technische universität
dortmund

Bachelor Thesis

# Analyzing and Implementing Resource Synchronization Protocols on RTEMS in Multi-core Systems

Nadzeya Liabiodka

Juni, 2017

Advisors:
Prof. Dr. Jian-Jia Chen
M.Sc. Kuan-Hsun Chen

Faculty of Computer Science XII
Embedded Systems (LS12)
Technische Universität Dortmund
http://ls12-www.cs.tu-dortmund.de

## Abstract

In the real time system, several tasks may request a shared resource currently. However such resource accesses are required to be mutual exclusive. One sound way is to use resource synchronization protocols to prevent from race condition without deadlock and starvation . Several resource synchronization protocols are proposed for many-core systems. Generally, they are developed from the basic protocols for uniprocessors. The proposed approaches on multiprocessor inherit the proved advantages of synchronization protocols on uniprocessors.

In this thesis we concentrate on four specific multiprocessor resource sychronization protocols: the Multiprocessor Priority Ceiling Protocol (MPCP), the Distributed Priority Ceiling Protocol (DPCP), the Distributed Non-Preemptive Protocol (DNPP), and the Multiprocessor Resource Synchronization Protocol (MrsP). Most of them use a generalization of the Priority Ceiling Protocol (PCP) on the multicore systems. However these protocols provide totally different mechanisms to synchronize a resources access in practice.

In this thesis, we implement three resource synchronization protocols, for multi-core systems on a real-time operating system called the Real-Time Executive for Multiprocessor Systems (RTEMS). As far as we know, it has only some available resource synchronization protocols: the Priority Inheritance Protocol (PIP), the Priority Ceiling Protocol (PCP), and the Multiprocessor Resource Sharing Protocol (MrsP). Towards our perspective, we consider to use Symmetric Multiprocessing (SMP) RTEMS in this thesis. We implement our own SMP schedulers to handle the executions of tasks for each protocols, because the schedulers which are already supported in RTEMS don't provide the appropriate mechanisms for new protocols. Finally, we verify our implementation with the Application Programming Interface (API) RTEMS.

**Keywords**: RTEMS, real-time, MPCP, DPCP, resource synchronization.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Using multiprocessors and many-core systems has dramatically increased in the last few years due to the several advantages in compare with uniprocessor systems. In particular, many-core systems reduce power consumption due to using of multiple processing units with a lower frequency and provide high computational capability. The good scheduling algorithm is one of the most important factors which can guarantee the timeliness of the system. Therefore, researches on multiprocessor real-time scheduling has increased the last few years. However, the transition to multicores makes necessary to devise the new solutions which are more efficient than the paradigms for the uniprocessors. After researching for several years, there are still several issues due to the task migrations and the necessity to part the execution of the tasks between processors.

In this thesis we focus on real time operating system (RTOS) with multiprocessor. The difference between the RTOS and the general-computing operating system is the predictability of the timing behavior. The implementation in the real-time applications is generally a complicated task. The developers need to follow not only the application logic but also the fulfillment such temporal requirements as deadlines [10]. We will discuss it in detail in the next chapters.

In this thesis, the resource synchronization protocols have been implemented and analyzed on Real-Time Executive for Multiprocessor Systems (RTEMS). There are several protocols requirements which we have to consider while implementing. These requirements specify the blocked tasks behavior in case the resource is occupied by other tasks (e.g. suspending, spinning), the tasks migrations between processors, order to a resources access (e.g. FIFO, priority based), and the allocation of resources on the processors.

The main goal of this thesis is to implement the MPCP, the DPCP, and the DNPP on the latest version of RTEMS [9] and finally to verify them with the appropriate applications. We will discuss the details in the next chapters.

## 1.1 Thesis Structure

The thesis is divided into 6 chapters. In chapter 1 we will introduce the basics of resource sharing and the most known uniprocessor resource synchronization protocols, which principles are used for the implementation in the multiprocessor. Chapter 2 describes multiprocessor resource synchronization protocols, in particular the Multiprocessor resource sharing

Protocol (MrsP), the Distributed Priority Ceiling Priority Protocol (DPCP), the Multiprocessor Priority Ceiling Protocol (MPCP), and the Distributed Non-Preemptive Protocol (DNPP). The MrsP is already implemented in RTEMS and will be used for the analyzing of the multiprocessor resource synchronization protocols development on the same RTOS. The scheduling process and the structure of RTEMS are described in Chapter 3. Chapters 4 and 5 focus on the implementation and verification of the new protocols. Finally, the results of this thesis and the future work are summarized in Chapter 6.

## 1.2 Basics of Resource Sharing

At first, we introduce some basic knowledge about resource synchronization. The resources imply the files, data structures or a set of variables that are used for tasks to complete the execution [14]. In concurrent program running, where several tasks need to access the same resource, some program sections must be protected. For enforcing exclusive access, this piece of code is called critical section [14]. The protocols define the way how the tasks share the resources. The task cannot use the resource while any other tasks hold it. In this case, the task is blocked and waits until the resource is free to use.

One of the most known technique is semaphore which is used to protect the execution in the critical section. It uses only two primitives which called signal and wait. When we use the semaphore shown in Figure 1.1 [12] two tasks $t_1$ and $t_2$ access an exclusive resource $R_k$ via the semaphore $S_k$. Each critical section must start with a wait ($S_k$) primitive and close with a signal ($S_k$) primitive [12]. Therefore, each exclusive resource must be used

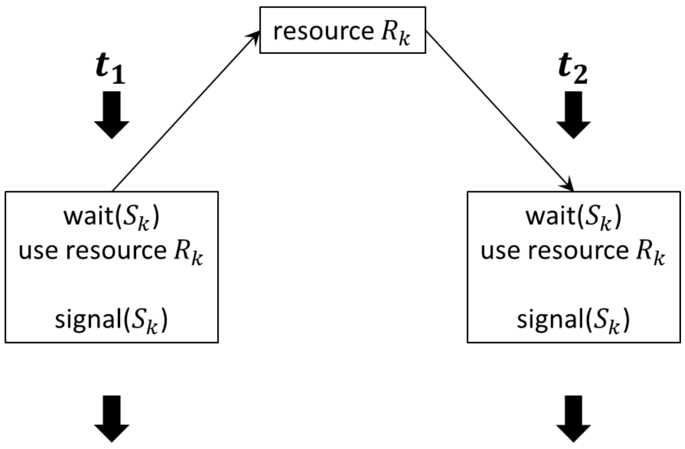


**Figure 1.1:** The semaphores structure [12]

with its own semaphore during the critical section. The tasks which are blocked on the same resource are ordered in the waiting queue. When a task executes a wait primitive on a semaphore which is already locked, this task moves to the waiting queue and waits until the task holding this resource executes a signal primitive. It means that the resource is available and the first task in the queue can use the resource. There are several paradigms in the scheduling algorithms which manage the order of the task's execution. We will discuss some of them later in Chapter 2.

There are several problems when tasks try to use the same resource concurrently. Some of them can be a reason for the increasing of the systems response time or can cause a system crash in the case of deadline misses which are forbidden in some real-time systems [12]. If the resource synchronization is not carefully handled, the most important issue which may lead to deadline misses is priority inversion.The priority inversion happens when the lower priority job blocks a higher priority job. The priority inversion phenomenon is presented in Figure 1.2 when the task $T_1$ with the highest priority blocks during the time interval [3, 9].

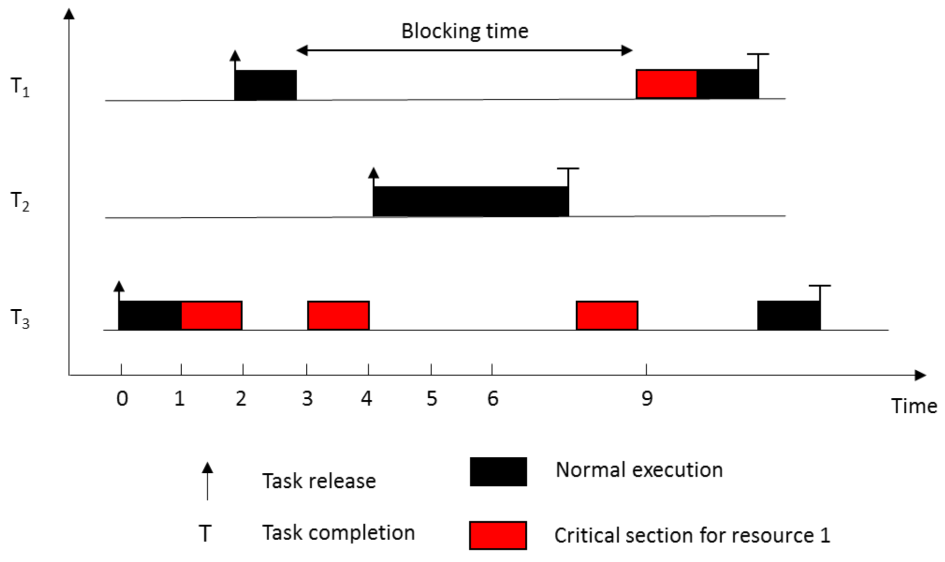


**Figure 1.2:** Priority inversion phenomenon

Priority inversion phenomenon destroys the predictability of the RTOS. However when we analyze the schedulability of the system, the worst case is when the middle task has extremely long execution time, the highest priority task must miss it's deadline. However with a good synchronization protocols, this crucial situation can be avoided. In the literature [12, 14] can be found two types of priority inversion. In the first one, called direct blocking, the blocking time of the high priority task is the equal to the time that needed for the execution of critical section by low priority task. This type of priority inversion is not avoidable. The second type of priority inversion, called indirect blocking, happens when other medium level priority task, which doesn't use an exclusive resource blocks the low priority task by its critical section. So in this case, the blocking time of the high priority task doesn't directly depend only on the execution time of critical section by low priority task but also on the worst-case execution time of medium level task. This type of priority inversion can cause the critical problems with the response time [12].

There are several approaches that can help to decrease the influence of the priority inversion on the systems. Some of them use the raising of the jobs priority by entering of the critical section. As the next we review into the resource synchronization protocols which are used in uniprocessor. Then we will introduce the multiprocessor resource synchronization protocols which use the basic principles of the uniprocessor protocols.

## 1.3 Resource Synchronization Protocols in Uniprocessors

In this chapter we study the most known uniprocessor resource synchronization protocols which were developed to solve the problem discussed above. The first solution that can help to avoid the indirect priority inversion is the disallowing of the preemption when the task enters a critical section.

This solution proposes a protocol called as Non-Preemptive Protocol (NPP). The priority of task that enters the critical section should be increased to the highest priority level in the system. NPP solves the problem of the indirect priority inversion but at the same time it can be cause unnecessary blocking [12]. For example, we have three tasks and only two with the lower original priority have to share one exclusive resource. In this case, the task with the highest priority must be blocked by other tasks in the critical sections. To guarantee the correct execution of the non-preemption part we need to increase the priority of the tasks which enter the critical section to the highest priority level in the system.

The second protocol that solves the problem of the indirect priority inversion in uniprocessor is the Priority Inheritance Protocol (PIP). This protocol changes the priority only of those tasks that can generate blocking. For example, if the lower priority task blocks the task with the higher priority, it temporarily raises to this level priority. The PIP bounds the priority inversion phenomenon, but it can cause a chained blocking, the situation when a higher priority task is blocked twice by a lower priority tasks by sharing two resources [18].

The third protocol that can protect from the chained blocking and bounds the priority inversion problem is the Priority Ceiling Protocol (PCP). The concept of the PCP is widely used in the multiprocessor to access to an exclusive resource. The idea of the PCP is similar to the PIP. The task cannot be blocked during its critical section, because the priority of the task is raised to the ceiling priority that must be higher than the highest priority level of all tasks that use the same resource [12]. Moreover, the critical section can be preempted by another critical section with the higher ceiling priority. This ceiling priority is assigned to each semaphores and the priority of each task that enters a critical section inherits to the ceiling level. Figure 1.3 presents the execution of three tasks with the different priorities that share two resources under the PCP.

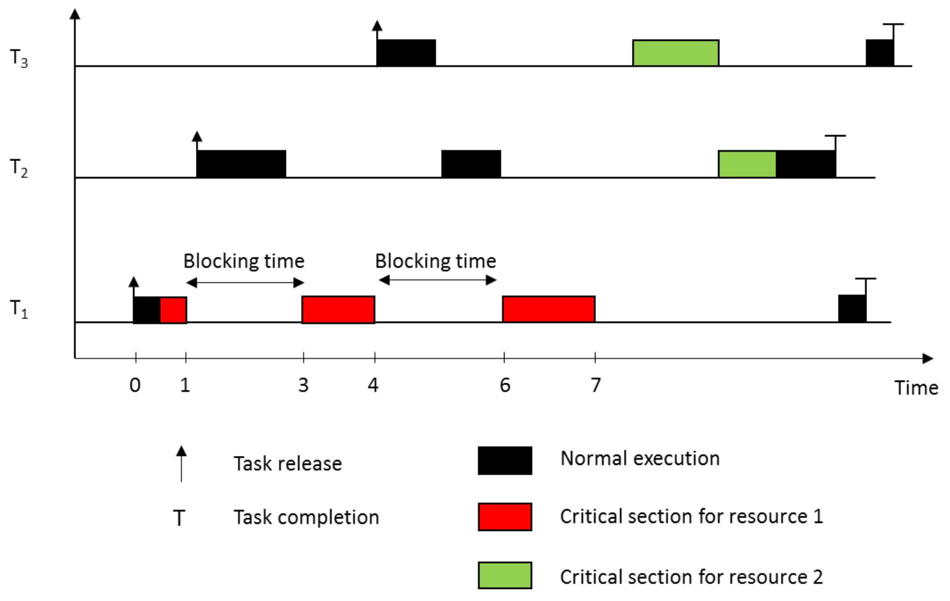


**Figure 1.3:** PCPs behavior

The ceiling priority of the second resource is 1 because the tasks $T_2$ and $T_3$ use the resource 2 for their execution. The ceiling priority of the resource 1 is equal to the priority level of $T_1$. The execution of $T_1$ is blocked during the time intervals [1; 3] and [4; 6] due to the normal execution of tasks which regular priorities are higher than ceiling priority level of the resource 1.

The concepts of these resource synchronization protocols in uniprocessor are used as the basic for protocols on multi-core. We will discuss in detail some of them in next Chapter.

# 2 Resource Synchronization Protocols for Multiprocessor

In this thesis, the resource synchronization protocols have to be implemented and analyzed for multi-core. The assignments of the tasks to several processors differ the protocols implementation for uniprocessor and multiprocessor. Also some protocols allow the tasks migrations between different processors to access a resource. Furthermore, we have to consider how the resources should be allocated on the processors. It means the global resource synchronization is also managed by the corresponding protocols on multi-core.

In this chapter, we will present the theories of three multiprocessor resource synchronization protocols, i.e., MPCP, DPCP, and DNPP, the detailed implementation will be discussed in Chapter 4. Also we review the Multiprocessor resource sharing protocol (MrsP), which has been already implemented on the real-time operation system used in this thesis.

## 2.1 Basics of Resource Synchronization Protocols for Multiprocessor

The resource synchronization protocols can be classified into several groups according such factors as locks mechanism, migrations of tasks, and resources allocation.

At first, we discuss different locks techniques, which are used to block the task while it can access a resource. The multiprocessor protocols can use suspending and spinning mechanisms for the locks. The suspension-based protocol blocks the task if the resource is already obtained by another task. It means, that the processor can execute another task at the time when the higher-priority task is blocked by the execution of the critical section on another processor. In other words, the suspension-based protocols can guarantee the full usage of the processors, even the higher-priority tasks wait for the access to a resource. However, the spinning-based multiprocessor use a busy waiting for the blocking tasks. It means, that these tasks keep the processor and any other tasks cannot start the execution. We will discuss the differences between the suspending and spinning locks in detail in Chapter 4.

There are three classes of the multiprocessor scheduling algorithms, which characterize the migrations of the tasks: global, partitioned, and semi-partitioned. Global approach uses one queue for tasks which wait for the execution. In this case a job can migrate between processors. Partitioned and semi-partitioned paradigms divide the multiprocessor problem into uniprocessor problems, where standard solutions can be used. The tasks are ordered in the separated queues for each processor in partitioned scheduling. In these queues, tasks can be scheduled using uniprocessor algorithms. The basic characteristics for semi-partitioned approach are equal to partitioned paradigm. The tasks migrations between processors differ the semi-partitioned from the partitioned approach. On the one hand, the partitioned approaches are simpler as global solution in implementation, but on the other hand partitioning is an NP-hard problem and it can be a reason for the decreasing of systems performance. Figure 2.1 presents the workflow of the different scheduling approaches.

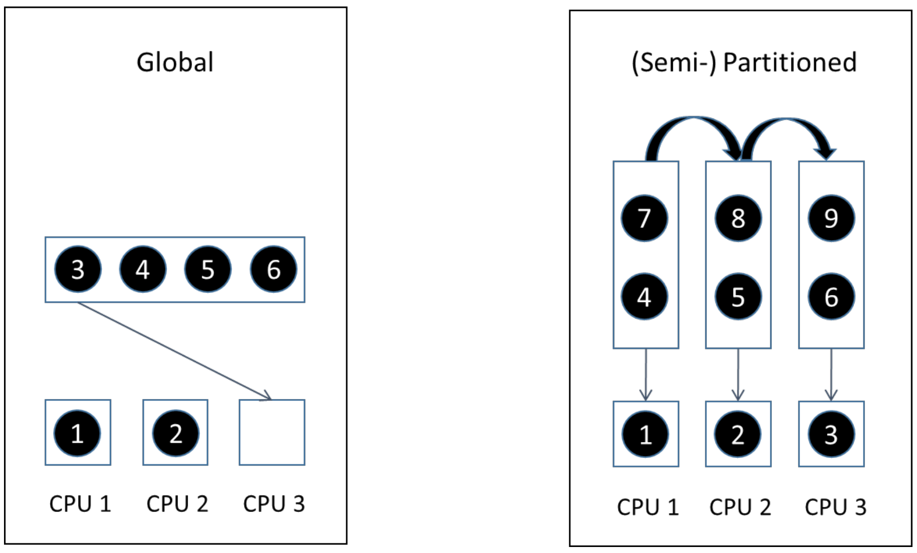


**Figure 2.1:** Workflow of different scheduling approaches

The protocols in multiprocessors can be also categorized by the type of the allocation resources into processor. In the shared-memory protocols, tasks are executed during the critical sections in their own processors. This type of the resources allocation is mostly used by the synchronization protocols in multiprocessor. The second type called distributed-based is used when resources are executed on the designated processor and the tasks have to migrate to this processor in the case of the critical sections executions. The most known protocols which are used the distributed-based type of the resources allocation are the Distributed Priority Ceiling Protocol (DPCP) and the Distributed Non-Preemptive Protocol (DNPP) [2], which will be discussed in the details in Section 2.3 and Section 2.4.

