1

INTRODUCTION

In this chapter we explain the reasons why soft real-time computing is being deeply investigated during the last years for supporting a set of application domains for which the hard real-time approach is not suited. Examples of such application domains include multimedia systems, monitoring apparatuses, robotic systems, real-time graphics, interactive games, and virtual reality.

To better understand the difference between classical hard real-time applications and soft real-time applications, we first introduce some basic terminology that will be used throughout the book, then we present the classical design approach used for hard real-time systems, and then describe the characteristics of some soft real-time application. Hence, we identify the major problems that a hard real-time approach can cause in these systems and finally we derives a set of features that a soft real-time system should have in order to provide efficient support for these kind of applications.

1.1 BASIC TERMINOLOGY

In the common sense, a real-time system is a system that reacts to an event within a limited amount of time. So, for example, in a web page reporting the state of a Formula 1 race, we say that the race state is reported in real-time if the car positions are updated "as soon as" there is a change. In this particular case, the expression "as soon as" does not have a precise meaning and typically refers to intervals of a few seconds.

When a computer is used to control a physical device (e.g., a mobile robot), the time needed by the processor to react to events in the environment may significantly affect the overall system's performance. In the example of a mobile robot system, a correct maneuver performed too late could cause serious problems to the system and/or the

environment. For instance, if the robot is running at a certain speed and an obstacle is detected along the robot path, the action of pressing the brakes or changing the robot trajectory should be performed within a maximum delay (which depends on the obstacle distance and on the robot speed), otherwise the robot could not be able to avoid the obstacle, thus incurring in a crash.

Keeping the previous example in mind, a real-time system can be more precisely defined as a computing system in which computational activities must be performed within predefined timing constraints. Hence, the performance of a real-time system depends not only on the functional correctness of the results of computations, but also on the time at which such results are produced.

The word *real* indicates that the system time (that is, the time represented inside the computing system) should always be synchronized with the external time reference with which all time intervals in the environment are measured.

1.1.1 TIMING PARAMETERS

A real-time system is usually modeled as a set of concurrent *tasks*. Each task represents a computational activity that needs to be performed according to a set of constraints. The most significant timing parameters that are typically defined on a real-time computational activity are listed below.

- **Release time** r_i: is the time at which a task becomes ready for execution; it is also referred to as *arrival time* and denoted by a_i;
- **Start time** s_i: is the time at which a task starts its execution for the first time;
- **Computation time** C_i: is the time necessary to the processor for executing the task without interruption;
- **Finishing time** f_i: is the time at which a task finishes its execution;
- **Response time** R_i: is the time elapsed from the task release time and its finishing time ($R_i = f_i - r_i$);
- **Absolute deadline** d_i: is the time before which a task should be completed;
- **Relative deadline** D_i: is the time, relative to the release time, before which a task should be completed ($D_i = d_i - r_i$);

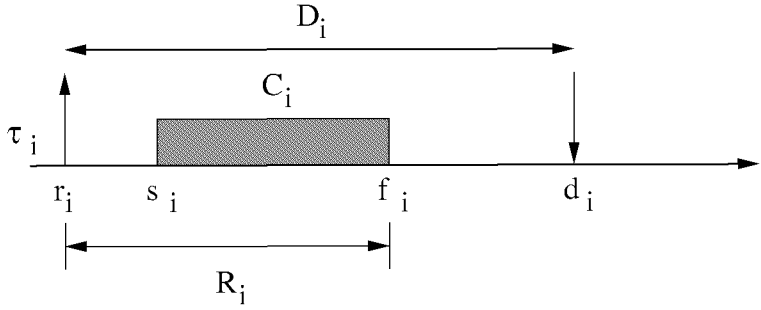

Figure 1.1 Typical timing parameters of a real-time task.

Such parameters are schematically illustrated in Figure 1.1, where the release time is represented by an up arrow and the absolute deadline is represented by a down arrow.

Other parameters that are usually defined on a task are:

- **Slack time** or **Laxity**: denotes the interval between the finishing time and the absolute deadline of a task ($slack_i = d_i - f_i$); it represents the maximum time a task can be delayed to still finish within its deadline;
- **Lateness** L_i: $L_i = f_i - d_i$ represents the completion delay of a task with respect to its deadline; note that if a task completes before its deadline, its lateness is negative;
- **Tardiness** or *Exceeding time* E_i: $E_i = max(0, L_i)$ is the time a task stays active after its deadline.

If the same computational activity needs to be executed several times on different data, then a task is characterized as a sequence of multiple instances, or *jobs*. In general, a task τ_i is modeled as a (finite or infinite) stream of jobs, $\tau_{i,j}$, $(j = 1, 2, \ldots)$, each characterized by a release time $r_{i,j}$, an execution time $c_{i,j}$, a finishing time $f_{i,j}$, and an absolute deadline $d_{i,j}$.

1.1.2 TYPICAL TASK MODELS

Depending on the timing requirements defined on a computation, tasks are classified into four basic categories: hard, firm, soft, and non real time.

A task τ_i is said to be *hard* if all its jobs have to complete within their deadline ($\forall j \;\; f_{i,j} \leq d_{i,j}$), otherwise a critical failure may occur in the system.

