Journal of Systems Architecture 60 (2014) 258–270

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Exploring the design space of multiprocessor synchronization protocols for real-time systems

Andreu Carminati [a,*], Rômulo Silva de Oliveira [a], Luís Fernando Friedrich [b]

[a] Universidade Federal de Santa Catarina, DAS-CTC-UFSC, Caixa Postal 476, Florianópolis, SC, Brazil
[b] Universidade Federal de Santa Catarina, INE-CTC-UFSC, Caixa Postal 476, Florianópolis, SC, Brazil

ARTICLE INFO

Article history:
Available online 7 December 2013

Keywords:
Real-time
Scheduling
Synchronization
Multiprocessors

ABSTRACT

The goal of this paper is to explore the design space of protocols for multiprocessor systems with static priority and partitioned scheduling. The design space is defined by a set of characteristics that can vary from one protocol to another. This exploration presents new protocols with different characteristics from existing ones. These new protocols are considered variations of the Multiprocessor Priority Ceiling Protocol (MPCP), but they can also be seen as variations of the Flexible Multiprocessor Locking Protocol (FMLP), since they include features common to both protocols. Schedulability tests are provided for these new variations and they are compared with the original versions of MPCP and FMLP. Such comparisons include an empirical comparison of schedulability and an overhead evaluation of a real implementation. Such comparisons show that these new variations are actually competitive in relation to the existing protocols.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

In recent years, the increasing use of multiprocessor systems has motivated the fast development of many solutions for the scheduling of tasks in these systems in the presence of real-time requirements. One of the greatest challenges of real-time scheduling is to synchronize the access to mutual-exclusive resources efficiently, since that the existing synchronization protocols for uniprocessor systems can not be directly extended to multiprocessors. In this context, to schedule a set o real-time tasks on a multiprocessor system, we must limit the blocking time of each of these tasks using an appropriated synchronization protocol. Without a bounded blocking time, a task cannot have its deadline restrictions guaranteed.

The goal of this paper is to explore the design space of synchronization protocols for real-time systems in multiprocessor environment. The design space is considered as a set of basic characteristics, that include: queuing policy in case of blocking (FIFO, priority order, etc.), preemptability of critical sections, and execution control policy in case of blocking (suspension or spin.) In this context, a protocol can be seen as a carefully selected set of characteristics from the design space of protocols. Existing protocols do not explore all possible features within the design space. With this exploration of characteristics as premise, we propose new protocols that will be treated as variations of the Multiprocessor Priority Ceiling Synchronization Protocol (MPCP.) The MPCP was originally proposed by Rajkumar et al. [1,2] for multiprocessor systems with static priority and partitioned scheduling, where tasks are statically allocated to processors. These new variations can also be seen as variations of FMLP [3], since they include features common to both protocols.

Schedulability tests are provided for these new variations and they are compared with two existing protocols for the same system model (partitioned and static priority scheduling): the Multiprocessor Priority Ceiling Protocol for Shared Memory (MPCP) and the Flexible Multiprocessor Locking Protocol (FMLP). In this paper we consider only global resources (which are accessed by more than one processor) in both equations and in the empirical comparison of protocols. Local resources can be handled by protocols designed for uniprocessor systems, which has been extensively studied in the literature. This is the usual approach of the literature on multiprocessor synchronization protocols.

The objective of this paper is not to propose better protocols than the existing ones for all system configurations. Previous studies have shown that, for multiprocessor systems, there is not a synchronization protocol that dominates all others in all situations (different task sets and system models). It is always possible to hand craft a task set that favors one or another existing protocol. We present in this paper variations of the MPCP that improve in some cases the system schedulability reducing the number of processors needed to schedule a system. Some variations are actually simplifications that favor the implementation in real systems.

* Corresponding author. Tel.: +55 04896318322.
E-mail address: andreu@das.ufsc.br (A. Carminati).

1383-7621/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.sysarc.2013.11.010

The rest of the paper is organized as follows: Section 2 presents the system model and notations used throughout the text. In Section 3 we present a review of the MPCP protocol. In Section 4 and 5 we do the same for the FMLP and MSRP protocols, respectively. Section 6 presents our proposals. In Section 7 we show a comparison between the proposed solutions and existing protocols. Section 8 presents an implementation of one variation, which will be used in Section 9 for overhead evaluation. Finally, Section 10 presents the conclusions of this paper.

2. System model

In this work it is considered only partitioned and static priority scheduling of periodic task sets. In partitioned scheduling, tasks are statically allocated to processors via dedicated queues. Global scheduling will not be considered in this work. The partitioned scheduling with static priority is also known as P-SP (partitioned static priority). This type of scheduling is addressed by uniprocessor algorithms in each processor, so the main problem is how to perform the partitioning of the tasks. The partitioning of the tasks among processors is effectively the Bin Packing problem, as pointed by Coffman et al. [4], which is NP-hard with combinatorial complexity. One possible strategy is to partition according to some heuristics, such as BFD (Best-fit decreasing), RM-FFDU described by Oh et al. [5] or group by resource usage as in [6]. In this work we do not explicitly consider nested critical sections and deadlock avoidance. It is perfectly possible to do as in [3] where techniques such as group locking are used with synchronization protocols to allow nested critical sections.

2.1. System definition

The notation that will be used throughout the text is the same as in [6], because we based the schedulability analysis of the protocols in that article. Task τ_i is defined by the tuple $((C_{i,1}, C'_{i,1}, C_{i,2}, C'_{i,2}, \ldots, C'_{i,s(i)-1}, C_{i,s(i)}), T_i)$ where:

- $s(i)$ is the number of normal execution segments of τ_i, whereas $s(i)-1$ is the number of critical section segments of τ_i.
- $C_{i,j}$ is the WCET of the jth normal execution segment, whereas $C'_{i,j}$ is the WCET of the jth critical section segment.
- C_i is the total WCET of task τ_i. In this notation, the execution time of each task is segmented into sections of normal execution and critical sections. The worst-case execution time of a task τ_i can be calculated as the sum of all task segments, being they normal or critical. The task execution time can be calculated by the following equation:

$$C_i = \sum_{j=1}^{s(i)} C_{i,j} + \sum_{k=1}^{s(i)-1} C'_{i,k}. \tag{1}$$

- $\tau_{i,j}$ is the jth normal execution segment of a task τ_i, whereas $\tau'_{i,j}$ is the jth critical section segment of a task τ_i.
- T_i is the period of τ_i.
- $R(\tau_{i,j})$ is the resource corresponding to the jth critical section segment of a task τ_i.
- $P(\tau_i)$ is the processor that executes task τ_i.Given two tasks τ_i and τ_j, if $i < j$ then the priority of τ_i is higher than the priority of τ_j.

2.2. Blocking aware response time analysis

Based on the notation presented and the blocking times (which will be presented later), one can calculate the worst-case response time (WCRT) W_i of a task τ_i.

The equations described in this section were presented in [6,7]. The WCRT of a task is calculated iteratively:

$$W_i^{n+1} = C_i + B_i^r + I_i^n + B_i^{low}. \tag{2}$$

In Eq. (2), C_i is the task WCET, B_i^r is the total remote blocking time (that is dependent of the synchronization protocol utilized), I_i^n represents the interference imposed by higher priority tasks. B_i^{low} represents the blocking time imposed by lower priority tasks. This blocking time exists because a task can be prevented from executing at its release as the result of a lower priority task running on a non-preemptive way (inside a critical section in this case). The initial value of the convergence must be $W_i^0 = C_i + B_i^r$.

The calculation of interference follows two approaches, one for suspension-based protocols and another for spin-based protocols:

- For suspension-based protocols (tasks are suspended and they do not spin-lock), the calculation of interference must be done by Eq. (3).

$$I_i^n = \sum_{h<i \& \tau_h \in P(\tau_i)} \left\lceil \frac{W_i^n + B_h^r}{T_h} \right\rceil C_h. \tag{3}$$

In Eq. (3), the blocking times of a higher priority task may increase the WCRT of a lower priority task. This equation captures an effect called back-to-back execution pointed by Rajkumar [8] and Lakshmanan et al. [6], where a task suffers additional interference from auto-suspending (when blocked on some mutex) higher-priority tasks (not accounted in normal interference scheduling analysis). This happens because a higher priority task can suspend itself in the middle of its execution and come to preempt a lower priority task more than once during its activation. An upper bound for this type of interference is B_h^r, that must be added to W_i^n in a way similar to release jitter modeling.

- For spin-based protocols, the calculation of interference must be done by the following equation:

$$I_i^n = \sum_{h<i \& \tau_h \in P(\tau_i)} \left\lceil \frac{W_i^n}{T_h} \right\rceil (C_h + B_h^r). \tag{4}$$

In Eq. (4), the spin time of a task (when blocked) appears as an increasing on its own computing time from the standpoint of lower priority tasks. So, this time must be summed to the computing time of each higher priority tasks, as can be seen in Eq. (4).

