OSPERT2010 - Sixth International Workshop on Operating Systems Platforms
for Embedded Real-Time Applications (in conjunction with ECRTS10)

Implementation and Evaluation of the Synchronization Protocol Immediate Priority Ceiling in PREEMPT-RT Linux

Andreu Carminati, Rômulo Silva de Oliveira, Luís Fernando Friedrich, Rodrigo Lange
Federal University of Santa Catarina (UFSC)
Florianópolis, Brazil
{andreu,romulo,lange}@das.ufsc.br
{fernando}@inf.ufsc.br

Abstract

In general purpose operating systems, such as the mainline Linux, priority inversions occur frequently and are not considered harmful, nor are avoided as in real-time systems. In the current version of the kernel PREEMPT-RT, the protocol that implements the priority inversion control is the Priority Inheritance. The objective of this paper is to propose the implementation of an alternative protocol, the Immediate Priority Ceiling, for use in drivers dedicated to real-time applications, for example. This article explains how the protocol was implemented in the real-time kernel and compares tests on the protocol implemented and Priority Inheritance, currently used in the real-time kernel.

1 Introduction

In real-time operating systems such as Linux/PREEMPT-RT [7, 8], task synchronization mechanisms must ensure both the maintenance of internal consistency in relation to resources or data structures, and determinism in waiting time for these. They should avoid unbounded priority inversions, where a high priority task is blocked indefinitely waiting for a resource that is in possession of a task with lower priority.

In general purpose systems, such as mainline Linux, priority inversions occur frequently and are not considered harmful, nor are avoided as in real-time systems. In the current version of the kernel PREEMPT-RT, the protocol that implements the priority inversion control is the Priority Inheritance (PI) [9].

The objective of this paper is to propose the implementation of an alternative protocol, the Immediate Priority Ceiling (IPC) [5, 9], for use in drivers dedicated to real-time applications, for example. In this scenario, an embedded Linux supports an specific known application that does not change task priorities after its initialization. It is not the objective of this paper to propose a complete replacement of the existing protocol, as mentioned above, but an alternative for use in some situations. The work in this article only considered uniprocessor systems.

This paper is organized as follows: section 2 presents the current synchronization scenario of the mainline kernel and PREEMPT-RT, section 3 explains about the Immediate Priority Ceiling protocol, section 4 explains how the protocol was implemented in the Linux real-time kernel, section 5 compares tests made upon the protocol implemented and Priority Inheritance implemented in the real-time kernel and section 6 presents an overhead analysis between IPC and PI.

2 Mutual Exclusion in the Linux Kernel

Since there is no mechanism in the mainline kernel that prevents the appearance of priority inversions, situations like the one shown in Figure 1 can occur very easily, where task T2, activated at *t = 1*, acquires the shared resource. Then, task T0 is activated at *t = 2* but blocks because the resource is held by T2. T2 resumes execution, and is preempted by T1, which is activated and begins to run from *t = 5* to *t = 11*. But task T0 misses the deadline at *t = 9*, and the resource required for its completion was only available at *t = 12* (after the deadline).

In the real-time kernel PREEMPT-RT exists the implementation of PI (Priority Inheritance). As mentioned above, it is a mechanism used to accelerate the release of resources in real-time systems, and to avoid the effect of indefinite delay of high priority tasks that can be blocked waiting for resources held by tasks of lower priority.