A task is said to be *firm* if only a limited number of jobs are allowed to miss their deadline. In [KS95], Koren and Shasha defined a firm task model in which only one job every S is allowed to miss its deadline. When a job misses its deadline, it is aborted and the next $S-1$ jobs must be guaranteed to complete within their deadlines. A slightly different firm model, proposed by Hamdaoui and Ramanathan in [HR95], allows specifying tasks in which at least k jobs every m must meet their deadlines.

A task is said to be *soft* if the value of the produced result gracefully degrades with its response time. For some applications, there is no deadline associated with soft computations. In this case, the objective of the system is to reduce their response times as much as possible. In other cases, a *soft deadline* can be associated with each job, meaning that the job should complete before its deadline to achieve its best performance. However, if a soft deadline is missed, the system keeps working at a degraded level of performance. To precisely evaluate the performance degradation caused by a soft deadline miss, a performance value function can be associated with each soft task, as described in Chapter 2.

Finally, a task is said to be *non real time* if the value of the produced result does not depend on the completion time of its computation.

1.1.3 ACTIVATION MODES

In a computer controlled system, a computational activity can either be activated by a timer at predefined time instants (time-triggered activation) or by the occurrence of a specific event (event-triggered activation).

When jobs activations are triggered by time and are separated by a fixed interval of time, the task is said to be *periodic*. More precisely, a periodic task τ_i is a time-triggered task in which the first job $\tau_{i,1}$ is activated at time Φ_i, called the task phase, and each subsequent job $\tau_{i,j+1}$ is activated at time $r_{i,j+1} = r_{i,j} + T_i$, where T_i is the task period. If D_i is the relative deadline associated with each job, the absolute deadline of job $\tau_{i,j}$ can be computed as:

$$\begin{cases} r_{i,j} = \Phi_i + (j-1)T_i \\ d_{i,j} = r_{i,j} + D_i \end{cases}$$

If job activation times are not regular, the task is said to be *aperiodic*. More precisely, an aperiodic task τ_i is a task in which the activation time of job $\tau_{i,k+1}$ is greater than or equal to that of its previous job $\tau_{i,k}$. That is, $r_{i,k+1} \geq r_{i,k}$.

If there is a minimum separation time between successive jobs of an aperiodic task, the task is said to be *sporadic*. More precisely, a sporadic task τ_i is a task in which the difference between the activation times of any two adjacent jobs $\tau_{i,k}$ and $\tau_{i,k+1}$ is greater than or equal to T_i. That is, $r_{i,k+1} \geq r_{i,k} + T_i$. The T_i parameter is called the *minimum interarrival time*.

1.1.4 PROCESSOR WORKLOAD AND BANDWIDTH

For a general purpose computing system, the processor workload depends on the amount of computation required in a unit of time. In a system characterized by aperiodic tasks, the average load $\overline{\rho}$ is computed as the product of the average computation time $\overline{C}$ requested by tasks and the average arrival rate λ:

$$\overline{\rho} = \overline{C}\lambda.$$

In a real-time system, however, the processor load also depends on tasks' timing constraints. The same set of tasks with given computation requirements and arrival patterns will cause a higher load if it has to be executed with more stringent timing constraints.

To measure the load of a real-time system in a given interval of time, Baruah, Howell and Rosier [BMR90] introduced the concept of *processor demand*, defined as follows:

Definition 1.1 *The processor demand $g(t_1, t_2)$ in an interval of time $[t_1, t_2]$ is the amount of computation that has been released at or after t_1 and must be completed within t_2.*

Hence, the processor demand $g_i(t_1, t_2)$ of task τ_i is equal to the computation time requested by those jobs whose arrival times and deadlines are within $[t_1, t_2]$. That is:

$$g_i(t_1, t_2) = \sum_{r_{i,j} \geq t_1, d_{i,j} \leq t_2} c_{i,j}$$

For example, given the set of jobs illustrated in Figure 1.2, the processor demand in the interval $[t_a, t_b]$ is given by the sum of computation times denoted with dark gray, that is, those jobs that arrived at or after t_a and have deadlines at or before t_b.

The total processor demand $g(t_1, t_2)$ of a task set in an interval of time $[t_1, t_2]$ is equal to the sum of the individual demands of each task. That is,

$$g(t_1, t_2) = \sum_{i=1}^{n} g_i(t_1, t_2)$$

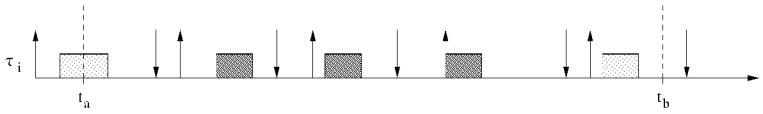

Figure 1.2 Processor demand for a set of jobs.

Then, the processor workload in an interval $[t_1, t_2]$ can be defined as the ratio of the processor demand in that interval and the length of the interval:

$$\rho(t_1, t_2) = \frac{g(t_1, t_2)}{t_2 - t_1}.$$

In the special case of a periodic hard task, the load produced by the task is also called the task *utilization* (U_i) and can be computed as the ratio between the task worst-case computation time C_i and its period T_i:

$$U_i = \frac{C_i}{T_i}.$$

Then, the total processor utilization is defined as the sum of the individual tasks' utilizations:

$$U_p = \sum_{i=1}^{n} U_i.$$

The utilization factor U_i of a periodic task τ_i basically represents the computational bandwidth requested by the task to the processor (assuming that each job will execute for its worst-case execution time). The concept of requested bandwidth can also be generalized to non periodic tasks as follows.

