In Proceedings of the 7th Int. Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 2002), Oldenburg, Germany, September 2002. Vol. 2469 of Lecture Notes in Computer Science, Springer-Verlag, pp. 37-51, 2002.

Real-Time Operating Systems: Problems and Novel Solutions

Giorgio Buttazzo

University of Pavia
buttazzo@unipv.it

Abstract. This work presents some methodologies for enhancing predictability in real-time computing systems, where explicit timing constraints have to be enforced on application processes. In order to provide an off-line guarantee of the critical timing constraints, deterministic and analyzable algorithms are required in all kernel mechanisms, especially involving scheduling, inter-task communication, synchronization and interrupt handling. This paper illustrates some problems that may arise in real-time concurrent applications and some solutions that can be adopted in the kernel to overcome those problems. In particular, task scheduling algorithms and resource management policies will be considered in detail, as they have great influence on system behavior. Finally, a novel approach will be introduced for handling transient overloads and execution overruns in soft real-time systems working in dynamic environments. These techniques provide efficient support to real-time multimedia systems.

1. Introduction

Often, people say that real-time systems must react fast to external events. Such a definition, however, is not precise, because processing speed does not provide any information on the actual capability of the system to react timely to events. In fact, the effect of controller actions in a system can only be evaluated when considering the dynamic characteristics of the controlled environment.

A more precise definition would say that a real-time system is a system in which performance depends not only on the correctness of the single controller actions, but also on the time at which actions are produced [24]. The main difference between a real-time task and a non real-time task is that a real-time task must complete within a given *deadline*. In other words, a deadline is the maximum time allowed for a computational process to finish its execution. In real-time applications, a result produced after its deadline is not only late, but can be dangerous. Depending on the consequences caused by a missed deadline, real-time activities can be classified in *hard* and *soft* tasks [23]. A real-time task is said to be *hard* if missing a deadline may have catastrophic consequences in the controlled system. A real-time task is said to be *soft* if missing a deadline causes a performance degradation, but does not jeopardize correct system behavior. An operating system able to manage hard tasks is called a *hard real-time system* [4][25].

In general, hard real-time systems have to handle both hard and soft activities. In a control application, typical hard tasks include sensory data acquisition, detection of critical conditions, motor actuation, and action planning. Typical soft tasks include user command interpretation, keyboard input, message visualization, system status representation, and graphical activities. The great interest in real-time systems is motivated by the growing diffusion they have in our society in several application fields, including chemical and nuclear power plants, flight control systems, traffic monitoring systems, telecommunication systems, automotive devices, industrial automation, military systems, space missions, and robotic systems.

Despite this large application domain, most of today's real-time control systems are still designed using ad hoc techniques and heuristic approaches. Very often, control applications with stringent time constraints are implemented by writing large portions of code in assembly language, programming timers, writing low-level drivers for device handling, and manipulating task and interrupt priorities. Although the code produced by these techniques can be optimized to run very efficiently, this approach has several disadvantages. First of all, the implementation of large and complex applications in assembly language is much more difficult and time consuming than using high-level programming. Moreover, the efficiency of the code strongly depends on the programmer's ability. In addition, assembly code optimization makes a program more difficult to comprehend, complicating software maintenance. Finally, without the support of specific tools and methodologies for code and schedulability analysis, the verification of time constraints becomes practically impossible.

The major consequence of this state of affairs is that control software produced by empirical techniques can be highly unpredictable. If all critical time constraints cannot be verified a priori and the operating system does not include specific features for handling real-time tasks, the system apparently works well for a period of time, but may collapse in certain rare, but possible, situations. The consequences of a failure can sometimes be catastrophic and may injure people or cause serious damage to the environment. A trustworthy guarantee of system behavior under all possible operating conditions can only be achieved by adopting appropriate design methodologies and kernel mechanisms specifically developed for handling explicit timing constraints.

1.1 Achieving predictability

The most important property of a real-time system is not high speed, but predictability. In a predictable system we should be able to determine in advance whether all the computational activities can be completed within their timing constraints. The deterministic behavior of a system typically depends on several factors, ranging from the hardware architecture to the operating system, up to the programming language used to write the application.

