Kernel Internals for Real-Time

David R. Donari, Leo Ordinez, Rodrigo Santos, and Javier Orozco

Instituto de Investigaciones en Ingeniería Eléctrica
Universidad Nacional del Sur - CONICET
Av. Alem 1253 - (8000) Bahía Blanca
Buenos Aires - Argentina
{ddonari,lordinez,ierms,jorozco}@uns.edu.ar

Abstract. In this paper, some main features of the design and implementation of a Real-Time Operating System are presented. The hierarchical model to schedule task from the theoretical point of view to the implementation is explained. In addition, several implementation improvements related to the reduction of kernel overhead are also proposed. A comparative analysis shows how differents RTOS implement the scheduling process. Finally, an experimental evaluation is exposed. This evaluation shows the influence of each kernel improvement to the overall performance of the system.

1 Introduction

In the last years there has been an important growth in the production of Embedded Systems (ES) understood as those in which a computer forms part of a bigger system and that depends on its own microprocessor. This growth has brought new challenges for the software developers.

The rapid development of new hardware architectures make possible important improvements in performance that would require a design from scratch of the software if no OS is present to handle the hardware. In fact, there is an important research activity in the OS area oriented to ES. These OS have particular requirements that are not present in general-purpose OS. Among the main ones, the OS should have a small footprint on memory as this is a scarce resource in ES. It also has to provide real-time primitives since most ES are built upon physical processes with stringent timing constraints. Finally, the OS overhead or interference on the execution of the processes of the system should be minimal.

In this paper, some main features of design and implementation of the SOOS kernel [1] are presented. The first one is directly related to the way in which SOOS schedules and dispatches tasks. SOOS provides real-time scheduling for both soft and hard tasks by means of the resource reservation paradigm [2]. In this sense, it introduces this kind of hierarchical scheduling approach in a native and inexpensive way. The second contribution is related to the way in which kernel implements the scheduling time. This implementation significantly reduces the overhead involved, because of the interaction between chosen scheduling policy and the data structures used. Finally a syscall is introduced to show how to

take advantage of the *gained slack time* [3]. These three contributions are deeply explained, in the rest of the paper, both from a theoretical and practical point of view.

The rest of the paper is organized as follows: several previous works on Real-Time Operating Systems are analyzed in Section 2; Section 3 shows how to model server-based real-time systems; the SOOS scheduling process is explained in Section 4; in Section 5 an experimental study is performed to show how the theoretical concepts applied on SOOS are reflected in the practice; finally, conclusions are exposed.

2 Related Work

Within the academy, there are many papers that described and proposed different RTOS for ES, which can be highlighted:

EMERALDS is a real-time operating system designed for small-memory embedded applications proposed by Zuberi *et at* in [4]. The authors introduce a mixed scheduling policy that combines fixed and dynamic priorities to reduce the overhead of the kernel in the scheduling and dispatching of tasks. The new algorithm orders the tasks by increasing periods and divides the ready queue in two. The higher priority one is scheduled by dynamic priorities, Earliest Deadline First, and the second by fixed priority with Rate Monotonic.

In [5] a very simple real-time kernel is proposed for performing flexible control on low-cost microcontrollers. The objective of the kernel is to provide a multitasking platform on low performance processors in such a way that the control algorithms can be implemented in a simple manner. There are three different scheduling policies that are selected at system startup which are Earliest Deadline First, Rate Monotonic and Deadline Monotonic. The main contribution of the paper is the creation of an intermediate layer named the *reflective layer* that is in charge of transferring the data between the kernel and user spaces. However, there is a small number of tasks that the OS can handle, 13 plus the background task that keeps the system running. Asterix is a real-time kernel that provides some tools for the implementation, analysis and debugging of real-time applications [6]. The kernel can schedule tasks with fixed or dynamic priorities. The OS provides also a monitoring tool for debugging and performing time analysis of the system under construction.