Definition 1.2 *A task τ_i is said to request a bandwidth U_i if, in any interval of time $[t_1, t_2]$, its computational demand $g_i(t_1, t_2)$ never exceeds $(t_2 - t_1)U_i$, and there exists an interval $[t_a, t_b]$ such that $g_i(t_a, t_b) = (t_b - t_a)U_i$.*

1.1.5 OVERLOAD AND OVERRUN

A system is said to be in overload condition when the computational demand of the task set exceeds the available processing time, that is, when there exists an interval of time $[t_a, t_b]$ such that $g(t_a, t_b) > (t_b - t_a)$. In such a situation, computational activities

start to accumulate in system's queues (which tend to become longer and longer, if the overload persists), and tasks response times tend to increase indefinitely. When tasks have timing constraints, an overload condition implies that one or more tasks will miss the deadline (assuming that all tasks execute for their expected computation time).

For a set of periodic tasks, the overload condition is reached when the processor utilization $U_p = \sum_{i=1}^{n} U_i$ exceeds one. Notice, however, that, depending on the adopted scheduling algorithm, tasks may also miss deadlines when the processor is not overloaded (as in the case of the Rate Monotonic algorithm, that has a schedulability bound less than one [LL73]).

While the overload is a condition related to the processor, the overrun is a condition related to a single task.

A task is said to overrun when there exists an interval of time in which its computational demand g_i exceeds its expected bandwidth U_i. This condition may occur either because jobs arrive more frequently than expected (*activation overrun*), or because computation times exceed their expected value (*execution overrun*). Notice that a task overrun does not necessarily cause an overload.

1.2 FROM HARD TO SOFT REAL-TIME SYSTEMS

Real-time systems technology, traditionally used for developing large systems with safety-critical requirements, has recently been extended to support novel application domains, often characterized by less stringent timing requirements, scarce resources, and more dynamic behavior. To provide appropriate support to such emerging applications, new methodologies have been investigated for achieving more flexibility in handling task sets with dynamic behavior, as well as higher efficiency in resource exploitation.

In this section we describe the typical characteristics of hard and soft real-time applications, and present some concrete example to illustrate their difference in terms of application requirements and execution behavior.

1.2.1 CLASSICAL HARD REAL-TIME APPLICATIONS

Real-time computing technology has been primarily developed to support safety-critical systems, such as military control systems, avionic devices, and nuclear power plants. However, it has been also applied to industrial systems that have to guarantee a

certain performance requirements with a limited degree of tolerance. In these systems, also called *hard* real-time systems, most computational activities are characterized by stringent timing requirements, that have to be met in all operating conditions in order to guarantee the correct system behavior. In such a context, missing a single deadline is not tolerated, either because it could have catastrophic effects on the controlled environment, or because it could jeopardize the guarantee of some stringent performance requirements.

For example, a defense missile could miss its target if launched a few milliseconds before or after the correct time. Similarly, a control system could become unstable if the control commands are not delivered at a given rate. For this reason, in such systems, computational activities are modeled as tasks with hard deadlines, that must be met in all predicted circumstances. A task finishing after its deadline is considered not only late, but also wrong, since it could jeopardize the whole system behavior.

In order to guarantee a given performance, hard real-time systems are designed under worst-case scenarios, derived by making pessimistic assumptions on system behavior and on the environment. Moreover, to avoid unpredictable delays due to resource contention, all resources are statically allocated to tasks based on their maximum requirements. Such a design approach allows system designers to perform an off-line analysis to guarantee that the system is able to achieve a minimum desired performance in all operating conditions that have been predicted in advance.

A crucial phase in performing the off-line guarantee is the evaluation of the worst-case computation times (WCETs) of all computational activities. This can be done either experimentally, by measuring the maximum execution time of each task over a large amount of input data, or analytically, by analyzing the source code, identifying the longest path, and computing the time needed to execute it on the specific processor platform. Both methods are not precise. In fact, the first experimental approach fails in that only a limited number of input data can be generated during testing, hence the worst-case execution may not be found. On the other hand, the analytical approach has to make so many assumptions on the low-level mechanisms present in the computer architecture, that the estimation becomes too pessimistic. In fact, in modern computer architectures, the execution time of an instruction depends on several factors, such as the prefetch queue, the DMA, the cache size, and so on. The effects of such mechanisms on task execution are difficult to predict, because they also depends on the previous computation and on the actual data. As a consequence, deriving a precise estimation of the WCET is very difficult (if not impossible). The WCET estimations used in practice are not precise and are affected by large errors (typically more than 20%). This means that to have an absolute off-line guarantee, all tasks execution times have to be overestimated.

