Implementing Real-Time services in MINIX

Gabriel A. Wainer
Departamento de Computación. Facultad de Ciencias Exactas y Nat urales.
Universidad de Buenos Aires.
Pabellón I. Ciudad Universitaria.
Argentina.
gabrielw@dc.uba.ar

Abstract

In this work we present the results of a project devoted to provide programming facilities to develop hard real-time software. We have used MINIX operating system as a tool for our experience. We allow the programmer to define timing constraints for the tasks, letting to the Operating System the work of running these tasks in a timely fashion. In this way, we can improve productivity, security and costs in the system development cycle.

1. Introduction

In these days the existing number of Hard Real-Time systems is growing rapidly. These systems must control events occurring in the real world, and each task in the system must respond to these events within a maximum time (the task deadline).

At present, most of the general purpose operating systems does not provide support to develop this kind of systems. They usually entitle concurrent programing, task synchronization and communication, resource sharing and other services, but do not include primitives to define timing constraints. Due to these reasons, real-time designers still develop real-time applications using "ad-hoc" techniques.

With this sight, we started a project devoted to provide facilities to ease the work of a real-time programmer through services in the Operating System. To avoid developing software from scratch, we decided to extend the services provided by MINIX Operating System [Tan87]. We selected using MINIX motivated by many factors. First of all, we wanted to use our results with academic purposes. Hence, the availability of the hardware and software in our Department was a key issue. Another decisive point was our previous experience in the subject.

We devote the rest of this work to present the sketch of our project, the changes we made, and some experimental results we got.

2. Real-Time scheduling

The real-time scheduling theory relates with the way of meeting the timing constraints of the tasks in a real-time environment. We can recognize two kinds of tasks: ***periodic*** and ***aperiodic*** (***sporadic***). The periodic tasks must run repeatedly, and within fixed times. The aperiodic tasks run sporadically, and only once when we invoke them. To avoid unpredictable behavior, we also must schedule the distribution of the system resources. To attain to these goals we must carefully synchronize the tasks in the system. Other issues to consider include the precedence constraints between tasks, and the criticality of each task.

As we can see, real-time scheduling is very complex. We must execute the system tasks within their timing constraints, responding to the high critical tasks first. We must consider the precedence dependencies, and also should make sure that the tasks will have the necessary resources in the moment they need them. The scene is even more intricate in distributed systems.

The scheduling algorithms of many operating systems used at present for real-time processing are simple extensions of those used in Time-Sharing systems. Most of them use priority algorithms, letting the programmer to adjust the task priorities to fulfill the timing constraints. Moreover, the designer must map many conflicting considerations (timing constraints, criticality, task dependencies and others) in only one number: the task priority. The only way to guarantee predictable behavior is through exhaustive testing.

The use of these Operating Systems leads to high development costs and lack of flexibility. Any changes made to the system implies a new round of exhaustive testing. The goal of a real-time scheduler is to decide if there is a schedule to meet the timing constraints of a task set. In that case, we will say that such set is schedulable. We also will say that each task in the set has a predictable behavior, or that we can guarantee the task timing constraints. A real-time scheduling algorithm must insure [Sha90]:

1. ***Predictable*** response time of tasks.
2. High degree of resource employment (***schedulability***), while keeping predictable responses.
3. ***Stability*** under transient overloads. In these cases, the scheduler must guarantee the response time of a selected group of critical tasks.

Most of the real-time schedulers also use priorities' schemes (static or dynamic). Instead, the dynamic approach allows the task priority to change during the program execution. It is also important to the scheduler to be preemptive. A non preemptive scheduler could lead to run a low priority task while a high priority ta sk is waiting.

When we apply a static scheduler we assume prior knowledge of the task constrains for the controlled system. These schedulers have little overhead, but are very rigid. Sometimes we use them to make off-line scheduling. Instead, dynamic schedulers must work on-line. This approach is more flexible because in every moment the tasks have a priority reflecting their properties. In this kind of system, the overhead can be significant. Every time we call the scheduler, it must compute task priorities.

