Supporting single and multi-core resource access protocols on object-oriented RTOSes

Lucas Matheus dos Santos, Giovani Gracioli, Tomasz Kloda, Marco Caccamo

▶ **To cite this version:**

Lucas Matheus dos Santos, Giovani Gracioli, Tomasz Kloda, Marco Caccamo. Supporting single and multi-core resource access protocols on object-oriented RTOSes. Design Automation for Embedded Systems, 2023, 27 (1-2), pp.31-50. 10.1007/s10617-023-09268-6 . hal-04803541

HAL Id: hal-04803541

https://laas.hal.science/hal-04803541v1

Submitted on 25 Nov 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Design Automation for Embedded Systems manuscript No.
(will be inserted by the editor)

Supporting Single and Multi-Core Resource Access Protocols on Object-Oriented RTOSes

Lucas Matheus dos Santos[†] · **Giovani Gracioli**[†] · **Tomasz Kloda**[‡] · **Marco Caccamo**[§]

Received: date / Accepted: date

Abstract Real-time resource access protocols are fundamental to bound the maximum delay a task can suffer due to priority inversions. Several real-time protocols have been proposed, for both static and dynamic scheduling approaches in single and multi-core processors. One of the main factors for performance efficiency in such protocols is the way they are implement within a real-time operating system (RTOS).

In this paper we present an object-oriented design of real-time access protocols considering single and multi-core systems and also suspension- and spin-based protocols (7 protocols in total). Our design aims at reducing the run-time overhead and increasing code re-usability. By implementing the proposed design in an RTOS and running the protocols in a modern multi-core processor, we provide an analysis regarding the memory footprint, run-time overhead, and the impact of the overhead into the schedulability analysis of synthetically generated task sets. Our results indicate that proper implementation provides

Giovani Gracioli was partially supported by Fundação de Desenvolvimento da Pesquisa - Fundep Rota 2030/Linha V 27192.02.01/2020.09-00. Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research.

This paper was originally published in the 2020 X Brazilian Symposium on Computing Systems Engineering (SBESC) [1].

Authors contact
[†] *Software/Hardware Integration Lab - Federal University of Santa Catarina*
Florianópolis, Brazil
{lucasm,giovani}@lisha.ufsc.br

[‡] *LAAS-CNRS, Université de Toulouse, INSA*
Toulouse, France
tkloda@laas.fr

[§] *Technical University of Munich*
Munich, Germany
mcaccamo@tum.de

low run-time overhead (up to 6.1 µs) and impact on the schedulability of real-time tasks.

Keywords Real-time resource access protocols · real-time operating systems · priority ceiling protocol · priority inheritance protocol · stack resource policy · MrsP

1 Introduction

Any concurrent operating system (OS) must offer protocols to control the access to resources shared (*i.e.,* a piece of code, such as data structures, I/O devices, buffers, and so on) by competing tasks, thus ensuring *mutual exclusion* in their respective critical sections (a code segment where shared variables can be accessed) [2,3]. Typically, mutual exclusion is guaranteed by the use of binary semaphores, such as mutexes and suspension-based locks [4]. Therefore, a task willing to enter a critical section must wait until another task, which holds the resource, exits the critical section. This is accomplished by calling the semaphore operations p (or wait) and v (or signal) before entering and after leaving each critical section, respectively.

Real-time embedded applications also use semaphores to synchronize access to shared resources. However, specific resource access protocols are required to avoid unbounded priority inversions [2–4]. For instance, consider a high-priority task τ_1 and a low-priority task τ_3 that share a resource. It might happen that τ_3 is holding the resource, but is preempted by τ_1, as demonstrated by Figure 1. Then, τ_1 executes until it tries to enter the critical section. However, τ_1 cannot continue, because the *shared* resource is already in use by τ_3. Thus, a high-priority task is blocked by a low-priority task. Worse, if a task τ_2 with a priority higher than τ_3 but lower than τ_1 is released and allowed to preempt τ_3 (as at the instant t_4 in Figure 1), the blocking delay can be unpredictable (for instance, other medium-priority tasks could execute).

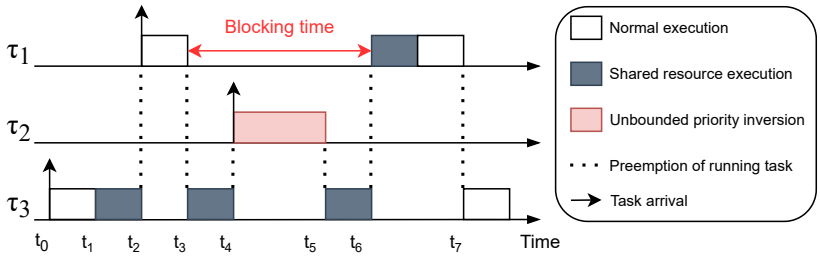


Fig. 1: Example of the priority inversion problem.

Priority inversion has caused many problems in real applications. The 1997 Mars Pathfinder mission is a well-known case. After landing on Mars, the

Pathfinder reset several times and experienced significant delays in capturing scientific data [5] due to the priority inversion caused by an uncontrolled bus sharing between high-, medium- and low-priority tasks.

Several real-time resource protocols have been proposed to bound priority inversion and consider the blocking time when performing schedulability analyses. For uniprocessor systems, a set of mature protocols have been proposed and studied, such as the ceiling- and priority inheritance-based protocols [6], typically used in static scheduling, and the Stack Resource Policy (SRP) [7] for both static- and dynamic-based schedulers. The bounded blocking time ensured by those protocols are accounted for in standard schedulability tests, as the Response Time Analysis (RTA) [8].

When it comes to a multiprocessor system, these protocols cannot be applied directly, because several resource access requests can be issued simultaneously from different cores[1]. Thus, multiprocessor resource sharing protocols, extending the uniprocessor ones, have been proposed, such as the Multiprocessor Priority Ceiling Protocol (MPCP) [9] and the Multiprocessor Stack Resource Policy (MSRP) [10]. Both MPCP and MSRP assume a restricted resource-accessing model, in which nested accesses to shared resources are not allowed [11]. Later on, protocols exclusively proposed for multiprocessor systems, such as the Multiprocessor resource sharing Protocol (MrsP) [12], assume a more flexible resource-accessing model with nested resources allowed and deadlocks avoided [12–14]. Moreover, there is still no consensus regarding the best approach for multiprocessor systems [14, 15], mainly because the characteristics of an application, like the length of critical sections, affect the performance of such protocols [16].

One of the main challenges for any real-time OS (RTOS) is provide support for all such variations of single- and multi-core resource access protocols. Although most RTOSes provide at least one real-time resource access protocol, there is no design and implementation discussion for both single- and multi-core protocols that allow easy extensions, code reuse, and low run-time overhead. The benefits of supporting both single- and multi-core protocols in the same RTOS are: (i) provide the same interface and easy modifications for the developers; and (ii) provide support to change the processor from a single- to a multi-core one.

Our previous paper was the first work to design and evaluate single- and multi-core variations of real-time resource access protocols in an RTOS designed from scratch, considering both static and dynamic scheduling approaches [1]. The proposed design is based on object-oriented techniques to maximize software reuse and minimize run-time overhead. However, in that work, we provided support only for the well-known single-core protocols – Priority Inheritance Protocol (PIP) [6], Priority Ceiling Protocol (PCP) [6], Immediate Priority Ceiling Protocol (IPCP), and SRP [7] – and the standard multi-core extensions (MPCP and MSRP).

[1] In this paper we use the terms core and processor interchangeably.

