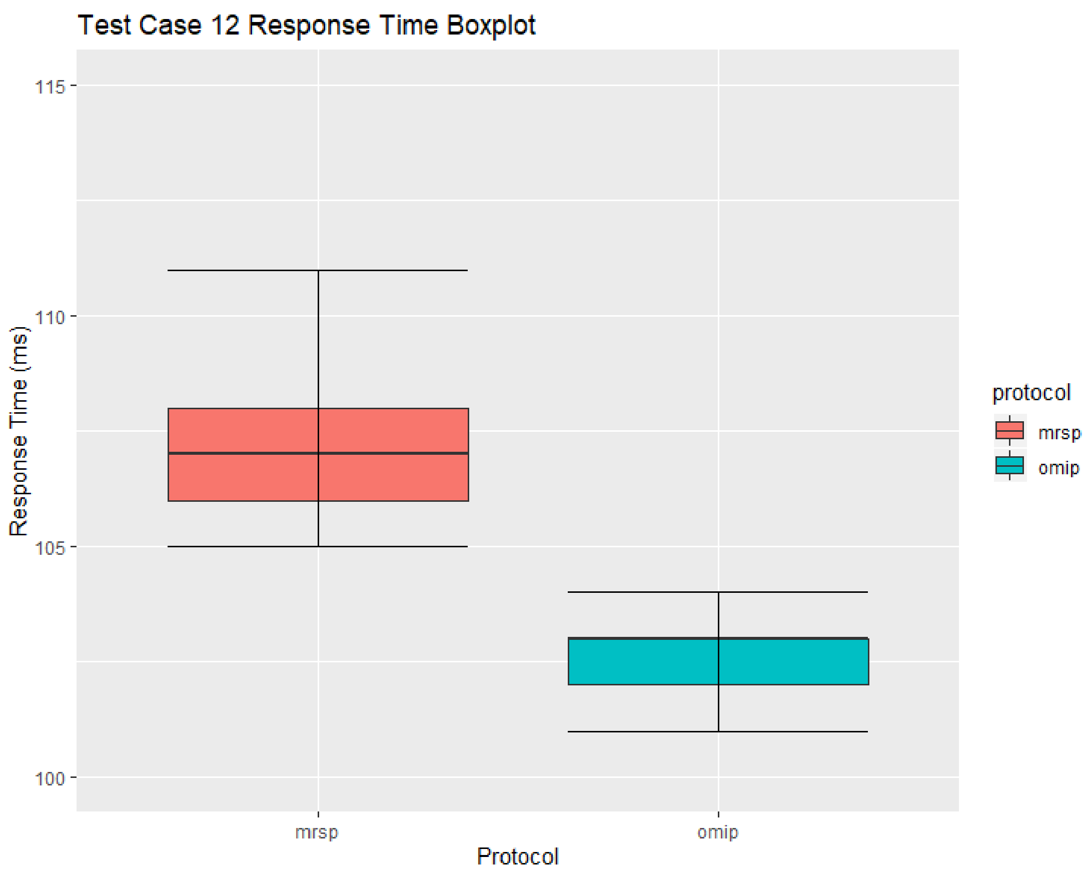


Figure 39 Response Time on Test Case 12

Therefore, it can be seen from the results presented in the graph above that, due to the fact that MrsP in RTEMS does not comply with this functionality, it may suffer a significant overhead, since while using OMIP, this sample executes on 102 milliseconds, on average, and using MrsP it takes 107 milliseconds, it should be noted that this time is completely dependent on the execution time of TASK3, because if it were longer, the execution time for other tasks would grow abruptly, whereas using OMIP , that would not happen, at least so violently.

7 Instrumentation Analysis

The set of tests presented in the previous section of the report, chapter 6, raises some questions about the performance of the RTEMS synchronization protocols used in this study. In fact, the protocol under study, MrsP, in comparison to OMIP, sometimes show differences at the level of timing performance. Thus, there is the possibility that the internal operations of the protocols are responsible for it. The goal of this chapter is to test this hypothesis.

In MrsP, since it is based on the ceiling priority protocol, all semaphore's obtain and release operations imply a priority change of the task obtaining the resource that is operating. On the obtaining, the ceiling priority, on the release operation, the priority is changed to the initial priority, which avoids possible preemptions.

Using priority inheritance, when obtaining or releasing a resource, as it happens in OMIP, there is no priority change, which may result on this operation to be briefer. However, when there is a task that executes a resource on a given processor, and another task of higher priority becomes ready and accesses the same resource on that same processor, the task that holds the resource is preempted and the higher priority task runs until it tries to get the resource. Then, its priority is inherited by the task that holds the resource and, thus, it may execute again.

This operation mode may decrease the performance of OMIP, something that does not happen on priority ceiling based protocols, since the ceiling priority is always higher than the priority of all tasks trying to allocate the resource, as mentioned earlier on this report.

Thus, it becomes completely pertinent to perform an instrumentation analysis, that is, to study the performance of the main internal operations of the protocols, in order to understand why they occur.

In this chapter, the tests presented were performed for both protocols under the same conditions, the response times of each operation were measured 500 times for each of them. During the measurement of the obtain and release operations, MrsP was subjected to some overhead, due to the measuring of the execution time of internal priority change related operations. However, this overhead was also induced on OMIP in order to produce reliable results.

7.1 Resource Obtention

The first operation analysed here is the obtention of a resource. As mentioned earlier, it is expected that the obtention takes longer for MrsP, since this operation induces priority changes to the resource allocating task.

In order to ascertain whether these differences are reasonable, the execution times related to priority change operations on MrsP have also been measured. In this way, it can be estimated if it is as efficient as OMIP, in terms of all the other operations performed during the obtention of a resource. The results of these measurements can be verified on the following image.