Once all computation times are evaluated, the feasibility of the system can be analyzed using several guarantee algorithms proposed in the literature for different scheduling algorithms and task models (see [But97] for a survey of guarantee tests). To simplify the guarantee test and cope with peak load conditions, the schedulability analysis of a task set is also performed under pessimistic assumptions. For example, a set of periodic tasks is typically analyzed under the following assumptions:

- *All tasks start at the same time.* This assumption simplifies the analysis because it has been shown (both under fixed priorities [LL73] and dynamic priorities [BMR90]) that synchronous activations generate the highest workload. Hence, if the system is schedulable when tasks are synchronous, it is also schedulable when they have different activation phases.
- *Job interarrival time is constant for each task.* This is assumption can be enforced by a time-triggered activation mechanism, so making tasks purely periodic. However, there are cases in which the activation of a task depends on the occurrence of an external event (such as the arrival of a message from the network) whose periodicity cannot be precisely predicted.
- *All jobs of a task have the same computation time.* This assumption can be reasonable for tasks having a very simple structure (no branches or loops). In general, however, tasks have loops and branches inside their code, which depend on specific data that cannot be predicted in advance. Hence, the computation time of a job, is highly variable. As a consequence, modeling a task with a fixed computation time equal to the maximum execution time of all its jobs leads to a very pessimistic estimate, which causes a waste of the processing resources.

The consequence of such a worst-case design methodology is that high predictability is achieved at the price of a very low efficiency in resource utilization. Low efficiency also means high cost, since, in order to prevent deadline misses during sporadic peak load conditions, more resources (both in terms of memory and computational power) need to be statically allocated to tasks for coping with the maximum requirements, even though the average requirement of the system is much lower.

In conclusion, if the real-time application consists of tasks with a simple programming structure that can be modeled by a few fixed parameters, then classical schedulability analysis can be effectively used to provide an off-line guarantee under all anticipated scenarios. However, when tasks have a more complex and dynamic behavior, the classical hard real-time design paradigm becomes highly inefficient and less suited for developing embedded systems with scarce resources.

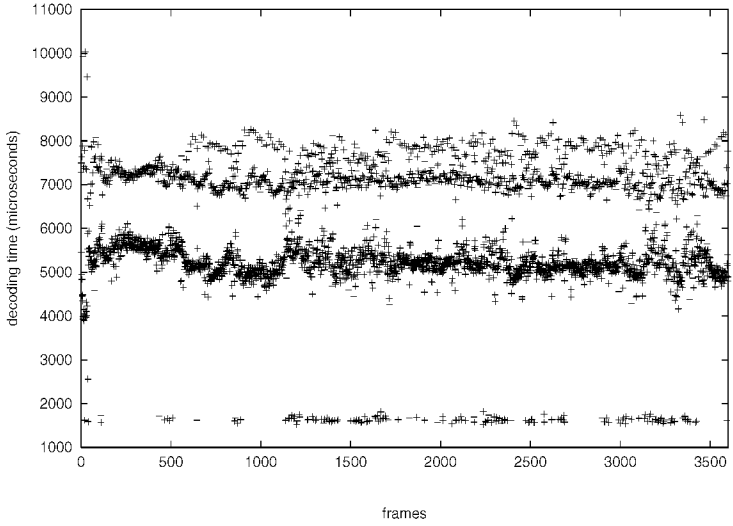

Figure 1.3 Decoding times for a sequence of frames taken from *Star Wars*.

1.2.2 NOVEL APPLICATION DOMAINS

In the last years, emerging time-sensitive applications brought real-time computing into several new different domains, including multimedia systems, monitoring apparatuses, telecommunication networks, mobile robotics, virtual reality, and interactive computer games. In such systems, also called *soft* real-time systems, application tasks are allowed to have less stringent timing constraints, because missing a deadline does not cause catastrophic consequences on the environment, but only a performance degradation, often evaluated through some quality of service (QoS) parameter.

In addition, often, such systems operate in more dynamic environments, where tasks can be created or killed at runtime, or task parameters can change from one job to the other.

There are many soft real-time applications in which the worst-case duration of some tasks is rare but much longer than the average case. In multimedia systems, for instance, the time for decoding a video frame in MPEG players can vary significantly as a function of the data contained in the frames. Figure 1.3 shows the decoding times of frames in a specific sequence of the *Star Wars* movie.

As another example of task with variable computation time, consider a visual tracking system where, in order to increase responsiveness, the moving target is searched in a

small window centered in a predicted position, rather than in the entire visual field. If the target is not found in the predicted area, the search has to be performed in a larger region until, eventually, the entire visual field is scanned in the worst-case. If the system is well designed, the target is found very quickly in the predicted area most of the times. Thus, the worst-case situation is very rare, but very expensive in terms of computational resources (computation time increases quadratically as a function of the number of trials). In this case, an off-line guarantee based on WCETs would drastically reduce the frequency of the tracking task, causing a severe performance degradation with respect to a soft guarantee based on the average execution time.

Just to give a concrete example, consider a videocamera producing images with 512x512 pixels, where the target is a round spot, with a 30 pixels diameter, moving inside the visual field. In this scenario, if U_i is the processor utilization required to track the target in a small window of 64x64 pixels at a rate T_i, a worst-case guarantee would require the tracking task to run 64 times slower in order to demand the same bandwidth in the entire visual field (which is 64 times bigger). Clearly, in this application, it is more convenient to perform a less pessimistic guarantee in order to increase the tracking rate and accept some sporadic overrun as a natural system behavior.

In other situations, periodic tasks could be executed at different rates in different operating conditions. For example, in a flight control system, the sampling rate of the altimeters is a function of the current altitude of the aircraft: the lower the altitude, the higher the sampling frequency. A similar need arises in robotic applications in which robots have to work in unknown environments, where trajectories are planned based on the current sensory information. If a robot is equipped with proximity sensors, in order to maintain a desired performance, the acquisition rate of the sensors must increase whenever the robot is approaching an obstacle. Another example of computation with variable activation rate is engine control, where computation is triggered by the shaft rotation angle, hence task activation is a function of the motor speed.

