On the Design and Implementation of Real-Time Resource Access Protocols

Conference Paper · November 2020

DOI: 10.1109/SBESC51047.2020.9277858

5 authors, including:

Lucas Matheus dos Santos
Federal University of Santa Catarina
1 PUBLICATION 0 CITATIONS
SEE PROFILE

Giovani Gracioli
Federal University of Santa Catarina
35 PUBLICATIONS 262 CITATIONS
SEE PROFILE

Tomasz Kloda
Technische Universität München
14 PUBLICATIONS 61 CITATIONS
SEE PROFILE

On the Design and Implementation of Real-Time Resource Access Protocols

Lucas Matheus dos Santos[1], Giovani Gracioli[1], Tomasz Kloda[2], and Marco Caccamo[2]
[1]Federal University of Santa Catarina, Brazil, {lsantos,giovani}@lisha.ufsc.br
[2]Technical University of Munich, Germany, {mcaccamo,tomasz.kloda}@tum.de

***Abstract*—Real-time operating systems (RTOS) should support resource access protocols to bound the maximum delay incurred by priority inversions. The implementation of such protocols must be lightweight because its performance affects the system schedulability. In this paper, we present an object-oriented design of real-time resource access protocols for single- and multi-core systems aiming at reducing the run-time overhead and increasing code re-usability. We implement the proposed design in an RTOS and measure the memory footprint and run-time overhead of the implementation in a modern processor. By applying the obtained overhead into the schedulability analysis of six protocols for synthetically generated task sets, our results indicate that proper implementation of resource access protocols has a low impact on the schedulability of real-time tasks.**

***Index Terms*—Real-time resource access protocols, real-time operating systems, priority ceiling protocol, priority inheritance protocol, stack resource policy;**

I. Introduction

Any concurrent operating system (OS) must offer protocols to control the access to resources shared by competing tasks, thus ensuring *mutual exclusion* in their respective critical sections [1, 2]. Typically, mutual exclusion is guaranteed by the use of binary semaphores, such as mutexes and suspension-based locks [3]. 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 [1]–[3]. For instance, consider a high-priority task τ_H and a low-priority task τ_L that share a resource. It might happen that τ_L is holding the resource, but is preempted by τ_H. Then, τ_H executes until it tries to enter the critical section. However, τ_H cannot continue, because the resource is already in use by τ_L. Thus, a high-priority task is blocked by a low-priority task. Worse, if a task τ_M with a priority higher than τ_L but lower than τ_H is released and allowed to preempt τ_L, the blocking delay can be unpredictable.

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 [4] 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. Ceiling- and priority inheritance-based protocols [5] are typically used in static scheduling, while the Stack Resource Protocol (SRP) [6] is typically used in dynamic scheduling. Also, there are extensions of these protocols for multi-core systems, such as the Multiprocessor Priority Ceiling Protocol (MPCP) [7] and the Multiprocessor Stack Resource Policy (MSRP) [8].

In this work, we present a design and implementation of resource access protocols for single- and multi-core systems, considering both static and dynamic scheduling approaches. More specifically, we present a design for the Priority Inheritance Protocol (PIP) [5], Priority Ceiling Protocol (PCP) [5], Immediate Priority Ceiling Protocol (IPCP), SRP [6], MPCP and MSRP. To the best of our knowledge, this is the first work to design and evaluate single- and multi-core variations of real-time resource access protocols in a real-time operating system (RTOS) designed from scratch. 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.

Our design uses object-oriented techniques to maximize software reuse and minimize run-time overhead. We have implemented the proposed software design in an RTOS and evaluated the memory footprint and run-time overhead in a modern processor. Our results indicate that the proposed design allows software reuse and low overhead, providing schedulability ratios close to the theoretical ones. In summary, we make the following contributions:

- We propose an object-oriented design for single- and multi-core real-time resource access protocols. The focus is on providing software reuse and low run-time overhead;
- We implement the proposed design in an RTOS and evaluate the memory footprint of this implementation as well as its run-time overhead on a modern processor. The maximum obtained overhead is less than 4 µs, considering both *p* and *v* semaphore operations;
- A comparison in terms of a task set schedulability ratio among the protocols, considering their run-time overheads. Our results indicate that the schedulability ratio remains close to theoretical bounds, proving the efficiency of the proposed design.