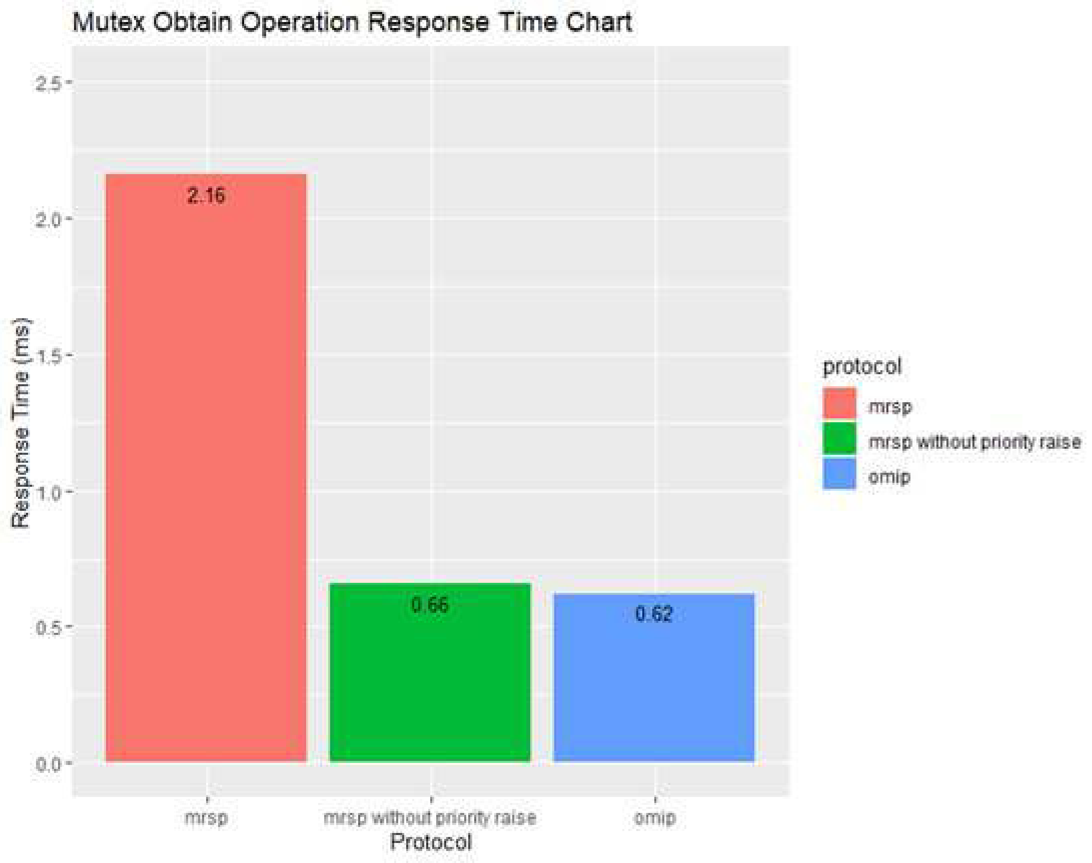


Figure 40 Obtain Operation Execution Time on MrsP and OMIP

In fact, the results show that the average execution time of MrsP presents higher values to achieve the resource obtention, 2.16 milliseconds. OMIP only presents 0.62 milliseconds, a considerably lower value. However, it is also found that, without the priority change operations, which is the cause of difference between the two protocols, MrsP presents response times of about 0.66 milliseconds, which is quite close to the time spent by OMIP. So, it can be considered that the obtained differences are acceptable, according to the functioning of the protocol, since those differences are related to the fundamental operations of MrsP**.**

7.2 Resource Release

Just as in obtaining a resource the priority of a task must be increased, in MrsP, in the release operation the priority must be changed to the initial priority of the task. Therefore, it is also relevant to analyse the release operation of a resource for both protocols, so as to once again gauge the differences between each protocol and look for a reason for their existence.

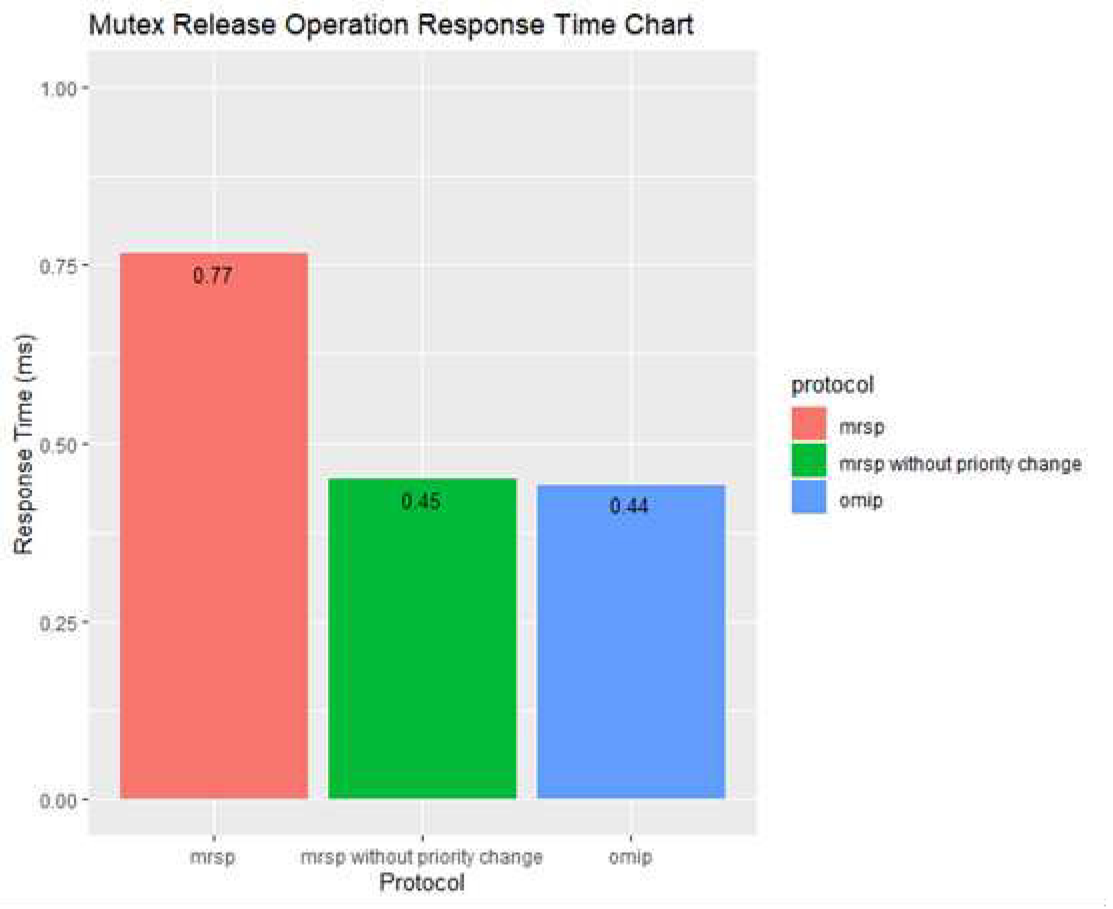


Figure 41 Release Operation Execution Time on MrsP and OMIP

As expected, the results obtained are relatively similar to the data for the previously studied operation, although priority related operations in MrsP are comparatively less expensive in terms of execution time, in relation to the obtain operation, since the task's priority was set to its initial state, which was already defined during its initialization.

7.3 Waiting Mechanism

Certain test cases performed in section 6 are directly related to the protocols waiting mechanism. On those tests, MrsP appears to be able to perform the waiting operations more efficiently than OMIP, i.e., with better response time results.

Since MrsP presents active waiting while in OMIP the waiting mechanism is suspension based, that is, mechanisms that despite having the same purpose, work differently because, as mentioned earlier, in the case of MrsP the waiting task is still executing, safeguarding the processor to avoid the execution of lower priority tasks, while in OMIP's case, the same task does not execute.

Using OMIP, some overhead related to the suspension operations in the execution of the task and its re-entry in the execution state may also exist. In this way, the timing performance of the protocol's waiting mechanism is presented next.