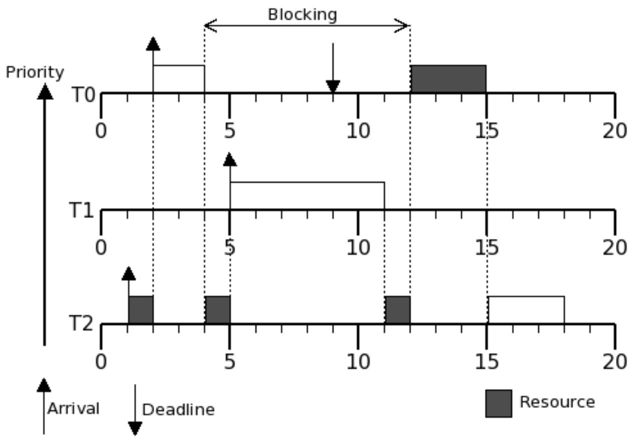

Figure 1. Priority inversion

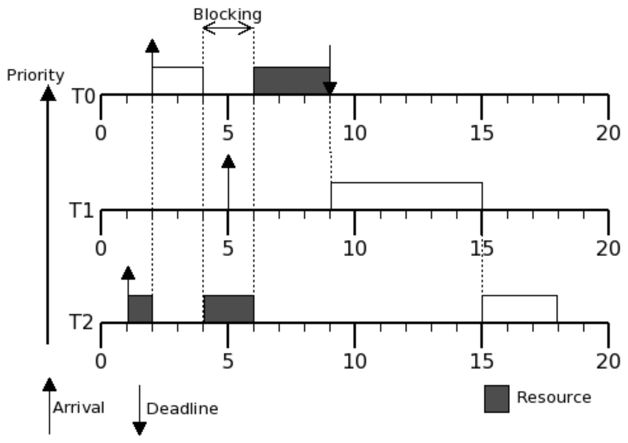

Figure 2. Priority inversion resolved by PI

In the PI protocol, a high priority task that is blocked on some resource, gives its priority to the low priority task (holding that resource), so will release the resource without suffering preemptions by tasks with intermediate priority. This protocol can generate chaining of priority adjustments (a sequence of cascading adjustments) depending on the nesting of critical sections.

Figure 2 presents an example of how the PI protocol can help in the problem of priority inversion. In this example, task T2 is activated at t = *1* and acquires a shared resource, at t = *1*. Task T0 is activated and blocks on the resource held by T2 at t = *4*. T2 inherits the priority from T0 and prevents T1 from running, when activated at t = *5*. At t = *6*, task T2 releases the resource, its priority changes back to its normal priority, and task T0 can conclude without missing its deadline.

Some of the problems [11] of this protocol are the number of context switches and blocking times larger than the largest of the critical sections [9] (for the high priority task), depending on the pattern of arrivals of the task set that shares certain resources.

Figure 3 is an example where protocol PI does not prevent the missing of the deadline of the highest priority task. In this example, there is the nesting of critical sections, where T1 (the intermediate priority) has the critical sections of resources 1 and 2 nested. In this example, task T0, when blocked on resource 1 at t = *5*, gives his priority to task T1, which also blocks on resource 2 at t = *6*. T1 in turn gives its priority to task T2, which resumes its execution and releases the resource 2, allowing T1 to use that resource and to release the resource 1 to T0 at t = *10*. T0 resumes its execution but it misses its deadline, which occurs at t = *13*. In this example, task T0 was blocked by a part of the time of the external critical section of T1 plus a part of the time of the critical section of T2. In a larger system the blocking time of T0 in the worst case would be the sum of many critical sections, mostly associated with resources not used by T0.

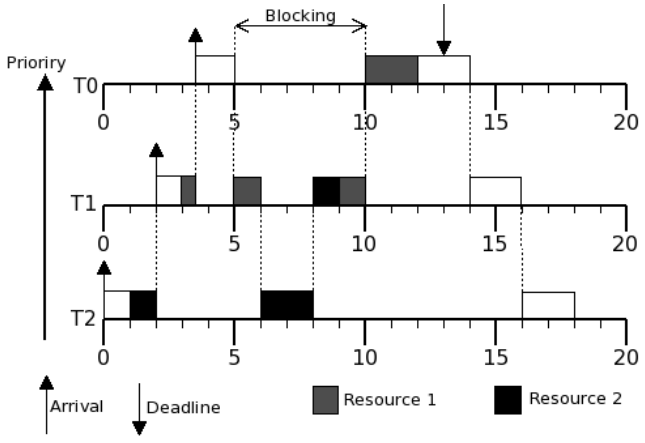

Figure 3. Priority inversion not resolved by PI

3 The Immediate Priority Ceiling Protocol

The Immediate Priority Ceiling (IPC) [2] synchronization protocol for fixed priority tasks, is a variation of Priority Ceiling Protocol [9, 1] or *Highest Locker Priority*. This protocol is an alternative mechanism for unbounded priority inversion control, and prevention of deadlocks in uniprocessor systems.

In the PI protocol, the priority is associated only to tasks. In IPC, the priority is associated with both tasks and resources. A resource protected by IPC has a priority ceiling, and this priority is the highest priority of all task priorities that access this resource.

According to [3], the maximum block time of a task under fixed priority using shared resource protected by IPC protocol is the larger critical section of the system whose priority ceiling is higher than the priority of the task in question and is also used by a lower priority task.