II. System Model and Background

A. 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) [9] and Partitioned Earliest Deadline First (P-EDF) [10]. 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 iff $j > i$. 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.

B. 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 [1, 2].

PIP is a classic mechanism for sharing resources in a single-processor with FP scheduling [5]. 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 [5]. 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) [6]. 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.

C. Multi-Core Resource Access Protocols

MPCP [7] 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 [8] 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.

III. Design and Implementation of the Protocols

Figure 1 presents an overview of the proposed software design for the resource access protocols through a UML class diagram. We have analyzed the common characteristics related to the protocols to achieve the final design, which has a total of 12 classes detailed below. The design is motivated by the following goals: i) to allow code reuse; ii) to be extensible to new protocols; and iii) to have low memory consumption and run-time overhead. We detail the design below.

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 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 [11]. 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.

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_PIP` class implements the priority inheritance behavior of the PIP protocol. It overwrites the *p* and *v* methods from the `Semaphore` to include the handling of the priority inheritance. For the *p* operation, the class first checks whether there is a thread inside the critical section by verifying if `_owner` is zero. If so, the calling thread becomes the new semaphore's owner. If not, there is a test to check whether the calling thread priority is greater than the owner's priority. If it is, the owner has its priority raised to the calling thread's priority. Then, the *p* operation from the `Semaphore` is called to conclude the work. This is all done in an atomic state (*i.e.,* interrupts are disabled). For the *v* operation, if there is a semaphore owner, `Semaphore_PIP` checks if there is another thread waiting on the Semaphore's queue to enter the critical section. If so, the `_owner` and `_priority` are updated. If not, `_owner` and `_priority` receives zero as value. Then, the *v* operation from the `Semaphore` is called to conclude the work. The instructions within the *v* are also performed with interrupts disabled.

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. It also has the set and get methods for the attribute. The `Semaphore_PCP` class implements the PCP protocol by overwriting the *p* and *v* methods. The class controls the rule to grant a lock request by comparing the current task priority with the highest ceiling among all locked semaphore. The class also implements the priority inheritance mechanism defined in PCP and priority transitivity. If the semaphore has no owner, then the current thread becomes the new owner and the semaphore's priority is set to the thread's priority. After that, the *p* method of the `Semaphore` is called. The *v* operation reestablishes the owner thread's priority (in case it was raised) and checks whether there is another thread waiting on the semaphore's queue. Then, it calls the *p* method of the `Semaphore`.

The `Semaphore_IPCP` class implements the IPCP protocol in a similar way to the `Semaphore_PCP` class. The only difference is that the owner's priority is raised to the semaphore's ceiling whenever a thread enters the critical section. Also, the owner's priority is always reestablished to its original