3 Real-Time Systems Modeling

When modeling a particular real-time application that will be implemented on an RTOS, it is necessary to clearly distinguish between policies and mechanisms. Moreover, the analysis is tightly related to the concrete definition of tasks and servers. In addition, a data structure is proposed to support the implementation of the concepts explained.

3.1 System Modeling

A general-purpose operating system (GPOS) generally manages software entities named *processes*. In an RTOS, those entities are called *tasks* and the set of them compose a particular real-time application. However, real-time tasks differ from normal processes in that the former ones have intrinsic timing constraints.

When analyzing tasks according to the real-time scheduling theory [7] the actual functions that they perform in the system are stood on the sidelines and the analysis focuses only on their timing restrictions. This is, a task is seen as a tuple (C_i, T_i, D_i), where C_i stands for its worst-case execution time (WCET), T_i for the task's period and D_i for its deadline.

A server is defined as an abstract software entity in which a task is contained. A task within a server is subject to the server own rules. Usually, a server is characterized by an activation period P_s and a budget Q_s, which is its available time to serve a task. The relation $U_s = \frac{Q_s}{P_s}$ establishes the bandwidth of a server. In this sense, a task is said to run on a virtual processor whose speed is U_s times the actual processor speed.

In SOOS, the server's paradigm applied is based on BIDS. This algorithm uses dynamic priorities [8, 9] and a parameter α_s to establish a variation in the server reactivation.

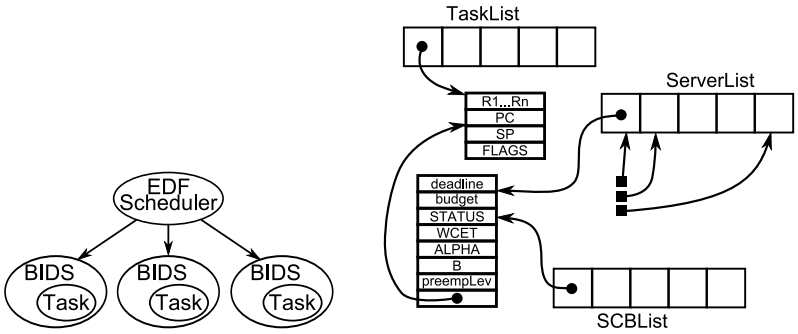
3.2 Linking Tasks and Servers

The SOOS implementation provides a priority management based on BIDS and a task dispatching mechanism. In this way, a server is considered linked to a particular task (see Figure 1(a)). Therefore, a server will have its period P_s equal to the task's interarrival time associated to it and a budget Q_s greater or equal to the WCET. In the case of a soft task, the server budget is defined according to the expected Quality of Service (QoS) of the task.

The set of servers are the *scheduling entities*. On the other hand, it will be defined as *execution entities* the set of parameters that establishes the task execution in SOOS (program counter, registers, flags, etc). This is, a logical viewpoint is concerned with scheduling policies; and a physical viewpoint with dispatching mechanisms. In this sense, SOOS *schedules servers, but executes tasks*. Figure 1(b) shows the set of fields that belong to each entity. On the one hand, `TaskList` saves the information relative to the execution of a task. On the other, `SCBList` stores information relative to scheduling parameters. Both lists are fixed, in the sense that no insertion or deletion is done on any of them during runtime. In particular, they are implemented with arrays of n elements (being n the number of servers in the system).

3.3 Server List

The `ServerList` structure is used to keep track of active servers during runtime. Since this is a dynamic list, a pointer indicating the end of the list is maintained.



(a) Logical scheme. (b) Implementation scheme.

Fig. 1: Server scheme.

In addition, the structure has two pointers to the first and second place of the list, this is the two highest priority servers.

The previous list design allows to find the highest priority server in an $O(1)$ time. Deletion time is $O(n)$, since the deletion is always of the first component, then the second one takes its place and finally a new second component must be found. However, insertion of a new element is $O(1)$ [1]. This is because it is first checked against the highest priority elements, if its priority is higher than any of the two, the second one is always sent to the end of the list. If the new element is not greater than any of the two first ones, it is put at the end. Thus, the list can be seen as queue with the two highest priority elements ordered but respecting the natural behavior of a queue.

