Proceedings of the 2006 ANIPLA
International Congress on
Methodologies for Emerging
Technologies in Automation (ANIPLA)

Why Real-Time Computing?

Giorgio Buttazzo
Scuola Superiore Sant'Anna, Italy
Email: buttazzo@sssup.it

Abstract

Today's control systems are implemented in software and often require the concurrent execution of periodic and aperiodic activities interacting through a set of shared memory buffers storing global data. Most of the times, controllers are designed assuming an ideal and predictable behavior of the execution environment; in other cases, a constant delay is only considered in the loop to take computer interferences into account. Unfortunately, reality is more complex, and the operating system may introduce unpleasant surprises in control applications.

This paper presents the potential problems that may arise in a computer controlled system with real-time characteristics and proposes a set of methods for ensuring predictability and guaranteeing that the system behaves as predicted in the design.

1 Introduction

Most of today's control systems are implemented in software and often require the concurrent execution of several periodic and aperiodic activities interacting through a set of shared resources (typically consisting of memory buffers storing global data). In a modular design, each sensory acquisition activity is normally associated with a periodic task that gets data from the sensory board and puts them into a shared memory buffer. Control activities are also implemented as periodic tasks that read data from buffers, compute the output values of the control variables and put the results in other buffers, used by device drivers to actually drive the actuators. In some cases, especially when the system is based on a hierarchial architecture, control and acquisition activities are required to run at different rates.

Figure 1 illustrates a typical control application consisting of six tasks: boxes represent shared resources and disks represent tasks (acquisition, processing, control, and output tasks are denoted by A, C, P, and O, respectively). Each layer of this control hierarchy effectively decomposes an input task into simpler subtasks executed at lower levels. The top-level input command is the goal, which is successively decomposed into subgoals, or subtasks, at each hierarchical level, until at the lowest level, output signals drive the actuators. Sensory data enter this hierarchy at the bottom and are filtered through a series of sensory-processing and pattern-recognition modules arranged in a hierarchical structure. Each module processes the incoming sensory information, applying filtering techniques, extracting features, computing parameters, and recognizing patterns.

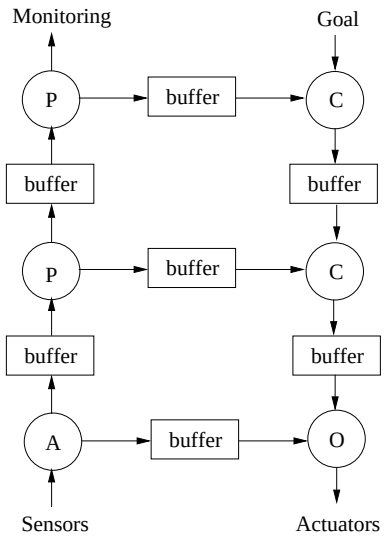

Figure 1. A hierarchical control system.

Sensory information that is relevant to control is extracted and sent as feedback to the control unit at the same level; the remaining partially processed data is then passed to the next higher level for further processing. As a result, feedback enters this hierarchy at every level. At the lowest level, the feedback is almost unprocessed and hence is fast-acting with very short delays, while at higher levels feedback passes through more and more stages and hence is more sophisticated but slower. To correctly implement such a hierarchical control structure and execute all tasks in a predictable fashion, several issues have to be taken into account:

1. First of all, the operating system should provide a proper support for periodic tasks execution and include an internal mechanism to automatically activate a task at the beginning of each period.

2. The scheduling algorithm should maximize the processor utilization and enable the user to analyze the system behavior to verify the schedulability of the task set (that is, to check whether each task is able to complete within its deadline).

3. The kernel should also detect possible deadline misses due to execution overruns, to allow the application to react with proper recovery actions.

4. When tasks cooperate through mutually exclusive resources protected by classical semaphores, the system is prone to priority inversion phenomena [21], which can cause unbounded delays on tasks execution due to complex blocking interferences. To prevent such a problem, the kernel should provide suitable concurrency control protocols to correctly access shared resources.

5. Non blocking communication mechanisms are also essential for exchanging information among periodic tasks running at different rates. In fact, a blocking factor could cause a fast task to slow down, synchronizing with the rate of the slower task. This problem can be solved by using special asynchronous buffers with override semantics and non-consumable messages.