The calculation of the blocking time caused by lower priority tasks must follow three approaches, one for suspension-based protocols and two for spin-based protocols:

- For suspension-based protocols, such calculation can be done by the following equation:

$$B_i^{low} = s(i) \times \sum_{l>i \& \tau_l \in P(\tau_i)} \max_{1 \le k < s(l)} C'_{l,k}. \tag{5}$$

In Eq. (5), every time a task blocks (in the worst-case, a task will block on every resource acquisition attempt) and also before its activation, it will allow lower priority tasks to execute. This lower priority tasks can block on resources. When the lower priority task receives the resource, it will preempt the higher priority tasks, because all synchronization protocols that use suspension execute critical sections with a priority higher than normal priorities. This equation offers an upper bound for this blocking.

- For spin-based protocols, this calculation varies whether the protocol treats blocking with preemptive or non preemptive spin:

- For preemptive spin: when the spin is done in a way to prevent the execution of lower priority tasks, the blocking times must be calculated with the following equation:

$$B_i^{low} = \sum_{l>i \& \tau_l \in P(\tau_i)} \max_{1 \leqslant k < s(l)} C'_{l,k}. \quad (6)$$

In Eq. (6), when a task blocks, it inhibits the execution and subsequent blocking of lower priority tasks. However, before the activation of the task at hand, in the worst case, there may be a chain of blockings, as in the following example:

- Task A, with lowest priority, blocks on a resource and starts its spin.
- Task B, with the second lowest priority, preempts Task A and blocks on a resource, and so on with Task C (higher priority than B) and D (higher priority than C).
- Task E (highest priority) begins its execution, with all other tasks in preemptive spin. During the execution of Task E, it will be preempted because all blocked tasks (A, B, C and D) are unblocked to use their resources with increased priority, imposing the blocking.
- For non preemptive spin: when the spin is done in a way to prevent the execution of any task, the blocking times must be calculated with the following equation:

$$B_{low} = \max_{l>i \& \tau_l \in P(\tau_i) \& 1 \leqslant k < s(l)} (C'_{l,k} + B^r_{l,k}). \quad (7)$$

In Eq. (7), when a task is ready to execute, it can be prevented from executing by a lower priority task performing spin and/or executing a critical section. The task in question must wait the lower priority task exit the critical section before starting its execution. An upper bound for this blocking is the larger sum of spin time and critical section time among all lower priority tasks, as showed in Eq. (7), where $B^r_{l,k}$ is the blocking of the kth critical section of τ_l.

$B^r_{i,j}$ defines the remote blocking corresponding to the largest waiting time experienced by task τ_i when it requests the jth global critical section. The total remote blocking of a task τ_i is denoted by B^r_i and can be calculated using the following equation, which sums all possible blocking times of each critical section (Eq. (8)):

$$B^r_i = \sum_{q=1}^{s(q)-1} B^r_{i,q}. \quad (8)$$

How the values of $B^r_{i,q}$ are calculated depends on each protocol. Several factors can influence this calculation, such as queuing policy in case of blocking and preemptability of critical sections. The following sections present a review/presentation of existing protocols.

3. Review of the multiprocessor priority ceiling protocol for shared memory

This protocol was proposed by Rajkumar [2], and was based on the Distributed Multiprocessor Priority Ceiling Protocol described by Rajkumar et al. [1], which in turn was based on the Priority Ceiling Protocol [9].

According to this protocol, for the case of tasks accessing local resources or when there is only one processor in the system, this reduces to the PCP.

3.1. Definition of the priority ceilings

We define as $Prio_H$ the highest priority among all priorities of all tasks of all processors. The ceiling of global resources is defined by a composition of two ceilings:

1. $Ceil_G$ such that $Ceil_G > Prio_H$.
2. $CeilR_{i,j}$ which is defined for each processor P_j as the highest priority among all priorities of tasks that access the resource R_i and are allocated to a processor other than P_j.

The ceiling will be $Prio_G + CeilR_{i,j}$, with the second component variable for each processor. When a task on processor P_j accesses resource R_i, it must do it with its priority adjusted to $Prio_G + CeilR_{i,j}$.

3.2. Resource acquisition rules

For global resources, when a task on a processor P_j accesses a resource R_i, it must do it with its priority adjusted to $Prio_G + CeilR_{i,j}$, i.e., in a selective non-preemptive mode. This ensures that the task will not be preempted by tasks running outside critical sections. A task accessing a critical section can only be preempted by tasks accessing critical sections with higher ceiling. When a task tries to access a global resource that is not available, it will then be inserted into a priority (its normal priority) ordered queue and then suspended allowing the execution of lower priority tasks. When a global resource is released, the task that is at the head of the queue (highest priority task in the queue, if any) will be resumed. The original MPCP is based on suspension.

In [6] it is presented a spin-based version of the this protocol. The spin-based version uses preemptive spin, i.e., allows the execution of higher priority tasks. Other characteristics remains the same.

4. Review of FMLP

The Flexible Multiprocessor Locking Protocol was proposed by Block et al. [3] to global and partitioned EDF scheduling and Pfair PD^2 [10]. The protocol extension for partitioned static priority is presented in [11]. The protocol is considered flexible for two reasons. The first is that it can be used with both partitioned and global scheduling. The second is that it is agnostic as to the type of execution control policy adopted (suspension or spin). This protocol is based on the definition of two types of global resources, short ones and long ones. The definition is based on access time to these resources (critical section length). Short resources are accessed via FIFO queue-based spinlocks, while long resources are accessed according to a semaphore based protocol (with FIFO queuing). The classification in short or long resources is left to the system designer.

As this paper takes into account only the version with partitioned static priority scheduling, future descriptions will be based on [11]. In MPCP, tasks holding global resources should have their priorities boosted to not be preempted by tasks running without resources. FMLP uses a simplified approach, which is to raise the priority of tasks holding resources to the same highest priority, i.e., the processor where the task executes becomes non-preemptive. Tasks with the highest priority (holding resources) are scheduled in FIFO order.

4.1. Resource acquisition rules

Given the protocol characteristics outlined above, the following rules are defined for resource acquisition according to FMLP, assuming no resource nesting:

- *Request for short resources:* a task τ_i executes a request for a global resource R_j. This task becomes non-preemptible. If the resource is available, it is assigned to τ_i, and if not, the task should be blocked in a spin-lock (busy-wait).

- *Request for long resources:* a task τ_i executes a request for a global resource R_j. If the resource is available, τ_i becomes non-preemptible (which is called priority boost in [11]) and enters the critical section, otherwise it will be blocked (suspended).

The fact that no priority adjustment is necessary for short resources is due to the busy waiting. The task receives a resource after a waiting time, so it does not need a high priority to resume as soon as possible, because it was actually running (busy-wait in a non-preemptive way).

For local resources, the protocol suggests the use of the SRP [12]. According to Brandenburg and Anderson [11], the classification of resources in short and long is only for global resources.

Recently, it was proposed a preemptible version of FMLP called FMLP+ [13,14]. In this version, a critical section can be preempted by another critical section (executed by other task) on the same CPU. In FMLP tasks are scheduled in FIFO order of unblocking as tiebreaker, in case of two or more tasks being ready to execute their own critical sections on the same CPU (always non-preemptible). In FMLP+, timestamps of resource requisition time are used as tiebreaker, i.e., higher priority for smaller timestamps (preemptions may occur).

5. Multiprocessor stack resource policy

The Multiprocessor Stack Resource Policy (MSRP) was proposed by Gai et al. [15]. It is based on the Stack Resource Policy (SRP) from Baker [12]. This protocol can be used with static priority, although it was initially designed for EDF. This protocol is very similar to FMLP short, except by stack reuse. Due to this similarity, we will not use this protocol in empirical comparisons.

6. Exploring the design space of protocols

In the previous sections we reviewed the protocols for multiprocessor systems with partitioned static priority scheduling. Table 1 summarizes the characteristics of each protocol. These characteristics represent the design space of protocols: queuing policy in case of blocking (order to access the resource); execution control policy, which may be suspension or spin; and whether the protocol allows preemption of critical sections or not. For example, MPCP allows preemption (only by tasks that will also execute critical sections) because each critical section can be executed with a different ceiling, while FMLP and MSRP execute critical sections in a fully non-preemptive way. Indeed, a protocol can be constructed by a selection of characteristics from the design space, and not all combinations are explored by the literature. It is also shown in this table the variations proposed in this paper. A preliminary version of some of these new protocols was presented in [7].

6.1. Proposed variation 1: MPCP non-preemptive

This first variation, called MPCP non-preemptive (MPCPNP), consists in executing critical sections in a non-preemptive way, similarly to MSRP and FMLP. For this protocol we also propose a spin-based version that is non preemptive. Since critical sections are not preemptive, treating blockings with a non-preemptive spin presents a more homogeneous solution. When non-preemptive spin is used at any given moment, there may be at most one blocked task on each processor, similarly to FMLP short.