This kernel structure is useful when a bounded search time is required. In the following sections several advantages of this design will be explained.

4 Scheduling Real-Time Tasks

The Server Oriented Operating Systems implements a two-level hierarchical scheduling scheme. This is, the SOOS kernel schedules servers and the servers dispatch tasks. Particularly, the highest level scheduling policy is Earliest Deadline First (EDF) [10]. This policy establishes that the server whose deadline is the closest one to the current time is the one with highest priority. In the second level is the Behavioral Scheduling Server (BIDS) [11, 12], that is in charge of dispatching the actual tasks that compose the real-time system. The BIDS policy is a server-based approach that adjusts dynamically the execution frequency of each encapsulated task according to its last instance behavior. In this sense,

[1] O(1) stands for a bounded time. This is, it is independent of the number of tasks in the system.

each task before ending returns a value to its server that is used to determine its next reactivation period.

4.1 Scheduling Process Description

In SOOS the scheduling points (SP) are determined by a periodic timer, which imposes a time base to the system. This time base is known as *slot* and it has the duration of the timer period. When the timer expires, it generates an interrupt that is attended by the Timer Tick Handler (TTH). This handler is in charge of giving control to the SOOS kernel.

In addition to the periodic scheduling points set by the timer, when a task ends (generally in some point in the middle of a slot), it makes the syscall `taskEnds()` in order to inform the SOOS kernel of that situation. With this, the kernel could quickly send the highest priority pending task to execute. Thus, the SOOS kernel exploits what is commonly known as *gained slack time* [3] to execute tasks, when it is possible. This is, the time left by tasks that require less than their WCET.

Timer Tick Handler

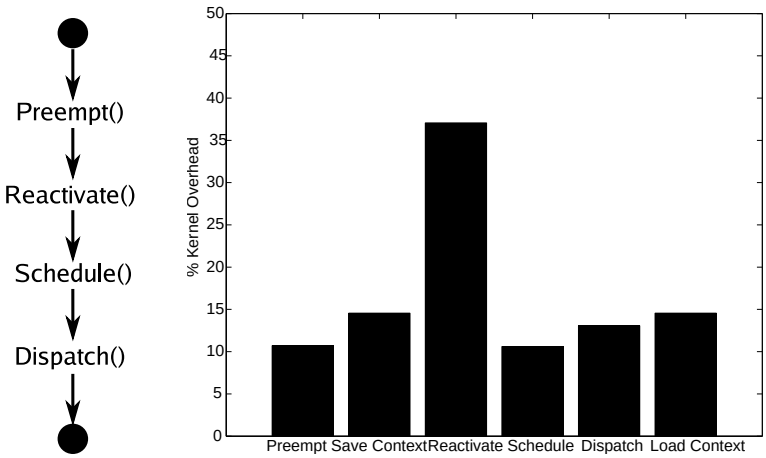
This kernel operation is invoked in each SP to schedule and dispatch task. In addition, the TTH uses the kernel data structures to guarantee the correct system behavior. The kernel execution time has to be just a percentage of the slot time. Thus, the set functions called during a timer interrupt must be bounded. Therefore, the TTH implementation is of major importance for the overall functioning of the system both for correctness purposes and real-time constraints.

Basically, *preemption* mechanism saves all information about the running task when it is deallocated the processor. This instance uses the physical data structure `TaskList` to save the execution parameters (program counter, flags, stack, registers, etc).

In the `reactivation` stage, the SOOS kernel checks the `SCBList` to determine which BIDS server has to be reactivated. This is done by analyzing the field r_i of every server to see whether they are equal to zero or not. SOOS improves the process by keeping track of a variable named `nextReactivation`, that indicates the closest reactivation time. In this way, only the value of the variable is tested. When the value of the variable reaches the calculated reactivation time (*i.e.*, one or more servers must be reactivated in that slot), `nextReactivation` is recalculated. Particularly, in any other slot (i.e. a slot where there would be no reactivations) the reactivation function takes $O(1)$ to execute (see Figure 3) due to the insertion policy.