In all these examples, task parameters are not fixed, as typically considered in a hard task, but vary from a job to the other, depending on the data to be processed.

The problem becomes even more significant when the real-time software runs on top of modern hardware platforms, which include low-level mechanisms such as pipelining, prefetching, caching, or DMA. In fact, although these mechanisms improve the average behavior of tasks, they worsen the worst case, so making much more difficult to provide precise estimates the of worst-case computation times.

To provide a more precise information about the behavior of such dynamic computational activities, one could describe a parameter through a probability distribution derived by experimental data. Figure 1.4 illustrates the probability distribution function of job computation times for the process illustrated in Figure 1.3.

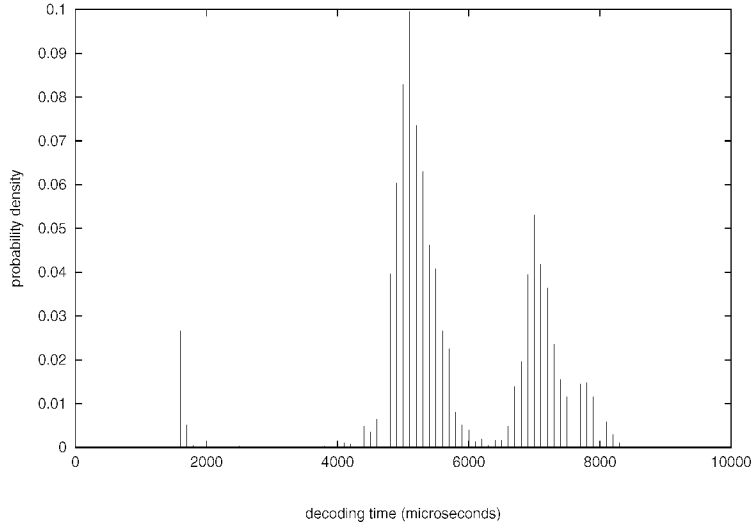

Figure 1.4 Distribution of job computation times for the frame sequence shown in Figure 1.3.

1.3 PROVIDING SUPPORT FOR SOFT REAL-TIME SYSTEMS

Considering the characteristics of the applications describes above, what is the most appropriate way to provide a predictable, as well as efficient, support for them? Can classical (non real-time) operating systems, such as Windows or Linux, be used to develop soft real-time applications? What are their limitations? Can hard real-time systems provide a sufficient support for soft real-time computing? If not, what are the desired features that a real-time kernel should have for this purpose?

In this section we explain why classical general purpose operating systems are not suited for supporting real-time applications. We also explain the limitations of the hard real-time systems and finally we conclude the section with a list of desired features that should be included in a kernel for providing efficient support for soft real-time applications.

1.3.1 PROBLEMS WITH NON REAL-TIME SYSTEMS

The fact that a soft real-time application may tolerate a certain degree of performance degradation does not mean that timing constraints can be completely ignored. For example, in a multimedia application, a quality of service level needs to be enforced on the computational tasks to satisfy a desired performance requirement. If too many deadlines are missed, there is no way to keep the system performance above a certain threshold.

Unfortunately, classical general purpose operating systems, such as Windows or Linux, are not suited for running real-time applications, because they include several internal mechanisms that introduce unbounded delays and cause a high level of unpredictability.

First of all, they do not provide support for controlling explicit timing constraints. System timers are available at a relatively low resolution, and the only kernel service for handling time is given by the *delay()* primitive, which suspends the calling tasks for a given interval of time. The problem with the delay primitive, however, is that, if a task requires to be suspended for an interval of time equal to Δ, the system only guarantees that the calling task will be delayed *at least* by Δ. When using shared resources, the delay primitive can be very unpredictable, as shown in the example illustrated in Figure 1.5. Here, although in normal conditions (a) task τ_1 has a slack equal to 4 units of time, a delay of 2 time units causes the task to miss its deadline (b).

Much longer and unpredictable delays can be introduced during task synchronization, if classical semaphores are used to enforce mutual exclusion in accessing shared resources. For example, consider two tasks, τ_a and τ_b, having priorities $P_a > P_b$, that share a common resource R protected by a semaphore. If τ_b is activated first and is preempted by τ_a inside its critical section, then τ_a is blocked on the semaphore to preserve data consistency in the shared resource. In the absence of other activities, we can clearly see that the maximum blocking time experienced by τ_a on the semaphore is equal to the worst-case duration of the critical section executed by τ_b. However, if other tasks are running in the system, a third task, τ_c, having intermediate priority ($P_b < P_c < P_a$), may preempt τ_b while τ_a is waiting for the resource, so prolonging the blocking time of τ_a for its entire execution. This phenomenon, known as a *priority inversion* [SRL90], is illustrated in Figure 1.6.

In general, if we cannot limit the number of intermediate priority tasks that can run while τ_a is waiting for the resource, the blocking time of τ_a cannot be bounded, preventing any performance guarantee on its execution. The priority inversion phenomenon illustrated above can be solved using specific concurrency control protocols when accessing shared resources, like the *Priority Inheritance Protocol*, the *Priority Ceiling Protocol* [SRL90], or the *Stack Resource Policy* [Bak91]. However, unfortunately, these protocols are not yet available in all general purpose operating systems.