6.1.1. Resource acquisition rules

For this protocol, we define the following rules for resource acquisition:

Suspension-based version: A task τ_i performs a request for a global resource R_j. If the resource is available, the task becomes non preemptible (the maximum priority is assigned to it) and enters the critical section, otherwise it will be blocked (suspended). The order of resource assignment to tasks is by nominal priority (the task with the highest priority accesses first). The task will execute again with its nominal priority when it releases the resource. When a task releases a resource, it is allocated the task that is at the head of the waiting queue, which is assigned the highest priority.

Spin-base version: The rules are the same as for the suspension-based version, however, when a task blocks it stays running in non-preemptive spin with maximum priority. The task becomes preemptible again when it finally releases the resource.

Table 1
Comparison between the protocols regarding the design space.

Protocol	Features				
	Execution control		Access order		Crit. sec. preempt
	Suspension	Spin	Prio. order	FIFO	
Existing protocols					
MPCP: [2]					
MPCP-Susp	×		×		×
MPCP-Spin [6]		×	×		×
MSRP: [16]		×		×	
FMLP: [3]					
FMLP Short		×		×	
FMLP Long	×			×	
FMLP+ [13]	×			×	×
Proposed in this paper					
MPCPNP:					
MPCPNP-Susp	×		×		
MPCPNP-Spin		×	×		
MPCPF:					
MPCPF-Susp	×			×	×
MPCPF-Spin		×		×	×

The motivation behind this variation is the observation that this only effects, in the worst case, remote blocks. This is due to the fact that, each lower priority task, regardless the ceiling of the requested resource, could potentially block higher priority tasks on the same processor. A normal execution segment of a higher priority task can always be preempted by any other critical section of a lower priority task, no matter the ceiling of the resource. In this case, executing critical sections in a non-preemptive way can be seen as executing all sections with the same priority ceiling.

From the point of view of a specific processor, in the worst case, the real difference between the MPCP and MPCPNP (when both are suspension based) is the instant of preemption. With MPCP it can occur within a critical section (the critical section of a recently unblocked task has a ceiling higher than the critical section of the running task) or at the exit of this critical section (the critical section of the recently unblocked task has a ceiling lower or equal). In MPCPNP this preemption can only occur at the exit of a critical section (same effect as equal ceilings in MPCP). One way to transform MPCP in MPCPNP is to configure the ceiling of all resources with the same value.

For the case of remote blocks, in the worst-case, the waiting time for a resource will be increased, because now the order of execution of different critical sections on the same processor will be FIFO (mutually non preemptive tasks). Instead of a task having to wait for the execution of the critical section itself added to the critical sections with higher ceilings, it will have to wait (worst-case) all critical sections on the processor that blocks the task.

This variation can be seen in Table 1 where it is shown that it is not equivalent to any other protocol outlined above. The difference between this variation and FMLP/MSRP is the queuing policy and between it and MPCP, is that the first is non preemptive.

6.1.2. Blocking factors

For MPCPNP, there are several possible situations of blocking.

Local blocking by lower priority tasks: All protocols present some kind of local blocking caused by lower priority tasks. This type of blocking is not related to local resources, but to the fact that lower priority tasks can access global resources. In MPCP non preemptive, this blocking is manifested differently in the different versions (suspension or spin).
In the suspension-based version, a task (τ_i) may suffer blocking caused by lower priority tasks accessing critical sections. These tasks whose priorities are lower, may have been blocked in two distinct situations: before activating the task at hand, or when executed while this task was blocked, waiting for some other resource. When a task is unblocked in a critical section, it can always preempt higher priority tasks that are running outside a critical section.
In the spin-based version (non-preemptive spin), when a higher priority task (τ_i for example) starts executing, it can be blocked (manifested in the form of release jitter) by a lower priority task that is running inside a critical section (in a non-preemptive way) or waiting on spin for some critical section. In the worst case, the task at hand (τ_i) activates immediately after a lower priority task starts the spin (the task with the largest sum of spin time with critical section time). That is, in the worst case, this blocking will be restricted to only one lower priority task, because once it ends the access to its critical section, the task at hand (τ_i) immediately begins its execution. If the lower priority tasks don't access resources, this blocking will not exist for tasks with higher priorities.
Remote blocking: Whenever a task requests a resource, it will be possibly blocked. This blocking occurs because, in the worst case, all tasks want to access that resource at a given time. This blocking is influenced by the queuing policy, which is by priorities. This blocking is also directly influenced by non preemptive execution, as will be shown later in the schedulability analysis. If a task does not access resources, it will not suffer remote blocking.
Back-to-back execution: A task may suffer additional interference due to higher priority tasks that suspend themselves (resource not available) to resume execution at a later instant (when the resource is available). Normal response time tests only capture interference of tasks that do not self suspend. This type of blocking occurs only with the suspension-based version. If a higher priority task does not access resources, this factor does not exist with regard to this task.

In summary, remote blockings are a consequence of resources accessed by the task itself, while local blocking and back-to-back execution are the result of resources accessed by lower and higher priority tasks, respectively.

6.1.3. Schedulability analysis

Using the approach of schedulability analysis of the Section 2.2, we can calculate the response time of the tasks using the MPCPNP protocol. First, we must define the parameters that are protocol dependent.

To calculate the worst-case response time of a particular critical section, we have two approaches, one for the suspension-based version and another for the spin-based version:

Suspension-based: The worst-case response time of a critical section $C'_{i,k}$ using the suspension-based version is given by:

$$W'_{i,k} = C'_{i,k} + \sum_{\tau_u \in P(\tau_i)} \max_{1 \leqslant v \leqslant s(u)} C'_{u,v}. \tag{9}$$

As the protocol is non-preemptive, this is equivalent to saying that all critical sections are executed with the same level of priority (highest priority in this case). The worst-case of a critical section of a task occurs when all other tasks on the same processor are also able to execute their largest critical sections (related to others resources), with the task at hand being the last to be unblocked (all tasks with the same elevated priority level implies FIFO order of access to the processor).

Spin-based: For the spin-based version, the worst-case response time of a critical section $C'_{i,k}$ must be calculated in a different way:

$$W'_{i,k} = C'_{i,k}. \tag{10}$$

In Eq. (10), the worst-case of a critical section is the execution time of this section, identically to FMLP short. This occurs because the spin-based version uses non-preemptive spin as the mechanism for execution control. Thus, only one task can be blocked on some resource at a given time on a processor, and when unblocked in its critical section, the task does not need to compete with any other for processor utilization.

For the calculation of the blocking time ($B^r_{i,j}$) of each critical section $C'_{i,j}$, the following convergence must be used for both versions (suspension and spin-based), with initial value $B^{r,0}_{i,j} = \max_{l>i \& (\tau'_{l,u}) \in R(\tau'_{i,j})}(W'_{l,u})$, being $R(\tau'_{i,j})$ the set of critical segments related to the same resource used by the segment $\tau'_{i,j}$:

$$B^{r,n+1}_{i,j} = \max_{l>i \& (\tau'_{l,u}) \in R(\tau'_{i,j})}(W'_{l,u}) + \sum_{h<i \& (\tau'_{h,v} \in R(\tau'_{i,j}))} \left(\left\lceil \frac{B^{r,n}_{i,j}}{T_h} \right\rceil + 1 \right)(W'_{h,v}). \tag{11}$$

In Eq. (11), in the worst-case, as in MPCP, task τ_i can be blocked by only one lower priority task τ_l (first term in Eq. (11)), which has the largest critical section related to the resource $R(\tau'_{i,j})$ (among all lower priority tasks), and was already executing inside its critical section when τ_i made an attempt to access this resource. However, τ_i can be blocked several times by each higher priority task that access the resource in question (second term of the equation). This occurs because the resource access queues are ordered by priorities. It is noteworthy that, in the previous equation, one must use values of $W'_{i,k}$ appropriate for each protocol version. For the calculation of the response times, we must use a suitable approach for each protocol version:

Suspension-based: For the calculation of the response time using the suspension, one must use Eq. (3) for interference calculation and Eq. (5) for local blocking calculation, using the blocking times previously shown (convergence of equation 11 for remote blocking calculation, considering suspension). The final equation is:

$$W_i^{n+1} = C_i + B_i^r + \sum_{h<i \& \tau_h \in P(\tau_i)} \left\lceil \frac{W_i^n + B_h^r}{T_h} \right\rceil C_h + s(i) \sum_{l>i \& \tau_l \in P(\tau_i)} \max_{1 \leqslant k < s(l)} C'_{l,k}. \quad (12)$$

Spin-based: For the calculation of the response time using the spin-based version, one must use Eq. (4) for interference calculation and Eq. (7) for local blocking calculation for non-preemptive spin-based protocols, using also the blocking times previously shown (convergence of Eq. (11), considering spin). The final equation results in:

$$W_i^{n+1} = C_i + B_i^r + \sum_{h<i \& \tau_h \in P(\tau_i)} \left\lceil \frac{W_i^n}{T_h} \right\rceil (C_h + B_h^r) + \max_{l>i \& \tau_l \in P(\tau_i) \& 1 \leqslant k < s(l)} (C'_{l,k} + B^r_{l,k}). \quad (13)$$

6.2. Proposed variation 2: MPCP with FIFO queuing

The second variation (MPCPF) presented in this paper changes the order in which blocked tasks access a particular resource. This variation is closer to the original MPCP than the previous one. In original MPCP, tasks access resources in priority order. A task blocked on some resource may wait for several higher priority tasks and only one lower priority task, and each higher priority task can use the critical section more than once before the task in question accesses. In this second variation, blocked tasks access resources in FIFO order. This does not mean that this variation is equal to FMLP for fixed priority, because in the latter, tasks access resources in FIFO order and in a non preemptive way. This variation is also different from FMLP + because this uses a different tie-breaking rule. In FMLP + tie-breaking between the active critical sections in a processor (unblocked tasks in their critical sections) occurs with the use of *timestamps* of the resource requisition instant (the first to request is the first to run). In MPCPF the tie-breaking is made in the same way as in MPCP, i.e., with the use of the *ceiling* associated with the critical section. The comparison of this variation with other protocols can also be seen in Table 1.

