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

Andreu Carminati
Department of Automation and Systems, Federal University of Santa Catarina, Florianópolis, Brazil
Email: andreu@das.ufsc.br

Rômulo Silva de Oliveira
Department of Automation and Systems, Federal University of Santa Catarina, Florianópolis, Brazil
Email: romulo@das.ufsc.br

Luís Fernando Friedrich
Department of Informatics and Statistics, Federal University of Santa Catarina, Florianópolis, Brazil
Email: fernando@inf.ufsc.br

Abstract— **In general purpose operating systems, such as the mainline Linux, priority inversions occur frequently and are not considered harmful. They are not avoided or limited as in real-time systems. In the current version of the kernel PREEMPT-RT Linux, the protocol used for 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. This paper explains how the protocol was implemented in the real-time kernel and compare the protocol implemented with the Priority Inheritance implementation, currently used in the real-time kernel.**

Index Terms— **real-time systems; Linux; process synchronization**

I. Introduction

In real-time operating systems such as Linux/PREEMPT-RT [1] [2], task synchronization mechanisms must ensure both the maintenance of the internal consistency of resources and data structures, and the determinism of waiting times. They should avoid unbounded priority inversions, where a high priority task is blocked indefinitely waiting for a resource that is possessed by 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 Linux, the protocol used for priority inversion control is the Priority Inheritance (PI) [3].

The objective of this paper is to propose the implementation of an alternative protocol, the Immediate Priority Ceiling (IPC) [4] [3], for use in drivers dedicated to real-time applications. In this scenario, an embedded Linux supports a 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 paper only considered uniprocessor systems. A preliminary version of this paper was presented at [5].

A disadvantages of IPC for wider use is the need for manual determination of the priority ceiling of IPC mutexes. But this is usually not a problem for embedded systems. Dedicated device-drivers are fully aware of the priorities of the tasks that access them, justifying the manual setting of the ceiling (either at compile time or initialization) in this case.

The Linux kernel was chosen because it is an attractive alternative for a large spectrum of applications, from laptops and desktops to big servers. It is also used more and more in embedded applications, in part because of the growing popularization of 32 bits architectures.

The widespread use of Linux is consequence of all the advantages offered by this modern general purpose operating system, such as a multitask environment, communication protocol stacks, graphical resources, wide hardware support, code stability, continuous evolution and constant modernization for elimination of bugs. Another advantage of the use of Linux is the possibility of studying it, to alter and to do any kind of adjustment it may be necessary in order to adapt it to a certain embedded application.

This paper is organized as follows: section II presents the current synchronization scenario of the mainline kernel and PREEMPT-RT, section III explains the Immediate Priority Ceiling protocol, section IV explains how the protocol was implemented in the Linux real-time kernel, section V describes tests made upon the protocol implemented and the original Priority Inheritance implemented in the real-time kernel and section VI presents an overhead analysis of IPC and PI.

doi:10.4304/jsw.7.3.516-525

II. Mutual Exclusion in the Linux kernel

There is no mechanism in the mainline kernel that prevents the appearance of priority inversions in kernel code (it is available for use in code that runs in userspace through futexes). 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 $t = 9$, since the resource required for its completion was only available at $t = 12$ (after the deadline).

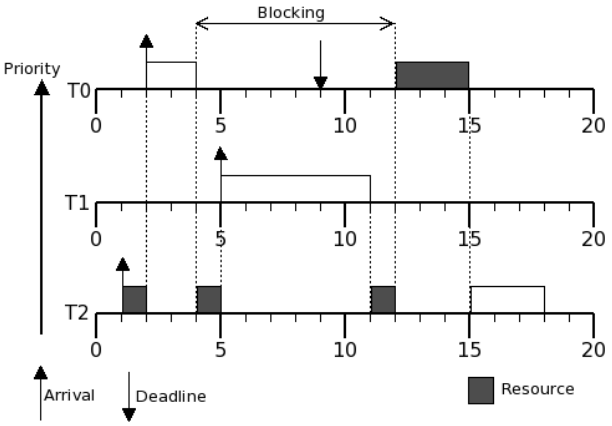

Figure 1. Unbounded priority inversion

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

In the PI protocol, a task of high priority that is blocked on some resource gives its priority to the task of lower priority (holding that resource), so this will release the resource without suffering preemptions of 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 [6] of this protocol are the number of context switches and blocking times larger than the largest of the critical sections [3] (for the task of high priority), depending on the arrival pattern of the tasks that share certain resources.