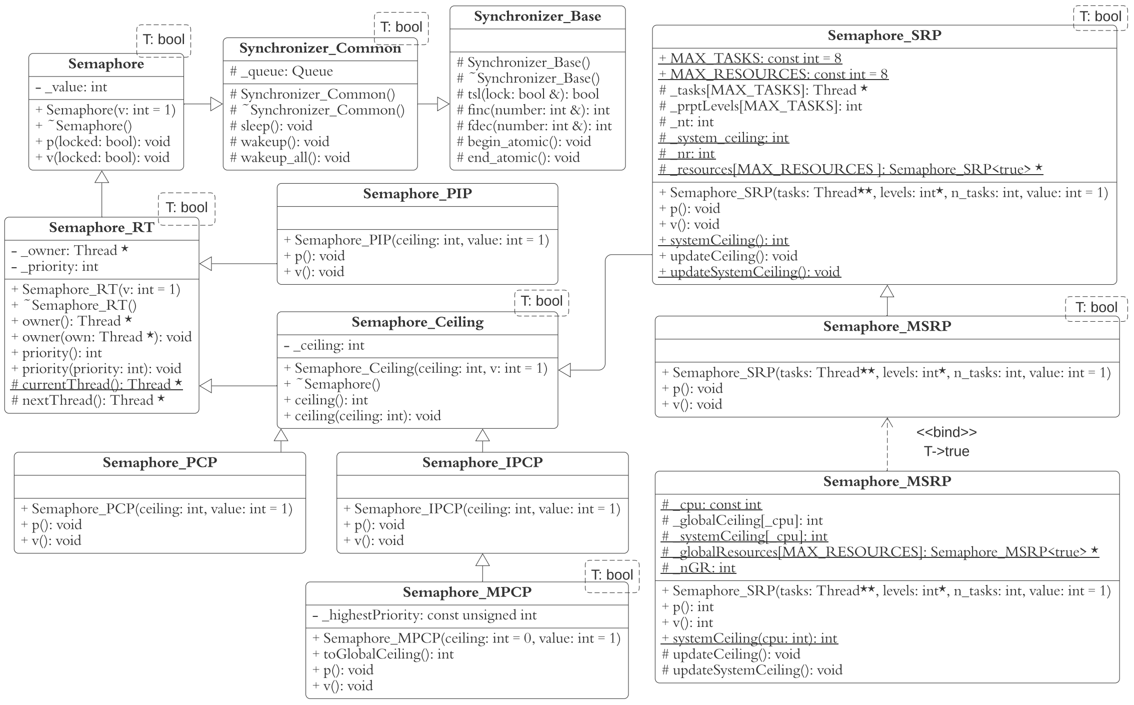

Semaphore
T: bool
- _value: int
+ Semaphore(v: int = 1)
+ ~Semaphore()
+ p(locked: bool): void
+ v(locked: bool): void
Synchronizer_Common
# _queue: Queue
# Synchronizer_Common()
# ~Synchronizer_Common()
# sleep(): void
# wakeup(): void
# wakeup_all(): void
Synchronizer_Base
# Synchronizer_Base()
# ~Synchronizer_Base()
# tsl(lock: bool &): bool
# finc(number: int &): int
# fdec(number: int &): int
# begin_atomic(): void
# end_atomic(): void
Semaphore_SRP
+ MAX_TASKS: const int = 8
+ MAX_RESOURCES: const int = 8
# _tasks[MAX_TASKS]: Thread *
# _prptLevels[MAX_TASKS]: int
# _nt: int
# _system_ceiling: int
# _nr: int
# _resources[MAX_RESOURCES]: Semaphore_SRP<true> *
+ Semaphore_SRP(tasks: Thread**, levels: int*, n_tasks: int, value: int = 1)
+ p(): void
+ v(): void
+ systemCeiling(): int
+ updateCeiling(): void
+ updateSystemCeiling(): void
Semaphore_RT
- _owner: Thread *
- _priority: int
+ Semaphore_RT(v: int = 1)
+ ~Semaphore_RT()
+ owner(): Thread *
+ owner(own: Thread *): void
+ priority(): int
+ priority(priority: int): void
# currentThread(): Thread *
# nextThread(): Thread *
Semaphore_PIP
+ Semaphore_PIP(ceiling: int, value: int = 1)
+ p(): void
+ v(): void
Semaphore_Ceiling
- _ceiling: int
+ Semaphore_Ceiling(ceiling: int, v: int = 1)
+ ~Semaphore()
+ ceiling(): int
+ ceiling(ceiling: int): void
Semaphore_MSRP
+ Semaphore_SRP(tasks: Thread**, levels: int*, n_tasks: int, value: int = 1)
+ p(): void
+ v(): void
<<bind>>
T->true
Semaphore_PCP
+ Semaphore_PCP(ceiling: int, value: int = 1)
+ p(): void
+ v(): void
Semaphore_IPCP
+ Semaphore_IPCP(ceiling: int, value: int = 1)
+ p(): void
+ v(): void
Semaphore_MPCP
- _highestPriority: const unsigned int
+ Semaphore_MPCP(ceiling: int = 0, value: int = 1)
+ toGlobalCeiling(): int
+ p(): void
+ v(): void
Semaphore_MSRP
# _cpu: const int
# _globalCeiling[_cpu]: int
# _systemCeiling[_cpu]: int
# _globalResources[MAX_RESOURCES]: Semaphore_MSRP<true> *
# _nGR: int
+ Semaphore_SRP(tasks: Thread**, levels: int*, n_tasks: int, value: int = 1)
+ p(): int
+ v(): int
+ systemCeiling(cpu: int): int
# updateCeiling(): void
# updateSystemCeiling(): void