Architectural features that have major influence on task execution include interrupts, DMA, cache and pre-fetching mechanisms. Although such features improve the average performance of the processor, they introduce a non deterministic behavior in process execution, prolonging the worst-case response times. Other factors that significantly affect task execution are due to the internal mechanisms used in the operating system, such as the scheduling algorithm, the synchronization

mechanisms, the memory management policy, and the method used to handle I/O devices. The programming language has also an important impact on predictability, through the constructs it provides to handle the timing requirements specified for computational activities.

2. Periodic task handling

Periodic activities represent the major computational load in a real-time control system. For example activities such as actuator regulation, 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 characterized by an infinite sequence of *instances*, or *jobs*. Each job is characterized by a *request time* and a *deadline*. The request time *r(k)* of the *k*-th job of a task represents the time at which the task becomes ready for execution for the *k*-th time. The interval of time between two consecutive request times is equal to the task period. The absolute deadline of the *k*-th job, denoted with *d(k)*, represents the time within which the job has to complete its execution, and $r(k) < d(k) \leq r(k+1)$.

2.1 Timeline scheduling

Timeline Scheduling (TS), also known as a *cyclic executive*, is one of the most used approaches to handle periodic tasks in defense military systems and traffic control systems. The method consists in dividing the temporal axis into *slices* of equal length, in which one or more tasks can be allocated for execution, in such a way to respect the frequencies derived from the application requirements. A timer synchronizes the activation of the tasks at the beginning of each time slice. In order to illustrate this method, consider the following example, in which three tasks, A, B and C, need to be executed with a frequency of 40, 20 and 10 Hz, respectively. By analyzing the task periods, it is easy to verify that the optimal length for the time slice is 25 ms, which is the Greatest Common Divisor of the periods. Hence, to meet the required frequencies, task A needs to be executed every time slice, task B every two slices, and task C every four slices. A possible scheduling solution for this task set is illustrated in Figure 1.

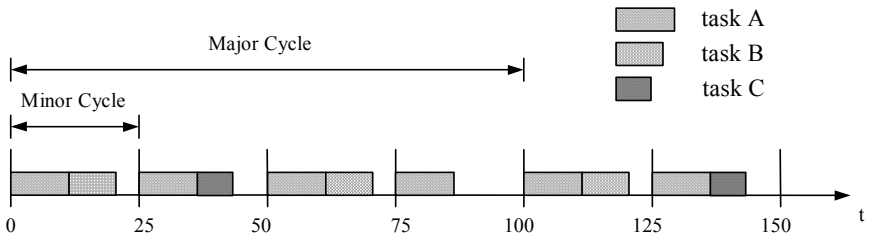

Fig. 1. Example of timeline scheduling

The duration of the time slice is also called a *Minor Cycle*, whereas the minimum period after which the schedule repeats itself is called a *Major Cycle*. In general, the major cycle is equal to the least common multiple of all the periods (in the example it is equal to 100 ms).

In order to guarantee a priori that a schedule is feasible on a particular processor, it is sufficient to know the task worst-case execution times and verify that the sum of the executions within each time slice is less than or equal to the minor cycle. In the example shown in Figure 1, if C_A, C_B and C_C denote the execution times of the tasks, it is sufficient to verify that

$$C_A + C_B \leq 25 \text{ ms}$$

$$C_A + C_C \leq 25 \text{ ms}$$

The major relevant advantage of timeline scheduling is its simplicity. The method can be implemented by programming a timer to interrupt with a period equal to the minor cycle and by writing a main program that calls the tasks in the order given in the major cycle, inserting a time synchronization point at the beginning of each minor cycle. Since the task sequence is not decided by a scheduling algorithm in the kernel, but it is triggered by the calls made by the main program, there are no context switches, so the runtime overhead is very low. Moreover, the sequence of tasks in the schedule is always the same, can be easily visualized, and it is not affected by jitter (i.e., task start times and response times are not subject to large variations).

In spite of these advantages, timeline scheduling has some problems. For example, it is very fragile during overload conditions. If a task does not terminate at the minor cycle boundary, we can either let it continue or abort it. In both cases, however, the system may enter in a risky situation. In fact, if we leave the failing task in execution, it can cause a domino effect on the other tasks, breaking the entire schedule (*timeline break*). On the other hand, if the failing task is aborted, the system may be left in an inconsistent state, jeopardizing correct system behavior.