Several global resource sharing protocols can be found in the literature. In this thesis we concentrate on the Multiprocessor Priority Ceiling Protocol (MPCP), the Distributed Priority Ceiling Protocol (DPCP), and the Distributed Non-Preemptive Protocol (DNPP). Also we will analyze the Multiprocessor resource sharing Protocol (MrsP) which has been implemented in the targeted platform used in this thesis. These protocols use partitioned and semi-partitioned, fixed-priority scheduling (it means that each task has a unique priority). We will start our consideration from a review of the MPCP, the DPCP, and the DNPP. Then we introduce the MrsP, which will be used to analyze our development.

## 2.2 Multiprocessor Priority Ceiling Protocol (MPCP)

The multiprocessor priority ceiling protocol was developed by Rajkumar over 20 years ago when multiprocessor real-time application wasn't used in practice [12]. The basic MPCP concept allowed nested resource access, but it was forbidden with the generalization of this protocol. The MPCP is a suspension-based protocol, i.e., a task relinquishes its assigned processor when it tries to get the resource that was already used by other task [2]. In this thesis we concentrate only on global resources. The local resources can be accessed via the Priority Ceiling Protocol (PCP). When the task is blocked to access a global resource, the process becomes available for the local resources. The tasks that use the same resource order in the queue by the priority and execute with the ceiling priority that is greater than the highest tasks priority in this queue. It means, that the critical section is executed with the priority higher than the nominal priorities of the tasks in the waiting queue. The critical section can been preempted only by other critical section with the higher ceiling priority.

At first, we consider two processors environment and two global resources. We synchronize the access for four tasks with different priorities to these resources. The priorities are ordered as the following P $(T_1)$>P $(T_2)$>P $(T_3)$>P $(T_4)$, i.e the priority of $T_1$ is the highest and the priority $T_4$ is the lowest. We assign $T_1$ and $T_3$ to the first processor and $T_2$ and $T_4$ to the second processor. The result is presented in Figure 2.2

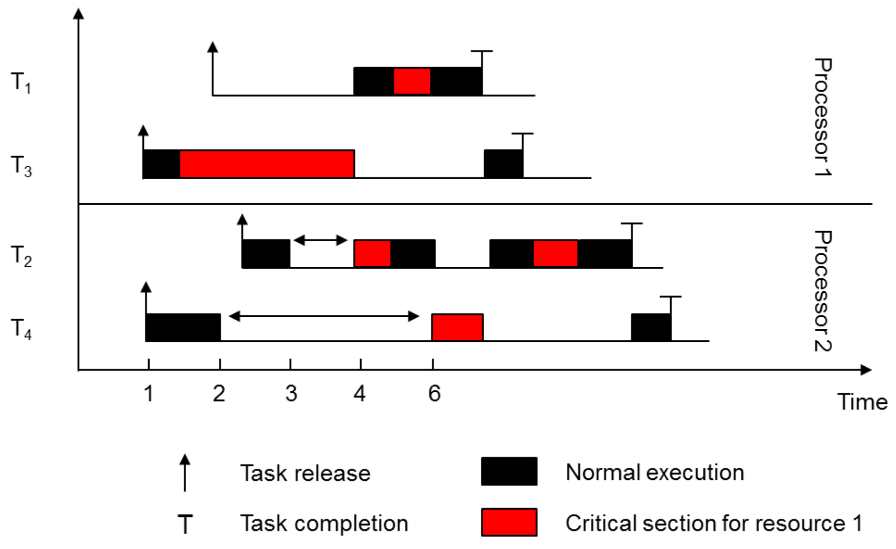


**Figure 2.2:** MPCPs behavior

The task $T_2$ is blocked at time 3 due to the critical section of the task $T_3$. Meanwhile, $T_4$ is also suspended until it accesses a resource at time 6 as the less important priority task from all tasks that execute with this resource. We can see that once a task enters a critical section it cannot be blocked. For example, $T_1$ cannot preempt $T_3$ on the processor 1 despite of the highest original priority. The same situation is on the processor 2 when $T_2$ cannot preempt $T_4$ during its critical section. Figure 2.3 shows the order of the ceilings assignment.

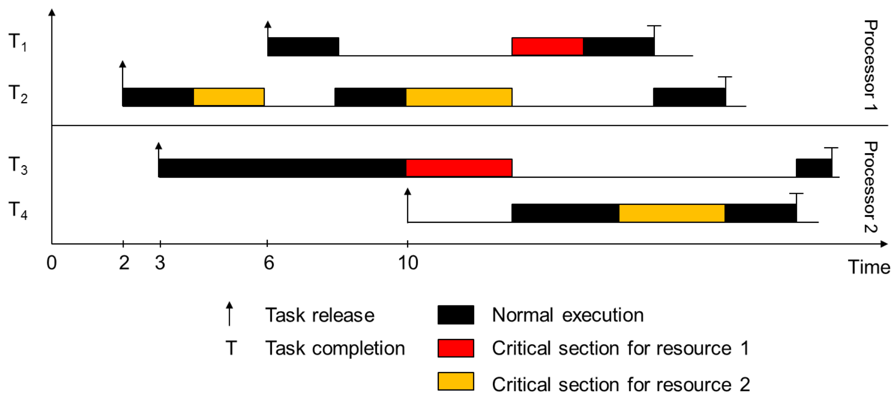


**Figure 2.3:** Ceilings assignment in MPCP

Similar to the previous example, we have four tasks with the different priorities and two resources that are assigned on two processors. The task $T_1$ with the priority $P_1$ and the task $T_3$ with the priority $P_3$ want access to the resource 1. It means that the ceiling priority of the resource 1 is $P_1$, which is equal to the highest priority level of the tasks waiting for the same resource. Similarly, the ceiling priority of the resource 2 is $P_2$. The ceiling priority of the resource 1 is accepted by $T_3$ at the time unit 3, in the similar way $T_2$ gets the ceiling priority of the resource 2 at the time unit 2 when the task enters a critical section. Both ceiling levels are presented in Table 2.1.

| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Resource 1 | - | - | - | - | - | - | 1 | 1 | 1 |
| Resource 2 | - | - | 2 | 2 | 2 | 2 | 2 | 2 | 2 |

**Table 2.1:** Assignment of MPCP ceiling priority

The MPCP uses shared memory to access global resources. It means, that all processors access to the resources directly. It is the contrast to the Distributed Priority Ceiling Protocol (DPCP), where global resources are assigned to the particular processors. We will discuss the DPCP in detail as the next.

## 2.3 Distributed Priority Ceiling Protocol (DPCP)

The Distributed Priority Ceiling Protocol (DPCP) was developed by Rajkumar in [20]. In contrast to the MPCP, the DPCP provides RPC-like invocation of an agent on the resource's processor [2]. It means, that each task has to migrate to the special processor during obtaining of resource. The task has to migrate back after the critical sections execution on its own processor where it was released. The DPCP defines two types of the processors. All critical section are assigned to the processor, called synchronization processor. Meanwhile, the tasks with the normal execution run on the processor, called application processor [4]. The tasks are accessed a resource on the synchronization processor perform PCP on the uni-processor system. Similar to the MPCP, it is forbidden to request for the global resources with the nesting. It can guarantee that the protocol avoids deadlock situation. The tasks are ordered by the priority to access a resource. The DPCP uses one local agent for each task. These agents have a higher priority than any normal task. It is important for the DPCP due to the preemption of any normal tasks execution by the critical sections on the same processor. Similar to the MPCP, the DPCP uses suspending for the blocked tasks, it means that another task can execute during the blocking time on the same processor.

Figure 2.4 depicts the execution of three tasks under the DPCP on two processors. The resource is assigned to the synchronization processor, where the tasks $T_1$, $T_2$, and $T_3$ migrate to access a resource. The agents are provided for the critical sections of the tasks, which are executed on another processor. $A_3$ starts at time 5 when $T_3$ enters a critical section. Similarly, $A_1$ and $A_2$ become active at time 8 when $T_1$ and $T_2$ want access a resource which is already obtained by $T_3$. $T_1$ as the highest priority task obtains a resource one time unit later when $A_3$ releases the resource.

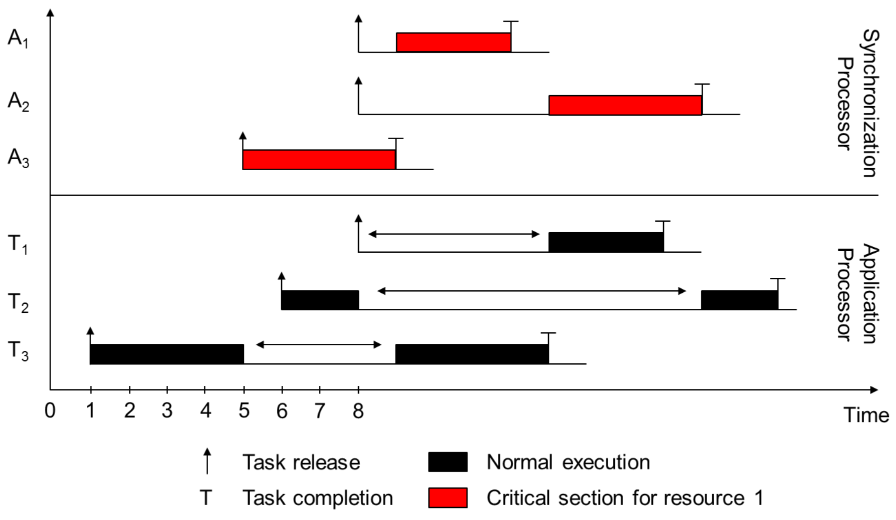


**Figure 2.4:** DPCPs behavior

The DPCP as a protocol of resource-oriented scheduling has one important advantage in compare with the MPCP and the MrsP. Using resource-oriented scheduling guaranties to avoid of the chained blocking: the normal execution of the higher priority tasks cannot be preempted by any critical sections of other tasks [4].

As the next we will discuss the DNPP, which uses the similar principles of the tasks execution as the DPCP.

## 2.4 Distributed Non-Preemptive Protocol (DNPP)

The behavior of the Distributed Non-preemptive Protocol (DNPP) is similar to the DPCP. The task is migrated to execute its critical section on the synchronization processor. The tasks in the normal execution are running on the application processors.

The tasks execution within the critical section is different for the DPCP and the DNPP. In particular, the task cannot be preempted during its execution in the critical section under the DNPP. The DPCP allows the preemption of the tasks during critical section by another task with the higher ceiling priority. In this case, the task resumes after the blocking task releases the resource. Figure 2.5 presents the execution of three tasks with different priorities under DNPP. The resource is assigned to the synchronization processor, where the tasks $T_1$, $T_2$, and $T_3$ migrate to access a resource. The task $T_2$ can not be preempted during its critical section by the task $T_1$ despite of the higher priority of the task $T_1$. The task $T_1$ can access a resource only after finishing of the critical section for $T_2$.

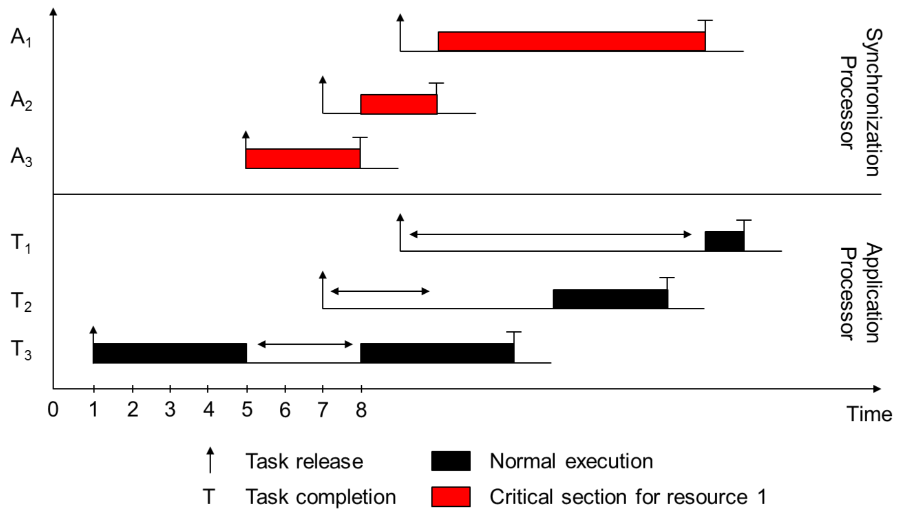


**Figure 2.5:** DNPPs behavior

We will use the implementation of the DPCP to develop the DNPP, due to the similar behavior of these two protocols. The only issue that we need to consider is the implementation of non-preemptive feature for the DNPP. It means we have to guarantee that the priority of the task in the critical section is always higher than priority of any other tasks in the system. We will discuss more details about implementation of the DNPP in Chapter 4.

As the next we will discuss the theory of the MrsP, which is used to analyze the implementations flow of the multiprocessor resource synchronization protocols in RTEMS.

## 2.5 Multiprocessor Resource Sharing Protocol (MrsP)

The Multiprocessor Resource Sharing Protocol (MrsP) was presented by A.Burns and A.J. Wellinls in [1].

The idea of the MrsP is the following: the nominal priority of the task has been raised to the ceiling priority when this task requests a resource. When the resource has already been used by another task, the requesting task was added in the queue according to a FIFO ordering and busy waits until it holds the resource. The ceiling priority level is defined as the highest priority of tasks that can require access to the same resource. The main difference between the MrsP and the MPCP (or the DPCP) is the helping mechanism between owner of the resource and the spinning tasks. It means if the current owner is not executing, the spinning task of the same resource can help the owner to make a progress. One of the main problem by the helping mechanism is a tasks migrations. The MrsP solves this problem by updating the priority of the migrated task to the level higher than priority of the spinning task in this partition (it is the equal to the ceiling priority in this partition) [1, 3]. The helping mechanism can reduce the execution time due to the full usage of processors performance. We can conclude three features of the MrsP which differ it from the resource synchronization protocols that we have introduced above in this chapter.

- The tasks which are waiting for the same resource order by FIFO in the waiting queue.
- The MrsP is the busy waiting protocol. It means it uses spinning instead of suspending for the blocked tasks.
- The helping mechanism is the most challenging part of this protocol. It is used for the tasks migration between processors. The tasks migration means that we used semi-partitioned approach under priority-fixed scheduling.

The helping mechanism should take place only in the case when a resource holder is not executing and there is the task spinning on the own processor until it can get the same resource [3]. When the semaphore owner uses the help of the spinning task, it has to migrate to the processor of the helping task. After the migration the priority of the task should be updated to the level higher than the current ceiling priority on this processor, it makes a possible to preempt the spinning task and start of the execution.

Figure 2.6 presents how FIFO ordering in the waiting queue and spinning for the blocked tasks works for the MrsP. We consider three processors and four tasks sharing one resource.

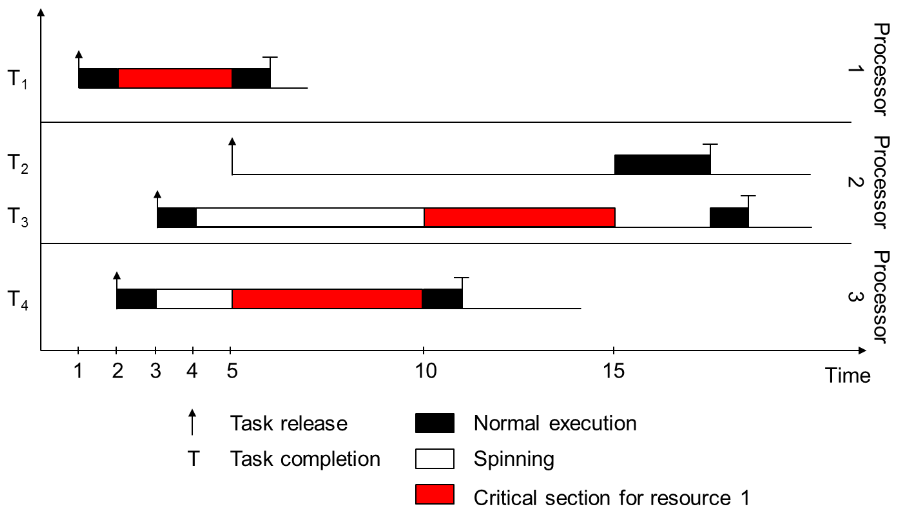
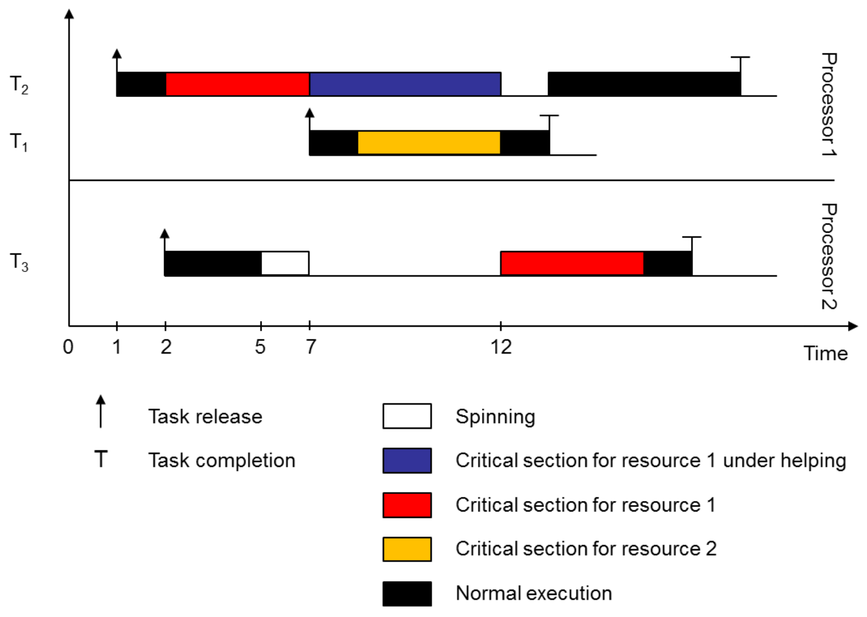


**Figure 2.6:** FIFO and spinning mechanisms in MrsP

The task $T_1$ accesses a resource at time 2 on processor 1, $T_4$ starts to spin for this resource on time unit later on processor 3. Meanwhile, $T_3$ wants to get the same resource at time 4 on the processor 2. After $T_1$ releases the resource at time 5, the task $T_4$ obtains it as the next, in despite of its nominal priority is less important than the priority of $T_3$. The tasks execution on the processor 2 shows us the principles of the spinning lock. $T_2$ is blocked during the spinning time of $T_3$. It means, that spinning task saves the control of the corresponding processor and other tasks cannot be executed during the blocking time.