In this paper, we extended our previous object-oriented design to also support resource access protocols exclusively proposed for multi-core systems (i.e., the Multiprocessor resource sharing Protocol (MrsP) [12]) and based on a spin lock mechanism (i.e., busy waiting instead of suspension-based). MrsP is specially interesting, because it provides two desirable theoretical properties: optimality and compliance to well-known uniprocessor response time analysis [17]. In summary, we make the following new contributions:

- The original design proposed in [1] was modified to include the MrsP protocol. Thus, the new design supports both single- and multi-core protocols and also suspension- and spin-based protocols. To the best of our knowledge, this is the first work to provide an RTOS-level design for such variation of protocols;
- We implement the proposed design in an RTOS and measure the memory footprint and run-time overhead of the implementation in a modern multi-core processor. The maximum overhead for the MrsP is around 6.1 µs for p and v operations and for the helping mechanism of the protocol (including IPIs and scheduling overheads). Compared to related works [17, 18], our implementation has presented better performance (up to 47% faster than the implementation of [17] even with an older and slower processor);
- We present a new schedulability analysis considering the run-time overhead obtained from the implementation of the new design. The low run-time overhead applied in the schedulability analysis keeps the schedulability ratio close to theoretical bounds indicate that the schedulability ratio remains, proving the efficiency of the proposed object-oriented design.

The remainder of this paper is organized as follows. Section 2 presents the considered task and resource model and reviews the resource sharing protocols used in this work. Section 3 discusses e implementation of the protocols. Section 4 evaluates the proposed design in an RTOS and real hardware platform. Section 5 presents the related work. Finally, Section 6 concludes the paper.

2 System Model and Background

2.1 Task and Resource Model

We consider a system with a finite set of synchronous periodic tasks. Each task τ_i is characterized by two positive integers: a worst-case execution time (WCET) e_i and a period p_i. All tasks are activated synchronously at the same time. From that time on, a new instance of a task τ_i is released at every period p_i. Each instance of task τ_i requires e_i processor time and must finish before the release of its next instance (*i.e.,* implicit deadlines). Tasks can suspend their executions only when waiting for the shared resource access. Task τ_i utilization factor is given by $u_i = e_i/p_i$ and the total system utilization U is a sum of all tasks utilizations within the task set.

The system also contains a possibly empty set of n_{res} shared resources. Each task might require exclusive access to one or more resources within this set and each shared resource might be used by at most one task instance at a time (*i.e.,* serially reusable resources). A part of the task code that uses a shared resource is called a critical section. $L_{i,r}$ denotes the maximum length of time that task τ_i might take to execute its critical section for the resource r where $0 < r \leq n_{res}$ (all task critical sections are accounted for in the task WCET). For the sake of simplicity, we assume only non-nested critical sections.

Tasks are executed sequentially upon m identical processors with $m \geq 1$ (in particular, for single-processor $m = 1$ and for multiprocessor $m > 1$). We assume that tasks are statically partitioned to processors, that is, all instances of a given task are executed only on one particular processor (*i.e.,* tasks cannot migrate among processors). Depending on the task allocation, a resource can be *local*, when shared by the tasks from the same processor, or *global*, when shared by the tasks from different processors. We assume that a task can access all shared resources directly from the processor it is assigned to.

A priority-based scheduling algorithm assigns at each time instant the processor to the task with the highest priority. Tasks' priorities can be fixed or dynamic. In this work, we consider the most common fixed- and dynamic-priority algorithms, respectively, Fixed-Priority (FP) and Earliest Deadline First (EDF), as well as their multi-core versions, Partitioned Fixed-Priority Preemptive (P-FP) [19] and Partitioned Earliest Deadline First (P-EDF) [20]. For FP and P-FP, we assume that the tasks are indexed in the increasing priority and we say that τ_j has a higher priority than τ_i if and only if $j > i$. In EDF and P-EDF, tasks dynamic priorities are decided based on their absolute deadlines. A set of tasks is said to be schedulable under a given scheduling algorithm if this algorithm can always schedule all instances generated by the tasks on the assigned processors without any deadline miss. Next, we review the major single-core and multi-core real-time resource access protocols.

2.2 Single-Core Resource Access Protocols

This section presents an overview of the single-core real-time synchronization protocols implemented in this work. The protocols are the Priority Inheritance Protocol (PIP), the Priority Ceiling Protocol (PCP), the Immediate Priority Ceiling Protocol (IPCP), and the Stack Resource Policy (SRP). For a complete overview, please refer to [2, 3].

PIP is a classic mechanism for sharing resources in a single-processor with FP scheduling [6]. It aims at avoiding priority inversion by elevating the priority of the resource-holding task to the highest-priority among the tasks it is currently blocking. Consequently, the protocol prevents the medium-priority tasks from preempting the lower-priority task that is blocking a higher-priority task. However, the protocol does not prevent the formation of chained blocking (*i.e.,* at task release, each task critical section can be held by a lower-priority task) and deadlocks (*i.e.,* two tasks can be waiting for each other).

PCP is another classic protocol for controlling priority inversion and bounding blocking time for a task set with shared resources. PCP prevents the formation of deadlocks and chained blocking [6]. In this protocol, as in PIP, a higher-priority task, when blocked by a lower-priority task holding a resource, transmits its priority to the lower-priority task. However, a task can be refused to access an unused shared resource whenever there is a lower-priority task holding a shared resource that might be requested at a later time by a higher-priority task. This rule is imposed by the priority ceilings defined, for each resource, as the maximum priority among all tasks that can access the resource. A task can enter a critical section only if its priority is higher than all priority ceilings of the currently held shared resources.

IPCP is a variant of PCP, aiming for performance and ease of implementation. The major difference is that the task owning the resource has its priority raised to the ceiling immediately when it first acquires the resource, and not when another task tries to lock the resource. The main effect of this change is a reduction of context switching overhead.

SRP provides resource access control for dynamic scheduling policies (*e.g.,* *EDF*) and supports multi-unit resources (*e.g.,* run-time stack) [7]. To handle dynamic priorities, SRP introduces *preemption levels* that, contrarily to the dynamic priorities, are constant and statically assigned to each task instance (*e.g.,* in *EDF*, the preemption levels are assigned inversely to task relative deadlines). The resource ceiling is defined by the highest preemption level of the task that may be blocked on that resource. For the multi-unit resources, the ceiling is a dynamic value equal to the highest preemption level of the task that may request more resource units than currently available and, as a consequence, be blocked on the resource. A task instance can start to execute when: i) its priority is the highest among all ready tasks, and ii) its preemption level is higher than the ceilings of all shared resources. We note that the above rules can cause unnecessary blocking (*e.g.,* if a task does not require any resource), but at the same time and more importantly, they guarantee the absence of chained blocking and deadlocks. Furthermore, due to the early blocking, the context switch number is reduced.

2.3 Multi-Core Resource Access Protocols

MPCP [9] is an extension to multi-core partitioned scheduling of described above single-core PCP. Both protocols apply the same policy for the local resources shared by the tasks allocated to the same processor. However, when acquiring a global resource shared by the tasks from different processors, a job priority is raised above the priority level of any task in the system. More precisely, a job within a global critical section R_k has its effective priority raised to $\pi_H + max_i\{i \,|\, \tau_i \; uses \; R_k\}$ where π_H is the highest priority among all tasks on all processors. If a global lock is already held on a different processor, a task requesting the lock is suspended. In the suspension-based version of the protocol that we consider in this work, other ready tasks can execute while

waiting for a global lock. Moreover, a task within a global critical section can be preempted by another task assigned to the same processor if the latter task is granted access to a global critical section whose effective priority is higher.

MSRP [10] is an extension of previously described single-core SRP to partitioned multi-core scheduling. In MSRP, the local resource sharing is handled in the same manner as in SRP whereas the global resource sharing is controlled by the use of a first in first out (FIFO) queue-based approach. Access to a global resource is granted accordingly to the order of task request arrivals. Tasks waiting for a global resource do not suspend and keep their processors busy. After acquiring a global shared resource, the task executes the corresponding critical section non-preemptively until completion.