Another big problem of the timeline scheduling technique is its sensitivity to application changes. If updating a task requires an increase of its computation time or its activation frequency, the entire scheduling sequence may need to be reconstructed from scratch. Considering the previous example, if task B is updated to B' and the code change is such that $C_A + C_{B'} > 25$ ms, then we have to divide B' in two or more pieces to be allocated in the available intervals of the timeline. Changing the task frequencies may cause even more radical changes in the schedule. For example, if the frequency of task B changes from 20 Hz to 25 Hz, the previous schedule is not valid any more, because the new Minor Cycle is equal to 10 ms and the new Major Cycle is equal to 200 ms.

Finally, another limitation of the timeline scheduling is that it is difficult to handle aperiodic activities efficiently without changing the task sequence.

The problems outlined above can be solved by using priority based scheduling algorithms.

2.2 Rate Monotonic (RM)

The *Rate-Monotonic* (RM) algorithm assigns each task a priority directly proportional to its activation frequency, so that tasks with shorter period have higher priority. Since a period is usually kept constant for a task, the RM algorithm implements a static priority assignment, in the sense that task priorities are decided at task creation and remain unchanged for the entire application run. RM is typically preemptive, although it can also be used in a non-preemptive mode.

In 1973, Liu and Layland [17] showed that RM is optimal among all static scheduling algorithms, in the sense that if a task set is not schedulable by RM, then the task set cannot be feasibly scheduled by any other fixed priority assignment. Another important result proved by the same authors is that a set $\Gamma = \{\tau_1, \ldots, \tau_n\}$ of n periodic tasks is schedulable by RM if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right)$$

where C_i and T_i represent the worst-case computation time and the period of task τ_i, respectively. The quantity

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

represents the *processor utilization factor* and denotes the fraction of time used by the processor to execute the entire task set. Table 1 shows the values of $n(2^{1/n} - 1)$ for n from 1 to 10. As can be seen, the factor decreases with n and, for large n, it tends to the following limit value:

$$\lim_{n \to \infty} n\left(2^{1/n} - 1\right) = \ln 2 \cong 0.69$$

Table 1. Maximum processor utilization for the Rate Monotonic algorithm

n	$n(2^{1/n} - 1)$
1	1.000
2	0.828
3	0.780
4	0.757
5	0.743
6	0.735
7	0.729
8	0.724
9	0.721
10	0.718

We note that the Liu and Layland test only gives a sufficient condition for guaranteeing a feasible schedule under the RM algorithm. Hence, a task set can be schedulable by RM even though the utilization condition is not satisfied. Nevertheless, we can certainly state that a periodic task set cannot be feasibly scheduled by any algorithm if $U > 1$. A statistical study carried out by Lehoczky, Sha, and Ding [14] on randomly generated task sets showed that the utilization bound of the RM algorithm has an average value of 0.88, and becomes 1 for periodic tasks with harmonic period relations.

In spite of the limitation on the schedulability bound, which in most cases prevents the full processor utilization, the RM algorithm is widely used in real-time applications, manly for its simplicity. At the same time, being a static scheduling algorithm, it can be easily implemented on top of commercial operating systems, using a set of fixed priority levels. Moreover, in overload conditions, the highest priority tasks are less prone to missing their deadlines. For all these reasons, the Software Engineering Institute of Pittsburgh has prepared a sort of user guide for the design and analysis of real-time systems based on the RM algorithm [11].

Since the RM algorithm is optimal among all fixed priority assignments, the schedulability bound can only be improved through a dynamic priority assignment.

2.3 Earliest Deadline First (EDF)

The *Earliest Deadline First* (EDF) algorithm consists in selecting (among the ready tasks) the task with the earliest absolute deadline. The EDF algorithm is typically *preemptive*, in the sense that, a newly arrived task can preempt the running task if its absolute deadline is shorter.

If the operating system does not support explicit timing constraints, EDF (as RM) can be implemented on a priority-based kernel, where priorities are dynamically assigned to tasks. A task will receive the highest priority if its deadline is the earliest among those of the ready tasks, whereas it will receive the lowest priority if its deadline is the latest one. A task gets a priority which is inversely proportional to its absolute deadline.

The EDF algorithm is more general than RM, since it can be used to schedule both periodic and aperiodic task sets, because the selection of a task is based on the value of its absolute deadline, which can be defined for both types of tasks. Typically, a periodic task that completed its execution is suspended by the kernel until its next release, coincident with the end of the current period. Dertouzos [8] showed that EDF is optimal among all on line algorithms, while Liu and Layland [17] proved that a set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n periodic tasks is schedulable by EDF *if and only if*

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

It is worth noting that the EDF schedulability condition is necessary and sufficient to guarantee a feasible schedule. This mean that, if it is not satisfied, no algorithm is able to produce a feasible schedule for that task set.

