Tasks Schedule Analysis in RTAI/Linux-GPL

Claudio Aciti and Nelson Acosta

INTIA - Depto de Computación y Sistemas - Facultad de Ciencias Exactas
Universidad Nacional del Centro de la Provincia de Buenos Aires
(7000) - Tandil - Argentina
`http://www.exa.unicen.edu.ar/~inca`

Abstract. Preemptive real-time operating systems allow the kernel to stop a running task to execute other task with higher priority. The tasks compete for the resources producing anomalies such as starvation, deadlock and priority inversion that can reduce the perceived performance of the system. The scheduler is in charge of the synchronization when priorities are considered and of avoiding these anomalies to occur.
Among the FOSS[1] community, the RTAI[2] real-time extension for the Linux kernel is one of the most applied and with major projection within the automatic control area. To synchronize priority tasks, RTAI has implemented the Priority Ceiling and the Priority Inheritance mechanisms. The main goal of this work is analyze the functioning of these mechanisms for different situations, in RTAI/Linux-GPL.

Key words: Real-Time Scheduling; RTAI; Priority Inversion; Priority Inheritance; Priority Ceiling

1 Introduction

By using preemptive real-time operating systems, the synchronization of tasks is difficult because the tasks compete for the access to the resources and this situation degrades the performance of the system. The resources, external hardware devices, computer components and software elements (mutex, queues, semaphores, etc), have critical sections to be used in a non interruptive and exclusive way by one task at a time. Though it is true that the execution of a task can be stoped for the system to do context switch and begin executing other task. This is the reason why when a task requires a resource it must request for it and then block it in order to be able to use it in a exclusive and non interruptive way; when finish using it the task must unblock the resource to release it. If the request for a resource fails, the requesting action is the one to be blocked waiting for the nedeed resource to be released. This situation causes that, in real time systems, the critical tasks have higher priority that the rest of the tasks [1] [2] [3].

[1] FOSS: Free Open Source Software.
[2] RTAI: Real Time Aplicattion Interfaces.

The use of priorities to schedule tasks can make the assignment of resources not to be fair and can cause long delays, and even the starvation of some of them. To implement a real time functionality requires a careful design of the scheduling and all the issues related to the operating system. First, the tasks must have a priority assigned (the more critical the task is the higher the priority it has) which should not degrade. Second, the delay of dispatching must be short for the a task to begin running as soon as possible once it is ready[4].

This kind of problems affects the normal functioning of control systems, causing a gradual degradation of the performance or even a temporary or total inactivity of the system. The better known example happened in 1997, the Mars Rovers Pathfinder landed on Mars in order to execute tests for researching. After a few days of right operation, the system began experiencing repeated and unexpected system resets, resulting in losses of data and the early abort of the mission. It was determined that the anomaly was due to a priority inversion problem [5] [6].

Among the FOSS community, the RTAI real-time extension for the Linux kernel is one of the most applied and with major projection within the automatic control area. To do the scheduling of priority_based preemption tasks, RTAI has implemented the Priority Ceiling and the Priority Inheritance mechanisms. The RTAI allows the tasks scheduler to preempt a running task. The main goal of this paper is to analyze the performance of the scheduler for this operating system, in different cases and using several mechanisms for controlling the priority inversion[7].

Next, in the 2^{nd}section, the RTAI scheduler is presented. In the 3^{rd} section the priority inversion problem is revised. The work carried out and the results obtained are detailed in the 4^{th} and the 5^{th} sections. In the 6^{th} section the conclusions are given and in the 7^{th} section the future works are presented. Finally, the used bibliography is detailed.

2 RTAI/Linux-GPL Tasks Scheduler

By using the RTAI scheduler a task can accomplish its job with hard real time constraints and be able to execute in a deterministic way, which means that the task can be executed exactly as it was designed and it is not restricted by the general scheduler of Linux. In RTAI, the real time tasks have priorities with values among `0` (highest priority) and `0x3fffffff` (lowest priority); the scheduler takes them as always active, so a real-time task inhibits (prevents) the execution of a non real-time one.

In the `rt_task_struct` structure, all the information related to a real time task is kept. Within that structure, the scheduling aspects, identifiers, relation with other tasks, memory used by the task, open files, etc. can be specified. The most important fields the `rt_task_struct` structure contains are: the task state (`state`), whether the task is running or not (`running`), the CPU where the task is running (`runnable_on_cpus`), the current priority of the task (`priority`), the policy used when the task arrives to the scheduler (FIFO by default, Round

Robin, Monotonic Scheduling , Shortest Task Scheduling), the base priority of the task (`base_priority`), the time when the task has to resume execution (`resume_time`) and the time in which the task must yield the CPU (`yield_time`).