One of the limitations of RTOS concerning scheduling is the excessive overhead involved in the case of simultaneous reactivations. Generally, to overcome this situation, an RTOS establishes a limit on the number of supported tasks in the system. As a consequence, it restricts the application developer to build its system with a bounded number of tasks. For instance, in [5] the maximum number of tasks is set to 13 due to the previous situation. The mentioned overhead is given when every task must be checked and inserted in the ready tasks

list. Particularly, in the case of SOOS the worst-case scheduling scenario is given when all the servers must be reactivated and inserted in the `ServerList`. Generally, the previous process would take an $O(n^2)$ time, since it involves searching in an unsorted list and inserting in an ordered one. However, SOOS reduces this time to $O(n)$ by means of the `ServerList` structure presented in Section 3.3. Recall that the `ServerList` has only two elements ordered, the two highest priority servers. With this, SOOS increases the number of possible reactivations and consequently the potential number of supported servers. Figure 2(b) shows the average distribution of time in the TTH.



(a) TTH (b) TTH Time Distribution

Fig. 2: Timer tick handler

Based on the structure of `ServerList` and the reactivation algorithm, the scheduling stage remains simple, since it only checks which is the ready server with the closest deadline and the chosen one is sent to execute. In the actual implementation, the scheduler just takes the server in the front of the `ServerList`. Thus, the scheduling stage takes an $O(1)$ execution time. The chosen server identifier is then stored in a variable named `ExePid`, that is taken by the dispatcher to perform the necessary functions (*i.e.*, loading context, adjusting stack, etc.) to put the corresponding task in an *exe* state.

Task Ends()

The syscall *taskends()* is an operation implemented in the kernel whose fundamental objectives are to correctly complete the execution of a task and allow a fast scheduling, when it is possible. This operation must be invoked for ev-

```
Reactivate(){
 if (nextReact == t){
    for 1 to n server {
          if (server.ri == timeElapsed){
              server.state=ready
              server.ri=task.Ti
              insertToServerList(serverPid)}
         else
              server.ri = server.ri - timeElapsed;
         nextReact = MIN(nextReact,server.ri);}
    timeElapsed = nextReact;}
  else
     nextReact = nextReact-1;}
}
```

Fig. 3: Reactivation algoritm

ery task at the end of its source code, to indicate completion. When invoked, the kernel starts the process of ending a task, saving the information about the execution and setting a internal parameter. In the next SP, the previously mentioned parameter is used to indicate that the task must be removed from the ServerList.

Once the kernel correctly performed the ending of a task, the next step is to make a fast scheduling. This fast scheduling is given by a sequence of three actions: take the second element of the ServerList, dispatch the task associated to the chosen server and report success.

The execution of this fast scheduling is an atomic process. So, if the timer interrupts in the middle of the process, operations are discarded. The atomicity is achieved through the use of internal kernel flags that indicate the status of operation reached. These flags can be seen as internal checkpoints.

Finally, it could be a situation where the two highest priority tasks finish their execution within a slot. In that case, the gained slack time obtained by the ending of the second task is wasted. Therefore, that time is used by the dummy task.

4.2 Comparative Analysis

In general, when there is a reactivation in a system, the kernel performs basically three activities: 1) it has to check which scheduling entity must be reactivated; 2) each reactivation has to be marked in order to be considered for scheduling (*i.e.*, change a field of the entity control block or insert it in a READY list). This activity is done while doing the previous one. Finally, activity 3) is a rescheduling process. These three activities are the main sources of the kernel overhead. In what follows, a comparative analysis of them is done. The analysis is made comparing SOOS with two other real-time kernels: one proposed by Thane *et al.* in [6] and the other proposed by Marau *et al.* in [5].