6. When control applications perform intensive I/O operations, interrupt handling routines can create high and unpredictable interference on control activities, causing extra delays and jitter that could even jeopardize system stability. To limit the interference of interrupt drivers, suitable aperiodic service mechanisms need to be adopted in the kernel to protect the execution of critical tasks.

Unfortunately, these problems are often ignored and digital controllers are normally designed by neglecting the characteristics of the machine running the application, so implicitly assuming an ideal and predictable behavior of the execution environment. For simple control systems running on a computer with abundant resources, such a design practice does not cause any problem in the control system. However, when the application has a high degree of concurrency or it runs on an embedded platform with limited resources, the consequences become visible and may significantly degrade system performance. For some critical conditions, the non-ideal behavior of the execution environment can even cause the instability of the controlled system [7]. For example, an unpredictable delay introduced in a task controlling an inverted pendulum could cause the pole to cross the critical angle and fall down.

The objective of this paper is to illustrate the possible problems that can be introduced in a control application by the operating system and present a set of solutions that can be easily adopted to make the control system more predictable.

The rest of the paper is organized as follows: Section 2 describes the mechanisms that a real-time kernel should provide for supporting period tasks execution; Section 3 presents the scheduling algorithms that can be used to handle periodic tasks and the analysis that can be performed to guarantee the feasibility of the application; Section 4 illustrates some problems caused by mutual exclusion and some solutions to overcome them; Section 5 presents an asynchronous communication mechanism for sharing data among periodic tasks with different rates; Section 6 describes problems and solutions caused by interrupts; and, finally, Section 7 states our conclusions.

2 Supporting periodic tasks

Periodic activities represent the major computational load in a real-time control system. For example activities such as actuator control, signal acquisition, filtering, sensory data processing, action planning, and monitoring, need to be executed with a frequency derived from the application requirements.

A periodic task is modeled as an infinite sequence of instances, or jobs, that cyclically execute the same code on different data. Each task τ_i is characterized by a worst-case execution time C_i, a period T_i, and a relative deadline D_i. The k^{th} job of task i is denoted as $\tau_{i,k}$, (where $k = 1, 2, \ldots$). Assuming that all the tasks are simultaneously activated at time $t = 0$, the release time $r_{i,k}$ of job $\tau_{i,k}$ (that is, the time at which the task becomes ready for execution for the k^{th} time) is given by $r_{i,k} = (k-1)T_i$. Then, the job executes for a time $c_{i,k} \leq C_i$ and must complete its execution by an absolute deadline equal to $d_{i,k} = r_{i,k} + D_i$. The finishing time of a job $\tau_{i,k}$ is denoted as $f_{i,k}$ and the worst-case response time of a task is defined as the maximum finishing time relative to the release time, that is, $R_i = \max_k (f_{i,k} - r_{i,k})$.

To correctly support periodic task execution and relieve the user from defining its own timers, a real-time operating system should have the following features:

1. First of all, it should include a programming interface for creating periodic tasks, allowing the user to specify the typical periodic task parameters (that is, the computation time C_i, the period T_i, and the relative deadline D_i).

```
/*------------------------------------------------------------*/
    task_id = task_create(body, ex_time, period, deadline);
    task_activate(task_id);
/*------------------------------------------------------------*/
    task control()
    {
    <local variables>

        while (TRUE) {
            <get sensor data from the input buffer>;
            <process sensory data>
            <put results in the output buffer>
            task_endcycle();
        }
    }
/*------------------------------------------------------------*/
```

Figure 2. Typical operating system calls for periodic tasks.

2. Then, it should enforce periodicity by a kernel mechanism for automatically activating a task at the beginning of each period.

3. Finally, it should perform a runtime check to verify that each job executes for no more than its worst-case execution time C_i and completes no later than its absolute deadline $d_{i,k}$. An overrun exception should be generated if $c_{i,k} > C_i$, and a deadline miss exception if $f_{i,k} > d_{i,k}$.

Figure 2 illustrates typical system calls used for creating and activating a periodic task. A sample periodic control task is also shown.