The Multiprocessor Resource Sharing Protocol (MrsP) [12] aims to achieve an effective schedulability analysis by mixing properties from two other protocols, MSRP and SPEPP [21]. Using suspension-based protocols queues can take longer than the defined cost, because more than one task at the same processor could be in the blocked queue affecting the desired schedulability analysis results. Thus, MrsP uses spinlocks to handle mutual exclusion of the resources. MrsP establishes that tasks waiting to gain access to a resource must service the resource on behalf of other preempted tasks waiting for the resource. Meaning that tasks can use their time spinning (waiting for the resource) to decrease overall finishing time of other tasks that were preempted while running in the resource. This is accomplished by migrating the preempted resource owner to a core where there are tasks spinning waiting to get access to the critical section. The migrated task raises its priority to the global ceiling of the resource, and after finishing its execution, returns to its original core and restores its priority. The migrated task owning the resource is preempted outside its original processor, it should search for another helper core to migrate and continue to execute.

3 Design and Implementation of the Protocols

In our previous work, we have analyzed the common characteristics related to the suspension-based resource access protocols (i.e., PIP, PCP, IPCP, SRP, MPCP, and MSRP) [1]. In this work, we have included the MrsP protocol (spin-based protocol) into the proposed design. Figure 2 presents an overview of the updated design, representing the common structure for all protocols.

The `Synchronizer_Base` class offers support for operations common to all synchronization primitives, including, for instance, atomic increment and decrement (*finc* and *fdec*), test and set lock (*tsl*), and interrupt enabling/disabling (*begin_atomic* and *end_atomic*). The parameterized class `Synchronizer_Common` implements an interface for common thread operations, such as sleep, wakeup, and wakeup all. These thread operations put a thread to sleep into the synchronization queue (the `_queue` attribute), wakeup a thread that was sleeping in the queue, and wakeup all threads that were sleeping in the queue. All operations are protected, which means that they are only accessible by

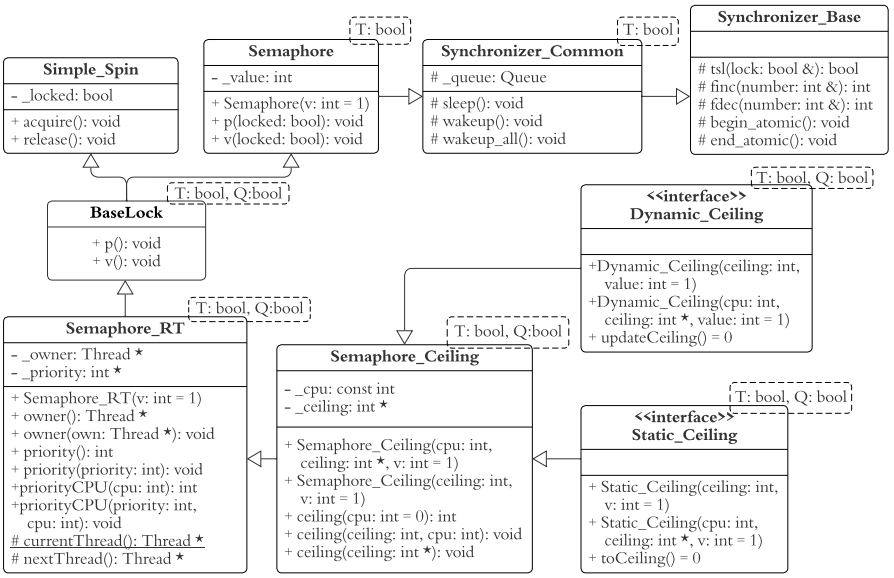


Fig. 2: Common structure of base classes.

its subclasses. The boolean parameter defines the type of the queue (either a priority- or FIFO-based). The use of template avoids the overhead of choosing the appropriate queue at run-time (MSRP, for instance, uses a FIFO-based queue, while ceiling-based protocols use a priority-based queue) and allows the use of different protocols in the same system.

The `Semaphore` class implements the traditional p and v semaphore operations [22]. The class has an integer (`_value`) as attribute, which is used to count the signals issued by the p and v (decrement and increment, respectively) through the `fdec` and `finc` operations implemented in the base class. It also uses the `sleep`, `wakeup`, and `wakeup_all` operations from the base class. The boolean class parameter is passed to `Synchronizer_Common` to define the type of the queue.

Comparing to the previous design, the common base classes structure has four new classes: `BaseLock`, `Simple_Spin`, `Dynamic_Ceiling`, and `Static_Ceiling`. The `BaseLock` class is a parameterized class that chooses between a suspension- or spin-based protocol. If the `T` parameter is true, then the used protocol is a spin-based one and the class inherits from `Simple_Spin`, which implements a traditional spin lock. Otherwise, the `BaseLock` class inherits from `Semaphore` and follows the structure described above. The second parameter received by `BaseLock` is the choice of the queue type (either a priority- or FIFO-based).

The `Semaphore_RT` class is common to all real-time resource access protocols. It has two attributes, `_owner` and `_priority` that represent, respectively, the current thread that owners the semaphore (*i.e.,* a thread that has entered a

critical section through the p operation) and the priority it had when entering the critical section. The class offers public methods to set and get attributes. Also, it has two protected methods, `current_thread` and `next_thread`, that returns the current thread being executed (note that it can be different from the `_owner`) and the next thread that is the head of the semaphore's queue. These two operations are required by the PIP, PCP, IPCP, and MPCP protocols.

The `Semaphore_Ceiling` class is common to all ceiling-based protocols, such as IPCP and PCP. It adds an integer attribute (`_ceiling`), which represents the semaphore's ceiling, and a new attribute `_cpu` that represents the current assigned CPU of a thread. It also has the set and get methods for the attribute. We added two new interface classes, `Dynamic_Ceiling`, and `Static_Ceiling`, that inherit from `Semaphore_Ceiling` and offer support for those protocols that need to update the ceiling (SRP and MRSP) and those protocols that use static ceilings (PCP, IPCP, MPCP, and MrsP).

Figure 3 shows the new proposed design for the protocols. The `Semaphore_PIP`, `Semaphore_PCP`, `Semaphore_IPCP`, `Semaphore_MPCP`, `Semaphore_SRP`, and `Semaphore_MSRP` classes remains basically the same as in the original design proposed in [1], the only difference is that they now implement the respective interface (either static or dynamic ceiling). These classes implement the behavior of each protocol and a complete description of each of them can be found in [1].

The main difference between our design and implementation and other implementations [17, 18, 23] is the use of object-orientation and templates that allow code reuse, extensibility, low memory footprint and overhead (as will be demonstrated in Section 4. For instance, the `Semaphore_MPCP` expects a boolean value that represents the type of the critical section. If the resource is a local one, the user should create a class instance passing false as a template argument. Otherwise, the user should pass true as the class template argument to represent a global resource. When a local resource is used, then `Semaphore_MPCP` behaves as the IPCP protocol. When guarding a global critical section, on a p operation, the class raises the `owner` priority to the `_highestPriority` instead of `_ceiling` priority. The `_highestPriority` attribute is defined at compile-time and stores the highest priority of all tasks that use global resources among all cores.

At a similar way, `Semaphore_MSRP` also expects a boolean to indicate protection of a local or global critical section (GCS). When true is passed, the class implements p and v operations for MSRP. Otherwise, the protocol works as SRP. Moreover, `Semaphore_MSRP` defines the type of queue as template argument as well (true to indicate a queue ranked by priority, whenever a local critical section is going to be guarded, and false to use a FIFO-based queue).