The dynamic priority assignment allows EDF to exploit the full processor, reaching up to 100% of the available processing time. When the task set has a processor utilization factor 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 context switches, thus causing less runtime overhead. On the other hand, RM is simpler to implement on a fixed priority kernel and is more predictable in overload situations, because higher priority tasks are less viable to miss their deadlines.

2.4 Tasks with deadlines less than periods

Using RM or EDF, a periodic task can be executed at any time during its period. The only guarantee provided by the schedulability test is that each task will be able to complete its execution before the next release time. In some real-time applications, however, there is the need for some periodic task to complete within an interval less than its period.

The *Deadline Monotonic* (DM) algorithm, proposed by Leung and Whitehead [16], extends RM to handle tasks with a relative deadline less than or equal to their period. According to DM, at each instant the processor is assigned to the task with the shortest relative deadline. In priority-based kernels, this is equivalent to assigning each task a priority $P_i \propto 1/D_i$ inversely proportional to its relative deadline.
With D_i fixed for each task, DM is classified as a static scheduling algorithm. In the recent years, several authors [2][10][14] independently proposed a necessary and sufficient test to verify the schedulability of a periodic task set. For example, the method proposed by Audsley et al. [2] consists in computing the worst-case response time R_i of each periodic task. It is derived by summing its computation time and the interference caused by tasks with higher priority:

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

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*, proposed by Baruah, Howell, and Rosier [3]. According to this method, a task set is schedulable by EDF if and only if, in every interval of length L (starting at time 0), the overall computational demand is no greater than the available processing time, that is, if and only if

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

This test is feasible, because L can only be checked for values equal to task deadlines no larger than the least common multiple of the periods.

3. Aperiodic task handling

Although in a real-time system most acquisition and control tasks are periodic, there exist computational activities that must be executed only at the occurrence of external events (typically signalled through interrupts), which may arrive at irregular times. When the system must handle aperiodic requests of computation, we have to balance two conflicting interests: on the one hand, we would like to serve an event as soon as possible to improve system responsiveness; on the other hand, we do not want to jeopardize the schedulability of periodic tasks.

If aperiodic activities are less critical than periodic tasks, then the objective of a scheduling algorithm should be to minimize their response time, while guaranteeing that all periodic tasks (although being delayed by the aperiodic service) complete their executions within their deadlines. If some aperiodic task has a hard deadline, we should try to guarantee its timely completion off-line. Such a guarantee can only be done by assuming that aperiodic requests, although arriving at irregular intervals, do not exceed a maximum given frequency, that is, they are separated by a *minimum interarrival time*. An aperiodic task characterized by a minimum interarrival time is called a *sporadic task*.

Let us consider an example in which an aperiodic job J_a of 3 units of time must be scheduled by RM along with two periodic tasks, having computation times $C_1 = 1$, $C_2 = 3$ and periods $T_1 = 4$, $T_2 = 6$, respectively. As shown in Figure 2, if the aperiodic request is serviced immediately (that is, with a priority higher than that assigned to periodic tasks), then task τ_2 will miss its deadline.

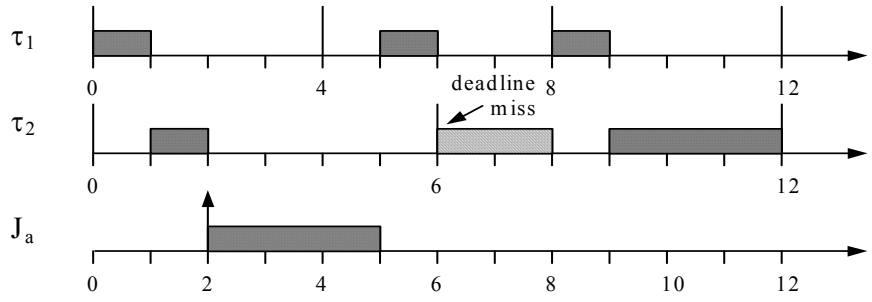

Fig. 2. Immediate service of an aperiodic task. Periodic tasks are scheduled by RM

The simplest technique for managing aperiodic activities while preserving the guarantee for periodic tasks is to schedule them in background. This means that an aperiodic task executes only when the processor is not busy with periodic tasks. The disadvantage of this solution is that, if the computational load due to periodic tasks is high, the residual time left for aperiodic execution can be insufficient for satisfying their deadlines.