The `task_endcycle()` system call is needed to notify the kernel that the job finished its cycle and needs to be suspended until the beginning of the next period. All suspended jobs have to be enqueued in an IDLE queue, ordered by release times. The kernel is responsible for checking the system clock to wake up all the jobs that reached the beginning of their next period. Figure 3 illustrates the typical state transition diagram for a periodic task.

Notice that, in order to check for execution overruns, the kernel must have a precise time accounting mechanism to keep track of the actual job execution. Unfortunately, only very few operating systems (e.g., Shark [22]) include such a feature.

3 Task scheduling

The test to verify the schedulability of the task set heavily depends on the scheduling algorithm and on the

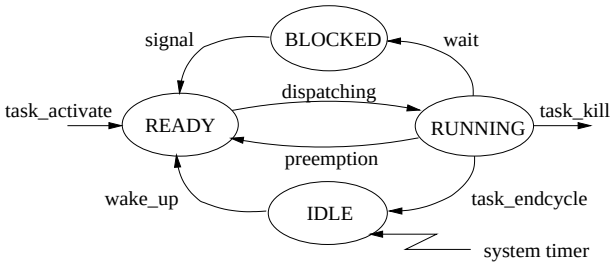

Figure 3. Task state diagram for a periodic task.

task model. In 1973, Liu and Layland [17] analyzed two scheduling algorithms for handling periodic tasks: Rate-Monotonic (RM), which assigns each task a fixed priority directly proportional to its activation rate, and Earliest Deadline First (EDF), which assigns tasks a dynamic priority inversely proportional to the current absolute deadline, so the task with the highest priority is the one with the earliest absolute deadline. Given a set of n periodic tasks with relative deadlines equal to periods ($D_i = T_i$), Liu and Layland were able to relate the feasibility of the task set with the processor utilization, defined as $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$. They proved that, if tasks do not block on synchronous operations,

- A set of n periodic real-time tasks is schedulable by RM if $U \leq n(2^{1/n} - 1)$.
- A set of n periodic real-time tasks is schedulable by EDF if and only if $U \leq 1$.

Note that the RM bound provides only a sufficient condition for guaranteeing the schedulability, hence a task set could be schedulable by RM even for higher utilizations less than or equal to 1. Clearly, no algorithm can schedule a task set if $U > 1$. The RM bound decreases with n and tends to the following limit value:

$$\lim_{n \to \infty} n(2^{1/n} - 1) \; = \; \ln 2 \; \simeq \; 0.69.$$

In spite of its lower utilization bound, the RM algorithm is widely used in real-time applications, manly for its simplicity. In fact, being a static scheduling algorithm, it can easily be implemented on top of commercial operating systems, using a set of fixed priority levels, whereas EDF requires an explicit kernel support for managing absolute deadlines. However, EDF is superior to RM under several aspects and allows the application to fully exploit the available resources, reaching up to 100% of the available processing time. Dertouzos [9] showed that EDF is optimal among all on line algorithms, meaning that if a task set is not schedulable by EDF, then no algorithm can ever produce a feasible

schedule for that task set. When the utilization is less than one, the residual fraction of time can be efficiently used to handle aperiodic requests activated by external events. In addition, compared with RM, EDF generates a lower number of preemptions, thus causing less runtime overhead due to context switches. A detailed comparison between RM and EDF can be found in [6].

A more precise schedulability test for RM can be performed using the Hyperbolic Bound [4], according to which a periodic task set is schedulable by RM if

$$\prod_{i=1}^{n} \left(\frac{C_i}{T_i} + 1 \right) \leq 2. \quad (1)$$

The schedulability analysis for tasks with relative deadlines less than or equal to periods can also be performed, but it is more complex for both algorithms. The Deadline Monotonic (DM) algorithm, proposed by Leung and Whitehead [16], extends RM to handle tasks with relative deadlines less than or equal to periods. According to DM, at each instant the processor is assigned to the task with the shortest relative deadline. A necessary and sufficient test to verify the schedulability of a periodic task set was independently proposed several authors [2, 13, 15]. It consists in computing the worst-case response time R_i of each task and verifying whether it is less than or equal to its relative deadline. The worst-case response time is derived by summing its computation time and the interference caused by tasks with higher priority:

$$R_i = C_i + \sum_{k \in hp(i)} \left\lceil \frac{R_i}{T_k} \right\rceil C_k. \quad (2)$$