What happens in IPC can be considered as preventive inheritance, where the priority is adjusted immediately when occurs a resource acquisition, and not when the resource becomes necessary to a high priority task, as in PI (you can think of PI as the IPC, but with dynamic adjustment of the ceiling). This preventive priority setting prevents low priority tasks from being preempted by tasks with intermediate priorities, which have priorities higher than low priority tasks and lower than the resource priority ceiling.

Figure 4 shows an example similar to that shown in Figure 3, but this time using IPC. In this example, the high priority task does not miss its deadline, because when task T2 acquires resource 2 at *t = 1*, its priority is raised to the ceiling of the resource (priority of T1), preventing task T1, activated at *t = 2*, from starting its execution. At *t = 3.5*, task T0 is activated and begins its execution. The task is no longer blocked because resource 1 is available. Task T0 does not miss its deadline.

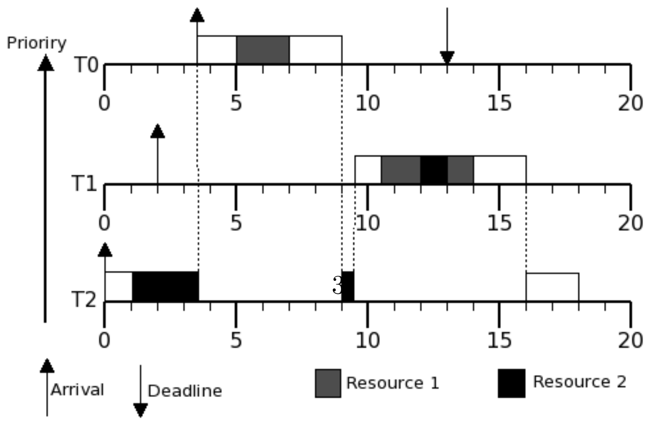

Figure 4. Priority inversion resolved by IPC

4 Description of the Implementation

The Immediate Priority Ceiling Protocol was implemented based on the code of rt_mutexes already in the patch PREEMPT-RT. The rt_mutexes are mutexes that implement the Priority Inheritance protocol. The kernel version used for implementation was the 2.6.31.6 [10] with PREEMPT-RT patch rt19. Although rt_mutexes are implemented in PREEMPT-RT for both uniprocessor and multiprocesors, our implementation of IPC considers only the uniprocessor case.

The implementation was made primarily for use in device-drivers (kernel space), as shown in Figure 5, where there is an example of tasks sharing a critical section protected by IPC and accessed through an ioctl system call to a device-driver.

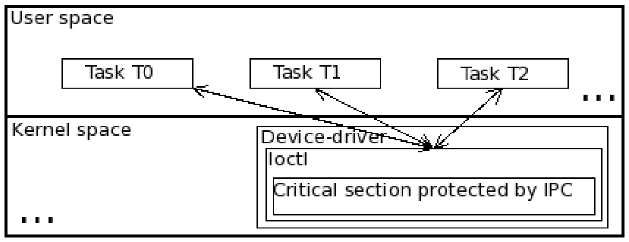

Figure 5. Diagram of interaction between the IPC protocol and tasks

The type that represents the IPC protocol was defined as *struct ipc_mutex*, and it is presented in code 1. In this structure, *wait_lock* is the spinlock that protects the access to the structure, *wait_list* is an ordered (by priorities) list that stores pending lock requests, *on_task_entry* serves to manage the locks acquired (and, consequently, control of priorities), *owner* stores a pointer to the task owner of the mutex (or null pointer if the mutex is available) and finally the *ceiling*, which stores the priority ceiling of the mutex.

Code 1 Data structure that represents a IPC mutex

```
struct ipc_mutex {
        atomic_spinlock_t wait_lock;
        struct plist_head wait_list;
        struct plist_node on_task_entry;
        struct task_struct *owner;
        int ceiling;
...
};
```

The proposed implementation presents the following

API of functions and macros:

- **DEFINE_IPC_MUTEX(mutexname, priority):** This macro is provided for the definition of a static IPC mutex, where *mutexname* is the identifier of the mutex and priority is the ceiling of the mutex, or a value in the range of 0 to 99 (according to the specification of static priorities for real-time tasks on Linux). The current version can only create mutexes with priorities set at compile time, thus, the priority ceiling should be assigned by the designer of the application. This is not a too restrictive assumption when an embedded Linux runs a known application that does not change task priorities after its initialization.