We decided to implement real-time scheduling algorithms for centralized systems and analyze the properties stated earlier. We choose two traditional ones: the Rate-Monotonic and the Deadline-Driven algorithms ([Liu73], [Che88], [Sha90]). Both of them can guarantee predictable execution of a task set if the processor load is below a given bound. In that case the designer can run the tasks in the set without consider the activities performed by every task in each moment. This fact eases the work involved in the development, testing and maintenance process, reducing the costs involved.

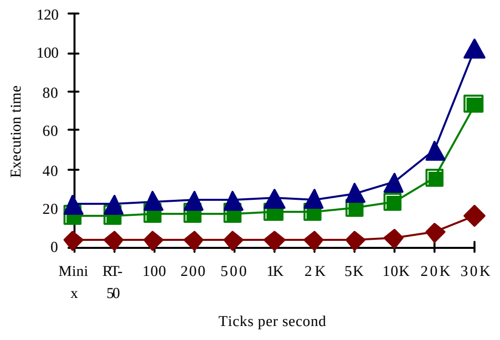
The Rate-Monotonic algorithm schedule periodic independent tasks. Each task has a fixed priority, inversely proportional to its execution period. Instead, the Deadline-driven is a dynamic priorities' algorithm. The tasks with earlier deadlines run before those with later deadlines. Both algorithms are preemptive.

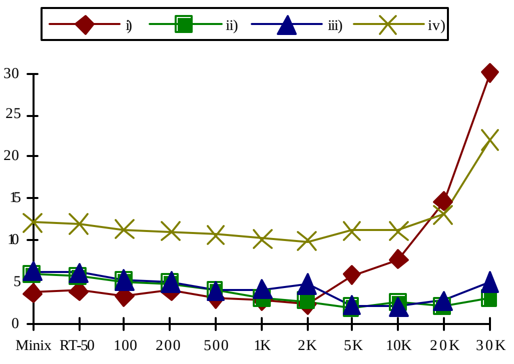
In the following section we will explain the changes we made to the MINIX kernel to provide real-time services.

3. MINIX kernel changes.

MINIX uses a Round-Robin user task's scheduler (100 millisecond time-out), that we have changed to allow the execution of periodic and aperiodic real-time tasks.

The sporadic tasks will have only one instance, with a certain deadline to meet. Instead, to set up scheduling points for periodic tasks, the clock driver must decide which instance start in each clock tick. As the periodic tasks start under the timer control we included a set of system calls to change its activation rate. The granularity changes can lead to more overhead, as we can see in figure 1. The overhead does not alter the behavior of the system up to 10000 ticks per second, allowing us to handle a precision of 100 microseconds (instead of the original 20 milliseconds). We also added a simple call allowing to measure time in clock ticks. In this way, we can program, in a very simple way, tasks with tighter timing constraints. Originally, the MINIX system calls have a precision measured in seconds.





Cpu-bound task sets. Ticks per second/task completion time (seconds).

Task mix. i) CPU-bound; ii)-iii) Interactive; iv) mixed

Figure 1. Analysis of overloads produced by reductions in the clock granularity

We also changed the main structure of the task scheduler to suit our task model. We have used a preemptive multiqueue scheduler with three ready queues in total. We can see the scheduler sketch in Figure 2. The first queue keeps real-time instances ordered using real-time scheduling algorithms. The second level queue devotes to schedule interactive tasks using Round-Robin (100 ms. quantum) and Priorities (fixed or variable) algorithms. We also added a third queue to run CPU-bound tasks in background (Round-Robin; quantum=200 ms). The tasks exceeding the Round-Robin quantum of the Interactive queue goes to this queue.