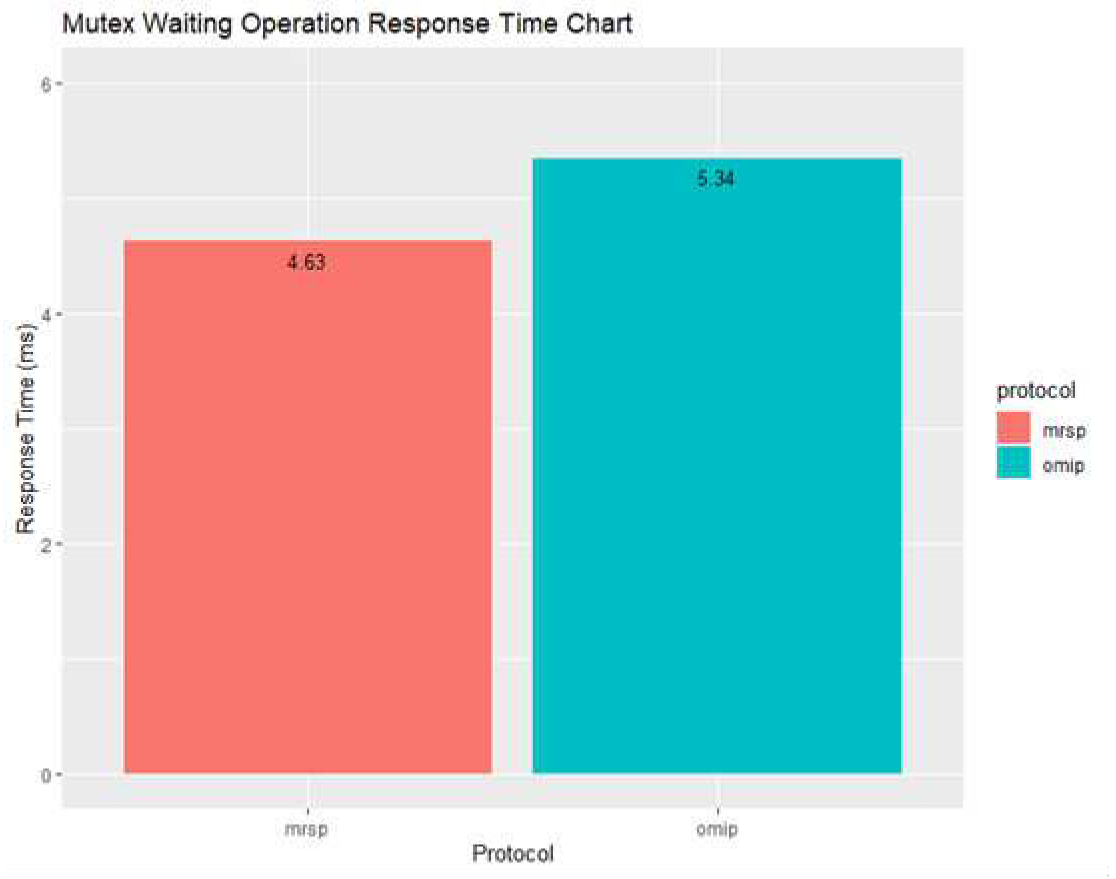


Figure 42 Waiting Mechanism Execution Time on MrsP and OMIP

The results presented in the table above show that MrsP, in terms of the waiting mechanism, can circumvent the usual dominance of the OMIP protocol. The response times of internal operations present average values with a difference of 1.31 milliseconds on average, a fact that is possibly explained due to the complex queue structures of OMIP, explained on chapter 3.

It is extremely important to mention that when a task waits to access a resource, it does not perform the direct resource obtention operation, which is faster when using OMIP. That is, in cases of greater stress to the system, which do not occur in the testbed presented here, MrsP will tend to be faster due to its efficient waiting mechanism.

7.4 Single Processor Instrumentation tests

This section of the report describes an analysis performed to the Priority Ceiling Protocol and the Priority Inheritance Protocol timing costs, in terms of their internal operations.

In this way, the results obtained from MrsP and OMIP and the obtained differences in the above experiments may be confirmed, since those differences must also exist when comparing their single processor foundation, PCP and PIP.

For the SMP protocols, the instrumentation tests are presented to their three main operations, the obtention of a resource, its release and the waiting procedure. Although, in this section, only the obtain and release are considered, since the waiting mechanism does not exist when using PCP on RTEMS.

Since when a task obtains a resource, using a PCP mutex its priority is immediately raised, other tasks that will obtain that resource, will only be able to perform such attempt when the resource is free. Because, only tasks with lower priority than the semaphore's priority ceiling are able to access it.

7.4.1 Resource Obtention

The results obtained from the multiprocessor protocols, show that MrsP, in fact takes longer than OMIP during the obtention of the result. However, it is also verified that, the obtained differences are clearly justified due to the priority raise present on the MrsP's internal operation. The obtained results are presented on the following chart.

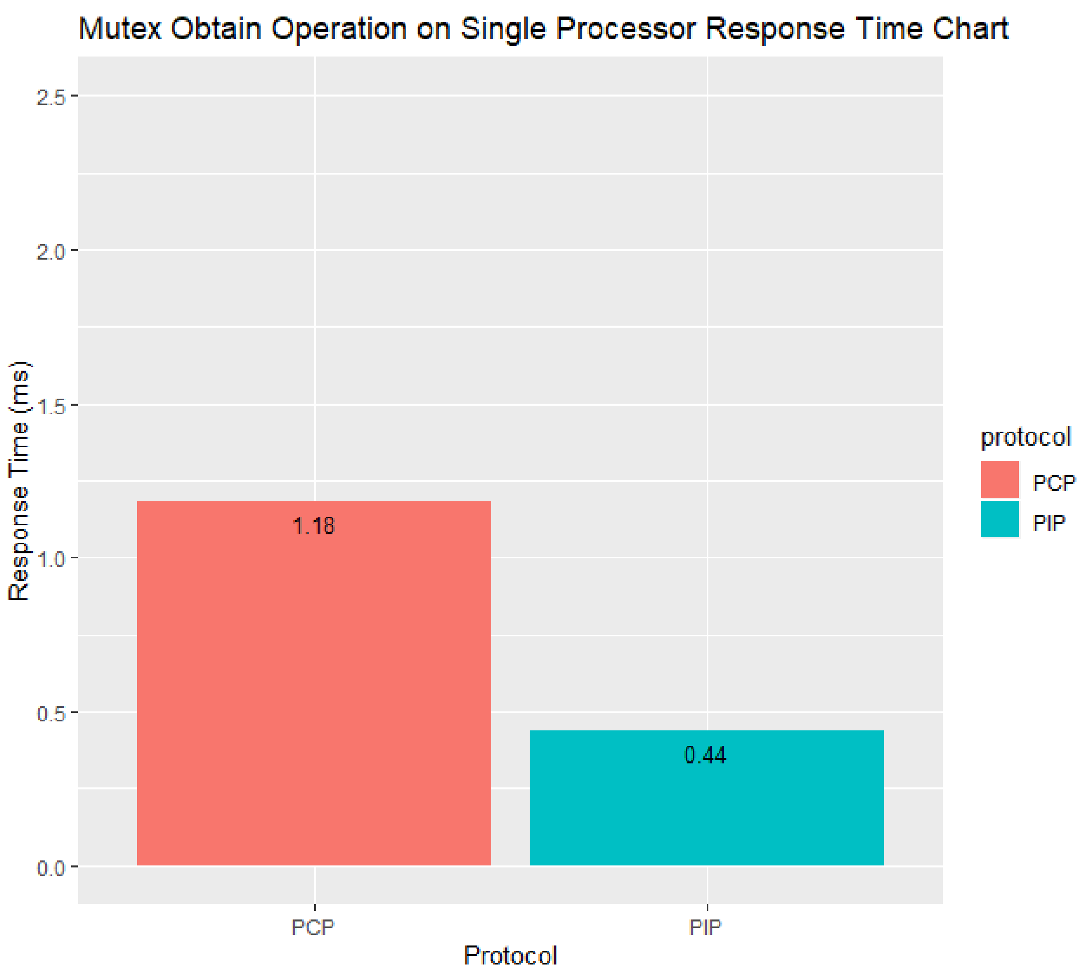


Figure 43 *Obtain Operation Execution Time on PCP and PIP*

In fact, as expected, it is possible to verify that the obtention of a single processor mutex, using the priority ceiling protocol, is expensive, when compared to the same operation using priority inheritance.