- **void ipc_mutex_lock(struct ipc_mutex * lock):** Mutex acquisition function. In uniprocessor systems this is a nonblocking function because, according to the IPC protocol, if a task requests a resource, it is because this resource is available (the owner is null). In multiprocessor systems this function can generate blocks, because the resource can be in use on other processor (the *owner* field differs from zero). In this article, only the uniprocessor version will be taken into consideration. The main role of this function is to manage the priority of the calling task along with the resource blocking, taking into account all ipc_mutexes acquired so far.

- **void ipc_mutex_unlock(struct ipc _mutex * lock):** Effects the release of the resource and the adjustment of the priority of the calling task. In multiprocessor systems, this function also makes the job of selecting the next task (by the *wait_list*) that will use the resource. What also occurs in multiprocessor systems is the effect "next owner pending" (also present in the original implementation of priority inheritance) also known as steal lock, where the task of highest priority can acquire the mutex again, even though it has been assigned to another task that did not take possession of it.

One major difference between the proposed implementation and the already existent in the PREEMPT-RT is that the latter one enables the following optimizations:

- PI can perform atomic lock: if a task attempts to acquire a mutex that is available, this operation can be performed atomically (operation atomic compare and exchange) by what is known as fast path. But this is only possible for architectures that have this type of atomic operation. Otherwise if the lock is not available and/or the architecture does not have exchange and compare instruction, the lock will not be atomic (slow path).

- PI can perform atomic unlock: when a task releases a mutex which has no tasks waiting, this operation can be performed atomically.

As mentioned earlier, these optimizations are possible only for PI mutexes. In the case of IPC, there will always be a need for verification and a possible adjustment of priority.

5 Implementation Analysis

We developed a device-driver that has the function of providing the critical sections necessary to perform the tests. This device-driver exports a single service as a service call ioctl (more specifically unlocked_ioctl, because the ioctl is protected by traditional Big Kernel Lock, which certainly would prevent the proper execution of the tests). It multiplexes the calls of the three tasks in their correspondent critical sections. This device-driver provides critical sections to run with both IPC and PI.

In order to carry out tests for the analysis of the implementation it was used a set of sporadic tasks executed in user space. The interval between activations, the resources used and the size of the critical section within the device-driver used by each task are presented in Table 1. All critical sections are executed within the function ioctl, within Linux kernel. A high-level summary of actions performed by each task (in relation to resources used) is presented in Table 2.

Table 1 shows the intervals between activations expressed with a pseudo-randomness, ie, with values uniformly distributed between minimum and maximum values. This randomness was included to improve the distribution of results because, with fixed periods, arrival patterns were being limited to a much more restricted set. Table 1 also presents the sizes of the critical sections of each task. Other information shown in Table 1 is the number of activations performed for each task. For the high priority task, there were 1000 monitored activations (latency, response time, critical section time, lock time, etc). For other tasks there was no restriction on the number of activations.

The high priority task has one of the highest priorities of the system. The other tasks were regarded as medium and low priorities. But they also have absolute high priorities. All tasks have been configured with the scheduling policy SCHED_FIFO, which is one of the policies for real-time [4] available in Linux.

Even with the use of a SMP machine for testing, all tasks were set at only one CPU (CPU 0). The tests were conducted using both IPC and PI for comparison purposes.

Task	T0/High	T1/Med.	T2/Low
Priority	70	65	60
Activation interval	rand in [400,800] ms	rand in [95,190] ms	rand in [85,170] ms
Resource	R1	R1,R2	R2
Critical section size	aprox. 17 ms	aprox. 2x17 ms	aprox. 17 ms

Table 1. Configuration of the set of tasks

Task	T0/High	T1/Med.	T2/Low
Action 1	Lock(R1)	Lock(R1)	Lock(R2)
Action 2	Critical Sec.	Critical Sec.	Critical Sec.
Action 3	Unlock(R1)	Lock(R2)	Unlock(R2)
Action 4		Critical Sec.	
Action 5		Unlock(R2)	
Action 6		Unlock(R1)	

Table 2. Actions realized by tasks

Mutex R1 has been configured with priority ceiling 70 (which is the priority of task T0) and R2 has been configured with priority ceiling 65 (which is the priority of task T1).

5.1 Results of the Use of the PI mutex

With priority inheritance, the high priority task had activation latencies as can be seen in the histogram of Figure 6 appearing in the interval [20000, 30000] nanoseconds (range where the vertical bar is situated in the histogram). Because of finding the resource busy with a certain frequency (as illustrated in Figure 7, waiting time for the resource), the task was obligated to perform volunteer context switch for propagation of its priority along the chain of locks.