Figure 2.7 presents how works the helping mechanism under the MrsP. We consider three tasks with the different priorities on two processors. The task $T_1$ with the priority $P_1$ and the task $T_3$ with the priority $P_3$ want access the resource 1, the resource 2 is needed for the execution of the task $T_2$ with the priority $P_3$. At the time 7, the task $T_2$ can migrate to the processor 2 due to spinning of the task $T_3$. At the same time the task $T_1$ can start its execution of the processor 1. The task $T_2$ has to migrate back to the processor 1 after the execution in the critical section.



$T_2$
$T_1$
$T_3$
Processor 1
Processor 2
0
1
2
5
7
12
Time
Task release
Task completion
Spinning
Critical section for resource 1 under helping
Critical section for resource 1
Critical section for resource 2
Normal execution

**Figure 2.7:** Helping mechanism in MrsP

The MrsP has been already implemented in RTEMS, the real time operation system which is used in our thesis. We will discuss more details how aforementioned protocols are implemented in Chapter 4.

# 3 RTEMS Operation System

This chapter introduces RTEMS, the real-time operating system used in this thesis. Section 3.1 presents the basic principles of RTEMS. Introduction for SMP Support in RTEMS is made in Section 3.2. Section 3.3 describes the process scheduling and basics about the integrated schedulers in RTEMS. Finally, Section 3.4 focuses on the Application Program Interface in RTEMS.

## 3.1 Basics of RTEMS

In this thesis we focus on implementation and analyzing the resource synchronization protocols in multiprocessor in RTEMS, which is the abbreviation from the Real-Time Executive for Multiprocessor Systems. RTEMS is managed by OAR Corporation in cooperation with Steering Committee [8]. The RTEMS code is written in C and Ada. We will concentrate only on C in this thesis, since Ada is not supported by SMP RTEMS (Symmetric Multiprocessing RTEMS) [11].

Figure 3.1 [22] presents the architecture a real-time operation system on the RTEMS basis.

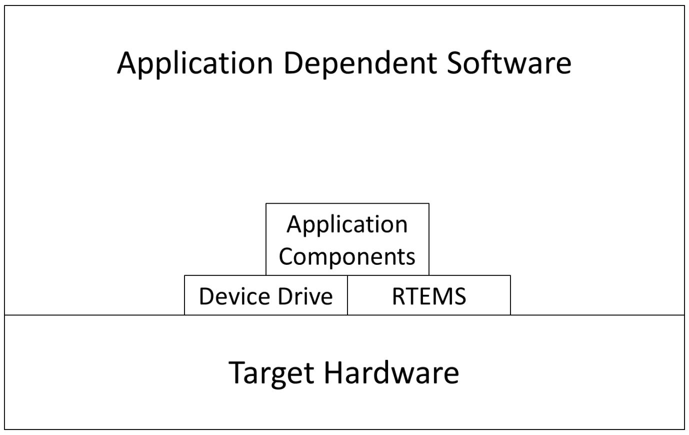


**Figure 3.1:** RTEMS architecture [22]

As an operation system, RTEMS works between the hardware and application layers. This RTOS uses the hardware resources with the providing I/O Interface. RTEMS uses their own software libraries to support different object types. These types are the special objects to control and modify all features of the system. For example, these objects are tasks, semaphores, timers, or message queues. Each object is assigned to its identical ID called rtems_id which is used for the identification. This identical ID is a thirty-two bit entity which contains of index, node, and class. Figure 3.2 presents the structure of the identical ID.

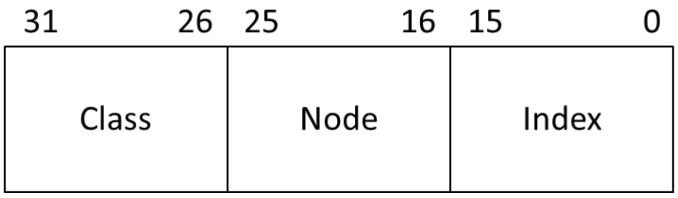


**Figure 3.2:** Structure of the identical id

We provide not only the identical ID for each object, but also the objects name which is connected with the ID in the name table.

The following example presents how to create an object name within the directive rtems_task_create [9]:

rtems_task_create (rtems_build_name ('T','A','S','K'), 2, RTEMS_MINIMUM_SEIZE, RTEMS_DEFAULT_MODES, RTEMS_DEFAULT_ATTRIBUTES, &id). The directive rtems_build_name returns the thirty-two bits by defining of the four characters.

RTEMS source tree structure is presented in Figure 3.3

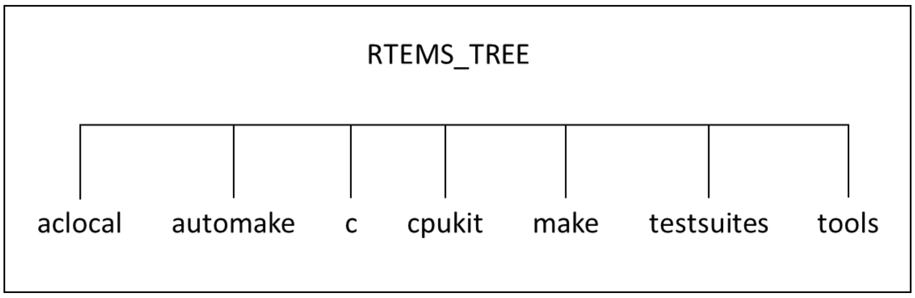


**Figure 3.3:** RTEMS source tree structure

The content of the directories under the RTEMS_TREE looks as follows [9]:

- testsuites/smptests/
  This directory contains the tests for the RTEMS Classic API on the multiprocessors.
- testsuites/sptests/
  The testsuites for the RTEMS Classic API are implemented in this directory.
- cpukit/posix/
  This directory contains the implementation of the Posix API.
- cpukit/rtems/
  RTEMS Classic API is developed in this directory.

- cpukit/score/
  RTEMS SuperCore is included in this directory. The source and header files are implemented here.

RTEMS provides all information about each task in the corresponding data structure called the Thread Control Block (TCB) which can be found in cpukit/score/include/rtems/score/thread.h [9]. The TCB is created with the rtems_task_create directive and is removed with rtems_task_delete directive. Each Thread Control Block contains a tasks name, object ID, nominal priority, resource count, pointers to the chains, the boolean attributes which define the object as preemtible and global.

The chain manager in RTEMS is responsible for the queues building. The chain is structured as a double-linked list of the nodes. The chain is controlled by a structure Chain_Control in the header file cpukit/score/include/rtems/score/chain.h [9]. This structure includes the pointers to the first, last, previous, and next elements in the queue. The chain manager provides the operations for initializing, inserting of new element, and extracting of an element. The tasks are allocated to the ready queue until the processor is available for their execution. The ready queue provides the lists where each layer is assigned to the corresponding priority level [9, 11]. The tasks which have an equal priority are ordered by FIFO within their priority group. The current running task is the first element of the highest priority level group. A task is deleted from the queue after the completing of its execution or the blocking by another task. Then the scheduler selects the next tasks in the ready queue and allocates a processor for it.

RTEMS has some advantages in compare with other Real Time Operation Systems (RTOS). It supports many interface standards and open standard application programming interfaces such as Classic and POSIX [11]. Each of them can configure set of tasks that are automatically started after RTEMS initialization. In this thesis we will use Classic API due to its better integrating feature set in compare with other standards.

There are many features that can characterize RTEMS as a competitive and a modern RTOS: it provides symmetric, asymmetric, and distributed multiprocessing, priority inversion avoidance protocol for uniprocessor such as the PIP, the PCP, etc. Depending upon the design requirement, the unnecessary managers can be removed from each application according to its own requirements to decrease the code size of RTEMS.

## 3.2 Introduction for SMP Support in RTEMS

In this thesis we use the symmetric multiprocessor support (SMP) in RTEMS for the implementation of the resource synchronization protocol. The SMP support has to be enabled in the configuration with the option –enable-smp. The SMP application must be configured with the corresponding macros [9]. RTEMS API provides several symmetric multiprocessing directives rtems_get_processor_count, rtems_scheduler_ident, rtems_task_set/get_scheduler, rtems_scheduler_add/remove_processor, etc.

The SMP framework has a structure like as plugin. A set of operations (e.g. obtain/release semaphore, create/delete object) can be called within the API. Also an application can be executed with several schedulers on different processors. As far we know, there are three implemented scheduling algorithms for multiprocessor in RTEMS. We will discuss more details about it in the next chapter.

The important entity which is used to contain all scheduled and ready tasks is the scheduler node. So, we can say that the scheduler node is the special box which is implemented in each scheduler with the tasks information. A scheduler instance consists of a scheduler algorithm which provides the scheduler operations (e.g. block/unblock thread, asking for help), and a data structure, which defines a scheduler name, operations, and context of scheduler instance [3].

Overall, RTEMS is a good choice for the implementation and evaluation for the resource synchronization protocols on many-core systems due to the integrated SMP support.

## 3.3 Process Scheduling in RTEMS

The main point that we need to understand before we start the implementation in RTEMS is the process scheduling. The process scheduling in the real-time kernel is similar to the process scheduling in the Linux kernel. We have several task states, semaphores that protect its execution during the critical sections, and the waiting queue for the tasks which cannot immediately access a resource.

RTEMS provides five states which can be entered by a task during its execution: NON-EXIST, DORMANT, READY, EXECUTING, and BLOCKED (see Figure 3.4). We describe these states in the details [11].

- NON-EXIST: The task is not created or is already deleted.
- DORMANT: This state describes the task when it was created but is not started. This task cannot compete for resources.
- READY: The task is started and it can be assigned to the processor. Also the task can get this status after the unblocking operation or the yielding operation on processor.
- EXECUTING: The task is assigned to the processor and can access a resource. The scheduling algorithms have several criteria's that select the order of tasks execution. RTEMS SMP provides one executing thread per core.

- BLOCKED: The task takes this state after a blocking action and it cannot be allocated to the processor. The task is unblocked only after such actions as resume, release of resource, or timeout.

The Figure 3.4 [11] shows the connections between each task state in RTEMS.

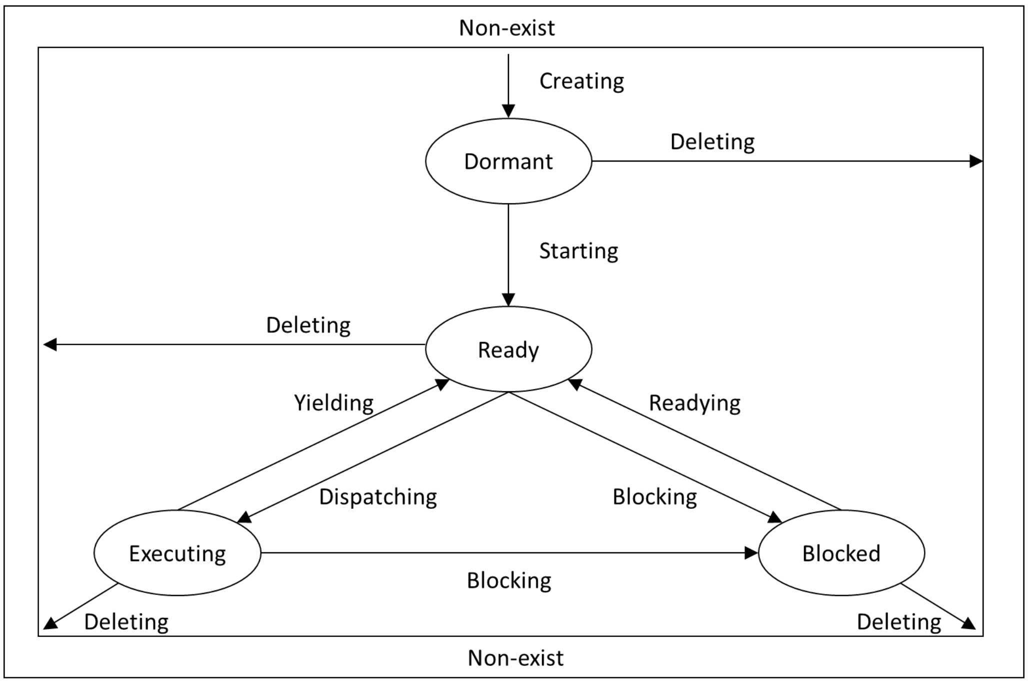


**Figure 3.4:** RTEMS task states [11]

The directives rtems_task_suspend,rtems_wake_after, rtems_semahore_obtain, and rtems_wake_when can change the state of the task to BLOCKED. Meanwhile, using of such directives as rtems_task_resume, rtems_semaphore_release, and rtems_task_resume unblocks the task and the state is set as READY. The possible states transitions can be found in the header file rtems-smp/cpukit/score/include/rtems/score/statesimpl.h [9].

RTEMS uses a new scheduler that provides a virtual table of functions pointers to connect scheduler-specific code with the existing thread management [5]. The main idea of scheduler can be defined as the management of the tasks in the ready state and the solution which of the tasks will be run as the next on the processor. The scheduler algorithms are strictly coupled with the structure defined for each processor in the Per_CPU_Control within the cpukit/score/include/rtems/score/percpu.h [9]. The fields which are used by a scheduler in this structure are the pointer to the executing task and to the next task that is ready to execute.

The scheduler algorithm should provide the connections between the process states. The multiprocessor scheduling algorithms which are already implemented in SMP have the same behavior as the Deterministic and Simple Priority Schedulers. Moreover, they provide the helping mechanism to force the tasks migration between processors.

We will discuss more details about it in the next Chapter.

## 3.4 RTEMS Application Programming Interface

We have discussed in Section 3.1 that RTEMS provides two application programming interfaces. We use in this thesis one of them called Classic API, because it is already supported by the system. Meanwhile, the second API called POSIX is still under development [11].

Figure 3.5 [22] presents the managers of RTEMS API.

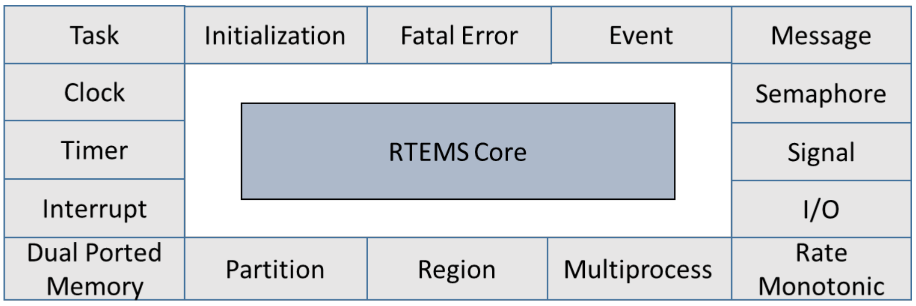


**Figure 3.5:** RTEMS Managers [22]

To verify the implemented protocols,the used directives are from the following RTEMS managers [11]:

- Initialization Manager: it initializes RTEMS, device drivers, the root filesystem, and the applications.
- Semaphore Manager: this manager supports synchronization and mutual exclusion.
- Interrupt Manager: techniques for the handling of externally generated interrupts are provided by this manager.
- Task Manager: it supports all necessary directives to administer tasks.
- Event Manager: this manager provides a mechanism to handle communication and synchronization among objects in RTEMS.
- User Extension Manager: extension routines are handled by this manager.
- Clock Manager: it provides the time related capabilities.

We will discuss the implementation of the tests application in RTEMS API in Chapter 5.

# 4 Implementation for New Protocols on RTEMS

In this thesis we need to implement three resource synchronization protocols which have not been implemented in RTEMS. There are the Multiprocessor Priority Ceiling Protocol (MPCP), the Distributed Priority Ceiling Protocol (DPCP), and the Distributed Non-Preemptive Protocol (DNPP) which we have introduced in Chapter 2.

## 4.1 Challenges in the Implementation of MPCP

Before we start to discuss the challenges that are in the MPCP implementation in RTEMS, we need to review the most important features of this protocol. Since the MrsP has been already implemented under SMP RTEMS, we can leverage on it to quickly get the idea of the protocol design in RTEMS.