For MPCPF, we also propose suspension-based and spin-based versions. For the spin-based version, we adopted the preemptive mechanism. With non preemptive spin this variation would be identical to FMLP short, presenting the same blocking times. If the protocol uses non-preemptive spin, then it will be possible to have only one blocked task on each processor at each instant. So, an unblocked task would never suffer preemption by another unblocked task, exactly as in FMLP. The protocol has the same ceiling definition of the original MPCP, i.e., $Prio_G + CeilR_{i,j}$.

6.2.1. Resource acquisition rules

The following rules must be used to do the resource acquisition:

Suspension-based version: A task τ_i performs a request for a global resource R_j. If the resource is available, then τ_i takes possession of it and has its priority changed to the resource ceiling, otherwise the task will be blocked (suspended). The order of resource assignment to tasks is FIFO. The task will execute again with its nominal priority when it releases the resource. When a task releases a resource, it must release the task that is at the head of the resource access queue, if any, and adjust the priority of this task to the ceiling of the resource.

Spin-based version: The spin-based version follows the same resource acquisition rules of the suspension-based version. The only difference is that when a task blocks, it remains in preemptive spin, allowing the execution of higher priority tasks. The task priority will also be adjusted to the resource ceiling, when the resource is acquired. The task will also execute again with its nominal priority when it releases the resource.

6.2.2. Blocking factors

This protocol has the same blocking factors of MPCPNP, but the way they are manifested is different:

Local blocking by lower priority tasks: For the case of the suspension-based version, this blocking will be equal to the MPCP and MPCPNP, because it involves auto-suspension. For the spin-based version (preemptive spin), a task may suffer blocking from lower priority tasks that have been blocked before its activation (equal to MPCP).

Remote blocking: Remote blocking may occur whenever a task tries to access a resource which is not available. Since the queuing is FIFO, in the worst case, the task at hand will be at the end of the resource access queue, being all other tasks ready to use that resource. Unlike the MPCP, a task may be blocked only

once by each task that also uses the same resource, independent of priorities.

Back-to-back execution: A task may suffer additional interference from higher priority tasks that suspend themselves (resource not available), to resume the execution at a later time, when the resource becomes available. This blocking factor only occurs with the suspension-based version.

As in MPCPNP, remote blockings are a consequence of resources accessed by the task itself, while local blocking and back-to-back execution are the result of resources accessed by lower and higher priority tasks, respectively.

6.2.3. Schedulability analysis

Using the schedulability analysis approach previously shown, for this protocol, we can calculate the response times of the tasks. First, we must define the equation parameters that are protocol dependent.

The first parameter to be defined is the worst-case response time of a critical section $C'_{i,k}$, which is $W'_{i,k}$. For both versions (suspension and spin), we use the same equation, where $gc(i,k)$ is the ceiling of the kth critical section of the task τ_i:

$$W'_{i,k} = C'_{i,k} + \sum_{\tau_u \in P(\tau_i)} \max_{1 \leqslant v \leqslant s(u) \& gc(u,v) > gc(i,k)} C'_{u,v}. \quad (14)$$

In the worst case (Eq. (14)), when a task is ready to execute a critical section (the access to a resource was granted), it will wait for the largest critical section of each task whose resource has ceiling equal or higher than the resource associated with his own critical section.

Finally, $B^r_{i,j}$ represents the remote blocking time suffered by τ_i when requesting the global critical section $C'_{i,j}$. As the queuing is in FIFO order, in the worst case, a task will have to wait for all other tasks that access the resource (sum of the worst-case response times of critical sections). $B^r_{i,j}$ is given by:

$$B^r_{i,j} = \sum_{h \neq i \& (\tau'_{h,v} \in R(\tau_{i,j}))} W'_{h,v}. \quad (15)$$

Eq. (15) should be used for both suspension and spin-based versions. To calculate the total response time of the tasks, we must use the appropriate approach for each protocol version:

Suspension-based: In the same way as in the previous protocol, to calculate the response times using the suspension-based version, we should use the interference equation (Eq. (3)) and local blocking Eq. (5). Eq. (15) must be used to calculate all remote blockings. The final equation is:

$$W_i^{n+1} = C_i + B^r_i + \sum_{h<i \& \tau_h \in P(\tau_i)} \left\lceil \frac{W_i^n + B^r_h}{T_h} \right\rceil C_h + s(i) \times \sum_{l>i \& \tau_l \in P(\tau_i)} \max_{1 \leqslant k < s(l)} C'_{l,k}. \quad (16)$$

Spin-based: To calculate the response time when using the spin-based version, we can use the interference equation (Eq. (4)), and the local blocking equation for protocols based on preemptive spin (Eq. (6)). The blocking time is obtained from Eq. (15). The resulting equation is:

$$W_i^{n+1} = C_i + B^r_i + \sum_{h<i \& \tau_h \in P(\tau_i)} \left\lceil \frac{W_i^n}{T_h} \right\rceil (C_h + B^r_h) + \sum_{l>i \& \tau_l \in P(\tau_i)} \max_{1 \leqslant k < s(l)} C'_{l,k}. \quad (17)$$

7. Empirical comparison of schedulability

This section aims to compare the proposed solutions with the existent protocols, which are MPCP and FMLP. For comparison of the many protocols, the equations of schedulability analysis were all implemented in a single framework developed using the C language. In the experiments, the number of processors needed to schedule a task set using each protocol are compared. The number of processors necessary to schedule a task set using a specific protocol is determined by a bin packing (allocation) algorithm. In this context, a bin packing algorithm must use the equations provided for the schedulability analysis of each protocol. The task sets are generated randomly with specific parameters. Thus, the same set of tasks will be submitted for allocation with each protocol. For the sake of naming the following convention is used throughout the empirical experiments:

- PLAIN: denotes the simple analysis of response time, ignoring locks, will be used only as a baseline.
- MPCP-SUSP: represents the suspension-based of classical MPCP.
- MPCP-SPIN: represents the spin-based version of classical MPCP.
- MPCPNP-SUSP: represents the suspension-based version of MPCP non-preemptive.
- MPCPNP-SPIN: represents the spin-based version of MPCP non-preemptive.
- MPCPF-SUSP: represents the suspension-based version of MPCP with FIFO queuing.
- MPCPF-SPIN: represents the spin-based version of MPCP with FIFO queuing.
- FMLP-LONG: long version of FMLP (based on suspension).
- FMLP-SHORT: short version of FMLP (based on spin).

In order to proceed with the experiments, we need to define an allocation algorithm and present a strategy of task set generation.

7.1. Allocation algorithm

The allocation of tasks in multiprocessor systems with partitioned scheduling is equivalent to the bin-packing problem, which is NP-hard in the strong sense. This problem must be addressed with sub-optimal heuristics, even those are subject to scheduling anomalies. In the context of this work, to perform the allocation of tasks to processors, we used a variation of the algorithm RM-FFDU (Rate-Monotonic First-Fit Decreasing Utilizations) described by Oh et al. [5]. The fit function (check if the inclusion of a task keeps the system schedulable) uses the schedulability tests from the previous sections.