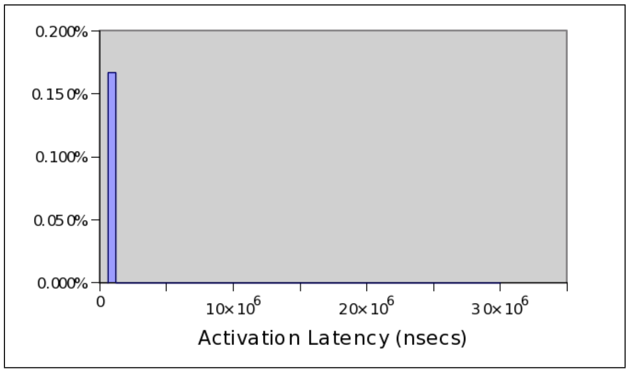

Figure 6. Histogram of activation latencies (high priority task using PI)

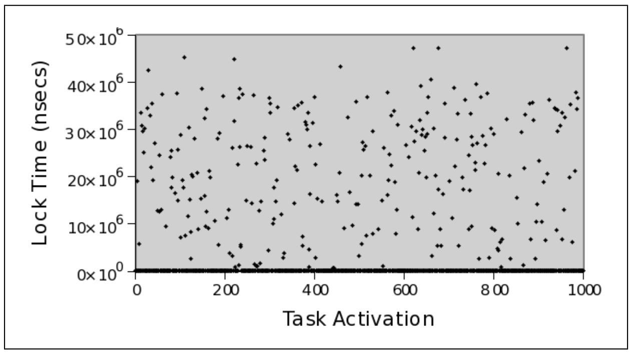

Figure 7. Blocking time (high priority task using PI)

Regarding the response time (as can be seen in the histogram of Figure 8) it was consistent with the blocking time sustained, with a maximum of nearly 3 times the size of the critical section, in accordance with the task set. It can be seen in Table 3 the worst-case response time observed is 64,157,591 ns. The theoretical worst-case response time for this test would be, with an appropriate synchronized activation, 68 ms, or 17 ms own critical section of task T0 added to 34 ms of task T1 and 17 ms of task T2. In this test, there is a good approximation of the theoretical limit.

5.2 Results of the Use of the IPC mutex

Using IPC, it can be noted in the histogram of Figure 9 that the task of highest priority presented, with low

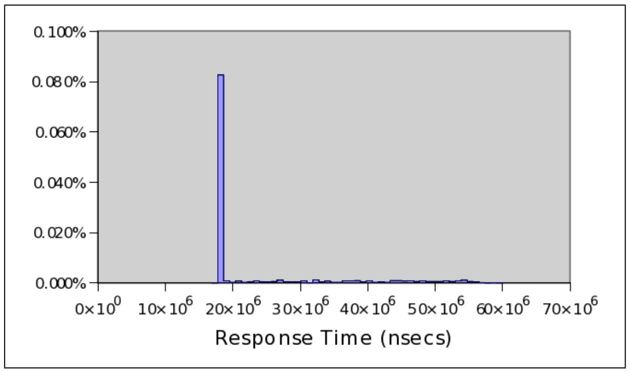

Figure 8. Histogram of response time (high priority task using PI)

Protocol:	PI	IPC
Average response time:	22,798,549 ns	21,014,311 ns
Std dev:	11,319,355 ns	8,723,159 ns
Max:	64,157,591 ns	50,811,328 ns

Table 3. Average response times and standard deviations

frequency, varying values of activation latency (seen in the tail of the histogram). Waiting times set by the resource appear in Figure 10, which is expected according to the definition of the protocol implemented. A tail appears in the histogram of response time (Figure 11) due to activation latency (higher values, but with only a few occurrences).

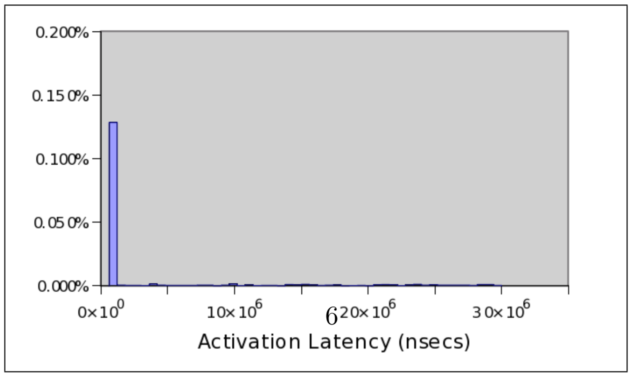

Figure 9. Histogram of activation latencies (high priority task using IPC)

As it can be seen in Table 3, the worst-case response time observed is (maximum) 50,811,328 ns. In this test, the theorical limit is 51 ms, ie, 17 ms own critical section of task T0 added to 34 ms of task T1. Also in this test there is a good approximation of the theorical limit.