Although it is important to mention that, the relation of the cost of operation between MrsP and OMIP is not very similar to the relation presented by PCP and OMIP. MrsP and OMIP presents bigger differences than the ones seen in the single processor protocol, which means that probably the cost of the priority related operations is bigger in multiprocessor. This is expected as it is usually normal that multiprocessor operations are more complex, since they take into account concurrency and parallelism.

7.4.2 Resource Release

The operation of releasing a semaphore, presents slower execution times using MrsP than OMIP, once again, due to the priority change of the holder of the resource to its original priority level.

So, it is expected that PCP also performs the release of a resource slower than PIP, which was verified according to the results presented on the following image.

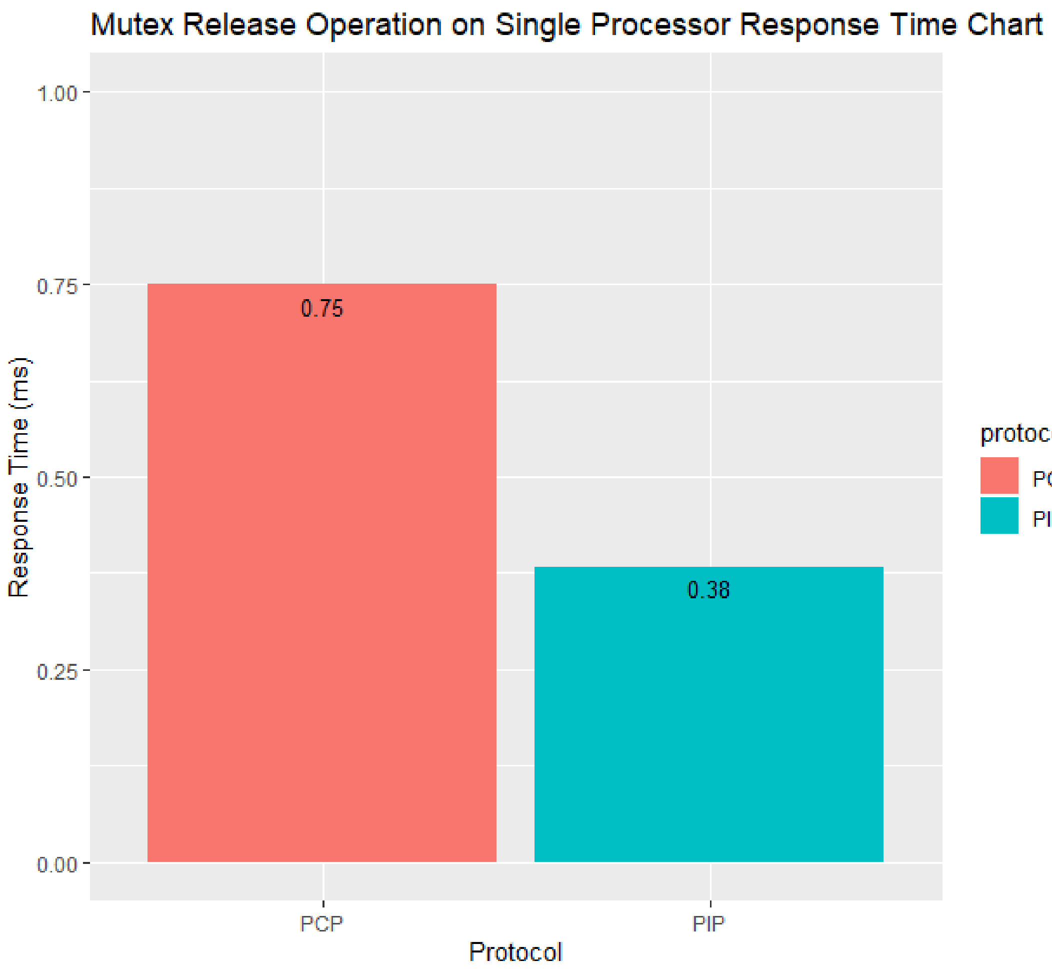


Figure 44 *Release Operation Execution Time on PCP and PIP*

It is important to mention that the obtained values of the release operation on single processor are quite like the ones obtained in multiprocessor, since MrsP presented, in average 0.77 milliseconds to release a resource and OMIP presented 0.44 milliseconds, values presented on the section 7.2 (see Figure 45).

These differences are quite small, so it is possible to affirm that in relation to the single processor protocols, PCP and PIP, the multiprocessor resource sharing protocols, on RTEMS, must be well optimized, at least in terms of resources releasing.

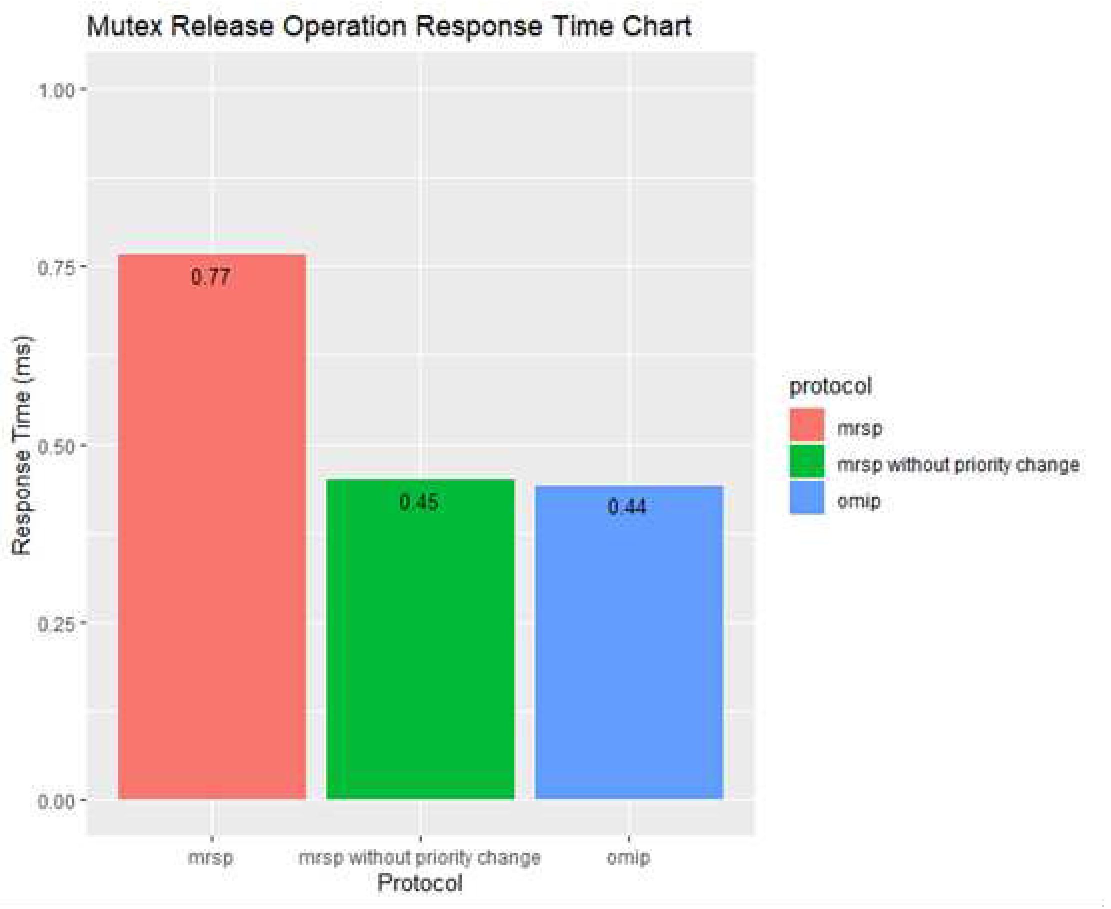


Figure 45 Release Operation Execution Time on MrsP and OMIP

7.5 Results Discussion

In conclusion, as expected, MrsP and OMIP present differences in terms of execution time of their internal operations. MrsP is more time consuming than OMIP, when it comes to the obtention and the releasing of a resource. As mentioned earlier, this difference of timing performance between the protocols raises suspicions that MrsP may have optimization problems. However, it is known that MrsP is a priority ceiling-based protocol whereas OMIP is based on priority inheritance. Thus, during these operations, while using MrsP, one of the fundamental operations of this protocol is that, during the obtention of a resource, the priority of the task accessing it has to be raised to the priority ceiling of the mutex and, during the release, the priority of the leaving task must be replaced to its original priority. While, using OMIP, during the release and obtention, by default, priority changes do not occur.