Considering the same task set as before, Figure 3 illustrates how job J_a is handled by a background service.

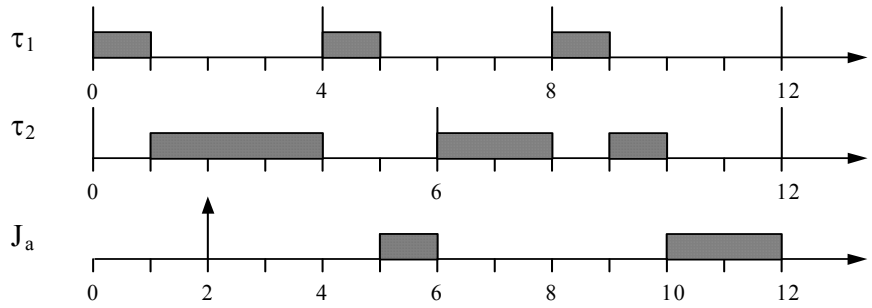

Fig. 3. Background service of an aperiodic task. Periodic tasks are scheduled by RM

The response time of aperiodic tasks can be improved by handling them through a periodic *server* dedicated to their execution. As any other periodic task, a server is characterized by a period T_s and an execution time C_s, called the server *capacity* (or *budget*).

In general, the server is scheduled using the algorithm adopted for periodic tasks and, once activated, it starts serving the pending aperiodic requests within the limit of its current capacity. 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. Among the fixed priority algorithms we mention the *Polling Server* and the *Deferrable Server* [13][27], the *Sporadic Server* [20], and the *Slack Stealer* [15]. Among those servers using dynamic priorities (which are more efficient on the average) we recall the *Dynamic Sporadic Server* [9][21], the *Total Bandwidth Server* [22], the *Tunable Bandwidth Server* [5], and the *Constant Bandwidth Server* [1].

In order to clarify the idea behind an aperiodic server, Figure 4 illustrates the schedule produced, under EDF, by a *Dynamic Deferrable Server* with capacity $C_s = 1$ and period $T_s = 4$. We note that, when the absolute deadline of the server is equal to the one of a periodic task, priority is given to the server in order to enhance aperiodic responsiveness. We also observe that the same task set would not be schedulable under a fixed priority system.

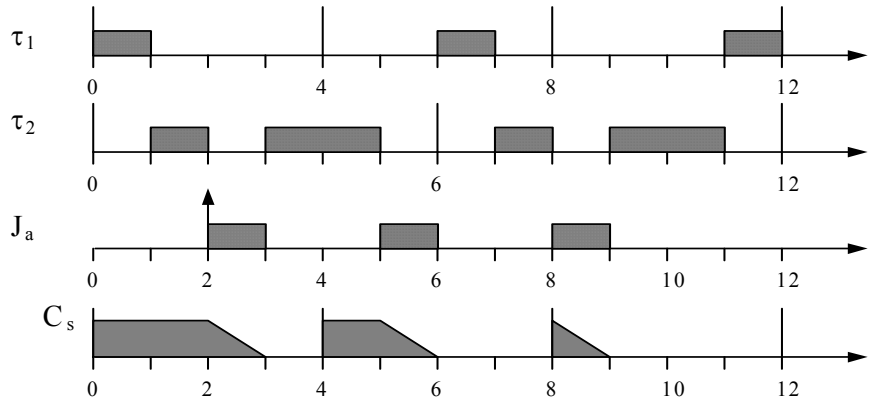

Fig. 4. Aperiodic service performed by a Dynamic Deferrable Server. Periodic tasks, including the server, are scheduled by EDF. C_s is the remaining budget available for J_a

Although the response time achieved by a server is less than that achieved through the background service, it is not the minimum possible. The minimum response time can be obtained with an optimal server (TB*) which assigns each aperiodic request the earliest possible deadline which still produces a feasible EDF schedule [5]. The schedule generated by the optimal TB* algorithm is illustrated in Figure 5, where the minimum response time for job J_a is equal to 5 units of time (obtained by assigning the job a deadline $d_a = 7$).

As for all the efficient solutions, the better performance is achieved at the price of a larger runtime overhead (due to the complexity of computing the minimum deadline). However, adopting a variant of the algorithm, called the *Tunable Bandwidth Server* [5], overhead cost and performance can be balanced in order to select the best service method for a given real-time system. An overview of the most common aperidic service algorithms (both under fixed and dynamic priorities) can be found in [4].