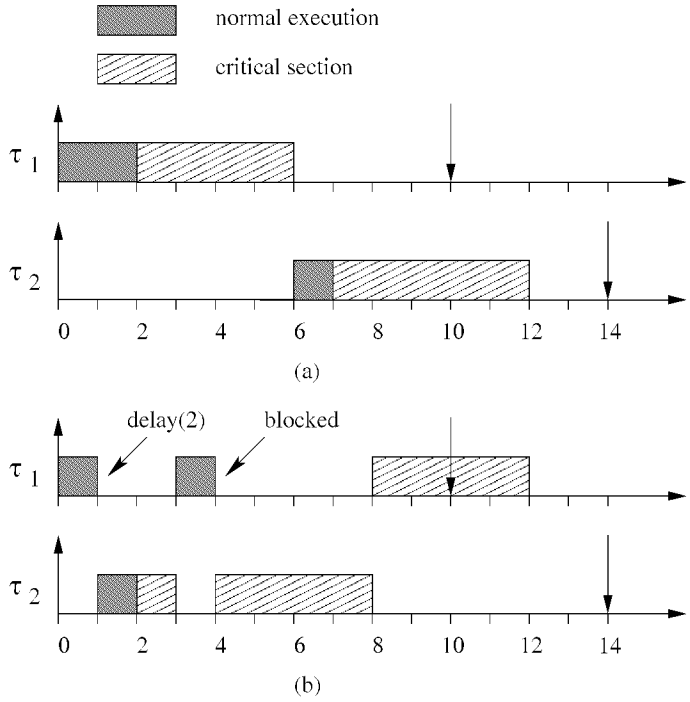

Figure 1.5 Scheduling anomaly caused by the *delay()* primitive: although τ_1 has a slack equal to 4 units of time (a), a *delay(2)* causes the task to miss its deadline (b).

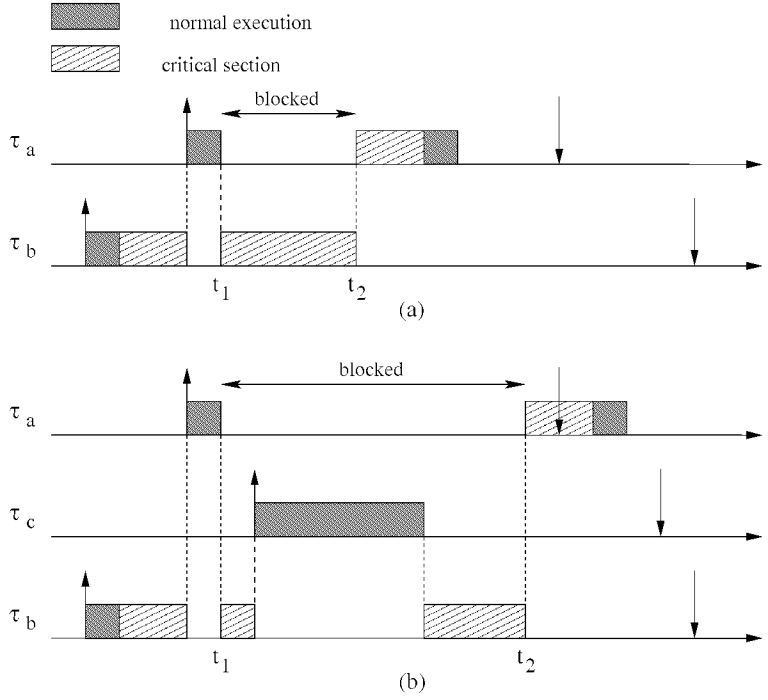

Figure 1.6 The priority inversion phenomenon. In case (a) τ_a is blocked for at most the duration of the critical section of τ_b. In case (b) τ_a is also delayed by the entire execution of τ_c, having intermediate priority.

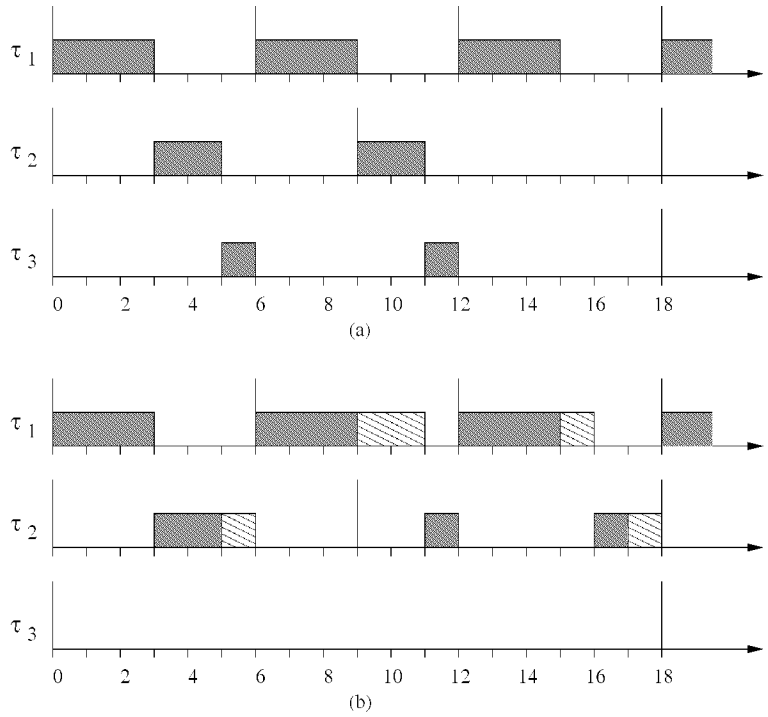

Figure 1.7 Effects of an execution overrun. In normal conditions (a) all tasks execute within their deadlines; but a sequence of overruns in τ_1 and τ_2 may prevent τ_3 to execute (b).