In its simplest form, the first-fit algorithm starts from a set of unallocated tasks and a set of processors. This does not apply to tasks with dependencies (such as mutual exclusion), because if the algorithm decides to put tasks "A" and "B" on the same processor at first (the system is schedulable), this may cause the system to never reach a schedulable partitioning for the whole task set. There may be a task "C" that regardless of on which processor it is placed, it makes "B" to miss its deadline for sharing resources with it (induction of blocking). For the previous example, it is necessary that "A" and "B" are not placed on the same processor, and this is only possible if the partitioning algorithm has a global view

of the system. Briefly, when tasks are not independent, any reasonable allocation of a previous task may influence the allocation of future tasks, and more, any decision that does not consider the overall system can be optimistic, i.e., a feasible allocation of part of the system can make the whole system not schedulable. This also applies to other allocation algorithms, such as BF (Best fit) and WF (Worst fit) for example. This problem was also pointed out by Nemati et al. [17]. In order to work around this problem, the algorithm used starts with the worst possible case, which is a task per processor (since that one task per processor is a schedulable partitioning) with tasks/processors sorted in descending order of utilization. From this point, the algorithm tries to move the task of the processor 1 to processor 0, then the task of the processor 2 to processor 0 or 1 (via First fit) and so on. After this step, the algorithm must remove the empty processors. The input of the algorithm is a set of tasks and its output is a superset of task sets allocated to each processor.

7.2. Generation of task sets

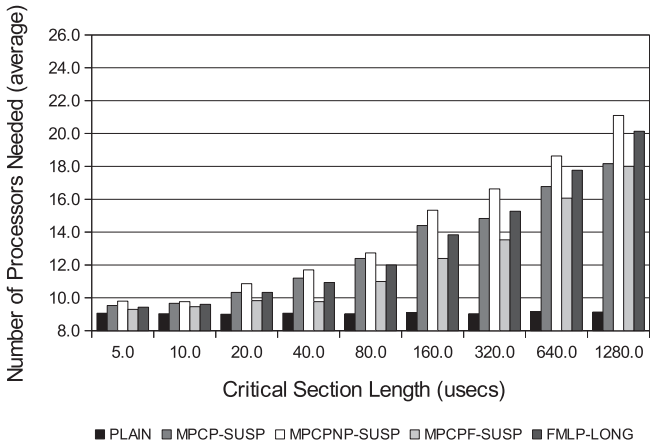
The experiments were performed disregarding the effects of overheads. The sets of periodic tasks with implicit deadlines were generated according to the article [6]. According to that article, the task sets, with utilization greater than one (for multiprocessors) are always generated as composition of subsets with utilization equal to one. This means that, if the utilization of the task set is 8, then, internally, the algorithm generates eight subsets with utilization equal to one, which are combined at the end of the process. The UUniFast proposed by Bini and Buttazzo [18] was utilized for generation of utilizations (U_i). Task periods (T_i) were generated in the range [10 ms, 100 ms], uniformly. Computation times were calculated based on the data previously calculated, i.e., $C_i = U_i \times T_i$.

7.3. Experiments

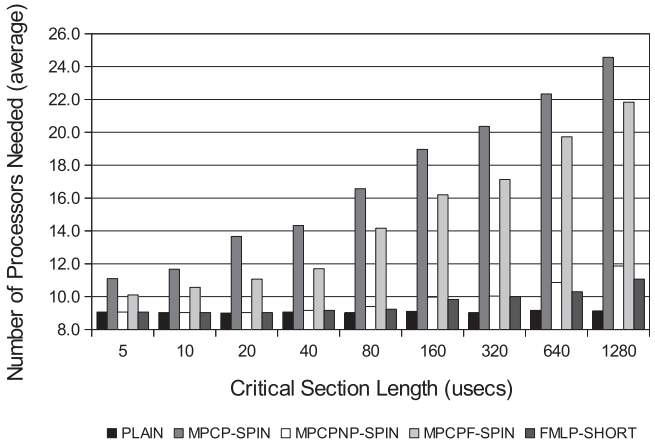
For each experiment, 30 task sets for each configuration were submitted to allocation using each protocol. Each subconfiguration of each experiment is presented on the x axis of the graphs. As the objective of this paper is not to compare execution control policies, the results are shown separately for suspension and spin. Indeed, such comparisons are presented in works like Brandenburg et al. [19] and Lakshmanan et al. [6]. The results of the experiments are the number of processors needed to schedule a system, this is a parameter that indicates the efficiency of scheduling using each protocol.

7.3.1. Experiment 1: variation of the length of critical sections

The first experiment aims to evaluate the behavior of the protocols for different critical section sizes. In the task sets generated, each task accesses two global critical sections, and each resource is shared by two tasks. The sections varied in the range [5] μs, doubling the length at each test. All task sets comprise 40 tasks, with total utilization equal to 8. The results of this experiment are shown in Fig. 1(a).

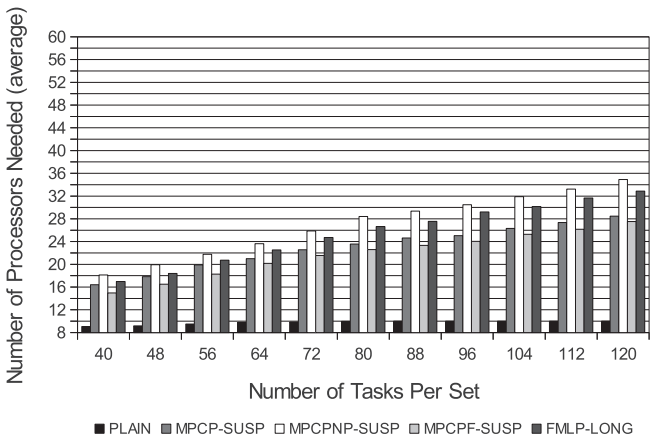


(a) Using Suspension

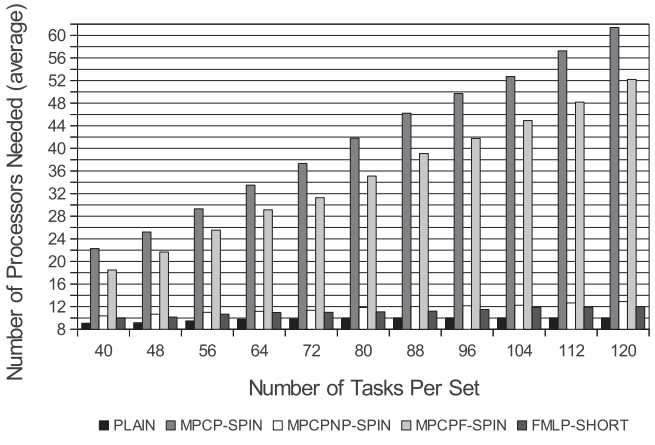


(b) Using Spin

Fig. 1. Variation of the size of critical sections.

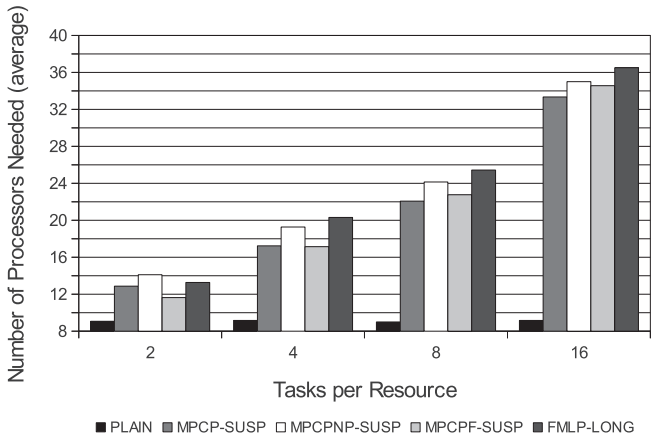


(a) Using Suspension

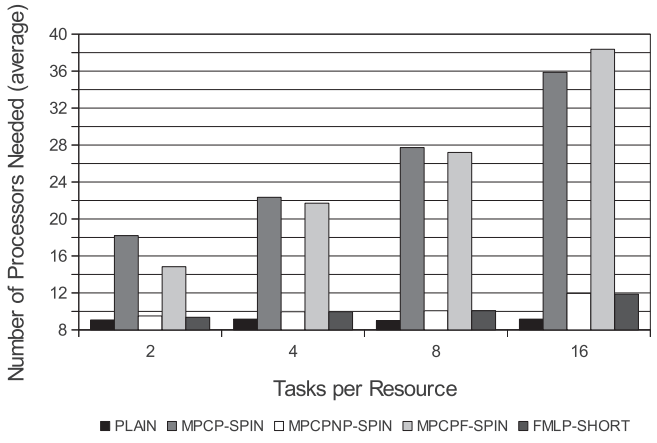


(b) Using Spin

Fig. 2. Variation of the number of tasks per set.



(a) Using Suspension



(b) Using Spin

Fig. 3. Variation of the number of users per resource.

For this experiment, the results were quite different for spin-based and suspension-based versions. For the suspension-based versions (Fig. 1(a)), the best results were obtained for MPCPF-SUSP. In second place, are FMLP-LONG, which is better in the range [5 ms, 160 ms] and MPCP-SUSP in the interval [320 ms, 1280 ms]. In the case of spin-based versions (1)(b), the best results were obtained to FMLP-SPIN (best) and MPCPNP-SPIN (second best).