where $hp(i)$ denotes the set of tasks having priority higher than task i and $\lceil x \rceil$ denotes the ceiling of a rational number, i.e., the smaller integer greater than or equal to x. The equation above can be solved by an iterative approach, starting with $R_i(0) = C_i$ and terminating when $R_i(s) = R_i(s-1)$. If $R_i(s) > D_i$ for some task, then the task set cannot be feasibly scheduled by DM. Under EDF, the schedulability analysis for periodic task sets with deadlines less than periods is based on the *Processor Demand Criterion* [3]. According to this method, a task set is schedulable by EDF if and only if, $U < 1$ and, for every interval of length $L > 0$ the overall computational demand is no greater than the available processing time, that is, if and only if

$$\forall L > 0 \quad \sum_{i=1}^{n} \left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor C_i \leq L. \quad (3)$$

Note that this test can only be checked for values of L equal to those deadlines no larger than a given bound $L_{max} = min(L^*, H)$, where H is the task hyperperiod (that is, the least common multiple of the periods) and L^* is given by

$$L^* = \sum_{i=1}^{n} \frac{(T_i - D_i)C_i}{(1-U)T_i}. \quad (4)$$

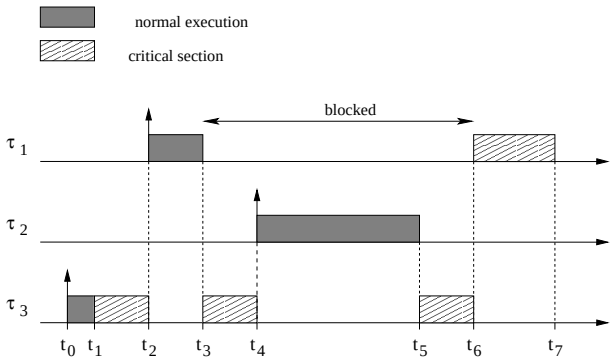

Figure 4. Example of priority inversion.

4 Shared resource protocols

When tasks interact through shared memory buffers, the use of classical synchronization mechanisms, such as semaphores or monitors, can cause a phenomenon known as priority inversion [21]. It refers to the case in which a high priority task is blocked by a low priority task for an unbounded interval of time. Such a situation can create serious problems in critical control systems, since it can jeopardize system stability. For example, consider three tasks, τ_1, τ_2 and τ_3, having decreasing priority (τ_1 is the task with the highest priority), and assume that τ_1 and τ_3 share a data structure protected by a binary semaphore S. As shown in Figure 4, suppose that at time t_1 task τ_3 enters its critical section and that, at time t_2, it is preempted by τ_1.

At time t_3, when τ_1 tries to access the shared resource, it is blocked on semaphore S and we would expect that it is blocked for an interval no longer than the time needed by τ_3 to complete its critical section. Unfortunately, however, the maximum blocking time for τ_1 can be much larger. In fact, if a medium priority task (like τ_2) preempts τ_3 before it can release the resource, the blocking interval of τ_1 is prolonged for the entire execution of τ_2! The problem could be solved by simply disabling preemption inside critical sections. This solution, however, is appropriate only for very short critical sections, because it could introduce unnecessary delays in high priority tasks, even though they do not share resources. A more efficient solution is to regulate the access to shared resources through the use of specific concurrency control protocols [19], designed to limit the priority inversion phenomenon.

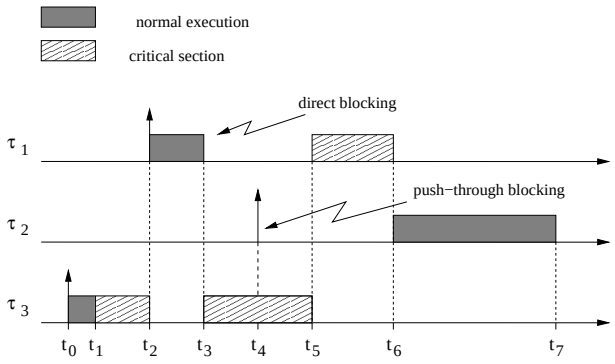

Figure 5. Schedule produced under Priority Inheritance.