Figure 1. UML class diagram of the proposed design for the synchronization protocols.

priority whenever it calls *v*.

Semaphore_MPCP is a template class implemented in a similar way to the Semaphore_IPCP class. The template expects a boolean value, and the user must choose it according to the type of critical section that is going to be guarded. 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. Semaphore_MPCP has a private attribute named _highestPriority which defines, at compile-time, the highest priority of all tasks that use global resources among all cores. If the class is used to guard a local critical section, it behaves like IPCP. When guarding a global critical section, on *p* operation, the class raises the owner priority to the _highestPriority instead of _ceiling priority.

The Semaphore_SRP class implements SRP. It is statically configured at compile-time to support a given number of tasks per resource and the number of resources in the system. Every instance has a resource ceiling (inherits Semaphore_Ceiling), which corresponds to the highest preemption level amongst the tasks that would block if accessing the resource. Whenever the semaphore value changes, the resource recalculates its resource ceiling and also the system ceiling, according to the algorithm rules.

To make this dynamic accounting possible, the semaphore stores a list of tasks that access the resource, and the maximum number of resource units the task can hold at once (_prio_levels attribute) and also keeps track of all existing SRP resources (static attribute _resources).

The Semaphore_MSRP is a template class that implements MSRP. Its template parameter is similar to the MPCP one, a boolean indicating protection of a local or global critical section (GCS). When true is passed, the semaphore implements *p* and *v* operations for MSRP. Otherwise, the protocol works as SRP. The class inherits Semaphore_SRP passing true as its template argument to use a queue ranked by priority, whenever a local critical section is going to be guarded. It passes false to SRP template argument whenever a GCS is going to be protected, indicating the use of a FIFO queue.

MSRP is also statically configured at compile-time to support a predetermined number of tasks per resource and a maximum amount of resources in the system. MSRP stores a ceiling value for every core that uses the resource at _globalCeiling attribute. By having reference to every GCS created (within the _globalResources array), we can calculate the system ceiling for every processor at the system.

Whenever the semaphore value changes, the _globalCeiling and _systemCeiling are recalculated. When a task tries to preempt another one to enter a GCS, the scheduler tests if the task is eligible by comparing the task's preemption level with the system ceiling of the processor that the task is running on. If the task's preemption level is higher than the current system ceiling for that core, the task gets access to the resource. Otherwise, the task is inserted in a FCFS queue with other tasks that had tried and failed to get access to the GCS.

The classes Synchronizer_Common, Synchronizer_Base, Semaphore_RT, and Semaphore_Ceiling are the main components in the design that allow code reuse and extensibility. To add a new protocol to the system, one of these classes should be extended accordingly to the target protocol. Moreover, the design allows the RTOS to easily change the target protocol, because the interface is the same (just need to adjust the offline configuration parameters, like the numbers of tasks and resources in `Semaphore_SRP` and `Semaphore_MSRP` classes for instance).

A. *Implementation in an RTOS*

We have implemented the described design in the Embedded Parallel Operating System (EPOS) [12]. EPOS is the first open-source RTOS designed from scratch that supports partitioned, global, and clustered versions of EDF, RM, LLF, and DM scheduling policies and it is written in C++ [13]. A complete review of the real-time support on EPOS can be found in [13]. We choose EPOS because it supports static and dynamic scheduling and it is written in an object-oriented language. Thus, it is compatible with our proposed design. Moreover, EPOS until this work did not have complete support for real-time resource access protocols. 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.