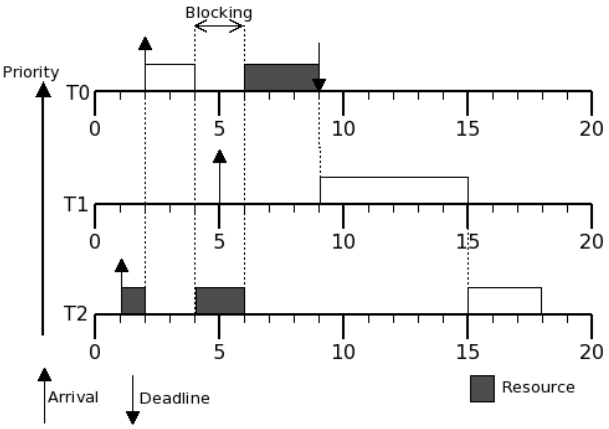

Figure 2. Priority inversion avoided by PI

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. T1 (the intermediate priority) has the critical sections defined by resources 1 and 2 nested. In this example, task T0, when blocked on resource 1 at $t = 3$, gives its priority to task T1, which also blocks on resource 2 at $t = 3$. 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 $t = 7$. T0 resumes its execution but it misses its deadline, which occurs at $t = 11$. In this example, the worst case blocking time of task T0 is the time of the external critical section of T1 plus 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.

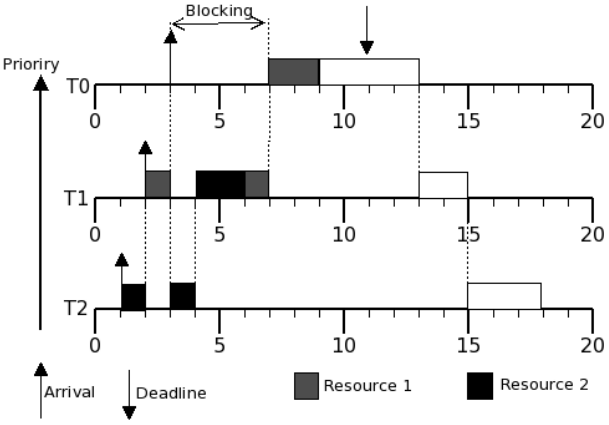

Figure 3. Priority inversion not avoided by PI

III. The Immediate Priority Ceiling Protocol

The Immediate Priority Ceiling (IPC) [7] synchronization protocol for fixed priority tasks is a variation of the Priority Ceiling Protocol [3] and the Stack Resource Protocol [8]. It is sometimes referenced as the 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, priority is associated only to the tasks. In IPC, 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 [9], the maximum blocking time of a task under fixed priority using a shared resource protected by IPC protocol is the larger critical section of the system among those which priority ceiling is higher than the priority of the task in question and is used by a lower priority task.

What happens in IPC can be considered 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. One 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 misses its deadline, because when task T2 acquires the 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*, task T0 is activated and begins its execution. The task is no longer blocked because the resource 2 is available. Task T0 does not miss its deadline.

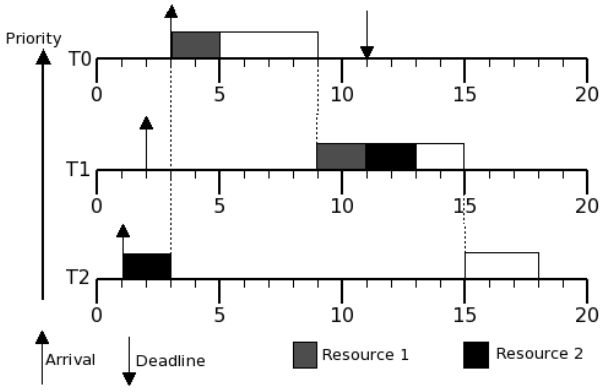

Figure 4. Priority inversion avoided by IPC

IV. Description of the Implementation

The Immediate Priority Ceiling Protocol was implemented based on the code of rt_mutexes existing in the PREEMPT-RT. The rt_mutexes are mutexes that implement the Priority Inheritance protocol [10]. Our implementation is currently based on the tip tree and rt/head branch [11]. 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). Figure 5 shows an example of tasks sharing a critical section protected by IPC and accessed through an ioctl system call in a driver. This is a very common scenario when Linux is used in an embedded system.