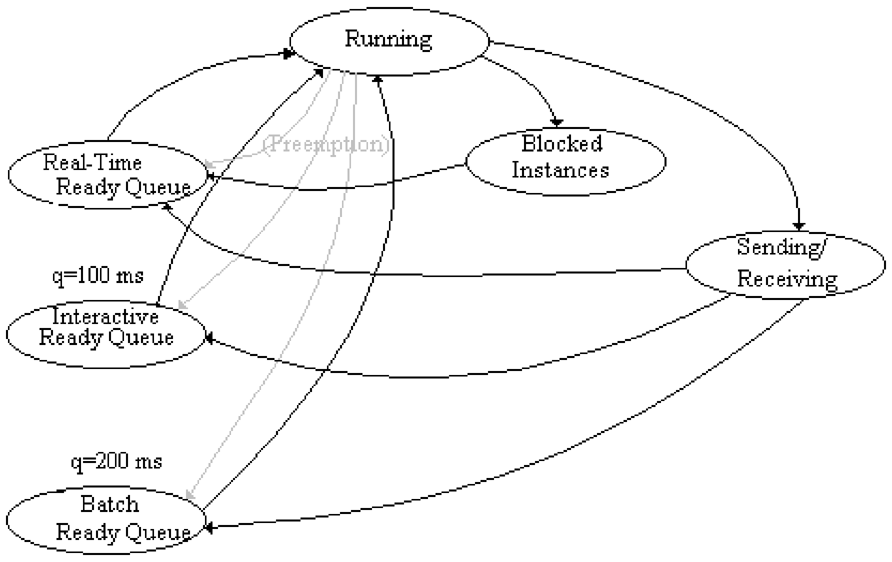


Figure 2. New scheduler structure.

The clock task manages a new queue for blocked instances, used to delay real-time tasks. When a new period starts for a task, the clock driver removes its Process Control Block from the queue and inserts it in the real-time ready queue.

We also paid special attention to the management of these queues to avoid extra overhead. We made big efforts to keep this overhead to a minimum, using efficient data structures. Surprisingly, the results of the benchmark tests did not show improvements. We studied the queue statistically using a M/M/1 model, and we could see that for typical MINIX workloads there are between two and five tasks in the queue. This adds only 100 microseconds of overhead to the clock interrupt.

After changing the scheduler operation, we built libraries to use the new services of the Operating System. We can see a summary of the functions provided in Table 1. First, we show the calls related with the real-time clock management. Then, we show the functions used to define periodic and aperiodic tasks. The primitives include periods or deadlines of the tasks, and worst cases of execution. We also show a set of calls to select different schedulers. Finally, we show some calls used to count deadline misses.

System call	Function provided
set_grain(ticks);	Changes the granularity of the timer.
ticks=get_grain();	Reads present granularity value from the timer
clock();	Tells the time in clock ticks.
set_per(Ti, Ci);	Sets a running task as a Real-Time periodic task. Ti=task period; Ci=worst execution time.
set_aper(Dl, Ci);	Sets a running task as real-time aperiodic. Dl: task's deadline.
inst_end();	Marks the end of a periodic task instance.

set_pri_int(pri);
set_rt_sch(kind);
	Sets the priority of an interactive task.
	Selects a new real-time scheduler.

set_int_sch(kind);	Selects a new interactive scheduler.
start_count();	Start a new count for deadline missing.
c=miss_result();	Returns the number of deadlines missed.

Table 1 - New system calls

On the basis of the information provided by the system calls, the scheduler executes a guarantee routine allowing to know if we can schedule in a timely fashion. To do so, we used Theorem 2 in [Sha90], and Theorem 4 in [Liu73]. In the case that we detect that we cannot predictably execute a task set, we send a new signal, SIG_UNCERTAIN, to the conflicting task. In this way, the programmer can run an alternate task with softer timing requirements.

In the case that we detect an overload, we send a signal SIG_OVERLD to the task set. In the moment an instance finish, the scheduler analyzes the deadline missing. In that case, we send a signal SIG_DL_MISS, so the task can take the proper action. We summarize the signals added in Table 2.

Signal	Event occurred
SIG_OVERLD	An overload is detected
SIG_DL_MISS	A deadline was missed
SIG_UNCERTAIN	The scheduler tells it is uncertain to meet the task set deadlines.

Table 2 - New signals

We decided to add the new services instead of changing the old ones. This approach allows to keep the original semantics of the UNIX System V system calls[1].

In the next section, we will explain some tests we made, and their results.

[1] Unix is a trade mark of AT&T.