IV. Experimental Evaluation

This section describes the experimental evaluation of the previously described implementation. Our objectives are threefold: i) to measure the memory consumption of the implementation; ii) to evaluate the run-time overhead of the implementation; and iii) to verify the impact of the run-time overhead into the system schedulability. The next subsections show the obtained results for the three objectives.

A. *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 I shows the obtained memory footprint for each class depicted in Figure 1. 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 consumptions. It is important to highlight that code from the `Synchronizer_Base` and `Synchronizer_-Common` classes are not represented in the Table I, because it is inlined into the methods that use the base class. It means that its code is 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 III.

Table I
Memory footprint (in bytes) and lines of code of the implemented classes.

Class	Code	Data	Static Data	Total Mem.	Lines of Code	
				Consum.	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

B. *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 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.

The p and v methods of every semaphore type were instrumented, as well as the system code responsible for queuing/dequeuing tasks and performing a rescheduling, 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 2 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. The overheads 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 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,

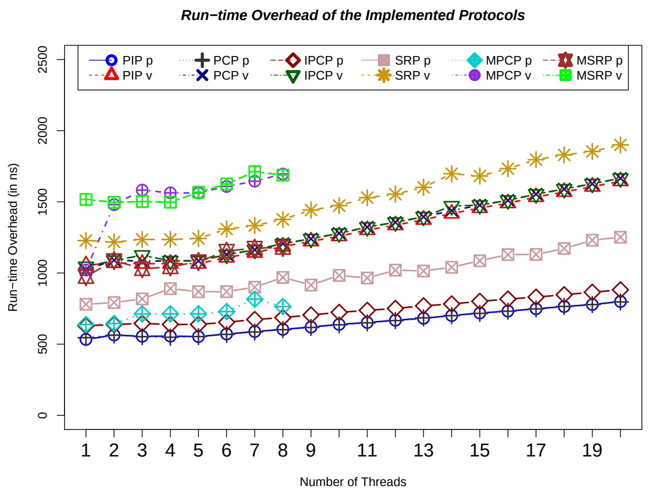


Figure 2. Obtained run-time overhead (on the Intel i7 processor).

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 [13], in the schedulability impact discussed next.

C. 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 [3, 14].

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 [10]. We acknowledge that better results could be achieved with specialized resource-sharing-aware heuristics [15]. 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 [2, 16]. For the fixed-priority scheduling, the task priorities were assigned by *Rate Monotonic* [16] (*RM* and *P-RM*).

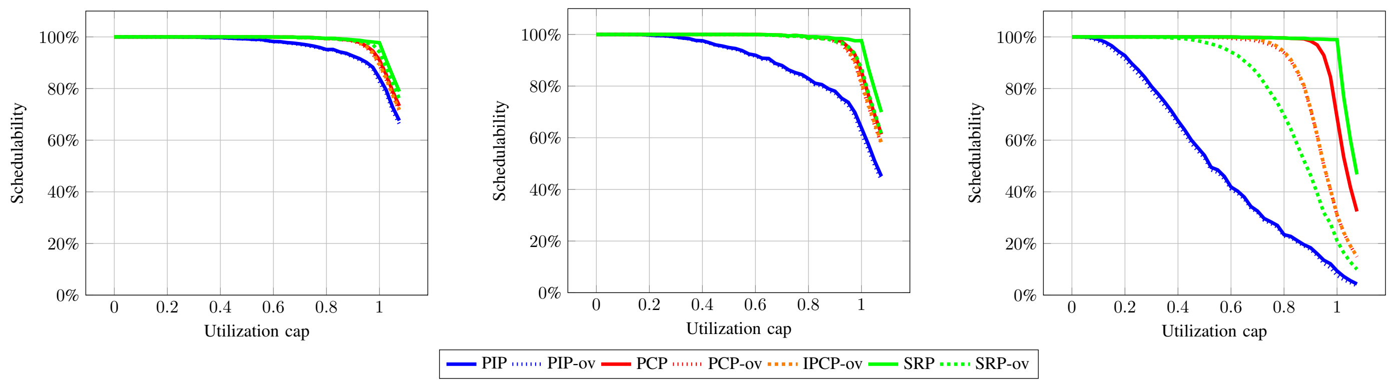
We analyzed four resource access protocols on a single-core platform ($m = 1$) with the associated analyses for blocking: PIP [5], PCP and IPCP [5] (all for FP) and SRP [6] (for EDF). On multi-core platforms ($m > 1$), we studied MPCP [15] under P-RM and MSRP [8] under P-EDF. Task set generation and schedulability tests were implemented in *SchedCAT* [17].