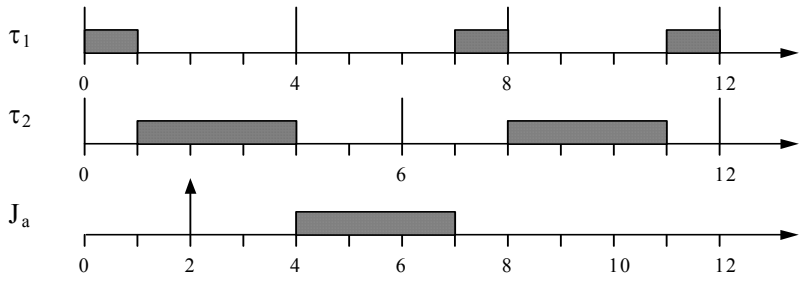

Fig. 5. Optimal aperiodic service under EDF

4. Protocols for accessing shared resources

When two or more tasks interact through shared resources (e.g., shared memory buffers), the direct use of classical synchronization mechanisms, such as semaphores or monitors, can cause a phenomenon known as *priority inversion*: a high priority task can be blocked by a low priority task for an unbounded interval of time. Such a blocking condition can create serious problems in safety critical real-time systems, since it can cause deadlines to be missed.

For example, consider three tasks, τ_1, τ_2 and τ_3, having decreasing priority (τ_1 is the task with highest priority), and assume that τ_1 and τ_3 share a data structure protected by a binary semaphore S. As shown in Figure 6, suppose that at time t_l task τ_3 enters its critical section, holding semaphore S. During the execution of τ_3, at time t_2, assume τ_1 becomes ready and preempts τ_3.

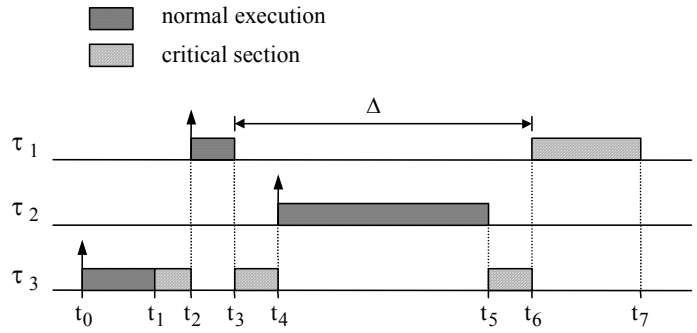

Fig. 6. Example of priority inversion

At time t_3, when τ_1 tries to access the shared resource, it is blocked on semaphore S, since the resource is used by τ_3. Since τ_1 is the highest priority task, we would expect it to be 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 become much larger. In fact, task τ_3, while holding the resource, can be preempted by medium priority tasks (like τ_2), which will prolong the blocking interval of τ_1 for their entire execution!

The situation illustrated in Figure 6 can be avoided by simply preventing preemption inside critical sections. This solution, however, is appropriate only for very short critical sections, because it could cause unnecessary delays for high priority tasks. For example, a low priority task inside a long critical section would prevent the execution of a high priority task, even though they do not share any resource.

A more efficient solution is to regulate the access to shared resource through the use of specific concurrency control protocols, designed to limit the priority inversion phenomenon.

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) [19]. 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 7 illustrates how the schedule shown in Figure 6 is changed when resources are accessed using the Priority Inheritance Protocol. Until time t_3 the system evolution is the same as the one shown in Figure 6. At time t_3, the high priority task τ_1 blocks after attempting to enter the resource held by τ_3 (*direct blocking*). In this case, however, the protocol imposes that 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 execute, 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.