Another negative phenomenon that frequently occurs in general purpose operating systems is a high interference among task executions. Since there is not explicit control on the amount of time each task actually executes on the processor, a high priority task can basically steal as much time as it can to a lower priority computation. When tasks have a highly dynamic behavior, the consequences of the interference on task executions are very difficult to predict, causing a significant performance degradation in the system. In other words, in the absence of specific protection mechanisms in the kernel, an execution overrun occurring in a high priority task may cause very negative effects on several other tasks having lower priority. The problem is illustrated in Figure 1.7, where three periodic tasks, with periods $T_1 = 6, T_2 = 9, T_3 = 18$, and execution times $C_1 = 3, C_2 = 2, C_3 = 2$, are scheduled with the Rate Monotonic algorithm. As we can see, in normal conditions (Figure 1.7a) the task set is schedulable, however some execution overruns in τ_1 and τ_2 may prevent τ_3 to execute (Figure 1.7b).

The problem illustrated above becomes more serious in the presence of permanent overload conditions, occurring for example when new tasks are activated and the actual processor utilization is greater than one.

When using non real-time systems, several kernel mechanisms can cause negative effects on real-time computations. For example, typical message passing primitives (like *send* and *receive*) available in kernels for intertask communication adopt a blocking semantics when receiving a message from an empty queue or sending a message into a full queue. If the blocking time cannot be bounded, the delay introduced in task executions can be very high, preventing any performance guarantee. Moreover, a blocking semantics also prevents communication among periodic tasks having different frequencies.

Finally, also the interrupt mechanism, as implemented in general purpose operating systems, contributes to decrease the predictability of the system. In fact, in such systems, device drivers always execute with a priority higher than those assigned to application tasks, preempting the running task at any time. Hence, a bursty sequence of interrupts may introduce arbitrary long delays in the running tasks, causing a severe performance degradation.

1.3.2 PROBLEMS WITH THE HARD REAL-TIME APPROACH

In principle, if a set of tasks with hard timing constraints can be feasibly scheduled by a hard real-time kernel, it can also be feasibly scheduled if the same constraints are considered to be soft. However, there are a number of problems to be taken into account when using a hard real-time design approach for supporting a soft real-time application.

First of all, as we already mentioned above, the use of worst-case assumptions would cause a waste of resources, which would be underutilized for most of the time, just to cope with some sporadic peak load condition. For applications with heavy computational load (e.g., graphical activities), such a waste would imply a severe performance degradation or a significant increase of the system cost. Figure 1.8a shows that, whenever the load has large variations, keeping the load peaks always below one causes the average load to be very small (low efficiency). On the other hand, Figure 1.8b shows that efficiency can only be increased at the cost of accepting some transient overload, by allowing some peak load to exceed one, thus missing some deadlines.

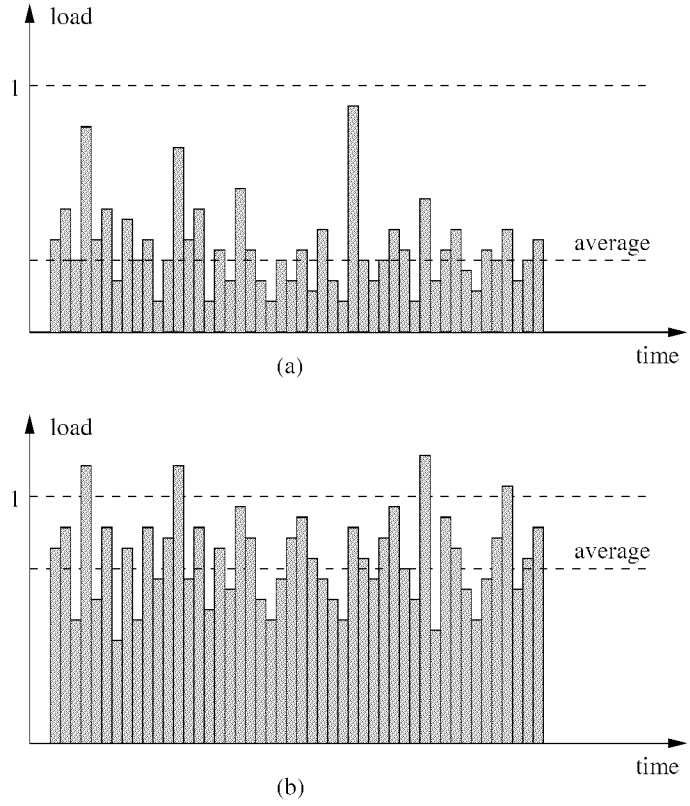

Figure 1.8 Two different load conditions: an underloaded system with low average resource usage (a), and a system with transient overloads but high average resource usage (b).