4.1 Priority Inheritance Protocol

An elegant solution to the priority inversion phenomenon caused by mutual exclusion is offered by the Priority Inheritance Protocol (PIP) [21]. Here, the problem is solved by dynamically modifying the priorities of tasks that cause a blocking condition. In particular, when a task τ_a blocks on a shared resource, it transmits its priority to the task τ_b that is holding the resource. In this way, τ_b will execute its critical section with the priority of task τ_a. In general, τ_b inherits the highest priority among the tasks it blocks. Moreover, priority inheritance is transitive, thus if task τ_c blocks τ_b, which in turn blocks τ_a, then τ_c will inherit the priority of τ_a through τ_b.

Figure 5 illustrates how the schedule shown in Figure 4 is changed when resources are accessed using the Priority Inheritance Protocol. At time t_3, when τ_1 blocks on the semaphore (direct blocking), τ_3 inherits the maximum priority among the tasks blocked on that resource, thus it continues the execution of its critical section at the priority of τ_1. Under these conditions, at time t_4, task τ_2 is not able to preempt τ_3, hence it blocks until the resource is released (push-through blocking).

In other words, although τ_2 has a nominal priority greater than τ_3, it cannot interfere during the critical section, because τ_3 inherited the priority of τ_1. At time t_5, τ_3 exits its critical section, releases the semaphore and recovers its nominal priority. As a consequence, τ_1 can proceed until its completion, which occurs at time t_6. Only then, τ_2 can start executing.

The Priority Inheritance Protocol has the following property [21]: Given a task τ, if λ is the number of lower priority tasks sharing a resource with a task with priority higher than or equal to τ and σ is the number of semaphores that could block τ, then τ can be blocked for at most the duration of $min(\lambda, \sigma)$ critical sections.

Although the Priority Inheritance Protocol limits the priority inversion phenomenon, the maximum blocking time for high priority tasks can still be significant, due to possible chained blocking conditions. Moreover, deadlock can occur if semaphores are not properly used in nested critical sections.

4.2 Priority Ceiling Protocol

The Priority Ceiling Protocol [21] provides a better solution for the priority inversion phenomenon, also avoiding chained blocking and deadlock conditions. The basic idea behind this protocol is to ensure that, whenever a task τ enters a critical section, its priority is the highest among those that can be inherited from all the lower priority tasks that are currently suspended in a critical section. If this condition is not satisfied, τ is blocked and the task that is blocking τ inherits τ's priority. This idea is implemented by assigning each semaphore a *priority ceiling* equal to the highest priority of the tasks using that semaphore. Then, a task τ is allowed to enter a critical section only if its priority is strictly greater than all priority ceilings of the semaphores locked by the other tasks. As for the Priority Inheritance Protocol, the inheritance mechanism is transitive. The Priority Ceiling Protocol, besides avoiding chained blocking and deadlocks, has the property that each task can be blocked for at most the duration of a single critical section.

4.3 Schedulability Analysis

The importance of the protocols for accessing shared resources in a real-time system derives from the fact that they can bound the maximum blocking time experienced by a task. This is essential for analyzing the schedulability of a set of real-time tasks interacting through shared buffers or any other non-preemptable resource, e.g., a communication port or bus. To verify the schedulability of task τ_i using the processor utilization approach, we need to consider the utilization factor of task τ_i, the interference caused by the higher priority tasks and the blocking time caused by lower priority tasks. If B_i is the maximum blocking time that can be experienced by task τ_i, then the sum of the utilization factors due to these three causes cannot exceed the least upper bound of the scheduling algorithm, that is:

$$\forall i,\ 1 \leq i \leq n, \quad \sum_{k \in hp(i)} \frac{C_k}{T_k} + \frac{B_i}{T_i} \ \leq \ i(2^{1/i} - 1). \quad (5)$$

where $hp(i)$ denotes the set of tasks with priority higher than τ_i. The same test is valid for both the protocols described above, the only difference being the amount of blocking that each task may experience.

5 Asynchronous communication buffers

This section describes how blocking on shared resources can be avoided through the use of Cyclical Asynchronous Buffers [5], or CABs, a kind of wait free mechanism which allows tasks to exchange data without forcing a blocking synchronization.