For each combination of the described above parameters, we generated 1000 task sets increasing the task set utilization cap by 0.05 at each step. The schedulability of each task set was then verified (solid lines on Figures 3 and 4). Additionally, we considered the overheads (dotted lines) by inflating each task's WCET with the obtained run-time overheads (see Figure 2) and context-switch penalties incurred by each protocol [6] (dotted lines on Figures 3 and 4). The experiments covered more than 2500 cases. Here, we report on our main findings[1].

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. Figure 3 shows 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 two protocols for multi-processor: MPCP under *P-RM* and MSRP under *P-EDF*. Figure 4 shows 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 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.

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



a. *heavy*, $p^{acc} = 0.25$, $n_{res} = 4$ b. *medium*, $p^{acc} = 0.5$, $n_{res} = 8$ c. *light*, $p^{acc} = 0.75$, $n_{res} = 16$

Figure 3. Selected results for single-processor setup (periods heterogeneous, long critical sections).

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; and v) due to higher overhead, even with better schedulability, MSRP can be worse than MPCP.

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.

V. Related Work

Suspension-based resource access protocols for uniprocessor real-time systems were first proposed by Sha et al. [5]. 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 [6] and it was also a seminal work, mainly for providing resource access protection for dynamic scheduling.

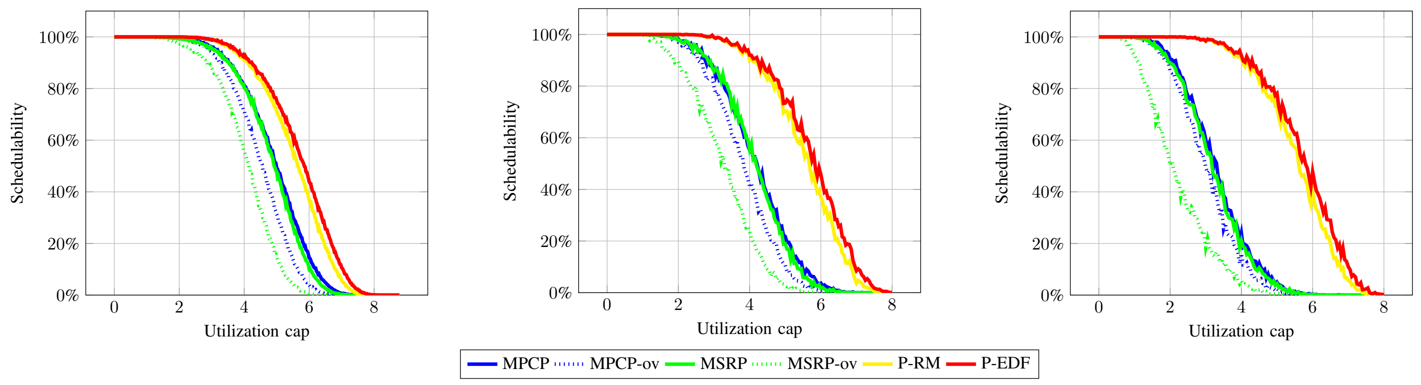
Several general-purpose OSes and RTOSes implement some of the analyzed real-time resource access protocol. For instance, FreeRTOS employs PIP in the mutex primitive [18] and also supports SRP [19]. $LITMUS^{RT}$ supports PCP, SRP, and several multiprocessor resource access protocols [20]. The L4 microkernel [21] 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 [22]. The observed run-time overhead for the *p* and *v* semaphore operations was 30.5 µs.