7.3.2. Experiment 2: variation of the number of tasks

The second experiment aims to evaluate the behavior when varying the number of tasks in each generated set. Initially it starts with 40 tasks and increases the number of tasks successively until 120. The utilization of the task sets were fixed to 8 and the length of the critical sections were fixed in 500 μs. For this experiment, it was also used 2 tasks accessing each resource. The results are shown in Fig. 2.

This experiment showed similar results compared to the previous one. For the suspension-based version (Fig. 2)(a), the best results were also obtained by MPCPF-SUSP (best) and MPCP-SUSP (second best). In relation to the spin-based versions, the best results were also obtained by FMLP-SPIN and MPCPNP-SPIN (second best).

7.3.3. Experiment 3: variation of the number of lockers per resource

In the third experiment, the parameter that is varied is the number of tasks/lockers per resource, keeping two critical sections (all with 100 μs) per task, and total utilization equal to 8. The number of lockers varies from 2 to 16 for each resource, always multiplying by two at each test. The results of this experiment are shown in Fig. 3.

Table 2
Subjective analysis of results.

Exp.	Rank			
	1°	2°	3°	4°
Suspension-based protocols				
1	MPCPF	MPCP/FMLP	MPCPNP	
2	MPCPF	MPCP	FMLP	MPCPNP
3	MPCPF/MPCP	MPCPNP/FMLP		
Spin-based protocols				
1	FMLP	MPCPNP	MPCPF	MPCP
2	FMLP	MPCPNP	MPCPF	MPCP
3	FMLP	MPCPNP	MPCPF/MPCP	

In this experiment, considering the case of the suspension-based versions (in Fig. 3(a)), there are two extremes: with 2 users per resource, the best results were obtained by the MPCPF-SUSP, whereas the second best was obtained by MPCP-SUSP. At the other extreme, with 16 lockers per resource, the best results was by MPCP-SUSP and the second best by MPCPF-SUSP. This experiment shows that FIFO queuing in suspension-based protocols tends to degrade the results as the number of lockers increases. For spin-based protocols (in Fig. 3(b)), the best results were obtained by FMLP-SHORT and MPCPNP-SPIN. FIFO queuing does not affect FMLP-SHORT, because the blocking is limited by the number of processors, and not by the number of lockers, due to the non preemptive spin.

7.3.4. Comments

The results of the experiments are summarized in Table 2. In this table, the protocols were ordered in decreasing order of performance obtained in experiments. One can see that, in general, the MPCPF-SUSP showed the best results in relation to suspension based protocols. The MPCPNP had the second best results for the spin-based protocols, losing only to FMLP. There were also cases of tie, where a protocol was better up to a certain range of configurations, and other protocol was better after this range.

Overall, the results were different from those shown in [6], due to the following factors: same parameters, but with UUniFast for utilizations. In [6] it was used a variation of BFD (Best fit Decreasing) for allocation and we used a RM-FFDU variant. And finally, we evaluated more protocols than in the cited article.

Evaluating protocols based on allocation, one can see that there may be significant differences among all protocols, at least in the test scenarios used.

8. Implementation on PREEMPT-RT Linux

We implemented the MPCPNP protocol in the Linux PREEMPT-RT [20] kernel, version 3.0.18-rt34+. The PREEMPT-RT is a patch for the Linux kernel, developed primarily by Ingo Molnar. Its main objective is to allow deterministic execution of real-time tasks in the Linux kernel. It generates a kernel with low latencies and preemptive in kernel mode (executing system calls and kernel threads), except for interrupt handling code and some segments such as scheduling code, for example.

Although there are academic versions of real-time Linux, like LITMUS^{RT} [21], this work was based on PREEMPT-RT due to its fully preemptive architecture and to the fact that this system is already used in production applications. The proposed implementation was made primarily for use in device-drivers dedicated to real-time applications (in kernel space). The proposed implementation

does not replace the current implementation, being only an alternative for certain applications that require more determinism in terms of blocking times. It also serves as a proof of concept.

Nowadays, the PREEMPT-RT kernel uses priority inheritance for limiting blocking times in critical sections. However, this protocol has been developed for uniprocessor systems, being used as a best-effort mechanism in the case of PREEMPT-RT (that supports multiprocessor systems) to reduce uncontrolled priority inversions. In the case of critical real-time applications on multiprocessor systems which access hardware through device-drivers, it is necessary to know the blocking times. When these times are known, it is possible to guarantee the schedulability of the system. This is only possible with the use of proper protocols for synchronization on multiprocessor systems, implemented in operating systems or real-time monitors. Appropriate protocols for multiprocessor systems with partitioned scheduling and fixed priority are the Multiprocessor Priority Ceiling and the Flexible Multiprocessor Locking Protocol, as well as the protocol presented in the previous section.

The MPCP non preemptive (MPCPNP) is a self-contained protocol, in the sense of not requiring offline pre-configuration, which is required with MPCP. MPCP requires all resources to have their ceilings configured from a preliminary analysis of all tasks that access them. Another issue is that MPCPNP can be implemented without changes to the Linux operating system scheduler. The original MPCP needs an implementation of additional priority levels, which include more scheduling queues per processor. The MPCPNP protocol was implemented inside the operating system kernel, so that it is visible to drivers and loadable modules which are pieces of kernel that can be dynamically loaded and unloaded.

The protocol implementation provides a data type *struct mpcpnp_mutex*, a family of functions and a macro:

- DEFINE_MPCPNP_MUTEX (mutexname, type): defines a mutex statically, whose identifier is mutexname. Type must be used to select the execution control policy, i.e., suspension or spin.
- mpcpnp_mutex_lock (struct mpcpnp_mutex *lock): performs the acquisition of *mutex* whose identifier is named "lock". This function possibly blocks the calling task if the *mutex* is held by a task on another processor.
- mpcpnp_mutex_unlock (struct mpcpnp_mutex *lock): performs the release of *mutex* whose identifier is "lock". This function should wake up the next task to use the resource, if any.

For the sake of comparison, FMLP was also implemented. FMLP is also a self-contained protocol that does not need changes in the scheduler. The FMLP implementation made in LITMUS^{RT} by Brandenburg and Anderson [11] was not used because it can not be used directly in Linux PREEMPT-RT. A new implementation of FMLP was necessary. This implementation also provides a data type and similar functions, except for the field 'type', which should be LONG or SHORT.

The implementations proposed for both MPCPNP and FMLP have no fastpath, unlike the rt_mutexes with priority inheritance in the PREEMPT-RT kernel. Fastpaths are useful for performance reasons, but in the case of this paper, the gain would be constant at the cost of an implementation that make intrusive changes in the scheduler. These intrusive changes mean making postponed adjustments to the priority of tasks that acquire resources via fastpath, and the points where it should be done are spread across several locations of the scheduling code. Fastpaths only show gains in specific situations, such as when the mutex is completely available for acquisition and/or the mutex to be released has no task in its waiting queue. Anyway, a fastpath does not affect the worst case, only optimizes the common case.

9. Evaluation of overheads

Most experiments involving overhead comparison of synchronization protocols use schedulability tests inflated with specific measurements (event times such as context switching, for example) applied to task sets synthetically generated. In this paper, we aim to make the comparison of overheads based on the measurement of end-to-end activations of tasks that share resources. Overhead is considered to be the execution time of the protocol (duration of the resource acquisition/release primitive itself). This does not include delays of lower priority tasks (extra interference).

To evaluate the overhead of the MPCPNP implementation proposed, it was developed a test scenario where some tasks access a device-driver. This device-driver has a critical section (with constant length) protected with the implemented protocols (MPCPNP and FMLP long). We used a machine with 4 cores (Intel Xeon 5130 with 2.00 GHz of frequency), and each core runs a task that accesses the device-driver. These tasks were configured with the same release time and equal periods, so that all of them will always try to use the resource at the same time or at very close instants. It is used here the term "close" due to the fact that time sources of processors are not perfectly synchronized (Local APIC timer, in the used machine).

For the test scenario, both FMLP and MPCPNP will have the same total blocking time (sum of the blocking times of all tasks for a specific activation). When all tasks are identical (same period, with same resource and computation time), regardless of the protocol used, the sum of the blocking time is equal, and the only factor that can change is the order of tasks accessing resources.