Finally, the `Semaphore_MrsP` class provides the resources for the implementation of the MrsP protocol, such as references to the owner and current helper of the resource through the attributes `_mrsp_owner` and `_helperTask`. Those references are updated in the p and v methods, which uses the acquire and release methods defined in `Simple_Spin` base class, inherited by `BaseLock`. The resource acquire and release procedures also updates the core affinities, stored in `_resourceAffinities`. If the active task cannot access the resource,

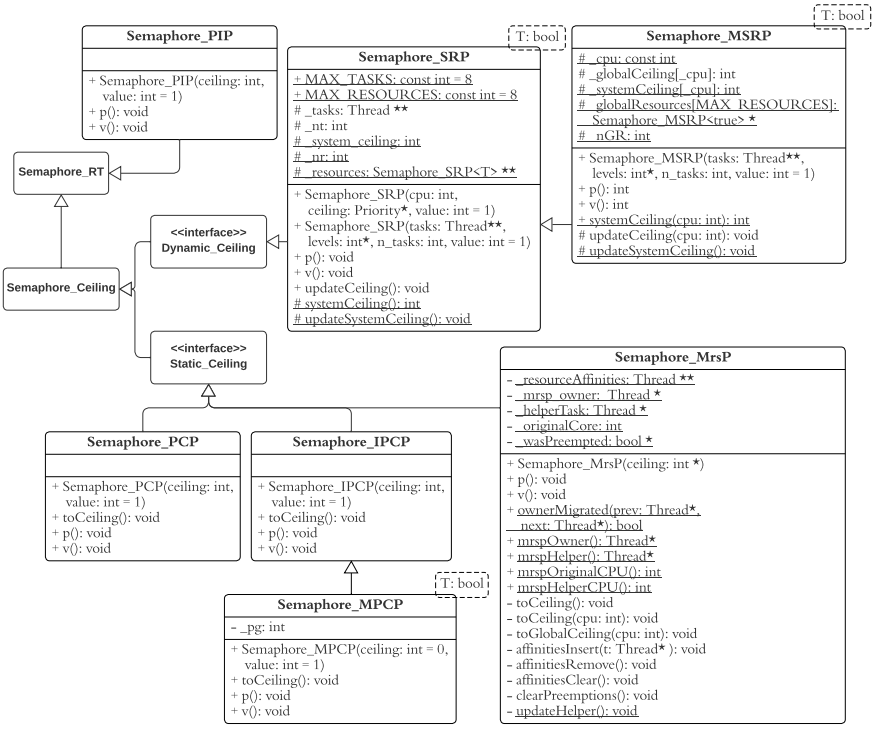


Fig. 3: Design representation of the resource access protocols.

its core is added to be one of the affinities, a helper candidate whenever the owner is preempted.

The `ownerMigrated` method checks if the current owner was preempted and if there is an available core (affinity) to help it to continue its execution. If there exists a helper core, the original core is stored in `_originalCore` and the method returns true, meaning that the owner task can migrate to the helper task core. In this process, an array containing the cores in which the owner was already preempted is also updated (`_wasPreempted`). In this way, the algorithm migrates the owner to a core where it was not preempted before.

3.1 Implementation in an RTOS

We have implemented the described design in the Embedded Parallel Operating System (EPOS) [24]. EPOS is a multi-platform, object-oriented, component-based, embedded system framework implemented in C++. It is the first open-source RTOS designed from scratch that supports partitioned, global, and clustered versions of EDF, RM, LLF, and DM scheduling policies [25]. A complete review of the real-time support on EPOS can be found in [25]. We

choose EPOS, because it supports static and dynamic scheduling, it is written in an object-oriented language, and it was used in our previous paper with the original proposed design [1]. We believe that the proposed design can be replicated in any object-oriented RTOS written in C++ and that has EDF, RM, P-EDF, and P-RM schedulers. Moreover, EPOS until this work did not have support for the MrsP. We have modified the implementation in [1] to reflect the changes in the software design described above. The implementation code of the proposed design in EPOS is publicly available and can be re-used to expand the model adding support to more protocols.[2].

To exemplify how we adapt a class at compile-time, the code in Figure 4 illustrates part of the implementation of the `Semaphore_MSRP` class. In the line 3, the class inherits from `Semaphore_SRP` passing the queue choice (a boolean) as parameter to handle local resources. In the lines 9 and 10, the p and v operations of the `Semaphore_SRP` are called. When `Semaphore_MSRP` must handle GCS, then a template specialization (line 15) is used. In this case, note that after calling p or v, there is an extra call to update the ceiling (lines 20 and 21).

```
1  //MSRP implementation when the resource is local
2  template<bool T>
3  class Semaphore_MSRP: protected Semaphore_SRP<T> {
4  private:
5      typedef Semaphore_SRP<T> Base;
6      // SRP inherits from Dynamic_Ceiling< T, true>
7  public:
8      //constructor here
9      void p() { Base::p(); }
10     void v() { Base::v(); }
11 };
12
13 //MSRP specialization when the resource is global
14 template<>
15 class Semaphore_MSRP<true>: protected Semaphore_SRP<false> {
16 private:
17     typedef Semaphore_SRP<false> Base;
18 public:
19     //constructor here
20     void p() { Base::p(); updateCeiling(); }
21     void v() { Base::v(); updateCeiling(); }
22 };
```

Fig. 4: Example of template specialization of the `Semaphore_MSRP`.

4 Experimental Evaluation

This section describes the experimental evaluation of the previously described implementation. Our objectives are: i) to measure the memory consumption of the implementation; ii) to evaluate the run-time overhead of the implementation;

[2] The code is available online at `https://github.com/santos-lucasm/epos-mcore-protocols`.

iii) to verify the impact of the run-time overhead into the system schedulability[3]; and iv) to compare the overheads of the MrsP implementation with related works to validate the proposed design and implementation. The next subsections show the obtained results for the objectives.

4.1 Memory Footprint

For measuring the memory consumption of our implementation, we have executed EPOS on top of an Intel i7-2600 processor (clock of 3.4 GHz, 8 logical cores, 8 MB L3 cache). We used the *GNU gcc* compiler at version 7.5.0 to generate the code. For measuring the memory footprint, we used the *GNU objdump tool* at version 2.30.

Table 1 shows the obtained memory footprint for each class depicted in Figures 2 and 3. The table also shows the total lines of code, just to correlate the memory footprint with the implementation. Memory usage is split into code, data, and static data sections. Data represents the memory consumed by an instance of the corresponding semaphore type, not including the footprint of the base classes. The total memory consumption describes the memory consumed by the implementation of the semaphore subclass and a single corresponding object instance. For instance, the total memory consumption of the `Semaphore_PIP` is 876 bytes, which represents its own 580 bytes summed with `Semaphore` and `Semaphore_RT` memory consumption. It is important to highlight that code from the `Synchronizer_Base`, `Synchronizer_Common`, `Simple_Spin` and `BaseLock` classes are not represented in the Table 1, because it is inlined into the methods that use the base class. This is also true for the interfaces `Static_Ceiling` and `Dynamic_Ceiling`, meaning that the code of these classes are implemented in the header file to improve the run-time overhead by avoiding explicitly method calls. MPCP local has the same memory footprint as IPCP and MSRP local has the same memory footprint as SRP due to the template class parameter as discussed in Section 3 and exemplified in Figure 4. The MrsP implementation consumes 1.6 KB of memory, including code and static data. The number of lines in the Table 1 confirms the code reuse and extensibility of our design.

4.2 Run-time Overhead

For measuring the run-time overhead of the implementations, we used the processor's Time Stamp Counter (TSC). For each single-core protocol, a set of 20 tasks tried to acquire the same resource in a cascade, being delayed by the system alarm in 1 ms after that, and subsequently releasing the resource. After the acquire and release, the task waits for its next period (50 ms). For multi-core protocols, the test consisted of up to 8 tasks, each one assigned to a

[3] Run-time overhead is defined as the additional computing time carried out by the protocols and RTOS. Systems schedulability refers to the rate of schedulable tasks in the system according to a scheduling algorithm (or scheduler).

Table 1: Memory footprint and lines of code of the implemented classes.

	Memory usage (bytes)					
Class	Code	Data	Static Data	Total Mem. Consum.	Lines of Code Header	Source
Semaphore	272	16	0	288	9	22
Sem. RT	0	8	0	296	29	0
Sem. Ceiling	0	4	0	292	8	0
PIP	580	0	0	876	6	31
PCP	634	0	0	926	6	31
IPCP	624	0	0	916	6	28
SRP	644	168	53	1157	80	31
MPCP Global	666	0	4	1586	10	40
MSRP Global	412	32	73	1674	81	22
MrsP	1552	0	60	1624	105	138

different core, trying to acquire the same global resource in a cascade, releasing the resource after that, using the same task flow, period and computation time from single-core protocols.