In a CAB, read and write operations can be performed simultaneously without causing any blocking. Hence, a task can write a new message in a CAB while another task is reading the previous message. Mutual exclusion between reader and writer is avoided by means of memory duplication. In other words, if a task τ_W wants to write a new message into a CAB while a task τ_R is reading the current message, a new buffer is created, so that τ_W can write its message without interfering with τ_R. As τ_W finishes writing, its message becomes the most recent one in the CAB. To avoid blocking, the number of buffers that a CAB must handle has to be equal to the number of tasks that use the CAB plus one.

CABs were purposely designed for the cooperation among periodic activities running at different rates, such as control loops and sensory acquisition tasks. This approach was first proposed by Clark [8] for implementing a robotic application based on hierarchical servo-loops, and it is used in the SHARK kernel [11] as a basic communication support among periodic hard tasks.

In general, a CAB provides a one-to-many communication channel, which at any instant contains the latest message or data inserted in it. A message is not consumed by a receiving process, but is maintained into the CAB structure until a new message is overwritten. As a consequence, once the first message has been put in a CAB, a task can never be blocked during a receive operation. Similarly, since a new message overwrites the old one, a sender can never be blocked.

Notice that, using such a semantics, a message can be read more than once if the receiver is faster than the sender, while messages can be lost if the sender is faster than the receiver. However, this is not a problem in many control applications, where tasks are interested only in fresh sensory data rather than in the complete message history produced by a sensory acquisition task.

To insert a message in a CAB, a task must first reserve a buffer from the CAB memory space, then copy the message into the buffer, and finally put the buffer into the CAB structure, where it becomes the most recent message. This is done according to the following scheme:

```
buf_pointer = reserve(cab_id);
<copy message in *buf_pointer>
putmes(buf_pointer, cab_id);
```

Similarly, to get a message from a CAB, a task has to get the pointer to the most recent message, use the data, and then release the pointer. This is done according to the following scheme:

```
mes_pointer = getmes(cab_id);
<use message>
unget(mes_pointer, cab_id);
```

5.1 An example

To better illustrate the CAB mechanism, we describe an example in which a task τ_W writes messages in a CAB, and two tasks, τ_{R_1} and τ_{R_2}, read messages from the CAB. As it will be shown below, to avoid blocking and preserve data consistency, the CAB must contain 4 buffers. Consider the following sequence of events:

- At time t_1, task τ_W writes message M_1 in the CAB. When it finishes, M_1 becomes the most recent data (*mrd*) in the CAB.
- At time t_2, task τ_{R_1} asks the system to read the most recent data in the CAB and receives a pointer to M_1.
- At time t_3, task τ_W asks the system to write another message M_2 in the CAB, while τ_{R_1} is still reading M_1. Hence, a new buffer is reserved to τ_W. When it finishes, M_2 becomes the most recent data in the CAB.
- At time t_4, while τ_{R_1} is still reading M_1, τ_{R_2} asks the system to read the most recent data in the CAB and receives a pointer to M_2.
- At time t_5, while τ_{R_1} and τ_{R_2} are still reading, τ_W asks the system to write a new message M_3 in the CAB. Hence, a third buffer is reserved to τ_W. When it finishes, M_3 becomes the most recent data in the CAB.
- At time t_6, while τ_{R_1} and τ_{R_2} are still reading, τ_W asks the system to write a new message M_4 in the CAB. Notice that, in this situation, M_3 cannot be overwritten (being the most recent data), hence a fourth buffer must be reserved to τ_W. In fact, if M_3 is overwritten, τ_{R_1} could ask reading the CAB while τ_W is writing, thus finding the most recent data in an inconsistent state. When τ_W finishes writing M_4 into the fourth buffer, the *mrd* pointer is updated and the third buffer can be recycled if no task is accessing it.
- At time t_7, τ_{R_1} finishes reading M_1 and releases the first buffer (which can then be recycled).
- At time t_8, τ_{R_1} asks the system to read the most recent data in the CAB and receives a pointer to M_4.

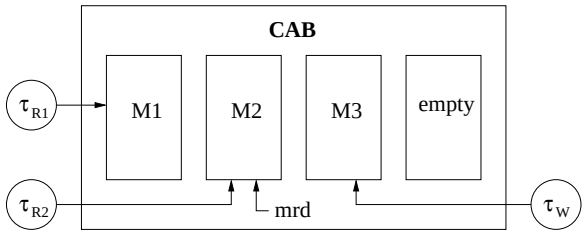

Figure 6. CAB configuration at time t_5.