It is shown in Fig. 4 two examples of identical four tasks, one for each processor. These tasks attempt to use a particular resource at the same time using two arbitrary protocols. For the first protocol (shown in Fig. 4(a)), the access order was τ_0, τ_2, τ_3 and finally τ_1. For this arbitrary protocol, the sum of the blocking times was

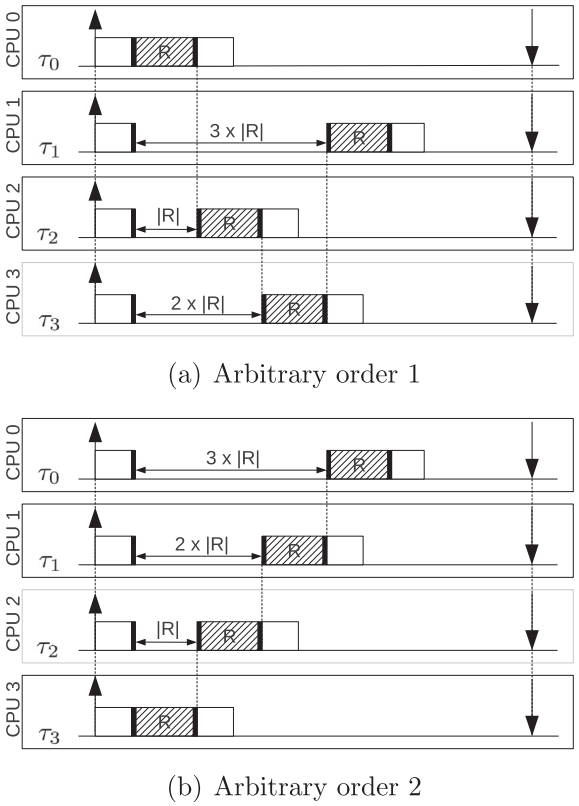


Fig. 4. Examples of access to a particular resource.

Table 3
Configuration of tasks used in experiment.

Task	Period (ms)	Priority	Processor	Activations
τ_0	40	60	1	10,000
τ_1	40	61	3	10,000
τ_2	40	62	0	10,000
τ_3	40	63	2	10,000

$6 \times |R|$, where $|R|$ is the length of the critical section related to resource R. For the second protocol (shown in Fig. 4(b)), it was produced the following access order: τ_3, τ_2, τ_1 and finally τ_0, but keeping the same sum of blocking times, which is $6 \times |R|$.

In real systems, one should not disregard the execution time of the lock and unlock primitives necessary on the entry and exit of critical sections, respectively. This time is shown as a small dark region in Fig. 4(a) and (b) before and after the critical sections.

For a given activation of the 4 tasks, we denote δ as the sum of the times of the lock and unlock primitives in this activation, disregarding blocking times. This idea is generalized in Eq. (18).

$$sum\_times = \sum_{0 \leqslant i < 4} (lock\_time(i) + unlock\_time(i)) \approx 6 \times |R| + \delta. \quad (18)$$

In Eq. (18), the sum of the lock/unlock times approximates the sum of 6 critical sections added of δ. If a protocol have higher overhead than other, then its δ will also be higher than the δ of this other. Thus, one can compare protocols according to δ. In other words, in the proposed scenario, the larger the δ, the longer the lock and/or unlock primitives.

9.1. Conditions of the experiments

For the experiments, the four tasks have been configured as shown in Table 3.

It is shown in Table 3 that all tasks have the same period, and are scheduled with the scheduling policy SCHED_FIFO. This is a policy for real-time POSIX compliant systems, assuming static priorities from 0 to 99 (the higher the number, the higher the priority). Thus, the activations and the access to the resource can be synchronized (synchronization of periods). For each activation (synchronized activations) the lock times and unlock times of the four tasks were summed. The experiment was performed with critical sections (internal to the device-driver) with lengths of 10 μs, 20 μs, 40 μs, 80 μs and 160 μs. For each critical section length, the experiment was repeated for each protocol. Times were measured with the tsc register (time-stamp counter) due to its low overhead of reading and accuracy. This register is incremented every processor clock cycle monotonically, thus the results are also presented in cycles of clock. Although the register tsc cannot be synchronized between processors on some microarchitectures, when used locally at each processor, no time discrepancy can occur (negative times). In the machine used, a cycle of clock corresponds approximately to 0.5 ns.

Table 5
Comparison of samples obtained in each experiment configuration through the Mann–Whitney–Wilcoxon statistical test (suspension-based protocols).

10 μs (length of the critical section)	
Protocol	FMLP-LONG
MPCPNP	>
20 μs (length of the critical section)	
Protocol	FMLP-LONG
MPCPNP	=
40 μs (length of the critical section)	
Protocol	FMLP-LONG
MPCPNP	>
80 μs (length of the critical section)	
Protocol	FMLP-LONG
MPCPNP	<
160 μs (length of the critical section)	
Protocol	FMLP-LONG
MPCPNP	<

Table 6
Comparison of samples obtained in each experiment configuration through the Mann–Whitney–Wilcoxon statistical test (spin-based protocols).

10 μs (length of the critical section)	
Protocol	FMLP-SHORT
MPCPNP-SPIN	>
20 μs (length of the critical section)	
Protocol	FMLP-SHORT
MPCPNP-SPIN	>
40 μs (length of the critical section)	
Protocol	FMLP-SHORT
MPCPNP-SPIN	>
80 μs (length of the critical section)	
Protocol	FMLP-SHORT
MPCPNP-SPIN	<
160 μs (length of the critical section)	
Protocol	FMLP-SHORT
MPCPNP-SPIN	<

The results (mean and standard deviation) of the measured data (corresponding to variable *sum_times*) are presented in Table 4. These results have been filtered to remove outliers, for reasons which will be discussed later. As stated earlier, these measurements refer to times which are described by Eq. (18).

9.2. Data analysis

For data analysis, one should compare statistically the samples obtained in order to check which protocol has more or less overhead. Since the measured data do not follow a normal distribution of probabilities, one must use non-parametric statistical test for comparison. In the context of this paper, it was used the Mann–Whitney–Wilcoxon test [22] to compare the distributions of the measured data.

Table 4
Basic statistical data of the experiment (mean ± standard deviation) with 10,000 samples for each configuration and results in cycles of clock.

Prot.	C.S. Len.				
	10 μs	20 μs	40 μs	80 μs	160 μs
MPCPNP-SUSP	240038 ± 40537	358564 ± 20504	597789 ± 41903	1069387 ± 23289	2025579 ± 23282
FMLP-LONG	235548 ± 40023	358784 ± 20694	595497 ± 49440	1074668 ± 32243	2048230 ± 29625
MPCPNP-SPIN	174380 ± 17162	295085 ± 13494	585654 ± 32127	1008573 ± 15230	1969719 ± 14804
FMLP-SHORT	172884 ± 14457	293065 ± 14919	533552 ± 84209	1011168 ± 13957	1967608 ± 17006

Although the chosen statistical test presents robustness against outliers in the samples, the measured data were previously filtered. In this work, outliers occur when there is interruption of the code that is under measurement. For example, an interrupt occurs when it is executing a lock primitive but that can not be handled because it runs with interrupts disabled. However, when the primitive enables interrupts again, the interrupt is finally handled. This interrupt is noticed in the measurement because of the calls to the lock and unlock primitives are surrounded by measurement code. The identification of these outliers generated by interrupt can be easily accomplished with the use of Linux tools, such as *ftrace* [23] for example. These outliers should be removed from the analysis because they show independent behavior which is not related to the implementation of the protocols.

Table 5 shows the comparison of samples using the statistical test mentioned and a significance level $\alpha = 0.05$ using the suspension-based versions. In the table, the relational operators show when it was obtained more or less overhead for a given configuration.

According to Table 5,, FMLP-LONG has lower overhead for critical sections of 10 μs and 40 μs. For critical sections of 20 μs, the overhead is equivalent. For sections of 80 μs and 160 μs, MPCPNP-SUSP presented lower overhead. Although this difference exists in the statistical distribution, one can see through the means (Table 4) that it is very small and they can be considered equivalent overheads in practice. Due to the fact that these protocols use suspension as control execution policy, they suffer direct influence of the schedule. Both resumption and suspension of tasks are functions assigned to the scheduler, being out of control of both experiment and protocol implementations.

It is also presented in Fig. 4 the averages and standard deviations obtained in the experiment, considering spin. It is presented in Figure 6 the statistical comparison of spin-based protocols. In Table 6, FMLP-SHORT showed lower overhead for critical sections of 10, 20 and 40 μs. For sections of 80 μs and 160 μs the lowest overhead was due to MPCPNP-SPIN. It is also worth observing that, statistically, there are differences in the distributions, but in practice, the values are very close. For this experiment, the results were better behaved than for suspension-based protocols. Protocols based on spin, unlike suspension-based, are not influenced by the scheduler implementation.

Generally, even with overheads statistically equal in only one configuration for the suspension-based version, it can be noted similarities when observed the means and standard deviations. Empirically it can be seen that the proposed protocol is equally suitable for implementation in terms of overhead such as FMLP. It is possible that the overheads of the protocols are actually identical, and the differences presented are a consequence of noise of measurement/hardware and the operating system behavior, which itself is often beyond the control of the experiment.

10. Conclusions

In this paper it was presented a review of existing protocols MPCP and FMLP. It was also proposed two variations of the MPCP that use unexplored combinations of characteristics within the design space of protocols, together with the corresponding schedulability analysis. In some sense, both proposed variations can also be seen as variations of FMLP, since they include features common to both protocols. The goal of the proposed variations is to facilitate analysis and implementation.