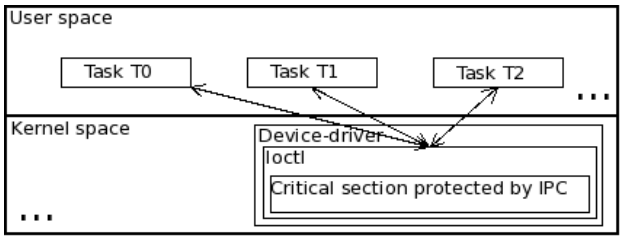

Figure 5. Diagram of interaction between the IPC protocol and tasks

The type that represents the IPC protocol was defined as *struct immpc_mutex*, and is presented in code 1. In this structure, *wait_lock* (line 2) is the spinlock that protects the access to the structure, *on_task_entry* (line 3) serves to insert the immpc_mutex structure in a list (and, consequently, control of priorities), *owner* (line 4) stores a pointer to the task owner of the mutex (or null pointer if the mutex is available) and finally the *ceiling* (line 5), which stores the priority ceiling of the mutex.

Code 1 Data structure that represents a IPC mutex

```
1 struct immpc_mutex {
2   raw_spinlock_t          wait_lock;
3   struct list_head        on_task_entry;
4   struct task_struct      *owner;
5   int                     ceiling;
6   /*other fields*/
7 };
```

Another auxiliary structure was defined, the struct *immpc_synchronization_ctx*, showed in code listing 2. This structure was included in the *task_struct* for the control of the IPC mutexes, where each mutex acquired by a task is added to the *mutex_list* (line 5) (by the *on_task_entry* field on *immpc_mutex* struct).

Code 2 Data structure that represents a IPC mutex synchronization context

```
1 struct immpc_synchronization_ctx{
2 #if defined(CONFIG_IMMPC_DELAYED_PRIO_ADJ)
3     int         need_prio_change;
4 #endif
5   struct    list_head mutex_list;
6 };
```

The proposed implementation presents the following API of functions and macros:

- **DEFINE_IMMPC_MUTEX(mutexname, priority):** This macro is provided for definition of a statically allocated 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).

- **void immpc_mutex_init(struct immpc_mutex *lock, int prio):** This function initializes a dynamically allocated (or embedded in a struct) mutex. The result is a unlocked mutex.
- **void immpc_mutex_lock(struct immpc_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 always null).
- **void immpc_mutex_unlock(struct immpc_mutex * lock):** Effects the release of the resource and the readjustment of the priority of the calling task.
- **void immpc_mutex_set_ceiling(struct immpc_mutex *lock, int newceiling):** This function changes the ceiling of the specified mutex. The adjustment policy (via procfs, on module loading, or ioctl, for example) must be defined by the driver developer.

A. Optimization by Priority Change Postponement

The execution of the lock/unlock operations may become very expensive, especially when these operations occur very frequently within some kernel segment (in a short loop for example). Ideally, there should be some form of optimization in order to accelerate this process, which is done by what is called a fastpath.

Fastpath is the strategy of acquiring or releasing a mutex without blocking the spinlock which protects the mutex structure. It is used primarily for performance reasons, and is implemented with the use of atomic instructions (instructions known as compare and exchange - cmpxchg). For architectures that do not include this instruction, the fastpath can not be used.

The fastpaths that we implemented for the Immediate Priority Ceiling are very similar to those existing in original implementation of rt-mutexes (a small amount of code was changed). In PI protocol (rt-mutexes), which is implemented to work on multiprocessors on Linux/PREEMPT-RT, when the mutex can not be acquired/released by fastpath because some required condition was not met it is executed the slowpath (the mutex is not available, ie, it was held by a thread which is not running or is running on another CPU). Slowpath is the traditional mechanism of mutex acquisition/release, which blocks the data structure.

In the IPC protocol (immpc_mutex) implementation (for monoprocessors on Linux/PREEMPT-RT), the fastpath will never fail (the mutex is always available when requested by IPC). In this case, the slowpath only will be used if the architecture does not support cmpxchg or the

Actually, one of the protocol problems is the fact that the IPC priority adjustment of tasks that acquire a mutex can not be performed atomically. What we did to enable the fastpath under this restriction was to implement a mechanism to postpone priority adjustments. A task that acquires a mutex has a flag (defined on code listing 2, line 3) turned on and passes to a state defined as "Task with priority adjustment pending." This postponement means that the adjustment should be done at a more favorable time. The most appropriate time to do this is the imminence of a preemption or a CPU rescheduling.

One of the advantages of priority change postponement is the creation of a fastpath. Another advantage is due to the fact that the adjustment does not need to be done in the case of small critical sections, since the smaller the critical section, the lower the probability of a rescheduling while it executes (motivated by a timer interrupt for example).

Even the acquisition of the mutex being atomic in the fastpath, it is still required two non atomic operations: enable the flag "Task with priority adjustment pending" and add/remove the mutex in the task mutex list. These operations are lockfree because they are made only by the task which just acquired the mutex.