In this way, the time spent changing the priority of the task operating the resource, is also measured during the instrumentation tests and it is verified that, in fact the differences presented between MrsP and OMIP, in terms of timing performance of the obtention and release of a resource are clearly justified by the priority related operations performed by

MrsP. So, MrsP does not seem to present any optimization issue concerning these internal operations.

In fact, this behaviour of MrsP, on most stressful cases may induce more efficient timing cost when compared to OMIP.

Using MrsP, the holder of a resource always presents its priority raised the priority ceiling. Other tasks attempting to access it, on the same processor of the holding task, will only be able to execute when the holder releases the resource, since the priority ceiling is always higher than the priority of all the tasks trying to obtain it.

On the other hand, while using OMIP, any task of the system may access a resource without concerning with their priority, since there is no priority ceiling. If a task that is going to access resource enters the system on the processor where a lower priority task, that currently holds it, is executing. A preemption must occur, allowing the higher priority task's progress. In this way, the holder of the resource will be only able to execute again, when the higher priority task also attempts to access the resource, since a task executing a priority ceiling-based resource always inherits the priority of the higher priority task attempting to access it.

In conclusion, while on MrsP, the execution of the task that holds a resource is always guaranteed, in relation to other tasks that intend to use it, while using OMIP, this is not guaranteed, and there may be several delays in the execution of shared resources, depending on the system flow.

In addition to the operations described above, the results obtained from the tests performed in the previous chapter raise the possibility that the OMIP waiting mechanism is less efficient than the one from MrsP. Thus, measurements were also performed to the execution times of the waiting mechanism of both protocols and, in fact, MrsP presents a less time-consuming waiting mechanism.

This difference may exist due to the complex queue structure of OMIP and the fact that OMIP presents a suspension-based waiting mechanism, so the task does not execute while waiting. Hence, it requires more operations related to the waiting mechanism, while in MrsP a task waiting to access a resource remains executing, not having to be suspended and, when it is able to access the resource, dispatched to its own processor.

Finally, in order to assert the conclusions regarding OMIP and MrsP, instrumentation tests were also performed using PCP and PIP. This tests only cover the operations of obtention and release of a resource, since PCP does not present waiting mechanism. The obtained results from this analysis show that, in fact, PCP tends to be more expensive than PIP performing these operations. So, in this way, the obtained results regarding the multiprocessor protocols are justifiable.

8 Conclusions

In this chapter, the conclusions related to the project described in this report are presented. In the first place, a summary of the main objectives proposed and the respective results is elaborated. Then, the limitations and future work are presented as well. Finally, a general appreciation of the project itself is also given.

8.1 Accomplished Objectives

The project described in this report had three main objectives:

- Analyse the theoretical proposal of MrsP;
- Study the implementation of MrsP on RTEMS;
- Evaluate the correctness of the behaviour of MrsP's implementation on RTEMS.

The first objective of this work was the analysis of the theoretical aspects of MrsP, considering the original proposals of the protocol, in order to understand its behaviour and its desirable properties. This understanding served the purpose of providing the requirements that must be covered during the evaluation of MrsP on RTEMS. The second objective involved the study and analysis of the application of MrsP on RTEMS, as it can be seen in chapter 4. This study performed a mapping between the implementation of MrsP protocol on RTEMS, considering its fundamental operations, and the requirements analysed in the first objective. Finally, a testbed, described in chapter 6, composed of twelve test cases was developed, covering all the fundamental rules that define the correct behaviour of MrsP, based on the tasks previously performed. From all the test cases, the most relevant were also adapted to be performed using OMIP, the unique alternative to MrsP on RTEMS in terms of resource sharing management in multiprocessing. This allows for establishing a comparison between both protocols (MrsP and OMIP), not just in terms of behaviour, but also considering their timing performance. In the end and in order to test the hypothesis that internal mechanisms have an influence on the timing behaviour of the protocols, an instrumentation analysis focused on the internal operations of RTEMS resource sharing protocols was also performed. This set of test cases allows us to assert the differences presented by both protocols, in order to be possible to draw safe and conscious conclusions about it.

8.2 Overall Results

From the tests, it appears that, in general, the implementation of MrsP in RTEMS presents a behaviour that considers the requirements of its theoretical proposal. However, a flaw was found on this implementation which in specific cases may affect the system performance. This flaw exists in the helping mechanism in which it is expected that it should be able to operate between tasks belonging to the same scheduler instance, property tested on Test Case 12.

In order to evaluate the protocol's performance, a comparison with OMIP was done. The results obtained on all the test cases using global scheduling, where one scheduler instance is responsible for all the processors of the system, MrsP presents a desirable behaviour, meeting the proposed requirements. In the case of OMIP this cannot be affirmed, since, on RTEMS, when using global scheduling, this protocol degenerates to the priority inheritance protocol, presenting an unusual behaviour. On these test cases, MrsP presented a better timing performance than OMIP.

Results from the test cases that use a partitioned scheduling show that, in general, MrsP and OMIP present a similar behaviour, except on the test case 12, where MrsP fails, while OMIP presents a solution that overcomes this problem. In terms of timing performance, OMIP, by default, presents a faster response time than MrsP, except for those tests that are directly related to the tasks' waiting mechanism of the protocols, where MrsP is faster. Thus, these discrepancies raised suspicions that equivalent internal operations have different execution times. This hypothesis was tested using instrumentations tests regarding the two protocols.

The obtained results for the instrumentation tests show, in fact, the existence of differences, namely in the obtention, release operations and the waiting mechanism, which are in accordance with the tests present in the testbed. MrsP presents a faster waiting mechanism than OMIP, while OMIP is faster dealing with the obtain and release operations.

These differences could imply that MrsP has optimization issues. This protocol is based on priority ceiling, while OMIP is based on priority inheritance, which means that those two operations are different for both protocols. During the obtention and release of a resource, on MrsP, priority changes must be performed. On the first operation, the priority of the obtaining task must be raised to the priority ceiling and, on the second one, this priority must be restored to the task's original one. The cost of performing these operations was also measured, and, in fact, it was observed that MrsP, without operations related to changes in priorities, presents similar times to the execution of internal operations as OMIP. That is, MrsP has no optimization issues.

The fact that OMIP has worse waiting mechanism timing results also means that in more stressful use cases it may present worse time performance than MrsP, since whenever a task may not directly obtain a resource, faster operation using OMIP, it must wait until the resource is free, which is slower when using OMIP rather than MrsP. So, it will tend to be more expensive to perform the resource management.