For measuring the run-time overhead of the MrsP implementation, a different application was built to force preemption of tasks executing in a critical section. In this application, one main task arrives first and gains access to the shared resource, after this, other tasks are launched in different cores, trying to access the same resource and end up being kept spinning without gaining access to the shared resource. Those tasks are the helpers, which borrows their processors for the task executing in the critical section. The owner of the resource is then preempted to force the helping mechanism of the MrsP to happen. In different test runs, the number of helpers was increased, varying from 1 to 8, with each possible helper task assigned to its own core.

The p and v methods of every semaphore type and spinlock (for MrsP) were instrumented, as well as the system code responsible for performing queuing/dequeuing and rescheduling of tasks, and inter-processor interruptions (IPIs), which is used by the base semaphore implementation. These sections were timed with a small helper utility, that wraps the code, setups the TSC, and uses it to count the amount of time consumed by the code section. Each task acquired and released the resource 1000 times, and the worst-case run-time overhead was extracted from the execution.

Figure 5 presents the obtained worst-case run-time overhead for each protocol in ns to perform p and v operations as a function of the number of tasks waiting on the semaphore, while Table 2 presents the obtained results exclusively for the MrsP protocol. The overheads in Figure 5 depend on the number of tasks, since whenever a task blocks it is queued in a linked list. As the queue grows, the operations take more time. The queue overhead overcomes the overhead of the protocols. However, we can note a very small difference when no tasks are waiting on the semaphore (PCP, PIP, and IPCP have mostly the same behavior).

Although the dequeuing operation performed at every v operation touches the head of the queue, we can note a decrease as the number of tasks increases. This is mainly caused by concurrent events (*i.e.,* the alarm that handles the

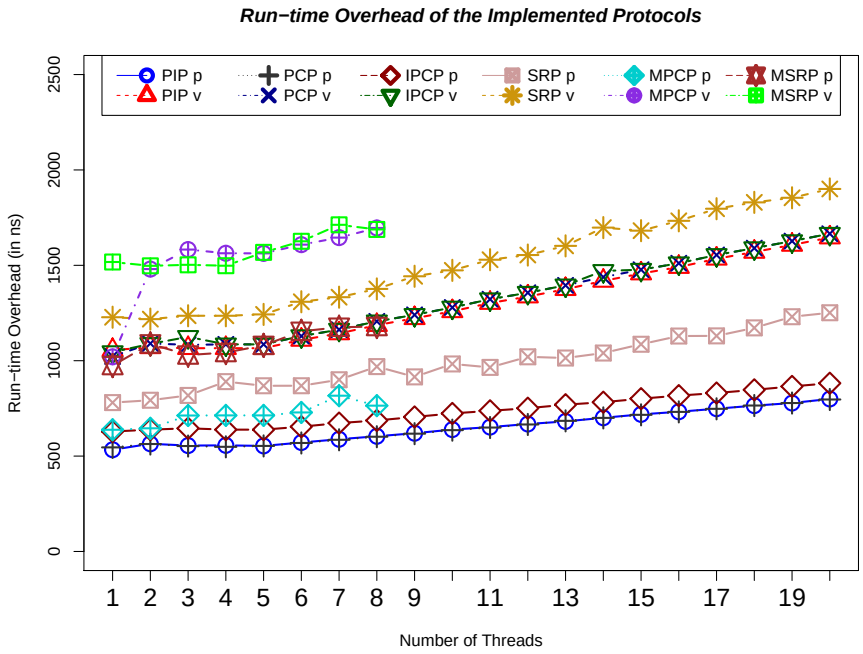


Fig. 5: Obtained run-time overhead (on the Intel i7 processor).

task releases shares internal data structures with the rescheduling operation). The v operation of the SRP is the worst, because it demands an updating of the system ceiling, which is performed in a loop (its size is equal to the number of tasks that use the resource, 20 in our experiments). The case of tasks being blocked on a semaphore under SRP should never happen in principle. This is because of the preemption test, that disables a task from starting execution if one of its resource accesses would cause it to block. However, in our practical implementation, non-real-time tasks do not have an assigned preemption level and are unaffected by SRP. Therefore, in the case of a soft real-time application that makes mixed-use of task types, tasks may still block on a resource.

The MSRP and MPCP v operations clearly show the effects of having more than one core accessing the same global resource. The p operation in the MPCP is the same as in IPCP (just uses a defined global ceiling). That is the reason why it presents a slightly higher overhead than IPCP. Another important source of run-time overhead is the time to switch the context between two tasks. Since the number of context switches may change depending on the protocol, we also use the worst-case context switch (0.3 µs), as measured in [25], in the schedulability impact discussed next.

We consider the overheads for MrsP (depicted in Table 2) in three different sections of the code, lock, unlock, and helping, as also done in [17, 18]. The

lock column presents the overhead of the acquire operation, responsible for storing the current owner and updating the core affinities when the access to the resource is denied. The unlock column presents the overhead of the resource release operation, which takes more time due to the reorganization of tasks: unlocking the resource outside its original core demands a task migration back to its original core, requires an update in the owner and helper task references, and in affinities and preemption arrays. The helping column contains the overhead for the preemption detection, task migration and inter-processor interruption (IPI). Note that due to our MrsP experiment organization, there is no IPI when we have only one core (there is no migration – the overhead in this case is only the preemption detection).

For the unlock operation, we see a different behavior with 4 cores. This was due to the absence of a preemption of the resource owner or no helper was available when the owner was preempted. Both cases would result in an unlock operation with less overhead, since the owner did not migrate to another core and it was executing in its original core. The helping mechanism overhead stays practically the same in all the cases, since this code section does not have many conditionals forks and possible time variants besides the IPI.

Table 2: MrsP overhead.

	Operations overhead (nanoseconds)		
Cores	Lock	Unlock	Helping
1	315	196	48
2	345	3098	1737
3	860	2962	1726
4	509	246	1726
5	679	3049	1726
6	552	3052	1724
7	1451	2580	1760
8	803	2682	1726

4.3 Schedulability Impact

For measuring the impact of the implemented locking protocols overheads on the system schedulability, we performed the experiments with randomly generated task sets. The experimental setup is similar to the previous works [4, 26].

A task period p_i was randomly chosen from a log-uniform distribution ranging over [10 ms, 100 ms] (*homogeneous periods*) or [1 ms, 1000 ms] (*heterogeneous periods*). A task utilization u_i was randomly chosen from an exponential distribution with a mean 0.1 (*light*), 0.25 (*medium*) or 0.5 (*heavy*).

Tasks share a number of resources within a set $n_{res} \in \{1, 2, 4, 8, 16\}$. For each task and for each resource, the task to resource assignment is given with a probability $p^{acc} \in \{0.25, 0.5, 0.75, 1\}$. The critical sections lengths were chosen uniformly from [1 µs, 25 µs] (*short*), [25 µs, 100 µs] (*medium*), or [100 µs,

500 µs] (*long*). Each task's instance accesses a resource only once per activation, all resources are single-unit and all critical sections are non-nested.

Tasks run on m processors with $m \in \{1, 2, 4, 8\}$. We assumed partitioned scheduling where the tasks are assigned to the processors using the worst-fit decreasing heuristic [20]. We acknowledge that better results could be achieved with specialized resource-sharing-aware heuristics [27]. We analyzed the tasks schedulability under P-FP and P-EDF policy. To verify the schedulability, we implemented response time analysis for the former, processor demand criterion for the later, and the respective polynomial-time utilization-based tests [3, 28]. For the fixed-priority scheduling, the task priorities were assigned by *Rate Monotonic* [28] (*RM* and *P-RM*).