Figure 6 illustrates the situation in the example, at time t_5, when τ_W is writing M_3 in the third buffer. Notice that at this time, the most recent data (mrd) is still M_2. It will be updated to M_3 only at the end of the write operation.

6 Interrupt handling

Interrupts generated by I/O peripheral devices may also introduce unpredictable delays in control task execution. In fact, in most operating systems, the arrival of an interrupt from an I/O device causes the immediate execution of a service routine (*device handler*) dedicated to the device management. In other words, a device driver always preempts an application task. This approach is motivated by the fact that, since I/O operations interact with the environment, they have real-time constraints, whereas most application programs do not. In the context of real-time control systems, however, this assumption is certainly not valid, because a control process could be more urgent than an interrupt handling routine. Since, in general, it is very difficult to bound a priori the number of interrupts that a task may experience, the delay introduced by the interrupt mechanism on tasks' execution becomes unpredictable.

To reduce the interference of the drivers on the application tasks and still perform I/O operations with the external world, the operating system should handle interrupts with appropriate aperiodic service mechanisms that can be analyzed to provide an a-priori guarantee of the real-time constraints [5]. In other words, interrupts should be handled as soon as possible, but without jeopardizing the schedulability of critical control activities.

A typical technique used in real-time systems to schedule aperiodic requests is based on the concept of aperiodic server, which is a kind of periodic task whose purpose is to serve aperiodic requests in a predictable fashion. Like a periodic task, a server is characterized by a period T_s and a budget C_s. In general, the server is scheduled with the same algorithm used for periodic tasks, and, once active, it serves the aperiodic requests within the limit of its budget. The order of service of the aperiodic requests is independent of the scheduling algorithm used for the periodic tasks, and it can be a function of the arrival time, computation time or deadline.

During the last years, several aperiodic service algorithms have been proposed in the real-time literature, differing in performance and complexity. Under fixed priority schemes, the most common algorithms are the Polling Server [14], the Deferrable Server [25], and the Sporadic Server [23]. Under dynamic priority schemes (which are more efficient in the average) we recall the Total Bandwidth Server [24] and the Constant Bandwidth Server [1].

7 Conclusions

This paper illustrated the importance of a suitable operating system support when executing complex control applications subject to real-time constraints. Among the most important features needed to execute control applications with high predictability, a real-time kernel should provide

1. a proper support for executing periodic tasks with explicit timing constraints;
2. an efficient scheduling algorithm to maximize the processor utilization and enable the user to analyze and verify the schedulability of the task set;
3. an runtime mechanism to detect execution overruns and deadline misses, to allow the application to react with proper recovery actions;
4. suitable concurrency control protocols to correctly access shared resources;
5. non blocking communication mechanisms for exchanging information among periodic tasks running at different rates; and
6. appropriate aperiodic service mechanisms to handle interrupt drivers in a controlled fashion and protect the execution of critical tasks.

Without these features, complex control applications could experience highly variable and unpredictable delays, which could significantly degrade the performance of the controlled system, or even jeopardize its stability.

Today, unfortunately, only a few research operating systems provide full real-time features able to cope with the problems presented in this paper. For example, VxWorks [26] and QNX-Neutrino [18], the most popular commercial real-time kernels, include the Priority Inheritance protocol for avoiding priority inversion, but do not have specific support for handling periodic tasks with explicit timing constraints, do not support deadline scheduling, do not have a time accounting mechanism for measuring the actual execution time of tasks (necessary to handle execution overruns

and overload conditions), and do not include advanced aperiodic service algorithms.

All these features are included into Shark [22], a free modular real-time kernel for PC platforms, developed at the Scuola Superiore Sant'Anna of Pisa. Each function (like scheduling, resource management policy, synchronization protocol, aperiodic service algorithm, etc.) is available as a software module that can be easily replaced, or combined with other modules, to build a custom kernel for specific real-time applications.