Code listing 3 shows a simplified algorithm for the fastpath acquisition/release. The non-atomic part is performed by calls to track_ipc_mutex and untrack_ipc_mutex. Functions track_ipc_mutex and untrack_ipc_mutex add and remove, respectively, the mutex to the task mutexes list, in addition to activating the flag of delayed priority adjustment. When the architecture does not support cmpxchg instructions, the macro immpc_mutex_cmpxchg (line 2) simply returns 0, forcing the acquisition/release by slowpath.

Code 3 Simplified algorithm of acquisition/release fastpath

```
1 #if defined(_HAVE_ARCH_CMPXCHG)
2 #define immpc_mutex_cmpxchg(l,c,n)\
3     (cmpxchg(&l->owner, c, n) == c)
4 #else
5 #define immpc_mutex_cmpxchg(l,c,n) (0)
6
7 void immpc_mutex_fastlock(struct immpc_mutex*
8                           lock){
9
10    /*will fail if the system does not supports
11      cmpxchg or is not compiled for pastpath
12      utilization.*/
13    if(immpc_mutex_cmpxchg(lock->owner, NULL,
14                           current)){
15        track_immpc_mutex(lock, current);
16    } else {
17
18        /*traditional process*/
19        immpc_mutex_slowlock(lock);
20    }
21 }
22
23 void immpc_mutex_fastunlock(struct immpc_mutex*
24                             lock){
25
26    /*will fail if the system does not supports
27      cmpxchg or is not compiled for pastpath
28      utilization.*/
29    if(immpc_mutex_cmpxchg(lock->owner,
30                           current, NULL)){
31        untrack_immpc_mutex(lock, current);
32    } else {
33
34        /*traditional process*/
35        immpc_mutex_slowlock(lock);
36    }
37 }
```

B. Maintainability

The IPC patch was implemented and maintained with the support of the Git (distributed version control tool [12]). The current version presents the following statistics of changes (in relation to branch rt/head of the kernel-tip kernel development tree):

```
include/linux/immpc_mutex.h |  162 +++++++
include/linux/sched.h       |   11 +-
kernel/Makefile             |    2 +
kernel/fork.c               |    5 +
kernel/immpc_mutex.c        | 1020 ++++++++++++++++
kernel/sched.c              |    5 +
6 files changed, 1202 insertions(+), 3 deletions(-)
create mode 100644 include/linux/immpc_mutex.h
create mode 100644 kernel/immpc_mutex.c
```

This indicates that two files were created (include/linux/immpc_mutex.h and kernel/immpc_mutex.c) and four others were altered (include/linux/sched.h, kernel/Makefile, kernel/fork.c kernel/sched.c), and the most changed file was include/linux/sched.h, and even then, minimally. This way, the IPC implementation is contained in the created files, and not in the previously existing files in the core kernel, changing only the strictly necessary for the operation of the mechanism.

V. Implementation Evaluation

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. It multiplexes the calls of three tasks in their correspondent critical sections. This device-driver provides both critical sections to run with IPC and with PI. This device-driver also reports the blocking times (time between request and access to a particular critical section) and the time spent on critical sections by the high-priority task.

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 I. 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 II.

Table I shows the intervals between activations expressed with a pseudorandomness, ie, with values uniformly distributed between minimum and maximum values. This randomness was included in tests to improve the distribution of results, because with fixed periods, patterns of arrivals were limited to a much more restricted set. Table I also presents the sizes of the critical sections of each task. Another information shown in Table I is the number of activations performed for each task. The duration of the test was defined by 1000 monitored activations (latency, response time, critical section time, lock time, etc) of the high-priority task.

The high-priority task has one of the highest priorities of the system. The other tasks were regarded as medium and low priorities although they also have absolute high priorities compared with normal tasks. All tasks have been configured with the scheduling policy SCHED_FIFO, which is one of the policies for real-time [13] available in Linux.

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

Even with the use of a SMP machine for testing, all tasks were set at only one CPU (CPU 0). The machine used has a Turion X2 TL-50 1.6 GHz dual-core processor, 2 GB of RAM and runs only the Linux kernel and basic daemons. Tests were conducted using both IPC and PI.

The time measurements were obtained using the TSC (time-stamp counter), which is a high resolution counter with low overhead access that is present on current x86 architecture implementations.

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 I.
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 II.
ACTIONS REALIZED BY TASKS

We measured three values regarding the high-priority task: the activation latency (the time between inserting a task on the ready queue and its effective execution), the blocking time (the time between requesting a resource and effectively gaining access to it) and response time (which is the time between inserting a task into the ready queue and the finish of its computation in a particular activation).