At first, the MPCP like the MrsP accesses the global resources directly, we don't need to use the fixed processors to assign the resources. Furthermore, each semaphore uses a ceiling priority for each processor. There are three variations between both protocols:

- The MPCP is a suspension-based protocol. The MrsP uses a spinning for the blocked tasks.
- The waiting queue in the MPCP is ordered by priority of the waiting tasks. The MrsP uses FIFO ordering.
- The MrsP unlike the MPCP provides a help mechanism to force the tasks migration between the processors.

The tasks migration is strictly coupled with the scheduling algorithms in RTEMS. The tasks can migrate between processors due to the several reasons [11]:

- The task can change assigned scheduler using rtems_task_set_scheduler directive in API.
- Each task can move to another scheduler after unblocking operation.
- By using such protocols as the MrsP or the MPI (Migratory Priority Inheritance).

The integrated tasks migration follows the rules that disable the tasks execution on different processors at the same time. It is performed due to using special indicator in the task context [11]. The challenge in the implementation of the MPCP is a new scheduler algorithm that can support MPCP features and avoid the integrated tasks migration in RTEMS. We will discuss it as the next.

## 4.2 Implementation of Scheduler for the MPCP

We have introduced the main characteristics of the MPCP in Chapter 2 and presented the challlenges of the MPCP implementation in Section above. As the next we are going to describe the scheduler of the MPCP in detail. First, we will discuss how a new scheduler is configured in Subsection 4.2.1. As the next, Subsection 4.2.2 describes the concept which is used for the implementation of the new scheduler algorithm in the SMP RTEMS.

### 4.2.1 Configuring SMP Schedulers

We have discussed in Chapter 3 that it is possible to assign different schedulers to different processors in RTEMS. Also it is important for an application to divide processors into clusters. Such clusters with a cardinality of one are called as partitions, which we have described in Chapter 2. At first, a user has to define which scheduler is more preferable for the application. To date, there are three integrated schedulers in SMP RTEMS: Deterministic Priority Affinity SMP Scheduler, Simple Priority SMP Scheduler, and Deterministic Priority SMP Scheduler [8]. The communication diagram for SMP Scheduler is presented in Figure 4.1 [8].

Therefore, Deterministic Priority Affinity SMP Scheduler extends Deterministic Priority SMP Scheduler by threads adding to core affinity support. The design of SMP Schedulers is based on the priority scheduling algorithms. It means, that the algorithm will always select the highest priority task to execute. In general, the priority based algorithm for the Deterministic Priority SMP Scheduler is implemented as an array of FIFOs with a FIFO per priority. The Simple Priority SMP Scheduler has the same behavior as the Deterministic Priority SMP Scheduler, but it proposes only one linked list for all ready tasks [11]. Our new scheduler extends the Deterministic Priority Scheduler SMP because we need to implement priority based scheduler that has to include the FIFO ordering per each priority group of the tasks. RTEMS SMP supports 256 priority levels where the priority level of one (1) is the highest in the system.

RTEMS as the real-time operation system must been configured as Time-Triggered System to guarantee a real-time execution [10]. We need to use the Clock Manager that provides an abstract parameter Tick. The duration of the ticks in the microseconds and the number of microseconds for the milliseconds value are configured as follows [9]:

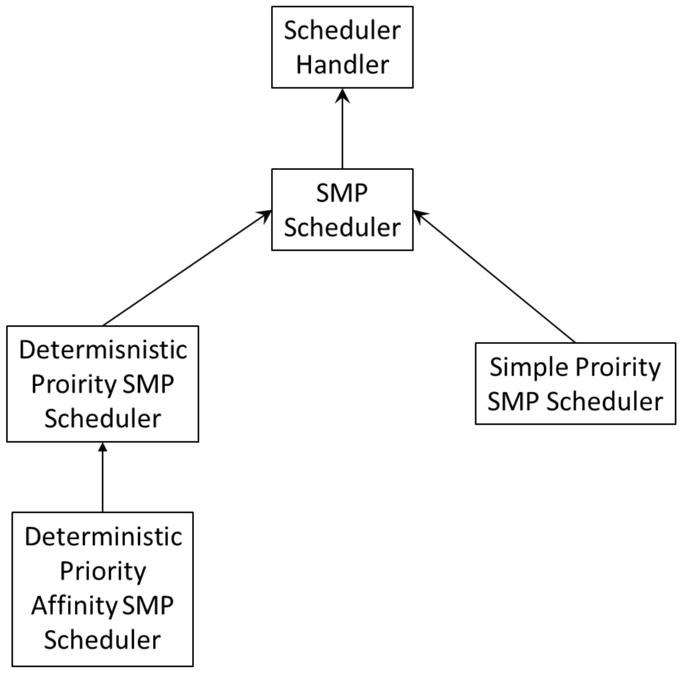

Scheduler Handler
SMP Scheduler
Determisnistic Proirity SMP Scheduler
Simple Proirity SMP Scheduler
Deterministic Priority Affinity SMP Scheduler

**Figure 4.1:** Communication diagram for SMP scheduler [8]

```
CONFIGURE_MICROSECONDS_PER_TICK 1000 // 1000 us pro Tick
RTEMS_MILLISECONDS_TO_MICROSECONDS(10)
```

In the case when the scheduler is provided by the user, the following macros have to be configured [9, 11]:

```
CONFIGURE_SCHEDULER_USER
CONFIGURE_SCHEDULER_CONTEXT CONFIGURE_SCHEDULER_USER_PER_THREAD
```

As discussed above, we implement own scheduler for the MPCP which is different from built-in SMP schedulers. Since the MPCP supports the priority based queue for the ready tasks, the MPCP scheduler should extend the Deterministic Priority SMP Scheduler. The scheduler of the MPCP in our implementation is configured as follows:

```
CONFIGURE_SCHEDULER_NAME rtems_build_name('M', 'P', 'C', ' ')
CONFIGURE_SCHEDULER_CONTEXT \
RTEMS_SCHEDULER_CONTEXT_MPCP_SMP(dflt,\ CONFIGURE
_MAXIMUM_PRIORITY + 1 \ )
CONFIGURE_SCHEDULER_CONTROLS \
RTEMS_SCHEDULER_CONTROL_MPCP_SMP(dflt, \ CONFIGURE_SCHEDULER_NAME\ )
```

Furthermore, the scheduler has to be called with the definition of the entry points, i.e. the functions which are developed in the RTEMS core for the scheduling algorithm. We define these functions for MPCP scheduler as follows:

```
# define SCHEDULER_MPCP_SMP_ENTRY_POINTS \
{
  _Scheduler_priority_SMP_Initialize, /* initialize scheduler entry point */
  _Scheduler_MPCP_SMP_Yield, /* yield scheduler node entry point */
  _Scheduler_MPCP_SMP_Block, /* block thread entry point */
  _Scheduler_MPCP_SMP_Unblock, /* unblock thread entry point */
  _Scheduler_MPCP_SMP_Update_priority, /* update priority of node*/
  _Scheduler_default_Map_priority,
  _Scheduler_default_Unmap_priority,
  _Scheduler_default_Reconsider_help_request,
  _Scheduler_default_Ask_for_help,
  _Scheduler_MPCP_SMP_Withdraw_node, /* withdraw scheduler node */
  _Scheduler_MPCP_SMP_Add_processor, /* add processor entry point */
  _Scheduler_MPCP_SMP_Remove_processor, /* remove processor entry point */
  _Scheduler_MPCP_SMP_Node_initialize, /* initialize node entry point */
  _Scheduler_default_Node_destroy,
  _Scheduler_default_Release_job,
  _Scheduler_default_Cancel_job,
  _Scheduler_default_Tick,
  _Scheduler_SMP_Start_idle /* start idle task entry point */
  SCHEDULER_OPERATION_DEFAULT_GET_SET_AFFINITY
}
```

Unfortunately, there is no document in RTEMS which describes how to properly design a scheduler. We have to examine condefs.h and scheduler.h [9] to figure out how the configuration of SMP schedulers works. We will discuss more details about it as the next.

### 4.2.2 Implementation for New SMP Scheduler

We have introduced the basic principles of the MPCP and SMP RTEMS scheduling in Chapter 3 and 4 respectively. In this Subsection we will introduce the concept of implementation for MPCP scheduler. We have discussed in Section 3.2 that the scheduler node is important entity in the scheduling concept. The scheduler nodes management in RTEMS is presented in Figure 4.2 [5].

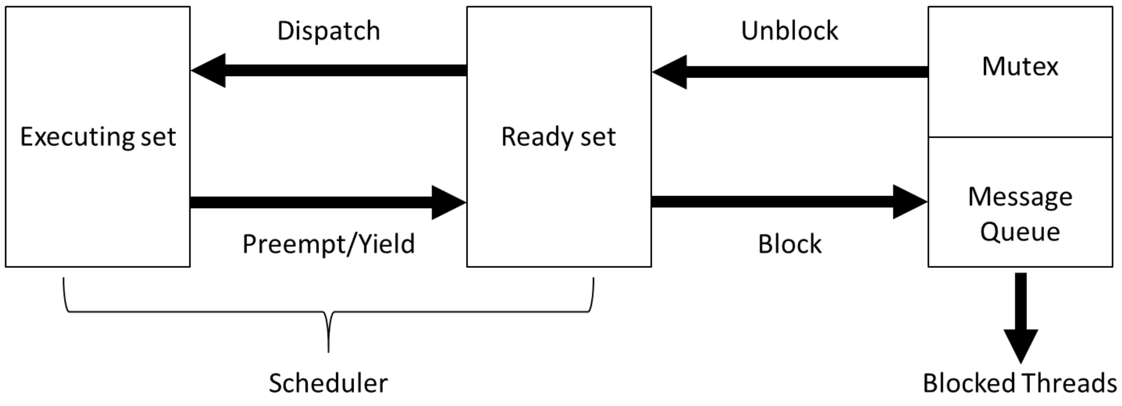


**Figure 4.2:** Scheduler node managment in RTEMS [5]

The scheduler uses two queues of threads to order them by the states. The scheduler solves which of the scheduler nodes will be executed as the next according the thread priority and the position in the ready chain. The node from the ready chain can moved to the executing chain by using of dispatch operation. Conversely, the scheduler node can be changed from executing to ready state by such functions as preempt and yield. The set of the blocked threads is protected by the semaphore which is used to control the execution in the critical section. We will discuss the implementation of functions for the scheduler in next Subsection.

The count of the scheduler nodes is equal to the count of processors which are used within this scheduler instance. Each scheduler node provides the tasks execution for exclusive one processor. The scheduler node can provide the execution of three kind of tasks: owner, user, and idle. The idle task is a special task with the lowest priority in the system (it is equal 255 for RTEMS) executing when no other tasks are ready to run on the processor. The owner of the scheduler node is defined with a task creation and it cannot be changed during the living time of the scheduler node [11]. SMP RTEMS provides a special scheduler helping protocol, which is responsible for the task migrations, it means the user of the scheduler node can be changed. In the MPCP such tasks migrations have to be avoid.

We have introduced the task states in RTEMS in Chapter 3. RTEMS provides also scheduler nodes sets to manage nodes that have a scheduler state READY, EXECUTE, and BLOCKED [9]. The difference among these states is important for our implementation. If the corresponding thread is not ready to execute, a scheduler node has a state BLOCKED. A scheduler node is in the state SCHEDULED if the corresponding thread is ready and the processor was already allocated for its execution. The state READY for a scheduler node means that the corresponding thread is ready, but the processor is not yet allocated for its execution. Figure 4.3 [9] presents the connections between the scheduler states.

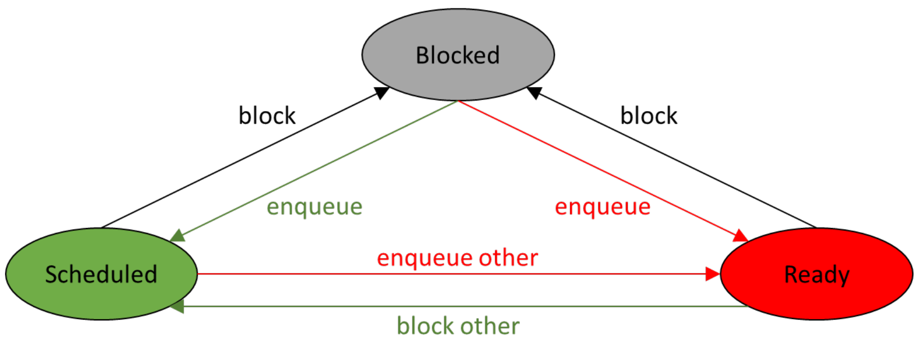


**Figure 4.3:** Connections between the scheduler nodes states in RTEMS [9]

The states of the scheduler nodes can be changed via the functions: Enqueue_ordered, Enqueue_scheduled_ordered, and Block. The scheduler node is transformed from the state READY to the state SCHEDULED only when the executing node was blocked.

Furthermore, the state node can be switched from SCHEDULED to READY via Enqueue_scheduled_ordered, i.e. when the thread with the higher priority waits for the execution. The node state can be switched to BLOCKED from both other states by directly or indirectly blocking.

We will discuss the implementation of these functions in the details.

The main goal of each scheduler is to determine which thread should be executed on the processor as the next. We call such thread as heir and the thread which is currently running on the same CPU as executing. The structure that protects the connection between the heir and executing threads in RTEMS is Per_CPU_Control. RTEMS provides a special structure called Thread Control Block (TCB) that presents the information about each thread. TCB includes such important properties of threads as state, priority, and count of resources. Thread Control Block connects with the related scheduler by using Thread_Scheduler_control that provides the following properties of the scheduler [9]:

```c
typedef struct {
  ISR_lock_Control Lock;
  Thread_Scheduler_state state;
  const struct Scheduler_Control *home;
  struct Per_CPU_Control *cpu;
  Chain_Control Wait_nodes;
  Chain_Control Scheduler_nodes;
  Scheduler_Node *requests;
  Scheduler_Node *nodes;
} Thread_Scheduler_control;
```

The pointers to Scheduler_Node structure are important to handle the scheduler node of this thread. The queues for the waiting and scheduling nodes are integrated via Chain_Control.

RTEMS SMP has already integrated task priority queues that consist of two queues levels [9, 11]. The first level queue is a global queue that provides FIFO order and contains priority queues of the second level. Each priority queue can be assigned to a single scheduler instance and holds only the tasks from this scheduler instance. The scheduler instances select the highest priority task from the first priority queue to dequeue in the global queue. It means, that the concept of integrated task priority queues can be applied to implement the MPCP that uses the priority order to access a resource and the FIFO order by tasks of the equal priority.

We have to provide two important scheduler structures for the new MPCP scheduler. There are Scheduler_MPCP_SMP_Node and Scheduler_MPCP_SMP_Context. Since we develop the MPCP scheduler based on the Deterministic Priority SMP Scheduler, the MPCP scheduler uses the structures Scheduler_Priority_SMP_ Node and Scheduler_Priority_SMP_Context. These structures provide the following attributes of the scheduler [9]:

```c
typedef struct {
  Scheduler_SMP_Context Base;
  Priority_bit_map_Control Bit_map;
  Chain_Control Ready;
} Scheduler_priority_SMP_Context;
```

```c
typedef struct {
  Scheduler_SMP_Node Base;
  Scheduler_priority_Ready_queue Ready_queue
} Scheduler_priority_SMP_Node;
```

The attribute Ready_queue in the Scheduler_priority_SMP_Node differs the Deterministic Priority SMP Scheduler from the Simple Priority SMP Scheduler.

Now we present the main functions which are provided in MPCP SMP scheduler. The ready queue of the tasks is controlled by the scheduler rules where the priority order is specified. The scheduler uses the call-out functions [9] from Subsection 4.2.1 to handle the scheduling decision.

As discussed in Chapter 3, the scheduler has to prepare several operations that protect the transactions between tasks. There are Scheduler_MPCP_SMP_Yield, Scheduler_MPCP_SMP_Block, Scheduler_MPCP_SMP_Unblock, _Scheduler_MPCP _Allocate_processor. These functions for the MPCP SMP scheduler work the same as the corresponding functions from the Deterministic Priority SMP Scheduler. The function Scheduler_MPCP_SMP_Block changes the state of the thread from READY or SCHEDULED to BLOCKED. It works as the following: at first, we call the function Scheduler_Block_node to block the own node of our thread, after that we check the status of the node before the blocking operation and depending on this state the node has to be removed from the corresponding queues. The function Scheduler_MPCP_SMP_Unblock executes by the following way. At first, we call the function that unblocks the thread on the corresponding scheduling node, after that the status of the scheduler node should be checked. If our node has the state READY or BLOCKED, the updating priority operation has to be provided. We call Scheduler_MPCP_SMP_Yield function in case when a task voluntarily releases control of the processor. This function is invoked by using the rtems_task_wake_after directive which blocks the task for the given time interval [11]. The function has to check the status of the node and remove it from the corresponding queues.

The original function _Scheduler_SMP_Allocate_processor_lazy in score/schedulersmpimpl.h [9] forces the tasks migration between processors. The function _Scheduler_MPCP _Allocate_processor is strictly connected with the function _Thread_Dispatch_update _heir that can change the status of the node from SCHEDULED to READY.

The previous operations are provided for the tasks transactions, but there are several operations that control the scheduler nodes behavior. These function are Enqueue_ordered, Enqueue_scheduled_ordered, and Withdraw_node. The Enqueue operations provide the updating of two queues that are used in the scheduler: queues of the ready and running tasks. Withdraw_node in RTEMS is used to change the state of the node from SCHEDULED or READY to BLOCKED, it means that this node doesn't have ready tasks for the execution. The function Withdraw_node is strictly coupled with the function Scheduler_block that blocks a thread with respect to the scheduler.