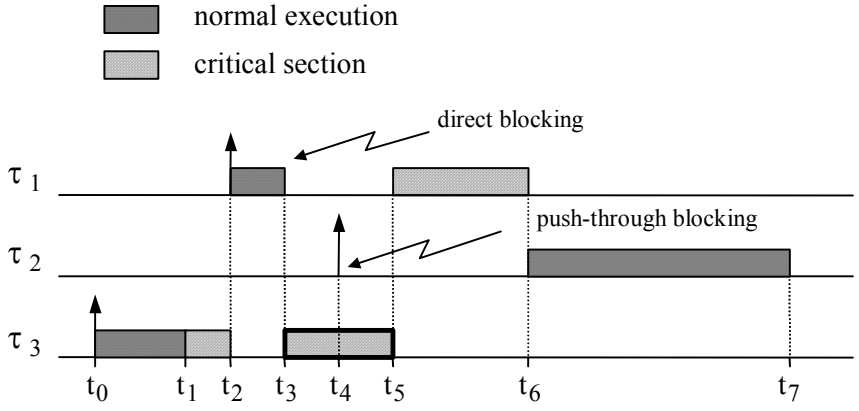

Fig. 7. Schedule produced using Priority Inheritance on the task set of Figure 6

The Priority Inheritance Protocol has the following property [19]:

> Given a task τ, if *n* is the number of tasks with lower priority sharing a resource with a task with priority higher or equal to τ and *m* is the number of semaphores that could block τ, then τ can be blocked for at most the duration of *min(n,m)* 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* (PCP) [19] 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 held 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:

$$\forall i = 1,\ldots,n \qquad \frac{C_i}{T_i} + \sum_{k \in hp(i)} \frac{C_k}{T_k} + \frac{B_i}{T_i} \leq i\left(2^{1/i} - 1\right)$$

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. New applications and trends

In the last years, real-time system technology has been applied to several application domains, where computational activities have less stringent timing constraints and occasional deadline misses are typically tolerated. Examples of such systems include monitoring, multimedia systems, flight simulators and, in general, virtual reality games. In such applications, missing a deadline does not cause catastrophic effects on the system, but just a performance degradation. Hence, instead of requiring an absolute guarantee for the feasibility of the schedule, such systems demand an acceptable *Quality of Service* (QoS). It is worth observing that, since some timing constraints need to be handled anyway (although not critical), a non real-time operating system, such a Linux or Windows, is not appropriate: First of all, such systems do not provide temporal isolation among tasks, thus a sporadic peak load on a task may negatively affect the execution of other tasks in the system. Furthermore, the lack of concurrency control mechanisms which prevent priority inversion makes these systems unsuitable for guaranteeing a desired QoS level.

On the other hand, a hard real-time approach is also not well suited for supporting such applications, because resources would be wasted due to static allocation mechanisms and pessimistic design assumptions. Moreover, in many multimedia applications, tasks are characterized by highly variable execution times (consider, for instance, an mpeg player), thus providing precise estimations on task computation times is practically impossible, unless one uses overly pessimistic figures.

In order to provide efficient as well as predictable support for this type of real-time applications, several new approaches and scheduling methodologies have been proposed. They increase the flexibility and the adaptability of a system to on-line variations. For example, temporal protection mechanisms have been proposed to isolate task overruns and reduce reciprocal task interference [1][26]. Statistical analysis techniques have been introduced to provide a probabilistic guarantee aimed at improving system efficiency [1].

Other techniques have been devised to handle transient and permanent overload conditions in a controlled fashion, thus increasing the average computational load in the system. One method absorbs the overload by regularly aborting some jobs of a periodic task, without exceeding a maximum limit specified by the user through a QoS parameter describing the minimum number of jobs between two consecutive abortions [7][12]. Another technique handles overloads through a suitable variation of periods, managed to decreased the processor utilization up to a desired level [6].

6. Conclusions

This paper surveyed some kernel methodologies aimed at enhancing the efficiency and the predictability of real-time control applications. In particular, the paper presented some scheduling algorithms and analysis techniques for periodic and aperiodic task sets. Two concurrency control protocols have been described to access shared resources in mutual exclusion while avoiding the priority inversion phenomenon. Each technique has the property to be analyzable, so that an off-line guarantee can be provided for feasibility of the schedule within the timing constraints imposed by the application.

For soft real-time systems, such as multimedia systems or simulators, the hard real-time approach can be too rigid and inefficient, especially when the application tasks have highly variable computation times. In these cases, novel methodologies have been introduced to improve average resource exploitation. They are also able to guarantee a desired QoS level and control performance degradation during overload conditions.

In addition to research efforts aimed at providing solutions to more complex problems, a concrete increase in the reliability of future real-time systems can only be achieved if the mature methodologies are actually integrated in next generation operating systems and languages, defining new standards for the development of real-time applications. At the same time, programmers and software engineers need to be educated to the appropriate use of the available technologies.