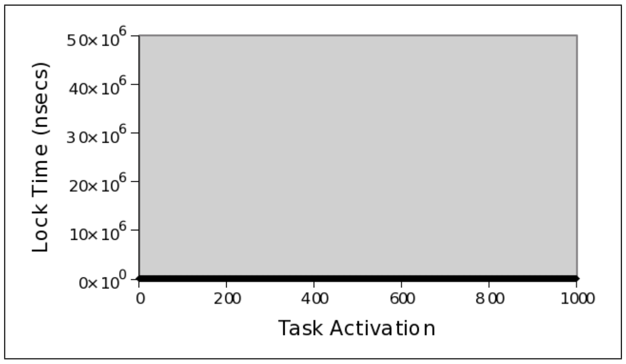

Figure 10. Blocking time (high priority task using IPC)

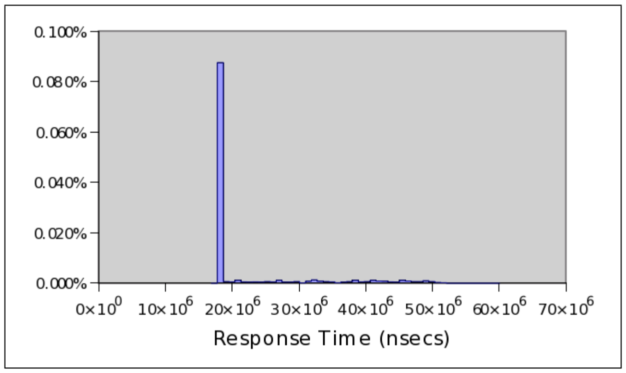

Figure 11. Histogram of response time (high priority task using IPC)

5.3 Comparison between PI and IPC

One can observe that, in general, IPC has behavior similar to PI. The differences appear in the lock time where, by definition, in uniprocessor systems, the resource is always available when using IPC protocol. For the PI protocol, the blocking time will appear with the primitive lock, and this time may be longer than that with IPC. In IPC the blocking time appears before the activation time, and it has a maximum length of a single critical section (in the conditions described above).

According to Table 3, the IPC protocol presented standard deviation and average response time smaller than PI. Another important point in Table 3 is that the worst-case response time observed in the IPC test was almost a critical section smaller than the PI (the size of a critical section is 17 ms, and the difference between the worst case of IPC and PI is around 14 ms).

Figure 12 shows the tail of the response time histograms of IPC and PI combined. In this figure, the response times of the IPC protocol concentrates on lower values. For the PI, these are distributed more uniformly to higher values, indicating an average response time smaller for the IPC protocol. This histogram also indicates in its final portion that the worst case, as it was also observed in Table 3, has a difference of one critical section in favor of the IPC protocol. This difference in the worst case was reported in the figure by two vertical lines, where the distance between them is about the duration of one critical section. Table 4 summarizes the results qualitatively.

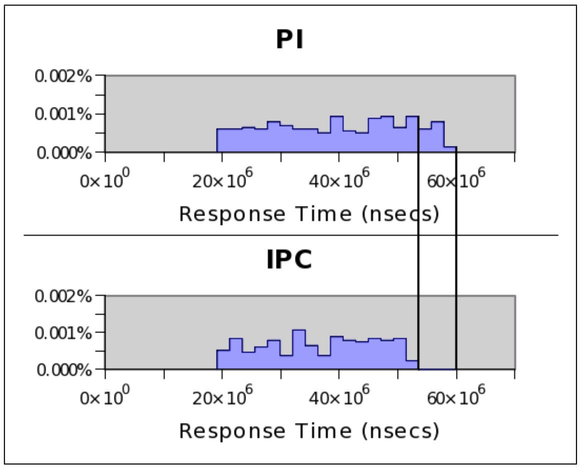

Figure 12. Histogram of the response time of the high priority task

Protocol	**PI**	**IPC**
Activation Latency	Not varied	Varied
Blocking time	Varied	Not varied
Response time	Blocking time dependent	Latency dependent

Table 4. Summary of expected results

6 Implementation Overhead

We define overhead as any decrease in the system's ability to produce useful work. Thus, for this study, the overhead will be considered as the reducing of the CPU time available to the rest of the system, given the presence of a set of higher priority tasks sharing resources protected by PI or IPC.