## 4.3 Implementation of the MPCP

One way to protect the symmetrical access to an exclusive resource is to use semaphores in RTEMS. The system provides three types of the semaphore: binary, simple, and counting. The binary semaphore allows nested access that is a difference between the simple and binary semaphores. The counting semaphore is used to protect access not only for the exclusive resource but also for the multiunit resources [11]. We define the MPCP as binary semaphore to provide the nested access during its execution.

We have discussed in Chapter 1 the principle of the semaphores construction which contains of the two main primitives: wait and signal. Each resource can be assigned to the fixed semaphore. It means that obtaining already blocked semaphore is not allowed until the targeted resource is available. The processors communicate during their execution using RTEMS Message Manager. Event and Signal Managers are also used to synchronize an access to resources in RTEMS. In this case, the blocking processors wait for the special signal or event from the processors which hold the corresponding resource. In the general case, two concepts called spinning and suspending can be used for the implementation of the wait primitive.

Table 4.1 depicts the main characteristics of these types blockings implementation.

| | Suspending | Spinning |
|---|---|---|
| Concept | If the resource is not available, the task is added to the waiting queue.<br>The processor can run during blocking time another task. | If an exclusive resource is not available, the task uses busy waiting to get a resource. |
| Use case | The task can be put to sleep. There will be significant time before the task can access a resource. | The task has to be always active. The resource will be available in reasonably short time. |
| Cons | The tasks context switch and scheduling cost. | The wasting of CPU cycles when the processor uses busy waiting. |

**Table 4.1:** The types of blocking

We need to implement the suspending locking and priority waiting queue for the MPCP. In RTEMS, there are two variants to order task in the waiting queues based on the FIFO and the priority aspects. We can configure a new semaphore as _Thread_queue_Operations _FIFO, _Thread_queue_Operations _priority_inherit, or _Thread_queue_Operations _Priority. The first two configurations define the corresponding waiting queues based on the FIFO ordering. The MPCP has to be configured with the variant where we use the priority waiting queue. The standard operations with the waiting queues are implemented in the file src/threadqops.c.

Futhermore, there are several important attributes which characterize the execution in the critical section under the MPCP. At first, the priority of the tasks waiting for a resource has to be raised to the ceiling priority of resource. Also we need to guarantee that priority of the critical section is always higher than the priority of the normal execution despite of the nominal tasks priorities. The importatnt issue that we consider about is how to implement the suspending locks and how to protect the highest priority for the critical sections. We allocate the blocked tasks into the priority based waiting queue that is assigned to the each resource. We have used for it such function from the thread handler as _Thread_queue_Enqueue that provides the insertion to the waiting queue without the busy waiting. For another, we add to the Thread_Control structure the new attribute called boost. This attribute is set to 1 when the task obtains a resource and set to 0 by releasing of it. Also we have modified the function _Scheduler_SMP_Enqueue_to_schedule to _Scheduler_SMP_Enqueue_to_scheduled_MPCP. We have implemented the mechanism which disables the normal execution of other tasks when the attribute boost of other tasks within corresponding processor is not equal to 0. After it, the task entering a critical section cannot be preempted by any normal tasks execution. Figure 4.4 presents the flowchart of the tasks which want to access a resource.

At first, each task must check whether the owner of the resource is set to NULL or not. If the owner is NULL, then we raise the priority of the task to the ceiling level. Then we need to set our task as the owner of the resource and start to execute it. In the case when the resource is already used by another task, we raise the priority to the ceiling level and put it to the waiting queue that is ordered by the priority. The prohibition of the nested resource access is implemented as the updating of the resources status as UNAVAILABLE. Such design can ensure that only one task can access a resource with the ceiling priority at the same time. All other tasks will be inserted into waiting queue where the tasks are suspended until the receiving of a signal.

The directive rtems_semaphore_obtain is used to acquire the corresponding semaphore. Figure 4.5 depicts the UML sequence diagram which presents the sequence of the functions starting from the application calling rtems_semaphore_obtain directive.

The semaphore calls the function get_owner, which receives the owner of the resource and calls the thread handler to add the corresponding thread to the priority node. The thread calls then Update_priority to provide the changes in the scheduler. The alt label defines if/else operator. When the owner of the resource is equal to NULL, the thread is set

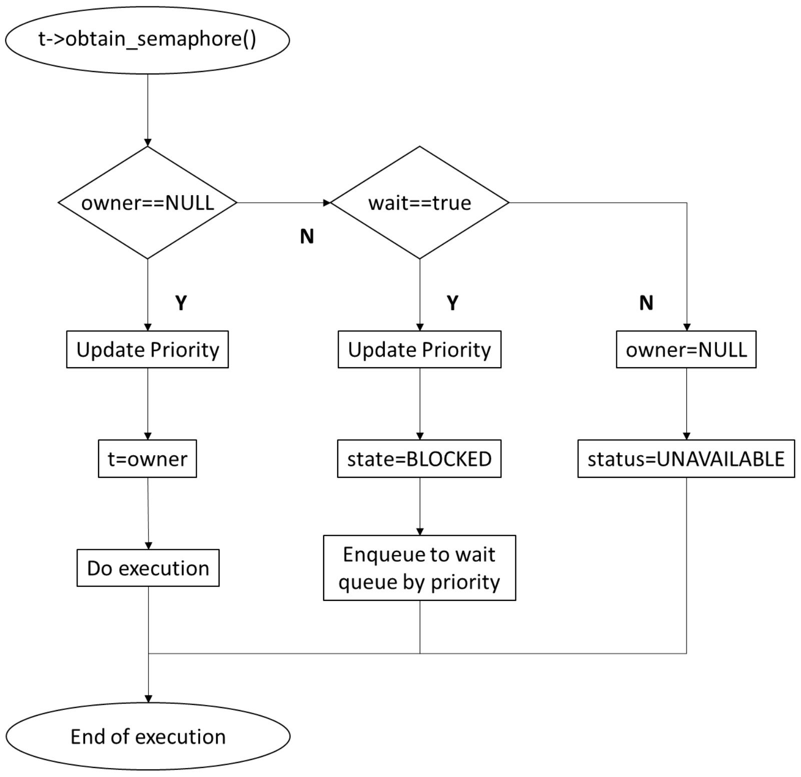


**Figure 4.4:** The flowchart of critical section under MPCP

as the owner by the semaphore handler. The thread executes the function Dispatch_thread and changes the state of the thread to EXECUTE. If the resource is already used by another thread, the blocked thread is inserted in the waiting queue by the thread queue handler. The semaphore calls the Thread Queue handler to protect the forbidden of the nested resource access. When the executing task calls the directive rtems_directive_release the semaphore dequeues the first task from the waiting queue and adds it in the ready chain. Then the priority of the calling task is returned to the nominal priority. Figure 4.6 presents the UML sequence diagram which shows the functions sequence starting from the application calling rtems_semaphore_release directive.

After the application executes rtems_semaphore_release, the semaphore calls the function, which defines the actual owner of the resource and changes its priority from the ceiling to the nominal level. As the next the Thread Queue handler returns the next task in the queue which is ready to execute. The semaphore executes the extract_critical function from the Thread Queue Handler which calls the function Unblock to the scheduler. As the result, the state of this task is changed from the BLOCKED to EXECUTE and the header task starts its execution.

According to the MPCP theory that was discussed above in Chapter 2, the tasks priority is raised to the ceiling priority after the task obtains a semaphore. This feature is provided by calling MPCP_Raise_priority function which increases the real priority level of the

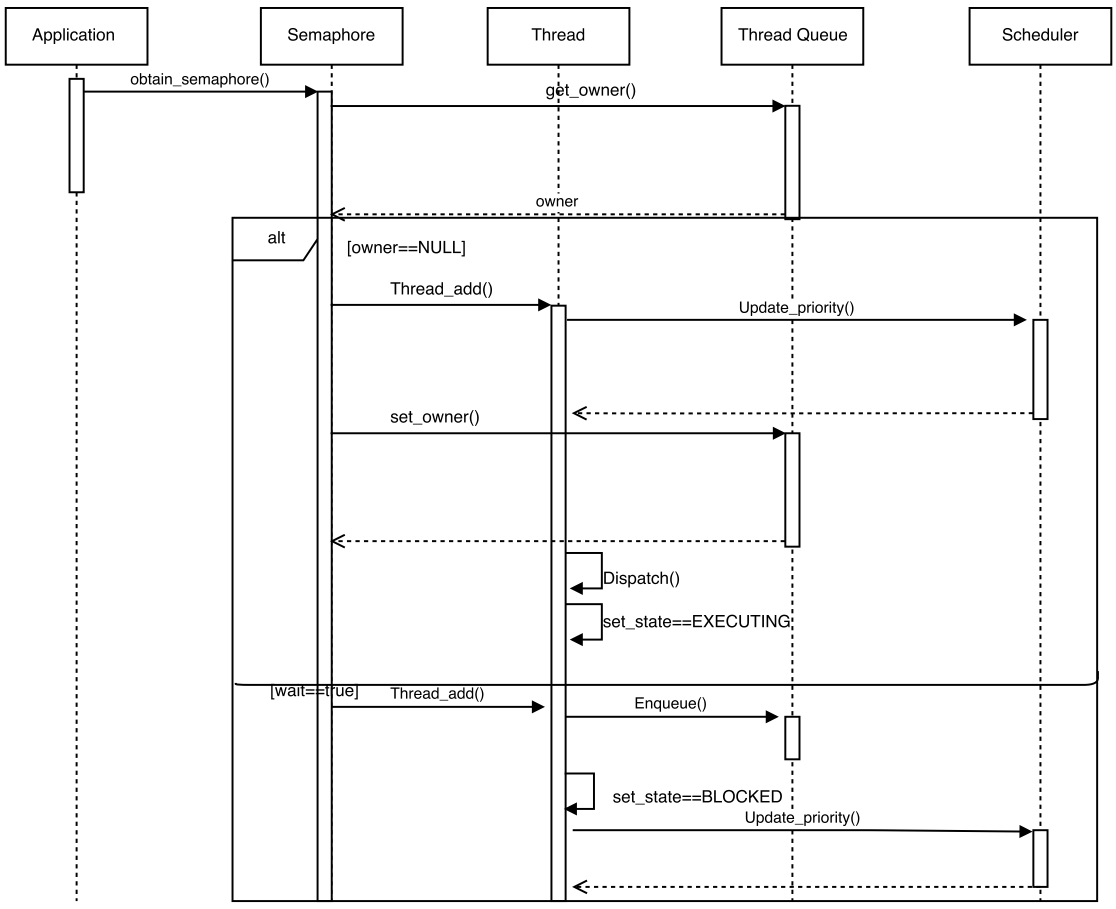


**Figure 4.5:** UML sequence diagram rtems_semaphore_obtain method

calling task to the ceiling priority. The ceiling priority is defined for the each semaphore after the execution of the directive rtems_semaphore_set_priority.

We have discussed that the tasks migration is forbidden in the MPCP scheduler. The assignment of the tasks on the processors is executed with the directive task_set_sched uler, where are defined the task, corresponding processor and the nominal priority of the task. In the application we have to connect the schedulers and the processors with the directive rtems_scheduler_ident.

Figure 4.7 depicts the UML sequence diagram which presents the sequence of the functions starting from the application calling task_set_scheduler directive.

The application calls the function task_set_scheduler, which assigns the tasks to the processors and calls the thread handler to add the corresponding thread to the scheduler. The thread calls Scheduler_Block and Scheduler_Unblock to provide the changes in the scheduler. These functions are executed only for the tasks in the READY state. The Scheduler calls the function Chain_Extract_Unprotected and Chain_Initialize to set the new scheduler which must be not equal to the previous one.

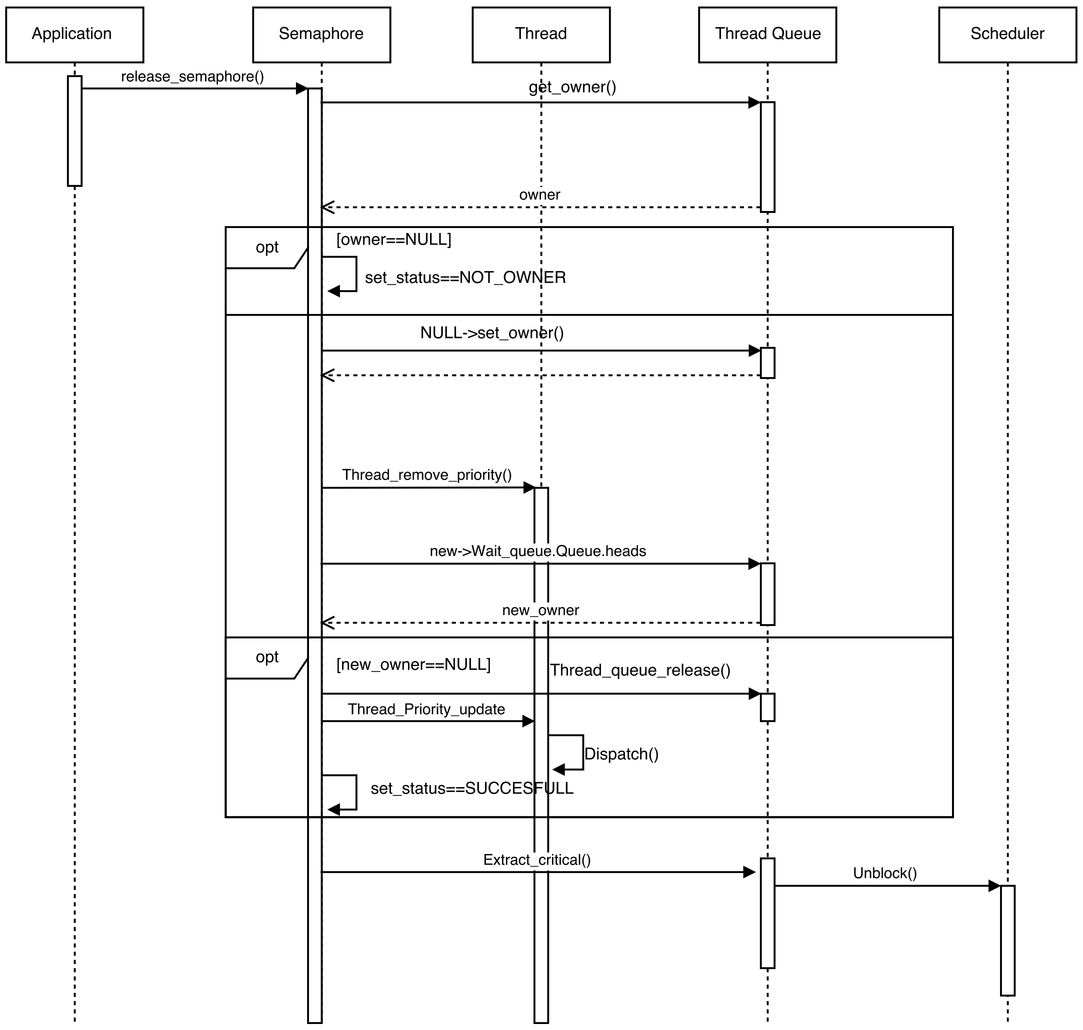


**Figure 4.6:** UML sequence diagram rtems_semaphore_release method

The directive task_set_scheduler can be used also to provide a new tasks priority. The scheduler runs the function Scheduler_node_set_priority to set up the new priority.