4. Some empirical results

We have built a framework to test scheduling algorithms. Using the theoretical basis, a task set satisfying certain restrictions should run predictably. Our idea is to use our framework to analyze the theory in an experimental way. We paid special attention to implementation issues and performance overheads of the classical algorithms. We run tasks not respecting the theoretical bounds, analyzing their behavior. We want to study the problems in detail, to allow the proposal of new solutions.

Our main goal is to analyze the reliability of the scheduling algorithms to run predictably a given task set. With this purpose, we studied the guarantee ratio of each task set (i.e., the relationship between scheduled instances and deadline missing). We can see some of the results we obtained in the Appendix.

First, we analyzed the performance of the original time-sharing algorithm when running real-time tasks. Then, we compared these results with those got using the new CLOCK system call. Finally, we compared the results got against a real-time scheduling algorithm (in this case, the rate-monotonic).

We built different task sets, some of them consisting of different number of tasks with the same period and execution times, and other with variable periods. We used these task sets to compare the two real-time scheduling algorithms. Finally, we tested the results of running sporadic tasks.

Our first results concern with the development and testing times. We can see the figures in Table 3. These figures involve very simple real-time tasks, used with benchmarking purposes. Even, the differences are significant. The increase of the development times relates with the manual tuning of the developed tasks under the original environment.

	Time-Sharing Scheduler		Real-Time schedulers	
I-O bound tasks	Equal Period	Variable Period	Equal Period	Variable Period
5 tasks	1.7	2	1	1.1
15 tasks	2.1	2.7	1.1	1.2
CPU-bound tasks				
5 tasks	2.1	2.4	1.2	1.3
15 tasks	2.5	2.9	1.2	1.4

Table 3. Developing times of real-time tasks using different environments (man-hours)

The different benchmark tests we run let us conclude that [Wai94]:

- The Round-Robin scheduler does not work properly when scheduling real-time tasks. The difference is much greater when the task execution time is greater than the quantum.

- The differences reduce when we use the CLOCK service, and the tasks' periods are alike. This is not a good solution even we were careful while developing the system. If a task period must change, the results will be uncertain. The algorithm behaves less predictably when running tasks with different periods.

- When there is an overload in the system, the rate-monotonic algorithm behaves better than the deadline-driven, because it runs first the most critical tasks (i.e., those with smaller periods) in a stable fashion.

- When the overload reduces, the deadline-driven algorithm behaves better. This happens because it has a more relaxed bound for CPU utilization.

- The values for both algorithms are the same (100% of guarantee ratio) when we respect the theoretical bounds. The overheads do not influence the algorithm performance.

- The situation changes when introducing sporadic tasks. The rate-monotonic behaves better when there are overloads (both for periodic and aperiodic tasks).

- Instead, the deadline-driven algorithm has better guarantee ratio when the overload reduces. The difference is even higher when the sporadic tasks have shorter deadlines than the periodic task's periods.

To finish with this section, we must say that the results always meet the theoretical bounds. After finishing each benchmark, we analyzed the results using the corresponding theorems. Where we respected the theoretical bounds, the benchmarks showed predictable execution.

5. Present work and conclusion.

We have used MINIX operating system to build a framework to test real-time schedulers. We provided new services, allowing the programmers to define tasks with time restrictions, and leaving the scheduler to run them at the needed times. We avoided the programmer intervention in matters related with time control. In this way, he can pay greater attention to solve the application problem, reducing the problems related with accomplishing the timing constraints. Finally, we could show advantages and disadvantages of using different real-time schedulers.

We have just tested the properties of two classic scheduling algorithms. Our next step will be to experience with different schedulers. Our last goal is trying to find new solutions, and test them empirically.

We could see that the real-time scheduling algorithms we implemented have some problems. The Rate Monotonic algorithm bases its results in the notion that the criticality of a task depends on its period. This notion is not always true, for example with emergency aperiodic tasks with long execution times. This algorithm also has low time-loading.

The deadline-driven algorithm is more dynamic, and allows to run aperiodic tasks more safely. The main disadvantage is that the algorithm behaves badly in cases of overloading.