A. Results Using the PI mutex

With priority inheritance, the high-priority task had most of its activation latencies in the interval [$2x10^6$, $3x10^6$] nanoseconds as can be seen in the histogram of Figure 6. The portion of the activation latencies that are not in the interval (measured values out of the interval [$2x10^6$, $3x10^6$]), presents values linearly distributed around the vertical bar and with frequencies very close to zero, which makes them difficult to be seen in the histogram. The task finds the resource busy with a certain

frequency (as illustrated in Figure 7, waiting time for the resource), and it must perform context switch for propagation of its priority along the chain of locks.

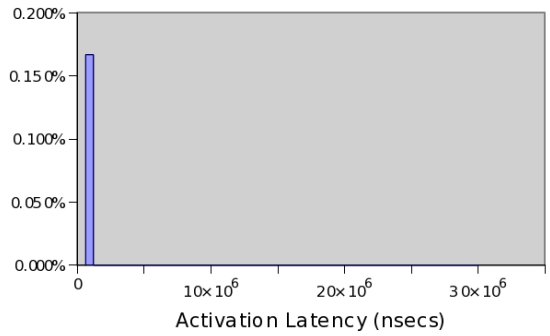

Figure 6. Histogram of activation latencies (high priority task using PI)

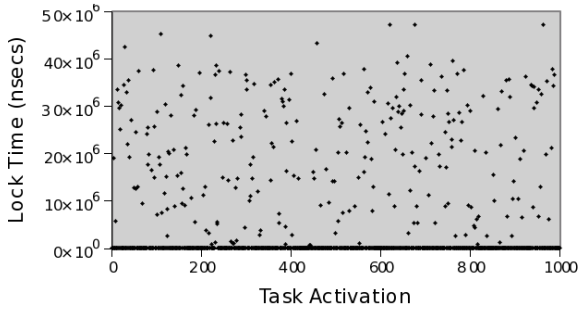

Figure 7. Graph of blocking time (high priority task using PI)

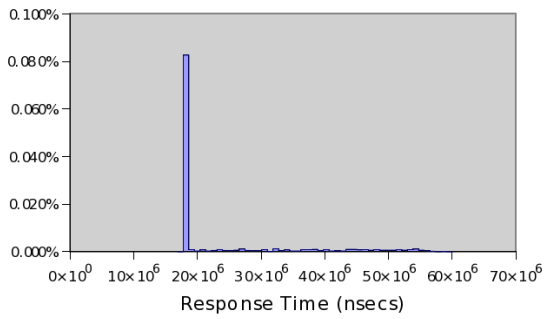

Figure 8. Histogram of response time (high priority task using PI)

The response time (Figure 8) was consistent with the blocking time sustained, with a maximum of nearly 3 times the size of the critical section, in accordance with what was expected. It can be seen in Table III the worst-case response time observed is 64,157,591 ns. The theoretical worst-case response time for this test would be, with a properly synchronized activation, 68 ms, being 17 ms from the critical section of task T0 plus 34 ms of task T1 and 17 ms of task T2. In this test, there is a good approximation of the theoretical limit.

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 III.
AVERAGE RESPONSE TIMES AND STANDARD DEVIATION

B. Results Using the IPC mutex

It can be noted in the histogram of Figure 9 that the high-priority task presented, with low frequency, varying values of latency of activation (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 (values are too small compared to the scale, so they appear close to zero.) In the response time histogram (Figure 11) it appears a tail (higher values, but with only a few occurrences) due to the activation latency.

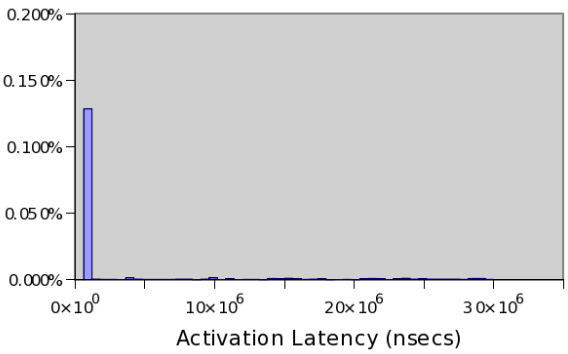

Figure 9. Histogram of activation latencies (high priority task using IPC)

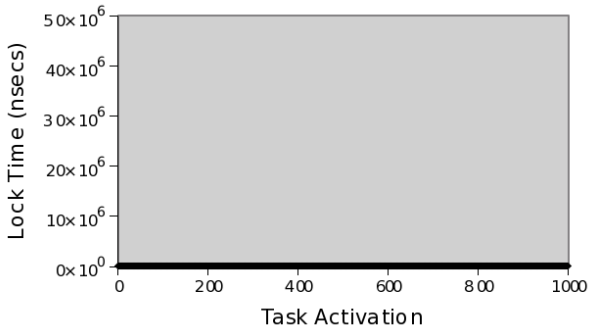

Figure 10. Graph of blocking time (high priority task using IPC)

In this test, it can be seen in Table III the worst-case response time observed is (maximum) 50,811,328 ns. Theoretic limit is 51 ms, ie, 17 ms from the critical section of task T0 plus 34 ms of task T1. Also in this test there is a good approximation of the theoretical limit.