After the MPCPs implementation in the header files score/mpcpimpl.h and score/mpcp.h we need to configure the new semaphore. MPCP is configured as follows:

```
RTEMS_BINARY_SEMAPHORE|RTEMS_PRIORITY|RTEMS_MULTIPROCESSOR_PRIORITY_CEILING
```

It means that we use the semaphore to access a single resource where the tasks are ordered by priority in the waiting queue. The attribute RTEMS_MULTIPROCESSOR _PRIORITY_CEILING is connected with the functions from score/mpcpimpl.h. In general, the semaphores attributes are connected with each other by a bitwise OR operator. It is important to define a new attribute in the header file rtems/attr.h correctly. We set the MPCP attribute as follows:

```
#define RTEMS_MULTIPROCESSOR_PRIORITY_CEILING 0x00000072
```

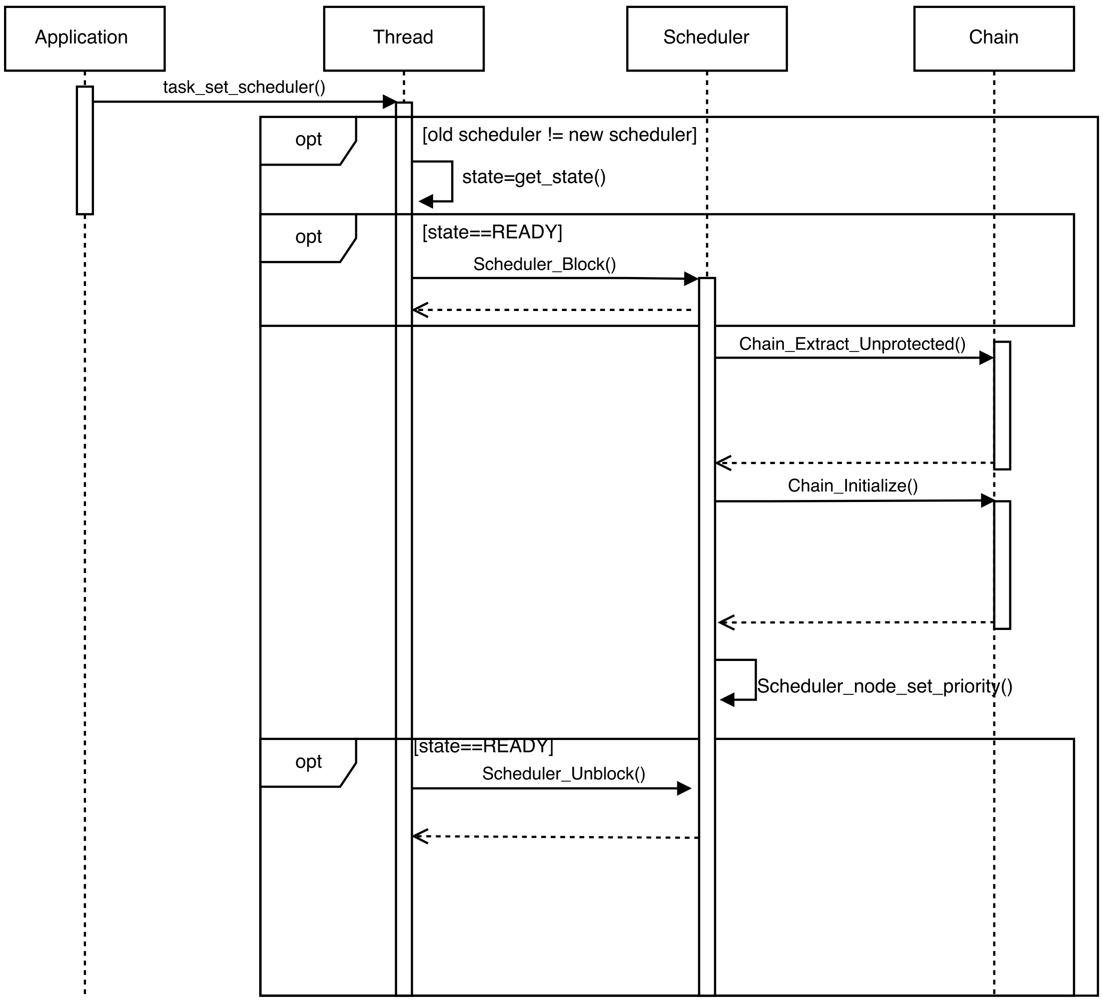


**Figure 4.7:** UML sequence diagram set_scheduler method

After the determination of all attributes we have check the identity of the implemented protocol, i.e. there are any already existing objects with the equal check sum. For example, if we define RTEMS_MULTIPROCESSOR _PRIORITY_CEILING attribute as 0x00000068 the check sums of the MPCP and the PCP will be equal through the initialization of the attribute RTEMS_PRIORITY_CEILING as 0x00000080. Such conflicts problem has the significant impact to both protocols execution and should be avoided by definition of the new attributes. The configuration for directive rtems_semaphore _create is provided in the rtems/src/semcreate.c. We connect the calling rtems_semaphore _create in API with the core function _MPCP_Initialize as the following [9]:

```
case SEMAPHORE_VARIANT_MPCP:
scheduler = _Thread_Scheduler_get_home (executing);
priority = _RTEMS_Priority_To_core ( scheduler , priority_ceiling , &valid );
if (valid){
status = _MPCP_Initialize(
&the_semaphore->Core_control.MPCP, scheduler, priority, executing, count ==0
);
}
else{
status = STATUS_INVALID_PRIORITY;
}
break;
```

The direktives rtems_semaphore_obtain, rtems_semaphore_release, rtems_semaphore _delete, rtems_semaphore_set_priority, and rtems_semaphore_flush are connected with the corresponding MPCP functions by the same way.

The correct configuration of the semaphore is the key point to the protocols execution. Unfortunately, RTEMS provides no document about the configuration of new semaphore. We refer to the design of the MrsP to configure the MPCP. The corresponding changes have to be done for each semaphore in the following kernel files [9]:

- rtems/src/: semcreate.c, semobtain.c, semrelease.c, semflush.c, semdelete.c, semset-priority.c
- sapi/include/confdefs.h
- libmisc/monitor/mon-sema.c
- rtems/include/rtems/rtems/semimpl.h, sem.h
- rtems/include/rtems/rtems/attr.h, attrimpl.h.

We need to verify that the protocol works as expected. The verification of the MPCP will be described in next Chapter.

## 4.4 Challenges in the Implementation of the DPCP

As we have discussed in Chapter 2, the DPCP uses the PCP as the base for the execution on each processor. It means that we can use the MPCP to implement the DPCP. The difference between these two protocols is the migration of the critical section to the specific processor, called synchronization processor [4]. The most important issue that we have is to implement such migrations mechanism dynamically. The new API directive rtems_task_create_dpcp with the special attribute scheduler_id should be created. The corresponding task is connected with the processor by the function _Scheduler_Get_by_id. To date, RTEMS initializes the threads statically. It means, that all threads are assigned to the processor with the index 0 automatically by their creation (see cpukit/score/src/threadinitialize.c [9]). So we have to change the files which are connected with the Thread handler. Due to the limited time execution for the bachelor thesis, we implement the DPCP statically. The development of the corresponding directives for the dynamic execution can be considered as the future work.

## 4.5 Implementation of the DPCP

We have discussed the principles of the DPCP in Chapter 2. Based on that, we have to implement the migration mechanism, which is provided for the execution of the critical sections on the synchronization processor. The type of the tasks blocking and the ordering within the waiting queues are equal to the corresponding features in the MPCP. The migration mechanism for the DPCP includes two parts. The first one is provided for the tasks migration to the synchronization processor. Then we have to consider the mechanism to migrate back on the application processor after the execution in the critical section. When the tasks calls the directive rtems_obtain_semaphore, the corresponding task migrates to the synchronization processor with the function _DPCP_Migrate, which assigns the task to the specific CPU with calling of the directive _Thread_Set_CPU.

```
RTEMS_INLINE_ROUTINE void _DPCP_Migrate(
Thread_Control         *executing,
Per_CPU_Control           *cpu
)
{
  _Thread_Set_CPU(executing,cpu);
}
```

The tasks migrate to the synchronization processor only in the cases when the current owner of the resource is NULL or the resource is obtained by other task at the moment. It means, that the tasks are in the states EXECUTING or BLOCKED on the synchronization processor. So we have modified the code [9] of the function _Size which is connected with rtems_semaphore_obtain directive as follows:

```
RTEMS_INLINE_ROUTINE Status_Control _DPCP_Seize(
DPCP_Control            *dpcp,
Thread_Control          *executing,
bool                     wait,
Thread_queue_Context *queue_context
)
{
  const Scheduler_Control *scheduler;
  Status_Control  status;
  Thread_Control *owner;
  Scheduler_Node              *scheduler_node;
  Per_CPU_Control             *cpu_semaphore = _Per_CPU_Get_by_index(1);

  _DPCP_Acquire_critical(dpcp, queue_context );
    owner = _DPCP_Get_owner(dpcp);

  if ( owner == NULL )
  {
    status = _DPCP_Set_new( dpcp, executing, queue_context );
    _DPCP_Migrate(executing , cpu_semaphore);
  }
  else if (owner == executing)
  {
    _DPCP_Release (dpcp, queue_context);
    status = STATUS_UNAVAILABLE;
  }
  else if ( wait )
  {
    status = _DPCP_Wait(dpcp, DPCP_TQ_OPERATIONS, executing , queue_context );
    _DPCP_Migrate(executing ,cpu_semaphore);
  }
  else
  {
    _DPCP_Release( dpcp, queue_context );
    status = STATUS_UNAVAILABLE;
  }
    return status;
  }
```

The function _DPCP_Surrender which is connected with rtems_semaphore_release directive is modified by the same way. We define the processor assigned by the tasks creation as the application processor and execute the function _DPCP_Migrate. We have created a new scheduler Scheduler_Priority_DPCP, which uses the functions from Scheduler_Priority_MPCP with the exception of boosting function. We have discussed in Section 2.3 that the normal execution of the higher priority tasks cannot be preempted by any critical sections of other tasks [4]. It means that the boosting function is redundant for the DPCP.

For the configuration of the DPCP will be applied the similar way described in Section 4.3 After the implementation of the new protocols in RTEMS core we have to verify their correct behavior. We describe the tests execution for the DPCP in detail in next Chapter.

## 4.6 Implementation of the DNPP

As we have discussed in Section 2.4, the DNPP works similar to the DPCP. The resources assigned to the specific synchronization processors. Meanwhile, the normal execution is running on the application processors, where the tasks order by priority.

To implement the DNPP we concentrate only on non-preemption part during execution in the critical section. It means, that the priority of the tasks executing critical section has to be raised to the highest priority level in the system. After releasing of the resource the task is running with its nominal priority. We implement a new function _DNPP_Set_Ceiling, which increases the ceiling priority to the 1 (that is the highest priority level in RTEMS) when the task enters a critical section.

```
RTEMS_INLINE_ROUTINE void _DNPP_Set_Ceiling(
DNPP_Control            *dnpp,
 const Scheduler_Control *scheduler
)
{
 uint32_t scheduler_index;

 scheduler_index = _Scheduler_Get_index( scheduler );
 dnpp->ceiling_priorities[ scheduler_index ] = 1;
}
```

The function _DNPP_Set_Ceiling is called from the function _DNPP_Seize. It means that the priority of the task is raised to 1 when it obtain a semaphore and moves to the synchronisation processor for the execution. Increasing of priority to the highest level can guarantee that once a task enters a critical section, it can not be preempted by any other tasks. Futhermore, the task is updated with the new priority after adding of the task in the waiting queue in the case when the resource is not available now. It guarantees that the tasks are ordered in the waiting queue according their nominal priorities. Due to the similarities with the DPCP we use Scheduler_Priority_DPCP for the DNPP. After the implementation of DNPP we should configure it in RTEMS. Due to the similarities to the DPCP, the configuration of the DNPP looks as follows:

```
RTEMS_BINARY_SEMAPHORE | RTEMS_PRIORITY |RTEMS_DISTRIBUTED_NO_PREEMPTIV
```

```
#define RTEMS_DISTRIBUTED_NO_PREEMPTIV 0x00000081
```

All other configuration are the same as by the MPCP and the DPCP (see Section 4.3).

After the implementation of the DNPP we need to verify that the protocol works as we need. We will discuss the verification of the specific DNPP features in Section 5.4.

# 5 Verification for New Protocols

The verification of the implemented protocol takes important place in this thesis. It helps to verify whether the protocol executes properly. At first we present the verification for the general routine in Section 5.1. Section 5.2 describes the verification for the MPCP. Then we report on the testing of the DPCP in Section 5.3. Section 5.4 represents the verification for the DNPP. Finally, we describe the overheads measurement in Section 5.5.

## 5.1 Verification for the General Routines

As was presented in Chapter 2, we consider the global resources without nested access. The local resources which are accessed via the PCP protocol on each processor are not the part of this thesis. We execute the test to control of the nested access as follows. The resource is obtained more than once by one task at a time. After it we should get the status RTEMS_UNSATISFIED if the protocol was implemented correctly.

Also we have to consider how to check the prohibition to unblock of all tasks waiting for the same resource. The protocol has to return the status RTEMS_NOT_DEFINED after the execution of the flush directive.

We provide a test case that has to check the protocols behavior by obtaining of the initially locked semaphore. In the test example initially_locked_error we create the semaphore with the count attribute equal to zero. It means that this resource was already obtained by another task which has decremented the count attribute by one, i.e the resource is become available only by the positive value. The protocol has to set the status code to RTEMS_INVALID_NUMBER.

We have implemented the applications smpmpcp01/init.c, smpdpcp01/init.c, and smpdnpp01/init.c which verificate the implemented protocols for the routines described above, respectively the results of the all test cases are successful. The verifications of the specific protocols features are described as the next.

## 5.2 Verification for the MPCP

We have implemented the application mpcp02/init.c that includes such test examples as multiple_obtain, obtain_critical, ceiling_priority, block_critical, and critical_semaphore. The setting of the same starting time for the tasks is necessary condition for several tests execution. The symmetrical tasks execution can be obtained with using rtems_task_suspend and rtems_task_resume directives. The directive rtems_task_suspend blocks the task specified by id. The directive rtems_task_resume removes the corresponding task from the suspended state. The symmetrical execution can be provided by additional using of the rtems_task_wake_up directive, which obtains a sleep timer for the calling task. The sleep timer allows a task to sleep for a corresponding time interval in ticks. The task is blocked by the scheduler for the given interval, after which the task is unblocked. The state of the calling task is changed to BLOCKED and then is returned to READY.

We use Event and User Extensions Managers to track the flow of the protocols execution. The applications for the MrsPs verification provide the mechanism which traces the protocols execution. Each test case includes the functions reset_switch_events and reset_print_events. The first one is used to reset the switch_index argument for each test case. The function reset_print_events has to give out such parameters as cpu index, executing thread, heir thread, heir node, and the priority. The initial extensions are configured as follows [9]:

```
#define CONFIGURE_INITIAL_EXTENSIONS \
{ .thread_switch = switch_extension }, \
RTEMS_TEST_INITIAL_EXTENSION
```

We will use the CPU usage statistic manager to control the executing time of the tasks on each processor. The directives rtems_cpu_usage_report and rtems_cpu_usage_reset are provided to check the report of the processors usage by the tasks. This report contains the following information:

- task id
- task name
- execution time in ticks
- percent's of the execution time for each task

The first feature to verificate is a preemption the critical section by another critical section with the higher ceiling priority. It is implemented in the block_critical example. For this test case we created two tasks and two semaphores with the different ceiling priority. The tasks obtain the semaphores consequentially. If the MPCP was implemented correctly, the critical section with the lower ceiling is blocked. Figure 5.1 shows the expected behavior of the MPCP for this test case.

This application executes correctly for the MPCP. Figure 5.2 presents the report of the CPU usage for the tasks and the execution flow for the test block_critical. This report indicates that the system has two processes due to two IDLE tasks, which are the lowest priority tasks in RTEMS. Thery are executed on the CPU when there are any other tasks ready to run on this processor.

Moreover, we have the initialization task which is configured in the file sapi/include/-confdefs.h as the task with the name UI1 with the priority equal to 1. This task is respon-

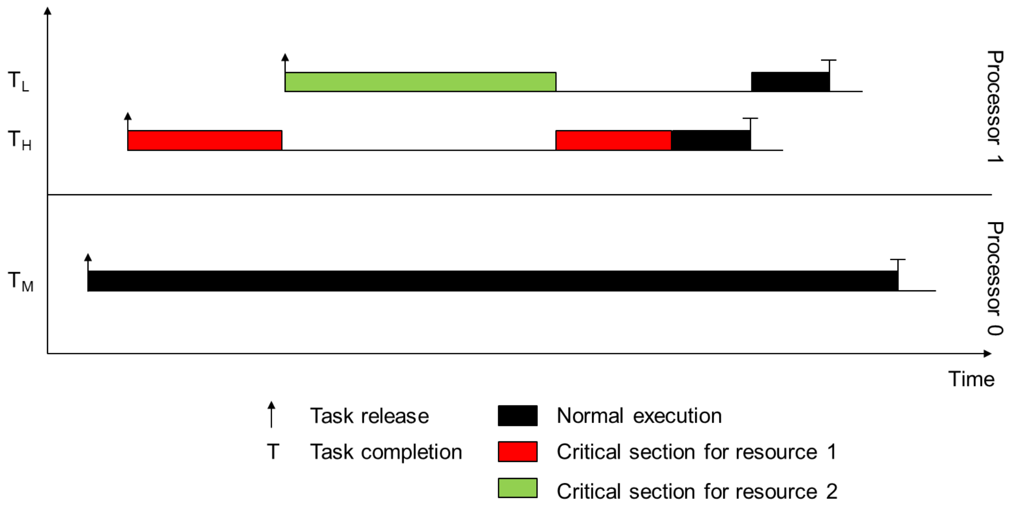


**Figure 5.1:** The timeline of block_critical test for MPCP

sible for the creation of other tasks in the application. The user can change the name and the priority of this task with the following definitions [9]:

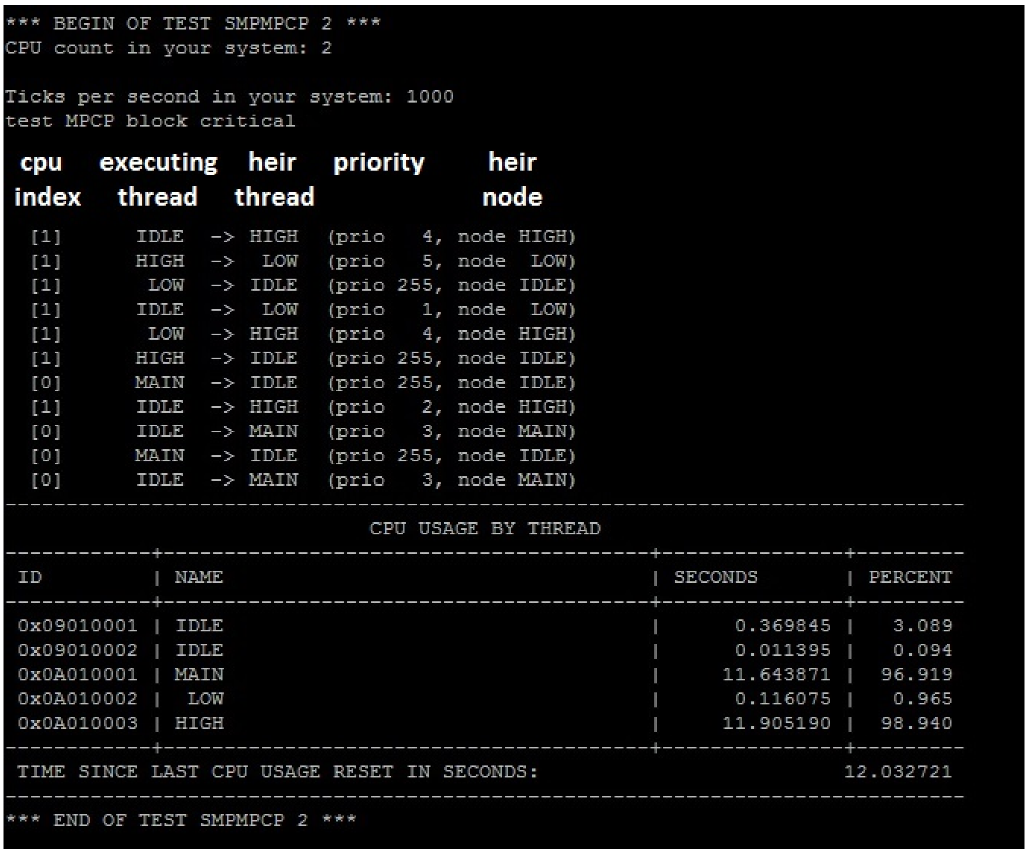
```
#define CONFIGURE_INIT_TASK_NAME
#define CONFIGURE_INIT_TASK_PRIORITY
```

In our test examples we configured the initial task UI1 as the task named MAIN with the nominal priority 3.