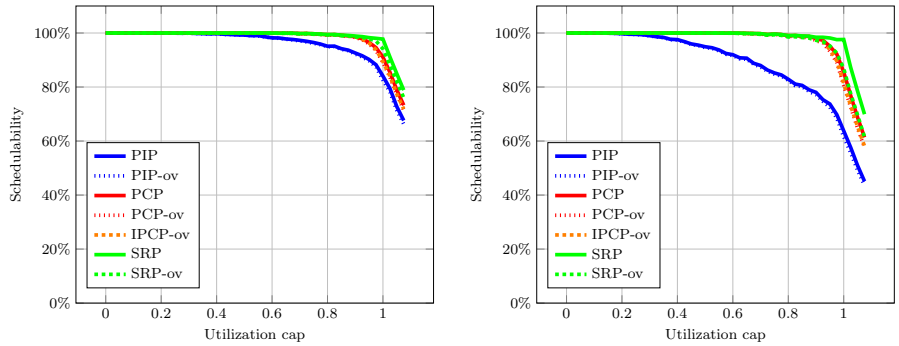
We analyzed four resource access protocols on a single-core platform ($m = 1$) with the associated analyses for blocking: PIP [6], PCP and IPCP [6] (all for FP) and SRP [7] (for EDF). On multi-core platforms ($m > 1$), we studied MPCP [27] and MrsP [12, 16] under P-RM and MSRP [10] under P-EDF. Task set generation and schedulability tests were implemented in *SchedCAT* [29]. In our experiments for MsrP, we considered the length of the non-preemptive critical section equal to ten times its incurred migration overhead (see *Helping* in Table 2).

For each combination of the described above parameters, we generated 1000 task sets increasing the task set utilization cap (*i.e.,* utilization limit of a generated task set) by 0.05 at each step. The schedulability of each task set was then verified (solid lines on Figures 6 – 9). Additionally, we considered the overheads (dotted lines) by inflating each task's WCET with the obtained run-time overheads (see Figure 5 and Table 2) and context-switch penalties incurred by each protocol [7] (dotted lines on Figures 6 – 9). The experiments covered more than 2500 cases. Here, we report on our main findings[4].

The first batch of experiments focused only on the single-processor lock sharing protocols ($m = 1$). The tasks were generated with *heterogeneous* periods and *long* critical sections. Figures 6 and 7 show the impact of shared resource number and task number on the schedulability (utilization *heavy* means that task number is low and vice versa). PIP performance degrades significantly with the number of shared resources due to the chained blocking. Early blocking in IPCP reduces the number of context switches but its benefit compared to PCP is barely discernible. We note that for high utilizations, the large overhead of SRP degrades its theoretical performance more than other protocols. The main reason for this is that SRP handles multi-unit resources and thus its overhead is higher.

The second part of our experiment covers three protocols for multi-processor: MPCP and MrsP under *P-RM* and MSRP under *P-EDF*. Figures 8 and 9 show the results for the task sets generated with *medium* utilization, *heterogeneous* periods, $p^{acc} = 0.75$, *short* critical section and $m = 8$ processors. For the sake of completeness, we also show the schedulability of the same task sets without any shared resources under *P-RM* and *P-EDF*. In the experiment, we vary the

[4] All graphs are available at `www.lisha.ufsc.br/Giovani`.



a. *heavy*, $p^{acc} = 0.25$, $n_{res} = 4$ b. *medium*, $p^{acc} = 0.5$, $n_{res} = 8$

Fig. 6: Selected results for single-processor setup (periods heterogeneous, long critical sections).

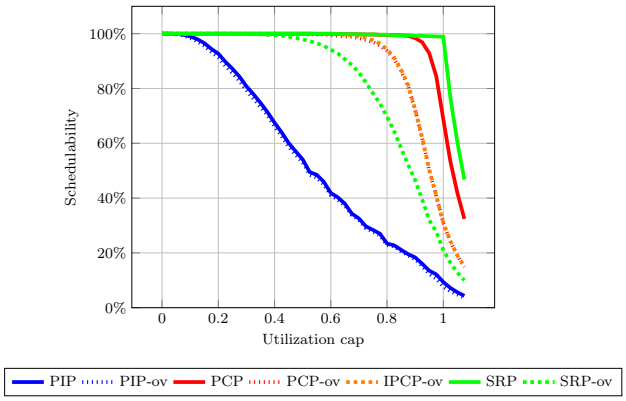


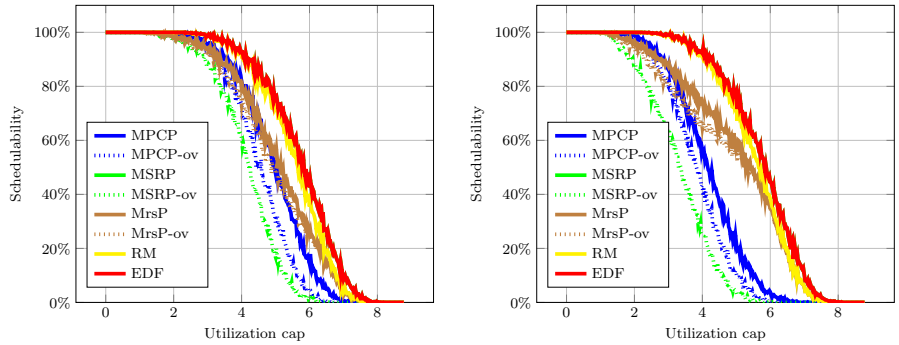
Fig. 7: Single-processor setup using periods heterogeneous, long critical sections, light utilization, $p^{acc} = 0.75$ and $n_{res} = 16$

number of shared resources n_{res}. As expected, having more shared resources leads to lower schedulability. In particular, MSRP performance is degraded and worse than MPCP. This is due to the preemption levels and system ceiling management that demands a loop to iterate over the tasks and preemption levels for each processor. The best schedulability performance is achieved by MrsP. Having short critical sections, as assumed in the experiments, results in low blocking. The increase in the schedulability performance in Figure 9 may appear counter-intuitive, but can result from the worst-fit task-to-processor allocation. Indeed, for low utilizations, the tasks are distributed among the processors and there is a high probability of having most of the resources on different processors (*i.e.,* global resources). With the increase of utilization, more tasks are allocated to the same processors and the resources tend to become local and do not incur additional delay due to the parallel access.

Moreover, we can note that MrsP performance tends to the theoretical P-RM and P-EDF bounds when increasing the utilization cap, which confirms the better schedulability test of the MrsP over MPCP and MSRP.

We derived the following observations from the experiments: i) PIP is less sensitive to the run-time overhead than PCP and IPCP; ii) schedulability is higher in task sets with medium utilizations (there are less tasks); iii) homogeneity of task periods has a major effect on the schedulability ratio (short period tasks were more affected by blocking-time bound terms on the schedulability analyses); iv) critical section sizes heavily impact schedulability for light-utilization task sets; v) due to higher overhead, even with better schedulability, MSRP can be worse than MPCP; vi) our design and implementation provides low overhead, specially for the multi-core protocols; and vii) MrsP is the best protocol among the analyzed ones for multi-core processors.

Our study shows that the locking overheads are non-negligible, especially when the system load is close to theoretical upper bounds and the number of shared resources is high. However, in the great majority of cases, our design/implementation does not adversely impact the system schedulability.



a. $n_{res} = 2$ b. $n_{res} = 4$

Fig. 8: Selected results for multi-processor setup (utilization medium, periods heterogeneous, $p^{acc} = 0.75$, short critical sections, $m = 8$ processors).

5 Related Work

Uniprocessor resource access protocols. Suspension-based resource access protocols for uniprocessor real-time systems were first proposed by Sha et al. [6]. The authors have proposed PCP and PIP and the work has served as a basis for much other research. SRP was proposed by Baker in 1991 [7] and it was also a seminal work, mainly for providing resource access protection for dynamic scheduling.