Another problem with the hard real-time approach is that, in many practical cases, a precise estimation of WCETs is very difficult to achieve. In fact, several low level mechanisms present in modern computer architectures (such as interrupts, DMA, pipelining, caching, and prefetching) introduce a non deterministic behavior in tasks' execution, whose duration cannot be predicted in advance.

Even though a precise WCET estimation could be derived for each task, a worst-case feasibility analysis would be very inefficient when task execution times have a high variance. In this case, a classical off-line hard guarantee would waste the system's computational resources for preserving the task set feasibility under sporadic peak load situations, even though the average workload is much lower. Such a waste of

Task	C_i^{avg}	C_i^{max}	T_i
τ_1	2	2	6
τ_2	3	3	10
τ_3	3	10	12

Table 1.1 Task set parameters.

resources (which increases the overall system's cost) can be justified for very critical applications (e.g., military defense systems or safety critical space missions), in which a single deadline miss may cause catastrophic consequences. However, it does not represent a good solution for those applications (the majority) in which several deadline misses can be tolerated by the system, as long as the average task rates are guaranteed off line.

On the other hand, uncontrolled overruns are very dangerous if not properly handled, since they may heavily interfere with the execution of other tasks, which could be more critical. Consider for example the task set given in Table 1.1, where two tasks, τ_1 and τ_2, have a constant execution time, whereas τ_3 has an average computation time ($C_3^{avg} = 3$) much lower than its worst-case value ($C_3^{max} = 10$). Here, if the schedulability analysis is performed using the average computation time C_3^{avg}, the total processor utilization becomes 0.92, meaning that the system is not overloaded; however, under the Earliest Deadline First (EDF) algorithm [LL73] the tasks can experience long delays during overruns, as illustrated in Figure 1.9. Similar examples can easily be found also under fixed priority assignments (e.g., under the Rate Monotonic algorithm [LL73]), when overruns occur in the high priority tasks (see for example the case illustrated in Figure 1.7).

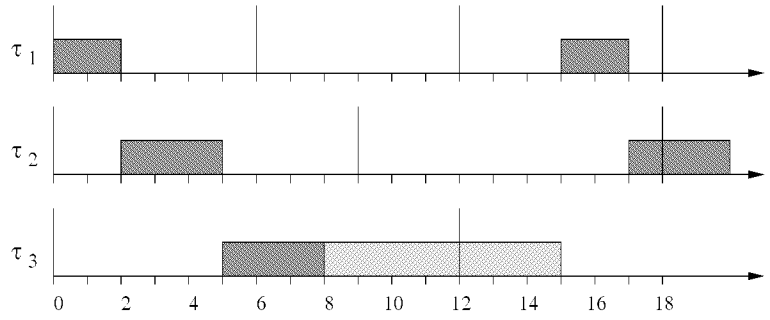

Figure 1.9 Negative effects of uncontrolled overruns under EDF.

To prevent an overrun to introduce unbounded delays on tasks' execution, the system could either decide to abort the current instance of the task experiencing the overrun or let it continue with a lower priority. The first solution is not safe, because the instance could be in a critical section when aborted, thus leaving a shared resource with inconsistent data (very dangerous). The second solution is much more flexible, since the degree of interference caused by the overrun on the other tasks can be tuned acting on the priority assigned to the "faulty" task for executing the remaining computation.

A general technique for limiting the effects of overruns is based on a resource reservation approach [MST94b, TDS+95, Abe98], according to which each task is assigned (off line) a fraction of the available resources and is handled by a dedicated server, which prevents the served task from demanding more than the reserved amount. Although such a method is essential for achieving predictability in the presence of tasks with variable execution times, the overall system's performance becomes quite dependent from a correct resource allocation. For example, if the CPU bandwidth allocated to a task is much less than its average requested value, the task may slow down too much, degrading the system's performance. On the other hand, if the allocated bandwidth is much greater than the actual needs, the system will run with low efficiency, wasting the available resources.

1.3.3 DESIRED FEATURES

From the problems illustrated in the previous sections, we can derive a set of ideal features that a real-time kernel should have to efficiently support soft real-time applications while guaranteeing a certain degree of performance. They are listed below.

- **Overload management**. Whenever the total computational load exceeds the processor capacity, the systems should properly decrease the demand of the task set to avoid uncontrolled performance degradation.
- **Temporal isolation**. The temporal behavior of a computational activity should not depend on the execution characteristics of other activities, but only on the fraction of processor (CPU bandwidth) that has been allocated to it. Whenever a job executes more than expected, or a sequence of jobs arrives more frequently than predicted, only the corresponding task should be delayed, avoiding reciprocal task interference.
- **Bounded priority inversion**. When tasks interact through shared resources, the maximum blocking time caused by mutual exclusion should be bounded by the duration of one or a few critical sections, preventing other tasks to increase blocking delays with their execution.

- **Aperiodic task handling**. Asynchronous arrival of aperiodic events should be handled in such a way that the performance of periodic tasks is guaranteed off-line, and aperiodic responsiveness is maximized.
- **Resource reclaiming**. Any spare time saved by early completions should be exploited for increasing aperiodic responsiveness or coping with transient overload conditions.
- **Adaptation**. For real-time systems working in very dynamic environments, any change in the application behavior should be detected and cause a system adaptation.

Most of the features outlined above are described in detail in the remaining chapters of this book. Aperiodic task scheduling is not treated in detail since it has been already discussed in [But97] in the context of hard real-time systems.