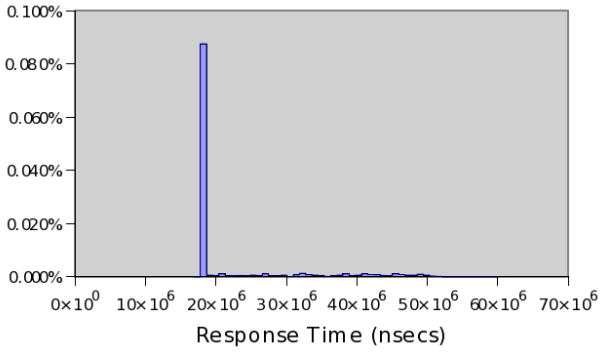

Figure 11. Histogram of response time (high priority task using IPC)

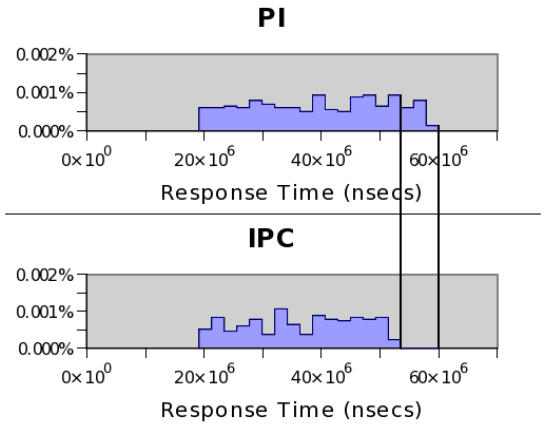

Figure 12. Histogram of the response time of high priority task

C. Comparison between PI and IPC

One can observe that, in general, protocol 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. For the protocol PI, the blocking time will appear with the primitive lock, and this time may be longer than with IPC. In IPC the blocking time appears before the activation time, and have a maximum length of a critical section (in the conditions of the test). The fact that the blocking time appears in the form of activation latency can be seen by comparing the graphs of Figure 6 and Figure 9, where the activation latency of IPC is noticeably greater than PI.

According to Table III, protocol IPC presented a standard deviation and average response time smaller than PI. Another important point in Table III is that the worst-case response time observed with IPC was almost a critical section smaller than with 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 histogram of the tail of the activations. 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 the average response time is smaller for the protocol IPC. This histogram also indicates in its final portion that the worst case, as it was also observed in Table III, has a difference of one critical section in favor of the IPC protocol. This difference in the worst case is high lighted in the figure by two vertical lines, where the distance between them is about the duration of a critical section. Table IV summarizes qualitatively the results.

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

TABLE IV.
SUMMARY OF RESULTS

VI. 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 (always with fasthpath) or IPC (both versions, with and without fastpath).

To evaluate the protocol implemented in terms of overhead imposed on the system, we used a set of test tasks as specified in Table V. The same table presents the tasks configurations.

In order to obtain an overhead estimative, it was created a measuring task with priority 51 (with policy SCHED_FIFO). This priority is above the default priority of threaded irq handlers [2] and softirqs [14] . 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). As described in the previous section, the machine used has a Turion X2 TL-50 1.6 GHz dual-core processor, 2 GB of RAM and runs only the Linux kernel and basic daemons.

The measurement task is activated at the same time of the real-time tasks and ends before they terminate. In each test iteration, the measurement task runs for 17 seconds. As shown in Table V, task T0' executes 40 activations, the others will run until the end of this task.

To obtain the overhead estimative, the measurement task executes a loop for 17 seconds. The overhead will be noticed by how much the measurement task actually uses the CPU, taking into account the execution of the set of tasks synchronized by IPC or PI (always pairing one execution of the IPC case with one of the PI case in order to minimize environment effects). The tests for IPC were repeated for both cases (with (*FP*) and without (*NOFP*) fasthpath). In the tests, PI always has its fastpath enabled.