Our final goal is to propose scheduling algorithms to run dynamically real-time tasks, assuring high guarantee ratio. We will use our framework to test them comparing them with the results we have presented in this work.

Even these traditional algorithms have some disadvantages they are much better than traditional time-sharing schedulers to run real-time tasks. We got better guarantee ratio, and the development times also have shown to be much lower, and the overhead is so small that we suggest to use them. For example, we could implement the rate-monotonic algorithm in any operating system having a preemptive priorities' scheduler, as a real-time executive devoted to schedule real-time tasks.

At present, we are trying to test some solutions that improve the performance of these algorithms. We are addressing some important issues, including:

1. Sporadic tasks: We are implementing some known algorithms, such as sporadic servers and aperiodic servers. We will compare these solutions with the results we got running aperiodic tasks.

2. Schedulability: We used the algorithms in a dynamic fashion, to make on-line scheduling. This fact allowed good time loading. We also implemented a multiqueue scheduler, allowing higher time-loading and resource utilization. Now, we are implementing a mixed scheduler to increase the time-loading while reducing the deadline miss ratio. We use a heuristic that entitles the selection of different schedulers depending on the actual workload conditions.

3. Stability: We have incorporated new signals to inform that a task could not reach a deadline. In this way, the programmer can run alternate routines in an emergency state. If there is a transient overload, we can gratefully degrade the system doing imprecise computations. This kind of solution consumes less CPU time, allowing timely execution of a higher number of tasks. We will compare this solution with other algorithms, such as period transformation and imprecise computation models. We have provided facilities letting the user to use multiple version methods, that we will compare with other known solutions.

In the future, real-time systems will have the same application areas than the present systems. However, the systems will be more complex. They will be distributed and with intelligent, adaptive and dynamic behavior [Sta88a]. So, the scientific and engineering problems handled to develop these systems will be greater. The current ad-hoc solutions will not suffice.

With this work we tried a first approach to solve such kind of problems. To do so, we built a framework to develop real-time software, running at predictable times in dynamic environments.

6. BIBLIOGRAPHY

[Che88] CHENG, S; STANKOVIC, J.; RAMAMRITHAM, K. "Scheduling Algorithms for Real-time systems: a brief survey". In "Real-Time Systems", IEEE Press, 1993. pp. 150-173.

[Liu73] LIU, C.; LAYLAND, J. "Scheduling algorithms for multiprogramming in a Hard Real Time System Environment". Journal of the ACM, Vol. 20, No. 1, 1973, pp 46-61.

[Liu91] LIU, W.S. et al. "Algorithms for scheduling imprecise computations". IEEE Computer. May 1991.

[Sha90] SHA, L.; GOODENOUGH, J. "Real-Time Scheduling Theory and Ada". IEEE Computer, April 1990. pp 53-62.

[Sta88] STANKOVIC, J. "Misconceptions about Real-Time computing". IEEE Computer, October 1988. pp 10-19.

[Tan87] TANNENBAUM, A.. "A Unix clone with source code for Operating Systems courses". Operating Systems Review, vol. 21, January 1987.

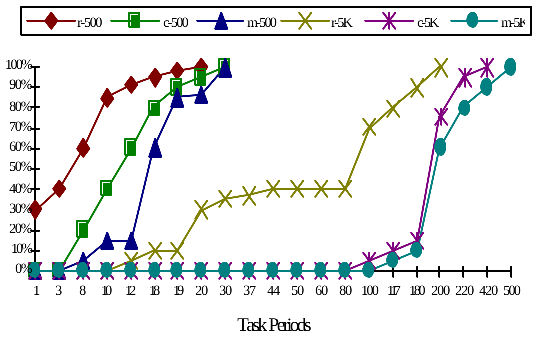
[Tan91] TANNENBAUM, A. et al. "MINIX 1.5 Reference manual". Prentice-Hall, Englewood Cliffs, New Jersey 07632. 1991.

[Wai92] WAINER, G. "A survey of the results of using Minix as a tool for teaching in Operating Systems Courses". Proceedings of the XII International Conference of the SCCC. Editorial de la USACH. 1992.