To evaluate the protocol implemented in terms of overhead imposed on the system, we used a set of test tasks as specified in Table 5. In the same table, it is presented the tasks configurations, some of which are identical to the tasks used to evaluate the protocol in the previous section. For example, for task T0', the size of the critical section is equal to the size of the critical section of task T0 of the previous test (represented by "== T0").

To perform an estimative of the overhead, it was created a measuring task with priority 51 (with policy SCHED_FIFO). This priority is above the default priority of threaded irq handlers [8] and softirqs [6] . This was done to keep the measuring task above the interference of the mechanisms of interrupt handling and work postponement of Linux. Every CPU time that remains (not used by the test tasks synchronized by IPC or PI) is then assigned to the measurement task. Both the measurement task and the task set synchronized by IPC or PI were fixed to a single CPU (CPU 0 in a system with 2 cores.)

The measurement task is activated before the activation of real-time tasks and ends after they terminate. In each test iteration, the measurement task runs for 9 seconds, and the higher priority tasks begin 1 second after it starts. As shown in Table 5, task T0' executes 10 activations, the others will run until the end of this task.

Task	**T0'**	**T1'**	**T2'**	**T3'**	**T4'**	**T5'**	**T6'**
Priority	70	65	64	63	62	61	60
Activation interval	== T0	== T1	== T1	== T1	== T2	== T2	== T2
Resource	R1	R1, R2	R1, R2	R1, R2	R2	R2	R2
Critical section size	== T0	== T1	== T1	== T1	== T2	== T2	== T2
Number of activ.	10	T0' dep	T0' dep	T0' dep	T0' dep	T0' dep	T0' dep

Table 5. Actions realized by tasks

To obtain the overhead estimative, the measurement task is executed in an infinite loop incrementing a variable by the time specified above (9 seconds). The overhead will be noticed by how much the measurement task could increment a count, taking into account the execution of the set of tasks synchronized by IPC or PI. The values of the counts made by the measurement task are presented in Table 6, which was ordered to facilitate visual comparison.

IPC	PI
187,882,717	188,776,389
188,035,384	189,155,169
188,160,733	189,202,563
188,194,113	189,263,630
188,207,825	189,331,186
188,240,432	189,361,353
188,563,788	189,387,326
188,603,802	189,418,120
188,616,385	189,428,218
188,703,718	189,437,569
188,736,889	189,447,533
188,742,876	189,471,453
188,935,538	189,475,286
188,952,045	189,489,740
188,962,343	189,492,896
188,993,374	189,494,791
189,000,638	189,569,661
189,178,721	189,572,258
189,203,245	189,604,953
189,307,878	189,638,715
189,478,899	189,696,190
189,536,986	189,778,227
189,674,412	189,825,606
189,785,580	189,867,362
189,858,951	190,046,853
189,900,585	190,207,326
190,030,444	190,252,943
190,047,436	190,342,313
190,066,156	190,349,981
190,105,987	190,387,518
190,328,052	190,538,130
190,338,011	190,539,875

Table 6. Counter values of measurement task

Table 7 presents the basic statistical data related to the samples presented in Table 6.

To evaluate the results we used the statistical hypothesis test for averages with unknown variance (Student t-test). By hypothesis, the overhead of PI and IPC are equal, ie, the average of IPC and PI are equal ($H_0 : \mu_{PI} = \mu_{IPC}$).

Protocol	IPC	PI
Average(μ)	189,136,685.72	189,682,847.91
Var.(S_x^2)	524,003,070,588.27	191,603,683,258.35
Minimum	187,882,717	188,776,389
Maximum	190,338,011	190,539,875

Table 7. Basic statistics of the found values

The data presented in Table 8, which provides the data necessary for the hypothesis test, was obtained from the data showed in Table 7 plus the information of the number of samples (n = *32*)

$Sa^2_{IPC,PI}$	357,803,376,923.31
$Sa_{IPC,PI}$	598,166.68
n	32
α	0.1%
$d.f.$	60
t	-3.65

Table 8. Student's t-test data

6.1 Analysis of the Results

Because the data produced the value of t = 3.65, which does not belong to the region of acceptance (in t-Student distribution), the test rejects H_0 with a significance level of 0.1 %. At significance level (α) of 0.1%, the collected data indicates a difference between PI and IPC. There is a probability smaller than 0.1% that the differences observed on the presented data are from casual factors of the system only.