In order to evaluate the new protocols, it was used test scenarios with utilizations generated via UUniFast-discard, whose distribution is proved to be unbiased. Task set generation parameters were mostly the same ones described in [6]. The paper presented empirical comparisons between the variations and the existing protocols.

Regarding the resulting schedulability of the many synchronization protocols for multiprocessor systems, existing and new, it is always difficult to make definitive statements due to partitioning anomalies, induced bias in the selection of task set parameters and pseudo-random generation of utilizations. Considering the scenarios of our empirical study, the proposed variation MPCPNP showed the worst results in the case of suspension-based protocols and good results with the spin-based counterpart. The second variation MPCPF showed good results among the suspension-based protocols, especially in scenarios with low rate of resource sharing. In fact, there is no dominance of a single protocol over the others in the sense that it is always better than all others. This does not occur universally, even in probabilistic terms, because the profile of the application heavily impacts which protocol is better or worse.

Regarding the implementation, we showed that MPCPNP, in both versions, is suitable to be implemented, when compared to FMLP (which was also implemented). The implementation was made on Linux/PREEMPT-RT, which is a low latency version of the widely used Linux kernel and available as a patch. The patch of our implementation is available from www.das.ufsc.br/romulo/artigos/jsa2012.zip.

The implementation was also used in an experiment for the purpose of overhead evaluation. For the case of suspension-based protocols, the results indicate that the overheads of MPCPNP and FMLP are equal, although we can not prove it statistically. This is due to the fact that suspension-based protocols use the scheduler directly, on suspension and resumption of tasks, putting noise from kernel into the experiment. Using spin this noise is drastically reduced, because blocked tasks stay in execution (busy-wait), not involving scheduling code. FMLP had less overhead for small critical sections, whereas MPCPNP had less overhead for larger critical sections.

In short, the proposals showed competitive results, in both schedulability and implementation overhead (in the case of MPCPNP). MPCPF-SUSP showed the best results for suspension-based protocols. MPCPNP with spin presented results similar to FMLP-SHORT, which had the best results among all spin-base protocols. From a practical point of view, the proposed variations facilitate implementation in real systems. Ceiling handling needs offline preprocessing and additional priority levels, requirements that are eliminated in MPCPNP. Another point is that priority ordered queues required by MPCP cannot be implemented by $O(1)$ algorithms, situation solved by MPCPF.

Acknowledgment

The authors would like to thank CNPq and CAPES for financial support.

References

[1] R. Rajkumar, L. Sha, J. Lehoczky, Real-time synchronization protocols for multiprocessors, in: Proceedings of the Real-Time Systems Symposium, IEEE Computer Society, 1988, pp. 259–269. doi:10.1109/REAL.1988.51121.

[2] R. Rajkumar, Real-time synchronization protocols for shared memory multiprocessors, in: Proceedings of the 10th International Conference on Distributed Computing Systems, IEEE Computer Society, 1990, pp. 116–123. doi:10.1109/ICDCS.1990.89257.

[3] A. Block, H. Leontyev, B.B. Brandenburg, J.H. Anderson, A Flexible Real-Time Locking Protocol for Multiprocessors, in: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, IEEE Computer Society, 2007, pp. 47–56. doi:10.1109/RTCSA.2007.8.

[4] E.G. Coffman, M.R. Garey, D.S. Johnson, An application of bin-packing to multiprocessor scheduling, SIAM J. Comput. 7 (1) (1978) 1–17, http://dx.doi.org/10.1137/0207001.

[5] Y. Oh, S.H. Son, T. Hall, Fixed-Priority Scheduling of Periodic Tasks on Multiprocessor Systems, Department of Computer Science, University of Virginia, Tech. Rep. CS-95-16, 1995, pp. 1–37.
[6] K. Lakshmanan, D.D. Niz, R. Rajkumar, Coordinated task scheduling, allocation and synchronization on multiprocessors, in: Proceedings of the 30th IEEE Real-Time Systems Symposium, IEEE, 2009, pp. 469–478. doi:10.1109/RTSS.2009.51.
[7] A. Carminati, R.S. de Oliveira, On variations of the suspension-based multiprocessor priority ceiling syncronization protocol, in: Proceedings of the 17th IEEE International Conference on Emerging Technologies and Factory Automation, IEEE Computer Society, Kraków, Poland, 2012.
[8] R. Rajkumar, Dealing with suspending periodic tasks, IBM Thomas J. Watson Research Center, Yorktown Heights.
[9] L. Sha, R. Rajkumar, J. Lehoczky, Priority inheritance protocols: an approach to real-time synchronization, IEEE Trans. Comput. 39 (9) (1990) 1175–1185, http://dx.doi.org/10.1109/12.57058.
[10] J. Anderson, A. Srinivasan, Mixed Pfair/ERfair scheduling of asynchronous periodic tasks, in: Proceedings of the13th Euromicro Conference on Real-Time Systems, IEEE Comput. Soc., 2001, pp. 76–85. doi:10.1109/EMRTS.2001.934004.
[11] B.B. Brandenburg, J.H. Anderson, An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUSRT, in: Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, IEEE Computer Society, 2008, pp. 185–194. doi:10.1109/RTCSA.2008.13.
[12] T. Baker, A stack-based resource allocation policy for realtime processes, in: Proceedings of the Real-Time Systems Symposium, IEEE Computer Society, 1990, pp. 191–200. doi:10.1109/REAL.1990.128747.
[13] B.B. Brandenburg, Scheduling and Locking in Multiprocessor Real-time Operating Systems, Ph.D. thesis, University of North Carolina at Chapel Hill, 2011.
[14] B.C. Ward, J.H. Anderson, Supporting nested locking in multiprocessor real-time systems, in: Proceedings of the 24th Euromicro Conference on Real-Time Systems, 2012.
[15] P. Gai, G. Lipari, M. Di Natale, Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip, in: Proceedings of the 22nd IEEE Real-Time Systems Symposium, IEEE Computer Society, 2001, pp. 73–83. doi:10.1109/REAL.2001.990598.
[16] P. Gai, G. Lipari, M. Di Natale, Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip, in: Proceedings of the 22nd IEEE real-time systems symposium, IEEE Computer Society, 2001, pp. 73–83. doi:10.1109/REAL.2001.990598.
[17] F. Nemati, T. Nolte, M. Behnam, Blocking-Aware Partitioning for Multiprocessors, Mälardalen Real-Time Research Centre, Mälardalen University, Sweden. Tech. Rep. 2137.
[18] E. Bini, G.C. Buttazzo, Measuring the performance of schedulability tests, Real-Time Syst. 30 (1–2) (2005) 129–154, http://dx.doi.org/10.1007/s11241-005-0507-9.
[19] B.B. Brandenburg, J.M. Calandrino, A. Block, H. Leontyev, J.H. Anderson, Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin?, in: Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, 2008, pp. 342–353. doi:10.1109/RTAS.2008.27.
[20] I. Molnar, T. Gleixner, PREEMPT-RT – Linux with real-time pre-emption patches. https://rt.wiki.kernel.org.
[21] J.M. Calandrino, H. Leontyev, A. Block, U. Devi, J.H. Anderson, LITMUSRT: a testbed for empirically comparing real-time multiprocessor schedulers, in: Proceedings of the 27th IEEE International Real-Time Systems Symposium, IEEE Computer Society, 2006, pp. 111–126. doi:10.1109/RTSS.2006.27.
[22] H.B. Mann, D.R. Whitney, On a test of whether one of two random variables is stochastically larger than the other, Ann. Math. Statist. 18 (1) (1947) 50–60, http://dx.doi.org/10.1214/aoms/1177730491.
[23] S. Rostedt, ftrace – Function Tracer. http://www.mjmwired.net/kernel/Documentation/trace/ftrace.txt.

Andreu Carminati is graduated in Computer Science from Federal University of Santa Catarina (2010). Masters in Automation and Systems Engineering from Federal University of Santa Catarina (2012). He is currently a Ph.D. candidate in the Department of Automation and Systems at the Federal University of Santa Catarina. He has experience in Computer Systems. The main topics of interest are: operating systems and real-time systems.

Rômulo Silva de Oliveira is graduated in Electrical Engineering from Pontificia Universidade Catolica do Rio Grande do Sul (1983), Masters in Computer Science from Federal University of Rio Grande do Sul (1987) and Ph.D. in Electrical Engineering from Universidade Federal de Santa Catarina (1997). He is currently an associate professor in the Department of Automation and Systems, Federal University of Santa Catarina. Also serves on the Graduate Program in Engineering of Automation and Systems at UFSC. The main topics of interest are: real-time systems, scheduling and operating systems.

Luís Fernando Friedrich is a faculty member at the Department of Computer Science, Federal University of Santa Catarina, Brazil. His research interests are in operating systems, distributed computing, real-time computing and embedded computing. Friedrich received his Ph.D. in Engineering from the Federal University of Santa Catarina. He is a member of the Brazilian Computing Society (SBC).