The scheduler points to the active task, and from this one, there has the pointers to the previous and next tasks. The tasks are sorted by priority and state. The `rt_queue` structure, links the `rt_task_struct` for all the running tasks. Similarly, for common tasks, there exist the `runqueue` structure which in combination with the `rt_queue` structure, sorts all the tasks in a ready state(Fig. 1).

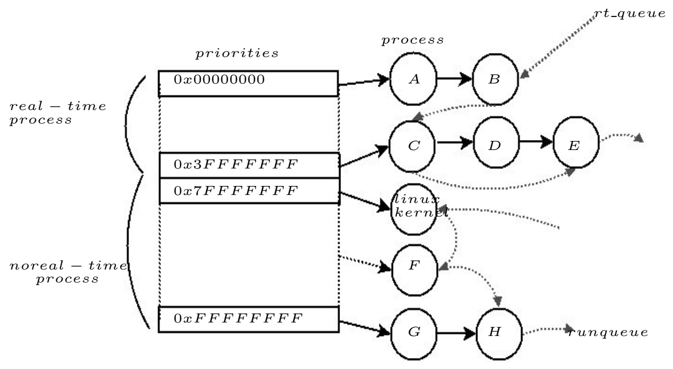


Fig. 1. Queues of real-time and non real-time ready processes

The tasks scheduler executes in one of two situations: 1) if a timer interruption occurs (`rt_timer_handler()` is executed for this case); 2) if there exists a scheduled system call (`rt_schedule()` is executed ; the system call can be due to a task being inserted in the ready queue, the current task is put to sleep or a task is blocked. `rt_timer_handler()` and `rt_schedule()` are similar, only that the first has to deal with timer and interrupts to be transmitted to the Linux kernel.

The main goal of the scheduler is to replace the running task for the next one in the list of ready tasks. If there is no task with higher priority than the running one, no change is made. In the case of having to change the task that is executing, the scheduler has to do a context switch. The function to call for doing the switching is `rt_switchto()`, responsible for saving all the associated information related to the task being stopped in order to be able to resume.

2.1 Threads

The RTAI pthread module implements the POSIX 1003.1c. standard. It provides dynamic creation and termination of threads, so the amount of threads has not to be known till run-time. The POSIX threads use attribute objects to represent the threads properties. Features such as stack size and scheduling policy are set for each of the threads by means of its attributes. A thread has a type identifier `pthread_t`, a stack, an execution priority and an initial address for executing. A POSIX thread is created dynamically using the `pthread_create` function which creates a thread and places it on the tail of the ready queue. During its lifetime a thread can be in any of these states: Ready to execute, running, blocked or terminated.

In most of cases the applications that use pthreads have the responsibility of sharing data among threads and guarantee that certain actions are performed in a coherent sequence. To accomplish this the activity of the pthreads needs to be synchronized when accessing to the shared data in order to avoid wrong functioning and non desired effects. In RTAI, the synchronization of functions for the applications is available through mutex and condition variables.

Mutual exclusion is the most common synchronization method for multiple threads to share a resource. A mutex is used to provide mutual exclusion locking capability for the pthreads to control the access to code sections and data that require atomic access. In this conditions only one thread can hold the lock and, hence, use the resources the mutex is protecting. A mutex can also be used to guarantee an exclusive entry to code sections o routines known as critical sections.

One of the main differences between a mutex and a condition variable is that a mutex allows the pthreads to be synchronized controlling the access to the data, and a condition variable permits the threads within a data value to be synchronized. A condition variable provides a method for transmitting information about the state of the shared data. E.g., this information could be a counter reaching certain value, a queue becoming empty.

The mutexes are defined using three alternative synchronization protocols:

- NO_PRIO_INHERIT: The thread priority does not depend on its owner relationship over any mutex (a thread is the owner of the mutex that it blocks).
- PRIO_INHERIT: The thread owner of a mutex inherits the priorities of the threads waiting for the mutex. This is the priority basic inheritance protocol.
- PRIO_PROTECT: When a thread acquire a mutex, it inherits the priority called ceiling priority of the mutex, defined usually as the priority of the task with higher priority able to block that mutex.