Task	T0'	T1'	T2'	T3'	T4'	T5'	T6'
Priority	70	65	64	63	62	61	60
Activation interval (ms)	rand in [500,1000]	rand in [100,200]	rand in [100,200]	rand in [100,200]	rand in [90,180]	rand in [90,180]	rand in [90,180]
Resource	R1	R1, R2	R1, R2	R1, R2	R2	R2	R2
Critical section size	aprox. 17 ms	aprox. 2x17 ms	aprox. 2x17 ms	aprox. 2x17 ms	aprox. 17 ms	aprox. 17 ms	aprox. 17 ms
Number of activ.	40	T0' dep	T0' dep	T0' dep	T0' dep	T0' dep	T0' dep

TABLE V.
ACTIONS REALIZED BY TASKS

The values of the CPU time (in nanoseconds) utilized by the measurement task are presented in Table VI, which was sorted to facilitate visual comparison. Table VI also presents the basic statistical data related to the samples.

In order 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 CPU time available to the measurement task under the workload synchronized by IPC and by PI are equal (for both cases of IPC, with and without fastpath), then we have two independent hypothesis:

1) $H_0 : \mu_{PI} = \mu_{IPC_{NOFP}}$
2) $H_{0'} : \mu_{PI} = \mu_{IPC_{FP}}$

The parameters presented in Table VII, which provides the data necessary for the hypothesis test, were obtained from the data showed in Table VI plus the information of the number of samples (n = *40*).

	Without fastpath (IPC_{NOFP})	With fastpath (IPC_{FP})
$Sa^2_{IPC,PI}$	1,621,289,369,039	7,137,183,653,740
n	40	40
α	0.1%	0.1%
$d.f.$	80	80
t	-12.51	1.64

TABLE VII.
STUDENT'S T-TEST DATA

A. Analysis of the Overhead

The test rejects H_0 with a significance level of 0.1 %, because the data produced the value of t = -12.51, which does not belong to the region of acceptance (t-Student distribution). At significance level (α) of 0.1%, the collected data indicate a difference between PI and IPC$_{NOFP}$. There is a probability smaller than 0.1% that the differences observed in the presented data are from casual factors of the system only.

However, for the case of IPC$_{FP}$ (with fastpath), the value of t = 1.64 indicates an equivalence between the overhead caused by IPC$_{FP}$ and PI. That is, it is noted that for architectures that support atomic instructions cmpxchg the fastpath can really make a difference in terms of overhead.

In IPC$_{NOFP}$ (without fastpath), there is always a need of priority verification and adjustment. Another point is that if a task with priority lower than 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 priority propagation algorithm (PI) will run, but there will almost always priority adjustments (IPC$_{NOFP}$), except for the task that defines the priority ceiling of the resource.

In the case of IPC$_{FP}$ (with fastpath), a priority adjustment will happen only if there is imminence of preemption by another task. Thus, there is a chance of not occurring the priority adjustment. Another important point that helps to explain the results is that the IPC with fasthpath executes a few instructions more than the PI fastpath. However, the PI protocol leads to more context switches, resulting in an equivalent overhead.

VII. CONCLUSIONS

Task synchronization is fundamental in multitasking and/or multithread systems, specially in real-time systems. These mechanisms must protect against race conditions and prevent the appearance of uncontrolled priority inversions, which could cause the missing of deadlines, leading real-time applications to present incorrect and possibly harmful behavior. In this context, it was proposed the Immediate Priority Ceiling as an alternative to the protocol implemented in the real-time Linux branch. The proposed implementation was shown with two variations, the first non-optimized (using priority adjustments every lock/unlock operation) and the second with fastpath (using the concept of Priority Change Postponement and atomic instructions).

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

In terms of average response time, the two solutions were similar, but IPC still showed lower average response time probably due to the latency of activation being less