Finally, it is not possible to assume that MrsP is completely implemented on RTEMS, since it presents a flaw on a specific desirable property. Although, generally it can be considered fit to perform resource management on this operating system, fulfilling all other requirements. The protocol also proves to be quite generic, presenting a very cohesive behaviour and somewhat independent of the type of system scheduling, in comparison with OMIP, which presents an especially poor behaviour on global scheduling.

Thus, it can be said that in fact MrsP is a safe choice as a resource sharing protocol on RTEMS, meeting most of its requirements and presenting a multidimensional behaviour, without being dependant on the systems features. Nevertheless, it is necessary to consider the refinement regarding the helping mechanism, so that it can be considered completely implemented with success.

8.3 Limitations and Future work

The greatest limitation of this work was the impossibility of executing the developed tests in a real board using of RTEMS. Although it is still possible to perform the desired relative comparisons correctly, between OMIP and MrsP, using QEMU, with a real board the results would be more realistic from an absolute time point of view. Efforts have been made during this project to compile the RTEMS on a Raspberry Pi 2 with support for SMP multiprocessing. However, it was not possible to use RTEMS SMP properties, only uniprocessor ones.

In terms of future work, it is proposed an analysis and correction of the flaws found on the implementation of MrsP in RTEMS, since the protocol presents no guarantee of helping possibility between tasks belonging to the same scheduler instances, one of the desirable properties of the protocol. In addition, it would be also pertinent to produce a more detailed analysis of OMIP, which presents a very similar behaviour to MrsP, but with some relevant difference in terms of its internal operations. Therefore, it is suspected that these may also have an impact on the behaviour of the protocol.

Finally, it is important to mention that, the testbed produced here has been presented to the RTEMS community. Since the beginning, it was part of the project the contribution to the

RTEMS community and the sharing of the obtained results to the official RTEMS source tree test samples. At this time, the process of integrating the test cases developed in this report in the RTEMS source tree is already moving forward, although it is still in its early stages.

8.4 Final Appreciation

8.4.1 Student's opinion

In my opinion, this project was highly motivating, not only personally, but also career-wisely, being my introduction to the real-time systems an area that, recently has been raising my interest.

As soon as I knew of the possibility of carrying out a project related to this area, despite the scarce knowledge in it, I was immediately motivated to proceed with it. I became really interested in exploring it, having developed a broader understanding of real-time systems in general terms and also about low level software and the way how an operating system operates.

In addition, it was extremely interesting to experience the world of work in an organization such as CISTER, a research centre, since one of my interests is, possibly, to do some research work during my career.

Finally, related to my supervisor, he was always available and interested in following my progress during the project. We scheduled a set of meetings and always kept in touch, in order to ensure that my work progress with quality in a healthy way, surpassing all the possible barriers to success.

8.4.2 Work Planing

In this project, from the beginning, the objectives were soon defined, with a strategic point of view that would allow the production of relevant results and the introduction to areas related to the concepts discussed here.

During the life cycle of the project, all the tasks were well defined aligned with the purpose and goals of it. Moreover, they were designed to be interconnected and incremental , in order to successfully achieve the projected objectives.

9 References

[1] Alan Burns; Andy Wellings, “A Schedulability Compatible Multiprocessos Resource Sharing Protocol - MrsP,” in *25th Euromicro Conference on Real-Time Systems (ECRTS)*, Paris, 2013.

[2] B. Brandenburg, “A Fully Preemptive Multiprocessor Semaphore Protocol for Latency-Sensitive Real-Time Applications,” in *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS)*, Paris, 2013.

[3] Sebastiano Catellani; Luka Bonato; Sebastian Huber; Enrico Mezzetti, “Challenges in the Implementation of MrsP,” in *Ada-Europe International Conference on Reliable software Technologies*, Madrid, 2015.

[4] Jorge Garrido; Shuai Zhao; Alan Burns; Andy Wellings, “Supporting Nested Resources in MrsP,” in *Ada-Europe International Conference on Reliable Software Technologies*, Vienna, 2017.

[5] S. Zhao, J. Garrido, A. Burns and A. Wellings, “New schedulability analysis of MrsP,” in *IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Hsinchu, 2017.

[6] S. Huber, “Embedded with RTEMS - RTEMS SMP Pre-Qualification (ECSS),” 26 02 2019. [Online]. Available: https://devel.rtems.org/ticket/3701. [Accessed 15 05 2019].

[7] “R Project,” R-Foundation, [Online]. Available: https://www.r-project.org/. [Accessed 2019 07 23].

[8] B. Boem, Software Engineering Economics, California: TRW Defense Systems Group, 1983.

[9] K. Beck, M. Beedle, A. v. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland and D. Thomas, “Agile Manifesto,” 2001. [Online]. Available: https://agilemanifesto.org/. [Accessed 05 2019].

[10] “Scruminc.,” Scrim inc., [Online]. Available: https://www.scruminc.com/. [Accessed 05 2019].

[11] J. Roslöf, M. Säisä and K. Tiura, “Waterfall vs. Agile Project Management Methods in University-Industry Collaboration Projects,” CDIO, Turku, 2018.

[12] J. Calandrino, H. Leontyev, U. D. A. Block and J. Anderson, “"LITMUSRT: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers ",” in *Proceedings of the 27th IEEE Real-Time Systems Symposium*, Rio de Janeiro, Brazil, 2006.

[13] B. Brandenburg, “Scheduling and Locking in Multiprocessor Real-Time Operating Systems,” UNC Chapel Hill, North Carolina, USA, 2011.

[14] T. Baker, “A stack-based resource allocation policy for realtime processes,” in *IEEE Real-Time systems Symposium (RTSS)*, Florida, USA, 1990.

[15] R. Rajkumar, L. Sha and J. Lehoczky, “Real time syncronization for multiprocessors,” in *IEEE real/time systems Symposium*, Alabama, 1988.

[16] R. Rajkumar, Synchronization in Real-Time Systems: A Priority Inheritance Approach, Kluwer Academic Publishers, 1991.

[17] P. Gai, G. Lipari and M. D. Natale, “Minimizing memory utilization of real-time task sets in single and multi-processor system-on-a-chip,” in *IEEE Real-Time Systems Symposium*, London, 2001.

[18] A. Block, H. Leontyev, B. B. Brandenburg and H. H. Anderson, “A flexible real-time locking protocol for multiprocessors,” in *International Conference of Embedded and Real-Time computing systems and Applicatios (RTCSA)*, Daegu, 2007.

[19] H. T. Ken Sakamura, “A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels,” in *Proceedings 18th IEEE Real-Time Systems Symposium (RTSS)*, San Francisco, 1997.

[20] eCos community, Free Software Foundation, “eCosCentric,” [Online]. Available: http://www.ecoscentric.com/. [Accessed 29 04 2019].

[21] Mentor Graphics Corp. A Siemens Business, “Mentor,” [Online]. Available: https://www.mentor.com/embedded-software/nucleus/. [Accessed 29 04 2019].

[22] Wind River systems, “windriver,” [Online]. Available: https://www.windriver.com/. [Accessed 29 04 2019].

[23] Texas Instruments, “TI-RTOS: Real-Time Operating System (RTOS) for Microcontrollers (MCU),” [Online]. Available: http://www.ti.com/tool/TI-RTOS-MCU. [Accessed 29 04 2019].

[24] blackberry, “https://blackberry.qnx.com/en,” [Online]. Available: https://blackberry.qnx.com/en. [Accessed 29 04 2019].