The non restricted priority inversion can be avoided by using different algorithms, hence getting high level utilization of systems with hard real time requirements.

3 Priority Inversion

Priority inversion is a problem that occurs when competing for resources. On the other hand, a task scheduler is responsible for solving the problem once this has happened, trying to reduce the amount of context switches and the latency of the tasks with higher priority.

A low priority task can stop the execution of a high priority task because of competing for the same resource. Also, a medium priority task can stop the execution of a high priority task without even compete for the same resource. These anomalies make the high priority task to be affected both by the increase of the total execution time and the latency till it enters its critical section. For the low priority task the increase of time is noticed running within its critical section (Fig. 2.a).

RTAI implements two mechanisms to control the priority inversion: Priority Inheritance and Priority Ceiling. When using Priority Inheritance a low priority task that holds a shared resource that is required by a high priority task "inherits" the priority of this high priority task from the moment of the request. A low priority task can stop the execution of a high priority task by holding a shared resource, but to inherit the highest priority keeps a medium priority task from pre-empting the low priority task; this way the medium priority task can not stop the execution of a high priority task without even compete for the same resource. In fact, the medium priority can not request for CPU use (Fig. 2.b).

The Priority Ceiling mechanism, also implemented in RTAI, works in a different way. Before execution, the scheduler must know which resources are to be more used and the tasks that will use them. In that way, a table for mapping resources-tasks is created before run-time to be looked up during the execution. In this case, a low priority task can stop the execution of a high priority one that competes for the same resource, but inheriting immediately the highest priority that the resource can have avoids the situation of a medium priority task stopping the execution of a high priority task. The medium priority task has no right even to request for CPU use, thus avoiding loss of time deciding whether it corresponds to preempt the active task or not. One of the anomalies of this mechanism is that the execution of the medium priority tasks can be delayed because of tasks of lower priority (Fig. 2.c).

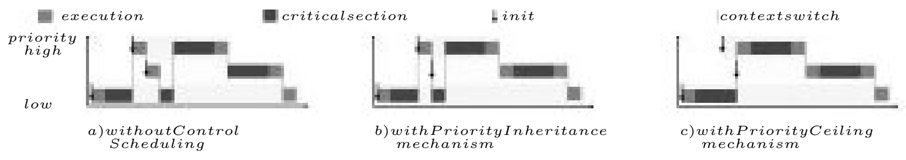

execution
criticalsection
init
contextswitch
priority high
low
a) withoutControl Scheduling
b) withPriorityInheritance mechanism
c) withPriorityCeiling mechanism

Fig. 2. Priority inversion mechanism

4 Cases of Study

The object of study is to analyze the performance of the tasks scheduler of the RTAI/Linux-GPL, at the moment of assigning the resources of the computer to the high priority tasks, considering all the possible load situations and applying different scheduling techniques.

4.1 Measure Strategy

The analysis of the scheduler performance is designed based on the use of proprietary time primitives of the RTAI operating system. Through the use of these primitives the exact time since the computer was turned on can be measured; also by means of some mathematical calculations that value can be converted to microseconds. It is possible to measure the time spent by some code sections by introducing those primitives at specific points.

The variables to be analyzed are:

- the latency of a real time task: The time a system takes to respond to a request for a process to begin operation.
- the blocking time on a critical region: The period time that the task can be prevented from execution within a critical region.
- total execution time: Time since an event occurs till the task is completely processed.

The following pseudocode is an example of strategy to be used:

```
example(t_process_init){

  /*Process init*/
  latency= rt_get_cpu_time_ns() - t_process_init;

         INIT SECTION

  /*critical section init*/
  t_critical_section_init= rt_get_cpu_time_ns();

         CRITICAL SECTION

  /*critical section exit*/
  t_critical_section_use= t_critical_section_init - rt_get_cpu_time_ns();

         FINAL SECTION

  t_process= t_process_init - rt_get_cpu_time_ns();

}
```

4.2 Samples

The studied population is the real time operating system, RTAI/Linux-GPL. The RTAI version is 2.0 and runs over a Linux kernel 2.6.24 or higher, with a 32 bit x86 architecture. This operating system is used in hard real time conditions. The programming language used is native C compiled with gcc 4.1. The measures were taken with software installed in the same computer used for testing through time primitives. The accuracy of these measures are tied to the performance of the computer hardware which is the non controlled variable.