Wang et al. implemented multi-resource versions of PIP and PCP in a component-based OS for controlling the access to shared stacks [23]. 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 [23]. 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. [24] proposed a scheduling framework for soft- and hard-real time tasks that share the same resources.

Resource access protocols for multiprocessors have been the subject for many researchers recently. MPCP [7] and MrsP [25] are two multiprocessor variants of the PCP. Flexible Multiprocessor Locking Protocols is a collection of protocols for global and partitioned scheduling [26]. 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 [27]. Biondi and Brandenburg [14] 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 [28] proposed a task allocation heuristic for global scheduling that leverages the concept of servers [29]. Yang et al. proposed new more precise schedulability analyses based on linear programming for several multiprocessor semaphore-based locking approaches [3]. Brandenburg surveys multiprocessor real-time locking protocols, from 1988 until 2018 [30]. 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 [31]. 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 [32].

VI. Conclusion

This paper presented an object-oriented design of six real-time resource protocols (PIP, PCP, IPCP, and SRP for single-core systems and MPCP and MSRP for multi-core systems). We have implemented the proposed design in an RTOS and



a. $n_{res} = 2$ b. $n_{res} = 4$ c. $n_{res} = 8$

Figure 4. Selected results for multi-processor setup (utilization medium, periods heterogeneous, $p^{acc} = 0.75$, short critical sections, $m = 8$ processors).

measured the memory footprint and run-time overhead of the implementation. By compiling and running our implementation in a modern multi-core processor, we proved that it allows small memory footprint (from 876 to 1674 bytes, depending on the protocol) and run-time overhead (less than 4 µs for 20 tasks). 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.

Acknowledgment

Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research.

References

[1] J. Liu, *Real-Time Systems*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.
[2] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 3rd ed. Springer, 2011.
[3] M. Yang, A. Wieder, and B. Brandenburg, "Global real-time semaphore protocols: A survey, unified analysis, and comparison," in *RTSS*, 2015, pp. 1–12.
[4] 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
[5] 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.
[6] T. P. Baker, "Stack-based scheduling for realtime processes," *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, Apr. 1991.
[7] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *10th ICDCS*, May 1990, pp. 116–123.
[8] 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.
[9] 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
[10] 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.
[11] E. W. Dijkstra, "Cooperating sequential processes," in *Programming Languages: NATO Advanced Study Institute*, F. Genuys, Ed. Academic Press, 1968, pp. 43–112.
[12] EPOS. (2020, Aug) Website. [Online]. Available: http://epos.lisha.ufsc.br
[13] 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.
[14] 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.
[15] 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.
[16] 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.
[17] (2020, Aug) Schedcat: Schedulability test collection and toolkit. [Online]. Available: http://www.mpi-sws.org/bbb/projects/schedcat
[18] (2016, Jul) Freertos web-site. [Online]. Available: http://www.freertos.org/
[19] 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.
[20] 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.
[21] J. Liedtke, "On micro-kernel construction," in *15th SOSP*. ACM, 1995, pp. 237–250.
[22] 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.
[23] 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.
[24] 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.
[25] A. Burns and A. J. Wellings, "A schedulability compatible multiprocessor resource sharing protocol – mrsp," in *ECRTS*, July 2013, pp. 282–291.
[26] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, "A flexible real-time locking protocol for multiprocessors," in *13th IEEE RTCSA*, Aug 2007, pp. 47–56.
[27] A. Easwaran and B. Andersson, "Resource sharing in global fixed-priority preemptive multiprocessor scheduling," in *30th RTSS*, 2009, pp. 377–386.
[28] 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.
[29] 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.
[30] B. B. Brandenburg, "Multiprocessor real-time locking protocols: A systematic review," *CoRR*, vol. abs/1909.09600, 2019.
[31] 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.
[32] 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.