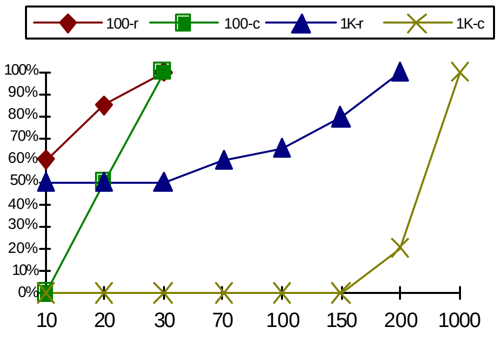
[Wai94] WAINER, G. "Experimental Evaluation of Real-Time Scheduling Algorithms in a Time-Sharing Operating System". Internal report, Computer Sciences Department. Submited to "Sigmetrics and Performance '95".

[Wai94b] WAINER, G. "Inclusión de mecanismos de Tiempo Real en un Sistema Operativo de Tiempo Compartido". Proceedings of the II Encuentro Chileno de Computación. 1994.

APPENDIX - Some guarantee ratio results

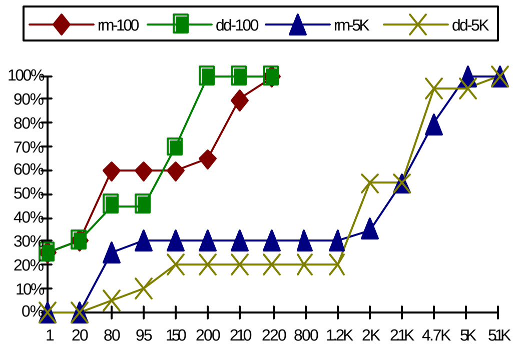


(a)

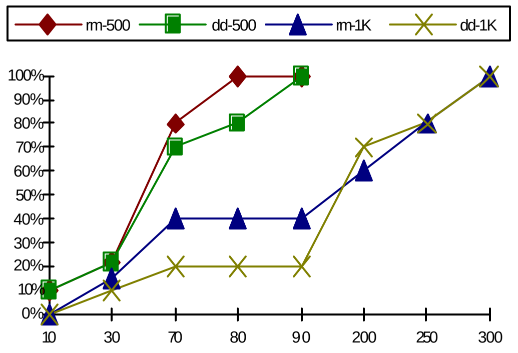


(b)

Figure I. Different clock grains (each curve in graphics). r: real-time scheduler; m: Minix scheduler; c: Minix with clock() service. (a) Tasks with the same period (b) variable periods.

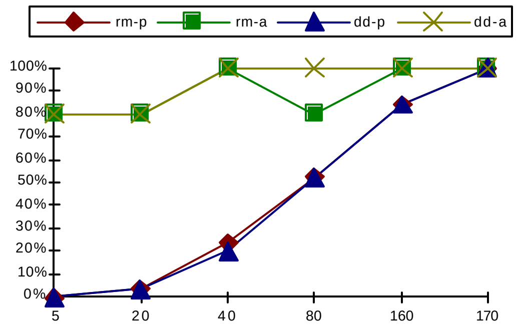


(a)



(b)

*Figure II. Comparison between Rate-Monotonic (rm) and Deadline-Driven (dd) algorithms. (a) tasks with same period; (b) **variable** periods.*



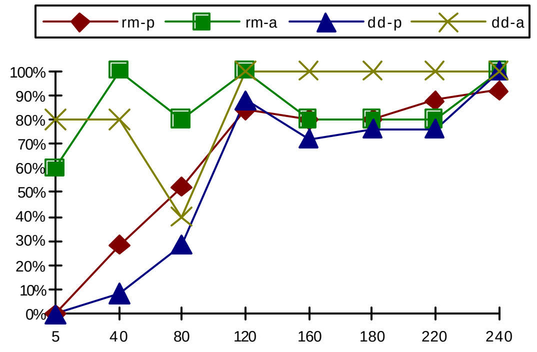


Figure III. Mix of periodic and sporadic tasks. Variable periods. (a)Period equal to sporadic deadline. (b) Sporadic deadline smaller than period.