Other kernels providing some of the real-time features mentioned in this paper are MarteOS [20] and Erika Enterprise [10]. MarteOS is a real-time kernel for PC platforms developed at the University of Cantabria (Spain); it is mostly written in ADA, has a POSIX interface, and allows the user to specify the scheduler at the application level. Erika Enterprise is an OSEK compliant real-time kernel specifically designed for minimal embedded systems with limited onboard resources; it is fully configurable from a minimal memory footprint of 2 Kbytes, up to more complete configurations.

References

[1] L. Abeni and G. Buttazzo, "Resource Reservation in Dynamic Real-Time Systems," Real-Time Systems, Vol. 27, No. 2, pp. 123–167, July 2004.

[2] N.C. Audsley, A. Burns, M. Richardson, K. Tindell and A. Wellings, "Applying New Scheduling Theory to Static Priority Preemptive Scheduling", *Software Engineering Journal* 8(5), pp. 284-292, Sept. 1993.

[3] S. K. Baruah, R. R. Howell, and L. E. Rosier, "Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic Real-Time Tasks on One Processor," *Real-Time Systems*, 2, 1990.

[4] E. Bini, G.C. Buttazzo and G.M. Buttazzo, "Rate Monotonic Analysis: The Hyperbolic Bound," *IEEE Transactions on Computers*, 52(7), pp. 933–942, July 2003.

[5] G. Buttazzo. Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications - Second Edition, Springer, 2005.

[6] G. Buttazzo, "Rate Monotonic vs. EDF: Judgment Day", Real-Time Systems, Vol. 28, pp. 1-22, 2005.

[7] Anton Cervin, "Integrated Control and Real-Time Scheduling," Doctoral Dissertation, ISRN LUTFD2/TFRT-1065-SE, Department of Automatic Control, Lund, April 2003.

[8] D. Clark, "HIC: An Operating System for Hierarchies of Servo Loops," *Proceedings of IEEE International Conference on Robotics and Automation*, 1989.

[9] M.L. Dertouzos, "Control Robotics: the Procedural Control of Physical Processes," *Information Processing*, 74, North-Holland, Publishing Company, 1974.

[10] The Erika Enterprise kernel, Evidence s.r.l., URL: http://www.evidence.eu.com/Erika.asp

[11] P. Gai, L. Abeni, M. Giorgi, G. Buttazzo, "A New Kernel Approach for Modular Real-Time Systems Development," *IEEE Proc. of the 13th Euromicro Conference on Real-Time Systems*, Delft (NL), June 2001.

[12] T. M. Ghazalie and T. P. Baker, "Aperiodic Servers In A Deadline Scheduling Environment," *Real-Time Systems*, 1995.

[13] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal*, 29(5), pp. 390-395, 1986.

[14] J. P. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 261-270, 1987.

[15] J. P. Lehoczky, L. Sha, and Y. Ding, "The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour", *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 166-171, 1989.

[16] J. Leung, and J. Whitehead, "On the Complexity of Fixed Priority Scheduling of Periodic Real-Time Tasks," *Performance Evaluation*, 2(4), pp. 237-250, 1982.

[17] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard real-Time Environment," *Journal of the ACM* 20(1), 1973, pp. 40–61.

[18] QNX Neutrino RTOS, QNX Software Systems, URL: http://www.qnx.com.

[19] R. Rajkumar, Synchronous Programming of Reactive Systems, Kluwer Academic Publishing, 1991.

[20] M. Aldea Rivas and M. Gonzlez Harbour, "MaRTE OS: An Ada Kernel for Real-Time Embedded Applications," Proc. of the 6th International Conference on Reliable Software Technologies (Ada-Europe 2001), Leuven, Belgium, May 2001.

[21] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, 39(9), pp. 1175-1185, 1990.

[22] The Shark real-time kernel. URL: http://shark.sssup.it/

[23] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time System," *Journal of Real-Time Systems*, 1, pp. 27-60, June 1989.

[24] M. Spuri and G.C. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems," *Real-Time Systems*, 10(2), 1996.

[25] J.K. Strosnider, J.P. Lehoczky, and L. Sha, "The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *IEEE Transactions on Computers*, 44(1), Jan. 1995.

[26] VxWorks Real-Time Operating System, Wind River Corp., URL: http://www.windriver.com/vxworks.