Several general-purpose OSes and RTOSes implement some of the analyzed real-time resource access protocol. For instance, FreeRTOS employs PIP in

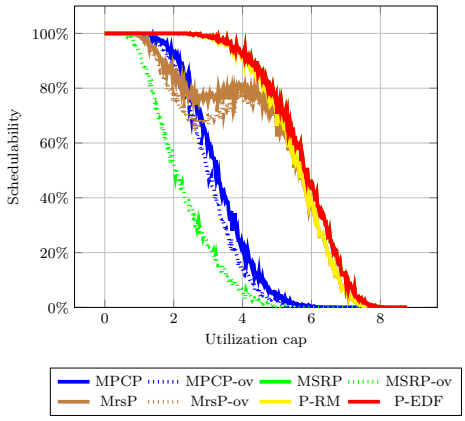


Fig. 9: Multi-processor setup using medium utilization, heterogeneous periods, $p^{acc} = 0.75$, short critical sections, $m = 8$ processors and $n_{res} = 8$.

the mutex primitive [30] and also supports SRP [31]. $LITMUS^{RT}$ supports PCP, SRP, and several multiprocessor resource access protocols [23]. The L4 microkernel [32] and Linux support priority inheritance. Linux also implements IPCP, under the name PRIO_PROTECT in the pthreads library. Lee and Kim implemented PIP in the μC/OS-II kernel and measured the run-time overhead running the implementation on top of the CalmRISC16 evaluation board [33]. The observed run-time overhead for the p and v semaphore operations was 30.5 µs. Researchers also proposed to move the mechanisms to control the priority inheritance [34] or the semaphore structures [35] to the hardware in order to reduce the run-time overhead.

Wang et al. implemented multi-resource versions of PIP and PCP in a component-based OS for controlling the access to shared stacks [36]. In their experimental evaluation considering the schedulability of generated tasks, PIP has performed better than PCP. Thus, they have concluded that PIP has the potential to provide a high-degree of schedulability [36]. In our experimental evaluation, however, PIP has presented a similar performance in terms of overhead when compared to PCP but had worse schedulability ratios. Caccamo et al. [37] proposed a scheduling framework for soft- and hard-real time tasks that share the same resources.

Multiprocessor resource access protocols. Resource access protocols for multiprocessors have been the subject for many researchers recently. Flexible Multiprocessor Locking Protocols is a collection of protocols for global and partitioned scheduling [38]. It was designed to efficiently deal with short non-nested access and to allow unrestricted critical section nesting. Parallel Priority Ceiling Protocol extends PIP to avoid unfavorable blocking situations but increases the run-time overhead [39]. Biondi and Brandenburg [26] have revisited four synchronization protocols under P-EDF scheduling and compared them in terms of schedulability. They concluded that the lock-free synchronization

approach offers advantages on asymmetric multiprocessing platforms. Recently, Teixeira and Lima [40] proposed a task allocation heuristic for global scheduling that leverages the concept of servers [41]. Yang et al. proposed new more precise schedulability analyses based on linear programming for several multiprocessor semaphore-based locking approaches [4]. The authors claim that the new analyzes are more accurate than prior approaches.

Brandenburg surveys multiprocessor real-time locking protocols, from 1988 until 2018 [42]. Robb and Brandenburg propose two multiprocessor protocols to support fine-grained nested locking, to guarantee independence preservation for fine-grained nested locking, and to ensure optimal priority-inversion bounds [43]. However, there is no implementation nor measurement of run-time overheads. Lock server paradigms were investigated as an alternative to decrease the blocking time caused by nested locking in multi-core processors [44].

In the preemptive resource sharing approach (PWLP), a task spins waiting for a shared resource with its original priority. If this task is preempted, its resource access request is canceled. When the task is rescheduled, it is placed at the end of the resource FIFO queue. The task becomes non-preemptable once it gets the resource lock [45].

The O(m) Locking Protocol (OMLP) combine the use of two queues, a priority and FIFO queue [15]. Tasks first contend for the acquiring a common m-exclusion priority lock (in the priority queue) and then are suspended in the FIFO queue to wait for the associated resource. When the task accesses the resource, it becomes non-preemptable under OMLP.

The Flexible Multiprocessor Locking Protocol (FMLP) requires the division of the resources by the programmer into long and short resources [38]. Then, it groups the resources according to its class (short or long), creating a resource group. Each resource group has a different mechanism of management. A group containing short resources uses a non-preemptive FIFO queue to handle the lock operation, while for a group containing long resources is protected by a semaphore lock. Groups are organized in such a way that resources can be nested, so a task can hold more than one resource at a time [12]. Resources that are used in a non nested fashion are grouped individually. As mentioned in [12], group locks reduce the parallelism and are an impediment to composability. FMLP can be used in global and partitioned-based systems.

The Real-time Nested Locking Protocol (RNLP) [13] extends the notion of fine-grained blocking bounds for nested resources firstly introduce in [46]. RNLP limits concurrency by a k-exclusion and access granting mechanisms. However, the algorithm provides sub-optimal results under different system configurations, limiting its applicability in real-world scenarios [14].

Several works have proposed improvements over the original MrsP [12], focusing on providing new features and improving the existing ones. For instance, Garrido et al. proposed the support of fine-grained nested resources [11]. Zhao and Wellings proposed a modified helping mechanism [47]. Zhao et al. proposed an analysis of the migration costs and an improved schedulability test based on RTA in [16], while in [14] the same authors extended the MrsP schedulability

test to support nested resource accesses and run-time costs (creating a complete run-time overhead-aware schedulability analysis for MrsP).

Considering the implementation of MrsP in real systems, Catellani et al. implemented the MrsP in two OSes, a real-time one (RTEMS) and LITMUS^{RT} [17]. The authors measured the run-time overhead in both OSes, obtaining around 13 µs and 52.3 µs, for RTEMS and LITMUS^{RT}, respectively. The authors claim that the increased kernel overhead, when compared to other protocols, is compensated by the improved RTA offered by MrsP.

Shi et al. also implemented the MrsP in LITMUS^{RT} and measured its related overhead in an Intel i7-5600U processor running at 2.6 GHz [18]. The authors only presented the average running overhead, not the worst-case. The obtained average overhead for MrsP was 7.1 µs. As proved in [25], the worst-case running overheads in LITMUS^{RT} are quite significant and must be accounted for a valid run-time overhead analysis. Using average time is not fair and may hide the inherent non real-time behavior of Linux-based systems. Moreover, neither [17] nor [18] discussed the design choices and the integration of MrsP in RTOSes that provide other real-time resource access protocols, focusing on providing code reuse, easy usability, and low run-time worst-case overhead. We have shown in our evaluation lower overheads than both works [17, 18], even running our system in an older and slower processor (for instance, Shi et al used a 5th generation of the i7, while we used an i7 of the 2nd generation).

6 Conclusion

In this paper we presented an object-oriented design of 7 real-time resource protocols for single and multi-core processors. The design supports both suspension- and spin-based protocols (PIP, PCP, IPCP, and SRP for single-core systems and MPCP, MSRP and MrsP for multi-core systems. MrsP is the only spin-based protocol). To the best of our knowledge, this work is the first to support such variation of resource access protocols in an RTOS.

Our design recalls on a clean class hierarchy and template specialization to allow code reuse, extensibility, low memory footprint and run-time overhead. We implemented the proposed design in a modern multi-core processor and measured the memory footprint (from 876 to 1674 bytes, depending on the protocol) and run-time overhead (less than 4 µs for 20 tasks and around 6.1 µs for the MrsP protocol). Moreover, we used the run-time overhead to analyze how it affects the system schedulability and proved that efficient RTOS implementation of resource access protocols has few impacts on the system schedulability. As future work, we want to extend the experiments for nested locks.

Compliance with Ethical Standards

Conflict of Interest: The authors declare that they have no conflict of interest.