These differences are likely due to the fast path of the PI implementation. In IPC, there is always a need of priority verification, and this can not be performed atomically. Another point is that if a task with priority below the priority ceiling of a given resource acquires that resource, its priority has to be changed, and this may influence the overhead. As those tests show, there is a reasonable probability of tasks finding resources available, not always the priorities propagation algorithm (PI) will run. But almost always there will be priority adjustments (IPC), except for the task that defines the priority ceiling of the resource.

7 Conclusions

Task synchronization is fundamental in multitasking and/or multithread systems, specially in real-time systems. In addition to protection against race conditions, these mechanisms must prevent the emergence of uncontrolled priority inversions, which could cause the missing of deadline, leading real-time applications to present incorrect behavior, and possibly harmful consequences (depending on the application). In this context, it was proposed an alternative for some applications to the protocol implemented in the real-time Linux branch.

The IPC protocol may be suitable for dedicated applications that use architectures without instruction compare and exchange because, in this way, the implementation may not use the fast path (via atomic instructions). Another advantage of the IPC is that it generates less context switches than the PI, inducing faster response times due to switching overhead as well as lower failure rates in the TLB.

One of the disadvantages of the IPC for wider use is the need for manual determination of the priority ceiling of IPC mutexes. But this is not a problem for automation and control applications for example. Dedicated device-drivers are fully aware of the priorities of the tasks that access them, justifying the manual setting of the ceiling in this case.

As seen in the tests, the PI protocol may be more appropriate if the latency of activation is important. But if the blocking time is more relevant, IPC may be the best solution. In terms of average response time, the two solutions were similar, but IPC showed lower average response time probably due to the latency of activation being less than the waiting time of the PI. Another point in favor of the IPC protocol appears when we compare the difference in the worst-case response time observed in the tests since the IPC case was about a critical section smaller than in the PI case, as can be seen in Table 3. The PI protocol has a response time that may vary depending of the pattern of resource sharing and sequences of activation, which does not occur with IPC. Its blocking time will always be at most one critical section.

Although blocking/response times are smaller in the IPC, tests show that the overhead of IPC implemented is greater than the native PI in PREEMPT-RT. This overhead is most likely caused by the absence of a fast path in the implementation of IPC. There is a set of operations on lock/unlock that can not be executed atomically as in PI. These operations involve priority changes and tracking mutexes acquired by tasks.

As future work, we intend to implement a version with adaptive ceiling, ie, ceiling can be automatically adjusted in run-time. There is still a possibility (which was not considered in this article) to build a fast-path. To make this possible, the priority adjustments should be postponed until the eminence of a system rescheduling (through changes in the scheduler). We also intend to expand the study of the IPC protocol to multiprocessor systems.

8 Acknowledgments

To CAPES and CNPq for financial suport.

References

[1] T. Baker. A stack-based resource allocation policy for realtime processes. In *IEEE Real-Time Systems Symposium*, volume 270, 1990.

[2] A. Burns and A. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Addison Wesley, 2001.

[3] M. Harbour and J. Palencia. Response time analysis for tasks scheduled under EDF within fixed priorities. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 200. Citeseer, 2003.

[4] C. S. IEEE, editor. *POSIX.13. IEEE Std. 1003.13-1998. Information Technology - Standardized Application Environment Profile - POSIX Realtime Application Support (AEP)*. The Institute of Electrical and Electronics Engineers, 1998.

[5] B. Lampson and D. Redell. Experience with processes and monitors in Mesa. 1980.

[6] R. Love. *Linux Kernel Development (Novell Press)*. Novell Press, 2005.

[7] I. Molnar. Preempt-rt. http://www.kernel.org/pub/linux/kernel/projects/rt - Last access 01/21, 2010.

[8] S. Rostedt and D. Hart. Internals of the RT Patch. In *Proceedings of the Linux Symposium*, volume 2007, 2007.

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

[10] L. Torvalds. "Linux Kernel Version 2.6.31.6", 2010. http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.31.6.tar.bz2 - Last access 03/21, 2010.

[11] V. Yodaiken. Against priority inheritance. *FSM-LABS Technical Paper*, 2003. Available at `http://yodaiken.com/papers/inherit.pdf`.