In the next tests we present only the flow of tasks execution without CPU usage statistic report.

We have implemented the example critical_semaphore where we test the execution of the tasks which access a resource on two processors. We check the case when these two tasks obtain a semaphore at the same time and the resource is accessed by a task with the higher nominal priority. The test critical_semaphore runs as was expected. The behavior of MPCP is correct for this test case.

Also we have to check the behavior of MPCP for the multiple resources. We create the test function multiple_obtain to control the access to several resources by the tasks on the different processors. The task named "RUN" is set to CPU 1 and acesses a resource. Meanwhile the task "MAIN" accesses two resources on CPU 0. The task "MAIN" obtains the resources with the ceiling priority 1 and 2 consequentially. Figure 5.3 presents the exepted tasks execution.



```
*** BEGIN OF TEST SMPMPCP 2 ***
CPU count in your system: 2

Ticks per second in your system: 1000
test MPCP block critical
 cpu   executing  heir   priority      heir
index   thread   thread                node
 [1]      IDLE  -> HIGH  (prio   4, node HIGH)
 [1]      HIGH  ->  LOW  (prio   5, node  LOW)
 [1]       LOW  -> IDLE  (prio 255, node IDLE)
 [1]      IDLE  ->  LOW  (prio   1, node  LOW)
 [1]       LOW  -> HIGH  (prio   4, node HIGH)
 [1]      HIGH  -> IDLE  (prio 255, node IDLE)
 [0]      MAIN  -> IDLE  (prio 255, node IDLE)
 [1]      IDLE  -> HIGH  (prio   2, node HIGH)
 [0]      IDLE  -> MAIN  (prio   3, node MAIN)
 [0]      MAIN  -> IDLE  (prio 255, node IDLE)
 [0]      IDLE  -> MAIN  (prio   3, node MAIN)
-------------------------------------------------------------------------------
                              CPU USAGE BY THREAD
------------+----------------------------------------+---------------+---------
 ID         | NAME                                   | SECONDS       | PERCENT
------------+----------------------------------------+---------------+---------
 0x09010001 | IDLE                                   |      0.369845 |   3.089
 0x09010002 | IDLE                                   |      0.011395 |   0.094
 0x0A010001 | MAIN                                   |     11.643871 |  96.919
 0x0A010002 |  LOW                                   |      0.116075 |   0.965
 0x0A010003 | HIGH                                   |     11.905190 |  98.940
------------+----------------------------------------+---------------+---------
 TIME SINCE LAST CPU USAGE RESET IN SECONDS:                          12.032721
-------------------------------------------------------------------------------
*** END OF TEST SMPMPCP 2 ***
```

**Figure 5.2:** The output of block_critical test for MPCP

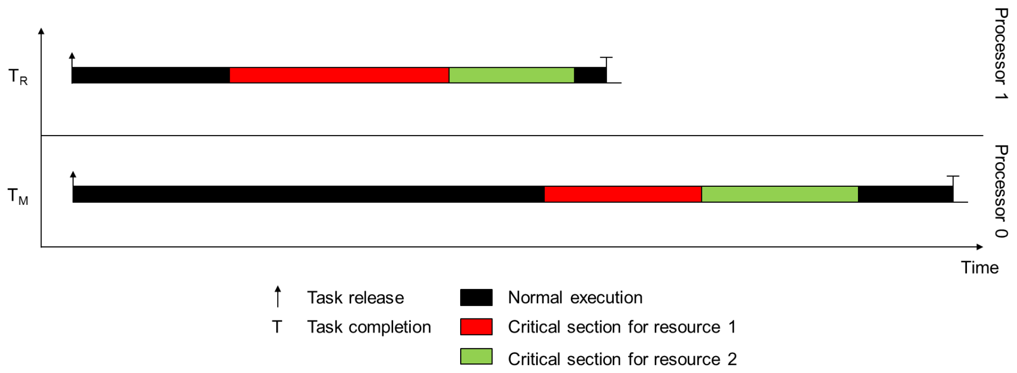
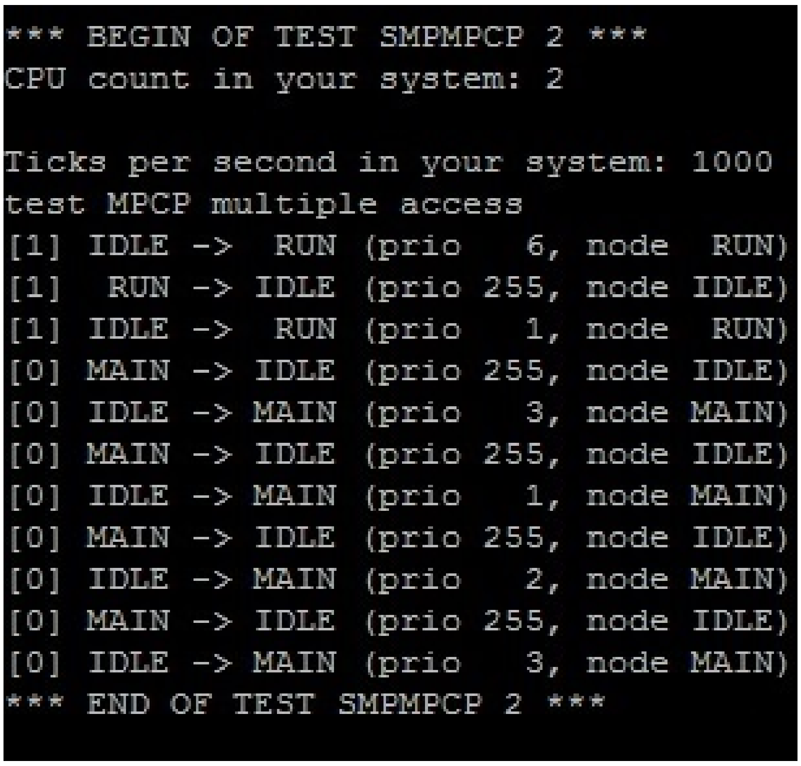


**Figure 5.3:** The timeline of multiple_access test for MPCP

Figure 5.4 presents the results of multiple_obtain tests execution. As the next we check that the critical section has higher priority than any normal execution of the same processor in the function obtain_critical. We have created several tasks with the different priorities and one semaphore. The task with the lower nominal priority obtaines a semaphore that is already updated with the ceiling priority. Meanwhile, the task with the higher priority



```
*** BEGIN OF TEST SMPMPCP 2 ***
CPU count in your system: 2

Ticks per second in your system: 1000
test MPCP multiple access
[1] IDLE ->  RUN (prio   6, node  RUN)
[1]  RUN -> IDLE (prio 255, node IDLE)
[1] IDLE ->  RUN (prio   1, node  RUN)
[0] MAIN -> IDLE (prio 255, node IDLE)
[0] IDLE -> MAIN (prio   3, node MAIN)
[0] MAIN -> IDLE (prio 255, node IDLE)
[0] IDLE -> MAIN (prio   1, node MAIN)
[0] MAIN -> IDLE (prio 255, node IDLE)
[0] IDLE -> MAIN (prio   2, node MAIN)
[0] MAIN -> IDLE (prio 255, node IDLE)
[0] IDLE -> MAIN (prio   3, node MAIN)
*** END OF TEST SMPMPCP 2 ***
```

**Figure 5.4:** The output of multiple_access test for MPCP

is started to execute on the same processor. The task obtaining a semaphore should block the task in the normal execution. Figure 5.5 depicts the exepted tasks execution.

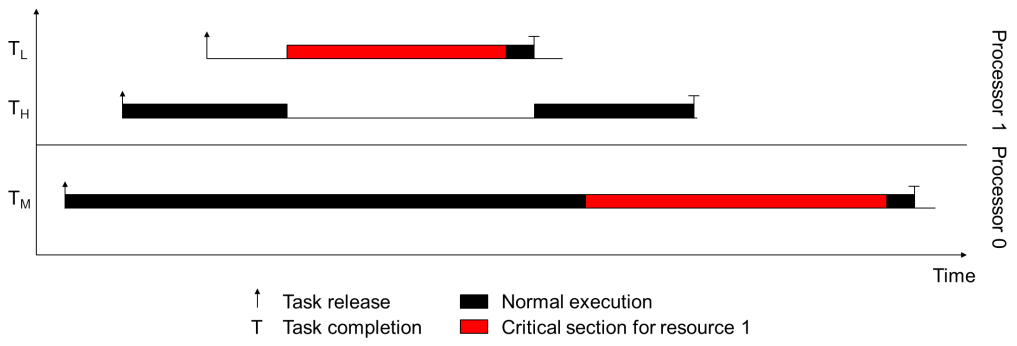
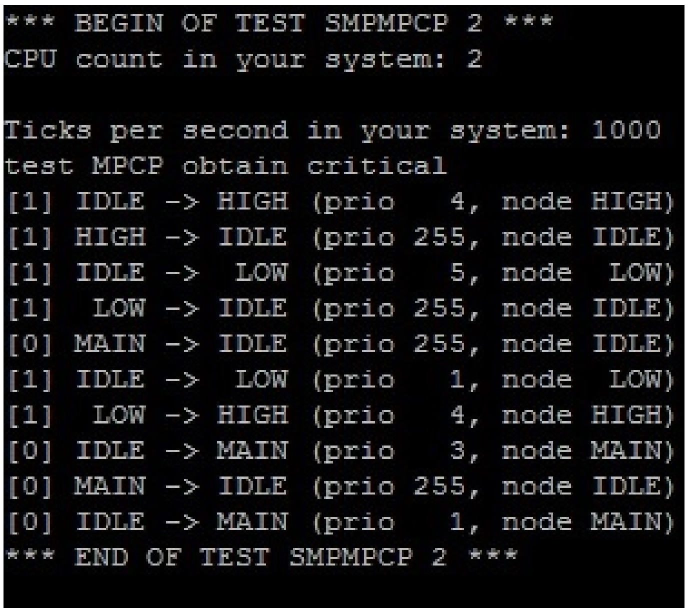


**Figure 5.5:** The timeline of obtain_critical test for MPCP

The results are presented in Figure 5.6. The task called HIGH is blocked during the critical section of task LOW, despite of the higher nominal priority. As the next we will discuss the verification for the DPCP.



```
*** BEGIN OF TEST SMPMPCP 2 ***
CPU count in your system: 2

Ticks per second in your system: 1000
test MPCP obtain critical
[1] IDLE -> HIGH (prio   4, node HIGH)
[1] HIGH -> IDLE (prio 255, node IDLE)
[1] IDLE ->  LOW (prio   5, node  LOW)
[1]  LOW -> IDLE (prio 255, node IDLE)
[0] MAIN -> IDLE (prio 255, node IDLE)
[1] IDLE ->  LOW (prio   1, node  LOW)
[1]  LOW -> HIGH (prio   4, node HIGH)
[0] IDLE -> MAIN (prio   3, node MAIN)
[0] MAIN -> IDLE (prio 255, node IDLE)
[0] IDLE -> MAIN (prio   1, node MAIN)
*** END OF TEST SMPMPCP 2 ***
```

**Figure 5.6:** The output of obtain_critical test for MPCP

## 5.3 Verification for the DPCP

Because the behavior of the DPCP on the each core is the same as by the PCP (as was discussed in Section 4.5), we can concentrate only on the specific features of the DPCP. We have implemented two applications to test the DPCP. The application called test_normal checks the execution of two tasks with the different priorities. One of the tasks obtains the resource and migrates on processor 1, which is defined as synchronization processor. After the execution in the critical section, the task returns to the application processor. Meanwhile, the second task runs on the application processor, due to the exclusively normal execution. Figure 5.7 represents the flow of the tasks execution under test_normal.

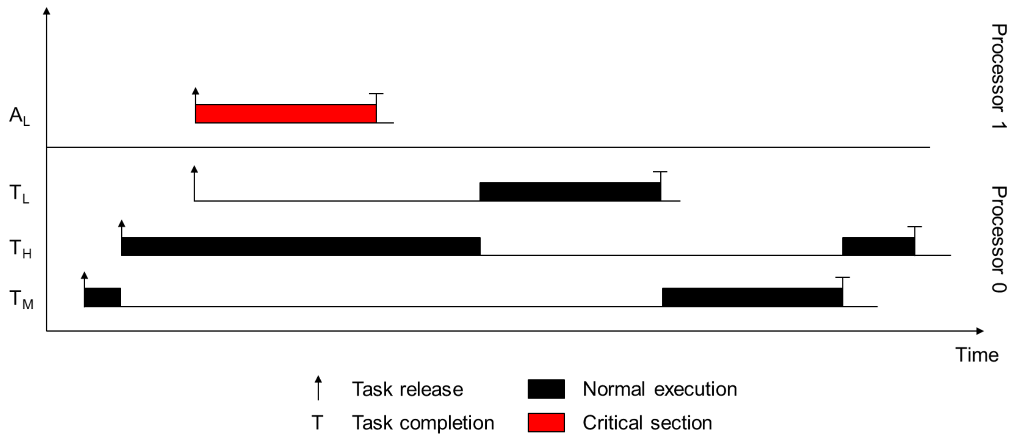
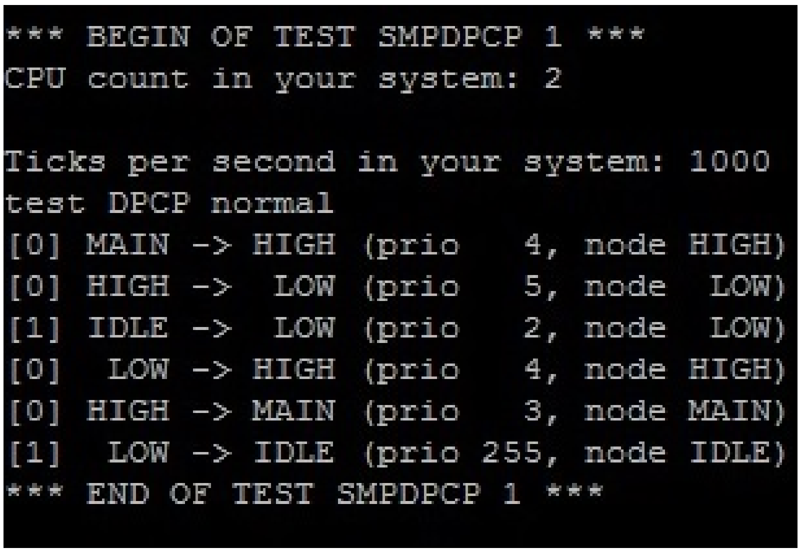


**Figure 5.7:** The timeline of normal_test for DPCP

Figure 5.8 shows the execution of test_normal for the DPCP.



```
*** BEGIN OF TEST SMPDPCP 1 ***
CPU count in your system: 2

Ticks per second in your system: 1000
test DPCP normal
[0] MAIN -> HIGH (prio   4, node HIGH)
[0] HIGH ->  LOW (prio   5, node  LOW)
[1] IDLE ->  LOW (prio   2, node  LOW)
[0]  LOW -> HIGH (prio   4, node HIGH)
[0] HIGH -> MAIN (prio   3, node MAIN)
[1]  LOW -> IDLE (prio 255, node IDLE)
*** END OF TEST SMPDPCP 1 ***
```

**Figure 5.8:** The output of normal_test for DPCP

The second application test_critical is used to test the protocols behavior when two tasks access two resources. The mechanism of the execution on the synchronization processor is the same as on the application processor. It means that the critical section can be preempted other critical section with the higher ceiling level. We have created two tasks with the different priorities. The task called LOW obtains the resource 1 with the higher ceiling than by the resource 2 which is accessed by the task HIGH. The execution of HIGH task on the synchronization processor should be blocked by LOW task, despite of the nominal tasks priorities. The expected tasks execution is represeted in Figure 5.9