The following are the taken samples:

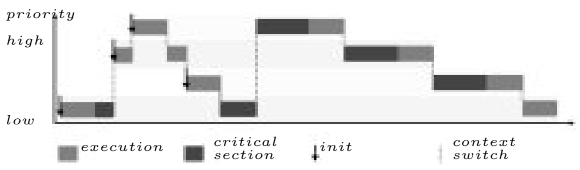
- A real time task of highest priority is scheduled, within an environment where no other highest priority task is requesting for resources.
- A real time task with a mutex and no priority inversion mechanism. This test is performed with different number of threads, both real-time and non real-time trying to block the same critical section.
- A real time task with a mutex and a priority inversion mechanism, the Priority Inheritance method.
- A real time task with a mutex and a priority inversion mechanism, the Priority Ceiling method.

5 General Results

By analyzing the taken samples, the generalities listed below are inferred. The studied variables are: total execution time T, latency L and blocking time *lock_time*; the tasks priorities: highest priority task MP, medium priority tasks IP, and lowest priority tasks mp; CPU use times: initial time i and final time use f; context switch CS; and for blocking, the variables are: time of blocking attempt tl, and blocking time C.

5.1 Samples without Control Scheduling

If the scheduler does not uses a control mechanism, when a task with higher priority than the running task arrives, the scheduler yields the CPU (Fig. 3).



$$L_{hp} = 0$$

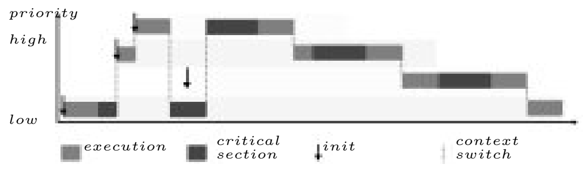
$$T_{hp} = L_{hp} + CS + ti_{hp} + tl_{mp} + \\ + T_{ip} + le + CS + C_{ip} + tf_{ip}$$

$$C_{lp} = t_{lp} + t_{ip} + C_{lp}$$

Fig. 3. Without Control Scheduling

5.2 Samples with Priority Inheritance mechanism

If the scheduler uses the Priority Inheritance control mechanism, when a task with higher priority than the running task blocking a resource arrives, the scheduler yields the CPU use to the arriving task. At the moment when the current running task requests for the shared resource, the blocking task automatically inherits the priority of the higher priority task (Fig. 4).



$$L_{hp} = 0$$

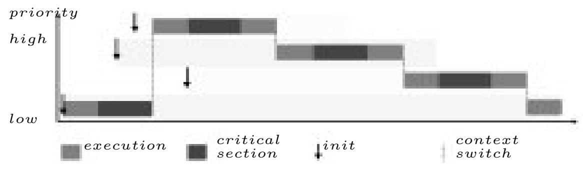
$$T_{hp} = L_{hp} + CS + ti_{hp} + tl_{mp} + \\ + T_{ip} + le + CS + C_{ip} + tf_{ip}$$

$$C_{lp} = t_{lp} + t_{ip} + C_{lp}$$

Fig. 4. With Priority Inheritance mechanism

5.3 Samples with Priority Ceiling mechanism

If the scheduler uses the Priority Ceiling control mechanism, when a task with higher priority than the running task blocking a resource arrives, it has to wait for the lower priority task to finish using the shared resource and then start executing (Fig. 5).



$$L_{hp} = 0$$

$$T_{hp} = L_{hp} + CS + ti_{hp} + tl_{mp} + \\ + T_{ip} + le + CS + C_{ip} + tf_{ip}$$

$$C_{lp} = t_{lp} + t_{ip} + C_{lp}$$

Fig. 5. With Priority Ceiling mechanism

5.4 Metrics

The more relevant metrics obtained are detailed next. In table 1, the total time for context switches for the highest priority task is shown. In table 2, the quantity of blocking attempt during the execution time for the highest priority task is shown. In table 3, the total time for context switches for the highest priority task is shown. In table 4, it is shown the quantity of blocking attempt during the blocking time for the lowest priority task when this task blocks a resource. Finally, in table 5, the latency for the highest priority task for this to begin executing.

	medium priority task					
mechanism	1	2	3	4	5	6
none	16 μS	16 μS	8 μS	16 μS	20 μS	16 μS
inheritance	8 μS	8 μS	0 μS	0 μS	0 μS	0 μS
ceiling	0 μS	0 μS	0 μS	0 μS	0 μS	0 μS