[25] sysgo, “sysgo: embedding innovations,” [Online]. Available: https://www.sysgo.com/products/pikeos-hypervisor. [Accessed 2019].

[26] “Litmus RT,” [Online]. Available: https://www.litmus-rt.org/. [Accessed 29 04 2019].

[27] RTEMS, “RTEMS Real Time Operating System (RTOS),” RTEMS, 2014. [Online]. Available: https://www.rtems.org/. [Accessed 2019].

[28] “The Linux Kernel Archives,” The Linux Foundation, [Online]. Available: https://www.kernel.org/. [Accessed 29 04 2019].

[29] J. Shi, K.-H. Chen, S. Zhao, W.-H. Huang, J. Chen and A. Wellings, “Implementation and evaluation of multiprocessor resource synchronization protocol (MrsP) on litmus rt.,” in *In Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, Duprovnik, Croatia, 2017.

[30] “RTEMS Overview,” 2014. [Online]. Available: https://docs.rtems.org/branches/master/c-user/overview.html. [Accessed 2019 04 30].

[31] S. Zhao, A. Wellings, A. Burns and J. Garrido, “New Schedulability Analysis for MrsP,” in *IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA).*, Hsinchu, Taiwan, 2017.

[32] F. Quartier, “microeletronics.esa.int,” 20 04 2015. [Online]. Available: http://microelectronics.esa.int/gr740/RTEMS-SMP-FinalReport-SpaceBelEmbeddedBrainsUnivPadova.pdf. [Accessed 2019 08 19].

[33] J. Andersin, J. Robin and J. Kevin, “Efficient Object Sharing in Quantum-Based Real-Time Systems,” in *Proceedings 19th IEEE Real-Time Systems Symposium*, Madrid, 1998.

[34] A. Goel and G. Sharma, “Scheduling Shallanges in Operating System,” in *Bilingual International Conference on Information Technology: Yesterday, Today, and Tomorrow*, DESIDOC, DRDO, Delhi, 2015.

[35] arm, "arm Developer," [Online]. Available: https://developer.arm.com/docs/ddi0406/latest. [Accessed 13 09 2019].

[36] Freescale, "npx," [Online]. Available: https://www.nxp.com/docs/en/white-paper/POWRPCARCPRMRM.pdf. [Accessed 13 09 2019].

[37] "RISC-V," [Online]. Available: https://riscv.org/. [Accessed 13 09 2019].

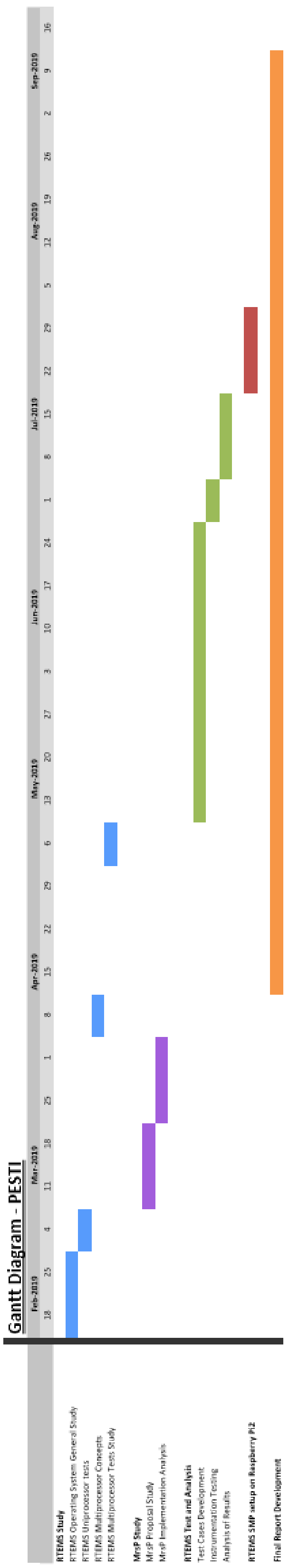
[38] Cobham, "Gaisler," [Online]. Available: https://www.gaisler.com/doc/sparcv8.pdf. [Accessed 13 09 2019].

[39] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," 1973, pp. 46-61.

[40] "RTEMS C User Manual - Chains," 10 07 2017. [Online]. Available: https://docs.rtems.org/releases/rtems-docs-4.11.2/c-user/chains.html. [Accessed 2019 08 13].

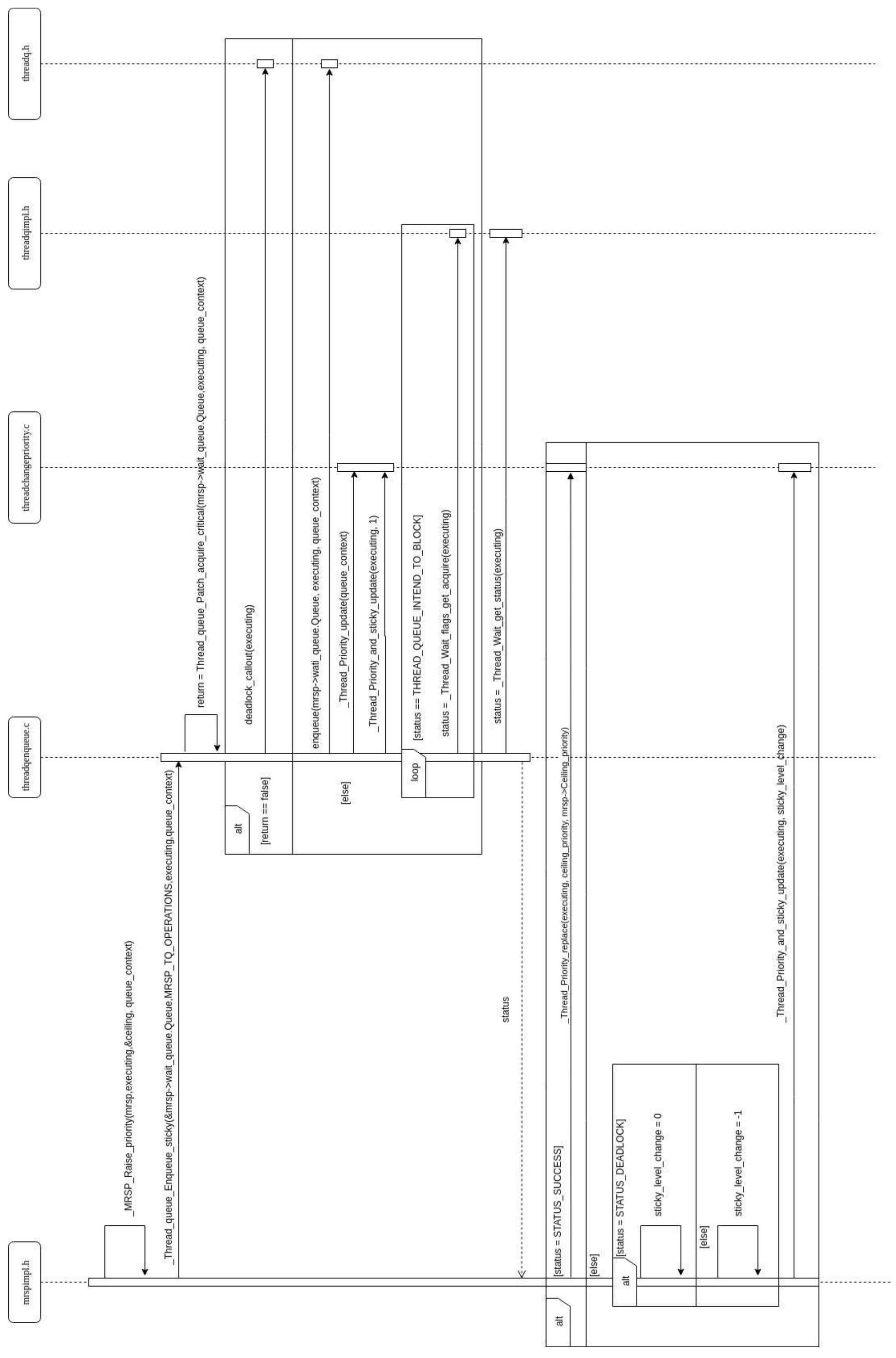
[41] "QEMU," [Online]. Available: https://www.qemu.org/. [Accessed 22 07 2019].