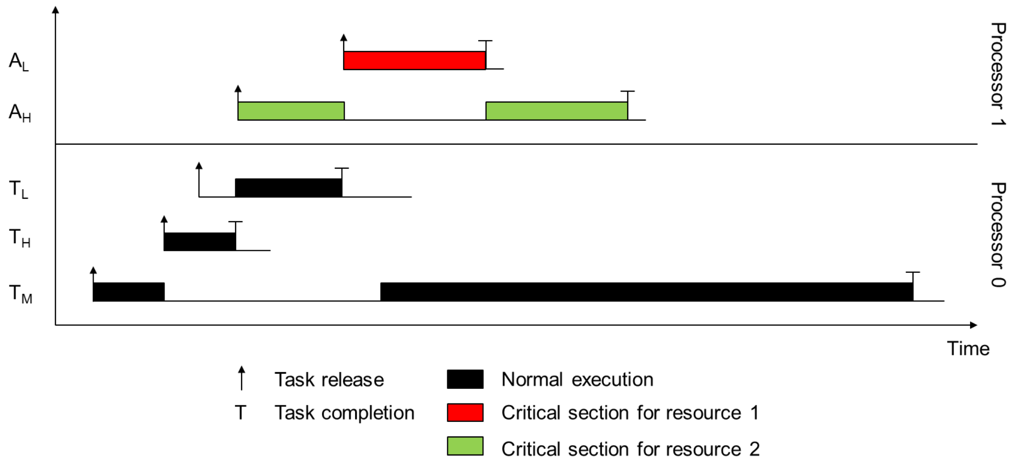
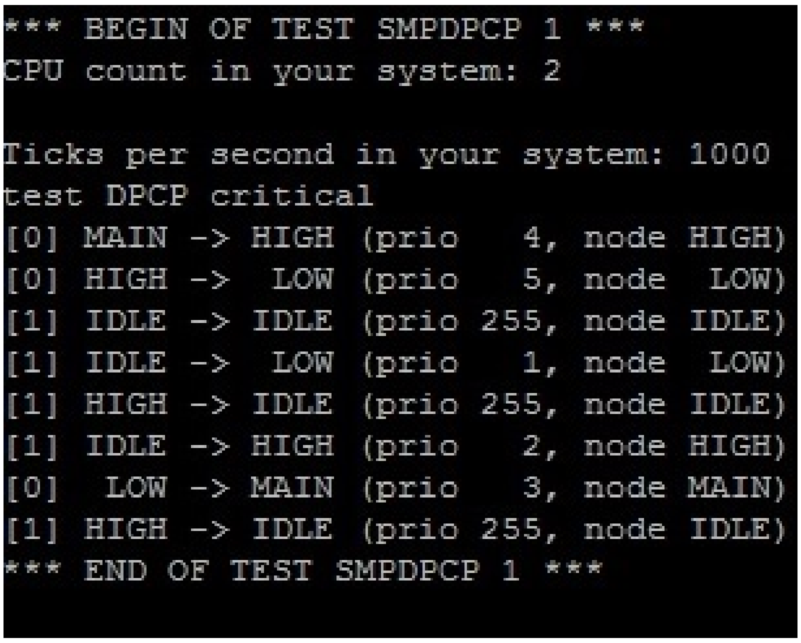


**Figure 5.9:** The timeline of critical_test for DPCP

The results which we have after execution critical_test are presented in Figure 5.10



```
*** BEGIN OF TEST SMPDPCP 1 ***
CPU count in your system: 2

Ticks per second in your system: 1000
test DPCP critical
[0] MAIN -> HIGH (prio   4, node HIGH)
[0] HIGH ->  LOW (prio   5, node  LOW)
[1] IDLE -> IDLE (prio 255, node IDLE)
[1] IDLE ->  LOW (prio   1, node  LOW)
[1] HIGH -> IDLE (prio 255, node IDLE)
[1] IDLE -> HIGH (prio   2, node HIGH)
[0]  LOW -> MAIN (prio   3, node MAIN)
[1] HIGH -> IDLE (prio 255, node IDLE)
*** END OF TEST SMPDPCP 1 ***
```

**Figure 5.10:** The output of critical_test for DPCP

We can notice that the behavior of the DPCP is correct for the test cases described above. The verification for the DNPP will be presented as the next.

## 5.4 Verification for the DNPP

In this Section we concentrate on verification of the specific DNPPs feature, which differs the DNPP from the DPCP. As we have discussed in Section 2.4 the preemprion of task in the critical section is strictly forbidden in the DNPP. Meanwhile, the DPCP allows the critical sections preemption by other critical sections with the higher ceiling priority. We implement application for the verification of this DNPPs feature. Two resources with the different ceiling priorities are obtained by two tasks. The task named LOW which obtains a resource with the lower ceiling priority (equal to 6) can not be preempted by a task accessed a resource with the higher ceiling (equal to 2). The priority of the tasks which enter a critical section are raised to the level equal to 1, that quarantees the non preemption part of the DNPP. The tasks which obtain a semaphore migrate to the synchronization processor (CPU 1). Meanwhile, the normal execution of tasks are run on the apllication processor (CPU 0). It means, that the DNPPs features which cover DPCPs behavior and specific non preemprion part of DNPP are executed correctly. Figure 5.11 shows the DNPPs behavior for the case described above.

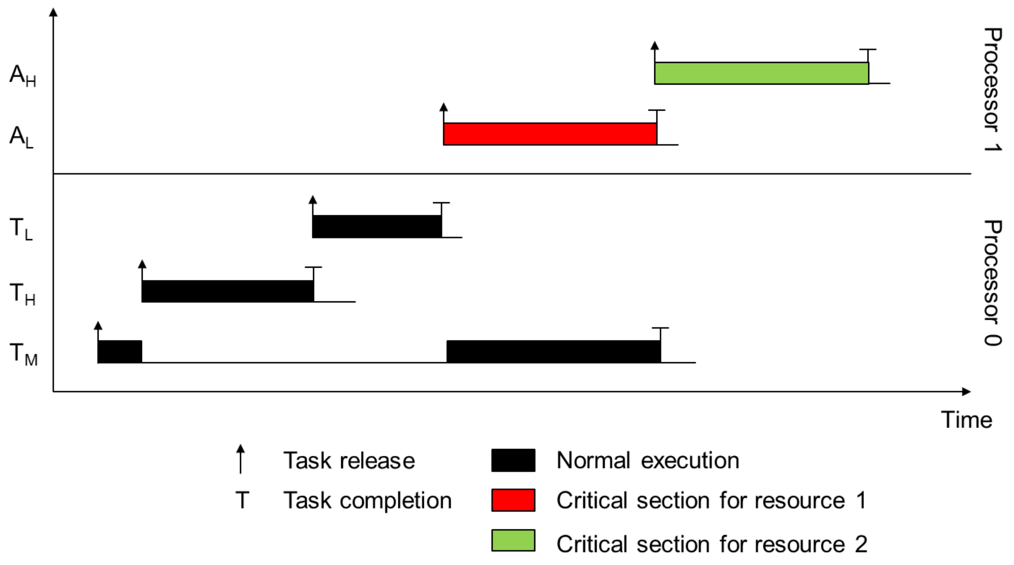
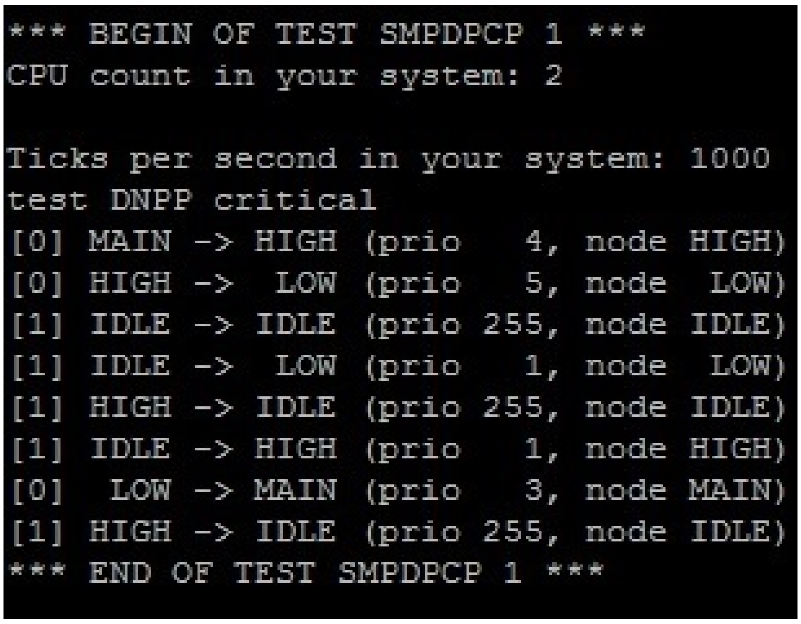


**Figure 5.11:** The timeline of DNPP_critical test

The results of applications execution are presented in Figure 5.12.



```
*** BEGIN OF TEST SMPDPCP 1 ***
CPU count in your system: 2

Ticks per second in your system: 1000
test DNPP critical
[0] MAIN -> HIGH (prio   4, node HIGH)
[0] HIGH ->  LOW (prio   5, node  LOW)
[1] IDLE -> IDLE (prio 255, node IDLE)
[1] IDLE ->  LOW (prio   1, node  LOW)
[1] HIGH -> IDLE (prio 255, node IDLE)
[1] IDLE -> HIGH (prio   1, node HIGH)
[0]  LOW -> MAIN (prio   3, node MAIN)
[1] HIGH -> IDLE (prio 255, node IDLE)
*** END OF TEST SMPDPCP 1 ***
```

**Figure 5.12:** The output of DNPP_critical test

These examples are executed for the DNPP correctly.

## 5.5 Overheads Measurement

In practice, run-time overheads have significant impact on the protocols performance. The processors can consume significant overhead to provide the operations like obtain resource, release resource, tasks migration between processors, etc. We use two directives to measure the protocols performance in RTEMS: rtems_semaphore_release and rtems_semaphore_obtain. Obtaining resource updates the priority of the calling task to the ceiling level and initializes the MCS lock. The procedure of releasing resource returns the tasks priority level to the nominal and updates the resource holder [3].

We have prepared special test called test_mpcp_overhead, which is created to measure the overhead by the execution of rtems_semahore_obtain and rtems_semaphore_release directives. The current tick's value can be returned by using of rtems_clock_get_ticks _since_boot directive. We call it two times to measure ticks value at the start and the end of the corresponding directives execution. This experiment is provided several times defined by the counter. The duration of the ticks in the microseconds is configured for these expreriments as follows [9]:

```
CONFIGURE_MICROSECONDS_PER_TICK 30 // 30 us pro Tick
```

The test will be executed with one exclusive resource for each task. We measure the minimum, maximum, and average of the time that is needed by the task to obtain and to release a resource.

The results for the MPCP are presented in Table 5.1:

| | max. | min. | avg. |
|---|---|---|---|
| obtain procedure | 56 ms | 0,02 ms | 12 ms |
| release procedure | 54 ms | 0,01 ms | 18 ms |

**Table 5.1:** The overhead of obtaining and releasing procedures for MPCP

We provide the same experiments for the MrsP. The results are presented in Table 5.2.

| | max. | min. | avg. |
|---|---|---|---|
| obtain procedure | 65 ms | 0,05 ms | 18 ms |
| release procedure | 72 ms | 0,04 ms | 23 ms |

**Table 5.2:** The overhead of obtaining and releasing procedures for MrsP

Table 5.3 and 5.4 depicts the overheads results for the DPCP and the DNPP respectively.

| | max. | min. | avg. |
|---|---|---|---|
| obtain procedure | 85 ms | 0,03 ms | 21 ms |
| release procedure | 93 ms | 0,03 ms | 34 ms |

**Table 5.3:** The overhead of obtaining and releasing procedures for DPCP

| | max. | min. | avg. |
|---|---|---|---|
| obtain procedure | 92 ms | 0,05 ms | 26 ms |
| release procedure | 102 ms | 0,08 ms | 42 ms |

**Table 5.4:** The overhead of obtaining and releasing procedures for DNPP

From the results shown on the tables, several points can be concluded. Firstly, the MPCP has the lower overheads than the MrsP. Secondly, the higher overhead by the MrsP can be caused by the help mechanism which forces the tasks migration between processors. And finally, the DPCP and the DNPP have the highest overheads due to tasks migration between application and synchronization processors during obtaining and releasing procedures.

# 6 Conclusion

The goal of this thesis is to analyze and implement resource synchronization protocols for multiprocessor system based on RTEMS. We have analyzed the theories of the most known resource synchronization protocols and the implementation of already integrated protocols in RTEMS to find out the way for the new development.

## 6.1 Summary and Future Work

To make the implementation of the resource synchronization protocols better to understand, we introduced in Chapters 2 and 3, the theory of these protocols and the basic knowledges about real-time operation system RTEMS. RTEMS is a competitive and a modern RTOS, which provides the symmetric, asymmetric, and distributed multiprocessing. In Chapter 4 , we focused on the structure and the features of the SMP support in RTEMS. Unfortunately, the work on the SMP features is still in a progress and there is no official RTEMS document which describes the implementation for the SMP in the details. We have to examine the current code of SMP to find out how the protocols and schedulers have to be implemented in RTEMS.

We have analyzed the already implemented protocols for the resource synchronization access for the uni- and multiprocessors. In particular, we have learned the MrsP implementation to find out the way to develop new protocols, the MPCP, the DPCP, and the DNPP.

We have provided the new schedulers to avoid tasks migrations during its execution between processors, priority ordering for the waiting queue, suspending instead of the spinning for the locking, and the mechanism which supports migrations of the critical sections to the specific processor. The new implemented protocols are verified in Chapter 5. We provided various test cases which obtain the general concepts of resource synchronization protocols and the specific features of the corresponding protocols. Also we have measured the overhead of the procedures of obtaining and releasing for each considered protocols.

Overall we can notice that RTEMS is a good choice for the implementation of the resource synchronization protocols for the multiprocessor due to the integrated SMP support. But there are many open areas in the SMP research in RTEMS, in particular the integrated tools for the worst time analysis and system tracing. Also the implementation of new SMP scheduling algorithms is an important evolution for RTEMS [5].

As mentioned in Section 4.4, the development of the DPCP has to be continued, in particular switching from the static to the dynamic execution. The new API directive rtems_task_create_dpcp has to be created, where the user can define the specific processor for the tasks execution.

# Bibliography

[1] Burns, A., Wellings, A.: A schedulability compatible multiprocessor resource sharing protocol - MrsP. *In 25th Euromicro Conference on Real-Time Systems (ECRTS),* pp 282-291, 2013

[2] Brandenburg,B.B., Anderson,J.H..: A comparison of the M-PCP, D-PCP, and FMLP on LITMUS. *In Proceedings of 12th International Conference on Principles of Distributed Systems (OPODIS'08),* pp 105–124, 2008

[3] Catellani,S., Bonato,L., Huber,S., Mezzeti,E.: Challenges in the Implementation of MrsP. *In Reliable Software Technologies – Ada-Europe,* pp 179-195, 2015 2016

[4] Huang,W.-H., Yang, M., Chen,J.-J.: Resource-Oriented Partioned Scheduling in Multiprocessor Systems: How to Partion and How to Share? *In Real-Time Systems Symposium (RTSS),* 2016

[5] Bloom,G., Sherrill J.: Scheduling and Thread Management with RTEMS. *ACM Digital library,* 2014

[6] Linux testbed for multiprocessor scheduling in real-time systems (LITMUSRT). *https://www.litmus-rt.org*

[7] Gracioli, G.: Real-time operating system support for multicore applications. *Doctor's thesis, Federal University of Santa Catarina, Florianópolis, Brasil,* 2014

[8] RTEMS: Real-Time executive for multiprocessor systems. *http://www.rtems.com/*

[9] rtems.git. *http://git.rtems.org/rtems/*

[10] Benes, P. Porting of resource reservation framework to RTEMS executive. *Master's thesis, Czech Technical University, Prague, Czech Republic,* 2011

[11] RTEMS documentation sets - RTEMS Applications C User's Guide. *http://www.rtems.org/onlinedocs.html*

[12] Buttazzo, G.: Hard real-time computing systems: predictable scheduling algorithms and applications, volume 24. *Springer Science & Business Media,* 2011

[13] Bastoni, A., Brandenburg, B., B., Anderson, J., H..: An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. *In Real-Time Systems Symposium (RTSS),* pp 14-24, 2010

[14] Marwedel,P.: Embedded System Design. *Springer,* 2011

[15] Molnar, M.: The EDF scheduler implementation in RTEMS Operating System. *Diplom's thesis, Czech Technical University, Prague, Czech Republic,* 2006

[16] Krutwig, A., Huber, S.: RTEMS SMP Status Report. *embedded brains GmbH,* 2015

[17] Bastoni, A., Brandenburg B. B., Anderson, J. H.: Is semi-partitioned scheduling practical? *In 2011 23rd Euromicro Conference on Real-Time Systems,* pp.125-135. IEEE, 2011

[18] Sha, L., Rajkumar, R., Lehoczky, J.: Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers, 39(9),* pp.1175-1185, 1990

[19] Rajkumar, R.: Real-time synchronization protocols for shared memory multiprocessors. *In Distributed Computing Systems, 1990. Proceedings., 10th International Conference on,* pp. 116-123. IEEE, 1990

[20] Rajkumar, R., Sha, L., Lehoczky, J.,P.: Real-time synchronization protocols for multiprocessors. *In Real-Time Systems Symposium (RTSS),* pp 259–269, 1988

[21] Baker, T.: Stack-based scheduling of realtime processes. *Jornal of Real-Time Systems,3(1),* 1991

[22] Wonneberger, S.: Kopplung von Scheduling- und Speicherallokationsstrategien zur Energieverbrauchsminimierung. *Diplom's thesis, Technical University Dortmund , Dortmund, Germany,* 2008

## Erklärung

Ich versichere, dass ich diese wissenschaftliche Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder Sinn nach entnommen sind, wurden in jedem einzelnen Fall durch Angabe der Quelle als Entlehnung kenntlich gemacht. Das gleiche gilt auch für beigegebene Skizzen und Darstellungen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Dortmund, den 29. Mai 2017

## Einwilligung

Hiermit erkläre ich mich damit einverstanden, dass diese wissenschaftliche Arbeit nach den Bestimmungen des §6 Absatz 1 des Gesetzes über Urheberrecht vom 9.9.1965 in die Bereichsbibliothek aufgenommen und damit für Leser der Bibliothek öffentlich zugänglich gemacht wird.

Ferner bin ich damit einverstanden, dass gemäß §54 Absatz 1 Satz 1 dieses Gesetzes Leser zu persönlichen wissenschaftlichen Zwecken Kopien aus der Arbeit anfertigen dürfen.

Dortmund, den 29. Mai 2017

# Eidesstattliche Versicherung

| | |
|---|---|
| Name, Vorname | Matr.-Nr. |

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

| | |
|---|---|
| Ort, Datum | Unterschrift |

*Nichtzutreffendes bitte streichen

**Belehrung:**

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG - )

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

| | |
|---|---|
| Ort, Datum | Unterschrift |