In the first place, the approach proposed by Thane *et al.* has an execution time of $O(n)$ in activities 1) and 3), and $O(1)$ in activity 2). This is because activity 1) checks in every slot the complete list of tasks; activity 2) just changes one field of the reactivated tasks and inserts the task in the READY list in an unordered way. Activity 3) searches the complete READY list for the highest priority task.

Secondly, the kernel proposed by Marau *et al.* has an execution time order of $O(n)$ or $O(1)$ in activity 1), depending whether there are reactivations or not, respectively. Activity 2) has $O(1)$, since it only changes one field of the tasks. Activity 3) is identical to case of Thane *et al.*, this is $O(n)$.

In SOOS, Activity 1) has the same execution time order of the approach proposed by Marau *et al.*, that is $O(n)$ or $O(1)$ depending on the slot. Activity 2) has an order $O(1)$, since it inserts in an orderly fashion in the `ServerList`. This process always involves two comparisons. Lastly, due to the ordered insertion in the previous activity, activity 3) has execution time order of $O(1)$. Note that the highest priority server is always in the front of the `ServerList`.

The previous discussion shows theoretically that SOOS implements the timer tick handler in a very efficient manner. This is because of the improvements done in the reactivate function and especially in the `ServerList` structure. Nevertheless, a price is always paid. In this case, SOOS pays that price only in the slot following that of a task ending, since the kernel has to find the second highest priority server, as explained in Section 4.1. With all, this search is only made when a task ends and only two tasks can be deleted in the same SP, so the price is not expensive per slot.

5 Experimental Study

The Server Oriented Operating System was implemented in the RCX Lego Mindstorms® as a prototype to analyze the features exposed in this paper. The decision of choosing the Lego platform was based on the hardware management facilities provided by that system. The main characteristics of the platform can be summarized as follows: the microcontroller used is a Hitachi H8/3297 operating at 16 Mhz; the system has 16KB of Flash memory and 32KB of RAM.

The prototype implementation of SOOS was executed with several configurations. In all cases, the Logic Analyzer HP 1651A was used to capture timing information. Figure 4 summarizes the SOOS characteristics for a randomly generated set of 15 tasks.

In Figure 5 the kernel overhead time is depicted for different SOOS implementations with several random sets of tasks in groups between 5 and 40. In the first place, it can be seen that the modification to the reactivate function and the implementation of the `ServerList` produce a significant outcome with respect to the implementation without any improvement. However, when both the reactivate improvement and the `ServerList` implementation are used together the result is even better, approximately 46% on average of kernel overhead decrease. From this comes up that the two improvements presented are not contradictory,

Description	Measure
Kernel Size	3.76KB
Executable Image Size	6.3KB
Workload	15 tasks
Slot Time	10ms
Experiment Length	10min
Average Kernel Overhead	180μs
Minimum Kernel Overhead	60μs
Maximum Kernel Overhead	435μs

Fig. 4: SOOS characteristics.

since the reactivate one alone presented a decrease of 20% and `ServerList` one a decrease of 35%. With this, both improvements can normally coexist and substantially decrease the kernel overhead time.

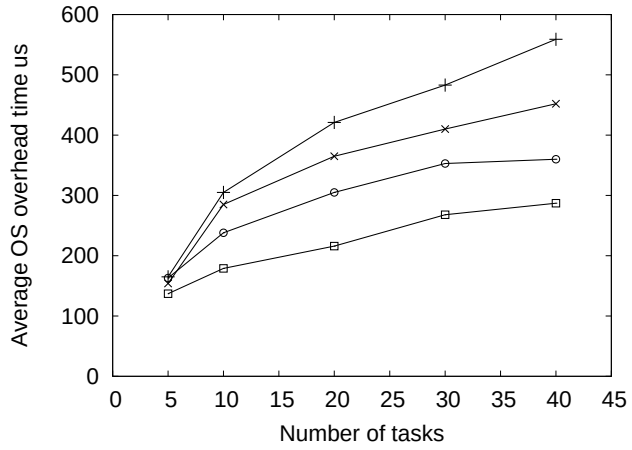


Fig. 5: Kernel overhead under different configurations: + SOOS without modifications, × SOOS with reactivate improved, ∘ SOOS with ServerList improved, □ SOOS with both improvements