References

1. L. M. dos Santos, G. Gracioli, T. Kloda, and M. Caccamo, "On the design and implementation of real-time resource access protocols," in *2020 X Brazilian Symposium on Computing Systems Engineering (SBESC)*, 2020, pp. 1–8.
2. J. Liu, *Real-Time Systems*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.
3. G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 3rd ed. Springer, 2011.
4. M. Yang, A. Wieder, and B. Brandenburg, "Global real-time semaphore protocols: A survey, unified analysis, and comparison," in *RTSS*, 2015, pp. 1–12.
5. M. Jones. (1997, Dec) What really happened on mars? [Online]. Available: http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html
6. L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sep 1990.
7. T. P. Baker, "Stack-based scheduling for realtime processes," *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, Apr. 1991.
8. N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, pp. 284–292(8), September 1993.
9. R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *10th ICDCS*, May 1990, pp. 116–123.
10. P. Gai, G. Lipari, and M. Di Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *22nd IEEE RTSS*, 2001, pp. 73–83.
11. J. Garrido, S. Zhao, A. Burns, and A. Wellings, "Supporting nested resources in mrsp," in *Reliable Software Technologies – Ada-Europe 2017*, J. Blieberger and M. Bader, Eds. Springer International Publishing, 2017, pp. 73–86.
12. A. Burns and A. J. Wellings, "A schedulability compatible multiprocessor resource sharing protocol – mrsp," in *ECRTS*, July 2013, pp. 282–291.
13. B. C. Ward and J. H. Anderson, "Supporting nested locking in multiprocessor real-time systems," in *24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*. Los Alamitos, CA, USA: IEEE Computer Society, jul 2012, pp. 223–232.
14. S. Zhao, J. Garrido, R. Wei, A. Burns, A. Wellings, and J. A. de la Puente, "A complete run-time overhead-aware schedulability analysis for mrsp under nested resources," *Journal of Systems and Software*, vol. 159, p. 110449, 2020.
15. B. B. Brandenburg and J. H. Anderson, "Optimality results for multiprocessor real-time locking," in *2010 31st IEEE Real-Time Systems Symposium*, 2010, pp. 49–60.
16. S. Zhao, J. Garrido, A. Burns, and A. Wellings, "New schedulability analysis for mrsp," in *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017, pp. 1–10.
17. S. Catellani, L. Bonato, S. Huber, and E. Mezzetti, "Challenges in the implementation of mrsp," in *Reliable Software Technologies – Ada-Europe 2015*, J. A. de la Puente and T. Vardanega, Eds. Cham: Springer International Publishing, 2015, pp. 179–195.
18. J. Shi, K.-H. Chen, S. Zhao, W.-H. Huang, J.-J. Chen, and A. Wellings, "Implementation and evaluation of multiprocessor resource synchronization protocol (mrsp) on litmus^{RT}," in *13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2017.
19. S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Oper. Res.*, vol. 26, no. 1, p. 127–140, Feb. 1978. [Online]. Available: https://doi.org/10.1287/opre.26.1.127
20. J. M. Lopez, M. Garcia, J. L. Diaz, and D. F. Garcia, "Worst-case utilization bound for edf scheduling on real-time multiprocessor systems," in *12th ECRTS*, 2000, pp. 25–33.
21. H. Takada and K. Sakamura, "A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels," in *Proceedings Real-Time Systems Symposium*, 1997, pp. 134–143.
22. E. W. Dijkstra, "Cooperating sequential processes," in *Programming Languages: NATO Advanced Study Institute*, F. Genuys, Ed. Academic Press, 1968, pp. 43–112.

23. B. B. Brandenburg and J. H. Anderson, "An implementation of the pcp, srp, d-pcp, m-pcp, and fmlp real-time synchronization protocols in litmusrt," in *14th IEEE RTCSA*. USA: IEEE, 2008, pp. 185–194.
24. EPOS. (2020, Aug) Website. [Online]. Available: http://epos.lisha.ufsc.br
25. G. Gracioli, A. A. Fröhlich, R. Pellizzoni, and S. Fischmeister, "Implementation and evaluation of global and partitioned scheduling in a real-time OS," *Real-Time Systems*, vol. 49, no. 6, 2013.
26. A. Biondi and B. B. Brandenburg, "Lightweight real-time synchronization under p-edf on symmetric and asymmetric multiprocessors," in *Proc. of the ECRTS 2016*, 2016, pp. 1–11.
27. K. Lakshmanan, D. d. Niz, and R. Rajkumar, "Coordinated task scheduling, allocation and synchronization on multiprocessors," in *2009 30th IEEE Real-Time Systems Symposium*, 2009, pp. 469–478.
28. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, p. 46–61, 1973.
29. (2020, Aug) Schedcat: Schedulability test collection and toolkit. [Online]. Available: http://www.mpi-sws.org/bbb/projects/schedcat
30. (2016, Jul) Freertos web-site. [Online]. Available: http://www.freertos.org/
31. R. Inam, J. Mäki-Turja, M. Sjödin, and M. Behnam, "Hard real-time support for hierarchical scheduling in freertos," in *OSPERT*, 2011, pp. 51–60.
32. J. Liedtke, "On micro-kernel construction," in *15th SOSP*. ACM, 1995, pp. 237–250.
33. J.-H. Lee and H.-N. Kim, "Implementing priority inheritance semaphore on uc/os real-time kernel," in *IEEE Workshop on Software Technologies for Future Embedded Systems, 2003*, May 2003, pp. 83–86.
34. B. E. S. Akgul, V. J. M. III, H. Thane, and P. Kuacharoen, "Hardware support for priority inheritance," in *24th IEEE Real-Time Systems Symposium, 2003. RTSS 2003*, Dec 2003, pp. 246–255.
35. H. Marcondes and A. A. Fröhlich, *A Hybrid Hardware and Software Component Architecture for Embedded System Design*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 259–270.
36. Q. Wang, J. Song, and G. Parmer, "Execution stack management for hard real-time computation in a component-based os," in *IEEE 32nd RTSS*, Nov 2011, pp. 78–89.
37. M. Caccamo, G. Lipari, and G. Buttazzo, "Sharing resources among periodic and aperiodic tasks with dynamic deadlines," in *Proc. 20th IEEE RTSS*, 1999, pp. 284–293.
38. A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, "A flexible real-time locking protocol for multiprocessors," in *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, 2007, pp. 47–56.
39. A. Easwaran and B. Andersson, "Resource sharing in global fixed-priority preemptive multiprocessor scheduling," in *30th RTSS*, 2009, pp. 377–386.
40. R. Teixeira and G. Lima, "Improved task packing for shared resources in multiprocessor real-time systems scheduled by run under sblp," in *2019 IX Brazilian Symposium on Computing Systems Engineering (SBESC)*, 2019, pp. 1–8.
41. P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in *2011 IEEE 32nd Real-Time Systems Symposium*, 2011, pp. 104–115.
42. B. B. Brandenburg, "Multiprocessor real-time locking protocols: A systematic review," *CoRR*, vol. abs/1909.09600, 2019.
43. J. Robb and B. B. Brandenburg, "Nested, but separate: Isolating unrelated critical sections in real-time nested locking," in *32nd ECRTS*, 2020, pp. 6:1–6:23.
44. C. E. Nemitz, T. Amert, and J. H. Anderson, "Using Lock Servers to Scale Real-Time Locking Protocols: Chasing Ever-Increasing Core Counts," in *30th ECRTS*, 2018, pp. 25:1–25:24.
45. T. Craig, "Queuing spin lock algorithms to support timing predictability," in *1993 Proceedings Real-Time Systems Symposium*, 1993, pp. 148–157.
46. H. Takada and K. Sakamura, "Real-time scalability of nested spin locks," in *Proceedings Second International Workshop on Real-Time Computing Systems and Applications*, 1995, pp. 160–167.

47. S. Zhao and A. J. Wellings, "Investigating the correctness and efficiency of mrsp in fully partitioned systems," in *10th York Doctoral Symposium on Computer Science and Electronic Engineering*, 2017.