Statistics	IPC_{FP}	PI		IPC_{NOFP}	PI
	407,035,114	406,435,450		401,424,676	405,553,163
	407,285,636	407,765,565		401,820,286	406,458,117
	407,373,521	407,835,019		401,948,422	406,658,942
	407,819,905	407,876,696		402,597,568	406,856,751
	408,377,891	408,048,768		403,542,887	407,004,448
	408,381,506	408,101,843		403,547,748	407,238,676
	408,406,446	408,219,724		403,874,521	407,350,956
	408,417,061	408,220,571		404,566,739	407,427,672
	408,711,409	408,433,649		404,686,516	407,513,400
	408,763,366	408,446,200		404,707,405	407,577,573
	408,801,061	408,549,755		404,757,367	407,596,261
	408,883,156	408,575,046		404,808,759	407,706,637
	408,949,145	408,575,106		404,826,216	407,785,273
	409,148,907	408,606,476		404,827,682	408,013,960
	409,202,266	408,783,361		404,970,791	408,315,982
	409,443,672	408,785,943		404,978,816	408,495,828
	409,518,531	408,790,334		405,086,464	408,602,577
	409,658,142	408,862,429		405,122,964	408,604,205
	409,662,722	408,987,951		405,165,352	408,702,488
	409,679,490	409,156,013		405,169,874	408,789,189
	409,721,677	409,171,710		405,245,988	408,846,701
	409,761,649	409,212,383		405,409,364	408,945,776
	409,772,169	409,230,027		405,441,947	408,956,142
	409,837,278	409,381,642		405,505,815	408,961,259
	409,899,266	409,399,327		405,560,936	408,984,062
	410,158,821	409,409,126		405,626,303	409,024,543
	410,207,755	409,424,264		405,750,834	409,159,619
	410,419,774	409,502,679		405,770,308	409,395,789
	410,477,452	409,626,412		405,784,920	409,459,483
	410,492,265	409,710,682		405,917,678	409,465,209
	410,496,194	409,833,770		405,974,737	409,513,038
	410,546,730	409,866,286		405,994,401	409,593,505
	410,661,307	409,872,273		406,031,717	409,602,110
	410,679,530	409,873,581		406,043,948	409,977,571
	410,715,126	410,066,899		406,253,972	410,018,274
	410,853,474	410,129,241		406,322,758	410,079,834
	410,857,619	410,179,304		406,406,613	410,284,319
	411,127,684	410,226,307		406,618,569	410,312,717
	411,289,886	410,617,801		406,661,586	410,467,174
	431,575,650	412,029,972		406,945,074	410,907,269
Average:	410,076,756	409,095,489		405,042,463	408,605,162
Variance:	13,330,393,417,558	943,973,889,922		1,706,936,654,588	1,535,642,083,490
Minimum:	408,711,409	408,433,649		404,686,516	407,513,400
Maximum:	431,575,650	412,029,972		406,945,074	410,907,269

TABLE VI.
CPU TIME AVAILABLE TO THE MEASUREMENT TASK AND RELATED STATISTICS

than the waiting time of PI. Another point in favor of IPC protocol appears when we compare the difference in the worst-case response time observed in the tests since the IPC was about a critical section faster than PI, as can be seen in Table III. Protocol PI has a response time that can vary depending on the resource sharing and sequences of activation patterns, which does not occur with IPC. IPC blocking time will always be at most one critical section.

Although blocking/response times are smaller with IPC, tests show that the overhead of the non-optimized version of IPC 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, for example. However, the optimized version of IPC showed equivalent overhead to the PI implementation. This equivalence arises due to the fact that IPC executes more instructions in lock/unlock primitives (however, considerably less than in non-optimized version of IPC). This is compensated by the fact that protocol PI performs more context switches and chained priority adjustments.

ACKNOWLEDGMENTS

To CAPES and CNPq for financial suport.

REFERENCES

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

[2] S. Rostedt and D. Hart, "Internals of the RT Patch," vol. 2007, 2007.

[3] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[4] B. Lampson and D. Redell, "Experience with processes and monitors in mesa," *Communications of the ACM*, vol. 23, no. 2, pp. 105–117, 1980.

[5] A. Carminati, R. de Oliveira, L. Friedrich, and R. Lange, "Implementation and evaluation of the synchronization protocol immediate priority ceiling in preempt-rt linux," in *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2010, p. 82.
[6] V. Yodaiken, "Against priority inheritance," *FSMLABS Technical Paper*, 2003, available at `http://yodaiken.com/papers/inherit.pdf`.
[7] A. Burns and A. Wellings, *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Addison Wesley, 2001.
[8] T. Baker, "A stack-based resource allocation policy for realtime processes," in *IEEE Real-Time Systems Symposium*, vol. 270, 1990.
[9] M. Harbour and J. Palencia, "Response time analysis for tasks scheduled under edf within fixed priorities," in *RTSS 2003: 24th IEEE International Real-Time Systems Symposium: 3-5 December, 2003, Cancun, Mexico*. IEEE Computer Society, 2003, p. 200.
[10] S. Rostedt, *RT Mutex Design - Linux Kernel Documentation*, 2006, http://www.kernel.org/doc/Documentation/rt-mutex-design.txt.
[11] I. Molnar, "Preempt-rt development branch," 2010, git://git.kernel.org/pub/scm/linux/kernel/git/tip/linux-2.6-tip.git.
[12] L. Torvalds and J. Hamano, "GIT-fast version control system," 2005.
[13] C. S. IEEE, Ed., *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.
[14] R. Love, *Linux Kernel Development (Novell Press)*. Novell Press, 2005.

Andreu Carminati is graduated in Computer Science from Federal University of Santa Catarina (2010). He is currently a M.S. Student 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 PhD in Engineering from the Federal University of Santa Catarina. He is a member of the Brazilian Computing Society (SBC).