6 Conclusions

In this paper some aspect related to the implementation of a RTOS was presented for embedded systems. Usually this kind of systems have limited memory capacity both in ROM and RAM, low speed processors, and a strong interaction with the external world in the form of I/O ports to sense variables and handle actuators. These characteristics impose strict restrictions on the RTOS that are

translated in a set of requirements or design goals: small size, low kernel overhead, I/O management, shared resources and programming facilities. The Server Oriented Operating System (SOOS) proposed here was oriented to fulfill these requirements and made advances in three basic areas that are detailed in what follows.

SOOS was designed following the resource reservation paradigm. With it, a hierarchical scheduling can be done providing the important property of temporal isolation. The reservation mechanism is used for every kind of task, hard, soft or non-real time. With this, the operation of the system can be guaranteed for each particular task based only on its own characteristics. SOOS introduces in this aspect new ideas for the scheduling and dispatching. The logical and physical viewpoints of tasks help in the creation an efficient management of data structures for that scheduling and dispatching. Experimental results validate the improvements in the timing behavior of SOOS.

References

1. Donari, D., Ordinez, L., Santos, R., Orozco, J.: Real-time server oriented operating system for embedded applications. In: Proceedings of the XXXIV Conferencia Latinoamericana de Informtica, Santa Fe, Argentina
2. Rajkumar, R., Juvva, K., Molano, A., Oikawa, S.: Resource kernels: a resource-centric approach to real-time and multimedia systems. (2001) 476–490
3. Davis, R., Tindell, K., Burns, A.: Scheduling slack time in fixed priority preemptive systems. In: Real-Time Systems Symposium, 1993., Proceedings.
4. Zuberi, K.M., Pillai, P., Shin, K.G.: Emeralds: a small-memory real-time microkernel. In: SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles, New York, NY, USA, ACM (1999) 277–299
5. Marau, R., Leite, P., Velasco, M., Marti, P., Almeida, L., Pedreiras, P., Fuertes, J.: Performing flexible control on low-cost microcontrollers using a minimal real-time kernel. Industrial Informatics, IEEE Transactions on **4**(2) (May 2008) 125–133
6. Thane, H., Pettersson, A., Sundmark, D.: The asterix real-time kernel. In: 13th EUROMICRO INTERNATIONAL CONFERENCE ON REAL-TIME SYSTEMS, INDUSTRIAL SESSION, IEEE Computer Society (June 2001)
7. Sha, L., Abdelzaher, T., Årzén, K., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J., Mok, A.K.: Real time scheduling theory: A historical perspective. Real-Time Syst. **28**(2-3) (2004) 101–155
8. Spuri, M., Buttazzo, G.: Efficient aperiodic service under earliest deadline scheduling. Real-Time Systems Symposium, 1994., Proceedings. (Dec 1994) 2–11
9. Abeni, L., Buttazzo, G.: Integrating multimedia applications in hard real-time systems. In: RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium, Washington, DC, USA, IEEE Computer Society (1998) 4
10. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in hard real time environment. ACM **20** (1973) 46–61
11. Ordinez, L., Donari, D., Santos, R., Orozco, J.: A behavior priority driven approach for resource reservation scheduling. In: SAC '08: Proceedings of the 2008 ACM symposium on Applied computing, New York, NY, USA, ACM (2008) 315–319
12. Ordinez, L., Donari, D., Santos, R., Orozco, J.: Resource sharing in behavioral based scheduling. In: SAC '09: Proceedings of the 2009 ACM symposium on Applied computing, New York, NY, USA, ACM (2009)