References

1. Abeni, L., and G. Buttazzo: "Integrating Multimedia Applications in Hard Real-Time Systems", Proceedings of the *IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

2. Audsley, N. C., A. Burns, M. Richardson, and A. Wellings: "Hard Real-Time Scheduling: The Deadline Monotonic Approach", *IEEE Workshop on Real-Time Operating Systems*, 1992.

3. Baruah, S. K., 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. Buttazzo, G. C.: *HARD REAL-TIME COMPUTING SYSTEMS: Predictable Scheduling Algorithms and Applications*, Kluwer Academic Publishers, Boston, 1997.

5. Buttazzo, G. C. and F. Sensini: "Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments", *3rd IEEE International Conference on Engineering of Complex Computer Systems* (ICECCS), Como, Italy, September 1997.

6. Buttazzo, G. C., G. Lipari, and L. Abeni: “Elastic Task Model for Adaptive Rate Control”, Proceedings of the *IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

7. Buttazzo, G. C., and M. Caccamo: “Minimizing Aperiodic Response Times in a Firm Real-Time Environment”, *IEEE Transactions on Software Engineering*, Vol. 25, No. 1, pp. 22-32, January/February 1999.

8. Dertouzos, M. L.: “Control Robotics: the Procedural Control of Physical Processes”, *Information Processing 74*, North-Holland Publishing Company, 1974.

9. Ghazalie, T. M. and T. P. Baker: “Aperiodic Servers In A Deadline Scheduling Environment”. *The Journal of Real-Time Systems*, 1995.

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

11. Klein, M.H., et al.: *A Practitioners’ Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Boston, MA: Kluwer Academic Publishers, 1993.

12. Koren, G., and D. Shasha: “Skip-Over: Algorithms and Complexity for Overloaded Systems that Allow Skips”, *IEEE Real-Time System Symposium*, December 1995.

13. Lehoczky, J. P., L. Sha, and J. K. Strosnider: “Enhanced Aperiodic Responsiveness in Hard Real-Time Environments”, *IEEE Real-Time Systems Symposium*, pp. 261-270, San Jose, CA, December 1987.

14. Lehoczky, J. P., L. Sha, and Y. Ding: “The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour”, *IEEE Real-Time Systems Symposium*, pp. 166-171, 1989.

15. Lehoczky, J. P., and S. Ramos-Thuel: “An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems”, *IEEE Real-Time Systems Symposium*, 1992.

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

17. Liu, C. L., and J. W. Layland: “Scheduling Algoritms for Multiprogramming in a Hard-Real-Time Environment”, *Journal of ACM*, Vol. 20, No. 1, January 1973.

18. Rajkumar, R.: *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishing, 1991.

19. Sha, L., R. Rajkumar, and J. P. Lehoczky: “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”, *IEEE Transactions on Computers*, Vol. 39, No. 9, September 1990.

20. Sprunt, B., L. Sha, and J. Lehoczky: “Aperiodic Task Scheduling for Hard Real-Time System”, *Journal of Real-Time Systems*, 1, pp. 27-60, June 1989.

21. Spuri, M., and G. C. Buttazzo: “Efficient Aperiodic Service under Earliest Deadline Scheduling”, *15th IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, 1994.

22. Spuri, M., and G. C. Buttazzo: “Scheduling Aperiodic Tasks in Dynamic Priority Systems”, *Journal of Real-Time Systems*, Vol. 10, No. 2, pp. 1-32, 1996.

23. Stankovic, J., and K. Ramamritham: *Tutorial on Hard Real-Time Systems*, IEEE Computer Society Press, 1988.

24. Stankovic, J.: “A Serious Problem for Next-Generation Systems”, *IEEE Computer*, pp. 10-19, October 1988.

25. Stankovic, J., M. Spuri, M. Di Natale, G. Buttazzo: “Implications of Classical Scheduling Results for Real-Time Systems”, *IEEE Computer*, Vol. 28, No. 6, pp. 16-25, June 1995.

26. Stoica, I., H-Abdel-Wahab, K. Jeffay, S. Baruah, J.E. Gehrke, and G. C. Plaxton: “A Proportional Share Resource Allocation Algorithm for Real-Time Timeshared Systems”, *IEEE Real-Time Systems Symposium*, Dec. 1996

27. Strosnider, J. K., J. P. Lehoczky and L. Sha: “The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments”, *IEEE Transactions on Computers*, Vol. 44, No. 1, pp. 73-91, January 1995.