Table 1. Context switches during the execution time for the highest priority task

mechanism	medium priority task					
	1	2	3	4	5	6
none	2	3	1	2	2	2
inheritance	1	1	1	1	1	1
ceiling	1	1	1	1	1	1

Table 2. Blocking attempt during the execution time for the highest priority task

mechanism	medium priority task					
	1	2	3	4	5	6
none	16 μS	20 μS	28 μS	36 μS	40 μS	40 μS
inheritance	8 μS	16 μS	12 μS	12 μS	12 μS	12 μS
ceiling	0 μS	0 μS	0 μS	0 μS	0 μS	0 μS

Table 3. Context switches during the blocking time for the lowest priority task

mechanism	medium priority task					
	1	2	3	4	5	6
none	2	3	4	5	5	5
inheritance	1	1	1	1	1	1
ceiling	0	0	0	0	0	0

Table 4. Blocking attempt during the blocking time for the lowest priority task

mechanism	medium priority task					
	1	2	3	4	5	6
none	0 KHz	0 KHz	0 KHz	0 KHz	0 KHz	0 KHz
inheritance	0 KHz	0 KHz	0 KHz	0 KHz	0 KHz	0 KHz
ceiling	400 KHz	300 KHz	600 KHz	600 KHz	400 KHz	600 KHz

Table 5. Latency for the highest priority task

6 Conclusions

The functioning of the scheduler without priority inversion control has a very poor performance compared with one that implements a control mechanism. Though, for this case, the tasks latency is almost zero, the total time of a high priority task will be excessively high, causing delay on release of some blocked resource. The lack of a control mechanism makes the scheduler to do more context switches increasing the execution time for all the tasks. The blocking time of a resource by a low priority task significatively affects the performance of the system; every higher priority task that requests for CPU takes the active control.

The latency of a highest priority task, when using an Inheritance mechanism, is almost null; the total execution time makes the performance of the system not to be affected. Something similar happens with the blocking time caused by a low priority task, since once a higher priority task arrives, the low priority one inherits the higher priority and is able to finish executing. The delay arises from all the intermediate tasks that may have to be executed till the highest priority task arrives.

The Priority Ceiling protocol has good performance when the load of tasks for the scheduler is high. This is due to the fact that the tasks sharing a resource acquire the priority of that resource, hence avoiding blocking delays and context switches. With this protocol, the tasks that use other resource can be repeatedly delayed by lower priority tasks, without any possibility of reverting this situation. In this case, the requirements of the system must be taken into account; i.e., whether it is better to handle the highest priority task without taking care of the intermediate tasks or if these latter are also rather important.

The problem with this mechanism is how to determine the value for a blocking ceiling in advance; this is not easy, since the compiler needs to have access to the whole code, to know which tasks uses which resources. There exists a table with a mapping among resources and its highest priority; the creation of this table causes an initial delay but then the performance of the mechanism is superior compared with the rest of the mechanisms.

7 Future Works

As future work it is suggested the implementation of other priority inversion control mechanisms over the RTAI/Linux-GPL operating system. Examples of these mechanisms are BandWidth Inheritance, Recursive Priority Inheritance, Immediate Priority Ceiling or some other one.

Acknowledge

To the Eng. Carolina Tommasi, for the discussions on the previous versions, and for her collaboration with the final revision of the article

References

1. Buttazzo, G. (2006). Why real-time computing? Proccedings of the 2006 ANIPLA. International Congress on Methodologies for Emerging Techonologies in Automation.
2. Gai, P., Abeni, L., Giorgi, M., and G., B. (2001). A new kernel approach for modular real-time systems development.
3. Barabanov, M. and Yodaiken, V. (1997). Introducing real-time linux. Linux Journal, 1997:5.
4. Silverchatz, A. and Galvin, P. B. (1999). Sistemas Operativos. Prentice Hall.
5. Jeffay, K., Smith, F., Moorthy, A., and Anderson, J. (1998). Proportional share scheduling of operating system services for real-time applications. Proceedings of the IEEE Real-Time Systems Symposium, pages 480-491.
6. Jones, M., Rosu, D., and Rosu, M. (1997). Cpu reservations and time constraints: Efcient, predictable scheduling of independent activities. In Proceedings of the 16th ACM Symposium on Operating Systems Principles. Francia, pages 198-211.
7. Real Time Applicattion Interfaces, `http://www.rtai.org`