Appendix 1. Gantt Diagram



Appendix 2. MrsP’s Waiting Mechanism

The waiting mechanism, performed through the *_MRSP_Wait_for_ownership()* function is meant to deal the task’s attempt to access the resource when it is already held by another one, a process described on the following diagram.



threadq.h
threadqimpl.h
threadchangepriority.c
threadqenqueue.c
mrspimpl.h
_MRSP_Raise_priority(mrsp,executing,&ceiling, queue_context)
_Thread_queue_Enqueue_sticky(&mrsp->wait_queue.Queue,MRSP_TQ_OPERATIONS,executing,queue_context)
return = Thread_queue_Patch_acquire_critical(mrsp->wait_queue.Queue,executing, queue_context)
alt
[return == false]
deadlock_callout(executing)
[else]
enqueue(mrsp->wati_queue.Queue, executing, queue_context)
_Thread_Priority_update(queue_context)
_Thread_Priority_and_sticky_update(executing, 1)
loop
[status == THREAD_QUEUE_INTEND_TO_BLOCK]
status = _Thread_Wait_flags_get_acquire(executing)
status = _Thread_Wait_get_status(executing)
status
alt
[status = STATUS_SUCCESS]
_Thread_Priority_replace(executing, ceiling_priority, mrsp->Ceiling_priority)
[else]
alt
[status = STATUS_DEADLOCK]
sticky_level_change = 0
[else]
sticky_level_change = -1
_Thread_Priority_and_sticky_update(executing, sticky_level_change)

Figure 46 Wait for the Ownership of a MrsP Semaphore on RTEMS

Since, on MrsP, when a task is attempting to access an already held resource, it must busy wait, executing until it earns the ownership of the resource.

The arguments passed to this function are the following:

- *mrsp*: the main MrsP structure
- *executing*: the executing task
- *context*: the queue context
- *ceiling*: the ceiling priority

The first step of the waiting mechanism is the definition of the priority raise of the task to the priority ceiling, using the MrsP's function *_MRSP_Raise_priority()*.

After that, the task must be placed on the waiting FIFO queue, through the *_Thread_queue_Enqueue_sticky()* function, that is responsible for managing the enqueuing and waiting process of the task.

The possibility of entrance of the task must be verified, in order to avoid deadlocks, using the *Thread_queue_Patch_acquire_critical()* function. If the return value is true, the waiting operation may proceed. Else, a deadlock may occur, so the function *deadlock_callout()* must be called to register it and the operation must stop immediately.

On the other hand, if the operation is performed successfully, the thread may be enqueue with the *enqueue()* function. The priority of the task must also be updated to the priority defined before, using the *_Thread_Priority_update()* to create a new priority node on the queue context corresponding to the new priority of the task. It is updated using the *_Thread_Priority_and_sticky_update()*, as mentioned before.

The active waiting mechanism is based on a loop, controlled by a flag, and the task must perform the spin until the flag is different than THREAD_QUEUE_INTEND_TO_BLOCK. That status flag is acquired using the function *_Thread_Wait_flag_get_acquire()*, implemented on the queue. Finally, the *_Thread_Wait_get_status()* function returns the final status of the waiting process, informing if it was successful or not.

In the end, if the waiting procedure was successful, it means that the task is now able to access the resource and claim its ownership, using the *_Thread_Replace_priority()* function, to replace the ceiling priority of the semaphore as the ceiling priority of the task that is elected as owner, and the task may take advantage of the mutex.

On another way, if the return from the waiting procedure was unsuccessful, i.e., the returned flag is different than *STATUS_SUCCESS*, all the operations performed before must be reverted.

The obtention attempt failed and the task must have its priority and sticky level updated to the original ones. In first place a verification is done, in order to assert the sticky level change that must be done, since, if the failing cause is a deadlock detention, the sticky level of the task is not even incremented, so it must not be updated remaining zero. For any other cause of failure, this value must be decremented by the same value as the incremented one. Finally, with the help of the *_Thread_Priority_and_sticky_change()* function, these values may be successfully reset.

Appendix 3. Implementation of Test Case 8

```c
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <sys/param.h>

#include <stdio.h>
#include <inttypes.h>

#include <rtems.h>
#include <rtems/libcsupport.h>
#include <rtems/score/schedulersmpimpl.h>
#include <rtems/score/smpbarrier.h>
#include <rtems/score/smplock.h>

#include "tmacros.h"

const char rtems_test_name[] = "SMPPESTI 8";

#define CPU_COUNT 3

#define MRSP_COUNT 4

static int flag;

typedef struct
{
    rtems_id main_task_id;
    rtems_id scheduler_ids[CPU_COUNT];
    rtems_id mrsp_id;
    rtems_id task_id[3];
} test_context;

static test_context test_instance;

static void Init(rtems_task_argument arg)
{
    TEST_BEGIN();
    test_context *ctx = &test_instance;
    uint32_t cpu_count = rtems_get_processor_count();

    rtems_status_code sc = rtems_semaphore_create(
        rtems_build_name('M', 'R', 'S', 'P'),
        1,
        RTEMS_MULTIPROCESSOR_RESOURCE_SHARING | RTEMS_BINARY_SEMAPHORE,
        9,
        &ctx->mrsp_id);
    rtems_test_assert(sc == RTEMS_SUCCESSFUL);

    printf("TASKS PRIORITY = 4\n");
    printf("RESOURCE PRIORITY = 9\n");

    printf("OBTAINING THE RESOURCE\n");
    sc = rtems_semaphore_obtain(ctx->mrsp_id, RTEMS_WAIT, RTEMS_NO_TIMEOUT);
    printf("EXPECTED RESULT = 19 (INVALID PRIORITY).\nOBTAINED = %d\n", sc);
    rtems_test_assert(sc == 19);

    TEST_END();
    rtems_test_exit(0);
}

#define CONFIGURE_MICROSECONDS_PER_TICK 1000

#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_SIMPLE_CONSOLE_DRIVER

#define CONFIGURE_MAXIMUM_TASKS 1
#define CONFIGURE_MAXIMUM_SEMAPHORES 1
#define CONFIGURE_MAXIMUM_MRSP_SEMAPHORES 1

#define CONFIGURE_MAXIMUM_PROCESSORS CPU_COUNT

#define CONFIGURE_INIT_TASK_NAME rtems_build_name('M', 'A', 'I', 'N')
#define CONFIGURE_INIT_TASK_PRIORITY 4

#define CONFIGURE_RTEMS_INIT_TASKS_TABLE

#define CONFIGURE_INIT

#include <rtems/confdefs.h>
```